HP 86/87
Disk Organization
Advanced Programming

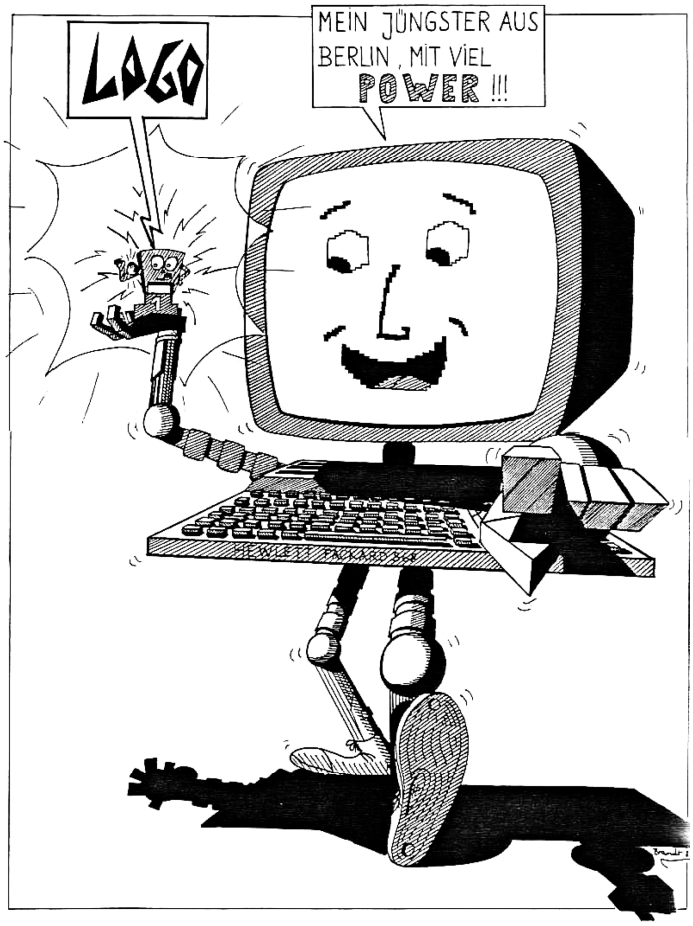
Bit/String Manipulation
Machine Language Control

SYSEXT

Operating System Extension

By André Koppel

A partial translation of the SYSEXT manual by Martin Hepperle, Revision 3, November 2023



Background

The SYSstem EXTensions documented here had been developed by André Koppel. They had been sold in form of binary programs as well as EPROMs. While the binary program could be loaded with the LOADBIN command, the EPROM version required the HP 82929A Programmable ROM Module.

1 Mass Storage Commands

DCAT$ (EntryNumber)

This is a string function with a single numeric parameter. It returns the catalogue entry with the given 1-based number in the following format, similar to a CAT command:

```
NNNNNNNNNNTTTTLLLLLAAAAA
```

Here N is the name of the file, T the type (DATA, BPGM, ASSM, etc.), L is the record length and A the number of records.

If the EntryNumber is larger than the number of files in the catalogue, a zero length string is returned.

DCATNEXT$

This function can only be used after a call to DCAT$. DCATNEXT$ returns the next catalogue entry as a string as described under DCAT$. The function gets ist data from a buffer which has been created by the DCAT$ function. When the buffer is empty the mass storage device is accessed to fill the buffer again.

When DCATNEXT$ has reached the last catalogue entry any subsequent calls return null length strings.

Example: A program shall be loaded by the *Autostart* program when the computer is started. The following requirements shall be met:

1. The *Autostart* program shall load the first program.
2. The *Autostart* program shall change some system settings, load some binary programs and then start the main program via a CHAIN command.

As the main program is changed regularly and the versions are distinguished by numbered program file names, the name used in the CHAIN command cannot be hardcoded. Instead, the *Autostart* program must search for the name with the highest version number... The names of the main program are structured like 2DINPUT112, 2DINPUT113, 2DINPUT114 etc...

```
10 ! Autostart program for automatic loading of the latest version
20 ! of the main program
30 DIM C$[25]
40 LOADBIN "SYSEXT"
50
60 PRINTER IS 704 @ PLOTTER IS 705 ! System settings
70 !
80 !
90 ! More instructions
100 !
110 !
120 TEST$="2DINPUT" @ NR=0
130 C$=DCAT$(1)
140 LOOP @ IF C$="" THEN GOTO ENDE
150 IF C$[1,7]="2DINPUT" THEN NR=MAX(NR,VAL(C$[8]))
160 C$=DCATNEXT$ @ END LOOP
170 ENDE: CLEAR @ AWRITE 0,0,"Loading 2DINPUT"&VAL$(NR)
180 CHAIN "2DINPUT"&VAL$(NR)
190 END
```

The program must be stored on the default disk using the file name `Autost`.

FLOCATE (FileName$)

This function searches for the given file name on a mass storage unit. If the file is found, the number of the first record of the file is returned. It is then possible to access this record using the functions RSECTOR and WSECTOR. If the file does not exist a result of zero is returned. The starting record number of a regular file can be between 3 and 65535. If the mass storage device is a HP8290X disk drive, the result is between 3 and 1059. The larger numbers are possible when a Winchester hard disk drive of up to 17 MB capacity is used (in conjunction with the Mass Storage ROM). The file name can contain the mass storage unit identifier or volume label.

Another example for DCAT$, DCATNEXT$, FLOCATE

```
CAT
[ Volume ]: HPLIF
Name        Type  Bytes   Recs
PEEK86S     ASSM    256      6
PEEK86      BPGM    256      1
GRAPH       PROG    256      3
... more files

DCAT$(1)
PEEK86S   ASSM0025600006
DCATNEXT$
PEEK86    BPGM0025600001
DCATNEXT$
GRAPH     PROG0025600003
FLOCATE("PEEK86S")
16
FLOCATE("PEEK86")
22
FLOCATE("GRAPH")
23
```

The FLOCATE function should not be used together with other I/O commands like PRINT as this can lead to error messages.

RSECTOR Buffer$, RecordNumber, Msus$

This statements reads an arbitrary record from the mass storage device specified by Msus$ and returns the content in Buffer$. The string variable Buffer$ must be allocated to a length of at least 256 characters using a DIM statement.

The Msus$ cannot be a volume label – it must be of the form ":Dxxx".

The RecordNumber depends on the drive type. For HP8290X drives the record number can be within 0 to 1119, larger disks using the Mass Storage ROM can have up to 65535 records.

An error message is produced when a record number which exceeds the limit is used.

WSECTOR Buffer$, RecordNumber, Msus$

This statement writes the 256 bytes contained in Buffer$ to the mass storage device specified by Msus$. The string variable Buffer$ must be allocated to a length of at least 256 characters using a DIM statement.

The Msus$ cannot be a volume label – it must be of the form ":Dxxx".

The RecordNumber depends on the drive type. For HP8290X drives the record number can be within 0 to 1119, larger disks using the mass storage ROM can have up to 65535 records.

An error message is produced when a record number which exceeds the limit is used.

2 Mathematical and String Functions

The following functions allow for bit, string and number manipulation and conversions.

ADR$ (Number)

This function converts a number between 0 and 16777215 into a string of exactly 3 characters. The result is „reversed“, i.e. the least significant byte is returned as the first character. The result equals the representation of an address in memory.

ADR (String$)

This function converts a string of up to 3 characters into a number between 0 and 16777215. The input string contains the bytes in this order: least significant to most significant byte. If less than 3 characters are specified, the missing characters are replaced by CHR$(0). If more than 3 characters are given, the trailing characters are dropped.

