Binary Program and ROM “UTIL85B”

Martin Hepperle,
June 2017 – November 2021

Table of Contents

1. Summary

The binary program “UTIL85B” extends the selection of utility and mathematical functions for the HP 85 computer. It can be loaded using the command

```
LOADBIN “UTIL85B”
```

Afterwards the new functions are available for BASIC programs as well as for execution on the command prompt.

The program requires almost 8 Kbytes of RAM (version of November 2021). The ultimate goal, to convert the source code into a ROMable version for the EPROM drawer or the PRM-85 module, was achieved in the summer of 2020. The last functions were added in November 2021.

If you want to modify the program you can load and compile the assembler source code using the usual commands provided by the Assembler ROM. Due to size the source code had been split into four segments, “UTIL85_1S” to “UTIL_4S”.

```
ASSEMBLER
ALOAD “UTIL_1S”
ASSEMBLE “UTIL85B”
```

In order to speed up the development cycles for the ROM version, the segmented source code was combined into a single file “MATHROM.ASM” because this work was performed with Everett Kaser’s ASM85 cross-assembler on a PC. I also have to thank Everett for performing the conversion for HP-86/87 systems with their Extended Memory Controller (EMC). During this work, he found a few bugs and improved the code in several places.

It should be possible to convert this code back to segments which can be assembled on the real hardware. A few ASM85-specific directives, like “HED” and the bit-inversion in the ROM# must be removed resp. replaced, though.

For assembling and copying to the HP-85 emulator I used the simple COMMAND script listed below.

Note that for testing I used the ROM number of the FORTH ROM because the emulator did not allow adding my own ROM number through its GUI. For “production” the ROM number must be changed as already prepared in the source code.

```
@echo OFF
set SRC=MATHROM.ASM
set DST=rom250
set DIR=..\roms85

echo Assembling %SRC%...
asm85 -85 %SRC% -o %DST% 1> oops
rem echo %ERRORLEVEL%

rem In CMD, scripting is ugly, but possible
rem Get line with "... errors" to X
findstr errors oops > oo
set /P X= < oo
type oops
rem Strip off "errors" from X so that only the error count remains
echo %X:errors=% > oo
set /P X= < oo
del oo
del oops


if %X% LEQ 0 (
   rem No errors: copy image to destination
   echo.
   echo Copying ROM image %DST% to %DIR%
   copy %DST% %DIR% > NUL
) else (
   rem errors: do nothing
   echo.
   echo Oops: errors occurred
```

).

2. Utility Functions

MATHREV

This function returns the revision number of the ROM respectively Binary program.

CREAL (M)

This function converts the number *M* into a Real number. If *M* is already Real, nothing is changed, if *M* is an Integer or Short it is converted to Real.

The Series-80 computers BASIC system allows you to define variables as `REAL`. However, the system may store floating point numbers as well as integer numbers in these Real variables. For bookkeeping the integer numbers are "tagged" and are therefore denoted "Tagged Integers". Looking at the tag, the system can always find out whether a value is a Real or a Tagged Integer – but at the cost of speed.

Sometimes it is necessary to make sure that a number (even if its value is accidentally integer) is stored exactly as a Real and not as a Tagged Integer. The function `CREAL` can be used to perform this conversion.

Example

```
10 REAL X      ! Declare X as a Real
20 X=1         ! Stores the Tagged Integer representation of 1 in X
30 X=1.0       ! Stores the Real representation of 1 in X, however a
40 !           ! subsequent LIST displays 30 X=1 and another [END LINE]
50 !           ! stores 1 in form of a Tagged Integer in X
40 X=CREAL(1) ! Same result as X=1.0, but permanent
```

RADIUS (X, Y)

This function returns the radius of a circle through the origin and the point $[X, Y]$, which is $\sqrt{X^2 + Y^2}$. Similar to the `POLAR` keyword but implemented as a function returning the result.

Example:

```
RADIUS(10,2)
10.1980390272   ! the length of the radius vector
ATN2(10,2)      ! (Y,X)
1.37340076695   ! the angle of the vector in RADIAN
```

POLAR X, Y, R, T

This keyword converts a point given in Cartesian coordinates *X* and *Y* to Polar coordinates *R*, *T* (radius, theta). While *X* and *Y* can be given as variables or constant numbers *R*, *T* must be variables to take the returned values. Complements the `RECTANGULAR` keyword.

RECTANGULAR R, T, X, Y

This keyword converts a point given in Polar coordinates *R*, *T* (radius, theta) to Cartesian coordinates *X* and *Y*. While *R* and *T* can be given as variables or constant numbers *X*, *Y* must be variables to take the returned values. Complements the `POLAR` keyword.

Examples

```
DEG                  ! we want use degrees
POLAR 1.0,1.0,R,T    ! from rectangular to polar
DISP R;T
 1.51521356237   45
RECTANGULAR R,T,X,Y  ! and back
DISP X;Y             ! limited accuracy due to SIN, COS, SQR functions
 .999999999998   .999999999998
```

BAND (M, N)

Returns the *AND* combination of the bits in *M* and *N*. *M* and *N* are considered unsigned 16-bit integers in the range [0…65535]. This function is similar to the function `BINAND` in the I/O ROM which handles only signed integers [–32768…32767].

BIOR (M, N)

Returns the *INCLUSIVE OR* combination of the bits in *M* and *N*. *M* and *N* are limited to unsigned 16-bit integers in the range [0…65535]. This function is similar to the function `BINIOR` in the I/O ROM which handles only signed integers [–32768…32767].

BEOR (M, N)

Returns the *EXCLUSIVE OR* combination of the bits in *M* and *N*. *M* and *N* are limited to unsigned 16-bit integers in the range [0…65535]. This function is similar to the function `BINEOR` in the I/O ROM which handles only signed integers (–32768…32767).

BSHIFT (M, N)

Returns the number obtained from shifting the bits in *M* by *N*. If *N* is positive the bits are shifted to the left, a negative value shifts to the right. The number *M* is considered an unsigned 16-bit number, i.e. it has a range of [0…65535]. *N* must be a 16-bit integer and should not exceed ±16. If the shift operation moves bits out of the 16-bit window they are lost, i.e. carry bits are dropped.

Examples

```
BIOR(65534,1)     ! function from I/O ROM produces unexpected result
32767
BIOR(65534,1)     ! produces the expected result
65535
BAND(9,5)         ! 9 = 1001b, 5 = 101b    9 AND 5 = 0001b
1
BIOR(9,5)         ! 9 = 1001b, 5 = 101b    9 OR 5  = 1101b
13
```

```
CHR$(BXOR(NUM("A"),32))      ! toggle the bit
a
CHR$(BXOR(NUM("a"),32))      ! toggle the bit
A
BSHIFT(32767,4)              ! 0x7FFF -> 0xFFF0 (high nibble lost)
65520
BSHIFT(BSHIFT(43981,8),-8)   ! 0xABCD -> 0x000C isolate nibble 1
12
```

EVEN? (M)

Returns *1* if the given integer *M* is even, *0* otherwise.

M must be an integer in the range [–99999…99999].

Example

Test whether the number 99991 is even:

```
IF EVEN?(99991) THEN DISP "EVEN" ELSE DISP "ODD"
ODD
```

ODD? (M)

Returns *1* if the given integer *M* is odd, *0* otherwise.

M must be an integer in the range [–99999…99999].

Example

Test whether the number 99991 is odd:

```
IF ODD?(99991) THEN DISP "ODD" ELSE DISP "WEIRD"
ODD
```

BITSET? (M,N)

Returns *1* if bit *N* in the integer *M* is set, *0* otherwise.

M must be an unsigned 16-bit integer in the range [0…65535], *N* in [0…15].

Example

Test whether the lowest bit 0 in the number 17 is set:

```
IF BITSET?(17,0) THEN DISP "ODD" ELSE DISP "EVEN"
ODD
```

MODDIV I, J, M, D

This keyword performs the division of *I* by *J* and returns the remainder *M* as well as the result of the division *D* in a single function call. The same result can be obtained by calling `MOD` and `DIV` with the parameters *I* and *J*.

I and J can be real or integer.

Examples

Dividing 13 by 7 yields 6 + 1·7. Dividing 2.5 by 0.2 gives 0.1 + 12·0.2

```
MODDIV 13,7,M,D
DISP M;D
6   1
MODDIV 2.5,0.2,M,D
DISP M;D
0.1   12
```

FRE

Returns the amount of free memory in bytes. This is the same value that is listed after executing a `PLIST` statement.

RNDI (M)

Returns a random integer number between zero and the given integer *M*. Complements the `RND` function which returns a floating point result in the range [0 ≤ X < 1].

M must be an integer in the range [-99999 ≤ M ≤ 99999].

Example

```
DISP RNDI(25);RNDI(25);RNDI(25);RNDI(25);RNDI(25);RNDI(25)
7     0     13     21     3     25
```

SWAP X, Y

Swaps the values stored in the variables *X* and *Y*. Useful when implementing sorting algorithms.

X and *Y* must both be either scalar or array elements - you cannot mix parameter types. The variable type can be either REAL, INTEGER or SHORT but must also be the same for both parameters.

Example

```
10 REAL A(10)
20 A(7)=7.7 @ A(2)=2.2
30 INTEGER X,Y
40 X=1 @ Y=2
50 PRINT X;Y;"->";
60 IF X<Y THEN SWAP X,Y
70 PRINT X,Y
80 PRINT A(7);A(2);"->";
90 IF A(7)>A(2) THEN SWAP A(7),A(2)
100 PRINT A(7);A(2)
110 END
RUN
 1    2   -> 2    1
 7.7  2.2 -> 2.2  7.7
```

EXECUTE S$

Executes the BASIC expressions stored in the variable *S$*. Useful for implementing plotting or integration programs with user supplied expressions.

S$ must contain a valid BASIC expression which will be parsed, allocated and interpreted. It must not alter program memory, i.e. a leading line number is forbidden. Typically it would be an expression of the form `Y=f(X)`.

All variables used in the expression must be pre-allocated in the main program by assigning a value to them. The `EXECUTE` keyword can only be used inside a program, not from the command line.

The execution is, of course, slower than the direct implementation because the expression has to be parsed, allocated and interpreted each time it is `EXECUTE`d. For a simple expression like `Y=SIN(X)` the speed factor is about 4-5.

Example

```
10 Y=0 ! Note: output variable MUST be pre-allocated
20 F$="Y=2^X"
30 FOR X=0 TO 7
40 EXECUTE F$
50 DISP X;Y
60 NEXT X
70 END
RUN
 0  1
 1  2
 2  4
 3  8
 4  16
 5  32
 6  64
 7  128
```

3. String Functions

LEFT$ (A$, M)

Returns the leftmost *M* characters of the given string *A$*.

If the length of the string is less than *M* the shorter string is returned. Equivalent to the BASIC construct `A$[1,M]`.Complements the `RIGHT$` and `MID$` functions.

RIGHT$ (A$, M)

Returns the rightmost *M* characters of the given string *A$*.

If the length of the string is less than *M* the shorter string is returned. Equivalent to the BASIC construct `A$[LEN(A$)-M,LEN(A$)]`. Complements the `LEFT$` and `MID$` functions.

MID$ (A$, M, N)

Returns a substring of the string *A$* having *N* characters starting at index *M*. The first character has the index 1.

If the starting index *M* is larger than the length of the string an empty string is returned. If the extent of the substring [M+N-1] exceeds the length of the string a shorter string is returned. Equivalent to the BASIC construct `A$[M,M+N-1]`. Complements the `LEFT$` and `RIGHT$` functions.

LTRIM$ (A$)

Trims leading blanks ("space" characters, ASCII 32) from the given string *A$* and returns the result.

If the string is composed of nothing but space characters an empty string of zero length is returned. Complements the `RTRIM$` and `TRIM$` functions.

RTRIM$ (A$)

Trims trailing blanks ("space" characters, ASCII 32) from the given string *A$* and returns the result.

