SAMPLE PROGRAMME

```
LIST

100 COUNT=TAPCNT
110 OPEN "O",#1,"CAS0:TEST.DAT"
120 FOR A=1 TO 10
130 PRINT#1,A;SQR(A)
140 NEXT
150 CLOSE #1
160 WIND COUNT
170 OPEN "I",#1,"CAS0:TEST.DAT"
175 PRINT :PRINT" A ","SQR(A)"
180 IF EOF(1) GOTO 220
190 INPUT#1,A,B
200 PRINT A,B
210 GOTO 180
220 CLOSE #1
230 END
RUN

 A             SQR(A)
 1              1
 2              1.41421
 3              1.73205
 4              2
 5              2.23607
 6              2.44949
 7              2.64575
 8              2.82843
 9              3
 10             3.16228
>
```


CHAPTER 4
Functions

ABS

FORMAT ABS(<numeric expression>)
PURPOSE To return the absolute value of a numeric expression.
EXAMPLE A=ABS(−1.6)

REMARKS ABS returns the absolute value of <numeric expression>.

ASC

FORMAT ASC(<string>)
PURPOSE To return the character code of a character.
EXAMPLE A=ASC("A")

REMARKS ASC returns a numerical value that is the ASCII code of the first character of <string>. See APPENDIX F, "Character Code Table" for the relationship between characters and ASCII character codes.
If <string> is null, an FC ("Illegal function call") error is returned.

(See CHR$, and APPENDIX F, "Character Code Table".)

ATN

FORMAT ATN(<numeric expression>)
PURPOSE To return the arc tangent of a numeric expression.
EXAMPLE A=ATN(0.5)

REMARKS ATN returns the arc tangent of <numeric expression> in radians. The result of the operation is in the range $-\pi/2$ to $\pi/2$.

CDBL

FORMAT CDBL(<numeric expression>)
PURPOSE To convert integers and single precision numbers into double precision numbers.
EXAMPLE A#=CDBL(B!/2)

REMARKS CDBL converts the value of <numeric expression> to a double precision number. Only the type conversion is performed and there is no change in the number of significant digits.

CHR$

FORMAT CHR$(<numeric expression>)
PURPOSE To return the character corresponding to a specified character code.
EXAMPLE A$=CHR$(&H41)

REMARKS CHR$ returns the ASCII character whose code is the value of <numeric expression>. If the value of <numeric expression> is not in the range of 0 to 255, an FC ("Illegal function call") error occurs.
You may include real numbers in <numeric expression> but the fractional portion of the real number must be rounded before using it as the value of <numeric expression>.

(See ASC, and APPENDIX F, "Character Code Table.")

CINT

FORMAT CINT(<numeric expression>)
PURPOSE To convert single and double precision numbers into integers.
EXAMPLE A%=CINT(B#/2)

REMARKS CINT converts the value of <numeric expression> into an integer by rounding the fractional portion. If the value of <numeric expression> is not in the range of −32768 to 32767, an OV ("Overflow") error occurs.

(See CDBL, FIX, and INT.)

COS

FORMAT COS (<numeric expression>)
PURPOSE To return the cosine of a numeric expression.
EXAMPLE A=COS(3.1415926/2)

REMARKS COS returns the cosine of <numeric expression> in radians.

CSNG

FORMAT CSNG(<numeric expression>)
PURPOSE To convert integers and double precision numbers into single precision numbers.
EXAMPLE A!=CSNG(B#)

REMARKS CSNG converts the value of <numeric expression> into a single precision number in 6 significant digits.
If the value of <numeric expression> is not in the range −1.70141 E+38 to 1.70141 E+38, an OV ("Overflow") error occurs.

(See CDBL and CINT.)

CSRLIN

FORMAT CSRLIN
PURPOSE To return the vertical position of the cursor on the virtual screen.
EXAMPLE Y=CSRLIN

REMARKS CSRLIN returns the value of the vertical position of the cursor on the virtual screen. The value of the vertical position must be in the range of 0 to (number of lines on the virtual screen −1).

(See POS.)

DATE$

FORMAT DATE$ [=MM/DD/YY]
PURPOSE To set the current date in, and return the date kept by, the internal calendar clock.
EXAMPLE PRINT DATE$

REMARKS DATE$ displays the date kept by the internal calendar clock in the HX-20. Using this statement, you can set the date in the form of a string such as MM/DD/YY (e.g., "08/15/82") where MM represents the month, DD represents the day and YY represents the year.
The date is displayed in the same format as that when it was input. Once you set the correct current date with a DATE$, you are not required to set it again, as the clock in the HX-20 keeps track of the time and date.

(See DAY and TIME$.)

DAY

FORMAT DAY
PURPOSE To set the current day of the week in, and display the day of the week kept by, the internal calendar clock.
EXAMPLE PRINT DAY

REMARKS In the HX-20, the day of the week is kept by the internal calendar clock by integers 1 to 7 corresponding to the 7 days of the week. Using this statement, you can set the current day of the week such as DAY=7(Saturday). The day of the week is displayed by one of integers 1 to 7. Once you set the correct current day of the week with a DAY, you are not required to set it again.

(See DATE$ and TIME$.)

EOF

FORMAT EOF(<file number>)
PURPOSE To return the end-of-file code.
EXAMPLE IF EOF(3) THEN CLOSE #1 ELSE GOTO 100

REMARKS The file specified by <file number> must have been opened for the input mode. EOF checks if the file specified by <file number> has reached its end. EOF returns −1 (true) if the end of the file has been reached and returns 0 (false) if not.
If the specified file is RS-232C port ("COM0:"), EOF returns −1 when the buffer is empty and returns 0 when the buffer is not empty. The EOF function always returns 0 (false) for the file assigned to the keyboard.

ERL/ERR

FORMAT ERL
ERR
PURPOSE To return the error code of an error occurred and the line number where the error occurred.
EXAMPLE A=ERL
B=ERR

REMARKS When an error occurs, the error code is stored in the variable ERR and the line number where the error occurred is stored in the variable ERL. If the statement that caused the error was executed in the direct mode, line number 65535 is stored in the variable ERL.
Normally, the ERL and ERR variables are used in the error trapping routine specified by an ON ERROR GOTO statement to control the processing flow.

(See ON ERROR GOTO.)

EXP

FORMAT EXP (<numeric expression>)
PURPOSE To return the value of an exponential function with e as its base.
EXAMPLE A=EXP(1)

REMARKS The value of <numeric expression> must be the result of an exponential function. If the value of <numeric expression> is greater than 88.02969, an OV ("Overflow") error occurs.

FIX

FORMAT FIX (<numeric expression>)
PURPOSE To return the truncated integer part of a numeric expression.
EXAMPLE A=FIX(−B/3)

REMARKS FIX returns the value of <numeric expression> as a truncated integer part.

(See CINT and INT.)

FRE

FORMAT FRE(<expression>)
PURPOSE To return the size of an unused memory area.
EXAMPLE PRINT FRE(0)
PRINT FRE("A$")

REMARKS If the <expression> is a numeric expression, FRE returns the number of free bytes in the BASIC text area. If <expression> is a string expression, FRE returns the number of bytes in the BASIC string area. <expression> is merely a dummy. Any arguments to FRE may be assigned as long as they are numeric or string expressions. Since the size of the unused memory area displayed includes a work area for BASIC programme execution, please use the displayed memory capacity as a guide only.

