HRAST-BASIC [32K v1.01 & 48K v2.24]

https://www.hrastprogrammer.com/

1. Introduction

1.1. What is this all about?

It makes me very happy to announce the availability of my HRAST-BASIC high-level programming language, interpreter and development environment for HP-48G/G+/GX, HP-49G and HP-49G+/50G calculators. This isn't really an ordinary BASIC as used to be on various home and pocket computers. It is inspired by BASIC but in many aspects it is more like Pascal, C, COMAL or Forth (internally). This is the culmination of my work as far as HP/calculator development is concerned and I must admit that I am very proud of it. The final result is "the BASIC as I always wanted it to be" because I was never really satisfied with the choice of development tools available on HP calculators.

As far as I know, this is the first and only BASIC interpreter made for any HP calculator after HP-71B released in 1983. And this is the first and only BASIC interpreter developed outside Hewlett-Packard for any HP calculator ever. Those facts make me even more proud and happy!

The first version was HRAST-BASIC 32K because it should work on the basic HP-48G with 32K RAM only and after installing the interpreter you can still (although barely) allocate a reasonable 8K of RAM for BASIC program and data. The feature set is fixed and no new features will be implemented for 32K version because I want it to stay under 20K. The final version is HRAST-BASIC 48K which needs at least 48K of free RAM for normal work. Although I consider 48K version finished as well, there are still a few non-essential things left to be implemented if/when time permits.

Three main goals during the development were to make it as powerful as possible, to make it as fast as possible and to use as little memory as possible. Those goals don't usually go very well together so every time I had to choose between speed and memory - I choosed speed. Furthermore, I wanted everything to be limited by the available memory only - there is no fixed expression stack, no fixed GOSUB/FOR/NEXT and PROC/FUNC stack, expressions can be as complex as you want, you can call subprograms/procedures/functions as deeply as you want, you can have as many nested control structures as you want, etc, etc. This system is very complex but works great and the result is very powerful and fast BASIC interpreter, exceeding the speed and feature-set of even the most powerful home computers like, for example, Acorn BBC B which is well known for its great and very fast BASIC. And it is definitely much faster and much more powerful than all other BASICs on pocket computers and calculators. I made a lot of benchmarks and I will present some results here. The first benchmark is "8-Queens" problem as described on the following site:

http://www.hpmuseum.org/cgi-sys/cgiwrap/hpmuseum/articles.cgi?read=700

HRAST-BASIC running on HP-48GX finishes this benchmark in 16.4 seconds.
HRAST-BASIC running on HP-49G+/50G finishes this benchmark in 7.2 seconds.

This makes it much faster than all other BASICs on this page except QBasic, Power, Turbo and Quick BASIC running on HP-200LX. The benchmark program is here, so you can get a picture how an optimized program for HRAST-BASIC looks like (various strange details from this program will be explained throughout the manual):

```
WATCH CALL QUEEN,8 ? TICKS
PROC QUEEN(INTEGER N) INTEGER A[N+1],D,S=1,X,Y A[1]=N
FOR X=2 TO N S> A[X]=N FOR Y=X-1 D=A[X]-A[Y]
IF NOT D OR X-Y=ABS D@ A[X]< WHILE NOT A[X]@ X< A[X]< ENDWHILE S> Y=X
NEXT NEXT ? A[1],A[2],A[3] ? A[4],A[5],A[6] ? A[7],A[8]
```

A little slower but more clear version of the same benchmark:

```
WATCH CALL QUEEN,8 ? TICKS
PROC QUEEN(INTEGER N)
INTEGER A[N+1],D,S=1,X,Y: A[1]=N
FOR X=2 TO N: S=S+1: A[X]=N
FOR Y=X-1: D=A[X]-A[Y]
IF D=0 OR X-Y=ABS D@ {
_ A[X]=A[X]-1
_ WHILE NOT A[X]@ X=X-1: A[X]=A[X]-1: ENDWHILE
_ S=S+1: Y=X
}
NEXT: NEXT
? A[1],A[2],A[3]
? A[4],A[5],A[6]
? A[7],A[8]
```

Another interesting benchmark is the Sieve of Eratosthenes (or Sieve benchmark for short). This is a simple, ancient algorithm for finding prime numbers up to any given limit. Here is an example of this algorithm implemented in plain unstructured BASIC which will generate 500 prime numbers starting from the number 3:

http://www.retroprogramming.com/2010/01/8-bit-home-computer-benchmarks.html

This is a direct conversion to HRAST-BASIC:

```
ARRAY 1 WATCH
10 W=500: REAL F[W]: P=1: A=3
20 F[P]=A: P=P+1: IF P>W ? TICKS STOP
30 A=A+2: X=1
40 S=A/F[X]: IF S=INT(S) GOTO 30
50 X=X+1: IF X<P AND F[X]*F[X]<=A GOTO 40
60 GOTO 20
```

The execution time is 150.9 seconds on HP-48GX and 104.1 seconds on HP-49G+/50G. So, both HP-48GX and HP-49G+/50G versions of this benchmark are significantly faster than the ones running on most home computers. Of course, this is very good per se but doesn't really show a full potential of HRAST-BASIC because there are a lot of unnecessary balast here. After incorporating some of HRAST-BASIC (unique) features I came up with this one:

```
ARRAY 1 WATCH
INTEGER W=500,X REAL F[W],A=3
F[1]=3 FOR P=2 TO W A+2 X=1
4 IF NOT MOD(A,F[X]) AGAIN
X> IF X<P AND ^F[X]<=A GOTO @4
F[P]=A NEXT ? TICKS
```

Now we are talking! The execution time is 75.4 seconds on HP-48GX and 49.7 seconds on HP-49G+/50G which is much, much faster. And after converting it into a pure INTEGER version by changing the second line to:

```
INTEGER W=500,X,F[W],A=3
```

... the execution is even faster, with 62.9 seconds on HP-48GX and 29.7 seconds on HP-49G+/50G. So, it is always a good idea to spend some time to rework a particular BASIC program, optimize it, use all available advantages of a particular interpreter architecture and squeeze as many cycles as possible ;-)

Note: The above Sieve benchmarks are using AGAIN and ARRAY statements which are available in 48K version only. Here are modified 32K versions:

```
WATCH
10 W=500: REAL F[W+1]: P=1: A=3
20 F[P]=A: P=P+1: IF P>W ? TICKS STOP
30 A=A+2: X=1
40 S=A/F[X]: IF S=INT(S) GOTO 30
50 X=X+1: IF X<P AND F[X]*F[X]<=A GOTO 40
60 GOTO 20
```

```
WATCH
INTEGER W=500,X REAL F[W+1],A=3
F[1]=3 FOR P=2 TO W
3 A+2 X=1
4 IF NOT MOD(A,F[X]) GOTO @3
X> IF X<P AND ^F[X]<=A GOTO @4
F[P]=A NEXT ? TICKS
```

I also executed all PCW home computer benchmarks (slightly modified for HRAST-BASIC):

```
REAL K,M[5] RAD               REM BM5                 REM BM7
                              K=0                     K=0
REM BM1                       5 K>                    7 K>
FOR I=1 TO 1000               A=K/2*3+4-5             A=K/2*3+4-5
NEXT                          GOSUB @9                GOSUB @9
                              IF K<1000 GOTO @5       FOR I=0 TO 4
REM BM2                                               M[I]=A
K=0                           REM BM6                 NEXT
2 K>                          K=0                     IF K<1000 GOTO @7
IF K<1000 GOTO @2             6 K>
                              A=K/2*3+4-5             REM BM8
REM BM3                       GOSUB @9                K=0
K=0                           FOR I=0 TO 4            8 K>
3 K>                          NEXT                    A=K^2
A=K/K*K+K-K                   IF K<1000 GOTO @6       B=LN K
IF K<1000 GOTO @3                                     C=SIN K
                                                      IF K<1000 GOTO @8
REM BM4
K=0                                                   9 RETURN
4 K>
A=K/2*3+4-5
IF K<1000 GOTO @4
```

REAL results for BM1..BM8 on HP-48GX are: 0.24s, 2.78s, 9.09s, 8.82s, 9.33s, 11.85s, 19.75s, 27.43s
Average = 11.16s

INTEGER results for BM1..BM8 on HP-48GX are: 0.24s, 1.78s, 5.58s, 5.64s, 6.14s, 8.66s, 15.83s, 26.64s
Average = 8.81s

REAL results for BM1..BM8 on HP-49G+/50G are: 0.07s, 1.97s, 6.03s, 6.08s, 6.41s, 7.75s, 11.70s, 17.92s
Average = 7.24s

INTEGER results for BM1..BM8 on HP-49G+/50G are: 0.07s, 0.73s, 2.37s, 2.49s, 2.81s, 3.99s, 7.49s, 16.83s
Average = 4.60s

All in all, not bad for an ancient 4-bit CPU and the emulation of an ancient 4-bit CPU ;-)

One of the reasons for such performance is because I avoided dynamic/heap memory allocation, fragmentation and garbage collection at all costs. Dynamic memory allocation needs copying/moving and copying/moving kills speed and performance. I hate garbage collectors, they are totally unpredictable. And if you don't leave garbage than you don't have to collect it, do you? That's why HRAST-BASIC works a lot like FORTH internally. Actually, at first I intended to develop HRAST-FORTH but I decided not to do it because I concluded that a higher-level language would be much more appropriate for practical use. FORTH is a great language but a little cumbersome to work with various data types: floating point numbers, complex numbers, strings, arrays, matrices, etc. But a lot of code I developed for HRAST-FORTH made it into HRAST-BASIC, although I may eventually continue working on HRAST-FORTH standalone sometime in the future.

Note #1: All times are for 48K version. The times for 32K version could deviate a little in some cases.

Note #2: The above benchmarks show some interesting details not directly related to HRAST-BASIC. Namely, HP-49G+ and HP-50G are not always twice as fast compared to HP-48GX and HP-49G! This is very interesting and surprisable discovery. Obviously, some Saturn CPU instructions are emulated more efficiently than the others. Or, to be more precise, operations on A (Address) and B (Byte) fields of 64-bit Saturn CPU registers are faster than operations on larger fields like M, WP and W because INTEGER versions of those benchmarks are always ~2x faster on HP-49G+/50G while REAL ones aren't as fast. And INTEGER code relies mostly on A field while REAL code uses larger fields like M (Mantissa) and W (Whole Register). There is no other explanation for such behavior because interpreter code is the same for all versions.

1.2. Disclaimer

Please, keep in mind that HRAST-BASIC is not a free software. I put a lot of my own time, knowledge and energy into this project, so I am expecting some compensation for those efforts. That's why I am releasing HRAST-BASIC as donation-ware. It is not crippled or limited in any way and you can use it freely for 32 days. If you decide to use it after 32 days you have to make a fixed donation of 48 USD to hrastwerk@hrastprogrammer.com PayPal account. This amount covers the installation of both 32K and 48K versions on as many calculators as you want. It doesn't cover commercial or group usage (a single donation for the whole classroom, for example) - contact me by mail and we'll arrange something. HRAST-BASIC is not a commercial product and I don't have a business which can sell it to you. So, I accept PayPal donations only - if this is not OK with you then, please, don't donate and don't use HRAST-BASIC!

*** Individuality, not conformity! ***

As with all my software, HRAST-BASIC has that "HrastProgrammer's" way-of-working and look-and-feel which is not appealing to everyone because I developed it for myself in the first place. I didn't have to make it compatible with any particular existing BASIC, so I was free to design it as I wanted to. The feature set is fixed and I will not change anything in this regard, no matter how some features could look strange or quirky at first sight. I am sharing HRAST-BASIC because such development tool/environment didn't exist before and it could possibly be helpful to someone else as well. But, if you don't like it or it doesn't work for you for whatever reason and don't want to use it after (or before) 32 days then just delete it, simple as that. Donations cannot be returned so think twice before donating. I didn't protect this software in any way and I will rely on the honesty of the users only :-)

Important: This is a reference manual, not a book teaching BASIC language - the reader is expected to have a good programming knowledge in order to use HRAST-BASIC and to understand the content of this manual. The manual is kind-of "work in progress" and some things may be missed from the initial version, or may not be fully explained. Please, report all errors to me so I can correct them. Also note that english is not my native language, so I will appreciate very much all corrections in regard to grammar, spelling, etc.

Note: HRAST-BASIC has no connection to the so-called "HP BASIC" available on HP-49G and HP-49G+/50G which is not BASIC at all, just an awkward algebraic wrapper over UserRPL. Furthermore, the interpreter doesn't use RPL operating system except some SysRPL code during the initialization and machine language math routines - there is really no need to reinvent the wheel here.