If the string is composed of nothing but space characters an empty string of zero length is returned. Complements the `LTRIM$` and `TRIM$` functions.

TRIM$ (A$)

Trims leading and trailing blanks ("space" characters, ASCII 32) from the given string *A$* and returns the result.

If the string is composed of nothing but space characters an empty string of zero length is returned. Works like a sequence of calling the `LTRIM$` and the `RTRIM$` function.

Examples

```
DISP "<"&LEFT$("HELLO HP85",5)&">"
<HELLO>
DISP "<"&RIGHT$("HELLO HP85",4)&">"
<HP85>
DISP "<"&LTRIM$("   TEXT   ")&">"
<TEXT   >
DISP "<"&RTRIM$("   TEXT   ")&">"
<   TEXT>
DISP "<"&TRIM$("   TEXT   ")&">"
<TEXT>
```

FMT$ (A$, M)

Formats the given number *M* according to the format specified in *A$*. The format is identical to the `IMAGE` descriptors used for output by the `PRINT USING` or `DISP USING` statements. This function enhances the built-in `VAL$` function allowing full control over the formatting. This is very useful when collating formatted numbers for tabular output using `DISP ;` statements (`DISP USING` does not honor a trailing semicolon).

Two format specifiers cannot be used:

- the string field specifier "A" (only numeric data can be formatted) and,
- the repeat count (only a single number can be formatted).

If the number does not fit into the given format a “WARNING 2 : OVERFLOW” is raised.
If the image descriptor is incorrect “ERROR 52 : IMAGE” is raised.

Examples

```
FMT$("DDDD.DD",12.3456)
  12.35
FMT$("SDDD.DD",12.3456)
 +12.35
FMT$("ZZZZ.DD",12.3456)
0012.35
FMT$("****.DD",12.3456)
**12.35
FMT$("DDCDDCDD",123456)
12,34,56
FMT$("DDD.DE",12.3456)
123.5E-001
A$="DDD.DDD"
FMT$(A$,1)&FMT$(A$,-2)& FMT$(A$,3)
  1.000 -2.000  3.000
```

RPT$ (A$, M)

Returns the string created by concatenating *M* copies of the given string *A$*.

Example

```
RPT$("He",3)
HeHeHe
```

REV$ (A$)

Returns the string created by reversing the sequence of the characters in the given string *A$*.

Example

```
REV$("58-PH")
HP-85
```

LWC$ (A$)

Returns the string created by converting all characters in the given string *A$* to lower case. Characters are changed if they fall in the range [“A”…”Z”, “Ä”, “Ö”, “Ü”, “Å”, “Æ”]. Note that the complementary system function `UPC$` only handles the range [“A”…”Z”].

Example

```
S$="SCH"&CHR$(23)&"NE Augen"
S$
SCHÖNE Augen
LWC$(S$)
schöne augen
```

NOC (A$, B$)

Counts the number of occurrences of the string *B$* inside the string *A$*. No overlapping occurrences are considered. Instead of reinventing the wheel this function has been copied from the User's Library respectively the assembler sources provided by M. Cragg.

SAR$ (A$, B$, C$)

Searches the string *A$* and replaces all occurrences of the string *B$* by the string *C$*. No overlapping occurrences are considered. This function has been copied from the User's Library respectively the assembler sources provided by M. Cragg.

Examples

```
NOC("Sexy Hexy","ex")
2
SAR$("Sexy Hexy","ex","us")
Susy Husy
```

AND$ (A$, M)

Returns the *AND* combination of the bytes in *A$* and *M*. *A$* can have one or more characters and the bits of each of them are ANDed with the bits in *M*. *M* must be in [0…255].

IOR$ (A$, M)

Returns the *INCLUSIVE OR* combination of the bytes in *A$* and the number *M*. *A$* can have one or more characters and the bits of each of them are ORed with the bits in *M*. *M* must be in [0…255].

EOR$ (A$, M)

Returns the *EXCLUSIVE OR* combination of the bytes in *A$* and the number *M*. *A$* can have one or more characters and the bits of each of them are XORed with the bits in *M*. *M* must be in [0…255].

ADD$ (A$, M)

Returns the bytes in *A$* incremented by the number *M*. *A$* can have one or more characters and to each of them the number *M* is added. *M* must be in [–255…255].

Examples

```
"down"&IOR$("under",128)    ! set bit 7 to change to underlined
downunder
IOR$("ABC",32)              ! set bit 5 to change to lower case ASCII
abc
EOR$("AbC",32)              ! flip bit 5 to toggle case
aBc
AND$("5678",31)             ! leave only bits 0, 1, 2, 3, 4
ÄäÖö
ADD$("ABC",1)               ! adds 1 to each character code
BCD
```

```
ADD$("CDE",-2)              ! subtracts 2 from each character code
ABC
```

4. Base Conversion Functions

DTB8$ (M)

Returns a string of 8 characters representing the binary value of the given unsigned 8-bit integer number *M*. *M* must be in [0…255].

DTB16$ (M)

Returns a string of 16 characters representing the binary value of the given unsigned 16-bit integer number *M*. *M* must be in [0…65535].

DTH8$ (M)

Returns a string of 2 characters representing the hexadecimal value of the given unsigned 8-bit integer number *M*. *M* must be in [0…255].

DTH16$ (M)

Returns a string of 4 characters representing the hexadecimal value of the given unsigned 16-bit integer number *M*. *M* must be in [0…65535].

Examples

```
!
! decimal to binary conversion
DTB8$(255)&"     "&DTB8$(1)
11111111     00000001
DTB16$(65535)&"     "&DTB16$(1)
1111111111111111     0000000000000001
!
! decimal to hexadecimal conversion
DTH8$(255)&"     "&DTH8$(1)
FF     01
DTH16$(65535)&"     "&DTH16$(1)
FFFF     0001
```

STD8 (A$)

Returns the decimal value of the unsigned byte given in *A$*. Only the first character in *A$* is converted to a number in the range [0…255]. This function behaves like the `NUM` function of the BASIC system and is included here for completeness.

STD16 (A$)

Returns the decimal value of the first two bytes in *A$* forming an unsigned 16-bit integer number. According to the byte order of the Capricorn CPU the first byte is the least significant (LSB) and the second byte is the most significant (MSB). If the length of A$ is lower than two only the LSB byte will be converted. The result will be in the range [0…65535]

STB8$ (A$)

Returns a string of 8 characters representing the binary value of the byte given in *A$*. Only the first character in *A$* is converted.

STB16$ (A$)

Returns a string of 16 characters representing the binary value of the first two bytes in *A$* forming an unsigned 16-bit integer number. According to the byte order of the Capricorn CPU the first byte is the least significant (LSB) and the second byte is the most significant (MSB). If the length of A$ is lower than two only the LSB byte will be converted.

STH8$ (A$)

Returns a string of 2 characters representing the hexadecimal value of a byte given in *A$*. Only the first character in *A$* is converted.

STH16$ (A$)

Returns a string of 4 characters representing the hexadecimal value of the first two bytes in *A$* forming an unsigned 16-bit integer number. According to the byte order of the Capricorn CPU the first byte is the least significant (LSB) and the second byte is the most significant (MSB). If the length of A$ is lower than two only the LSB byte will be converted.

Examples

```
! --- string to hexadecimal conversion
STH16$("LH")                    ! 16-bit number is composed
484C                            ! of Low and High byte
DTH16$(NUM("H")*256+NUM("L"))   ! do it by hand
484C
STH8$("H")&STH8$("L")           ! or like this...
484C
! --- string to binary conversion
STB8$("L")
01001100
STB8$("LH")
0100100001001100
! --- string to decimal conversion
STD8("L")                       ! same as NUM("L")
76
STD16("LH")                     ! same as NUM("H")*256+NUM("L")
18508
```

STR2NUM (A$, M)

This function is similar to the `NUM` function of the BASIC system or the `STD8` function of this binary program. Instead of converting a single character it determines the decimal value (character code) of all characters in the given string *A$* and stores them in the given integer array *M*. The function returns the number of characters in *A$*. *M* must be declared as an `INTEGER` array at least as large as the length of *A$*. If the type of *M* is not `INTEGER`, you will receive an `Error 33: DATA TYPE`. To maximize speed no test for the array bounds is performed, so you have to make sure the receiving array is large enough.

Example

```
10 OPTION BASE 0          ! optional
20 INTEGER M(10)          ! M must be INTEGER, large enough to hold ...
30 I=STR2NUM("ABC",M)     ! ... the codes for all characters in A$
40 DISP I;M(0);M(1);M(2) ! first index depends on OPTION BASE
3   65   66   67
```

5. Memory Access

PEEKC$ (M, N)

Returns the character stored at memory address *M*. The second parameter *N* can be used to select a ROM with the given ID. The address *M* must be in [0…65535]. The parameter *N* is only relevant if the address *M* is within the ROM window (all addresses between 24576 and 32767).

PEEK8 (M, N)

Returns the byte stored at memory address *M*. The second parameter *N* can be used to select a ROM with the given ID. The address *M* must be in [0…65535]. The parameter *N* is only relevant if the address *M* is within the ROM window (all addresses between 24576 and 32767).

PEEK16 (M, N)

Returns the unsigned 16-bit word stored at memory address *M*. The second parameter *N* can be used to select a ROM with the given ID. The address *M* must be in [0…65535]. The parameter *N* is only relevant if the address *M* is within the ROM window (all addresses between 24576 and 32767).

Examples

```
! use assembler ROM keyword to read the first bytes of the system ROM
MEM 0,2
026 000       !  == 22 0 decimal bytes
!
PEEK$(0,0)    ! read the first byte of the system ROM (which is 0x16)
ä
PEEK8(0,0)
22
PEEK16(0,0)   ! read the first 16-bit word of the system ROM ( 0x0016)
22            ! (LSB is first and the second (high) byte is zero)
```

POKEC$ (M, A$)

Stores the character *A$* at the memory address *M*. The address *M* must be in [32767…65535], i.e. in RAM or in the I/O address range; writing to ROM has no effect.

POKE8 (M, N)

Stores the byte *N* at the memory address *M*. The address *M* must be in [32767…65535], i.e. in RAM or in the I/O address range; writing to ROM has no effect.

POKE16 (M, N)

Stores the 16-bit word *N* at the memory address *M*. The address *M* must be in [32767…65535], i.e. in RAM or in the I/O address range; writing to ROM has no effect.

Examples

```
! the byte at address 32853 defines the current angle mode
RAD               ! start in RAD mode
PEEK8(32853,0)    ! read current setting of DEG/RAD mode
0                 ! RAD
SIN(PI/2)         ! the sine of PI/2 is...
1                 ! ...in RAD mode
POKE8(32853,144)  ! now switch to DEG
0                 ! previous value was: 0 == RAD
SIN(45)           ! the sine of 45 degrees is...
0.707106781187    ! ...in DEG mode
POKE8(32853,0)    ! switch back to RAD
144               ! previous value was: 144 == DEG
```

6. Complex Numbers

A basic set of functions for working with complex numbers has been implemented. In order to keep the implementation simple some restrictions apply:

- Each complex number is represented as a Real array having at least two elements. You must dimension these variables with a `DIM` or `REAL` statement before using them.
  You cannot use `INTEGER` or `SHORT` variables.
- All complex variables must be given in rectangular form ($a + \mathrm{i} \cdot b$). You can use the `PTR` and `RTP` functions for conversion from respectively to polar coordinates.
- The data must always be stored in the first two elements of the array. The first element contains the real part a, the second the imaginary part b of the complex number.
  If `OPTION BASE 0` is selected (the default) the real and imaginary components must be loaded into elements `(0)` respectively `(1)`.
  If you use `OPTION BASE 1` the components must be stored in elements `(1)` and `(2)`.
- The keywords generally write their result to their last argument. If the output is another complex variable (not a scalar) you can use the same variable as one of the inputs because the inputs are saved before operating on them (`CPX ADD C1,C2,C1` would write the sum of `C1` and `C2` to `C1` again). Note that this is a “can” not a “must”.