HEX$

FORMAT HEX$(<numeric expression>)
PURPOSE To return a string which represents the hexadecimal value of the decimal argument.
EXAMPLE A$=HEX$(65535)

REMARKS HEX$ converts the decimal value of <numeric expression> to a hexadecimal value and returns it as a string. The value of <numeric expression> must be in the range −32768 to 65535. If the value of <numeric expression> includes a decimal fraction, it is rounded to an integer before HEX$ (<numeric expression>) is evaluated.

(See OCT$ and VAL.)

SAMPLE PROGRAMME

```
LIST

100 PRINT " DEC   OCT HEX"
110 FOR I=5 TO 16
120 PRINT USING"####";I;
130 PRINT USING"&  &";" ",OCT$(I),HEX$(I
)
140 NEXT I
RUN
 DEC   OCT HEX
   5     5   5
   6     6   6
   7     7   7
   8    10   8
   9    11   9
  10    12   A
  11    13   B
  12    14   C
  13    15   D
  14    16   E
  15    17   F
  16    20   10
>
```

INKEY$

FORMAT INKEY$

PURPOSE To return a one-character string of the pressed character key or a null string if no character key is pressed.

EXAMPLE A$=INKEY$

REMARKS INKEY$ returns a null string if the keyboard buffer is empty. If the keyboard buffer contains any character key input, INKEY$ reads the character from the buffer and returns it as a one-character string. Any keys not included in the "Character Code Table" such as **SHIFT** key, etc., are ignored.

(See APPENDIX F, "Character Code Table".)

INPUT$

FORMAT INPUT$(<number of characters>[,[#]<file number>])

PURPOSE To return a string of characters read from a specified file.

EXAMPLE A$=INPUT$(5,#3)

REMARKS INPUT$ reads a string of characters in the number specified by <number of characters> from the file specified by <file number>. If <file number> is omitted, characters can be input from the keyboard; but the characters input from the keyboard are not echoed (i.e., not displayed on the screen), unlike the execution of an INPUT statement.

INPUT$ is in a wait state until a string of characters specified by <number of characters> is all input. However, if any input data exists in the input buffer, INPUT$ reads characters from the buffer.

With an INPUT$, all characters except **BREAK** key are read as is. Therefore, INPUT$ allows the input of characters, such as Carriage Return (character code 13), etc., which cannot be entered by INPUT and LINE INPUT statements.

INSTR

FORMAT INSTR([<numeric expression>,]<string 1>, <string 2>)

PURPOSE To search for the first occurrence of one string in another string and return the position of the searched string.

EXAMPLE B=INSTR(A$,"XYZ")

REMARKS INSTR searches for the first occurrence of <string 2> in <string 1> and returns the position at which the match is found. If <string 2> cannot be found, INSTR returns 0.
<numeric expression> is the position for starting the search. If <numeric expression> is omitted, the search is started from the beginning of <string 1>. If a null string is specified as <string 2>, INSTR returns the same value as that specified by <numeric expression>.

SAMPLE PROGRAMME

```
LIST

100 A$="ZXCVBNM,./AS FG JKL :"
110 B$=INPUT$(1)
120 C=INSTR(A$,B$)
130 IF C=0 GOTO 110
140 IF C>10 THENC=C+17
150 SOUND C+5,2
160 GOTO110
>
```


INT

FORMAT INT(<numeric expression>)

PURPOSE To return the largest integer value (truncated).

EXAMPLE PRINT INT (−B/3)

REMARKS INT returns the largest integer value which is equal to or less than the value of <numeric expression>.

(See FIX and CINT.)

SAMPLE PROGRAMME

```
LIST

100 PRINT "    I     CINT   INT    FIX"
110 FOR I=2.4 TO -2.4 STEP -0.3
120 A%=CINT(I)
130 B%=INT(I)
140 C%=FIX(I)
150 PRINT USING "  ##.#";I;
160 PRINT USING "  #### ";A%;B%;C%
170 NEXT I
180 END
RUN
    I      CINT   INT    FIX
  2.4       2      2      2
  2.1       2      2      2
  1.8       2      1      1
  1.5       2      1      1
  1.2       1      1      1
  0.9       1      0      0
  0.6       1      0      0
  0.3       0      0      0
  0.0       0      0      0
 -0.3       0     -1      0
 -0.6      -1     -1      0
 -0.9      -1     -1      0
 -1.2      -1     -2     -1
 -1.5      -1     -2     -1
 -1.8      -2     -2     -1
 -2.1      -2     -3     -2
 -2.4      -2     -3     -2
>
```

LEFT$

FORMAT LEFT$(<string>,<numeric expression>)
PURPOSE To return an arbitrary length of string from the leftmost characters of a string.
EXAMPLE B$=LEFT$(A$,4)

REMARKS The value of <numeric expression> must be in the range of 0 to 255. If <numeric expression> is greater than the total number of characters in <string>, the entire string will be returned. If <numeric expression> is 0, the null string (length zero) is returned.

(See MID$ and RIGHT$.)

LEN

FORMAT LEN(<string>)
PURPOSE To return the total number of characters in a string.
EXAMPLE A=LEN(A$)

REMARKS LEN returns the total number of characters in <string>. If <string> includes non-printing characters such as control codes and blanks, they are not actually output but are counted as characters.

(See APPENDIX F, Character Code Table.)

LOF

FORMAT LOF (<file number>)
PURPOSE To return the size of a specified file.
EXAMPLE A=LOF(3)

REMARKS The file specified by <file number> must have been opened in the input mode. If the specified file number is in a ROM cartridge, LOF returns the remaining length of the file in units of bytes. If the specified file number is in the RS-232C port, LOF returns the number of data stored in the buffer in units of bytes.

LOG

FORMAT LOG(<numeric expression>)
PURPOSE To return the natural logarithm of a numeric expression.
EXAMPLE PRINT LOG(2.7812818)

REMARKS LOG returns the natural logarithm of the value given by <numeric expression>.

MID$

FORMAT MID$(<string>,<expression 1>[,<expression 2>])
PURPOSE To return an arbitrary length of string from a string.
EXAMPLE B$=MID$(A$,2,3)

REMARKS MID$ returns a string of characters in the length specified by <expression 2> from the <string> beginning with the <expression 1>th character. The values of <expression 1> and <expression 2> must be in the ranges 1 to 255 and 0 to 255, respectively.
If <expression 2> is omitted or if there are fewer than <expression 2> characters to the right of the <expression 1>th character, all rightmost characters beginning with the <expression 1>th character are returned. When the number of characters in <string> is fewer than <expression 1>, MID$ returns a null string.

(See LEFT$ and RIGHT$.)

SAMPLE PROGRAMME

```
LIST

100 A$="ABCDEFGHIJKLMN"
110 PRINT "       MID$        LEFT$
RIGHT$ "
120 FOR I=1 TO 7
130 M$=MID$(A$,I,7)
140 L$=LEFT$(A$,I)
150 R$=RIGHT$(A$,I)
160 PRINT USING "    &       &";M$,L$,R$
170 NEXT I
RUN
      MID$        LEFT$        RIGHT$
    ABCDEFG      A            N
    BCDEFGH      AB           MN
    CDEFGHI      ABC          LMN
    DEFGHIJ      ABCD         KLMN
    EFGHIJK      ABCDE        JKLMN
    FGHIJKL      ABCDEF       IJKLMN
    GHIJKLM      ABCDEFG      HIJKLMN
>
```


OCT$

FORMAT OCT$(<numeric expression>)
PURPOSE To return a string which represents the octal value of the decimal argument.
EXAMPLE PRINT OCT$(123+456)

REMARKS OCT$ converts the decimal value of <numeric expression> to an octal value and returns it as a string. The value of <numeric expression> must be in the range of −32768 to 65535. If the value of <numeric expression> includes a decimal fraction, it is rounded to an integer before OCT$ (<numeric expression>) is evaluated.