1.3. Installation

The installation is very simple and is the same for all calculators. First, send the appropriate HRAST file (HRAST48, HRAST49 or HRAST50) containing the interpreter to the calculator and store it to a variable. Then send the appropriate RAMH file (RAMH48 for HP-48G/G+/GX and RAMH49 for HP-49G and HP-49G+/50G) to the calculator and store it to (another) variable. Put the number of BASIC kilobytes (BTW, no "kibibytes" and similar crap, please) needed and execute RAMH. This will create HRAST-BASIC environment in form of an character string object of the required size (for example, 8 RAMH to create an 8K environment, minimum is 2K). Store this environment to the HRAM variable and start the interpreter by executing the variable where HRAST file has been stored. The interpreter will look for the HRAM variable in the current directory and (if it finds one and it was not already initialized) will initialize it and continue execution. If it doesn't find the HRAM variable an "Undefined Name" error will be reported and the interpreter will abort execution.

Please, note that bugs are inevitable in such a complex piece of software, especially under various "borderline" conditions, so watch for them and report them to me if you find any. I just hope I didn't make such a terrible mistakes like typing x=18.9 on Atari ST BASIC causing "function not yet done System error #%N, please restart" error ... So far, the interpreter has been tested on the following calculators: HP-48GX ROM version R, HP-49G ROM version 1.19-6, HP-49G+ ROM versions 1.23/2.15 and HP-50G ROM version 2.15. I don't intend to support other ROM versions and various non-official ROMs or alternate firmwares. I reserve the right to mark a particular bug as "feature" or "by design" if I judge that something isn't really a bug or even if it is a bug but could somehow be useful. I also reserve the right to fix some bugs in 48K version only, due to the lack of space in 32K version.

In order to save space, the interpreter is using one unsupported ROM font table entry on HP-48G/G+/GX. So, there is a slight possibility that something can go wrong with different ROM versions - if something like this occurs then, please, report this to me. On HP-49G and HP-49G+/50G the system font table stored in RAM is used. The interpreter can work with 8-pixels height fonts only, otherwise an "Font error" will be reported and the execution will be aborted.

4

2. General

Although HRAST-BASIC is, well, BASIC it has a lot of unique features which makes it stand apart from other BASICs around. As seen from the above "8-Queens" example, it doesn't have line numbers, it is procedural, it supports variable declarations, it has various high level program structures, etc, etc. Lots of language elements and features are designed in such a way to support the main goals: speed and low memory consumption.

2.1. Syntax

In general, HRAST-BASIC has moderately strict syntax modeled after what I consider a "moderately strict syntax". All statement, function and variable names must be separated from each other by a space or some other non-alphanumeric character. More statements can appear on the same line and colon separator is not needed in most cases. The comma can be used instead of a colon to separate assignment statements. In order to save code space the syntax check is splitted and executed in two phases: after entering and tokenizing the command/program line and during the execution. If an syntax error has been detected during tokenization the interpreter will show the error and position the cursor at the character where error has been detected. If an error has been detected during execution the interpreter will stop the execution and position the cursor at the line and after character where error has been detected. During the syntax check the tokenizer will eliminate all spaces from the program line and, in the case of an syntax error, will show you back the detokenized line so you can see what the interpreter understood, compare this to what it should have understood and make the necessary corrections. You can use vertical line or underscore (|_) for indentation should you need it.

Examples:

```
A=1: B=2: C=3
A=1,B=2,C=3
A=1 B=2 C=3
FOR Y=X-1: D=A[X]-A[Y]
FOR Y=X-1 D=A[X]-A[Y]
| FOR I=1 TO 5
|__ DISP I
| NEXT
```

2.2. Labels

HRAST-BASIC doesn't have the concept of line numbers. This is such an archaic concept that I never actually liked and decided not to support it. It has numeric and string labels instead. Numeric labels can range from 0..255 (1-byte label) and from 256..65535 (2-byte labels). String labels can be up to 7 characters in length. Labels can be placed at the beginning of the line only. The interpreter doesn't check for duplicates so you can have the same numeric/string label appear more than once inside a program. This doesn't have much sense in 32K version because only the first label will be found during the search. In 48K version you can use START statement to change label search start line. The statements which support labels are GOTO, GOSUB, RESTORE, EDIT and LIST.

Numeric label example:

```
K=0
10 K=K+1
A=K*K/K+K-K
IF K<1000 GOTO 10
```

String label example:

```
K=0
"10X" K=K+1
A=K*K/K+K-K
IF K<1000 GOTO "10X"
```

You can simulate the usual BASIC line numbering by using labels, but I don't encourage this because it doesn't have any practical sense:

```
10 K=0
20 K=K+1
30 A=K*K/K+K-K
40 IF K<1000 GOTO 20
```

2.3. Accelerators

Accelerators are very important HRAST-BASIC concept. In the above examples each time the interpreter execute GOTO statement it has to search for a label starting from the beginning of the program. Putting an accelerator after GOTO reserves some space where the destination address will be saved once the particular label has been found, so the interpreter won't have to search for the label each time. An accelerator is denoted by the monkey @ symbol. Many statements support accelerators - this will be noted for each statement where appropriate.

Examples:

```
K=0                              K=0
10 K=K+1                         "10X" K=K+1
A=K*K/K+K-K                      A=K*K/K+K-K
IF K<1000 GOTO @10               IF K<1000 GOTO @"10X"
```

2.4. Data Types

HRAST-BASIC supports the following native data types: INTEGER (20-bit), REAL15 (15-digit mantissa, 5 digit exponent), REAL12 (12-digit mantissa, 3-digit exponent), COMPLEX15 (15-digit mantissa, 5 digit exponent), COMPLEX12 (12-digit mantissa, 3-digit exponent) and STRING. All floating point calculations are performed using the REAL15 type and the results can be stored without loss of precision. Floating point values are always converted to REAL12 for display and when converting to strings. Also, REAL15 arguments are rounded to 12 digits when executing relational operators so, for example, $1/7*7=1$ evaluates to True. Some numeric data types conversions are executed automatically during the expression evaluation: INTEGER and REAL12 are converted to REAL15 whenever REAL15 come into play, INTEGER/REAL15/REAL12 are converted to COMPLEX15, etc. REAL/COMPLEX numbers are automatically converted to INTEGERs in statements and functions which expect INTEGER arguments (only lowest nibble or byte could be used in some cases, depending on the particular function or statement).

COMPLEX numbers must be entered as a numeric pair inside square brackets. 32K version of HRAST-BASIC supports native COMPLEX numbers evaluation for all basic operations, including SQR, while 48K version supports COMPLEX arguments for all (transcendental and other) functions and operators where this does have sense.

STRINGs can be up to 255 characters in length. The index of the first character in the string is 1. The conversion between string and numeric values is not automatic.

Examples:

```
123
123.456
[123,456]
[-123.456,789E5]
"123*456"
```

2.5. Variables

Variable names are always in uppercase and can be from 1 to 7 characters in length. The first character must be a letter followed by alphanumeric characters or underscores. One-letter variables are prefered where possible because they need one byte less for their internal representation. Contrary to most other BASICs out there accessing all variables require exactly the same time, no matter if it starts with A or Z, etc. This is because all variables have an internal index and link to their physical location. That's the reason why variables can exist only inside the context of a program.

When executing the RUN command the interpreter first builds the indexes and links for all variables inside the program. This process (called "program compilation") can take some time in case of very large programs but is a small price to pay in order to have very fast variable access. Due to the various optimizations, compilation on 48K version could be up to 2.5x faster compared to 32K version, depending on the particular program (compiling a big monolithic program needs more time than compiling a nicely structured program with lots of procedures and functions).

Each procedure and function has its own variables separated from the main program and other procedures/functions. There is a total maximum of 3276 different variables per single program. All variables are equal meaning that string variables don't have any special treatment and don't need $ at the end of their names (I never liked that $ thing).

The variables can be created in two ways: implicitly (using the assignment statement) or explicitly (using various declaration statements). If created implicitly their type is determined by the type of the evaluated expression. Explicit variable declarations can optionally include an initialization expression. LET statement is optional.

6

Once created, the variable type cannot be changed during the execution of the program, neither can a single variable be destroyed. When storing the value to a variable the interpreter does an automatic conversion between numeric types as needed. The conversion between string and numeric variables/values is not automatic. When declaring STRING variables the maximum length can be set inside the parenthesis after the variable name. The default maximum length for declared string variables without the explicit length is 22 (the width of the screen). When creating string variables implicitly the maximum size will be determined by the length of the evaluated (STRING) expression.

As already said in the introduction - all variables are created statically and there is no garbage collector inside, so all strings are static as well, and cannot be destroyed or enlarged once created. Strings shorter than maximum length can be stored to a variable, of course, but longer strings will be truncated automatically.

Examples:

```
INTEGER I,J=5,K=10,N=J+K
A=123.456,E1="123*456"
REAL X=1E25,TEST1
STRING S="ABCDEF",S2,SY(50)
C=[1.23E5,-666]
COMPLEX D=2*C
```

2.6. Arrays

Arrays can have up to 15 dimensions with elements of any of the above types (all elements of the same type, of course). Array indexes and dimensions must be enclosed inside a square brackets. For string arrays the maximum length can be set inside the parenthesis after the name and dimensions. All variables have equal treatment meaning that you cannot have a scalar and array variable with the same name. There is no DIM statement, all array declarations go through the regular declaration statements.

Examples:

```
INTEGER Q[5,5],ARR[77]
REAL A[2,3,4,5,6],R5[5]
STRING S[10],T[4,4](8)
COMPLEX C[9],CC[9,9]
```

Arrays are zero-indexed by default and this is the only option with 32K version. In 48K version array base index can be set using ARRAY statement. The number of indexes when accessing array elements can be less than number of dimensions so you can, for example, treat matrix elements as elements of an one-dimensional array like in the following example:

```
REAL A[4,4]
A[1,3]=123
DISP A[1,3]
123
DISP A[7] ... A[7]=A[1*4+3]=A[1,3]
123
```

48K version has the complete REAL/COMPLEX matrix engine built in. Matrix operations are based on the RPL engine, so only REAL12 and COMPLEX12 arrays/matrices are supported. All arrays/matrices which will be processed through the matrix engine must be declared using MAT REAL, MAT SHORT, MAT COMPLEX or MAT CSHORT statements.

2.7. Structured Programming

HRAST-BASIC has a complete set of enhanced control flow structures (including IF/THEN/ELSE, REPEAT/UNTIL, WHILE/ENDWHILE, WHEN/UNLIKE/ENDWHEN, EXIT/LOOP/LEAVE, CASE/OF/OTHER/ENDCASE, SKIP, code blocks, etc.) and a full implementation of procedures and functions with value/reference parameter passing, local declarations, return value for functions and even the ability for recursive procedure/function calls. There can be up to 127 procedures/functions inside a program. Procedure/function declarations must apper at the beginning of the line. The names are common for all procedures/functions so you cannot have a procedure or function with the same name. The interpreter will check for duplicates during program compilation and will raise an error if a duplicate has been found.

[32K only] There are no global variables in 32K version and you cannot access variables from the main program - only local variables are visible inside a procedure/function and everything from the main program should go through parameters (except when using an advanced READ/DATA technique described later).

[48K only] There are two types of variables in 48K version: global and local. Global variables are like COMMON variables in FORTRAN - they are visible accross main program and all procedures/functions, while local variables are visible only in main program or procedure/function where they are defined.

Global variables are denoted by ~ (tilde) character prior to the variable name. For example, A is local variable while ~A is global variable (those are two completely different variables). All global variables should be declared and values assigned to them inside a main program. Global variable declarations and assignments to unassigned global variables in procedures and functions are not allowed. It is a good programming practice to use local variables where possible. Among the other things, compiling a program with tons of global variables will need more time. Use global variables sparingly and only for values which are shared between main program and procedures/functions, like various constants etc. Do not use global variables to return results from procedures/functions if not really necessary.

All local variables in procedures and functions must be declared. Implicit creation by an assignment to undeclared local variable is not allowed and will result in an error. Procedures and functions are always located at the end of the program. There is no explicit ENDPROC/ENDFUNC statement because each procedure/function definition ends either at the end of the program or at the beginning od next procedure/function. Because function calls are executed inside the context of an expression there is a significant overhead associated with each call compared to the procedure call. For this reason it is recommended to use procedures where execution speed is essential. And if the maximum speed is really, really needed then a good old GOSUB/RETURN is the best choice because it has the less overhead.