Note: in the following equations all trigonometric functions are evaluated in radian mode – the global angle units setting (DEG, RAD, GRAD) does not matter.

CPX RTP C1, C2

This function converts the complex number *C1* given in rectangular coordinates to polar coordinates in *C2*.

You can use this function for output in the polar coordinate system. The result is not suitable for calculation in other CPX functions as these require all data defined in the rectangular coordinate system. The current angle units are taken into account, i.e. if DEG is selected, the angle θ will be given in degrees. If both components a and b are zero then r as well as θ are set to zero.

$$a + \mathrm{i} \cdot b \Rightarrow \begin{bmatrix} r \\ \theta \end{bmatrix}, \text{ where } \begin{matrix} r = \sqrt{a^2 + b^2} \\ \theta = \arctan\left(b \,/\, a\right) \end{matrix}$$

CPX PTR C1, C2

This function converts the complex number *C1* given in polar coordinates to rectangular coordinates in *C2*. You can use this function to convert input given in the polar coordinate system for calculation in other CPX functions as these require data defined in the rectangular coordinate system. The current angle units are taken into account, i.e. if DEG is selected, the angle must be given in degrees.

$$\begin{bmatrix} r \\ \theta \end{bmatrix} \Rightarrow a + \mathrm{i} \cdot b \text{ , where } \begin{matrix} a = r \cdot \cos\theta \\ b = r \cdot \sin\theta \end{matrix}$$

CPX ADD C1, C2, C3

This function adds the complex numbers *C1* and *C2*. The complex result is returned in *C3*.

$$\left(a + \mathrm{i} \cdot b\right) + \left(c + \mathrm{i} \cdot d\right) \Rightarrow \left(a + c\right) + \mathrm{i} \cdot \left(b + d\right)$$

CPX SUB C1, C2, C3

This function subtracts the complex number *C2* from *C1*. The complex result is returned in *C3*.

$$\left(a + \mathrm{i} \cdot b\right) - \left(c + \mathrm{i} \cdot d\right) \Rightarrow \left(a - c\right) + \mathrm{i} \cdot \left(b - d\right)$$

CPX MUL C1, C2, C3

This function multiplies the complex numbers *C1* and *C2*. The complex result is returned in *C3*.

$$\left(a + \mathrm{i} \cdot b\right) \cdot \left(c + \mathrm{i} \cdot d\right) \Rightarrow \left(a \cdot c - b \cdot d\right) + \mathrm{i} \cdot \left(a \cdot d + b \cdot c\right)$$

CPX DIV C1, C2, C3

This function divides the complex number *C1* by *C2*. The complex result is returned in *C3*.

$$\frac{a+\mathrm{i}\cdot b}{c+\mathrm{i}\cdot d} \Rightarrow \frac{a\cdot c+b\cdot d}{c^2+d^2}+\mathrm{i}\cdot\frac{b\cdot c-a\cdot d}{c^2+d^2}$$

CPX ABS C, A

This function returns the absolute value (length of the radius in polar coordinates) of the complex number *C*. The scalar result is returned in the Real variable *A*. It is the same as `RADIUS(C(0),C(1))`.

$$\mathrm{abs}\left(a+\mathrm{i}\cdot b\right) \Rightarrow \sqrt{a^2+b^2}$$

CPX ARG C, A

This function returns the argument (angle in polar coordinates) of the complex number *C*. The scalar result is returned in the real variable *A* and depends on the current setting of the angle units (`DEG`, `RAD` or `GRAD`). It is the same as `ATN2(C(1),C(0))`.

$$\arg\left(a+\mathrm{i}\cdot b\right) \Rightarrow \arctan\left(\frac{b}{a}\right)$$

CPX POW C1, C2, C3

This function calculates *C1* to the power of *C2*. Both are complex numbers. The complex result is returned in *C3*.

$$\left(a+\mathrm{i}\cdot b\right)^{(c+\mathrm{i}\cdot d)} \Rightarrow \left(a^2+b^2\right)^{\frac{c}{2}}\cdot \mathrm{e}^{-d\cdot\arctan(b/a)}\cdot \left\{\cos\left(\frac{1}{2}\cdot d\cdot\ln\left(a^2+b^2\right)+c\cdot\arctan\left(b\,/\,a\right)\right)+\mathrm{i}\cdot\sin\left(\frac{1}{2}\cdot d\cdot\ln\left(a^2+b^2\right)+c\cdot\arctan\left(b\,/\,a\right)\right)\right\}$$

For real powers (*d=0*) this general equation simplifies to

$$\left(a+\mathrm{i}\cdot b\right)^{c} \Rightarrow \left(a^2+b^2\right)^{\frac{c}{2}}\cdot\left\{\cos\left(c\cdot\arctan\left(b\,/\,a\right)\right)+\mathrm{i}\cdot\sin\left(c\cdot\arctan\left(b\,/\,a\right)\right)\right\}$$

You obtain the numeric result by setting the second element (the imaginary part) in *C2* to *0*. For example the square root of *C1* would be calculated with *C2 = (0.5, 0)*.

CPX EXP C1, C2

This function returns the natural exponential of the complex number *C1*. The complex result is returned in *C2*.

$$e^{a+i\cdot b} \Rightarrow e^{a}\cdot\left(\cos\left(b\right)+\mathrm{i}\cdot\sin\left(b\right)\right)$$

CPX LN C1, C2

This function returns the natural logarithm of the complex number *C1*. The complex result is returned in *C2*.

$$\ln\left(a+\mathrm{i}\cdot b\right) \Rightarrow \frac{1}{2}\cdot\ln\left(a^{2}+b^{2}\right)+\mathrm{i}\cdot\arctan\left(\frac{b}{a}\right)$$

CPX COS C1, C2

This function calculates the cosine of the complex number *C1*. The complex result is returned in *C2*.

$$\cos\left(a+\mathrm{i}\cdot b\right) \Rightarrow \cos\left(a\right)\cdot\cosh\left(b\right)-\mathrm{i}\cdot\sin\left(a\right)\cdot\sinh\left(b\right)$$

CPX SIN C1, C2

This function returns the sine of the complex number *C1*. The complex result is returned in *C2*.

$$\sin\left(a+\mathrm{i}\cdot b\right) \Rightarrow \sin\left(a\right)\cdot\cosh\left(b\right)+\mathrm{i}\cdot\cos\left(a\right)\cdot\sinh\left(b\right)$$

CPX TAN C1, C2

This function determines the tangent of the complex number *C1*. The complex result is returned in *C2*.

$$\tan\left(a+\mathrm{i}\cdot b\right) \Rightarrow \frac{1}{\cos\left(2\cdot a\right)+\cosh\left(2\cdot b\right)}\cdot\left(\sin\left(2\cdot a\right)+\mathrm{i}\cdot\sinh\left(2\cdot b\right)\right) .$$

The actual implementation calls CPX SIN and CPX COS and then CPX DIV. While this might be slower than implementing the equation above, it conserves code space.

Examples

```
! calculator mode with default OPTION BASE 0
REAL A(2),B(2),C(2),R,P
A(0) = 1.0  @ A(1) = 2.0
B(0) = 10.0 @ B(1) = 20.0
CPX ADD A,B,C
DISP "(";A(0);A(1);") + (";B(0);B(1)+") = (";C(0);C(1);")"
( 1  2 ) + ( 10  20 ) = ( 11  22 )
!
RAD
CPX ABS A,R
CPX ARG A,P   ! get radius and angle
DISP "(";A(0);A(1);") => (";R;P; ")"
( 1  2 ) => ( 2.2360679775  1.10714871779 )
!
CPX SIN A,B
DISP "SIN(";A(0);A(1);") = (";B(0);B(1);")"
```

```
SIN( 1  2 ) = ( 3.16577851322  1.95960104142 )
```

7. Mathematical and Statistical Functions

GCD (M, N)

This function returns the Greatest Common Divisor of the integer numbers *M* and *N*. It is an implementation of Euclid's Algorithm according to Knuth [1] (Algorithm A, page 320).

M and *N* must be integers in the range of HP 85 integers [0 ... 99999]. *N* must be lower or equal to *M*.

Example

The number 24 is $2 \cdot 3 \cdot 4$ and 9 is $3 \cdot 3$. Their largest common divisor is 3.

```
GCD(24,9)
3
```

LCM (M, N)

This function returns the Least Common Multiple of the integer numbers *M* and *N*.

M and *N* must be integers in the range of HP 85 integers [0 ... 99999].

Example

Multiples of 16 are [16, 32, 48, …] and multiples of 6 are [6, 12, 18, 24, 30, 36, 42, 48, …].

The lowest (least) multiplier which these two numbers have in common is 48.

```
LCM(16,6)
48
```

COMB (M, N)

Returns the "Binomial Coefficient" $\binom{M}{N}$ (also called as "from *M* chose *N*").

This is the number of combinations of *N* samples you can take from a pool of *M* items and if you don't care about the order of these *N* items.

$$COMB\left(M,N\right) = \frac{M!}{N! \cdot \left(M-N\right)!} \ .$$

When programming an algorithm closely following this definition the result may become inaccurate due to conversion to floating point numbers and overflow already when *M* is as low as about 16. The following alternate expression can be derived from the definition above:

$$COMB\left(M,N\right) = \frac{\prod_{i=M-N+1}^{M} i}{\prod_{i=2}^{N} i} \ .$$

It removes some of the overlapping regions of the products[1] and thus shifts the overflow towards larger values.

M and *N* must be integers in the range of HP 85 integers [0 ... 99999] and *M* must be greater than or equal to *N*.

Example

You have got 4 cars branded “A”, “B”, “C”, ”D” in your garage. How many different pairs can you combine from this collection?

You can compose six car pairs: “AB”, “BC”, “CD”, “AC”, “AD”, “BD”. If you also want to account for swapping (the pair “AB” is different from “BA”) use `PERM`.

```
COMB(4,2)
6
```

PERM (M, N)

Returns the number of permutations of *M* items taken *N* at a time.

This is the number of permutations of *N* samples you can take from a pool of *M* items. Implementing it by closely following the definition

$$PERM\left(M,N\right)=\frac{M!}{\left(M-N\right)!}$$

where $M!$ is calculated first and the result is then divided by $\left(M-N\right)!$ may lead to numerical overflow already when *M* is as low as 16. In the binary program the result is calculated by the product[1]:

$$PERM\left(M,N\right)=\prod_{i=M-N+1}^{M} i \ .$$

This alternate expression removes the overlapping parts of the products $M!$ and $\left(M-N\right)!$ and thus shifts the overflow problem towards larger values.

M and *N* must be integers in the range [0 … 99999]. *M* must be greater than or equal to *N* and *N* must be greater than zero. If *M* is greater than about 16, the result may become inaccurate due to conversion to floating point numbers.

Example

You have got 4 toy cars “A”, “B”, “C”, “D” in your drawer. How many different pairwise arrangements can you make from this pool?

[1] In case you forgot: in mathematical notation the sign Π (capital “pi”) denotes the product of a series of expressions – in this case a series of numbers. This is similar to the Σ (capital “sigma”) sign which is used to abbreviate a sum of expressions.

You can compose 12 combinations of cars: “AB” and “BA”, “BC” and “CB”, “CD” and “DC”, “AC” and “CA”, “AD” and “DA”, “BD” and “DB”.

```
PERM(4,2)
12
```

IPROD (M, N)

Returns the product of the integer numbers from *M* to *N* (inclusive).

M and *N* must be integers in the range [–99999…99999]. *M* must be lower than or equal to *N*. If the result exceeds the range of HP 85 integers it will be returned as a floating point number.

Example

Calculate the product of $4 \cdot 5 \cdot 6 \cdot 7 \cdot 8$:

```
IPROD(4,8)
6720
```

ISUM (M, N)

Returns the sum of the integer numbers from *M* to *N* (inclusive).

M and *N* must be integers in the range [-99999…99999]. *M* must be lower than or equal to *N*. If the result exceeds the range of HP 85 integers (99999) it will be returned as a floating point number.