Examples:

```
ADR("███")                    ! returns 16777215   ("█" is CHR$(255))
ADR("██")                     ! returns 65535
ADR("█")                      ! returns 255
ADR("█"&CHR(0)&CHR(0))  ! returns 255
ADR("123")                    ! returns 3355185
```

The function can be used in conjunction with PEEK$ and EMC PEEK$ to calculate system addresses. To calculate the start of a BASIC program it is only necessary to lookup the address of the system variable FWCURR (at 100006 octal) and convert the three bytes to a number. This is performed by the following command:

```
ADR(PEEK$(100006,0,3))
```

Notes:

If desired, the resulting number can be converted by D_O into the octal system. The string reversal can be performed by using the REV$ function.

AND$ (String1$, String2$)

This function performs a bitwise AND of the bytes contained in the two strings. The AND operation allows for example to clear a single bit or a set of bits.

Both strings must have the same length, otherwise an error is generated.

BLANK$ (Count)

This function returns a string composed of the given number of space characters (ASCII code 32). BLANK$ is about 2.5 times as fast as the RPT$ function for this purpose.

BSET? (String$, Mask)

This function tests the bytes in the given string parameter against a bitmask. The string is searched until a byte is found where at least one of the bits defined in Mask is set. If no such byte is found, the result is zero.

The following example for AND$ and BSET? tests whether bit 3 is set in the number 7.

```
AND$(CHR$(7),CHR$(4)) = CHR$(4)
 1
BSET?(CHR$(7),3)
 1
```

CBIT$ (String$, Mask)

This function clears the bits specified in the parameter Mask in all bytes of the given string.

CHKSUM (String$)

This function calculates the checksum of the bytes in the given string. The checksum is the sum of all character codes. The length of the string can be up to 8192.

D_O (DecimalNumber)

This function converts a number from the decimal to the octal system. Three bytes in the given unsigned number are converted so that the decimal number can be between 0 and 16777216.

The function is very useful for the PEEK and POKE functions which expect their arguments in the octal system.

The inverse conversion can be performed by the function O_D.

FACT (Number)

This function calculates the factorial of the given number.

HEX$ (String$)

This function converts the given string with a length of up to 16384 bytes into the hexadecimal system.

HEX_ASC$ (String$)

This function converts the given string which contains a hexadecimal representation of a sequence of bytes into a string of bytes. If the string length is odd, the high nibble of first byte is set to zero.

This functions is the inverse of HEX$.

Examples:

```
HEX$(CHR$(255)&CHR$(0))
FF00
HEX_ASC$("414243")
ABC
```

NUMBER? (String$)

This function tests whether the given string starts with a number. It is useful to avoid errors when the VAL function is used.

The function returns 0 if the string does not begin with a number. If the string starts with an integer number the return values is 2 and a real number produces a return value of 1.

The function does not test the range of the number. Thus the subsequent VAL function may return an overflow warning.

```
NUMBER("1E9999")
 1
VAL("1E9999")
Warning 2 : OVERFLOW
 9.99999999999E499
```

O_D (OctalNumber)

This function converts a number from the octal to the decimal system. The octal number can be between 0 and 77777777. The function is very useful for the PEEK and POKE functions which expect their arguments in the octal system.

The inverse conversion can be performed by the function D_O.

```
O_D(377)
 255
```

ODD (Number)

This function returns 1 if the given number is odd, otherwise it returns 0. The function is useful when the parity of a number has to be checked.

```
ODD(0)
 0
ODD(1)
 1
```

OR$ (String1$, String2$)

This function performs a bitwise OR of the bytes contained in the two strings. The OR operation allows for example to set a single bit or a set of bits.

Both strings must have the same length, otherwise an error is generated.

```
NUM(OR$(CHR$(1),CHR$(2)))
 3
```

REV$ (String$)

This function returns a string with the characters of string reversed.

```
REV$("ABC")
CBA
```

ROUND (Number, Digits)

This function rounds the given Number to the specified number of digits. Digits denotes the absolute position in the mantissa, not digits after the decimal character. This means that this command does not perform the more common rounding of the fractional part. The mantissa of a floating point number is internally stored in a normalized form so that 123456789 is stored as $0.1234567890 \times 10^{10}$ while 0.012345678 is stored as $0.1234567890 \times 10^{-1}$. Both have the same mantissa. The following example shows the effect of ROUND on these numbers.

Examples:

```
ROUND(1234567890,4)  => 1234000000
```

```
ROUND(1.234567890,4) => 1.235
ROUND(0.012345678,4) => 0.01235
```

RPT$ (String$, Count)

This function returns a string which contains a concatenation of Count occurrences of the given string. This function is useful to generate on-screen forms and for similar applications. If a row of blanks (space characters) is needed it is faster to use the BLANK$ function. Count cannot exceed 32768.

Example: Clear the screen with inverse blanks

```
DISP RPT$("█",1920)
```

SBIT$ (String$, BitMask)

This function returns the given string where each character has the bits defined in BitMask set. This means that each character is ORed with the given BitMask. This is useful for creating on-screen forms.

Example: Display the given string in inverse mode (black on white resp. green)

```
DISP SBIT$(A$,128)
```

TRIM$ (String$)

This function returns the given string with leading and trailing blanks deleted. Only the blank character (ASCII 32) is removed, other characters which display also as a space (e.g., ASCII 13) are not touched.

```
"<"&TRIM$(" ABC  ")&">"
<ABC>
```

XOR$ (String1$, String2$)

This function returns a string which contains the result of eXclusively ORing each character in String1$ with the corresponding character in String2$. Both strings must have the same length. The function is useful e.g., to store flags in a very compact form (8 per character).

Example: Flip the normal/inverse display mode of each individual character in the given string (this is not the same as the example given for the SBIT$ function)

```
DISP XOR$(A$,RPT$(CHR$(128),LEN(A$)))
```

3 Advanced BASIC Functions and Statements

The following describes the commands not in alphabetical order because some of them only make sense in combination. The descriptions of the corresponding commands are grouped together.

ROMs and Binary Programs

BPGM? (Number)

This function searches the system RAM for the given binary program. Each binary program has a unique number between 1 and 255 which identifies the program. This function allows to test whether a required binary program is already loaded without resorting to ON ERROR construct.

Example: A program needs the binary programs TDGRAPH and BINCALC1 (having number 41 and 48)

```
10 IF NOT BPGM? (41) THEN LOADBIN "TDGRAPH"
20 IF NOT BPGM? (48) THEN LOADBIN "BINCALC1"
```

If you don't know the number of a binary program you can use the following routine to find out which binary program(s) is/are loaded:

```
10 FOR I=1 TO 255
20 IF BPGM? (I) THEN DISP "Binary program ";I;" is loaded."
30 NEXT I
```

You simply load the unknown binary program and run the program. If you use the SYSEXT binary program instead of the ROM version you will also find the number of the SYSEXT BPGM as 56.

Note that you cannot load binary programs with the same number. If you try, you will receive a BAD BIN LOAD ERROR.

ROM? (RomID)

This function returns a 1 if the ROM with the specified ID is installed, otherwise it returns 0. It can be used to manually find out which ROMs are plugged into the computer or a program can test whether a required ROM is installed and then either issue an error message or use alternate routines. Thus you can avoid that a program runs for a long time to finally stop with a missing ROM error.

