Simple Interpreter for Embedded Computers

Version 01.07.03

February 17, 2011

TABLE OF CONTENTS

1. Overview

1.1. Background

Embedded systems include a wide range of execution environments – ranging from stand-alone (run-at-reset, or primitive RTOS with a flash file system), to emulator-based (E-JTAG, JTAG, BDM, etc.) execution with remote file access and other nice features, to "embedded Linux" (a full-blown OS.) Interp can be built for all these execution environments. (See Appendices C through E.)

Interp is indebted both to Mouse and FIG-forth, but the language itself is mouse-like rather than FIG-forth-like (See *Mouse, a language for microcomputers*, Grogono, Peter, 1983, Petrocelli Books Inc.)

You may download interp from Source Forge: http://interp.sourceforge.net/

1.2. Reverse-Polish Notation (RPN)

Everyone recognizes ordinary algebraic notation:

```
1 + 2 = 3
```

but postfix, or RPN may not be familiar to you. It is a streamlined, computer-oriented notation invented by Jan Lukasiewicz that eliminates the equal sign ("=") and the need for parentheses. Because of Hewlett-Packard, Inc., it is now customary for the math operator to appear after (to the right of) its operands, and the notation bears their moniker, Reverse-Polish Notation or RPN. The previous example looks like this in RPN:

```
1 2 +
```

With RPN, Interp, like FIG-forth and Mouse before it, simply applies this same idea to a programming language. The operands and arguments are automatically transferred to a traditional last-in-first-out data stack, as they appear (in order from left to right.) When a math operator, function call, etc., is encountered, it is immediately executed. It will take the necessary arguments from the data stack.

In the simple addition example, shown above, the 1 is read first, and pushed onto the top of the data stack. Then the 2 is read and pushed on the data stack. It is now "in front of" the 1. Then the addition operator ("+") is read and executed. It removes the top two items from the data stack, adds them and puts the result on top of the data stack (usually just referred to as "the stack.")

2. System Architecture

The interpreter is designed as two nested shells. The first shell- outer interpreter, receives serial input, preprocesses it, and passes it to the second shell, the interpreter proper- or inner interpreter, for execution. The "`interp>`" prompt is the outer interpreter's request for input from the world outside the computer. (See 3. Controlling the Outer Interpreter for more details.)

The inner interpreter (`interp()`), receives a zero-terminated string from the outer interpreter (`imain()`.) This string contains the commands to be executed. The zero-terminated string (c string) is the execution unit for the interpreter, which means the content of the string must contain complete function, if/then/else and while/until structures. The default maximum input length allowed, by the outer interpreter, for unbuffered strings is 4,096 bytes. If input buffering is enabled in the outer interpreter, the default maximum input length allowed is 65,536 bytes. In either case, these outer interpreter limits only restrict the maximum size of a conceptual unit - a single function, or just a line of input if you're not defining a function, not of your entire program.

The interpreter implements a stack-based, 32-bit, integer, virtual machine that executes single byte ASCII character opcodes, called tokens, operators, or commands. Some opcodes specify a class of related operations (e.g. stack operators) and are followed by a sub-operator that specifies the operation to be performed. Interp supports the standard operators: +, -, *, /, %, <<, >>, &, |, ^, and ~ (bit inversion), with their usual c-like meanings. (See 4. Categorical List of Interpreter Commands and 5. Alphabetical Listing of Interpreter Commands for details.)

It also supports byte (8-bit), halfword (16-bit), and word (32-bit) fetches and stores, and all the normal c-like comparison operators (<, >, >=, <=, <> (not equal), and = (equal).) The structured programming elements supported by the language are if/then/else and looping (while and until-like structures.) Formatted I/O and user-defined functions are also supported.