Example

The sum of the numbers from 1 to 100 (see also example for the `SUM` function):

```
ISUM(1,100)
5050
```

PRIME? (M)

Returns *1* if the given integer *M* is prime, *0* otherwise.

M must be an integer in the range [0…99999].

Example

Test whether the number 99991 is a prime:

```
IF PRIME?(99991) THEN DISP "YEP!" ELSE DISP "NOPE!"
YEP!
```

FACT (M)

Returns the factorial *M!* of the integer number *M*.

M must be integer and in the range [0…200]. Values above 10 produce already quite large results. If *M* is greater than 14, the result will be a floating point approximation because its magnitude exceeds the maximum for HP 85 integers (99999).

Example

$$6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$$

```
FACT(6)
720
```

FIB (M)

Returns the *Mth* element of the "Fibonacci Series". This series starts with the two numbers 0 and 1 and each subsequent number is the sum of the two preceding numbers.

$$0, 1, 1, 2, 3, 5, 8, \ldots$$

M must be integer in the range [0…99999]. The first element of the series is returned for *M=0*.

For accurate results, *M* should be lower than 60. If *M* is above 25 the results exceed the maximum magnitude of an HP 85 `INTEGER` (99999) and are therefore converted to `REAL`. Beyond 59 they are returned as real numbers with an exponent so that the last digits will be lost.

Example

```
FIB(0)
0
FIB(20)
6765
```

ROUND (X,M)

Returns the number *X* rounded to *M* decimal places. Positive numbers are rounded up, negative numbers are rounded down.

Example

```
ROUND(1.567,2)
1.57
ROUND(-1.567,2)
-1.57
```

SINH (X)

Returns the hyperbolic sine of the real number *X*.

To avoid under- or overflow $|X| < 1148$.

Example

```
SINH(1)
1.17520119365
```

COSH (X)

Returns the hyperbolic cosine of the real number *X*.

To avoid under- or overflow $|X| < 1148$.

Example

```
COSH(1)
1.54308063482
```

TANH (X)

Returns the hyperbolic tangent of the real number *X*.

To avoid under– or overflow $|X| < 1148$.

Example

```
TANH(1)
0.761594155957
```

ASINH (X)

Returns the area hyperbolic sine (inverse of hyperbolic sine) of the real number *X*.

Example

```
ASINH(1)
0.881373587018
```

ACOSH (X)

Returns the area hyperbolic cosine (inverse of hyperbolic cosine) of the real number *X*.

To avoid under– or overflow $X \geq 1$.

Example

```
ACOSH(1.25)
0.69314718056
```

ATANH (X)

Returns the area hyperbolic tangent (inverse of hyperbolic tangent) of the real number *X*.

To avoid under- or overflow $|X| < 1$.

Example

```
ATANH(0.25)
0.255412811884
```

SDIGS (M)

Returns the sum of the digits in the number *M*.

M can be any type (Integer, Short or Real) and can have up to 12 digits so that the maximum result will be 12·9 = 108. In case of floating point numbers the digits of the mantissa will be summed, any exponent and decimal point are neglected.

Example

```
1+2+3+4+5+6+7+8+9
45

10 T0=TIME               ! CONVENTIONAL
20 FOR L=1 TO 5000
30 K=123456789
40 S$=VAL$(K)
50 N=0
60 FOR I=1 TO LEN(S$)
70 N=N+VAL(S$[I,I])
80 NEXT I
90 NEXT L
100 DISP N
110 DISP TIME-T0
120 T0=TIME              ! USING SDIGS
130 FOR L=1 TO 5000
140 K=123456789
150 N=SDIGS(K)
160 NEXT L
170 DISP N
180 DISP TIME-T0
190 END

RUN
45
6.125                    ! TIME
45
0.358                    ! TIME
```

SUM (X, M, N)

Returns the sum of the elements from *M* to *N* of the `REAL` array *X*.

X must be a 1-dimensional array (vector) of type `REAL`. *M* and *N* must be integers and define the starting respectively ending indices of the range to be summed. You can divide the result by *(N–M+1)* to obtain the algebraic mean value of the selected elements.

Compared to performing the sum inside a *FOR / NEXT* loop this function provides a speed-up of about 5 to 10, depending on the number of elements to be summed.

Example

To keep quiet for some time, the Gauss boy was asked to calculate the sum of all integers from 1 to 100. If he had an HP 85 he could have written the following program

```
10 REAL X(100)          ! important: must be REAL
20 FOR I=1 TO 100
30   X(I) = I           ! fill with numbers
40 NEXT I
50 S = SUM(X,1,100)
60 DISP S
70 END
```

which produces the output

```
5050
```

He could also have used the `ISUM` function which is exactly tailored for this task.
(of course Gauss had no computer but he noticed the symmetry and he calculated $\left(1+100\right)\cdot 50$ to obtain the same result in no time.)

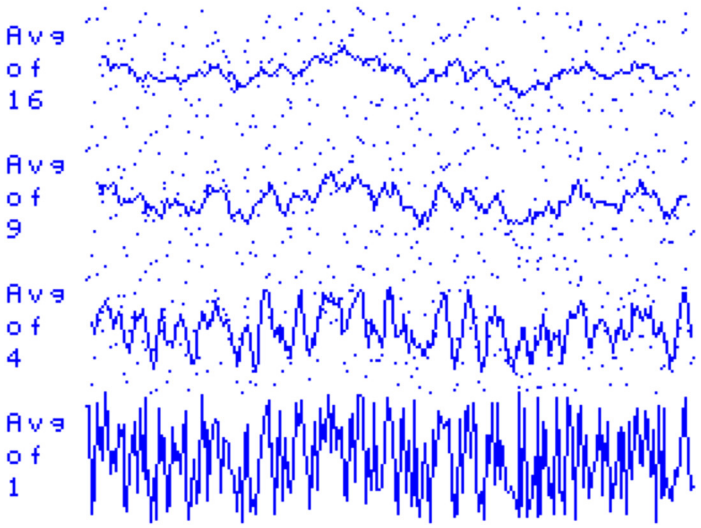

Figure 1: Sliding averages over different numbers of samples using the SUM function.

```
10 ! --------------------------
20 ! sliding average example
30 ! --------------------------
40 REAL X(256)
50 FOR I=1 TO 256
60 X(I)=RND
70 NEXT I
80 !
90 GCLEAR
100 FOR K=1 TO 4
110 SCALE -32,256,1-K,5-K
120 MOVE -32,.7
130 LABEL "Avg"
140 LABEL "of"
150 LABEL VAL$(K^2)
160 FOR I=1 TO 256
170 MOVE 1,X(1)
180 PLOT I,X(I)
190 NEXT I
```

```
200 P=0
210 M=K^2
220 D=(M-1)/2
230 FOR I=1+D TO 256-D
240 S=SUM(X,I-D,I+D)/M
250 IF P=0 THEN MOVE I,S ELSE DRAW I,S
260 P=1
270 NEXT I
280 NEXT K
290 END
```

Figure 2: Associated BASIC program.

SUMSQ (X, M, N)

Returns the sum of the square of each element from *M* to *N* stored in the `REAL` array *X*.

Same as `SUM` except that each element is squared before it is added to the sum.

Example

The `SUM` and `SUMSQ` functions allow for easy calculation of the algebraic mean

$$\bar{x} = \frac{1}{N} \cdot \sum_{i=1}^{N} x_i$$

and the sample standard deviation

$$\sigma = \sqrt{\frac{1}{N-1} \cdot \sum_{i=1}^{N} \left(x_i - \bar{x}\right)^2}$$

which can also be written as

$$\sigma = \sqrt{\frac{1}{N-1} \cdot \left(\sum_{i=1}^{N} x_i^2 - \frac{1}{N} \cdot \left(\sum_{i=1}^{N} x_i \right)^2 \right)} \ .$$

Example program:

```
10 REAL X(5)     ! must be declared as REAL!
20 X(1)=1.0
30 X(2)=2.0
40 X(3)=3.0
50 X(4)=4.0
60 X(5)=5.0
65 N=5
70 S=SUM(X,1,N)
80 Q=SUMSQ(X,1,N)
90 ! algebraic mean
100 A=S/N
110 ! sample standard deviation
120 S=SQR((Q-S^2/N)/(N-1))
130 DISP A;S
140 END
```

which produces the output

```
3   1.58113883008
```

ERF (X)

Returns an approximation of the Error function for the real number *X*. Its definition[2] is

$$erf\left(x\right) = \frac{2}{\sqrt{\pi}} \cdot \int_{t=0}^{x} e^{-t^2} dt$$

The approximation is taken from Abramowitz [2] (7.1.26) with an additional refinement in the range [*–0.005* ≤ *X* ≤ *0.005*].

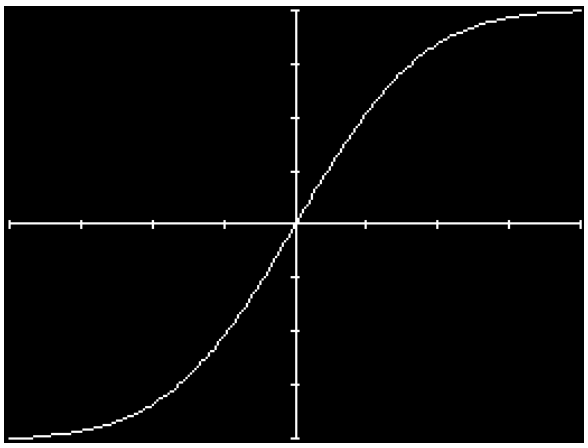

Figure 3: Plot of the function ERF(X) for -2 < X < 2.

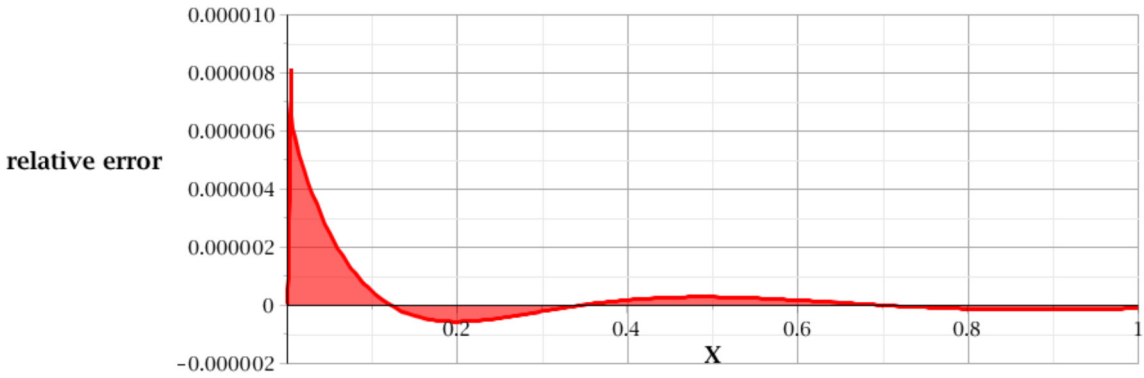

Figure 4: Error of the approximation relative to the "accurate" value $\left(ERF\left(x\right) - erf\left(x\right)\right) / erf\left(x\right)$ **over the range of 0 ≤ X ≤ 1. The "accurate" result was produced with the software *Maple*.**

GAMMA (X)

Returns an approximation of the Gamma function for the real number *X*. The value of *X* must be in the range [–250…250] to avoid overflow.

The approximation was generated by a series expansion around *X = 1.5* and is rather accurate for *1 < X < 2*. The results for *X<1* and *X>2* are determined by applying these relations:

---

[2] Note that this function is linked to the probabilistic integral $\Phi\left(x\right) = 2 / \sqrt{2 \cdot \pi} \cdot \int_{t=0}^{x} e^{-\frac{t^2}{2}} dt$ by the relation $\Phi\left(x\right) = erf\left(x / \sqrt{2}\right)$.

$$\Gamma\left(1+x\right) = x \cdot \Gamma\left(x\right) \ , \qquad \text{(recursively used for large } x\text{)}$$

and

$$\Gamma\left(1-x\right) = \frac{\pi \cdot x}{\Gamma\left(1+x\right) \cdot \sin\left(\pi \cdot x\right)} \ . \qquad \text{(used for small } x\text{)}$$

The overall result is a rather good accuracy over the whole range of interest which, however, comes at the cost of about 300 lines of assembler code.