Example: A program to list all plug-in ROMs

```
10 DIM A$[30]
20 FOR J=0 TO 255
30 A$=""
40 IF NOT ROM?(I) THEN GOTO 90
50 ON ERROR GOTO 80
60 RESTOREX I+200
70 READ A$
80 DISP "ROM# ";I; " ";A$
90 NEXT I
100 PAUSE
201 DATA System-Main
202 DATA System-Graphics
214 DATA MIKSAM
224 DATA Language
240 DATA Assembler
256 DATA Sysext
376 DATA Matrix 1
377 DATA Matrix 2
392 DATA I/O
407 DATA Extended Mass-Storage
408 DATA System Mass-Storage
409 DATA Electronic Disk
431 DATA Advanced programming 2
432 DATA Advanced programming 1
440 DATA Plotter
```

Example: A solution to catch missing ROM errors could be

```
5000 PlottingRoutine:
5010 IF ROM?(240) THEN PLOTTER IS 704 @ GOTO 5030
```

```
5020 DISP "No Plotter ROM installed" @ PLOTTER IS 1
5030 !
5040 ! actual plotting routine starts here
5050 !
```

Controlling multiple ON ERROR Statements

The Series 80 computers offer the ON ERROR construct to capture and handle errors. While this is a fine solution it is somewhat limited because it is not possible to nest multiple ON ERROR statements. Especially when working with subroutines such a feature would be very useful.

The SYSEXT ROM offers two functions to circumvent this problem. Before describing these functions a short explanation of how ON ERROR works is in order.

In the memory of your computer there is a memory location called ERGOTO. When the program starts its value is set to zero. If an error occurs, the system checks this location and outputs an ERROR message on screen if the value is still zero.

However, if you have used an ON ERROR statement the ERGOTO location contains the (relative) address to the error handler (i.e., GOTO XXX or GOSUB XXX) specified in the ON ERROR statement. If an error occurs, and ERGOTO contains a non-zero value the computer jumps to the address of the GOTO or GOSUB statement.

ERRBR?

This function returns a real number which represents the value stored in the memory location ERGOTO. If it is zero, no ON ERROR is active. A program should store this number for later usage. It can then activate a new ON ERROR statement, which will of course overwrite the value stored at ERGOTO. But we still have the copy of the previous value.

SET ERRBR Number

This command stores a number which has been produced by a previous call of the ERRBR? function in the memory location ERGOTO. Thus is restores the previous ON ERROR state.

The ERRBR? / SET ERRBR pair can be used inside a subroutine to preserve an active ON ERROR handler before a new ON ERROR handler for the subroutine is installed. Before returning from the subroutine the previous error handler can be reactivated again by SET ERRBR.

If it is zero, no ON ERROR is active. A program should store this number for later usage. It can then activate a new ON ERROR statement, which will of course overwrite the value stored at ERGOTO. But we still have the copy of the previous value.

Example: Preserving and restoring an ON ERROR handler

```
1000 Subroutine:
1010 Branch=ERRBR? ! save
1020 ON ERROR GOTO Local_handler
1030 !
1040 ! Subroutine code
1050 !
1060 SET ERRBR Branch ! restore
1070 RETURN
1080 Local_handler:
1090 ! do something about this local error
1100 GOTO 1060
```

Of course you can nest this construct to capture multiple levels of error handling in subroutine or user defined function call chains.

Extended Display Control

The normal DISP commands do not allow the development of masked on-screen forms. Also, DISP is relatively slow. SYSEXT provides several new functions to give the programmer (almost) absolute power over the display.

AWRITE [Row, Column [, String$]]

This function places the cursor at the given location and switches it off. The maximum allowable row position depends on the PAGESIZE. In case of PAGESIZE 16 it is 15, for PAGSIZE 24 it is 23. The column number wraps at 79, so that AWRITE 0,260 is equivalent to AWRITE 2,20.

The Row and Column parameters always refer to the current screen page. They do not scroll the page, (which can be achieved with the START CRT AT command).

Note that there is no clipping to the visible screen page, so that the cursor may be located outside of the visible area.

The optional string parameter defines a string which is written to the display at the specified position. In contrast to DISP no linefeed is performed and the cursor stays at the starting position.

AWRITE is not executed in GRAPHALL mode, in GRAPH NORMAL mode the command switches to the ALPHA screen.

Example: Clearing all screen pages with ALPHA ALL and PAGESIZE 24

```
10 ALPHA ALL @ PAGESIZE 24
20 FOR I=1 TO 9
30 AWRITE 24,0
40 CLEAR
50 NEXT I
60 END
```

AWRITE 24,0 places the cursor on the first character of the next page and CLEAR then clear this page but also makes this page the active page. After 9 pages have been cleared we are back at the first page.

Example: Composing a form with some visual effect

```
10 DIM EFFEKT$[80]
20 PAGESIZE 24
30 EFFEKT$="This is the first line of form" @ R=0
40 GOSUB VISUAL
50 EFFEKT$="This is the second line of form" @ R=1
60 GOSUB VISUAL
70 EFFEKT$="This is the third line of form" @ R=2
80 GOSUB VISUAL
90 !
100 ! do something useful
110 !
900 PAUSE
1000 VISUAL:
1010 FOR I=23 TO R STEP -1
1020 AWRITE I,0,BLANK$(80)
1030 AWRITE I,0,EFFEKT$
1040 BEEP I,I @ WAIT 70
```

```
1050 AWRITE I,0,BLANK$(80)
1060 NEXT I
1070 AWRITE R,0,EFFEKT$
1080 RETURN
```

Enter and enjoy!

If you want to display numeric values using AWRITE, you have to convert them with VAL$, of course.

AREAD String$

This function reads a string from the display at the current location. The length of the string defines how many characters are read. The current position can be set with a preceding AWRITE without a string parameter. This also switches the inverse cursor off, which would otherwise also be read into the string (as a character with its highest bit set). Unlike INPUT, AREAD read all characters and does not stop at separator characters like a commas or carriage returns.

Example:

```
10 DIM A$[100]
20 CLEAR @ M=0 @ A$=""
30 M,H=0 @ FOR K=1 TO 17 @ RESTORE
40 IF H THEN H=0 ELSE H=128
50 FOR I=1 TO 110
60 READ B
70 AWRITE 0,M,SBIT$(XOR$(CHR$(B),"0"),H)
80 M=M+1
90 NEXT I
100 NEXT K
110 PAUSE
120 DATA 119,66,81,69,28,16,68,85,69,66,85,66,16
130 DATA 118,55,85,69,94,84,28,16,89,67,68,16,81
140 DATA 92,92,85,16,100,88,85,95,66,89,85,16,69
150 DATA 94,84,16,87,66,42,94,16,84,85,67,16,124
160 DATA 85,82,85,94,67,16,87,95,92,84,85,94,85
170 DATA 66,16,114,81,69,93,30,16,107,125,85,64
180 DATA 88,89,67,68,95,64,88,85,92,85,67,28,16
190 DATA 81,69,67,16,119,95,85,68,88,85,23,67,16
200 DATA 118,81,69,67,68,109,16
```

Enter and enjoy!

START CRT AT Row

This command scrolls the screen so that the display starts at the given Row. The Row count starts at 1. The maximum number is either 204 or 54 depending on whether the screen is in ALPHA ALL resp. in ALPHA NORMAL mode. The cursor is not moved (use the CLEAR or AWRITE commands to move the cursor to the current page).