The programming building block is the function (or program.) Functions have a default of 1,024, 32-bit local variables (also accessible as halfwords or bytes.) Plus, functions may define local functions that are visible only to themselves (the parent function), and other local functions (siblings of the same parent.) The local functions also have their own local variables. All functions can access the default of 1,024, 32-bit global variables (also accessible as 2,048 halfwords or 4,096 bytes), and call other user defined, global functions. (See 4.4 Global and local variables for more details.)

3. Controlling the Outer Interpreter

Special commands that start with '#' in column 1 alter the behavior of the outer interpreter. Some are intended for interactive use, others are more commonly used in input files:

`#assign` *`name=sym`*	create a macro whose value was pre-computed by the inner interpreter and placed on the top of the data stack. After the assignment, the item is removed from the stack. The assignment is controlled by the value of *sym*, which may be one of the following: %n - pop the data stack, convert the value to its equivalent HEX-ASCII representation and use the result as the definition. %s - pop the data stack, treat the value as a pointer to a zero-terminated string, and use the string as the definition.
`#BUFFER`	buffer all lines that follow, up to the first line containing "`#EXECUTE`". The input lines are concatenated into a string in the order received. Leading blanks are collapsed into a single blank. Trailing blanks are removed. Blank lines are ignored completely. During buffering, the prompt is "\> ".
`#CONTINUE`	do not stop after an error when reading redirected input. Typically, this directive is used for testing language features.
`#define` *`name=defn`*	create a macro called *name*, with *defn* as the substitution text. Whenever *name* appears in the text, it is replaced with *defn*, and then rescanned for additional substitutions.
`#DISPLAY`	display all buffered input, if present. If no buffered input is present, do nothing. The "`#DISPLAY`" must appear between the "`#BUFFER`" and the "`#EXECUTE`". Typically, just before the "`#EXECUTE`".
`#erase`	erase all existing macro definitions.
`#EXECUTE`	terminate the buffered input, if present, with a zero and execute it. If no buffered input is present, do nothing. Input following the "`#EXECUTE`" is not buffered. Use another "`#BUFFER`" to re-initiate input buffering.
`#MACROEXPANSION`	(default) enable macro expansion.
`#macros`	list all existing macro definitions, newest first.
`#NOMACROEXPANSION`	disable macro expansion.
`#normal`	disable verbose-style error messages
`#PROMPT`	enable the interpreter prompt, useful at the end of files passed to interp.
`#quit`	stop after an error when reading redirected input.
`#restore`	Restore (add) the default macro definitions.
`#SILENCE`	disable interpreter prompt, useful at the beginning of files passed to interp.
`#USAGE`	list macro memory usage information intended as a guide for adjusting MAXPTR, MAXTBL, MAXDEF, and BUFSIZE (in the Makefile.)
`#undefine` *`name`*	delete the macro called *name*.
`#verbose`	enable verbose-style error messages

Comments start with two underlines ("__".) Everything after the comment marker, to the end of the line is ignored. You can use comments on a line-by-line basis to annotate your code or on separate lines, above sections of code.

The outer interpreter makes it easier for you to create programs that can be executed by the (inner) interpreter. Since the interpreter executes a single c string, the outer interpreter allows your program to exist in a file as a collection of lines of text. The first line of a multi-line function would be "`#BUFFER`". Then you can indent your commands any way you want, on the lines that follow. The last line of your function is "`#EXECUTE`". Examples 16, 17, 18, and 19 in Chapter 7, illustrate this.

Macros are a simple way to extend interp in ways that make your programs easier to develop and maintain. Here are common ways to improve your code with macros:

1. Create meaningful names for your global and local variables, etc.
2. Create alternative names for many of interp's operators.

The first suggestion is up to you, but here are examples:

```
#define Gwtotal=0V
#define Lbchar=0bL
#define NoOpAddr=10K
```

Predefined macros are provided for some of interp's operators. See Appendix A.