Various program structures can be spanned over more than one line by using blocks. Their function is to combine more statements/lines into single entities. The most common application of blocks is with conditional statements like IF and CASE because loops and procedures/functions can be spanned over more lines by design. Blocks must be denoted by { and } symbols and can be nested up to 15 levels.

Example:

```
IF 1<2 {
_  IF 3<4 ? "3<4" ELSE ? "3>4"
}
ELSE {
_  ? "1>2"
_  IF 6>5 ? "6>5" ELSE ? "6<5"
}
```

HRAST-BASIC supports recursion as well, but with some limitations and additional requirements forced by the interpreter architecture. Due to the static variable allocation every procedure/function has a static set of variable links. So, if you call the procedure XYZ from procedure XYZ the new call will overwrite the variable links of the previous call. This affects procedures and functions only, GOSUB statement is not affected because it doesn't handle local variables anyway. To overcome this problem you must make copies of all variables you want to use as parameters in a recursive call. If the recursive call appears at the end of a procedure/function then you don't have to do anything else. If it appears in the middle and you need to use variables defined before the call then you have to use PUSH and POP statements to save and restore the required variable links.

Examples:

```
DISP FN FACT(69)
PROC FACT(N) REAL P=N-1
IF N<2 RETURN 1 ELSE RETURN N*FN FACT(P)

CALL HANOI,"A","B","C",%3
PROC HANOI(S,T,D,N)
IF N<1 RETURN
STRING S0(1)=S,T0(1)=T,D0(1)=D
INTEGER N0=N
PUSH S0,D0,T0,N0
CALL HANOI,S0,D0,T0,N0-1 POP
DISP N0;": ";S0;" => ";D0
CALL HANOI,T0,S0,D0,N0-1
```

Of course, it is always a good idea to see if a particular problem can be solved without using recursion.

3. Modes of Operation

3.1 Command/Program Modes & Execution Context

HRAST-BASIC is not only an language interpreter, it is a full development environment as well. It features an command environment for immediate command execution and expression evaluation together with the integrated full-screen program editor. I wanted both editors to be as simple as possible but, on the other end, to have everything you need during the development of an program. After you start the interpreter it will boot into command editor and an flashing question mark will appear on the display. Whenever an ? flashes at the beggining of the command line you have the following choices: press Up/Left/Right arrow or Back/DEL keys to recall the content of the last/previous command line, press Down arrow key to start the program editor, or press any other key to start the new command line. The edit buffer is 127 characters in size so quite a long command/program lines can be entered.

Due to the interpreter architecture variables can only exist inside the context of an program. I emulated this functionality to some extent inside command environment as well, so 26 one-letter variables are available there to use in calculations etc. But, the primary purpose of the interpreter is programming, not manual calculations, so the command mode is there to support this purpose and is not as advanced as, for example, CALC mode on HP-71B.

When you RUN the program (or execute COMP to compile the program) this command context will be destroyed and the program context will take it's place. After the program finishes execution (either by END, STOP or ON/BREAK) this program context will remain to be active inside program environment and you can use all variables created by the program to look at the results or make additional manual calculations using those variables. If the program has been stopped by STOP statement or ON/BREAK key you can change the values of some variables, execute CONT and the program will continue running using those new values. After you don't need the program context anymore just execute RESET to destroy it and create new command context. This dual context nature is one of many unique characteristics of HRAST-BASIC.

### 3.2 [48K only] TEXT Programs

Beside the regular BASIC programs, 48K version supports special TEXT programs which can contain raw ASCII text instead of tokenized/compiled programs. TEXT programs do nothing when executed and can be used as scratchpad or notepad, for example. But, the main idea behind them is to use the editor to edit FORTH programs because I plan to continue working on HRAST-FORTH and eventually incorporate it into HRAST-BASIC environment. When I started developing HRAST-FORTH long time ago I didn't have a suitable editor and environment for what I wanted to achieve. Now I have everything I need for this :-)

3.3 [48K only] Graphics Mode

Graphics mode enables drawing lines and control of every single LCD pixel. It shares the same memory with the text screen, so it is possible to display graphics and text at the same time. Contrary to the text screen, graphics mode origin (0,0) has been set to the lower left corner because it is much more convenient to draw graphs in the first quadrant.

Graphics program example - sine graph display:

```
CLS
INTEGER H=YDIM/2,W=<XDIM
LINE W,H TO 0,H
REAL X=0,S=2*PI/XDIM
FOR I=0 TO W
DRAW I,H*>SIN X REM PLOT I,H*>SIN X
X+S NEXT
```
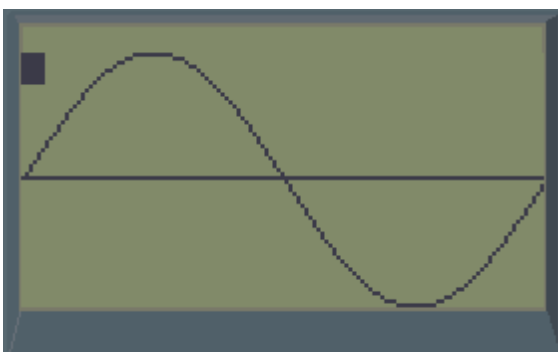
3.4 Keyboard

The following special keys are available in both command and program mode:

ENTER ... Execute the command line or save the line inside a program editor.
Back ... Delete the previous character.
DEL ... Delete a character under the cursor (HP-48G/G+/GX only).
Right Shift + Back ... Delete a character under the cursor.
Left Arrow ... Move cursor to the previous character.
Right Arrow ... Move cursor to the next character.
Left Shift + Left Arrow ... Move cursor to the start of the line.
Left Shift + Right Arrow ... Move cursor to the end of the line.
Right Shift + I ... Toggle between insert/overwrite mode.
Right Shift + L ... Toggle between upper/lower case.
ON ... Cancel the line (return to flashing ? in command mode or revert to the original line in program mode).
ON & Left/Right Shift ... Power off (the same as OFF statement).
ON & F ... Return to the operating system.
ON & U ... Toggle keyboard user mode on/off.
ON & +, ON & - ... LCD contrast increase/decrease.
ON & Any other key ... Cancel ON.

Note: Arrow keys needs Right Shift on HP-48G/G+/GX.

[48K only] Right Shift + Left Shift + Left Arrow ... Delete to the start of the line.
[48K only] Right Shift + Left Shift + Right Arrow ... Delete to the end of the line.

[48K only] Left Shift + ENTER ... Put the result from the last displayed numeric calculation into the current line.
[48K only] Left Shift + S ... Single-stepping (SST) through the program in debug/trace mode (command mode only).

To access shifted keys press-and-hold Left, Right or Alpha Shift while pressing other keys or (available in 48K version only) press-and-release Left, Right or Alpha Shift and then press another key. In the second case use ON key to cancel active Left or Alpha Shift if needed. Right Shift followed by the ON key will turn the calculator off (48K version only).

The keyboard supports autorepeat. After 5 minutes of (keyboard) inactivity the command environment will switch the calculator off - after powering it on again you will be positioned exactly at the same location inside a command line or program where you have been positioned before switching off. The cursor is solid when the editor is in insert mode and transparent (blinking between character and inverse character) in overwrite mode. The cursor doesn't blink while any of the shift keys is held down.

The following special keys are available in program mode only:

Up Arrow ... Move cursor to the previous line.
Down Arrow ... Move cursor to the next line.
Left Shift + Up Arrow ... Move cursor to the start of the program.
Left Shift + Down Arrow ... Move cursor to the end of the program.
Right Shift + D ... Delete the current line.
Right Shift + N ... Insert new line at the current position.

[48K only] Left Shift + W ... Swap the current and the next line.
[48K only] Left Shift + U ... Duplicate the current line (not possible if there is not enough room).
[48K only] Left Shift + M ... Merge the current line with the next one.

Merge is not possible if the next line starts with procedure/function declaration or indentation delimiter. Be very careful when merging lines because the editor doesn't check if you created a line which exceeds the length of the edit buffer!

There is no "split line" key but you can get around it by duplicating the line and using "delete to the start of the line" and "delete to the end of the line".

While in program mode you can freely navigate up and down through the program until you make some changes to the current line. After that you must either confirm the changes by pressing the ENTER key or cancel changes by pressing ON. This is to prevent loosing changes by accident while editing a line.

10

[48K only] Special two-keys combinations:

Right Shift + Alpha & / (- on HP-49G/49G+/50G) ... Enter () at the cursor position and move cursor inside brackets.
Right Shift + Alpha & * ... Enter [] at the cursor position and move cursor inside brackets.
Right Shift + Alpha & + ... Enter {} at the cursor position and move cursor inside brackets.

3.5 [48K only] User Mode

The whole keyboard is completely redefinable and you can assign a string to any key (including all Left/Shift/Alpha combinations), except ON and Left/Right/Alpha shifts themselves. User mode is active by default and you have to use ON & U to turn it off/on. Use STATE statement to see the current user mode state.

Look at the keyboard tables in section 9 for redefinable key codes. Right Shift increments the primary key code by 50, Left Shift increments it by 100 and Alpha Shift increments it by 200.

The following special functions can be assigned as well:

`CHR(07)` ... Enter () at the cursor position and move cursor inside brackets.
`CHR(08)` ... Enter [] at the cursor position and move cursor inside brackets.
`CHR(09)` ... Enter {} at the cursor position and move cursor inside brackets.
`CHR(10)` ... Delete to the start of the line.
`CHR(11)` ... Delete to the end of the line.
`CHR(12)` ... Put the result from the last displayed numeric calculation into the current line.
`CHR(13)` ... Execute the command line or save the line inside a program editor.
`CHR(14)` ... Toggle between upper/lower case.
`CHR(15)` ... Toggle between insert/overwrite mode.
`CHR(16)` ... Delete the previous character.
`CHR(17)` ... Delete a character under the cursor.
`CHR(18)` ... Move to the previous character.
`CHR(19)` ... Move to the next character.
`CHR(20)` ... Move to the previous line (program mode only).
`CHR(21)` ... Move to the next line (program mode only).
`CHR(22)` ... Move to the start of the line.
`CHR(23)` ... Move to the end of the line.
`CHR(24)` ... Move to the start of the program (program mode only).
`CHR(25)` ... Move to the end of the program (program mode only).
`CHR(26)` ... Delete the current line (program mode only).
`CHR(27)` ... Insert new line at the current position (program mode only).
`CHR(28)` ... Swap the current and the next line (program mode only).
`CHR(29)` ... Duplicate the current line (program mode only).
`CHR(30)` ... Merge the current line with the next one (program mode only).
`CHR(31)` ... Single-stepping (SST) through the program in debug/trace mode (command mode only).

Those must appear at the end of the assignment - if not, the rest of the string will be ignored. All codes less than 10 will be ignored as well.

11

4. Expressions & Operators

The depth of an arithmetic expression is limited by the free memory only. During the expression evaluation the usual operator precedence/priority applies - this can be changed by using the brackets. The interpreter will do automatic conversion between numeric types when needed. Depending on the expression context (the statement or function evaluating an expression etc.) the expression can start to be evaluated as INTEGER or REAL. For example, GOTO/GOSUB statements expect INTEGER results so it is logical to start the evaluation as INTEGER. Expressions related to FOR statement are INTEGER by default as well, while DISP expressions are REAL by default, etc. This can be changed using `%` and `#` operators. Furthermore, when the interpreter encounters decimal dot or exponent (E) character it will automatically switch to REAL evaluation. When maximum speed is needed it is always a good idea to try to calculate with INTEGERs as much as possible.

The following operators are infix - they must appear between two expression terms:

`+` ... Add two INTEGER/REAL/COMPLEX values
`-` ... Subtract two INTEGER/REAL/COMPLEX values
`*` ... Multiply two INTEGER/REAL/COMPLEX values
`/` ... Divide two INTEGER/REAL/COMPLEX values
`^` ... Power of two INTEGER/REAL/COMPLEX values (COMPLEX on 48K only)

`&` ... Concatenates two STRING values together

`=, <>, <, >, <=, >=` ... Relational operators on two INTEGER/REAL/COMPLEX/STRING values.

The result of an relational operation is -1 (True) if the condition has been satisfied or 0 (False) otherwise. In conditional expressions any nonzero INTEGER value is considered True and 0 is considered False.