Example: a simple typewriter program

```
10 DIM A$[80]
20 START CRT AT 1 @ PRINTER IS 704 @ PAUSE
30 FOR I=1 TO 62
40 START CRT AT I
50 AWRITE 0,0 @ AREAD A$
60 PRINT A$
```

```
70 NEXT I
80 PRINT CHAR$(12)
90 END
```

Type the program in and press the [RUN] key. The program stops at PAUSE in line 20. Now you can enter text into the whole screen memory using the cursor and [ROLL] keys as needed. When finished, press the [CONTINUE] key and the content of the screen is sent line by line to the printer.

Extended Keyboard Control

The following commands contain some which are useful in RUN mode while other can be used in CALC mode. First we describe the functions suitable for CALC mode.

MASK

The command masks the keyboard and adds new function to two previously unused keys:

- The [SHIFT][RUN] key combination is assigned to output the string "LIST". Pressing [SHIFT][RUN] causes that the command LIST is written to the screen and you can add parameters like starting line number etc. Thus you do not have to enter the 4 characters LIST manually.
- The [SHIFT][END LINE] key combination toggles between normal and inverse video typing. Pressing [SHIFT][END LINE] once switches the inverse video mode on. All subsequent keys will be displayed in inverse video. This is very useful to compose forms with DISP or AWRITE commands. Pressing [SHIFT][END LINE] again switches the input mode back to normal. Only the character keys are affected, cursor and other system control keys work as usual.

UNMASK

The command switches the keyboard masking (MASK) off. The SYSEXT ROM switches the masking automatically according to the following table

Action	**Result**
Switch On	MASK
SCRATCH	MASK
RESET	UNMAKS
RUN	UNMASK
Stop on error	MASK

The next commands control the keyboard.

TAKE KEYBOARD

The command locks the keyboard. Any subsequent key-press will not show a character on the display or stop a running program. Instead they are stored inside an internal 80 byte buffer. The only keys that cause immediate action are the control keys [k1] to [k14] and the [RESET] key.

In an error occurs during a program run, the SYSEXT ROM releases the keyboard.

RELEASE KEYBOARD

The command unlocks the keyboard. All subsequent key-presses will show a character on the display as usual.

KEY$

The function returns the next character from the keyboard buffer which has been activated by TAKE KEYBOARD. If the buffer is empty, a zero length string is returned. In order to read a key from the buffer the following code fragment could be used:

```
1000 K$=KEY$ @ IF NOT LEN(K$) THEN 1000 ELSE K=NUM(K$)
```

NOT BLOCKED KEYS String$

The command allows defining keys which will remain unlocked after a TAKE KEYBOARD command. The String$ can contain up to 20 characters. Some keys which may be unblocked could be [CLEAR], [A/G], [ROLL UP], [ROLL DN] keys.

Deleting Subroutine Levels

The Series-80 computers can nest up to 256 subroutine levels. While this limit is rarely exploited, the nesting of subroutines may lead to another problem: it is possible to leave a subroutine by a GOTO statement as this command works globally. This is not considered good programming style, but may sometimes come in handy when a quick return from a deep subroutine level is desired. If this happens in a program the subroutine level stays on the call stack which may eventually overflow. The following keyword can be used to fix this problem.

POP RETURN

The command deletes one level from the return stack. The command is ignored if no subroutine level is currently active. After calling POP RETURN the subroutine can be left with a GOTO statement or with a RETURN statement which then returns to the next higher subroutine or main program.

C_RETURNS

The command deletes all level from the return stack. The command is ignored if no subroutine level is active. It allows the program to return from a deeply nested subroutine level. This command can also be used in the main program from time to time if in doubt to reset the call stack (which is indeed a fix albeit a bad one for a more serious programming problem).

Processing BASIC Statements Contained in Strings

The BASIC system offers no simple possibility to process an equation or an arbitrary BASIC expression contained in a string variable. For many programs this would be a welcome feature to allow user customization or calculations as part of the user input.

For example most plotting programs require that the user inserts equations into certain program lines or subroutines. It would be more friendly to allow the user to enter the functions to be plotted in from of a string.

The following three keywords allow working with equations in a flexible way.

EXECUTE String$

The command can be used inside a program only, not in CALC mode. The String$ can contain a BASIC expression of up to 152 characters in length. This expression will be checked for syntax errors and then processed. Any variables in the expression must be defined and allocated before calling EXECUTE. Note also that EXECUTE processes any BASIC keywords so that some of them should be avoided:

SCRATCH, LOAD, STORE, DELETE, SAVE, GET, IF … THEN, GOTO

These functions would either destroy the current program or try to jump to non-existing line numbers. Normally this restriction should be no problem.

The main application of this command are equation processors as they are required for

- programs for numerical integration,
- programs for numerical root finding,
- plotting programs,
- data manipulation and statistics programs
- text processing programs.

Example: a simple program to demonstrate the EXECUTE$ statement

```
10 DIM A$[50]
20 X=0 ! allocate variable X
30 A$="FOR X=1 TO 20 @ BEEP X,X @ NEXT X"
40 EXECUTE A$
50 A$="SIN(15)*TAN(60)+LOG(10)*45"
60 EXECUTE "X=”&A$
70 DISP X
80 END
```

Line 20 is needed because EXECUTE cannot create new variables. Note how the result of a calculation is assigned to this variable in line 60.

TOKEN$ (String$)

The command tests the BASIC expression contained in String$ for proper syntax and then translates it into a tokenized form. The result is a string which contains the BASIC expression in the internal tokenized format which can be processed rapidly by the system. The result of the TOKEN$ function can be processed by a following call to TOKEN EXECUTE.

The same limitations as described for the EXECUTE command also apply to the functions that are allowed in the supplied String$.

Compared to repeated calls of the EXECUTE keyword a single call to TOKEN$() and repeated calls of TOKEN EXECUTE are faster.

TOKEN EXECUTE String$

The command processes the syntax-checked, parsed and tokenized string as returned by the TOKEN$() function. It is the programmers responsibility that the String$ parameter contains properly created data, otherwise the system will crash or hang.

Working with BASIC Program Line Numbers

This section describes commands which refer to the line numbers of the current program.

LINE?

The function returns the number of the current program line. It can be used in conjunction with the commands GOTOX and RESTOREX.

In CALC mode the function can be used to determine the line number of a stopped program. The function returns zero if the computer is not in a RUN or PAUSE state.

GOTOX LineNumber

The function performs a GOTO to the given line number. The line number can be specified either in form of a constant or by a variable. In contrast to this, the well-known GOTO statement can only handle constant line numbers, which have been defined at programming time.

Example: Comparison of ON Number GOTO and GOTOX Number:

First: using ON Number GOTO

```
1000 ON A GOTO 1010,1040, 1070
1010 ! first routine
1020 !
1030 ! ----------------------
1040 ! second routine
1050 !
1060 ! ----------------------
1070 ! third routine
1080 !
1090 ! ----------------------
```

Next: with GOTOX

```
1000 GOTOX LINE?+A*30-20
1010 ! first routine
1020 !
1030 ! ----------------------
1040 ! second routine
1050 !
1060 ! ----------------------
1070 ! third routine
1080 !
1090 ! ----------------------
```

The application of GOTOX not only reduces the programming (and counting) effort, but also the memory requirements as each target line number in an ON … GOTO statement requires 4 bytes of memory.

When using GOTOX one has to pay attention that the line numbers are not modified by a REN or RENUM command. The example program can be renumbered as long as the line number increment is 10 and the subroutines are spaced 30 lines apart.

If the given line number does not exist, GOTOX continues execution at the line following the missing line.