4. Categorical List of Interpreter Commands

4.1. Integer and character constants

```
0 1 2 3 4 5 6 7 8 9 0n 0xn 0Xn '
```

Numbers are pushed onto the data stack in left-to-right order. The last, or right-most number is on the top of the stack. Any numbers that start with a leading '0', such as `0377` or `011` are treated as OCTAL (base 8) numbers. Any numbers that start with a leading '0x' or '0X', such as `0x7f`, or `0xC800102C` are treated as HEXADECIMAL (base 16) numbers. Numbers that do not start with a zero, such as `4` or `237` are treated as DECIMAL (base 10) numbers. It is good practice to end numbers, especially hex numbers, with a space character (no-op) to prevent the command that follows from being interpreted as part of the number. All numbers on the data stack are represented as 32-bit values.

Only positive numbers may be entered directly, number signs are not allowed. If you need a negative value, put the positive value on the stack and follow it with the 'N' command to negate it (See 4.5. Math operators below for more details.)

To put the ASCII value of a single character on the stack, start with a single-quote character (" ' ".) The next character following the single quote is evaluated and its ASCII value is pushed on the stack.

WARNING:
If the " ' " operator is used inside of functions, if/else/end-if or loop/while structures, the character following the " ' " should not be one of these:
` [;] { } ^ \ ( )
Use the decimal or hexadecimal representation of these ASCII characters instead.

4.2. Stack operators

```
SC SD Sc Sd Sn So Sp Sr Ss 2drop 2dup depth drop dup over pick rot swap
```

The stack operators allow you to manipulate values directly on the stack without using local variables, global variables, or allocated memory. These operators are most often used to prepare input values for a function or command. Since most commands and functions consume stack values, you will often want to duplicate or arrange the stack values prior to calling the function or command that will consume them.

A special notation is used to document the stack use of all operators and user-defined functions. For the STACK-COPY command ("Sc" or “dup” macro), the stack diagram looks like this:

(n -- n n)

The items inside the parentheses represent the state of the stack before and after the "Sc" operator executes. The items to the left of the "--" illustrate the data that STACK-COPY needs as input. The items to the right of the "--" show the state of the stack after STACK-COPY has executed. When there is more than one data item shown, either as input or output, the right-most item is the top of the stack. In this particular example, STACK-COPY's stack output shows two items. The rightmost item is the top of the stack.

The operations performed by the stack operators are synonymous with their names; STACK-COPY ("Sc") duplicates the item on top of the stack; STACK-2COPY (“SC” or “2dup” macro) duplicate the top two stack items; STACK-DROP ("Sd" or “drop” macro) pops the item from the top of the stack and discards it; STACK-2DROP (“SD” or “2drop” macro) drops the top two items from the stack; STACK-NUM ("Sn" or “depth” macro) returns the

number of items already on the stack (not counting itself); STACK-OVER ("So" or “over” macro) makes a copy of the second item on the stack (the item below the top item), and places it on top of the stack; STACK-PICK (“Sp” or “pick” macro) makes a copy of the nth stack item on the top of the stack; STACK-ROT (“Sr” or “rot” macro) moves the third stack item to the top of the stack; STACK-SWAP ("Ss" or “swap” macro) reverses the order of the top two items on the stack.

Stack full and stack empty conditions are detected by all operators. If either of these conditions occur, a message will be displayed, execution of your command line or function will stop, and the "interp>" prompt will be displayed.

4.3. Memory operators

```
! @ MC MR MW Ma Mf Mc Ml Mm Ms b! b@ bC bR bW h! H@ hC hR hW malloc free
```

Sometimes you will need to manipulate memory-mapped peripheral devices, or access larger amounts of data than is practical to handle on the stack. The rich set of memory operators allow you to access 8-bit, 16-bit, and 32-bit memory. You can also allocate memory in 8, 16, or 32-bit groups and copy regions of 8, 16, or 32-bit memory.