(See HEX$.)

*PEEK

FORMAT PEEK(<address>)
PURPOSE To return the byte read from a specified memory location.
EXAMPLE A=PEEK(&H0C00)

REMARKS PEEK returns the byte read from the memory location specified by <address> <address> must be in the range of 0 to 65535 (&H0 to &HFFFF). If the value of <address> includes a decimal fraction, it is rounded to an integer.
Since the memory space from addresses &H0 to &H4D is a special area allocated for input/output, an overrun may occur simply by reading any of these addresses.
In EPSON BASIC, when a PEEK is executed for any of the abovementioned addresses, an FC ("Illegal function call") error occurs. To read these addresses, &H80 must be written into &H7E (to set MSB).

(See POKE.)

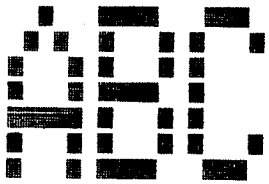
POINT

FORMAT POINT(<horizontal coordinate>,<vertical coordinate>)
PURPOSE To return the status of a dot at a specified location on the graphic screen.
EXAMPLE PRINT POINT(100,10)

REMARKS POINT checks whether or not a dot has been drawn at the location specified by <horizontal coordinate> and <vertical coordinate> on the graphic screen. With the LCD, POINT returns 1 if a dot has been set at the specified location and 0 if no dot has been set. With the external display, color codes 0 to 3 are returned. Please note that the LCD is different from the external display in the range within which you can specify graphic coordinates.

SAMPLE PROGRAMME

```
100 CLS:DEFINT A-Z
110 DIM P(17,7)
120 PRINT "ABC"
130 FOR Y=0 TO 7
140 FOR X=0 TO 17
150 P(X,Y)=POINT(X,Y)
160 NEXT X,Y
170 FOR Y=0 TO 7
180 PRINT
190 FOR X=0 TO 17
200 IF P(X,Y) THEN A$="■
" ELSE A$=" "
210 PRINT A$;
220 NEXT X,Y
```



POS

FORMAT POS(<digit>)
PURPOSE To return the horizontal position of the cursor on the virtual screen or the horizontal position of the printer head.
EXAMPLE X=POS(0)

REMARKS The value of <digit> must be in the range of 0 to 16. If 0 is specified, POS returns the horizontal position of the cursor on the virtual screen.
If any of the integers from 1 to 16 is specified, POS returns the number of characters stored in the buffer (namely, the number of characters that has been output following the output of the LF code and before execution of this function) on the file opened using the integer as a file number.
These numbers also correspond to the horizontal positions of the printer head.

(See CSRLIN and LOF.)

RIGHT$

FORMAT RIGHT$(<string>,<numeric expression>)
PURPOSE To return an arbitrary length of string from the rightmost characters of a string.
EXAMPLE PRINT RIGHT$("ABCD",3)

REMARKS The value of <numeric expression> must be in the range of 0 to 255. If <numeric expression> is greater than the total number of characters in <string>, the entire string will be returned. If <numeric expression> is 0, the null string (length zero) is returned.

(See LEFT$ and MID$.)

RND

FORMAT RND[(<numeric expression>)]
PURPOSE To return a random number.
EXAMPLE A=RND(1)

REMARKS RND returns a random number between 0 and 1. The random number generated varies with the value of <numeric expression> as follows.

- If <numeric expression> is negative, a new sequence of random numbers is generated.
- If <numeric expression> is 0, the last generated random number is repeated.
- If <numeric expression> is positive, the next random number in the sequence is generated.

If <numeric expression> is omitted, the next random number in the sequence is also generated. The same sequence of random numbers is generated each time a RUN or CLEAR statement is executed, if the random number generator is not reseeded by a RANDOMIZE statement.

(See RANDOMIZE.)

SAMPLE PROGRAMME

```
LIST

100 DEFINT N,B
110 DIM B(120)
120 INPUT "NUMBER OF REP
ETITIONS ";A
130 CLS
140 FOR I=0 TO A
150 N=RND*120

160 B(N)=B(N)+1
170 PSET (N,31-B(N))
180 NEXT

>
RUN

NUMBER OF REPETITION
S ?  2000
```

SGN

FORMAT SGN(<numeric expression>)
PURPOSE To return the sign of the value of a numeric expression.
EXAMPLE B=SGN(A)

REMARKS If the value of <numeric expression> is positive, SGN returns 1. If the value of <numeric expression> is 0, SGN returns 0. If the value of <numeric expression> is negative, SGN returns −1.

SIN

FORMAT SIN(<numeric expression>)
PURPOSE To return the sine of a numeric expression.
EXAMPLE PRINT SIN(3.1415926/2)

REMARKS SIN returns the sine of <numeric expression> in radians.

(See COS and TAN.)

SPACE$

FORMAT SPACE$(<numeric expression>)
PURPOSE To return a string of spaces of a specified length.
EXAMPLE A$="A"+SPACE$(10)+"C"

REMARKS SPACE$ returns a string of spaces of the length specified by <numeric expression>. The value of <numeric expression> must be in the range of 0 to 255.

(See SPC and TAB.)

SPC

FORMAT SPC(<digit>)
PURPOSE To output a specified number of blanks.
EXAMPLE PRINT SPC(10);"A"

REMARKS SPC outputs blanks in the number specified by <digit>. This function may only be used in output statements such as PRINT. The value of <digit> must be in the range of 0 to 255.

(See SPACE$ and TAB.)

SQR

FORMAT SQR(<numeric expression>)
PURPOSE To return the square root of a numeric expression.
EXAMPLE A=SQR(2)

REMARKS SQR returns the square root of <numeric expression>.
The value of <numeric expression> must be greater than 0.

STR$

FORMAT STR$(<numeric expression>)
PURPOSE To return a string representation of the value of a numeric expression.
EXAMPLE A$=STR$(123)

REMARKS STR$ converts the value specified by <numeric expression> to a string. For <numeric expression>, you can use any type of numeric constants.

(See STRING$ and VAL.)

STRING$

FORMAT STRING$ (<integer expression>, | <string expression> | <numeric expression> |)

PURPOSE To return a string of specified characters.

EXAMPLE PRINT STRING$(10,65)

REMARKS STRING$ returns a string of characters specified by <string expression> or <numeric expression> in the length specified by <integer expression>. If <string expression> is specified as a string of characters to be returned, all characters having the first character of the string are returned.
If <numeric expression> is specified, all characters having ASCII code specified by the numeric expression are returned. The value of <numeric expression> must be in the range of 0 to 255.

(See STR$.)

TAB

FORMAT TAB(<numeric expression>)

PURPOSE To space to a specified position on the line where the cursor is currently positioned.