`AND` ... Binary/logical AND between two INTEGER values
`OR` ... Binary/logical OR between two INTEGER values
`XOR` ... Binary/logical XOR between two INTEGER values
`SHF` ... Shift an INTEGER value left (positive right operand) or right (negative right operand)

The following operators are prefix - they must appear before an operand (or alone in case of `=`, `>=` and `<=`):

`!` ... Binary NOT of an INTEGER value
`%` ... Marks the following numeric value as INTEGER
`#` ... Marks the following numeric value as REAL
`-` ... Negative INTEGER/REAL/COMPLEX value
`+` ... Positive INTEGER/REAL/COMPLEX value
`/` ... Inverse of INTEGER/REAL/COMPLEX value
`^` ... Square of INTEGER/REAL/COMPLEX value
`&` ... Hexadecimal operator (must be followed by hexadecimal STRING value)
`=` ... Special READ operator (see READ/DATA statements for more info)
[48K only] `>` ... Increment INTEGER/REAL/COMPLEX value by 1
[48K only] `<` ... Decrement INTEGER/REAL/COMPLEX value by 1
[48K only] `<>` ... Multiply INTEGER/REAL/COMPLEX value by [0,1]
[48K only] `>=` ... Returns [0,1] COMPLEX value
[48K only] `<=` ... Returns [0,-1] COMPLEX value

The following operators are postfix - they must appear after an operand:

`!` ... Factorial function
`%` ... Converts the previous numeric value to INTEGER
`#` ... Converts the previous numeric value to REAL
`$` ... Converts the previous numeric value to STRING
`(N)` ... Substring operator - extract N-th character from the string
`(M,N)` ... Substring operator - extract N characters from the string starting with character M
`(,N)` ... Substring operator - extract right N characters from the string
`(N,)` ... Substring operator - extract left N characters from the string

Substring operator is an equivalent to LEFT$, RIGHT$ and MID$ functions from the regular BASIC.

Examples:

```
123.456%            = 123
-[1,2]              = [-1,-2]
^1E6                = 1.E12
1E6^2               = 1.E12
/128                = 0.0078125
/[1,2]              = [0.2,-0.4]
6!                  = 720
123.456$&"ABCD"     = "123.456ABCD"
"ABCDEFGHIJ"(4)     = "D"
"ABCDEFGHIJ"(3,4)   = "CDEF"
"ABCDEFGHIJ"(,4)    = "GHIJ"
"ABCDEFGHIJ"(4,)    = "ABCD"
&"ABCD"             = 43981
(2.1*6+11+^(6+8*(100/50+3)-7-35)/2+100+(2+4*^3-18)/4+(100*10-^8)-44/11)/12    = 89.05
(%2.1*6+11+^(6+8*(100/50+3)-7-35)/2+100+(2+4*^3-18)/4+(100*10-^8)-44/11)/12   = 89.05
(2.1%*6+11+^(6+8*(100/50+3)-7-35)/2+100+(2+4*^3-18)/4+(100*10-^8)-44/11)/12   = 89
([2,1]*6+11+^(6+8*(100/50+3)-7-35)/2+100+(2+4*^3-18)/4+(100*10-^8)-44/11)/12 = [89,0.5]
```

13

5. Functions

Function arguments are expressions and must be enclosed in parenthesis, except for atomic values with 1-argument functions (for example, SIN 0.5, SQR 9 or ABS [1,2]). If there are two (INTEGER and REAL) or three (INTEGER, REAL and COMPLEX) versions of the same function then the result is of the same type as function argument. If an function expects INTEGER argument only then REAL will be converted to INTEGER automatically. On the other way, if an function expects REAL argument then INTEGER will be converted to REAL automatically. Of course, you can always use % and # operators if an explicit conversion is needed.

***

ABS(INTEGER), ABS(REAL), ABS(COMPLEX)

Absolute value of an INTEGER, REAL or COMPLEX argument.

***

ALOG(REAL)
[48K only] ALOG(COMPLEX)

Base 10 antilogarithm of an REAL or COMPLEX argument (e.g. power of ten = 10 to an REAL argument).

***

ASIN(REAL), ACOS(REAL), ATAN(REAL)
[48K only] ASIN(COMPLEX), ACOS(COMPLEX), ATAN(COMPLEX)

Inverse trigonometric functions of REAL or COMPLEX argument.

***

ASINH(REAL), ACOSH(REAL), ATANH(REAL)
[48K only] ASINH(COMPLEX), ACOSH(COMPLEX), ATANH(COMPLEX)

Inverse hyperbolic functions of REAL or COMPLEX argument.

***

ANGLE(REAL, REAL)

The angle of an (X,Y) coordinate pair.

***

ASC(STRING)

ASCII value of the first character of an STRING argument or zero in the case of an empty STRING.

***

CEIL(REAL)
[48K only] CEIL(COMPLEX)

The ceiling of an REAL or COMPLEX argument (the first integer number greater than argument).

***

CHR(INTEGER)

The character with the ASCII code sent as an argument.

***

14

[48K only] CLOCK

Returns the number of seconds since the start of the interpreter or since interpreter power on.

***

COMB(REAL, REAL), PERM(REAL, REAL)

Combinations and permutations of two REAL arguments.

***

[48K only] DATE

Returns the current date as a numeric value (DD.MMYYYY or MM.DDYYYY). The current date is read from the calculator at the start of the interpreter and is managed by the interpreter itself afterwards.

Example:

```
? "D=";DATE, FIX 2 ? "T=";TIME
D=5.272015 T=18.31
```

***

[48K only] DMS(REAL)

Converts hours/minutes/seconds in HH.MMSSTTTTT format to decimal hours.

***

ERR

The code of last error (or zero if no error occurred). See TRAP for more info on error handling/trapping.

***

EVAL(STRING)

The numeric value of an STRING expression with full support for variables, operators and functions. You can use everything in EVAL expressions, including functions and all variables defined in the main program or inside a procedure/function where EVAL has been executed. But, due to the interpreter architecture, EVAL and VAL functions cannot be nested e.g. you cannot use EVAL/VAL from EVAL. If you need to convert just a simple expression (like numbers alone, for example) you can consider using VAL function.

***

EXP(REAL)
[48K only] EXP(COMPLEX)

Exponentional function of an REAL or COMPLEX argument.

***

EXPON(REAL)
[48K only] EXPON(COMPLEX)

The exponent of an REAL or COMPLEX argument.

***

[48K only] FIND(STRING, STRING)

Find the appearance of the first STRING inside the second STRING. Returns 0 if not found.

Example:

```
FIND("IJKL","ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

***

FN <Name>[([<Expression1>][,<Expression2>] ... [,<ExpressionN>])]

Function call with optional parameter list. See FUNC and section 2.7 for more info about functions.

Example:

```
DISP FN FACT(10)
FUNC FACT(N)
_ REAL X=1
_ FOR N=N X*N NEXT
RETURN X
```

***

FRC(REAL)
[48K only] FRC(COMPLEX)

The fractional part of an REAL or COMPLEX argument.

***

[48K only] HEX(INTEGER)
[48K only] HEX(INTEGER, INTEGER)

Convert INTEGER argument to hexadecimal STRING - with the specified number of digits (together with leading zeros) if the second argument has been provided.

***

[48K only] HMS(REAL)

Converts decimal hours to hours/minutes/seconds in HH.MMSSTTTTTT format.

***

IMAG(COMPLEX)

The imaginary part of an COMPLEX argument.

***

INKEY

Returns the ASCII code of the key down or zero if no key down.

***

INT(REAL)
[48K only] INT(COMPLEX)

The integer part (floor) of an REAL or COMPLEX argument (the first integer number less than argument).

***

16

KEY

Returns the native code of the key down or zero if no key down.

***

LEN(STRING)

Returns the length of an STRING argument.

***

LN(REAL)
[48K only] LN(COMPLEX)

The natural (base e) logarithm of an REAL or COMPLEX argument.

***

LOG(REAL)
[48K only] LOG(COMPLEX)

Base 10 logarithm of an REAL or COMPLEX argument.

***

MANT(REAL)
[48K only] MANT(COMPLEX)

The mantissa of an REAL or COMPLEX argument.

***

MEM

The amount of free memory in bytes.

***

MOD(INTEGER, INTEGER), MOD(REAL, REAL)

The remainder after the division of two INTEGER or REAL numbers. For example, MOD(111,7)=6.

***

NOT(INTEGER)

Logical negation of an INTEGER argument.

***

PI

Value of PI.

***

17

[48K only] PIXEL(REAL, REAL)

Checks the state of a particular pixel at given (X,Y) display coordinates and returns -1 (True) if the pixel is turned on (is visible) or 0 (False) otherwise.

Example:

```
IF PIXEL(120,60) ? "PIXEL ON"
```

***

RND(INTEGER), RND(REAL), RND(COMPLEX)

Returns random value between 0 and the argument specified. In case of INTEGER argument the result will be an INTEGER as well, otherwise the result will be REAL.

***

[48K only] RECV(0)

Waits for a byte to be received over the serial connection. With handshake enabled it will also send the received byte back to the sender for additional 2-way control. Connection should be opened prior to using RECV. If nothing received during the timeout period an TIMEOUT error will be raised.

See also: HANDS, SEND

***

[48K only] RECV(INTEGER)

Returns the STRING of requested number of characters received over the serial connection. With handshake enabled it will also send the received characters back to the sender for additional 2-way control. Connection should be opened prior to using RECV. If nothing received during the timeout period an TIMEOUT error will be raised.

See also: HANDS, SEND

***

[48K only] RECVB

Returns the byte received over the serial connection or -1 if nothing received. With handshake enabled it will also send the received byte back to the sender for additional 2-way control. Connection should be opened prior to using RECVB.

See also: HANDS, SEND

***

[48K only] RECVD

Checks the serial connection state and returns -1 (True) if a byte has been received (and is ready for reading) or 0 (False) otherwise. Connection should be opened prior to using RECVD.

See also: SEND

Example:

```
IF RECVD ? "RECEIVED: ";RECVB
```

***

ROOT(REAL, REAL)
[48K only] ROOT(REAL | COMPLEX, REAL | COMPLEX)

The root of two REAL or COMPLEX numbers. For example, ROOT(8,3)=2.

***

ROUND(REAL)
[48K only] ROUND(COMPLEX)

The rounded value of an REAL or COMPLEX argument.

***

[48K only] SHF(INTEGER)

Bit shift = shift 1 (bit 0) the requested numbers of bits to the left (SHF 8 = 256, for example).

***

SIGN(INTEGER), SIGN(REAL), SIGN(COMPLEX)

The sign of an INTEGER, REAL or COMPLEX argument.

***

SIN(REAL), COS(REAL), TAN(REAL)
[48K only] SIN(COMPLEX), COS(COMPLEX), TAN(COMPLEX)

Trigonometric functions of REAL or COMPLEX argument.

***

SINH(REAL), COSH(REAL), TANH(REAL)
[48K only] SINH(COMPLEX), COSH(COMPLEX), TANH(COMPLEX)

Hyperbolic functions of REAL or COMPLEX argument.

***

SPACE(INTEGER)

Returns an STRING of the specified number of spaces.

***

SQR(REAL), SQR(COMPLEX)

The square root of an REAL or COMPLEX argument.

***

STR(INTEGER), STR(REAL), STR(COMPLEX)

Returns an INTEGER, REAL or COMPLEX number converted to STRING. In 48K version STR with STRING argument will return the string converted to uppercase. One precious token code has been saved this way.

***

TAB(INTEGER)

Set the column for the next DISP/?, INPUT or PRINT statement. If DISP/? statements have been directed to the printer with PRINT TO then sets the column for the printer only.

Example:

```
FOR I=0 TO 9
_ DISP TAB I;I
NEXT
```

***

TICKS

Returns the number of seconds (as REAL) elapsed since the last TICKS, WATCH, CLOCK, DATE or TIME, restart the stopwatch and reset TICKS timer.

***

[48K only] TIME

Returns the current time as a numeric value (HH.MMSS...). The current time is read from the calculator at the start of the interpreter and is managed by the interpreter itself afterwards.

***

[48K only] TYPE(<Expression>)

Returns the type of the expression (0 = INTEGER, 2 = REAL, 4 = COMPLEX, 8 = STRING).

***

VAL(STRING)

The numeric value of an STRING expression. This a simpler version of EVAL function which doesn't use tokenizer and syntax checker, so it is much faster than EVAL but cannot handle variables and multi-character tokens (operators and functions).

***

[48K only] VARTYPE <Variable>

Returns the type of the variable:

0 = INTEGER	1 = INTEGER ARRAY
2 = REAL15	3 = REAL15 ARRAY
4 = COMPLEX15	5 = COMPLEX15 ARRAY
8 = STRING	9 = STRING ARRAY
12 = REAL12 (SHORT)	13 = REAL12 (SHORT) ARRAY
14 = COMPLEX12 (CSHORT)	15 = COMPLEX12 (CSHORT) ARRAY

***

[48K only] XDIM

Returns the horizontal LCD dimension in pixels.

***

[48K only] YDIM

Returns the vertical LCD dimension in pixels.

***

[48K only] XPOS

Returns the horizontal draw position.

***

[48K only] YPOS

Returns the vertical draw position.

***

Note about **INT**, **FRC**, **CEIL**, **DMS**, **HMS**, **MANT**, **EXPON**, **RND** and **ROUND** functions:

In case of COMPLEX argument only REAL part will be used and REAL value will be returned as a result.

21

6. Statements

[48K only] AGAIN [@]

Rewind the innermost FOR/NEXT, REPEAT/UNTIL or WHILE/ENDWHILE loop as well as CASE/ENDCASE or WHEN/ENDWHEN statement to the beginning and do the next iteration without condition checking (and without variable increment in case of FOR statement). Accelerator does make sense for WHILE/ENDWHILE statements only.

See also: LOOP

Example:

```
I=%1
REM ENDLESS LOOP
REPEAT
_ DISP I
_ I+1
_ IF I>5 AGAIN
_ DISP "NEVER EXECUTED"
UNTIL I>10
```

***

[48K only] ARRAY [<BaseIndex 0 | 1>]

Set the array base index to either 0 (default) or 1 so you can have 0-indexed or 1-indexed arrays. This is a global option and will affect all programs and procedures/functions. RUN will set array base index to zero.

***

[48K only] ASSIGN <KeyCode>[,<Assignment>]

Redefine a particular key using the string assignment or delete the current assignment. See sections 3.4 and 3.5 for more info on keyboard and user mode assignments. This statement can be executed from command mode only.

Examples:

```
ASSIGN 1,"123*456"
ASSIGN 2,"DISP "
ASSIGN 3
ASSIGN 4,"DISP 123*456"&CHR(13)
```

***

ATTR [<Attribute>]

Set the attributes for subsequent DISP statements. Bit 0 controls double-height characters and bit 1 controls inverse mode. ATTR without expression resets the attributes to zero. This is a global setting and will affect all programs and procedures/functions.

Examples:

```
ATTR 1 ? "DOUBLE-HEIGHT"
ATTR 2 ? "INVERSE"
ATTR 3 ? "DOUBLE-HEIGHT+INVERSE"
ATTR ? "NORMAL"
```

***

22

[48K only] BEEP <Pitch>[,<Duration>]

Produce a beep of the specified pitch (Hz) and duration (milliseconds). If the pitch has been omitted then 500Hz will be used. If the duration has been omitted then 250ms will be used.

Examples:

```
BEEP 261,1000
BEEP 4*261
BEEP ,777
```

***

BYE

Return to the HP-48/49/50 operating system (like ON&F but can be executed from the program).

***

CALL <Name>[,<Expression1>][,<Expression2>] ... [,<ExpressionN>]

Procedure call with optional parameter list. See PROC and section 2.7 for more info about procedures.

***

CASE <Statements>
[OF [@]<Expression1>[@] <Statements1>]
[OF [@]<Expression2>[@] <Statements2>]
...
[OTHER [@] <StatementsN>]
ENDCASE

Complex multiconditional statement. If <Expression1> evaluates to True then <Statements1> will be executed, if <Expression2> evaluates to True then <Statements2> will be executed, etc., and (otherwise) if all expressions evaluate to False then <StamentsN> will be executed. OF and OTHER statements are single-line only (like IF/ELSE) but you can use program blocks to span them over multiple lines, or for nesting CASE and other structures inside each other.

Example:

```
X=1,Y=1
CASE
_ DISP "START"
_ OF X=1 DISP "A"
_ OF X=2 { DISP "B"
__ CASE OF Y=1 DISP "1"
__ OF Y=2 DISP "2"
__ OTHER DISP "3" ENDCASE DISP "123" }
_ OF X=3 DISP "C"
_ OTHER DISP "X"
ENDCASE
DISP "DONE"
```

***

[48K only] CAT

Display the catalog of all programs in memory. The catalog will contain the name, type (BASIC/TEXT) and size (in bytes) of each program. The current program will be marked with an "greater than" character (>). ENTER key pauses the listing until released, ON key stops the listing and SPC key sets the currently displayed program to be the current program. If CAT has been executed from the program and SPC key has been pressed then a currently displayed program will be run.

***

[48K only] CAT 0

Display the catalog of all key assignments (in order they have been created).

***

CLEAR

Clear/destroy all variables.

***

[48K only] CLOSE

Close serial communication.

Note: PRINT (and DISP when redirected to printer) doesn't need serial port to be explicitly opened and closed.

***

CLS

Clear the screen.

***

COL [<Column>]

Set the column for the next DISP/? (LCD only) or INPUT statement. COL without expression defaults to column zero.

Example:

```
FOR I=0 TO 9
_  COL I
_  DISP I
NEXT
```

***

COMP

Destroy command mode context, compile current program and create the associated program context. After the compilation you can use all program variables in the command mode as well. RUN always executes COMP if the program hasn't been compiled already. Every change to the current program will automatically decompile the program and execute RESET. This statement can be executed from command mode only.

***

COMPLEX <Variable1>[=<Expression1>][,<Variable2>[=<Expression2>]] ... [,<VariableN>[=<ExpressionN>]]

Declaration of COMPLEX variables with optional initialization.

Example:

```
COMPLEX A,B,C
A=^(^[5-3,9/3]/[2,3*3/3-3+3])
B=[44/11,2+3]/[2,3]/[^2,1+^2]
DISP "A=";A
DISP "B=";B
C=A*B
DISP "C=";C
```

***

24

CONT

Continue program execution after and STOP statement has been encountered. Values of all variables are preserved.

[48K only] CONT statement will automatically turn trace mode off (if active) and resume normal execution.

***

CSHORT <Variable1>[=<Expression1>][,<Variable2>[=<Expression2>]] ... [,<VariableN>[=<ExpressionN>]]

Declaration of SHORT (REAL12) COMPLEX variables with optional initialization.

Example:

```
CSHORT C,C12=[1,2]
```

***

DATA [@]Expression1[,Expression2] ... [,ExpressionN]

Data list for READ statement. DATA statements are skipped during execution and the accelerator can be used to make skipping faster if desired (although it is a good programming practice to put DATA statements outside the main program flow).

See also: READ, RESTORE

***

DEG, RAD, GRAD

Sets the appropriate angle mode for trigonometric functions. This is a global setting and will affect all programs and procedures/functions.

***

DISP [<Expression1>][;<Expression2>][,<Expression3>] ...
? [<Expression1>][;<Expression2>][,<Expression3>] ...

Display the result of one or more expressions. In case of numeric expression the current FIX/SCI settings are used for display. The following delimiters are allowed:

; ... The next expression will be displayed immediately after the current one.
, ... The next expression will be displayed one character after the current one.

See also: ATTR, COL, TAB

Examples:

```
DISP "S=";S
DISP "T=";T
DISP A[1],A[2],A[3]
DISP "READY> "; INPUT S
? X;" ";Y;" ";Z
```

***

[48K only] DISP TO

Direct the output from DISP/?, CAT and STATE statements to the LCD. This is a global setting and will affect all programs and procedures/functions.

***

25

[48K only] DRAW < ExpressionX>,< ExpressionY> [STEP <Expression>]

Draw a line between the current draw position and the new display point. If the expression following the optional STEP keyword evaluates to 0 then a line will be cleared, if it evaluates to 1 then a solid line will be drawn (default behavior without STEP), otherwise pixels will be toggled (pixels which are turned on will be turned off and vice versa). Sets the new draw position to the point specified.

See also: LINE, MOVE, PLOT

Example:

```
MOVE 5,5 DRAW 125,5
DRAW 125,50 DRAW 5,50 DRAW 5,5
```

***

EDIT [<Expression>]
EDIT FN <Name>

Activate the program mode to edit the current program. If an expression has been provided then:

(1) Go to the specified numeric label in case of an numeric expression with positive result or zero.
(2) Go to the specified line in case of an numeric expression with negative result.
(3) Go to the specified string label in case of an string expression.
(4) Go to the specified procedure or function in case of FN <Name>.

Examples:

```
EDIT 123
EDIT -123
EDIT "123"
EDIT FN TEST
```

***

END [<Expression>]

Marks the end of the program or procedure/function if there is no return address on the stack, otherwise acts like RETURN.

***

ERROR [<ErrorCode>]

Display the specified error message, or the last error message (if any) in case no error code is provided. See TRAP for more info on error handling/trapping and Chapter 8 for the list of all available error messages with appropriate codes.

***

EXEC <Expression>

Evaluate the expression, tokenize it, check syntax, compile it and execute. You can use almost all language elements in EXEC expressions, including functions and all variables defined in the main program or inside a procedure/function where EXEC has been executed. But, due to the interpreter architecture, you cannot execute EXEC, EVAL, VAL and INPUT from EXEC.

[48K only] Statement abbreviations can be freely used inside the expression.

Examples:

```
EXEC "DISP 123*456"
INTEGER I EXEC "FOR I=1 TO 5 ? I NEXT"
```

***

EXIT [@]

Exit from the innermost FOR/NEXT, REPEAT/UNTIL or WHILE/ENDWHILE loop. Also exit from the innermost CASE/ENDCASE or WHEN/ENDWHEN statement.

Example:

```
FOR I=1 TO 10
_ DISP I IF I>5 EXIT
_ DISP "1..5 ONLY"
NEXT DISP "DONE"
```

***

FIX [<Decimals>] & SCI [<Decimals >]

Activate fixed decimals or fixed scientific mode. This mode will then be used in successive display statements or STR function. FIX/SCI without argument deactivates the current mode and returns to normal display. This is a global setting and will affect all programs and procedures/functions.

***

FOR <Var>=<Expr1> <Statements> NEXT
FOR <Var>=<Expr1> TO <Expr2> <Statements> NEXT
FOR <Var>=<Expr1> TO <Expr2> STEP <Expr3> <Statements> NEXT

Program loop structure with predetermined number of repetitions. FOR variable <Var> can be INTEGER (default) or REAL. INTEGER is preferred for faster execution. The first form counts from <Expr1> down to 1 with step -1. The second form counts from <Expr1> to <Expr2> with step 1. The second form counts from <Expr1> to <Expr2> with step <Expr3>. The loop will execute at least once. Modifications of FOR variable inside a loop are allowed although not recommended. NEXT statement denotes the end of an FOR loop. Contrary to other BASICs out there, the variable after NEXT is not allowed and NEXT always closes the innermost FOR loop.

Example:

```
N=%20,X=1
FOR N=N
_ FOR I=2 TO N
__ FOR J=1 TO 4 GOSUB @5 NEXT
__ X*I
_ NEXT
NEXT
```

***

FUNC <Name>[([=|<Param1>[,[=|<Param2>] ... [,[=|<ParamN>])]

Function definition. Declare the function with the optional parameter list. See PROC for more info on parameters and section 2.7 for more info on structured programming. Function should always return a value by using RETURN or END statements followed by the result expression.

Examples:

```
DISP FN FACT(10)
FUNC FACT(N)
REAL X=1
_ FOR N=N X*N NEXT
RETURN X

DISP FN FACT(50)
PROC FACT(INTEGER N)
INTEGER P=N-1
IF N<2 RETURN 1 ELSE RETURN N*FN FACT(P)
```

***

27

GOSUB [@]<Expression>

Transfer program execution to subroutine at the specified numeric or string label. Accelerator usage with calculated GOSUB is not recommended because the destination address will be fixed to the result of first expression evaluation.

See also: RETURN, END

***

GOTO [@]<Expression>

Transfer program execution to the specified numeric or string label. Accelerator usage with calculated GOTO is not recommended because the destination address will be fixed to the result of first expression evaluation.

Examples:

```
GOTO @123
GOTO 12*10+3
GOTO "12"&"3"
```

***

[48K only] HANDS [<Handshake>]

Turn handshake for serial connection off (0) or on (<>0). With handshake enabled, each RECV will send the received byte back to the sender for additional 2-way control. SEND statement will compare it with the last byte sent and raise TRANSMIT ERROR if they don't match (SEND waits for this handshake byte after each byte sent). This is a global option and will affect all programs and procedures/functions.

Note: Serial communication is not buffered, so handshake is mandatory when exchanging data with another calculator running HRAST-BASIC. Handshaking is rather slow process but this is the only way to ensure a reliable connection in this situation.

See also: SEND, RECV

***

IF <Expression>[@] <Statements>
IF <Expression>[@] <Statements1> ELSE [@] <Statements2>

Inline conditional statement. THEN statement required in other BASICs doesn't exist here. In the first form <Statements> are executed if <Expression> evaluates to True. In the first form <Statements> are executed if <Expression> evaluates to True and <Statements2> otherwise. The complete inline IF statement (including ELSE) must appear on the same line.

Inline IF examples:

```
IF X>3 GOTO 20
IF 1<2 DISP "1<2" ELSE DISP "1>2"
IF 1<2@ DISP "1<2" ELSE @ DISP "1>2"
```

If multiline conditional execution is needed then you must use program blocks (refer to section 2.7 for more info) like in this combined inline/multiline IF example:

```
IF 1<2 {                             ELSE {
_ ? "1<2"                            _ ? "1>2"
_ IF 3<4 {                           _ IF "A"<"B" {
__ ? "3<4"                           __ ? "A<B"
__ IF 5<6 ? "5<6" ELSE ? "5>6"       __ IF "C"<"D" ? "C<D" ELSE ? "C>D"
_ }                                  _ }
_ ELSE {                             _ ELSE {
__ ? "3>4"                           __ ? "A>B"
__ IF 7<8 ? "7<8" ELSE ? "7>8"       __ IF "E"<"F" ? "E<F" ELSE ? "E>F"
_ }                                  _ }
}                                    }
```

28

It is always a good idea to avoid such long sausages and try to use procedures/functions or combine inline/multiline IF statements with WHEN/UNLIKE/ENDWHEN to group statements into logical units where possible.

***

INPUT <Variable>

Prompts for a value of the specified variable. In case of a numeric variable the expression will be evaluated and the result will be assigned to the variable. In case of a string variable the string will be assigned without evaluation (so you don't have to enclose the string in quotes in this case). You can use almost all language elements in INPUT expressions, including functions and all variables defined in the main program or inside a procedure/function where INPUT has been executed. But, due to the interpreter architecture, you cannot use VAL and EVAL functions here. Furthermore, it is not possible to use starting character designation with string variables here e.g. STRING A INPUT A(1) is not allowed.

Main program only: If the variable didn't exist prior to INPUT it will be created automatically and the result of evaluated expression will be assigned to it (strings should be entered with quotes in this case). This doesn't apply to procedures/functions because all variables should be explicitly declared there.

Examples:

```
INPUT A
INPUT A[2] INPUT B[2,3]
DISP "X? "; INPUT X
```

***

INTEGER <Variable1>[=<Expression1>][,<Variable2>[=<Expression2>]] ... [,<VariableN>[=<ExpressionN>]]

Declaration of INTEGER variables with optional initialization.

Example:

```
INTEGER N,N12=12
```

***

[48K only] IRED [<Mode>][,<Timeout>]

Select infrared connection (HP-48G/G+/GX only) in LED/RedEye (Mode=0, default) or Serial (Mode<>0) mode. Additional timeout when receiving data over serial connection can be set as well (in 1/16 second units, from 0 to 15, default is 15 meaning 1 second total timeout which provides very large tolerance for serial communication). Those are global settings and will affect all programs and procedures/functions.

Examples:

```
IRED PRINT "REDEYE"
WIRE PRINT "RS-232"
```

***

LEAVE

Pops the innermost loop from the stack - for example, to allow the bad BASIC practice of exiting the loop with GOTO.

Example:

```
DISP "START"
FOR I=1 TO 10
_ FOR J=1 TO 5
__ DISP I,J LEAVE GOTO @1
_ NEXT
1 NEXT DISP "DONE"
```

***

[LET] \<Variable1\>=\<Expression1\>[,\<Variable2\>=\<Expression2\>] ... [,\<VariableN\>=\<ExpressionN\>]

Assignment statement. If the variable didn't exist prior to the assignment then it will be created automatically and the type will be determined by the type of the evaluated expression. LET statement is optional.

Examples:

```
J=5,K[10]=10,N=J+K[15]
LET A=123.456
LET E1[4]="123*456"
LET X[2,2]=1E25,S="ABCDEF"
C=[1.23E5,-666],D=2*C
```

In case of an string assignment the starting character can be specified by the expression in parenthesis after the variable name. The characters which excess the length of the destination string variable are ignored.

Examples:

```
S="ABCDEF"
S(4)="ABCDEF"
S[8](4)="ABCDEF"
S[8,1](4)="ABCDEF"
```

There are six special assignment "shortcut" statements:

```
VAR+Expression == VAR=VAR+Expression
VAR-Expression == VAR=VAR-Expression
VAR*Expression == VAR=VAR*Expression
VAR/Expression == VAR=VAR/Expression
VAR>           == VAR=VAR+1
VAR<           == VAR=VAR-1
```

Examples:

```
I+2             == I=I+2
I-2             == I=I-2
I*2             == I=I*2
I/2             == I=I/2
I>              == I=I+1
I<              == I=I-1
X[2,4]>         == X[2,4]=X[2,4]+1

IF NOT D OR X-Y=ABS D A[X]<
WHILE NOT A[X]@ X< A[X]< ENDWHILE
S> Y=X
```

***

[48K only] LINE \<ExprX1\>,\<ExprY1\> [TO \<ExprX2\>,\<ExprY2\>] [STEP \<Expression\>]

Draw a line between two display points (X1,Y1) and (X2,Y2). If the expression following the optional STEP keyword evaluates to 0 then a line will be cleared, if it evaluates to 1 then a solid line will be drawn (default behavior without STEP), otherwise pixels will be toggled (pixels which are turned on will be turned off and vice versa). Sets the new draw position to the second point. If the second point has been omitted then acts like PLOT.

See also: DRAW, PLOT

Example:

```
LINE 20,10 TO 80,50
LINE 20,50 TO 80,10 STEP 2
LINE 22,33 STEP 0
```

***

[48K only] LIST [<Expression>]
[48K only] LIST FN <Name>

List the current program to the printer. If an expression has been provided then:

(1) List from the specified numeric label until the end in case of an numeric expression with positive result or zero.
(2) List from the specified line until the end in case of an numeric expression with negative result.
(3) List from the specified string label until the end in case of an string expression.
(4) List the specified procedure or function in case of FN <Name>.

***

[48K only] LOAD

Load the BASIC program from the RPL stack and make it a current program. The program must be in the appropriate format - saved with the SAVE statement. This statement can be executed from command mode only.

***

LOOP [@]

Rewind the innermost FOR/NEXT, REPEAT/UNTIL or WHILE/ENDWHILE loop to the beginning and do the next iteration with condition checking (and with variable increment in case of FOR statement). When used with CASE/ENDCASE or WHEN/ENDWHEN statements it acts like EXIT.

See also: AGAIN

Example:

```
I=%1
REPEAT
_ DISP I
_ I+1
_ IF I>5 LOOP @
_ DISP "NEVER EXECUTED"
UNTIL I>10
```

***

[48K only] MAT <Operation1>[,< Operation2>] ... [,<OperationN>]

Perform various matrix operations one after another. The destination matrix must be of the same dimensions as the result of the requested operation. See section 2.6 for more info about arrays, matrices and the matrix engine.

The available operations are:

Matrix addition	`MAT C=A+B`
Matrix subtraction	`MAT C=A-B`
Matrix multiplication	`MAT C=A*B`
Scalar multiplication	`MAT B=(<Expression>)*A` or `MAT B=[<ComplexNumber>]*A`
Matrix inversion	`MAT B=/A`
Matrix negation	`MAT B=-A`
Matrix copy	`MAT B=A`
Matrix conversion	`MAT B=+A` (can convert REAL12 matrices to COMPLEX12 and vice versa)
Matrix transposition	`MAT B=!A`
Matrix determinant	`MAT X=*A`
Dot product	`MAT X=A.B`
Create complex matrix	`MAT C=A AND B` (create COMPLEX12 matrix from two REAL12 matrices)
Create identity matrix	`MAT A=#`

Note: A, B and C are matrices while X is a numeric variable.

Examples:

```
MAT REAL A[3,3],B[3,3]
A[0,0]=1 A[0,1]=0 A[0,2]=-3
A[1,0]=-1 A[1,1]=4 A[1,2]=3
A[2,0]=1 A[2,1]=2 A[2,2]=1
MAT B=A,B=/B,A=B*A
?A[0,0]; COL 7 ?A[0,1]; COL 14 ?A[0,2]
?A[1,0]; COL 7 ?A[1,1]; COL 14 ?A[1,2]
?A[2,0]; COL 7 ?A[2,1]; COL 14 ?A[2,2]
?B[0,0]; COL 7 ?B[0,1]; COL 14 ?B[0,2]
?B[1,0]; COL 7 ?B[1,1]; COL 14 ?B[1,2]
?B[2,0]; COL 7 ?B[2,1]; COL 14 ?B[2,2]


N=%9
MAT SHORT A[N,N]
FOR I=0 TO N-1
_ FOR J=0 TO N-1
__ A[I,J]=RND 1
_ NEXT
NEXT
MAT A=/A
```

***

[48K only] MAT COMPLEX <Matrix1>[,<Matrix2>] ... [,<MatrixN>]
[48K only] MAT CSHORT <Matrix1>[,<Matrix2>] ... [,<MatrixN>]

The declaration of COMPLEX12 matrices to use with the matrix engine.

```
CSHORT C,C12=[1,2]
```

***

[48K only] MAT REAL <Matrix1>[,<Matrix2>] ... [,<MatrixN>]
[48K only] MAT SHORT <Matrix1>[,<Matrix2>] ... [,<MatrixN>]

The declaration of REAL12 matrices to use with the matrix engine.

***

[48K only] MOVE <ExpressionX>,<ExpressionY>

Move the current draw position to the point specified.

See also: DRAW

***

[48K only] NAME "<Name>"

Rename the current program with the specified name. This statement can be executed from command mode only.

Example:

```
NEW "PRGM001"
NAME "PRGM002"
```

***

[48K only] NEW "<Name>" [TO]
[32K only] NEW

Create a new program with the specified name. With TO after the name an TEXT program will be created, otherwise an regular BASIC program will be created. This statement can be executed from command mode only. In 32K version NEW acts like PURGE without an argument.

***

OFF

Power the calculator off.

***

[48K only] OPEN

Open serial communication.

Note: PRINT (and DISP when redirected to printer) doesn't need serial port to be explicitly opened and closed.

***

[48K only] PAD [<Expression>]

Left/right padding for DISP statement. Positive values will pad spaces to the left while negative values will pad to the right. PAD without an expression will turn the padding off. This is a global setting and will affect all programs and procedures/functions.

Examples:

```
PAD 4 DISP 1;2;3; PAD ? "X"
PAD -4 DISP 1;2;3; PAD ? "X"
```

***

[48K only] PLOT <ExpressionX>,<ExpressionY> [STEP <Expression>]

Turn the pixel at given (X,Y) display coordinates on/off. If the expression following the optional STEP keyword evaluates to 0 then a pixel will be turned off, otherwise it will be turned on (default behavior without STEP). Sets the new draw position to the point specified.

Note: PLOT doesn't support pixel toggling so you'll have to use LINE X,Y if you need this functionality. In all other cases PLOT is preferred because it is faster.

See also: DRAW, LINE, PIXEL

Example:

```
PLOT 120,60
IF PIXEL(120,60) ? "PIXEL ON"
```

***

POP

Pop variable link(s) from the stack. See PUSH and section 2.7 for more info.

***

PRINT [<Expression1>][;<Expression2>][,<Expression3>] ...

Print the result of one or more expressions to the printer over wire or infrared port. In case of numeric expression the current FIX/SCI settings are used for printing. The following delimiters are allowed:

; ... The next expression will be printed immediately after the current one.
, ... The next expression will be printed one character after the current one.

Examples:

```
PRINT "S=";S
DISP "T=";T
PRINT A[1],A[2],A[3]
PRINT X;" ";Y;" ";Z
```

***

[48K only] PRINT TO

Redirect the output from DISP/?, CAT and STATE statements to the printer. This is a global setting and will affect all programs and procedures/functions. With this option active the command line will be printed after each entry as well. PRINT and LIST statements are always directed to the printer regardless of PRINT TO.

***

PROC <Name>[([=]<Param1>[,[=]<Param2>] ... [,[=]<ParamN>])]

Procedure definition. Declare the procedure with the optional parameter list. See section 2.7 for more info on structured programming. Parameters can be sent by value or by reference. Reference parameters must have = in front of them and in this case only an address will be sent, not the actual value, so procedure/function can change the original value. Arrays can be sent by reference only. Expressions cannot be sent by reference, of course. "By value" parameters can optionally be declared by type. Without the declaration the type of the parameter will be determined by the type of the associated expression.

Parameter declaration can include an optional expression initialization. Those expressions can use the special * operator which evaluates the expression from the parameter list. Procedures and functions are functionally equivalent and you can call procedure as function or vice versa providing that you include a correct RETURN statement with the result expression. This expression will never be evaluated during the procedure call but is needed for an function call.

Examples:

```
INTEGER X=123,Y[10],Z=789
FOR I=0 TO 9 Y[I]=I NEXT
Y[5]=456
DISP X,Y[5],Z
DISP "CALL" CALL XYZ,X,Y,Z DISP "DONE"
PROC XYZ(A,=B,C) DISP A,B[5],C

CALL FACT,50,1
PROC FACT(INTEGER N, REAL X)
INTEGER P=N-1 REAL Y=X*N
IF N<2 DISP X RETURN
CALL FACT,P,Y

CALL P,1,2,3 ' 6
CALL P,1+9,2,3 ' 15
CALL P,2*(1+9),2,3 ' 25
PROC P(REAL X=*+*+*) DISP X

REAL X CALL SUM,X,%10,1,2,3,4,5,6,7,8,9,10
DISP X DISP FN SUM(X,%5,1.1,2.2,3.3,4.4,5.5)
PROC SUM(=X,I LET X=0 FOR I=I LET X+* NEXT
END X
```

***

[48K only] PROGRAM "<Name>"

Make a program with the specified name to be the current program. Only current program can be edited and executed. This statement can be executed from command mode or from the program. If executed inside program it will run the program with the specified name - this allows chaining the programs during execution.

Example:

```
PROGRAM "PRGM001"
```

***

[48K only] PURGE ["<Name>"]

Delete a program with the specified name. If the name has been omitted then initialize the current program - delete all statements and leave an empty program in memory. This statement can be executed from command mode only.

Example:

```
PURGE "PRGM001"
```

***

PUSH <Variable1>[,<Variable2>] ... [,<VariableN>]

Push variable link(s) to the stack. All variables will be undeclared after PUSH if used from the main program - this basically allows redeclaring variables throughout the program. Variables cannot be redeclared in procedures/functions and the primary purpose of PUSH is to support recursion there (refer to section 2.7 for more info on recursion).

Examples:

```
X=%123 ? X
PUSH X
X=123.45 ? X
POP ? X

PUSH S0,D0,T0,N0
CALL HANOI,S0,D0,T0,N0-1
POP
```

***

[48K only] RANDOM

Randomize the seed for internal random numbers generator.

***

[48K only] READY [RECVB]

Wait for a byte to be received over the serial connection and continue execution. Connection should be opened prior to using READY. If followed by RECVB it will read the received byte and discard it. With handshake enabled it will also send the received byte back to the sender for additional 2-way control.

See also: RECV, SEND

***

READ <Variable1>[,<Variable2>] ... [,<VariableN>]

Read the next expression from the DATA list, evaluate it and assign the result to the variable. You can use almost all language elements in DATA expressions, including functions and all variables defined in the main program or inside a procedure/function where READ has been executed. But, due to the interpreter architecture, it is not possible to use starting character designation with string variables here e.g. STRING A READ A(1) is not allowed.

Main program only: If the variable didn't exist prior to READ it will be created automatically and the result of evaluated expression will be assigned to it. This doesn't apply to procedures/functions because all variables should be explicitly declared there.

Examples:

```
INTEGER A[10],X,Y,Z
READ X,Y,Z DISP X DISP Y DISP Z
READ A[1],A[2],A[3]
DISP A[1] DISP A[2] DISP A[3]
DATA 123,456,789,321,654,987
FOR I=1 TO 4
_ GOSUB @5
NEXT
END
5 RESTORE @"REST"
DISP "RES: ";
READ X,Y,Z
DISP X;" ";Y;" ";Z
RETURN
10 DATA 1,2,3,4
"REST" DATA 5,6,7,8
30 DATA 9,10,11,12
```

During the expression evaluation a special "=" (READ operator) can be used to read and evaluate the next expression from the DATA list.

Examples:

```
DATA 1,2,3
DATA 1+9,2,3
DATA 2*(1+9),2,3
DISP =+=+= ' 6
DISP =+=+= ' 15
DISP =+=+= ' 25
```

***

REAL <Variable1>[=<Expression1>][,<Variable2>[=<Expression2>]] ... [,<VariableN>[=<ExpressionN>]]

Declaration of REAL (REAL15) variables with optional initialization.

Example:

```
REAL X,X12=12
```

***

RECALL

Recall the last system value stored on the stack. See STORE for more info.

***

RELEASE

Release the stack frame. Possibly dangerous low level functionality - don't use!

REM, '

An remark inside program - the rest of the line is ignored during execution.

Examples:

```
REM DISP 123*456
DISP 123*456
' DONE
? "DONE"
```

***

REPEAT <Statements> UNTIL <Expression>

Conditional loop with the condition check at the end.

Example:

```
I=%1 X=1
REPEAT
_ X*I
_ I+1
UNTIL I>=70
? X
```

***

RESET

Destroy program context and create command mode context with 26 one-letter variables (A..Z).

***

RESTORE [@][<Expression>]

Reset DATA pointer to the specified numeric or string label (or to the beginning of the program if an expression has been omitted). If immediately followed by the DATA statement then a pointer will be set to this DATA statement.

See also: DATA, READ

Example of RESTORE DATA to emulate global variables:

```
REAL X=456 RESTORE DATA X CALL Q,123
PROC Q(X) REAL Y
READ Y ? X,Y
```

***

RETURN [<Expression>]

Return from the subroutine or procedure/function (with the result expression in case of an function return). END and PROC/FUNC statements will act like RETURN as well, if reached during the program execution. Only RETURN and END can be used to return an result from the function. The implicit end of the program will act as RETURN, too. It is possible to do a RETURN from inside a loop - the interpreter will remove all pending loops and restore all stored system values from the stack prior returning.

Examples:

```
GOSUB 10 DISP X
GOSUB 20 DISP X
CALL P10
X=FN F10 DISP X
DISP FN F(16)
END
```

```
10 X=123 RETURN
20 X=456 END
PROC P10 DISP "789"
FUNC F10 RETURN 321
FUNC F20 END 654
```

= can also be used to return the result from an function. This is done mainly to simplify the definitions of one-line functions like this one:

```
FUNC F(X)=X*X
```

***

RUN

Destroy command mode context, compile current program (if not already compiled), create the associated program context, initialize (clear) all variables and run the current program.

You can use a special READ/RUN constructs to send parameters from the command mode to the program and to extend RUN functionality in other ways:

RUN <Expression1>[,<Expression1>] ... [,<ExpressionN>] sets an initial DATA which should be READ at the start.
RUN <Statement1>[,<Statement1>] ... [,<StatementN>] executes statements before starting the program.

And the combination of the both of the above.

Examples:

Program mode:	`READ X,Y,Z DISP X+Y+Z`
Command mode:	`RUN 1,2,3`
Result:	`6`

***

[48K only] SAVE

Save the current program to the RPL stack. The character string object containing the program could then be transferred to other calculators and loaded into another HRAST-BASIC instance. There should be enough space inside RPL environment to save the program. I usually leave around 8K for this and allocate the rest for the BASIC interpreter.

***

[48K only] SEND [<Expression1>][,<Expression2>][,<Expression3>] ...

Send data over serial connection. Each REAL/COMPLEX value is converted to INTEGER and only the lowest byte is used for transmission. STRINGs are transmitted character by character. Connection should be opened prior to using SEND. Handshake can be turned on for additional 2-way control. In this case, SEND will wait for this handshake byte after each byte sent, compare it with the last byte sent and raise TRANSMIT ERROR if they don't match.

See also: HANDS, RECV, OPEN, CLOSE

Examples:

```
REM RECEIVER                              REM SENDER
WIRE OPEN HANDS 1                         WIRE OPEN HANDS 1
STRING S(255)                             ? "WAITING" READY RECVB
SEND 0 ? "RECEIVING"                      ? "SENDING"
S=RECV 255                                FOR I=1 TO 255
REM FOR I=1 TO 255 S(I)=RECV 1 NEXT       _ SEND I
REM FOR I=1 TO 255 S(I)=CHR RECV 0 NEXT   NEXT
REM FOR I=1 TO 255 READY S(I)=CHR RECVB NEXT    CLOSE
CLOSE                                     ? "DONE"
? "DONE"
```

SHORT <Variable1>[=<Expression1>][,<Variable2>[=<Expression2>]] ... [,<VariableN>[=<ExpressionN>]]

Declaration of SHORT (REAL12) variables with optional initialization.

Example:

```
SHORT X,X12=12
```

***

SKIP <Expression>[@]

Execute the next statement on the same line if the result of an expression is True or skip it if the result is False.

Example:

```
SKIP 1<2 DISP "1<2" DISP "ALWAYS"
```

***

[48K only] SLEEP

Turn the calculator off but, unlike OFF, timer and serial port interrupts can wake it up.

Example:

```
FIX 4
REPEAT TIMER 5*16 LEAVE ? "TIME:",TIME WAIT 2*16 UNTIL 0
REPEAT
_ ? "SLEEP"
_ SLEEP
UNTIL 0
```

***

[48K only] START [@][<Expression>]

Change label search start line to the specified numeric or string label (or to the beginning of the program if an expression has been omitted). The successive GOTO, GOSUB, RESTORE, EDIT and LIST statements will use this line as the new start when searching for a label.

See also: GOTO, GOSUB, RESTORE, EDIT, LIST

Example:

```
START 30 GOTO 1
10
1 ? "LABEL #1" END
20
1 ? "LABEL #2" END
30
1 ? "LABEL #3"
START 20 GOTO 1
```

***

[48K only] STATE

Show the state of various interpreter parameters (DEG/RAD/GRAD, FIX/SCI & USER).

***

STOP

Stop program execution and return to the command mode. ON key will break the running program and generate STOP as well. The program context will remain to be active inside program environment and you can use all variables created by the program to look at the results or make additional manual calculations using those variables. Use CONT statement to continue execution.

[48K only] Use TRON statement after stopping the execution to activate debugger and single-step through the program in trace mode.

***

STORE <SystemValue>

Store system value on the stack. This is the list of currently supported system values:

0 = DATA pointer	6 = Display position and attributes
1 = Error TRAP pointer	7 = Display attributes / flags
2 = FIX/SCI	8 = PAD value [48K only]
3 = FIX/SCI and Angle mode	9 = TIMER data [48K only]
4 = Angle mode	10 = START value [48K only]
5 = Display position	11 = Draw position [48K only]

Examples:

```
REAL A=1234.567809 FIX 2 STORE 2
CALL Q RECALL DISP A
PROC Q FIX 3 REAL I=4 REPEAT DISP I STORE 2 FIX 4
I< LOOP DISP "I=";I
I< UNTIL NOT I DISP I STORE 2 FIX 5
```

***

STRING <Variable1>[=<Expression1>][,<Variable2>[=<Expression2>]] ... [,<VariableN>[=<ExpressionN>]]

Declaration of STRING variables with optional initialization.

Examples:

```
STRING S,S12="12",S23(34)="23"
STRING S55[5,5](8)
```

***

SWAP <Variable1>,<Variable2>

Swap/exchange two variables. Variable links are exchanged internally, so it is not possible to swap the values of two array elements etc.

Example:

```
REAL A[2,3] S="ABCDEFG"
A[1,2]=3 ? A[1,2],S SWAP A,S ? A,S[1,2]
```

***

[48K only] TIMER [<Interval>][@] <Statements>

Set the timer handler. When the timer is due the current program flow will be abandoned, the timer will be turned off and the execution will resume with the statements following TIMER. Without parameter, the TIMER statement will turn the timer off. SLEEP statement can be used to put the calculator to sleep and wake it up after the specified interval. TIMER is automatically disabled in command mode.

Example:

```
FIX 4
? "START"
REPEAT TIMER 5*16 LEAVE ? "TIME:",TIME WAIT 2*16 UNTIL 0
REPEAT ?
_ "A=";
_ INPUT A
UNTIL 0
? "NEVER EXECUTED"
```

***

[48K only] TRON

Trace mode ON. The interpreter will stop program execution, activate the debugger and enter into trace mode. In this mode single-stepping (Left Shift + S) through the program (including subprograms, procedures and functions) is possible. The execution will stop after each instruction and the command mode will be activated so you can inspect variables etc. CONT statement will turn trace mode off and resume normal execution. TRON can be used from the program or from a command line (but only if a running program has been stopped with STOP or ON/BREAK). Use RUN TRON to activate trace mode from the start of the program.

***

[48K only] TROFF

Trace mode OFF. The debugger will be deactivated and the normal program execution will be resumed.

***

TRAP [@] <Statements>

Set an error-trapping handler. If an error occurrs in program then execution will resume with the statements following TRAP. MEMORY FULL error cannot be trapped. TRAP is automatically disabled in command mode.

Example:

```
1 DISP "INPUT? "; TRAP ERROR DISP "TRY AGAIN: ";ERR
INPUT A UNTRAP DISP A GOTO 1
```

***

UNTRAP

Disable the current TRAP statement. Error trapping handler is automatically disabled inside a debugger / trace mode.

***

WAIT [<Duration>]

Wait for the specified numbers of 1/16 second units. If the result of the expression is positive then a delay will be skipped if a key has been pressed during WAIT. If the result is negative and a key has been pressed during WAIT then execution will be suspended until the key is released. WAIT 0 waits for a key and then execution continues without delay until the key is released. WAIT without an argument waits for a key and suspend execution until the key is released. In all cases the code of the key pressed during WAIT will be stored for use with KEY/INKEY functions.

Note: Timer and serial port interrupts can terminate WAIT as well.