When 8-bit, or 16-bit values are read from memory and placed on the data stack by the BYTE-FETCH ("b@") or HALFWORD-FETCH ("H@") commands, they are treated as signed numbers and converted to 32-bit values. This means that sign-extension occurs when an 8-bit value greater than 127 (0x7F) or a 16-bit value greater than 32,767 (0x7FFF) is read from memory.

Occasionally, sign-extension from reading 8, or 16-bit values can produce undesirable results. If you need to read byte or halfword values from memory and treat them as unsigned numbers, use the AND operator to discard the bits you don't want. For 8-bit values, AND with 0xFF. For 16-bit values, AND with 0xFFFF. For example:

```
#define VERNUM=0xB8A0C610
...
{q VERNUM h@ 0xFFFF & " Fribbet version code: #h\n"}
```

This function definition reads a 16-bit halfword, discards the upper 16-bits, because of undesirable sign-extension, and displays the value in hexadecimal notation. This use of macro definition (“#define”) shows a way to factor hardware specifics out of function definitions. These macro definitions can be grouped together at the beginning of the file to make them easier to find and update.

The operators for reading bytes (8-bits), halfwords (16-bits), and words (32-bits) are BYTE-FETCH ("b@"), HALFWORD-FETCH ("H@"), and FETCH ("@") respectively. The operators for writing bytes, halfwords, and words are BYTE-STORE ("b!"), HALFWORD-STORE ("h!"), and STORE ("!") respectively.

Memory can be allocated as sequential groups of 32-bit words with the MALLOC command ("Ma" or “malloc” macro), and released with M-FREE ("Mf" or “free” macro.)

Memory-to-memory copy operations for groups of bytes, halfwords, and words are accomplished with the BYTE-COPY ("bC"), HALFWORD-COPY ("hC"), and MEM-COPY ("MC") commands, respectively.

Memory-mapped I/O ports can be read and written as bytes, halfwords, and words with the BYTE-READ (“bR”), BYTE-WRITE (“bW”), HALFWORD-READ (“hR”), HALFWORD-WRITE (“hW”), MEM-READ (“MR”), and MEM-WRITE (“MW”) commands, respectively.

Two regions of memory can be compared with the MEM-COMPARE ("Mc") command.

The M-SEARCH ("Ms") command allows a specified region of memory to be searched for an arbitrary string of bytes. The byte string is required to start in the specified region, but it may end either inside or outside the region.

If you are using Linux (or any POSIX-compliant OS) on your embedded system, you already know that you cannot directly access memory-mapped hardware from your own programs. The MAP-MEMORY ("Mm") command provides this capability using a concept called "mapping." You specify the size and start address of the hardware memory region, and MAP-MEMORY will "map" that range of addresses into system memory. Simply use the base address returned by MAP-MEMORY as the proxy (stand-in) for the hardware memory base address.

The M-LENGTH ("Ml") command returns the length of a zero-terminated string. The returned length does not include the terminator.

4.4. Global and local variables

```
L V bL bV hL hV l
```

Storage of individual data values or simple data structures can often be accomplished through the use of either global or local variables. Both storage areas are arrays 4,096 bytes long, and can be accessed as bytes (8-bit), halfwords (16-bit), or words (32-bit). The individual storage locations are accessed with an index. For byte access, the index range is 0-4095. For halfword access, the index range is 0-2047. For 32-bit word access, the index range is 0-1023. The L-VAR ("L") and VAR ("V") commands convert an index specifying the nth 32-bit word into the address for that local or global variable, respectively. For example:

```
0 V @
```

passes index zero to VAR. VAR returns the address of that global, 32-bit location. Then FETCH ("@") reads the 32-bit value from that address.

The BYTE-VAR ("bV") and L-BYTE-VAR ("bL") commands convert an index specifying the nth 8-bit byte into the address for that global or local byte value, respectively. For example:

```
0x7f 23 bL b!
```

stores 0x7f into the local byte variable at index 23. Please note that the BYTE-STORE ("b!") command is used to store into the local byte variable. You must always remember to use the correct fetch and store commands for the size of the data that you are accessing, because execution will cease when an unaligned address is detected.