RESTOREX LineNumber

The function places the data pointer to the given program line. As an enhancement of the standard RESTORE function RESTOREX accepts constants as well as variables.

Together with the LINE? Function it is possible to read selected elements from large DATA statements. The normal RESTORE function does not allow to position the data pointer dynamically like RESTOREX does.

Example: reading selected parts of a DATA statement using RESTORE

```
500 ! the variable A contains the relative position of the
510 ! desired row in the DATA block
520 RESTORE 600
```

```
530 FOR I=2 TO A
540 READ A$
550 NEXT I
560 ! the row of the DATA statement is now ready to be read
570 ! by the next READ statement
580 !
600 DATA Row#1-Item#1, Row#1-Item#2, Row#1-Item#3, Row#1-Item#4
620 DATA Row#2-Item#1, Row#2-Item#2, Row#2-Item#3
630 DATA Row#3-Item#1
640 DATA Row#4-Item#1, Row#4-Item#2, Row#4-Item#3, Row#4-Item#4
```

Using RESTOREX to select the desired row with a single function call

```
500 ! the variable A contains the relative position of the
510 ! desired row in the DATA block
520 RESTOREX 600+(A-1)*10
560 ! the row of the DATA statement is now ready to be read
570 ! by the next READ statement
580 !
600 DATA Row#1-Item#1, Row#1-Item#2, Row#1-Item#3, Row#1-Item#4
620 DATA Row#2-Item#1, Row#2-Item#2, Row#2-Item#3
630 DATA Row#3-Item#1
640 DATA Row#4-Item#1, Row#4-Item#2, Row#4-Item#3, Row#4-Item#4
```

The application of RESTOREX is very easy. The function is especially useful where large DATA statements of differing size are defined. For example to access a sequence of DATA statements consisting of a different number of characters per line can easily be accessed. In this application, the first element of a DATA statement would contain the number of codes to follow. Such an application could be used for example to select special sequences of printer control characters.

Another option to recognize the end of a DATA block of variable length is to raise an error condition. For example when reading numeric data, a character could be placed in at the end of the DATA block. When the READ statement reaches this character, an ERROR is thrown which can be caught by an ON ERROR handler.

If the DATA line does not exist but a DATA statement follows at some higher numbered program line RESTOREX moves the READ pointer to this statement.

Searching for Labels in BASIC Programs

The Advanced BASIC of the HP-86/87 allows to use labels as targets for the GOTO and GOSUB keywords. For example the line

```
1000 GOSUB Printing
```

Is much easier to understand then the line

```
1000 GOSUB 5760
```

On the other hand it can be rather cumbersome and time consuming to locate line labels inside larger programs. Even with the commands SCAN (Advanced Programming ROM) or FREFS (Assembler ROM) it can take up to two minutes to find a label. The following new commands reduce the search time drastically.

BFLABEL String$

This keyword searches for the given label String$. The string is given without the trailoing colon.

```
BFLABEL "Printing"
```

If the label exists in the file, the corresponding line is listed on the screen and the LIST pointer is set to the line number. If the label is not found, nothing is displayed. In both cases a final message READY is displayed.

Even in large programs BFLABEL finds a label within less than one second.

LIST LABELS

This keyword lists all labels immediately following a line number in the current program. The LIST pointer is set to the line which contains the last label. The process can be stopped by pressing any key.

Converting Strings for Printers

Many printers not manufactured by HP cannot print the special non-ASCII characters used in the HP Series-80 computers. A typical case are the German Umlauts. The following commands allow to define and use suitable conversion tables.

The application of these commands is not limited to conversions for printed output. They can be used where characters inside strings are to be replaced against other characters.

SET REPLACE$S String1$, String2$

This keyword defines which characters are replaced by the command RPL$. The first string contains the characters which are to be replaced. Each character at a position I in the first string is replaced by the character at the same position in the second string. Both strings may contain up to 20 characters and must have the same length.

During the initialization of the ROM a standard translation table is established. This table has been set up to convert HP specific characters to EPSON specific characters.

HP Code	Character	EPSON
5	ß	126
21	Ä	91
22	ä	123
23	Ö	92
24	ö	124
25	Ü	93
26	ü	125

REPLACE$S? String1$, String2$

This statement returns the strings defined by a SET REPLACE$S command. The two receiving strings must be defined to accept at least 20 characters. The statement is useful if several different replacement strings are needed during a program run. It allows to retrieve the current replacements strings so that they can be restored later by a SET REPLACE$S command.

RPL$ (String$)

This function replaces the characters in String$ according to the table defined by a preceding SET REPLACE$S command. If the function is added to each PRINT statement it is easy to adapt a program to different printers.

```
PRINT RPL$("Zeichenkette mit Umlauten wie ÄÖÜäöü")
```

Sorting of Strings

SORT String$, Length, From, To

This statement sorts the characters in a string or string array according to the given parameters.

Length The length of the substring to be sorted

From The numeric variable specifies the relative position of the first character to be sorted inside the substring. A “1” selects the first character.

To The numeric variable specifies the relative position of the last character to be sorted inside the substring. Naturally it must be equal or larger than the parameter From. It cannot be larger than Length.

Example: Sorting a list of names using various criteria

```
DIM A$[100]
A$[1,25]  ="#001#Dave       Packard"
A$[26,50] ="#002#Snoopy     Peanut"
A$[51,75] ="#003#Frank      Sinatra"
A$[76,100]="#004#Robert     Redford"
```

The string A$ contains four names with an ID number. Each substring has a length of 25 characters.

In order to sort the list by first name we can use the following command:

```
SORT A$,25,6,16
FOR I=1 TO 75 STEP 25 @ DISP A$[I,I+24] @ NEXT I
```

Output:

```
#001#Dave       Packard
#003#Frank      Sinatra
#004#Robert     Redford
#002#Snoopy     Peanut
```

Sorting by last name:

```
SORT A$,25,17,25
FOR I=1 TO 75 STEP 25 @ DISP A$[I,I+24] @ NEXT I
```

Output:

```
#001#Dave       Packard
#002#Snoopy     Peanut
#004#Robert     Redford
#003#Frank      Sinatra
```

Reverting back to serial order:

```
SORT A$,25,1,6
FOR I=1 TO 75 STEP 25 @ DISP A$[I,I+24] @ NEXT I
```

Output:

```
#001#Dave       Packard
```

```
#002#Snoopy     Peanut
#003#Frank      Sinatra
#004#Robert     Redford
```

As the last application demonstrated, the command can also be used to sort strings according to a numeric key because of the sequential encoding of ASCII codes for digits.

UPSORT String$, Length, From, To

The ASCII sequence separates upper and lower case characters. Thus a SORT of the characters in "aBbA" would return "ABab".

The command UPSORT can be used to sort strings consisting of upper and lower case characters, independent of upper or lower case. It would sort "aBbA" into "AaBb"

The parameters are the same as used for the SORT command.

4 Advanced BASIC Structured Programming Constructs

HP-BASIC already comes with a number of programming elements like GOTO, GOSUB, ON GOTO, ON GOSUB, ON ERROR, FOR … NEXT. Nevertheless you may find situations where these elements seem to lack elegance and structure.

The following Structured Programming Extensions add new constructs to the BASIC language.

Loops

WHILE LogicalExpression DO
WHILE NumericExpression DO

This keyword defines the beginning of a loop which is executed while the given expression is true. This command must be the first in a program line and may be followed by a comment initiated by a "!" or "REM". In order to keep the source code clearly structured you cannot append other commands with a "@" separator.

END WHILE

The loop must be terminated by a line with a matching END WHILE keyword.