EXAMPLE PRINT TAB(10);"ABC"

REMARKS TAB is used only in output statements such as PRINT and LPRINT.
TAB outputs blanks from the current cursor position to the position specified by <numeric expression> counted from the left-hand end on the virtual screen.
The value of <numeric expression> may be in the range of 0 to 255 with 0 as the leftmost position. In other words, value of <numeric expression> corresponds to the remainder when it is divided by the number of characters to be displayed horizontally. If the position specified by <numeric expression> is at the left of the current cursor position, TAB goes to that position on the next line.
Please note the difference between the SPC function and the TAB function.

(See SPC.)

TAN

FORMAT TAN(<numeric expression>)

PURPOSE To return the tangent of a numeric expression.

EXAMPLE A=TAN(3.1416/4)

REMARKS TAN returns the tangent of the value of <numeric expression> in radians.

(See COS and SIN.)

*TAPCNT

FORMAT TAPCNT
PURPOSE To return the value of the microcassette drive counter.
EXAMPLE PRINT TAPCNT
A=TAPCNT

REMARKS TAPCNT function is used to read the value of the microcassette drive counter. The returned value is in the range of −32768 to 32767. This counter value is always returned as positive after the counter has been reset by a WIND command. By assigning a value to TAPCNT, you can set the counter value.

(See WIND.)

SAMPLE PROGRAMME

```
LIST

100 TAPCNT=0
110 OPEN "O",#1,"CAS0:TEST"
120 FOR I=1 TO 10
130 PRINT#1,I;I*I
140 NEXT
150 CLOSE
160 WIND 0
170 OPEN "I",#1,"CAS0:TEST"
180 IF EOF(1) THEN 220
190 INPUT#1,A,B
200 PRINT A,B
210 GOTO 180
220 CLOSE
230 END
RUN
 1             1
 2             4
 3             9
 4             16
 5             25
 6             36
 7             49
 8             64
 9             81
 10            100
>
```


TIME$

FORMAT TIME$="HH:MM:SS"
PURPOSE To return the time kept by the internal calendar clock.
EXAMPLE PRINT TIME$

REMARKS TIME$ displays or sets the time kept by the internal clock of the HX-20. The time is set or displayed in the format "HH:MM:SS" where the value of HH ranges from 00 to 23 and the values of MM and SS range from 00 to 59. Once the correct time has been set, you are not required to set the time again, as the HX-20 clock keeps track of the time and date.

(See DATE$ and DAY.)

USR

FORMAT USR[<digit>](<argument>)
PURPOSE To call machine language subroutine defined by DEFUSR statement.
EXAMPLE A=USR 1(B)

REMARKS USR calls your machine language subroutine (user-defined function) with <argument>. Before calling the user-defined function, it must have been written into the memory, and its execution starting address must have been defined by the DEFUSR statement.
A maximum of 10 user-defined functions can be set by <digit> whose value is in the range of 0 to 9 and corresponds to the digit supplied with the DEFUSR statement for that routine. If <digit> is omitted, USR0 is assumed.
With <argument>, you can transfer a value from EPSON BASIC to your machine language subroutine.

(See DEF USR and Chapter 5, "Machine Language Subroutines".)

VAL

FORMAT VAL(<string expression>)
PURPOSE To return the numerical value of a string expression.
EXAMPLE A=VAL("−123")

REMARKS If the first character of <string expression> is not +, −, &, ., or a digit, VAL (<string expression>) = 0.
If a character other than digits (0 to F in hexadecimal numbers and 0 to 7 in octal numbers) appears, the following characters are ignored. Blanks in <string expression> are also ignored.

(See CHR$ and STR$.)

VARPTR

FORMAT VARPTR(<variable name>)
PURPOSE To return the address of a variable or array.
EXAMPLE PRINT HEX$(VARPTR(A))

REMARKS Any type of <variable name> (numeric, string, or array) can be specified. A value must be assigned to the variable specified by <variable name> before executing VARPTR.
The returned address will be an integer in the range of −32768 to 32767. If a negative address is returned, obtain the actual value by adding it to 65536.
Whenever a value is assigned to a new variable other than arrays, the addresses of the arrays change. Therefore, all simple variables must be assigned before calling VARPTR for an array.

(See Chapter 5, "Machine Language Programmes.")

CHAPTER 5
Additional Information

5.1 RAM files

RAM (random) files are one of the many features of EPSON BASIC.

With RAM files,
- Data can be accessed randomly.
- Data can be changed freely.
- High access speed is assured as compared with the I/O transfer speed to and from peripheral devices.

Because of the above advantages, RAM files may be regarded the same as array variables. RAM files also feature the following:

- Data is retained even after the power switch is turned OFF.
- Data can be shared by plural programmes.

In view of the above features, RAM files are ideal for data to be handled more frequently than those in sequential files, and are thus useful for:
- Storage of variable tables, conversion tables, etc., to be used constantly. (Scientific calculations)
- Storage of data necessary for daily transaction processing (Business management)

In EPSON BASIC, five separate programmes can be stored simultaneously. However, RAM files must be shared by these five programmes and cannot be used at the discretion of plural programmes. In using PUT% statments, be careful so that data files may not be accidentally destroyed by other programmes.

In fact, RAM files as a whole occupy a single area, but the RAM file area may be used as separate files apparently using a DEFFIL statement. The DEFFIL statement specifies the location of the first record in the RAM file area and the locations of records to be read or written by a GET% or PUT% statement after the execution of the DEFFIL will be shifted relatively.

The type of variable of the data to be read by a GET% statement must match that of the data written by a PUT% statement. The number of bytes occupied on the memory space by each type of variable is different as follows.

Integer variable	2 bytes
Single precision variable	4 bytes
Double precision variable	8 bytes
String variable	Indefinite

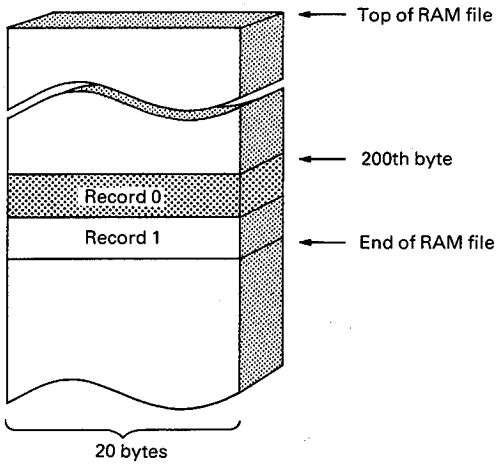
For this reason, if the data written as an integer value is to be read as a double precision or single precision value, the result will be an entirely different value.

The length of one record is determined by a DEFFIL statement and within that record, numeric variables occupy the length specified for each variable type. The remaining length of the record is allocated to string variables. This is because of the fact that the length of a string variable is indefinite, and in GET% and PUT% statements, string variables can be used only after numeric variables.

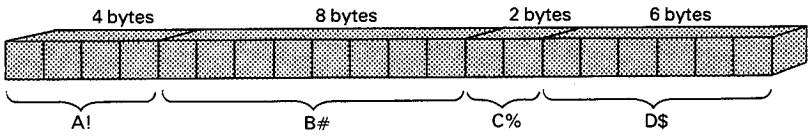
Example:

```
100  DEFFIL 20, 200
 :
200  PUT% 0, A!, B#, C%, D$
 :
300  GET% 0, E!, F#, G%, H$
 :
400  END
```

In this programme, the location of a file is as shown below.



The record length of, for example, Record 0 is allocated as follows.