The HALFWORD-VAR ("hV") and L-HALFWORD-VAR ("hL") commands work in a very similar way to convert indexes to global and local halfword variables, respectively.

The L-BASE ("l") operator returns the base address of the caller's local variables. This can be very useful when passing data to a function through local variables, because it allows the function to access the caller's local variables. No attempt is made to protect the caller's local variables from inadvertent writes.

4.5. Math operators

```
% * + - / N a
```

The usual integer operations of addition ("+"), subtraction ("-"), multiplication ("*"), and division ("/") are supported, along with remainder ("%") and negation ("N"). Negation is essential since it is not possible to directly enter a negative number without resorting to hexadecimal or octal notation. Another, less popular but important operator is ABSOLUTE-VALUE ("a".) This operator converts negative numbers to their positive equivalent.

Divide-by-zero is not allowed, and is considered to be an error. If this happens, a message is displayed and execution of your command line or function will stop, and the "interp>" prompt will appear.

4.6. Bit-wise operators

```
& << >> ^ | ~
```

In order, the above operators are AND, SHIFT-LEFT, SHIFT-RIGHT, EXCLUSIVE-OR, OR, and NOT (bit inversion.) As you will see in 4.8 below, you can use &, ^, |, and ~ in connection with the comparison operators to produce complex comparison phrases.

4.7. Comparison operators

```
< <= <> = > >=
```

In order, these operators are LESS-THAN, LESS-THAN-OR-EQUAL-TO, NOT-EQUAL, EQUAL, GREATER-THAN, and GREATER-THAN-OR-EQUAL-TO. If the comparison result if FALSE, a zero is left on the stack. If the comparison result is TRUE, a -1 (0xFFFFFFFF) is left on the stack.

4.8. Boolean operators

```
& ^ | ~
```

These are the AND, EXCLUSIVE-OR, OR, and NOT operators, respectively. Because the comparison operators return -1 for TRUE and 0 for FALSE, it is possible to use these bit-wise operators to create compound conditional statements. For example, if you need to know if the value on top of the stack is greater than zero but less than 10, without consuming the original value, use this phrase:

```
Sc 0 > So 10 < &
```

Alternatively, you can use the default macro definitions to make more readable:

```
dup 0 > over 10 < &
```

The result will be TRUE (-1) if both conditions are met, and FALSE (0) otherwise. And the original value will be underneath the boolean result.

4.9. I/O operators

```
" M" , . : ? ?s M? p Pd Po Px
```

The QUOTE (") operator allows you to display strings, and formatted and unformatted numbers on the screen. This allows you to display results, and prompts. The M-QUOTE (M") command performs a related function, but prints to memory. The READ-CHAR (":"), READ-NUM ("?"), and READ-STRING ("?s") operators allow you to read individual keys, numbers, or strings from the user. The POLL ("p") command allows you to find out if input is waiting to be read. (See Appendix C.2. Buffered Input with Embedded Linux.) The M-READ-NUM ("M?") command allows you to read a digit string from memory, convert it, and push the resulting number on the stack. The PRINT-CHAR (",") and DOT (".") commands output individual characters and numbers respectively. The DOT command always displays numeric values in the current output number base, followed by a carriage-return. The output number base is affected by the PRINT-DECIMAL ("Pd"), PRINT-OCTAL ("Po"), and the PRINT-HEX ("Px") commands. For example, converting from decimal to hexadecimal can be done like this:

```
461 Px .
```