```
X=1 REPEAT
_ ? X WAIT 8
_ ? KEY
_ X+1
UNTIL 0
```

WATCH

Start the stopwatch and reset TICKS timer to zero.

Example:

```
WATCH
X=1
FOR I=69 X*I NEXT
FIX 3 ? "TIME:"; COL 6 ? TICKS;"s"
FIX ? "RES:"; COL 6 ? X
```

***

[48K only] WIRE [<BaudRate>][,<Timeout>]

Select RS-232 connection over the serial cable with optional baud rate setting (0 = 1200 bauds, 1 = 1920 bauds, 2 = 2400 bauds, 3 = 3840 bauds, 4 = 4800 bauds, 5 = 7680 bauds, 6 = 9600 bauds, 7 = 15360 bauds). The default setting is 6 (= 9600 bauds). Additional timeout when receiving data over serial connection can be set as well (in 1/16 second units, from 0 to 15, default is 15 meaning 1 second total timeout which provides very large tolerance for serial communication). Those are global settings and will affect all programs and procedures/functions.

Example:

```
WIRE 2,3
PRINT "RS-232 @ 2400 BAUDS"
```

***

WHEN <Expression>[@] <Statements1> UNLIKE [@] <Statements2> ENDWHEN

Multiline conditional statement. I developed WHEN before I developed support for program blocks (so IF can be spanned over more lines) but I decided to keep it because WHEN/UNLIKE combination is very powerful and in many cases I prefer it over an multiline IF. You can even use EXIT statement to exit from WHEN/ENDWHEN prematurely. LOOP statement acts like EXIT if used with WHEN/ENDWHEN.

Example:

```
WHEN 1<2
_ ? "1<2"
_ WHEN 3<4 ? "3<4" UNLIKE ? "3>4" ENDWHEN
UNLIKE
_ ? "1>2"
_ WHEN 5<6 ? "5<6" UNLIKE "5>6" ENDWHEN
ENDWHEN
? "DONE"
```

***

WHILE <Expression>[@] <Statements> ENDWHILE

Conditional loop with the condition check at the beginning.

Example:

```
I=%1 X=1
WHILE I<70
_ X*I
_ I+1
ENDWHILE
? X
```

***

[48K only] ZERO <Variable1>[,<Variable2>] ... [,<VariableN>]

Zero the content of a particular variable(s). Numeric variables will be set to zero (0) while STRING variables will be set to an empty string. All elements of arrays and matrices will be zeroed.

Example:

```
REAL A=123 INTEGER B=456 STRING S="XYZ"
? A,B,S
ZERO A,B,S
? A,B,S
```

***

{ }

Delimiters used to combine more statements/lines into single entities. See 2.7. for more info about blocks.

***

⌐ _

Delimiters used for indentation inside program. They are skipped during execution but don't overuse them because, although small, they still need some time to execute. Indentation delimiters must appear at the beginning of the line (after line label if one exists).

***

7. Initialization

7.1 Default/Initial settings on the start of the interpreter:

ARRAY 0 (arrays are zero-indexed)
RAD
FIX -1 (FIX/SCI off)
ATTR 0
COL 0
PAD 0
TAB 0
WIRE 6,15 (9600 bauds RS-232 cable, timeout = 1 sec.)
DISP TO (DISP directed to the LCD)
UNTRAP (Error trapping off)
TIMER 0 (Timer off)
TROFF (Trace mode off)
MOVE 0,0

7.2 Default/Initial settings after RUN statement:

ARRAY 0 (arrays are zero-indexed)
RAD
FIX -1 (FIX/SCI off)
RESTORE (DATA pointer set to the beginning of the program)
START (Label search pointer set to the beginning of the program)
UNTRAP (Error trapping off)
TIMER 0 (Timer off)
TROFF (Trace mode off)
MOVE 0,0
Last key code = 0

7.3 Default/Initial settings after CONT statement:

TROFF (Trace mode off)
Last key code = 0

7.4 Default/Initial settings after OFF statement:

TIMER 0 (Timer off)

8. Errors

After the program encounters an error the execution will be terminated, error code will be stored to ERR pseudovariable and the error position will be updated so the editor can show the exact position where a problem occured. In general, syntax errors detected in command or program mode will put the cursor over the character which the interpreter didn't understand. Program errors will position the cursor after the program element (statement, operator, variable, etc.) where the error occured. In both cases the interpreter will echo back the detokenized line so you can see what exactly interpreter understood and what it didn't understand.

This is a list of all possible errors (error code & message):

01 SYNTAX ERROR

02 DATA ERROR

03 ZERO DIVISION

04 NONEXISTENT

05 LABEL ERROR

06 MISMATCH

07 INDEX ERROR

08 SIZE ERROR

09 ALREADY EXISTS

10 TOO MANY IDENTS

11 RANGE ERROR

12 NO DATA

13 NOT ALLOWED

14 TOO LONG

15 MEMORY FULL

16 MATRIX ERROR

17 TIMEOUT

18 TRANSMIT ERROR

19 RECEIVE ERROR

BATTERY LOW (on start only)

9. Keyboard Layouts

Alpha Code
Left Shift
Right Shift
Normal

9.1 HP-48G/G+/GX Keyboard Layout

a 01	*b* 02	*c* 03	*d* 04	*e* 05	*f* 06
			Delete Line		
A	**B**	**C**	**D**	**E**	**F**

g 07	*h* 08	*i* 09	*j* 10	*k* 11	*l* 12
		Insert		First Line ▲	Lower Case
G	**H**	**I**	**J**	**K**	**L**

m 13	*n* 14	*o* 15	*p* 16	*q* 17	*r* 18
Merge Lines '	New Line		Home ◄	Last Line ▼	End ►
M	**N**	**O**	**P**	**Q**	**R**

s 19	*t* 20	*u* 21	*v* 22	*w* 23	*x* 24
SST		Dup Line		Swap Lines ^	
S	**T**	**U**	**V**	**W**	**X**

= 27	*y* 25	*z* 26	45	28
Result =				DEL
ENTER	**Y**	**Z**	**DEL**	←

α **Alpha**	& 37	@ 38	? 39	# 44
	~		|	)
	&	‾@	?	(()
	7	**8**	**9**	**/**

◄ **Left Shift**	$ 34	% 35	^ 36	‾] 43
	$	%	^	]
				[[]
	4	**5**	**6**	*****

► **Right Shift**	! 31	" 32	# 33	" 42
				>
	!	"	#	<
	1	**2**	**3**	**−**

OFF **OFF** **ON/BREAK**	= 30	, 40	*SPC* 29	: 41
		;	;	}
	=	,	:	{ {}
	0	**.**	**SPC**	**+**

46

9.2 HP-49G/49G+/50G Keyboard Layout

a 01 A	*b* 02 B	*c* 03 C	*d* 04 Delete Line D	*e* 05 E	*f* 06 F

g 07 G	*h* 08 H	*i* 09 Insert I		48 First Line ▲

j 10 J	*k* 11 K	*l* 12 Lower Case L	46 Home ← ◄	49 Last Line ▼	47 End DEL ►

m 13 Merge Lines M	*n* 14 New Line N	*o* 15 ' O	*p* 16 P	28 DEL ←

q 17 ^ Q	*r* 18 R	*s* 19 SST S	*t* 20 T	*u* 21 Dup Line U

v 22 V	*w* 23 Swap Lines = W	*x* 24 < X	*y* 25 > Y	*z* 26 / Z /

	& 37 ~ & 7	@ 38 _ @ 8	? 39 | ? 9	" 43] [*	[]
α **Alpha**					

	$ 34 $ 4	% 35 % 5	^ 36 ^ 6	_ 42) (–	()
◄┐ **Left Shift**					

	! 31
	!
	1
		" 32
		"
		2
			# 33
			#
			3
				41
				{
				}
				+
					{}

┌► **Right Shift**					

	= 30 = 0	, 40 : , .	*SPC* 29 ; : **SPC**	= 27 Result = **ENTER**
OFF **OFF** **ON/BREAK**				

A. [48K only] Statement Abbreviations

The following statement abbreviations are available while entering/editing command and program lines:

C.	CALL
CO.	CONT
CX.	COMPLEX
D.	DISP
DA.	DATA
DW.	DRAW
EW.	ENDWHILE
F.	FOR
FU.	FUNC
G.	GOTO
GS.	GOSUB
I.	INPUT
IN.	INTEGER
L.	LINE
M.	MOVE
N.	NEXT
P.	PROC
PG.	PROGRAM
PR.	PRINT
R.	RUN
RD.	READ
RL.	REAL
RE.	REPEAT
RT.	RETURN
RS.	RESTORE
S.	STOP
SG.	STRING
ST.	STEP
U.	UNTIL
W.	WHILE
Z.	ZERO

Note: You have to be careful with abbreviations because you can eventually get into a situation where a particular program line couldn't be decompiled anymore because it will be too big for the edit buffer which is 127 characters long.

For example, this will produce a perfectly legal line with 15 ENDWHILE statements:

```
EW. EW. EW. EW. EW. EW. EW. EW. EW. EW. EW. EW. EW. EW. EW.
```

But this line requires 15 * 8 + 14 = 134 characters after decompiling and will not fit into the edit buffer anymore!

B. Known Issues

(1) Substring operator cannot be used with INPUT and READ.

For example, this doesn't work:

```
STRING SS(10)="ABCDEF"
INPUT SS(2)
READ SS(2)
DATA "123"
```

Intermediate variable has to be used instead:

```
STRING S(10),SS(10)="ABCDEF"
INPUT S SS(2)=S
READ S SS(2)=S
DATA "123"
```

(2) Global variables could contain invalid links if used incorrectly - wrong values, INDEX ERROR, DATA ERROR and similar artifacts can be experienced.

For example, if a global variable wasn't declared in a main program but inside a procedure or function (or was declared but without value assigned):

```
CALL Q
B=456
? "~A=";~A,"B=";B
PROC Q REAL ~A
~A=123 CALL W
? "Q~A=";~A
PROC W
? "W~A=";~A
```

In the above example, the value of ~A global variable inside a main program will be incorrect (456) because the variable link still points to the memory location allocated inside Q procedure. This link is no longer valid after returning to the main program and has been reallocated by the assignment of B variable. So, both ~A and B variable links now point to the same memory location!

Always declare global variables inside a main program only! If you really know what you are doing then you can declare them inside procedures or functions, but you should use such pseudo-global variables only inside a procedure or function where they have been declared (and only after they have been declared) and all nested procedures/functions, not from the main program or from procedures/functions called before they have been declared.

Correct global variable usage:

```
REAL ~A=789
? "~A=";~A
CALL Q
B=456
? "~A=";~A,"B=";B
PROC Q
~A=123 CALL W
? "Q~A=";~A
PROC W
? "W~A=";~A
```