Note that if you need results for integer numbers larger than 1, you should use the accurate result from

$$\Gamma\left(n\right) = \left(n-1\right)!$$

instead of the approximation by the `GAMMA` function.

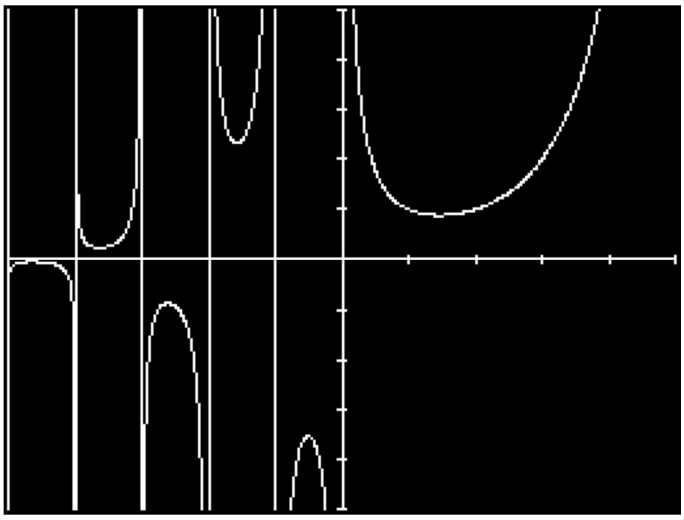

Figure 5: Plot of the function GAMMA(X) for -5 < X < 5.

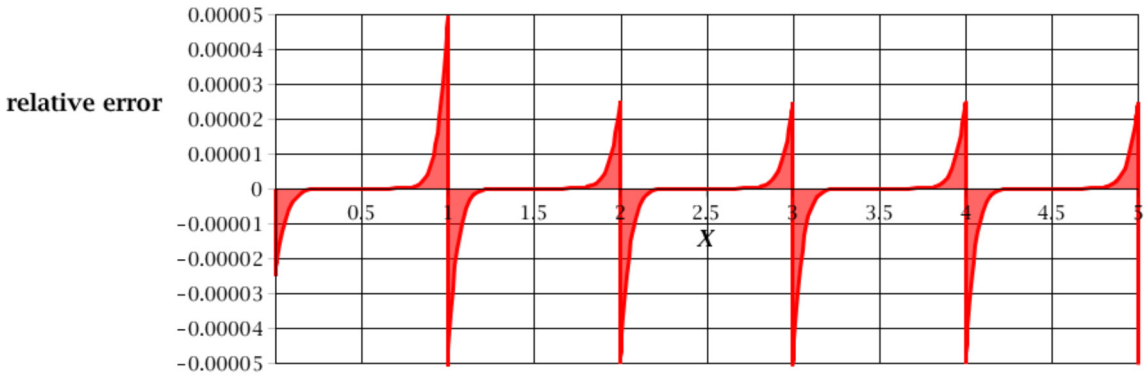

Figure 6: Error of the approximation relative to the "accurate" value $\left(GAMMA\left(x\right) - \Gamma\left(x\right)\right) \Big/ \Gamma\left(x\right)$ over the range of 0 ≤ X ≤ 5. The "accurate" result was produced with the software *Maple*.

Examples

```
GAMMA(1.0)
0.999949475315      (should be 1.0)
GAMMA(2.0)          (worst case test, better use FACT(2-1)=1 for integer parameter)
1.00002520061       (should be 1.0)
```

```
GAMMA(0.5)
1.77245385091          (should be 1.77245385091)
```

ELLIPTIC_E (X)

Returns an approximation of the Complete Elliptic Integral of the Second Kind *E* for the real number *X*. The return value is a real so that the input value of *X* must be in the range [−1…1].

$$E\left(k\right) = \int_0^{\pi/2} \sqrt{1 - k^2 \cdot \sin^2\left(\vartheta\right)} \cdot d\vartheta$$

ELLIPTIC_K (X)

Returns an approximation of the Complete Elliptic Integral of the First Kind *K* for the real number *X*. The return value is a real so that the input value of *X* must be in the range [−1…1].

$$K\left(k\right) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - k^2 \cdot \sin^2\left(\vartheta\right)}} \cdot d\vartheta$$

The determination of $E\left(k\right)$ and $K\left(k\right)$ follows the iterative Arithmetic-Geometric-Mean procedure as found in [2], p. 598 and also implemented in HP-25 RPN in [4].

8. Calendar Functions

These functions take a switch from the Julian to the Gregorian calendar on 15 October 1582 into account. Therefore you can use them for most practical applications. They do not check the validity of their input – if you call `JULDAY` with a date of 32-DEC-2021 you will obtain a result corresponding to 1-JAN-2022.

JULDAY (D, M, Y)

Returns the *Julian Day Number* for the calendar date given by day *D*, month *M* and year *Y*. The *Julian Day Number* is a sequential day number (not related to the Julian calendar) suitable for calendrical and astronomical calculations. It may include a fraction of day, as each Julian Day starts at noon, which is half into the Greenwich day, starting at midnight – therefore the result will carry a fraction of 0.5. The difference between two dates in form of their associated *Julian Day Numbers* yields their difference in days.

JULDAT (J, N)

Returns one date component of the given Julian day number *J*. The return value depends on the number *N*: *N=1* returns the day of the month, *N=2* returns the month and everything else returns the year.

DOW(D, M, Y)

Returns the ISO day of the week number for the date given by day *D*, month *M* and year *Y*. The return values range from Monday = 1, to Sunday = 7.

Examples

```
DIM D$[63]
D$="MONDAY   TUESDAY  WEDNESDAYTHURSDAY FRIDAY   SATURDAY SUNDAY   "
! which day is 14 days before the last day of the year?
J = JULDAY(31,12,2023)-14
D = JULDAT(J, 1)
M = JULDAT(J, 2)
Y = JULDAT(J, 3)
DISP D,M,Y
! output the corresponding day of the week
I = DOW(D,M,Y)*9-8
DISP D$[I,I+8]
```

9. Implementation Notes

While I would not claim to be an expert in math or in the Capricorn assembly language I would like to explain a few implementation details and maybe useful "tricks" for newcomers which I learned while writing the assembler routines.

Note that these notes were written during the development of the HP 85 version. Many aspects also apply to the HP 86/87 variant. Differences are mostly related to string and array accessing due to the extended address width supported by the extended memory controller (EMC) used in the 86/87 systems.

The Gamma Function

The approximation for Gamma function $\Gamma(x)$ in the region 1<x<2 is given by this series developed around x=1.5:

$$\Gamma(x) \approx \begin{pmatrix} 0.886226925453 + \\ (x-1.5)\cdot\begin{pmatrix} 0.0323383974489 + \\ (x-1.5)\cdot\begin{pmatrix} 0.414813453688 + \\ (x-1.5)\cdot\begin{pmatrix} -0.107294804565 + \\ (x-1.5)\cdot\begin{pmatrix} 0.144645359045 + \\ (x-1.5)\cdot\begin{pmatrix} -0.0775230522999 + \\ (x-1.5)\cdot\begin{pmatrix} 0.0586103038172 + \\ (x-1.5)\cdot\begin{pmatrix} -0.0380019355548 + \\ (x-1.5)\cdot 0.0258376064559 \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

Values for x<1 or x>2 are derived from applying the formula

$$\Gamma(x+1) = x \cdot \Gamma(x) \ .$$

The Error Function

The Error function $erf\left(x\right)$ is calculated according to [2] from

$$
\begin{aligned}
&s = signum(x)\\
&if\ \left|x\right| <= 0.005\ then\\
&\quad erf\left(x\right) \approx s \cdot \left|x\right| \cdot 1.1283791670955\\
&else\\
&\quad t = \frac{1}{1 + .3275911 \cdot \left|x\right|}\\
&\quad erf\left(x\right) \approx s \cdot (1 - t \cdot (0.254829592 + t \cdot (-0.284496739 + t \cdot (1.421413741 +\\
&\qquad\qquad t \cdot (-1.453152027 + t \cdot 1.061405429)))) \cdot e^{-\left|x\right|^2})\\
&end\ if
\end{aligned}
$$

Access to Array Elements

Unfortunately, none of the examples accompanying the Assembler ROM covers the usage of arrays as parameters for functions. The Series-80 system offers one function to read the value of an individual array element (`FETAV`) and one to obtain the address of an array element (`FETAVA`).

The `FETAV` function first performs tests to determine the type of array (`REAL`, `INTEGER`, `SHORT`) and then converts the desired element to an 8-byte Real or Tagged Integer representation. It also accounts for an active `TRACE` condition. Later we can use one of the many "universal" system routines to perform operations on these 8-byte numbers. These routines also test their input and if they require e.g. a Real they convert a Tagged Integer into a Real before calling the "specialized" system routines which can only handle one specific type.

All these tests and conversions lead to a rather robust system which can mix Reals and Integers in all calculations. But they also slow down accessing a large number of array elements, for example inside a tight loop.

If we constrain our binary functions to a certain type of parameter (say Real) we can speed up the access just obtaining the address of the first array element (using the system routine `FETAVA`) and then walking through the elements under the assumption that they all contain the desired type.

As a safety measure we can check the array type once without a huge speed penalty.

There is one pitfall, though: even if an array is declared as Real it may also contain Tagged Integer values. Thus one has to make sure that either "real Reals" are stored in the array or we would still have to test each array element for Tagged Integer inside our tight loop and perform the conversion.

I case of the `SUM` and `SUMSQ` functions I use the `ONER+` system function for the test and conversion without affecting the R12 stack.

Access to Array Elements from Keywords

If we need the address of a numeric variable we use the `REFNUM` function in the parse routine to push tokens 21 or 22 onto the execution stack. At run time the executive then leaves the name field and the

absolute address of the variable on the R12 stack. If we are in TRACE mode, and we have a reference to an array element, then some additional information is pushed to the R12 stack.

One and two-dimensional array indices are numbered left to right `X(Column)`, `X(Column,Row)`.

	Bytes	Name	Content	Presence
	2	`Address`	abs. address	always
	2	*`Column Index`*		*if tracing*
	2	*`Row Index`*	*=0 in 1-D arrays*	*if tracing*
	1	*`Dim flag`*	*= 1: 1-D array* *= 0: 2-D array*	*if tracing*
R12→	2	`Name`	name and type	

There seems to be an error in both known versions of the Assembler ROM manual (page 5-35): the `Row Index`. is always present, even if the array is one-dimensional.

Examples

Accessing the element X(8,1) of a two-dimensional array

	Bytes	Value	Description
TRACE MODE	2	103 214	address
	2	010 000	= 8
	2	001 000	= 1
	1	000	= 2-D array
R12→	2	112 130	name

	Bytes	Value	Description
NORMAL MODE	2	103 214	address
R12→	2	112 130	name

Accessing the element X(2) of a one-dimensional array

	Bytes	Value	Description
TRACE MODE	2	147 213	address
	2	002 000	= 2
	2	000 000	= 0
	1	001	= 1-D array
R12→	2	112 130	name

	Bytes	Value	Description
NORMAL MODE	2	147 213	address
R12→	2	112 130	name

Errors in the Assembler ROM Manual

When I received my HP-85 the bundle also included a printed Assembler ROM manual. I used this manual to develop my Capricorn assembler skills. While I was writing my programs I was sometimes wondering why some system calls failed and I re-read the (in some parts rather sparse) documentation again and again. By using the debug mode of Everett Kaser's invaluable Series-80 emulator and the source code of the system ROMs I discovered some errors in my manual. These are listed below. Only later I compared my copy with another Assembler ROM manual available in PDF format on some internet sites and discovered that the latter did not contain these errors.