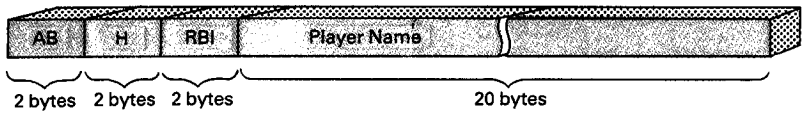
Execution of the programme results as follows.

```
E!=A!, F#=B#
G%=C%, H$=D$
```

As an example of the use of a RAM file, let's prepare a file to record the individual statistics for each of the players of a baseball team, using the player's uniform numbers as file numbers to reference the files.

The first thing you must consider is the record length for each player. In this example, let's deal with the At-Bats, Hits and RBI (Runs Batted In) for each player. You can handle all three values as integer values. You will need to record the player names in alphabetic characters. To this end, let's have a string variable of a maximum of 20 bytes.

AB (At-bats)	2 bytes
H (Hits)	2 bytes
RBI (Runs Batted In)	2 bytes
Player Name	20 bytes
Total	26 bytes



Next, you must determine the total file length required for the team's statistics. You already know the record length for each player. All that you must do here is to multiply the length of the individual record by the number of players on the team. Assuming that the team has 25 players, the total file length will be as follows.

$$26 \times 25 = 650 \text{ bytes}$$

You can now write programmes for recording the individual player's statistics according to the required functions.

5.1.1 Creation of a RAM file

Though somewhat unusual, let's limit the player's uniform numbers to the range 0 to 24. In this programme, you only need to write the AB, H and RBI data available for each player up to this point.

```
100 CLEAR 200,650
110 DEFINT U,A,H,R
120 DEFFIL 26,0
130 INPUT "UNIFORM NO. "
;U
140 IF U>24 THEN 210
150 INPUT "NAME    ";NA$
160 INPUT "AT BATS ";A
170 INPUT "HITS    ";H
180 INPUT "RBI     ";R
190 PUT% U,A,H,R,NA$
200 GOTO 130
210 END
```

Even if you input a player name longer than 20 characters, no error will occur. However, as only 20 bytes are available for the player name field on a record in the RAM file, only the first 20 characters of the name will be stored. Since each of the three variables (AB, H and RBI) requires 2 bytes, the variables are declared as integer type by a DEFFINT statement.

5.1.2 Retrieval and updating of data in RAM file

This programme is used to read the data written into the RAM file by the programme in the preceding example. Since each player name is already entered in the file, all that is necessary for you to retrieve the data relating to a player is to input the player uniform number. If you type "Y" following the displayed data, the individual record can be updated. The newly entered statistics will then be added to the statistics currently recorded.

```
220 CLEAR 200,650
230 DEFINT U,A,H,R
240 DEFFIL 26,0
250 PRINT
260 INPUT "UNIFORM NO. "
;U
270 IF U>25 THEN END
280 GET% U,A,H,R,NA$
290 AV=H/CSNG(A)
300 PRINT NA$
310 PRINT "AT BATS ";A
320 PRINT TAB(10);"HITS
";H
330 PRINT "RBI ";R
340 PRINT TAB(10);"AVERA
GE ";AV
350 PRINT "UPDATE (Y/N)"
;
360 IF INPUT$(1)<>"Y" GO
TO 250
370 PRINT
380 INPUT "AT BATS ";XA
390 INPUT "HITS    ";XH
400 INPUT "RBI     ";XR
410 PUT% U,A+AX,H+HX,R+R
X
420 GOTO 280
```

In addition to the two sample programmes given, you can also write a programme to list the players' performance records in the order of batting average or a programme to select and list the individual records of, say, top 5 players. With these programmes and data files prepared by the HX-20, you can manage your baseball team practically.

5.2 Sequential files

If you consider the recording of music on an audio cassette, you will realize that musical numbers are recorded in sequence and that to listen to a particular musical number, you must advance the tape to the section of the tape where that number is recorded. A sequential file operates on the similar principle except that, instead of music, a sequence of data is recorded on the tape.

The speed and flexibility of RAM files are very attractive, but their memory capacity limits the amount of data that can be handled. For this reason, much data cannot be handled on each RAM file.
Also, when using a RAM file, you must write a programme by taking the internal file structure into account.

In contrast, sequential files feature ease of operation in addition to:

- Unlimited data handling capacity
- Simple file structure

However, despite such advantages, sequential files have the following drawbacks as well.

- Data can only be accessed in sequence.
- Data can therefore not be retrieved quickly.
- Partial changes to the data file cannot be effected.

When you prepare a programme using a RAM or sequential file, the relative merits and demerits of both types of files must be weighed before making your selection.
To use a sequential file, you must observe the following procedure.

(1) Open the file.
(2) Input data from the file to memory using an INPUT# statement, or output data from memory to the file using a PRINT# statement.
(3) Close the file.

Each of sequential files are assigned a file number when they are created and all subsequent I/O operations to and from these files (using INPUT#, PRINT#, etc.) are performed by specifying a particular file with this file number. Therefore, no two files can exist at the same time under the same file number. Also, once a file has been OPENed under a certain file number, no other files can be opened using that number until the file first opened is closed.

Data to be read with an INPUT# statement must match the data written with a PRINT# statement in both the variable type and the number of variables. In a sequential file, data is read in sequence from the beginning of the file. Therefore, if there is any difference in the number of variables, subsequent INPUT# statement will read all the data in incorrect sequence. If there is a difference in the type of variable, a TM ("Type Mismatch") error will occur.

After the I/O transfer using the sequential file, you must always execute a CLOSE (or END) statement. With EPSON BASIC, data transfer between the HX-20 and peripheral equipment is performed in units of 256 bytes. For example, when writing data into the file, a write operation will not take place until 256 bytes of data have accumulated in the buffer. If a CLOSE (or END) statement is not executed at the end of the programme, data will remain in the buffer and not be output. In this case, the file will also be left incomplete without a delimiter being written at the end of the file.

As an example of the use of a sequential file, let's write a programme for an address directory. In the preceding example of RAM file, remember, you had to carefully consider the record structure such as record description items and record length. However, with sequential files, as there are no particular restrictions on the amount of data, you are only required to decide on the record description items and the order in which you wish to file them.

5.2.1 Creation of a sequential file

This programme is to record a list of names, addresses, phone numbers and birth dates on an audio cassette tape.
Although you can write such data as name, address, etc., all together as a string, it is better to record by delimiting these description items for subsequent data utilization.

```
100 OPEN"O",#1,"CAS1:ADR
S"
110 PRINT
120 INPUT "NAME ";NAME$
130 IF NAME$="" THEN 250
140 INPUT "ADDRESS";ADR$
150 INPUT "TELEPHONE ";T
EL$
160 PRINT"DATE OF BIRTH"
170 INPUT "  YEAR";Y
180 INPUT " MONTH";M
190 INPUT "   DAY";D
200 PRINT#1,NAME$
210 PRINT#1,ADRS$
220 PRINT#1,TEL$
230 PRINT#1,Y;M;D
240 GOTO 110
250 CLOSE
260 END
```

If you observe the operation of the audio cassette, you will probably notice that the tape does not advance all the time. This is because of the action of the buffer described in the preceding section. That is, a write operation is performed only after a specified amount of data has accumulated in the buffer. After you input all the data for the required number of persons, press only the RETURN key at line 120 to assign a null string to A$. This will branch the programme to line 250 to CLOSE the file. This step cannot be skipped.
If you press BREAK key to terminate the programme after all the data input has been completed, data will remain in the buffer and the file will not be closed.