You can nest up to 15 WHILE … DO … END WHILE loops. The loop should not be left by a GOTO statement because this would not clean up the nesting stack and could finally lead to a "WHILE Nesting Error".

EXIT WHILE

If the program must leave the loop early, the EXIT WHILE keyword should be used.

POP WHILE

This keyword drops the currently active WHILE … DO level. The loop can then be left with a simple GOTO statement.

REPEAT

The REPEAT starts a loop which extends to a following UNITL statement. The loop is executed at least once, because the terminal condition is tested at its end. This is in contrast to the WHILE … DO loop, which not executed at all if the condition is false.

UNITIL LogicalExpression
UNITIL NumericalExpression

The UNTIL statement terminates a REPEAT loop.

The REPEAT … UNTIL loop will be repeated until the expression is true. You can nest up to 15 REPEAT … UNTIL loops. If you try to nest more loops, an "REPEAT NESTING"-Error is raised.

Example: splitting a number into digits

```
10 DISP "Enter a number";
10 INPUT Number
30 REPEAT                  ! ------+
40 DISP Number MOD 10;     !       |
50 Number=Number DIV 10    !       |
60 UNTIL NOT Number        ! ------+
70 DISP
80 END
```

Example: determination of the smallest denominator of two numbers

```
10 DISP "Enter the two numbers";
20 INPUT X,Y
30 A=X @ B=Y
40 REPEAT             ! -------+
50 WHILE X>Y DO       ! ----+  |
60 X=X-Y              !     |  |
70 END WHILE          ! ----+  |
80 WHILE Y>X DO       ! ----+  |
90 Y=Y-X              !     |  |
100 END WHILE         ! ----+  |
110 UNTIL X=Y         ! -------+
120 DISP "smallest denomninator=";A DIV X*B DIV X*X
130 END
```

POP UNTIL

This keyword drops the currently active REPEAT … UNTIL level. The loop can then be left with a simple GOTO statement.

LOOP

This statement defines an endless loop which is closed by an END LOOP statement.

END LOOP

The statements between LOOP and END LOOP will be repeated until the loop is left with a GOTO statement.

Note that you cannot nest several LOOPs. The LOOP construct can be used where fast infinite loops are needed.

An infinite loop without a LOOP structure typically looks like this

```
100 FOR I=1 TO INF
110 !
120 ! some action
130 !
```

```
140 NEXT I
```

When the LOOP structure is used, the program looks like

```
100 LOOP
110 !
120 ! some action
130 !
140 END LOOP
```

The advantage of LOOP is that the jump from END LOOP to LOOP is executed about twice as fast as the jump from NEXT to FOR. This is a great advantage when a high number of loop cycles are needed.

Structures for Comparison

WHILE NumericExpression DO

The HP BASIC language provides the IF … THEN … ELSE construct. Sometimes it is impossible to fit the expression following the IF or the ELSE keywords into a single line. The following statements circumvent this problem.

BLIF LogicalExpression BLTHEN

BLIF NumericalExpression BLTHEN

BLIF (Block If) defines the start of a block of program lines. It must be placed at the start of a program line and after BLTHEN no following statement is allowed on the same line.

The program lines following BLTHEN are executed if the expression is true or not equal to zero. The BLELSE or END BLIF statement defines the end of the block.

BLELSE

BLELSE is an optional alternate branch of the BLIF statement. It must be the first statement on a program line and no statements may follow on the same line. The program lines following BLELSE will be executed when the expression between BLIF and BLTHEN is false or zero.

END BLIF

The END BLIF must be the first statement of a line. It defines the end of a BLIF block.

You cannot nest BLIF … BLTHEN structures. The following structures are allowed inside a BLIF … BLTHEN block:

```
FOR … NEXT
WHILE … DO … END WHILE
REPEAT … UNTIL …
IF … THEN … ELSE
LOOP … END LOOP
```

EXIT BLIF

This statement can be used to leave a BLIF … BLTHEN structure. The execution continues with the line following the END BLIF statement. EXIT BLIF cannot be used outside of a BLIF … BLTHEN block.

Example: Application of the BLIF … BLTHEN construct

```
100 BLIF FirstPass BLTHEN
110 Flag=0
120 DISP "Enter command>";
130 INPUT A$
140 IF UPC$(A$)="DONE" THEN EXIT BLIF
150 GOSUB SyntaxCheck
160 BLELSE
170 IF NOT Flag THEN EXIT BLIF
180 GOSUB MainTest
190 END BLIF
```

5 Support Commands for Assembler Programming

The statements and functions presented in this section have are provided for users with some experience of the Series-80 assembly language and hardware. This manual does not explain how to program in assembler, please refer to the Assembler-ROM manual.

Reading Memory

PEEK (Address)

This function returns the content of the memory location specified by Address. The address must be specified in octal and must be in the range between 0 and 177777_o (65535_d). The return value is a single byte and therefore between 0 and 377_o. The function is useful to work with pointers to single bytes.

Example: Determine whether the display is in ALPHA or GRAPHICS mode. For this purpose the byte at address CRTSTS can be tested and masked:

```
NUM(AND$(CHR$(O_D(PEEK(177702))),CHR$(128)))
```

If the result is zero, the ALPHA mode is active, otherwise it will be 128.

Example: Determine whether the PAGESIZE is 16 or 24. This information is also contained in the byte at address CRTSTS:

```
NUM(AND$(CHR$(O_D(PEEK(177702))),CHR$(8)))
```

If the result is zero, PAGESIZE is 16, otherwise it will be 8 and PAGESIZE would be 24. This test for PAGESIZE is a very elegant method to determine within an Autost program whether the program is running on an HP-86A/87/87XM or on a HP-86B. Only the HP-86B boots with PAGESIZE set to 24.

Example: Determine which angle units are active:

```
PEEK(100160)
```

When RAD is active the result will be 0, in DEG mode it will be 220 and in GRAD mode 231.

PEEK$ (Address, ROM#, Count)

This function reads several bytes of memory starting at the given address. The result is returned as a string with length Count. The parameter ROM# is only relevant for addresses within 60000_o (24K) and 77777_o (24K+8K-1). These addresses cover the window into which ROMs are mapped.

Example: Calculate the length of the current BASIC program. We simply calculate the difference between the addresses BOVAR (end of program) and FWCURR (start of program):

```
ADR(PEEK$(100006,0,3))-ADR(PEEK$(100014,0,3))
```

Example: Determine then address of the current default MSUS:

```
MS$=PEEK$(103500,0,3)
MS$=":D"&VAL$(NUM(MS$)+3)&VAL$(NUM(MS$[2]))
MS$=MS$&VAL$(NUM(MS$[3]))
```

Example: Find the volume label of the active MSUS:

```
PEEK$(103527,0,6)
```

EMC PEEK$ (Address, Count)

This function reads Count bytes of memory from extended memory which uses 3 address bytes. These addresses are above 177777_o. Here the BASIC programs and variables are located.

SADR (String$)

This function returns the starting address of the given string in decimal form. The programmer can then directly access the string.

Modifying Memory

POKE Address, Byte

Copies the given Byte to the memory location specified by two-byte address (given in octal). The address must be between 100000_o and 177777_o. The parameter Byte can be within 0 and 377_o.