So be aware that there are at least two issues of this manual available (The original December 1980 issue 00085-90444 can be found on the "Museum of HP Calculators" web site (www.hpmuseum.org), the improved issue 00085-90444 Rev. B of November 1981 can be downloaded from Vassilis Prevelakis' "HP Series 80 Web Site" (www.series80.org).

The `FETAV` and `FETAVA` system calls:

The R12 stack must contain the following items for these functions to work:

	Initial Assembler ROM Manual		**Correct Parameter Description (corrected in Rev. B)**	
	Pointer to variable area	(2 bytes)	Pointer to variable header	(2 bytes)
	Row dimension	(2 bytes)	Row index	(8 bytes, Tagged Integer or Real)
	Column dimension	(2 bytes)	Column index	(8 bytes, Tagged Integer or Real)
R12 →	Dimension flag	(1 byte)	Dimensions	(1 byte), 1 or 2

The `SEMIC$` system call:

The R12 stack must contain the following items for these functions to work:

Initial Assembler ROM Manual		**Correct Parameter Description (corrected in Rev. B)**	
Length of string	(8 bytes)	Length of string	(2 bytes)
Address of string	(8 bytes)	Address of string	(2 bytes)

The example for ROM programs:

The parse routine of the example ROM code on page 8-21 pushes a BPGM token # 371 instead of the ROM token # 370 onto the R12 stack.

Taking OPTION BASE into Account

The indices expected by `FETAV` and `FETAVA` must be given according to the current `OPTION BASE`. This means that user supplied indices can be directly forwarded to these functions. For example the index of the first element would be `0` if `OPTION BASE 0` is active respectively `1` if `OPTION BASE 1` is selected.

However, if you want to access a specific element (e.g. the first one) of an array from your assembler code, independent of the current `OPTION BASE`, you must calculate the proper index yourself.

This can be accomplished by either reading the current `OPTION BASE` from the programs PCB (see Assembler ROM manual) or, more convenient, by using the global variable `OPTBAS`. This variable contains `1` when `OPTION BASE 0` is set respectively `0` when `OPTION BASE 1` is active. To calculate the address of the first element, you could therefore load a "one-based" value of `1` into a register holding the index and then simply subtract the content of `OPTBAS` from this value. The result would be the proper index (`0` resp. `1`) of the first element.

Self-Modifying Code

One classic pattern in software development is “self-modifying code”. While often frowned at, this technique can lead to very compact and efficient programs. One application is to “patch” the code of a function at runtime to perform specific operations depending on the current set of input parameters. Usually this would require a test and a branch for a certain combination of input parameters. Consider an algorithm which must perform the comparison and the conditional branch inside a loop. If this condition can be evaluated beforehand, outside of the loop, then a test and branch instruction pair inside the loop can be replaced by a simple jump or even be completely left out.

As a personal challenge I tried to optimize the `SUM` and `SUMSQ` functions for speed. Because the two functions are almost identical except that `SUMSQ` needs to square each value before adding it to the sum, both share the same core routine. The distinction between the two is made with a flag.

Initially the flag was tested inside the summation loop each time a new value was to be added to the sum. If the simpler `SUM` function was executed, the code would jump over the branch which squares the number.

In order to pull this test out of the loop, it was moved in front of the loop and the proper jump distance was patched into the loop. In case of the `SUMSQ` function the result is a `JMP 0` jump, while the `SUM` function obtains the required `JMP 6` to jump over the square branch. Thus we can eliminate one `TSB` instruction eating 5 clock cycles – why not?

Because this would only work in a RAM-based binary program conditional assembly instructions must be used to select from either the faster or the non-modifying implementation if a ROM version of a binary program shall be produced.

Another solution would store a pointer to one of two worker routines in a register and then use an indirect call to call the appropriate routine from inside the tight loop. This would require slightly more time consuming `JSB` and a `RTN` instructions.

```
5420 ! ----- <BPGM>
5430         AIF BINCOD
5440 ! self modifying code!
5450 ! patch jump distance
5460 ! to avoid test in loop
5470 ! NOT FOR ROM CODE!
5480         LDM R36,=BYTSKP      ! where we want to patch
5490         LDMD R34,=BINTAB     ! base address of BPGM
5500         ADM R36,R34          ! R36-R37: abs addr. of BYTSKP
5510 ! zero jump if X^2 desired
5520         CLB R34
5530         TSB R20              ! X^2? test once outside of loop
5540         JNZ PATCH            ! patch  zero distance if X^2
5550 ! ------------------------
5560 ! calculate jump distance
5570 ! Example:
5580 ! A LBL  INSTR
5590 ! 1:     JMP    leaves PC
5600 ! 2:DIST 002    at 3:
5610 ! 3:     ... >-+
5620 ! 4:     ...   | 2 bytes
5630 ! 5:TARG ... <-+
5640 ! 5-3=2 TARG-(DIST+1)
5650 !    =  TARG-DIST-1
5660 ! ------------------------
5670         LDM R34,=NOSQ        ! jump target
5680         SBM R34,=BYTSKP      ! jump from here
5690         DCB R34              ! minus 1
5700 PATCH   STBD R34,R36         ! patch it in!
```

```
5710 ! patch done
5720         EIF
5730 ! ----- </BPGM>
5740 !
5750 SUM.3   TSM R75              ! start of summing loop
5760         JNG SUM.4
5770         DCM R75              ! N=N-1
5780         POMD R40,+R22        ! 8 bytes stride!
5790         PUMD R40,+R12        ! X(I)
5800 ! ----- <BPGM>
5810         AIF BINCOD
5820         BYT 360              ! JMP
5830 BYTSKP  BYT 000              ! distance in bytes
5840         EIF
5850 ! ----- </BPGM>
5860 ! ----- <ROM>
5870         AIF ROMCOD
5880         TSB R20              ! test each time
5890         JZR NOSQ             ! and possibly jump
5900         EIF
5910 ! ----- </ROM>              !--- for SUMSQ only ---
5920         PUMD R#,+R12         ! calculate square
5930         JSB =MPYROI          ! X(I)^2
5940 NOSQ    JSB =ADDROI          ! no square needed
5950         JMP SUM.3            !---------------------
5960 SUM.4   BSZ 0                ! sum is already on R12 stk for rtn
5970         RTN
```

Figure 7: Code fragment to patch a jump distance into memory.

Self-modifying code was also used to simplify the binary and string bitwise functions. In this case the logical operator (AND, OR, XOR) is written to common worker routines.

Another, more interesting approach to the efficient generation of "very similar" code was applied in the ASWAP function. This function must handle REAL, INTEGER and SHORT variables which differ only in storage size. Therefore the code is almost identical, only the number of bytes and thus the starting registers of s block of LDMD/STMD instructions change.

For this purpose, I placed self-modifying code on the R12 stack and executed it there. This would also work in a ROM and may be the best solution.

The code sequence for 8-byte REALs is short and can be prepared in register sets R40-R47 and R54-R57 like so:

```
2130 ! code sequence for REAL numbers
2140 ! 140,070,245 LDMD R40,R70
2150 ! 160,074,245 LDMD R60,R74
2160 ! 070,247     STMD R60,R70
2170 ! 140,074,247 STMD R40,R74
2180 ! 236         RTN             ! important!!!
2190 ! We adapt R40, R43, R54
2200 ! later for SHORT, INTEGER
2210         LDM R40,=140,070,245,160,074,245,070,247
2220         LDM R54,=140,074,247,236
```

The final `RTN` instruction returns to the caller of this whole subroutine from which only a code fragment is shown here. If we want to return to a different location, we could manipulate the R6 stack accordingly. If you forget this instruction, unknown code or data would be executed from dark areas of your R12 stack.

If the code has to be modified for INTEGER (3 bytes) or SHORT (4 bytes) I first adapt it for SHORT by replacing the starting registers R40 and R60 by R44 and R64

```
2260 ! prepare for SHORT
2270 ! SHORT: use R44, R64
2280         LDB R40,=144
2290         LDB R43,=164
2300         LDB R54,=144
```

If the code has to be modified for the even shorter INTEGER the three occurrences of the starting registers are incremented by one:

```
2450 SWAP.I ICB R40      ! SHORT to INTEGER: start one byte later
2460        ICB R43
2470        ICB R54
2480        JMP SWAP!
```

Compared with the complete LDB instructions this incremental approach saves a few bytes.

The complete code sequence is pushed to the R12 stack and the actual execution is then triggered by storing the initial address of the R12 stack into the program counter register R04-R05.

```
2370 SWAP!  LDM R22,R12          ! save R12 as pointer to start of code
2380 !DEBUG LDB R20,=336         ! DEBUG BREAK
2390 !      PUBD R20,+R12        ! insert DEBUG BREAK
2400        PUMD R40,+R12        ! code sequence to R12 stack
2410        PUMD R54,+R12
2420        LDM R12,R22          ! reset to empty R12 stack
2430        DCM R22              ! decrement because LDM increments PC
2440        LDM R04,R22          ! set PC to R12 => execute code
```

If you embed a `BYT 336` instruction in the stack code, Everett Kaser's emulator will break into the debugger so that you can inspect the code while it executes on the stack.

Note line 2420 to make sure that we leave an empty R12 stack. Otherwise the executive will complain with `ERROR SYSTEM`.

A similar approach was followed in the `PEEK` function where a short piece of code is executed on the R12 stack. If the code would be executed in a ROM, I could not switch the ROM, thus pulling the rag under my feet. Here, a local JSP call is executed, which prepares and executes the code on the stack and then RTNs to the routine which called it.

The FMT$ Function

Creating a string with a formatted number is not easy on the HP 85. The system only offers the `VAL$` function for unformatted conversion and the `IMAGE` specifier for formatted output to the display, the printer. The latter can be used with the `DISP USING` and `PRINT USING` statements. Unfortunately these statements do not have the ability to append multiple output blocks to a single line by specifying a trailing semicolon like the `DISP` and `PRINT` statements. Therefore I created my own formatting function. As I did not want to reinvent the wheel I searched for the wheel inside the system ROM source code listings. After some searching, tracing and, head-scratching I finally found the routines behind the `USING` and `IMAGE` statements. In the end the required code is surprisingly short and simple. It is similar to the system ROM routines except that the resulting text buffer is not output to the display or the printer device but returned to the caller.

Despite the final simplicity it took me a whole day to figure out the proper usage and sequence of the internal system routines.

The R12 Stack

The R12 stack is used for parameter passing to functions defined in binary programs as well as in the system ROMs. Parameters are pushed onto the stack first to last and later popped off last to first. This parameter stack is an "increasing stack"; R12 contains the address of the next free memory location. When a parameter is used multiple times it is usually popped into a register and pushed back onto the stack or sometimes the return stack R6 is used for temporary storage. If you want to access parameters several times or in a sequence out of sequence this often leads to a cascade of PUSH/POP operations.

```
1000 ! entry to a function with 2 numeric parameters
1010         BYT 40,55
1020 DEMO.   BSZ 0             ! DEMO(A,B)
1030 ! R12 stack on entry:
1040 !         A (8 bytes)
1050 !         B (8 bytes)
1060 ! R12 -> next free slot on R12 stack
1070 !       we need parameter "A" first
1080         POMD R50,-R12     ! get "B" out of the way
1090         POMD R40,-R12     ! now load "A" to R40
1100         PUMD R50,+R12     ! and save "B" again for later
... A has been taken off the stack
... B is again on the stack
1600         POMD R40,-R12     ! finally get and use B
1610         RTN
```

Figure 8: Code fragment using pop and push to get a specific parameter off the R12 stack.