5.2.2 Retrieval of data from sequential file

To demonstrate the advantages of using this computer, rather than simply outputting the recorded data as it was entered, let's write a more useful programme to output selected data from the file, for example, the data of only those people born on a certain year.

```
100 OPEN "I",#1,"CAS1:AD
RS"
110 PRINT "PEOPLE BORN W
HAT YEAR?"
120 INPUT N
130 LPRINT " *** BORN IN
";N;"***"
140 IF EOF(1) THEN GOTO
230
150 INPUT#1,NAME$,ADR$,T
EL$,Y,M,D
160 IF N<>Y GOTO 140
170 LPRINT
180 LPRINT NAME$
190 LPRINT ADR$
200 LLISTNT TEL$
210 LPRINT "BORN ";M;"/"
;D;"/";Y
220 GOTO 140
230 LPRINT
240 LPRINT " *** END OF
RETRIEVAL *** "
250 CLOSE
260 END
```

In the preceding file creation programme, you have indicated the end of file by pressing RETURN key. However, when reading a file, this determination is made by the EOF function. If you omit line 140 in the above programme, an IE ("Input past end") error will occur.

5.2.3 Correction of data in sequential files

In a sequential file, you cannot change only a portion of the data recorded on a file. To correct the file, you must prepare another file and write correct data into the new file while reading the old data file. As the HX-20 allows you to use both a microcassette drive ("CAS0:") and an audio cassette ("CAS1:"), you can perform the file updating utilising these two units.

```
100 CNT=TAPCNT
110 OPEN "I",#1,"CASI:AD
RS"
120 OPEN "O",#2,"CASO:WO
RK"
130 IF EOF(1) GOTO 320
140 INPUT #1,NAME$,ADR$,
TEL$,Y,M,D
150 PRINT NAME$
160 PRINT ADR$
170 PRINT TEL$
180 PRINT "BORN ";M;"/";
D;"/";Y
190 PRINT "DO YOU WISH T
O MAKE CORRECTIONS (Y/N)
 ?"
200 IF INPUT$(1)<>"Y" GO
TO 290
210 PRINT
220 INPUT "NAME      ";NA
ME$
230 INPUT "ADDRESS   ";AD
R$
240 INPUT "TELEPHONE";TE
L$
250 PRINT "DATE OF BIRTH
 "
260 INPUT "   YEAR";Y
270 INPUT "  MONTH";M
280 INPUT "    DAY";D
290 PRINT#2,NAME$,ADR$,T
EL$
300 PRINT#2,Y;M;D
310 GOTO 130
320 CLOSE
330 WIND CNT
340 PRINT "PLEASE REWIND
 THE CASSETTE"
350 PRINT
360 PRINT "IF OK, PLEASE
 INPUT "Y""
370 IF INPUT$(1)<>"Y" GO
TO 360
380 OPEN "O",#1,"CAS1:AD
RS"
390 OPEN "I",#2,"CAS0:WO
RK"
400 IF EOF(2) GOTO 440
410 INPUT#1,NAME$,ADR$,T
EL$,Y,M,D
420 PRINT#2,NAME$,ADR$,T
EL$,Y,M,D
430 GOTO 400
440 CLOSE
450 END
```


The actual transfer of data from the old to the new file terminates at line 320. In this state, the new file is in the microcassette drive. For subsequent data file utilisation, you must transfer the data in the new file back to the file in the audio cassette. The tape in the microcassette drive can be rewound using a WIND statement in a programme. The audio cassette, however, must be operated manually. In this sample programme, two files are used so that a larger amount of data may be stored.
If the amount of data to handle is small and all the data can be stored in the memory at one time, corrections of data file can be performed just the same as the RAM file.

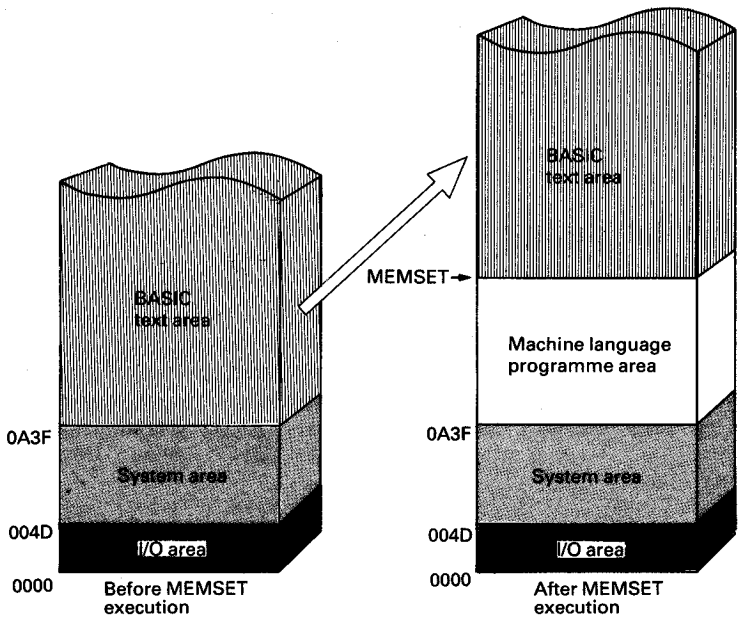
5.3 Machine language programmes

In EPSON BASIC, the USR function and the EXEC statement are provided as functions to call user programmes written in machine language. USR, like other functions, can pass data using an argument. The EXEC statement, however, has no function to pass any variables. In BASIC, even if a programme containing a bug is executed, execution stops but the programme itself will not be destroyed. However, in a machine language programme, even if a single bit is incorrect, all the programmes (including BASIC programmes) may be destroyed.

5.3.1 Memory allocation

When using user programmes written in machine language, memory space must be secured so that BASIC programmes and data are protected and that machine language programmes are not destroyed by any BASIC programmes. In EPSON BASIC, machine language programmes are placed before the BASIC programme area and the memory space for machine language programmes is secured using a MEMSET command. Therefore, you must execute this MEMSET command before loading any machine language programme.

The execution of the MEMSET command secures the machine language programme area and at the same time moves BASIC programmes. Therefore, BASIC programmes previously stored in memory are protected against destruction.



5.3.2 Writing and loading programmes

A machine language programme is loaded into the memory using the MONITOR function of the HX-20. Short programmes can be written into the memory using a POKE statement.
A machine language programme you have written can also be stored as a machine language programme file in the same manner as BASIC programmes using a SAVEM or LOADM command.
The HX-20 has a vacant location for an expansion ROM socket to enable you to store your completely debugged programmes in the expansion ROM, so that you can always use them as the utility software of the HX-20. You can also store less frequently used programmes in the ROM cassette as a programme file.
In this way, programme loading time can be reduced greatly and the operability of the HX-20 can be improved as well.

5.3.3 USR function

The format of the USR function to call a machine language programme is as follows.

USR[(<digit>)](<argument>)

<digit> may be an integer from 0 to 9 and must correspond to the digit supplied in the DEFUSR statement. <argument> may be any numeric or string expression.
The USR function has one argument. The accumulator A contains a value that specifies the argument type.

The values used and their meanings are as follows.