Example: Switching the video mode from inverse to normal:

```
POKE 177702,D_O(NUM(XOR$(PEEK$(177702,0,1)," ")))
```

Example: Toggle blinking RUN LED:

```
POKE 177702,1  ! LED blinks
POKE 177702,0  ! LED continuously on
```

Example: Changing the key repeat delay:

```
POKE 100154,N   ! N is the delay parameter, its default value is 34
```

Example: Changing the key repeat rate:

```
POKE 100155,N   ! N is the speed parameter, its default value is 2
```

EMC POKE$ Address, String$

Copies the bytes in the given String$ to the two- or three-byte memory location specified by the parameter Address (given in octal). The address must be between 100000_o and 2000000_o.

This function can be applied for example to create self-modifying code. It can also be used to directly access files on the EDISC or to manipulate strings directly. The main advantage is the increased speed – the risk is that no sanity checks are performed like most standard BASIC string functions do.

SETPTR2 Address, Value

Sets the value of an address pointed to by a two-byte pointer. Address and Value must be given in octal, where Address may be between 100000_o and 177777_o and Value within the range 0 to 177777_o. A typical two-byte pointer is CRTBAD, a CRT control register.

Example: Setting the cursor to row 10 and column 11:

```
SETPTR2 177701,D_O ((10-1)*80+11-1) ! 177701 = CRTBAD
```

A following execution of AREAD would then read from the CRT buffer from this position. When pressing a character key after setting the cursor with SETPTR2 like shown above it looks like the cursor would not have moved. This is because the operating system maintains a copy of the cursor position and uses this address. Therefore setting the cursor using SETPTR2 is only useful in RUN mode when a command like AREAD follows.

SETPTR3 Address, Value

Sets the value of an address pointed to by a three-byte pointer. Address and Value must be given in octal, where Address may be between 100000_o and 177777_o while Value can now be within the extended range (compared to SETPTR2) from 0 to 77777777_o. Typical three-byte pointers are the extended memory control pointers PTR1 and PTR2. Here we focus on PTR1. Storing a value via PTR1 makes the BASIC program pointer branch to the address given.

Example: a program shall branch to Address 300000, we can use the following statement:

```
SETPTR3 177710,300000 ! 177710 = PTR1
```

This provides the user with the capability to let a program jump to wherever location, even into the middle of a multi-statement line. It is also possible to write self-modifying programs by storing token sequences in strings which are then executed with SETPTR2. Of special interest are tokens which normally do not occur as single tokens in a regular program. For example the second token of the INPUT statement, some of the mass storage statements, READ# or PRINT#, as well as invisible secondary tokens contained in some ROMs. Testing these options is a wide field.

Encoding Numbers

The Series-80 BASIC uses three number representations: Integer, Short and Real. The Integer and Short numbers require less memory and hence less disc storage. This can be important when operating on large data arrays.

In machine language programs it is often necessary to load constants using a specific representation into registers before calling system routines. Preparing such machine language representations can be cumbersome. The following function solves this problem.

REGREP (Number)

This function returns a string which contains the given Number in octal form as needed for assembly programming.

Example: how to enter the number INF (which is 9.999999999999E4)?

```
REGREP (INF)
231 231 231 231 231 231 100 231
```

This result can be used to write the assembler instruction to load INF into registers R40-R47:

```
LDM R40,=231,100,231,231,231,231,231,231
```

You will notice that the string produced by REGREP has to be loaded in reversed order.

Sometime system routines require constants in a certain representation. The REGREP function recognizes whether its argument is a Real or an Integer and returns the corresponding representation.

Example: loading the Integer number 1000 into CPU registers R50-R57 and loading the Real number 1000.0 into R60-R67:

```
REGREP (1000)                     ! Integer number representation is
000 020 000 377 000 000 316 016   ! marked by the byte 377. The following
                                  ! bytes are undefined and not needed
LDM R54,=377,0,20,0               ! just load the required bytes and leave
                                  ! R50-R53 as is
REGREP (1000.0)                   ! Real number
020 000 000 000 000 000 000 003
LDM R60,=3,0,0,0,0,0,0,20
```

This more elaborate example shows how to calculate SIN(45)*15.89 while in assembler mode:

```
REGREP (45)                       ! Integer number
000 000 105 377 000 000 177 246
1000  LDM R44,=377,105,0,0         ! load Integer to R44-R47
1010  PUMD R40,+R12               ! push to operand stack
1020  JSB =SIN10                  ! calculate sine
REGREP (15.89)                    ! Real number indicated by decimal char.
025 211 000 000 000 000 000 001
1030  LDM R40,=1,0,0,0,0,0,211,25 ! load Real to R40-R47
1040  PUMD R40,+R12               ! push to operand stack
1050  JSB =MPYROI                 ! multiply
1060  POMD R40,-R12               ! pop result to R40-R47
```

REGREP$ (Number)

This function encodes a number into a string of 8 bytes. REGREP$ is used where SCALL() is used to call string buffers containing machine language programs. REGREP$ can be used to encode the numbers required in the string of machine language code.

Example: storage of the machine code for loading the decimal number 1.256E7 into registers R40-47: (LDM R40=1.256E7).

```
A$=CHR$ (96)                  ! opcode for set DRP 40 (96d = 140o)
A$=A$&CHR$ (169)              ! opcode for LDM
A$=A$&REV$ (REGREP$ (1.256E7)) ! append the number converted to octal
```

REGREP (String)

This function converts an encoded 8-byte string into a number. It can be used with PEEK$ when machine code has to be translated back.

Example: disassemble the number PI (which is stored as an 8-byte real at address 54376):

```
REGREP (REV$ (PEEK$ (54376,0,8)))
3.14159265359
```

Calling System Functions and Processing BPGM Strings

The operating system of the HP 86/87 computer contains many functions which can be useful for the Basic programmer. Unfortunately, the system provides no means to access those functions from BASIC. For example the functions which are executed by pressing the keys like “CURSOR LEFT”, “ROLL DN”, “ROLL UP” etc. (except with replacement functions contained in some extension ROMs).

The statement which is described here allows calling system functions.

SCALL Address [, ROM#]

SCALL String

The first variant (SCALL Address [,ROM#]) can be used to execute a routine in a system ROM or one of the option ROMs within the Address range of 0 to 77777.

The following table lists some interesting system functions and their address

SCALL	ROM#	Effect
11606	0	[A/G]
11520	0	[BACK SPACE]
11565	0	[SHIFT][BACK SPACE]
13671	0	[ROLL DN]
13736	0	[ROLL UP]
13651	0	[RIGHT]
13623	0	[LEFT]
13607	0	[DOWN]
13562	0	[UP]
13661	0	[HOME]
13447	0	[-LINE]
14225	0	[CLEAR]
14165	0	deletes the complete row
5407	0	[RESET]
1241	0	[INIT]
0	0	power on/off reset
12246	0	clear all screen pages
14030	0	cursor ON
13467	0	cursor OFF
12360	0	CRT ON
12374	0	CRT OFF
5601	0	SCRATCH
1074	0	[RUN] restart program

More can be found in Chapter 8 of the HP 86/87 assembler manual.

You can also store machine code using EMC POKE$ at address 101145 (usually used for key labels) and then call this address using SCALL.

The second variant (SCALL String) executes the machine language program contained in String.

Here one has to be reminded that the bytes must be store in reverse order, i.e. the program has to start at the end of the string. Another important constraint is that the string must be located in memory below 177777, i.e. it must be in low memory so that it can be address by two-byte pointers. SCALL checks the address and issues an error message if the string is in extended memory.

Parameters are best passed from Basic to a machine language program in the following manner:

Strings

The SADR function returns the position of the string. This position can be converted into a 3-byte address using ADR$ and then stored using EMC POKE$ at address 101145. Next, LEN returns the length of the string. This length is also converted into a string which is then written to address 101150 with EMC POKE$. The machine language program later reads the string and its length from these two memory locations.