In such cases it might be more convenient to copy parameters directly from the R12 stack (or store values there) without disturbing the stack pointer R12. This can be accomplished by code like this:

```
1000 ! entry to a function with 2 numeric parameters
1010         BYT 40,55
1020 DEMO.   BSZ 0               ! DEMO(A,B)
1030 ! R12 stack on entry:
1040 ! R20 -> A (8 bytes)          (R20 set in line 1090)
1050 !        B (8 bytes)
1060 ! R12 -> next free slot on R12 stack
1070 !
1080         LDM R20,R12         ! copy address in R12
1090         SBM R20,=20,0       ! subtract 2*8 to point to "A"
1100         LDMD R40,R20        ! load "A" into R40, can be repeated as
1110                             ! needed without disturbing R12 or R20
...
1500         ADM R20,=10,0       ! maybe later: point to B
1510         LDMD R50,R20        !              get a copy of "B"
... A is still on the stack
... B is still on the stack
1600         SBM R12,=20,0       ! forget A and B by decrementing R12
1610         RTN
```

Figure 9: Code fragment to copy parameters off the R12 stack or to use R20 as a stack pointer.

This works but looks rather bloated and there is a more straightforward way for accessing the stack: indexed addressing. For this method negative offsets from the current R12 can be used. The example above can be rewritten like this

```
1000 ! entry to a function with 2 numeric parameters
1010         BYT 40,55
1020 DEMO.   BSZ 0               ! DEMO(A,B)
1030 ! R12 stack on entry:
1040 ! -16     A (8 bytes)
1050 !  -8     B (8 bytes)
1060 ! R12 -> next free slot on R12 stack
```

```
1070 !
1080        LDMD R40,X12,OFF-16  ! load “A” into R40
1090                             ! without disturbing R12
...
1500        LDMD R50,X12,OFF-8   ! maybe later: get a copy of “B”
... A is still on the stack, at R12-16
... B is still on the stack, at R12-8
... R12 still points to the next free slot on R12 stack
1600        SBM R12,=20,0    ! finally: drop A, B by decrementing R12
1610        RTN
...
1800 OFF-8  DAD  177770   ! offset value  -8 (-10 octal)
1810 OFF-16 DAD  177760   ! offset value -16 (-20 octal)
```

Figure 10: Code fragment to copy parameters off the R12 stack using negative offsets.

Using zero or positive offsets allows placing local values onto the stack without using the push instruction. You must be careful when mixing with calls to system functions because these almost always transfer input and output via the R12 stack, of course using the current value of R12.

Another option would be to save R12 on entry to a subroutine and point the copy e.g. to the first parameter. Then R12 would be incremented to skip over any local variables. The parameters as well as the local variables would then be addressed by the copy of R12 and system routines can work as usual with R12. When leaving the subroutine you can restore R12 using the copy. This would be similar to how most high level languages work on the Intel x86 processor family with their SP and BP registers.

The only drawback is that the copy of the stack pointer takes a two byte register which must be reserved for this purpose.

```
1000 ! entry to a function with 2 numeric parameters
1010        BYT 40,55
1020 DEMO.  BSZ 0               ! DEMO(A,B)
1030 ! R12 stack on entry:
1040 ! R20 -> A (8 bytes)         (R20 as set in line 1080)
1050 !        B (8 bytes)
1060 ! R12 -> next free slot on R12 stack
1070 !        local variable 1
1075 !        local variable 2
1080        LDM R20,R12          ! copy current R12
1090        SBM R20,=20,0        ! point to parameter “A”
1100        ADM R12,=10,0        ! space for 2 local 8-byte variables
1110        LDMD R40,X20,PAR.1   ! load “A” into R40
1120        STMD R40,X20,VAR.1   ! store “A” into local variable 1
1130        LDMD R40,X20,PAR.2   ! load “B” into R40
1140        STMD R40,X20,VAR.2   ! store “B” into local variable 2

...  ! system routines push/pop on top of local variable area

2000 ! clean up before returning
2010                             ! drop parameters A and B and
2020        LDM  R12,R20         ! restore initial stack pointer
2030        RTN
...
5000 PAR.1  DAD  000000   ! offset value  0  parameter 1
5010 PAR.2  DAD  000010   ! offset value  8  parameter 2
5020 VAR.1  DAD  000020   ! offset value 16  local variable 1
5030 VAR.2  DAD  000030   ! offset value 32  local variable 2
```

Figure 11: Code fragment to work with local variable R12 stack using positive offsets.

JMPing around

The Capricorn CPU offers unconditional and conditional jump functions. However, these allow jumping over distances of ±128 bytes only. Often code can be rearranged to move the target address within reach. But sometimes it is necessary to bridge a larger distance.

There are several ways to accomplish unconditional long distance jumps. For conditional jumps you have to set up an intermediate stop.

Using “Trampolines”

This is the simplest way to solve the problem. You insert intermediate jump targets which just contain another `JMP` instruction. Thus the long distance `JMP` is performed by jumping from “trampoline” to “trampoline”. This is a simple and fast solution which consumes only a few bytes. However it leads to a high degree of “spaghettization” when used often or with multiple intermediate stops.

Using the Program Counter in R4-R5

The assembler ROM offers a pseudo Op-Code `GTO` which is useful to jump to fixed addresses, like seen in ROMs (or BPGMs with fixed load addresses).

Normally a BPGM has no fixed load address so that `GTO` cannot be used. However, in such a BPGM the same behavior can be simulated by the following sequence of operations:

```
        BIN               ! if not already set (required for ADMD below)
        LDM R20,=DEST     ! load the relative target address ...
        ADMD R20,=BINTAB  ! ... add load address of BPGM to make address absolute
        DCM R20           ! important: make PC one less because it will be
                          !            incremented by LDM
        LDM R4,R20        ! finally load R4-R5 which makes us GO!

... long distance > 128 bytes

DEST    BSZ 0     ! destination label
... continue
```

Abusing Subroutine Calls

Instead of using an unconditional `JMP` instruction one can also use a `JSB` subroutine call instruction. This instruction can cover the complete address space for its target, but there are no conditional variants of this instruction. It may take more cycles because the return address has to be set up before the jump is executed. As the `JSB` call also pushes this return address to the R6 stack, it is necessary to clean up the stack after jumping by dropping the return address like so:

```
        JSB DEST     ! use JSB as a long distance JMP
                     ! DEST must be an absolute address!
... long distance

DEST    BSZ 0        ! destination label
        POMD R20,-R6 ! drop 2-byte return address
... continue and do not use a RET instruction
```

This example works in a ROM (or rare BPGMs with fixed load addresses), where the absolute address of `DEST` is known at compile time.

However, with a slight modification this approach can also be used with an indexed call for position independent code in a binary program:

```
... headers, code and data
        LDMD R20,=BINTAB    ! load base address of BPGM
        JSB X20,RELDEST     ! call relative to BINTAB

... long distance > 128 bytes

RELDEST BSZ 0         ! destination label
        POMD R20,-R6 ! drop 2-byte return address
... continue and do not use a RET instruction
```

Abusing Return Instructions

Finally you can use the `RTN` instruction to jump to a nice place. In this case you place the target address onto the R6 return stack and execute a `RTN`. Again you need the absolute address.

```
        BIN                ! if not already set (required for ADMD below)
        LDM R20,=DEST      ! load the relative address of the target label ...
        ADMD R20,=BINTAB   ! ... and add the load address of the BPGM to this offset
        PUMD R20,+R6       ! load the address onto the return stack
        RTN                ! makes us return to the target (DEST)

... long distance > 128 bytes

DEST    BSZ 0      ! destination label
... continue
```

So we see that there are plenty ways to skin a `JMP`.

Typical Application of the Compare Instruction

Sometimes my brain had difficulties to link the proper compare instruction with my logical thinking. Therefore I created the following table (see HP-87 Assembler ROM Manual, 6-28):

Test	Instructions	Description
DR < AR	CMM DR,AR JNC LABEL	Carry flag should be 0 jump if DR < AR
DR ≥ AR	CMM DR,AR JCY LABEL	Carry flag should be 1 jump if DR ≥ AR
DR = AR	CMM DR,AR JZR LABEL	Zero flag should be 1 jump if DR = AR
DR ≠ AR	CMM DR,AR JNZ LABEL	Zero flag should be 0 jump if DR ≠ AR

Table 1: Summary of operators for comparisons.

Accessing a String Reference passed via the R12 stack

You can obtain the address and length by two pop operations:

```
POMD R46,-R12   ! 2-byte address to R46,R47
POMD R32,-R12   ! 2-byte length to R32-R33
```

Alternatively you can pop all 4 bytes at once:

```
POMD R44,-R12   ! 2-byte length to R44-R45, 2-byte address R46-R47
```

In case of the HP 86/87 addresses are 3 bytes long.

```
POMD R45,-R12   ! 3-byte address to R45-R47
POMD R32,-R12   ! 2-byte length to R32-R33
```

Alternatively you can pop all 5 bytes at once:

```
POMD R43,-R12   ! 2-byte length to R43-R44, 3-byte address R45-R47
```

Accessing a Number by Reference passed via the R12 stack

This is just a clarification of the terse Assembler ROM manual. A function's parsing routine may call the REFNUM system function when it requires the address of a numeric parameter and not only its actual value. This is required e.g. by a SWAP function which must read the value and replace it by another value. The REFNUM function pushes a token 21 onto the R12 stack if the content of R14 is 1. Later the runtime executive interprets the token and leaves an address/name pair on the R12 stack. The address is the address of the value of the variable, not of the name field. As the variable may contain a Real, Integer or Short we would have to interpret the 16-bit name field to select the proper size. If we want to use a system function to fetch the content of the variable as an 8-byte quantity we should instead use the FETSV system function. However, this function asks for the address of the name field of the variable. Therefore we have to decrement the address of the value field by two to obtain the proper address before calling FETSV.

Note also that the address is either absolute (in CALC mode) or relative (in RUN mode) and may need modification by subtracting the content of FWCURR.

```
! PARSE routine used REFNUM and Executive placed name and address on the R12 stack
...
! RUNTIME routine:
      POMD R54,-R12       ! R54-R55: 2-byte address of value, R56-R57: 2-byte name
      LDM R66,R54         ! transfer to 2-byte register for DCM
      DCM R66
      DCM R66             ! (absolute) address of name field
      CMB R16,=1          ! CALC mode? FETSV needs this absolute address
      JZR S.1
      SBMD R66,=FWCURR    ! RUN mode: make address relative
S.1   JSB =FETSV          ! takes address of name in R66
      RTN
```

Functions for Arithmetic Operations using Registers

This is just a clarification of the terse Assembler ROM manual. These functions do not expect their parameters on the R12 stack but in registers. They are often useful for starting a chain of numeric operations. The result is pushed on the R12 stack as well as held in a register.

```
SUB10  DAD 52137
Operation: R50 - R40
Input: R50, R40
Output R40 and R12 stack
```

```
ADD10  DAD 52233
Operation: R50 + R40
```

```
Input: R50, R40
Output R40 and R12 stack
```

```
DIV10  DAD 51644
Operation: R50 / R40
Input: R50, R40
Output R40 and R12 stack
```

```
MPY10  DAD 52562
Operation: R50 * R40
Input: R40, R50
Output R40 and R12 stack
```

```
CHS10  DAD 52105
Operation: R40 → -R40
Input: R40
Output R40 and R12 stack
```

```
COMFLT DAD 32612
Operation: test if R40 < R50
Input: R40, R50
Output R50 = R50 - R40, E=1 if R40 < R50, E=0 if R40 ≥ R50
```

10. Using Functions from System ROM 0 in a ROM

The operating system of the HP 85 is distributed over four ROMs. One of these, system ROM #0 is, like all external ROMs, bank switched to the address window between 60000_o and 77777^o. This leads to a problem if our bank switched ROM wants to call a function in this system ROM. The call requires switching ROM #0 into the address space of our ROM and a RTN would have to switch back to our calling ROM. The same problem occurs, when a function in system ROM #0 calls a function in our ROM. For this purpose the system offers the `ROMJSB` and `ROMRTN` functions.

`ROMRTN` is used instead of a RTN to return to system ROM #0. It is required in all parsing routines of our ROM because these are called from system ROM #0.