Value in accumulator	Argument type
2	Integer
3	String
4	Single-precision real number
8	Double-precision real number

If the argument type is numeric, the value in the index register shows the address of the "floating-point accumulator" where the argument is stored.
This floating-point accumulator does not refer to the memory location where a variable itself is stored, but refers to a special area used when BASIC performs an arithmetic operation. The actual value is stored in the "floating-point accumulator" in a different form depending on the argument type as follows.

- When the argument is an integer:

	← Address indicated by the index register
Upper 8 bits	
Lower 8 bits	

- When the argument is a single-precision real number:

Exponent	← Address indicated by the index register
Sign bit and upper 7 bits of mantissa	
Middle 8 bits of mantissa	
Lower 8 bits of mantissa	

- When the argument is a double-precision real number:

Exponent	← Address indicated by the index register
Sign bit and upper 7 bits of mantissa	
Lower 8 bits of mantissa	

- When the argument is a string:

  The value of the index register indicates the address of 3-byte data called "string descriptor".

Length of string	← Address indicated by the index register
Upper bits stored address	
Lower bits stored address	

The string descriptor is a pointer which indicates the address where a string type data is actually stored. By referring to the address indicated by this string descriptor, you can find the data passed as an argument.

When the USR function returns a value to BASIC, data must be stored in the same format in the location where the argument was stored (i.e., the location indicated by the floating-point accumulator or string descriptor). Therefore, the type of value returned from the USR function must be the same as that of the value specified as the argument.

To return to a BASIC programme from the USR function, machine language subroutine RTS (&H39) is used. For this reason, the value of the stack pointer when programme control is returned to BASIC must be the same as that when the USR function was called.

5.3.4 EXEC statement

Machine language subroutines can also be executed by an EXEC statement.

EXEC[<address>]

When a machine language subroutine is loaded into memory using a LOADM command, or when the execution starting address has been specified by EXEC, <address> can be omitted.

EXEC only executes the programme from the specified address and has no function to pass an argument. When variables are to be passed between a BASIC programme and a machine language programme called by an EXEC statement, it is performed through the direct read and write of the variables stored in memory. You can do this in the following ways.

(1) To POKE and PEEK the specified addresses of the machine language area using the BASIC programme.
(2) To directly read and write data in the BASIC variable area using the machine language programme.

When you use method (2), you must check beforehand the addresses of the variables to be written or read using the VARPTR function.

5.3.5 VARPTR function

With the VARPTR function, you can check at which address in memory your specified variable is stored. VARPTR is used to pass a variable to a machine language programme to be called by an EXEC statement, or to pass two or more variables using the USR function.

When using the VARPTR function, a value must have been assigned to the variable specified as the argument before execution of VARPTR.

The value returned by the VARPTR function is the top address of the specified variable data. If the variable is a string, the data is not the value of the variable but is a <string descriptor> which is a pointer indicating the address where the string type data is stored.

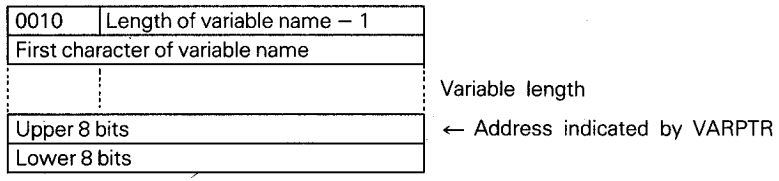
When a variable is stored in memory, the variable type, length of variable name and variable name (maximum 16 characters) are stored immediately before the value of the variable.

(1) Variable type
Using the first 4 bits of the first byte of data, variables are classified according to the length that the data occupies, as follows.

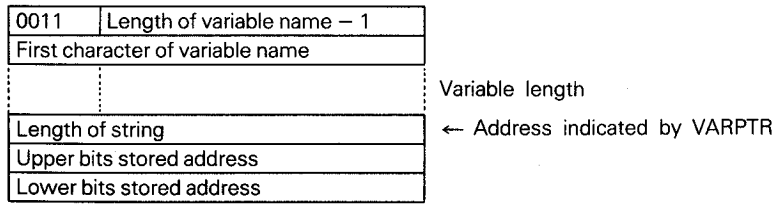
2 : Integer
3 : String
4 : Single-precision real number
8 : Double-precision real number

(2) Length of variable name
Using the last 4 bits of the first byte of data, the actual length of the variable name minus one is stored. Therefore, when the variable name is a single character, 0 is stored as the value.
(3) Variable name
Starting from the second byte of data, the variable name of the length indicated in (2) above is stored in ASCII format. Although the variable name is stored in a maximum of 16 characters, only the required number of bytes are used.

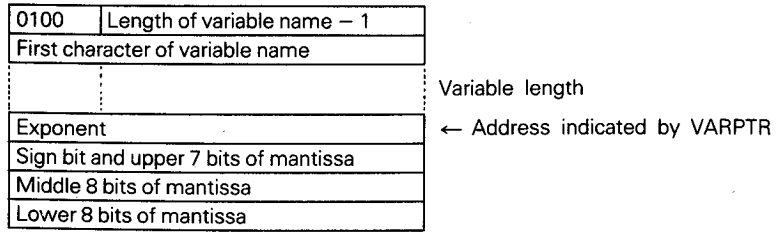
- When the variable type is an integer:



- When the variable type is a string:



- When the variable type is a single-precision real number:



- When the variable type is a double-precision real number:

1000	Length of variable name – 1
First character of variable name	
Exponent	
Sign bit and upper 7 bits of mantissa	
Lower 8 bits of mantissa	

Variable length

← Address indicated by VARPTR

Real-numbers type data are always normalized to omit the most significant bit of the mantissa. Also, whatever the value of the mantissa may be, if the exponent is 0, the value of the mantissa is assumed as 0. Compared to the index register which points the address of the floating-point accumulator when the USR function is called, the value of a variable specified by the VARPTR function indicates the address where the variable is stored. Do not confuse these two types of functions. Be especially careful with integer variables, as they are stored somewhat differently from other variable types.

5.4 How to use the RS-232C port

The HX-20 incorporates an RS-232C interface to allow communication with external devices. The RS-232C port in the HX-20 is normally used to connect an external printer with the HX-20. RS-232C is an EIA Standard for the interface between a MODEM and the associated data terminal equipment.

With this RS-232C interface, the HX-20 can communicate directly with other computers. By connecting an acoustic coupler to the RS-232C interface, the HX-20 can also transfer data to and from remote locations via subscriber lines.

The RS-232C standard is widely in use by various kinds of communications equipment. However, as this standard merely prescribes electrical characteristics, use of signal lines, transmission procedures, etc., differ from one type of equipment to another. Therefore, when connecting an external device to the HX-20, you must carefully check the external device for agreement of interface conditions with those of the HX-20 so that proper communications can be established under the same interface conditions.

With the HX-20, this setting of interface conditions can be performed using BASIC. (See OPEN"COM0:"). Once the uniform interface conditions have been established between the HX-20 and the external device, all other procedures are the same as when you use normal files. If you specify <file number> in a programme statement, you can perform I/O operations using PRINT# and INPUT# statements. With the HX-20 in the direct mode, you can directly transfer programmes between the HX-20 and external devices using LIST"COM0:" and LOAD"COM0:" commands.