The given addresses only serve as an example. They are used here because the key labels which are normally stored there are not essential for the operation of the computer and can be restored after execution of the machine language program.

Numbers

The REGREP$ function returns an 8-byte string with the internal number representation. This string can be stored with EMC POKE$ at address 101145. The machine language program later reads the number from these this memory location. The program can also return a value by writing to this address so that the Basic program can read it using PEEK$ and then convert it to a number with REGREP.

Catalog Functions for Binary Programs and ROMs

The large number of functions and keywords available in the Series-80 systems is impressive, but often difficult to manage. But even the experienced user sometimes may forget about some keywords and then implements code which could have been solved more efficiently with an existing keyword or function provided by the operating system. This becomes more difficult, when the quick reference guides for ROMs are not available and the user is not sure whether a certain command exists and what the exact syntax might be. The following statements help to organize your system.

RCAT ROM#

This statement lists the entries found in the given ROM# (their token number, the addresses of runtime and parse routines as well as their name and the function attributes). Output is in octal.

The statement RCAT 208 (Mass Storage ROM) produces output starting with

```
ROM:    000320
Runtim: 060012
ASCIIS: 060215
Parse:  060130
Init:   073631
Ermsg:  073440
Tok# Runtim  Parse   Name            Attributes
-------------------------------------------------------
001  065466  061237  ASSIGN#                          241
002  061414  060542  CAT                              241
003  072474  060600  CHECK READ OFF#                  241
004  072433  060600  CHECK READ#                      241
...
```

The header of the catalogue lists the ROM ID in octal, followed by the start of the table containing the keywords, the runtime table and the parser routine table, the address of the initialization routine and finally the address of the error message table.
The following catalogue lists all keywords with their start addresses and attributes in increasing token order.
The interpretation of the attribute bytes can be found in chapter 7 of the Assembler ROM manual.

BCAT BPGM#

BCAT BaseAddressOfBPGM

Called with the binary program number or its base address (decimal) this statement lists the entries found in this binary program (their token number, the addresses of runtime and parse routines as well as their name and the function attributes). Output is in octal.

To find the base address of a binary program you can use the following commands (REL is a function provided by the Assembler ROM):

```
LOADBIN „BinaryProgram“
REL (0)
122470
BCAT O_D (122470)
```

The output of the BCAT statement is identical to the output generated by RCAT, except that the addresses are in RAM, i.e. above 100000.

Appendix A – Disk Editor

This appendix lists a BASIC program of an editor for LIF disks.

Appendix B – Tables

Appendix C – CPU Instruction Set

Appendix D – BPGM and ROM Numbers

ROM IDs

ROM#	Name
0	System
1	System
14	Miksam
14	Lang
40	Assembler
56	Sysext
176	Matrix 1
177	Matrix 2
192	I/O
207	Extended Mass Storage
208	Mass Storage
209	Electronic Disc
231	Advanced Programming 1
232	Advanced Programming 2
240	Printer/Plotter

BPGM IDs

BPGM#	Name
12	SPKB87
13	BIN15
14	STEVIE
16	FORM
18	IPBINg
19	KEYONBg
20	BIN24
22	UTIL/1

23	LINCURg
26	REDZERg
33	GETSAVEg
34	ASKIOB
37	TRACKBg
39	FILE/80BIN
41	TDGRAPH
44	GCURSBg
45	STRNGBg

46	MATHBIg
47	LIFg
48	BINCALC
56	SYSEXT
72	GDUMP
73	SORTBg
74	FORMSBg
129	TEXTBIN

Appendix E – Index

Appendix F – Error Messages

The SYSEXT ROM outputs additional error messages beyond the standard error message numbers.

In case of an error, the function ERROM returns the ROM ID 56 and the error number can be requested by the function ERRN.

109	No Numeral	The function REGREP cannot convert the given string to a number.
110	Address > 16 bit	You tried to execute SCALL with a string which was located at an address above the 16 bit range.
111	-	Not used.
112	WHILE nesting	You tried to nest more than 15 levels.
113	Missing END WHILE	A WHILE level was not terminated by an END WHILE statement.
114	Missing WHILE DO	One END WHILE too many.
115	No active WHILE	EXIT or POP WHILE without an active WHILE level.
116	No active LOOP	END LOOP without LOOP
117	Incorrect syntax	Execution of TOKEN$ or EXECUTE with an incorrect expression.
118	No active REPEAT	POP UNTIL or REPAET … without active level.
119	REPEAT nesting.	You tried to nest more than 15 levels.
120	String len#1 != len#2	You tried to execute SET REPLACE, AND$, OR$ or XOR$ with strings having different lengths.
121	-	Not used.
122	Missing END BLF	There is no matching END BLIF statement for the current BLIF statement.
123	Missing BLIF	You tried to execute a BLELSE or EXIT BLIF statement without active BLIV level.
124	BLIF nesting	You tried to nest several BLIF blocks.
125	Element length = 0	A substring length given to SORT or UPSORT is zero.
126	Element length > string length	A substring length given to SORT or UPSORT is larger than the total string length.
127	First column > last column	The “to” variable given to SORT or UPSORT statements is larger than the substring length.
128	First column > last column	The “from” variable given to SORT or UPSORT statements is larger than the “to” variable.
129	First column = 0	The “to” variable given to SORT or UPSORT statements is zero.

Appendix X – Undocumented Statements

The ROM contains a few additional functions which are not mentioned in the original documentation. Thanks to Everett Kaser for finding and describing them.

SYSEXT

Outputs a revision/copyright string for this ROM.

```
SYSEXT
(c) Andre Koppel Software ver 30.87
```

CWHILE

This statement clears the WHILE/DO stack (removes ALL entries). See POP WHILE, which removes ONE entry.

CUNTIL

This statement clears the REPEAT/UNTIL stack (removes ALL entries). See POP UNTIL, while removes ONE entry.

EBEEP
RBEEP

These statements perform specific BEEPs: The "error beep" EBEEP executes sequence of BEEP 100, 90 followed by BEEP 200, 40. The statement RBEEP does a brief BEEP 45, 20.

EUL

Returns $e \times 10^{11}$ the largest power of 10 times the natural logarithm constant '*e*'. The largest number that can be represented in the Series 80's 12 significant digit mantissa: 271828182846.

TRUE
FALSE

TRUE returns 1 and FALSE returns 0.

ONE
ZERO

As you would expect, ONE returns 1 and ZERO returns 0.

SUBLEVEL

This statement returns the number of GOSUB RETURN addresses on the GOSUB/RETURN stack. i.e., how deep we are in nested GOSUBs.

N? (Number1, Number2, Number 3)

This function works like the "C" language construct *(Number1 ? Number2 : Number3)*. If *Number1* is non-zero, then *Number2* is returned as the function value, otherwise *Number3* is returned.

```
N?(1,100,200)
```

```
100
N?(0,100,200)
200
```

S? (Number, String1, String2)

This function works like N?, except this is a string function which returns either *String1* or *String2*, depending upon the value of *Number*.

```
S?(1,"AAA","BBB")
AAA
```

SUCC (Number)

PRED (Number)

The SUCC function increments the given *Number* and returns *Number+1*. This works like the same function in the Pascal language. The PRED function decrements the given *Number* and returns *Number-1*. This works like the same function in the Pascal language.

```
SUCC(2)
3
PRED(3)
2
```

Contents