`ROMJSB` is used to call a function in the system ROM #0 and to return to our ROM. The function catalogue in the Assembler manual contains a field ROMJSB which indicates whether a function has to be called via `ROMJSB` or not. Besides these documented functions other functions may also have to be called through this function because they may switch to system ROM #0 in their calling hierarchy.

The following table summarizes the functions marked in the Assembler ROM manual as needing `ROMJSB`. I have added a few additional functions which I also found needing to walk over this bridge.

```
! Example for returning from our parse routine
... do something with the command line
     JSB =ROMRTN   ! return to system parser routine in ROM #0
```

```
! Example for calling a function in system ROM #0 from a ROM
… prepare data as usual
     JSB =ROMJSB   ! call switching routine in ROM #0
     DEF ATN2      ! address of function to call
     BYT 0         ! ROM number
... test return values, flags etc. as usual
```

The same mechanism applies if you want to call a function contained in another external ROM. Normally you should try to avoid this, because it introduces a hard dependency on another ROM, but sometimes it is required. For example the *Extended Mass Storage* ROM uses functions in the basic *Mass Storage* ROM – as the name implies, it extends its functions. Both ROMs can only work as a pair.

Function	Type	Function	Type	Function	Type	Function	Type	Function	Type
DMNDCR	Parse	ATN2.	Runtime	CONBIN	Utility	ASIGN.	Tape	COPY.	CRT
G$N+NN	Parse	BEEP.	Runtime	CVNUM	Utility	CREAT.	Tape	CRTPUP	CRT
G$N	Parse	COMMA$	Runtime	DRV12.	Utility	P+ARAY	Tape	LABEL.	CRT
G01N	Parse	COMMA.	Runtime	PAPER.	Utility	PRNT#.	Tape	PEN.	CRT
G012N	Parse	CONCA.	Runtime	PRDVR1	Utility	PURGE.	Tape	PENUP.	CRT
G120R4	Parse	DEFA+.	Runtime	STOST	Utility	R#ARAY	Tape	SCALE.	CRT
G0OR2N	Parse	DEFA-.	Runtime	STOSV	Utility	READ#.	Tape		
GCHAR	Parse	DEG.	Runtime						
G1OR2N	Parse	DISP.	Runtime						
GET)	Parse	EQ.	Runtime						
GET$N?	Parse	GEQ.	Runtime						
GET1N	Parse	GRAD.	Runtime						
GET1$	Parse	GR.	Runtime						
GET4N	Parse	ICOS	Runtime						
GET2N	Parse	ITAN	Runtime						
GETCM?	Parse	ISIN	Runtime						
GETCMA	Parse	LEQ.	Runtime						
GETPAR	Parse	LT.	Runtime						
GETPA?	Parse	OFTIM.	Runtime						
INTEGR	Parse	PRINT.	Runtime						
NARRE+	Parse	PRLINE	Runtime						
NARREF	Parse	PRNT#$	Runtime						
NUMCON	Parse	PRNT#N	Runtime						

NUMVA+	Parse	RAD.	Runtime
NUMVAL	Parse	READ#$	Runtime
PUSH1A	Parse	READ#N	Runtime
PUSH32	Parse	SCRAT.	Runtime
PUSH45	Parse	RNDIZ.	Runtime
REFNUM	Parse	SEMIC.	Runtime
SCAN	Parse	SEMIC$	Runtime
SCAN+	Parse	TIME.	Runtime
SEQNO+	Parse	UNEQ.	Runtime
SEQNO	Parse	VAL.	Runtime
SMLINT	Parse	WAIT.	Runtime
STRCON	Parse	YTX5	Runtime
STREX+	Parse	EXP5[3]	Runtime
STREXP	Parse	LN5[3]	Runtime
STRREF	Parse		
TRY1N	Parse		
UNQUOT	Parse		

Table 2: Functions in System ROM #0 to be called via ROMJSB.

11. Some Useful HP RPL Programs

The assembler programmer often needs to convert decimal and octal values into specific representations used by the Series-80 machines. For example arithmetic operations require their input in BCD form with specific forms of negative numbers and exponents. Also addresses have to be converted from individual octal bytes to 16-bit values.

The following RPL programs have been written for the HP50g but should also work on other RPL machines of the 48G calculator family. They grew over time and are far from being elegant or optimized, but they do the job.

ADR85

This program calculates a 16-bit octal address from two octal bytes as read from memory of a HP-85 machine (low order byte, high order byte). The returned 16 bit address can be used in the debugger to inspect e.g. the content of the R12 stack or the content of variables.

Note that the HP-86/87 may use 3-byte addresses in their extended memory. These are not handled by this program.

[3] In addition to the Assembler ROM manual.

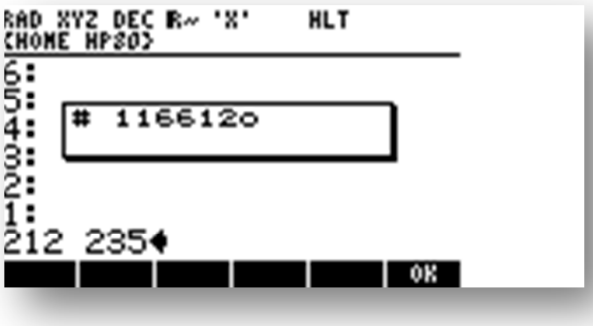

Figure 12: Given two octal values ADR85 shows the combined address in octal.

```
« DEPTH 2. <
  IF
  THEN
"Need 2 Oct Addresses"
  ELSE HEX 1. 2.
    FOR I R→I "# " SWAP
"o" + + STR→ SWAP
    NEXT 256. * + OCT
  END →STR DUP FONT→
SWAP FONT6 →FONT MSGBOX
→FONT
»
```

BIN80

Converts an integer number into the internal Series-80 binary representation. The number on the stack is converted to the internal binary representation which can be used in BIN mode in the CPU registers.

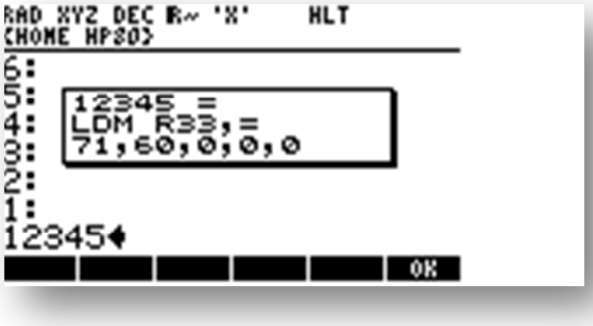

Figure 13: BIN80 converts the given integer into a 5 byte binary number for arithmetic operations in BIN mode. The output is given in form of octal bytes.

```
« DUP TYPE 10. ==
  IF
  THEN B→R
  END DUP DUP 0. <
  IF
```

```
  THEN 65536. +
  END HEX R→B →STR
DEC DUP SIZE 1. - 3.
SWAP SUB DUP SIZE 2.
MOD 1. ==
  IF
  THEN "0" SWAP +
  END "00000000" SWAP
+ DUP SIZE DUP 9. -
SWAP SUB 9. 1. OCT
  FOR I DUP I I 1. +
SUB "h" + "# " SWAP +
STR→ →STR DUP SIZE 1.
- 3. SWAP SUB "," +
SWAP -2.
  STEP DEC DROP + + + +
DUP SIZE 1. - 1. SWAP
SUB "LDM R33,=" 10. CHR
+ SWAP + SWAP R→I " =" +
10. CHR + SWAP + DUP
FONT→ SWAP FONT6 →FONT
MSGBOX →FONT
»
```

INT80

Converts a given integer number into the Series-80 BCD representation. The number on the stack is converted to the representation used by a tagged integer or a variable declared as integer. These numbers are used by the BASIC system and most system routines.

Figure 14: The given integer number is converted by INT80 into the bytes of a tagged integer to be loaded into a register or to be used for an Integer variable. The output is given in BCD as well as in form of octal bytes.

```
« DUP TYPE 10. ==
  IF
  THEN B→R
  END DUP DUP 0. <
  IF
  THEN 1000000. +
  END R→I →STR "00000"
```

```
SWAP + DUP SIZE DUP 5. -
SWAP SUB 5. 1.
  FOR I DUP I I 1. + SUB
"C," + SWAP -2.
  STEP DROP + + DUP SIZE
1. - 1. SWAP SUB
"LDM R34,=377," 10. CHR +
SWAP + SWAP R→I " = -1" +
10. CHR + SWAP + DUP DUP
DUP SIZE DUP 11. - SWAP
SUB STR→ 6. →LIST 1. 5.
  FOR I DUP I I SUB SWAP
2. STEP DROP + + OCT
  « "#" SWAP R→I + "h" +
OBJ→ B→R R→B →STR DUP
SIZE 1. - 2. SWAP SUB
  » MAP DEC OBJ→ DROP +
+ 10. CHR "=" + SWAP + +
DUP FONT→ SWAP FONT6
→FONT MSGBOX →FONT SWAP
0. ==
  IF
  THEN -105. CF
  END
»
```

FLT80

Converts a given floating point number into the Series-80 BCD representation. The number on the stack is converted into its BCD-8 representation as used by most arithmetic system routines or for storage in a variable declared as Real.

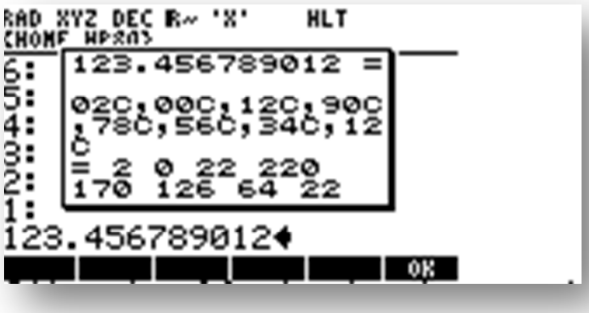

Figure 15: The given floating point number is converted by FLT80 into the internal BCD representation. The output is given in BCD as well as in form of octal bytes.

```
« -105. FS? SWAP -105.
SF STD DUP DUP SIGN 1. -
2. / ABS 9. * →STR 1. 1.
SUB SWAP ABS DUP LOG
FLOOR DUP ALOG ROT SWAP
```

```
/ 100000000000. * →STR
DUP 11. 12. SUB "C," +
2. PICK 9. 10. SUB "C,"
+ + 2. PICK 7. 8. SUB
"C," + + 2. PICK 5. 6.
SUB "C," + + 2. PICK 3.
4. SUB "C," + + SWAP 1.
2. SUB + "C" + SWAP DUP
0. <
  IF
  THEN 1000. +
  END →STR "000" SWAP +
DUP SIZE 1. - DUP 2. -
SWAP SUB DUP 2. 3. SUB
"C," + SWAP 1. 1. SUB 4.
ROLL + "C," + + SWAP +
SWAP " = " + 10. CHR +
SWAP + DUP DUP DUP SIZE
SWAP "=" POS 2. + SWAP
SUB STR→ 16. →LIST 1. 15.
  FOR I DUP I I SUB SWAP
  2. STEP
DROP + + + + + + + OCT
  « "#" SWAP R→I + "h" +
OBJ→ B→R R→B →STR DUP
SIZE 1. - 2. SWAP SUB
  » MAP DEC OBJ→ DROP +
+ + + + + + 10. CHR "="
+ SWAP + + DUP FONT→
SWAP FONT6 →FONT MSGBOX
→FONT SWAP 0. ==
  IF
  THEN -105. CF
  END
»
```

12. References

[1] Knuth, “The Art of Computer Programming, Volume 2, 2nd edition, 1981.

[2] Milton Abramowitz, Irene Stegun, “Handbook of Mathematical Functions”, United States Department of Commerce, National Bureau of Standards, June 1964.

[3] AVR204 “BCD Arithmetics”, Application Note, Amtel Corp., 2003.

[4] Peter Henrici, “Computational Analysis with the HP-25 Pocket Calculator”, Wiley&Sons, 1977.