NOTE:
If you execute an output command by specifying "COM0:" as the device name when no external device is connected to the RS-232C port or when the connection between the HX-20 and the external device is faulty, the operation of the HX-20 may terminate. Should this happen, press **BREAK** *key, to return the HX-20 to command level.*
Also, when a long line of characters is transferred without specifying the print width for "COM0:", the default value will be in effect and automatic line feed will take place at every 80 characters. When you send a long string exceeding 80 characters, you must first execute WIDTH "COM0:", 255 to set the print width as infinite.

5.4.1 Interfacing with optional devices

All the devices available as the options to the HX-20 have standardised signal lines. Therefore, as long as the exclusive interface cables specified by EPSON are used, you are not required to have a special knowledge of the RS-232C interface.

(1) Interfacing between two HX-20 units

You must use interface cables (optional cable set #715) when interfacing your HX-20 with another HX-20 unit. In this method, you do not need to set the interface conditions and can omit <BLPSC>.

NOTE:
When directly transferring programmes using a LIST "COM0:" or LOAD "COM0:" command, data may be lost during the data transfer if the data processing speed of the receiving HX-20 cannot catch up with the data transfer speed of the transmitting equipment. To transfer programmes properly in such a case, be sure to lower the bit rate of the transmitting equipment.

(2) Interfacing with a terminal printer

When connecting an external terminal printer to the HX-20, the printer must be equipped with an RS-232C interface. (For any of EPSON MX series printers, use interface board #8141 or #8145.) You must also use special interface cables (cable set #714).
There are two methods of using a terminal printer depending on whether you use the interface conditions of the HX-20 or those of the terminal printer.

- When setting the interface conditions at the terminal printer:

You need not set the interface conditions <BLPSC> for your HX-20. By simply specifying <device name> as "COM0:" in either direct mode or programme mode, you can use the terminal printer just the same as the built-in microprinter "LPT0:".

Set the interface conditions of the terminal printer as follows. (For details, see the user's manual of the applicable interface board.)

Bit rate	4,800 bps
Word length	8 bits
Parity bit	No parity
Stop bit length	2 bits

- When setting the interface conditions at the HX-20:

  If you specify <device name> as "COM0:", you must set the interface conditions <BLPSC> as follows.

Bit rate	
Word length	Set to the same parameters
Parity bit	of the terminal printer.
Stop bit length	
Control lines active	Specify "B" (hex) when using an EPSON MX series interface board.

Example:

Bit rate	300 bps
Word length	7 bits
Parity bit	Even parity
Stop bit length	1 bit
Control lines active	Handshaking by DSR signal (To match to the factory-set conditions interface board #8145, specify "COM0"(27E1B)".)

NOTE:
The maximum bit rate of the HX-20 is 4,800 bps. If the bit rate of the printer has been set at 9,600 bps, you must change the setting of the printer to 4,800 bps, as you cannot make this parameter compatible with that of the printer at the HX-20.

(3) Interfacing with an acoustic coupler
To interface the HX-20 with the optional acoustic coupler CX-20, you must use special interface cables (optional cable set #706).
When using the acoustic coupler, the interface conditions of both the HX-20 and CX-20 must be set to match to those of the transmitting/receiving equipment.

Word length Parity bit Stop bit length	Set to the same parameters of the transmitting/receiving equipment.
Bit rate	Must be the same as that of the transmitting/receiving equipment but must be 300 bps max. due to limitations of telephone lines. ($0 \leqslant B \leqslant 2$)
Control lines active	As all control lines are normally used, specify "2" (hex).

5.4.2 Interfacing with other external equipment

To interface the HX-20 with external devices other than those specified as the options to the HX-20, you must have a deep knowledge of the RS-232C interface. You can set the bit rate, word length, parity bit and stop bit length to match to those of the device to be connected to the HX-20. But the signal lines used, the method of handshaking, etc. are different from one device to another.

Simply connecting the signal lines of the same name will not assure successful communication between the two devices. You must have correct understanding of the function of each signal line.

The HX-20 uses the following 9 signal lines.

Pin No.	Signal	Signal direction	Function
1	GND	–	Signal GND
2	TXD	Out	Transmitted data
3	RXD	In	Received data
4	RTS	Out	Request to send
5	CTS	In	Clear to send
6	DSR	In	Data set ready
7	DTR	Out	Data terminal ready
8	CD	In	Carrier detect
E	FG	–	Protective GND

- Pin No. 1 – GND
  Signal of zero potential which serves as reference for all other signals.
- Pin No. 2 – TXD
  Output signal line to send data from the HX-20. Data is output in negative logic.
- Pin No. 3 – RXD
  Input signal line to receive data by the HX-20. Data is input in negative logic.
- Pin No. 4 – RTS
  Output signal line to request the grant for data transmission. When this signal (normally of positive potential) is active, it indicates that the HX-20 is ready for data transmission. By specifying "C" in <BLPSC> setting, it can be changed to that a signal of negative potential has significance.

- Pin No. 5 – CTS

  Input signal line to receive the grant for data transmission. If this signal line is at positive potential, the HX-20 judges that the request for data transmission has been granted, and starts data transmission.
  If the potential of this signal changes to negative, the HX-20 stops transmission and waits until the signal potential becomes positive to restart transmission. By specifying "C" in <BLPSC> setting, this signal can be ignored. If it is done so, the signal potential is always assumed as positive. When interfacing with equipment other than the acoustic coupler, this signal is often directly connected to the RTS signal, but in the HX-20, data transmission and reception can also be controlled by this signal.

- Pin No. 6 – DSR

  Input signal line used by the HX-20 to find if the counterpart is ready to receive data. When the potential of this signal is positive, the HX-20 judges that the counterpart is ready to receive data. Therefore, data tranmission can be terminated by changing this signal potential to negative. This is the main signal used by the HX-20 to control the transmission of data. By specifying "C" in <BLPSC> setting, this signal can be ignored. If this is done, the potential of this signal is always assumed as positive. This signal is originally intended to function as a response signal to the DTR signal to detect whether or not the communication circuit is ready. However, in the HX-20, this line is used as an input signal line to detect the BUSY state of the transmitting/receiving peripheral. Therefore during data transmission, the HX-20 always monitors this control line.

- Pin No. 7 – DTR

  This signal line is intended to output a signal by the HX-20 to the peripheral devices to request information as to whether the communication circuits are ready or not. This information request is made when the level of this signal is positive.
  In the HX-20, this signal line has another function. When the HX-20 is to function as a receiving equipment, this output signal informs the transmitting equipment of whether or not the HX-20 is ready to receive data. When the potential of this signal is high-impedence, the HX-20 is not ready to receive data, that is, BUSY.
  NOTE:
  DTR is connected to the power supply line of the RS-232C and high-speed serial interface driver. For this reason, if the RS-232C interface or high-speed serial interface is turned ON, DTR signal will automatically be activated.

- Pin No. 8 – CD

  Input signal line to detect that data is being sent from the transmitting equipment to the HX-20. When the potential of this signal is positive, the HX-20 judges that data is to be sent from the peripheral connected to the HX-20. This control line is required when the HX-20 is connected to the acoustic coupler. In all other cases, this signal can be ignored by specifying "C" in <BLPSC> setting.

In the HX-20, data transmission and reception can be controlled by signal lines CTS and DSR. You can find the status of data reception by signal line DTR. When you connect the HX-20 to any other external device, you must check carefully the functions of all the signal lines provided in the device and make required connections based on the function and not the nomenclature of each line.