The displayed result is:

```
0x1cd.
```

4.10. If/else/end-if operators

`[ ; ]` if else endif

When you need to make decisions based on the comparison and/or boolean operators mentioned above, you could use the structured-programming if/then/else construct. If the value on top of the stack is TRUE (-1), then the commands following the '[' (IF operator or “if” macro) are executed up to the first occurrence of either ';' (ELSE operator or “else” macro) or ']' (END-IF operator or “endif” macro), which ever comes first. On the other hand, if the value on top of the stack is FALSE (0), then the commands following the '[' are skipped over, and execution will resume with the commands that follow the first ';' or ']', which ever comes first. For example:

```
10 0 > [ "Good!\n" ; "Not Good.\n" ]
```

will print "`Good!`", but:

```
10 0 < if "Good!\n" else "Not Good.\n" endif
```

will print "`Not Good.`"

Nested if statements are supported to an unlimited depth. Notice how much more readable the second if statement is when compared to the first one.

4.11. Loop/while/end-loop operators

`( \ )` loop while endloop

When you need to perform the same operations over-and-over, these three operators are all that are needed. Typically, you place some values on the stack, enter the loop ("(" operator or “loop” macro), perform a comparison, and use the WHILE operator ("\" or “while” macro) to decide when the loop terminates. As long as the result of the comparison is TRUE, the commands between the "\" and the first ")" (or “endloop” macro) are executed. When the ")" is reached, control is immediately transferred back to the command following the "(". Then the sequence repeats. When the result of the comparison is FALSE (0), the while operator will transfer control to the next command following the matching ")". For example:

```
10 1 ( SCSs <= \ Sc " #" 1+ ) SD "\n"
```

displays " `1 2 3 4 5 6 7 8 9 10`" followed by a carriage-return. The "SCSs" sequence copies the top two items on the stack and reverses their order before they are consumed by the comparison operator "<=". Then the WHILE operator "\" tests the comparison result. As long as it is TRUE, a copy of the top stack item is printed, and the top stack item is incremented by one. At the end of the loop, the two stack items are removed, and a carriage-return is displayed.

When the macros are used, readability improves greatly:

```
10 1 loop 2dup swap <= while dup " #" 1+ endloop 2drop "\n"
```

Nested loops are supported to a default depth of 1,024.

4.12. Functions

```
F m@ mF ` f mf { }
```

WARNING:
Due to oversimplified implementation of functions, if/else/end-if and loop/while/end-loop operators, the following function names should be avoided:
` [;] { } " \ ( )
Programs using these function names could fail in function definitions, loops or ifs.

There are three kinds of functions, global, local and memory-resident. Global functions are defined using FUNCTION ("{") and FUNCTION-END ("}") respectively. This type of function can be called by any other global or local function you create, using the CALL operator ("F"). As you may have guessed, local functions are defined using LOCAL-FUNCTION ("`") both to start and end the definition. This second type of function can only be called by the parent function and other siblings (children of the same parent.) This allows you to break up complex functions into one global function that contains one or more local functions which, in turn, each perform one part of the complex action. You do not have to use local functions, but they let you simultaneously partition complex actions into simpler sub-actions, and conceal that complexity from the user (and prevent them from using the local functions.) Additionally, it keeps from cluttering up the global function table with definitions that are not intended to be used directly. Local functions are called by the LOCAL-CALL ("f") command. For example:

```
`t 2dup < if swap endif drop `
```

defines the local function "t", which tests the top two items on the stack and return the larger of the two. This command uses the new function directly from the "interp>" prompt:

```
23 24 ft .
```

and `24` is displayed. This is awkward and cryptic to use. Here is a global function that is more user-friendly:

```
{M ft " # is the largest.\n"}
```

This new function calls the first one, and is used like this:

```
62 34 FM
```

and will display:

```
62 is the largest.
```

You are not allowed to define one global function inside of another global function. You are also not allowed to define one local function inside of another local function. Functions, local or global, may call other functions up to a default depth of 1,024 function calls before returning. *Depending on your system, it may be possible to run out of memory before interp's maximum function depth is reached.*

Both global and local function calls have memory overhead associated with them. When global functions are called, a local function table and a local variable pool have to be allocated. Before the global function can return, the local functions, local function table, and local variable pool are freed. When local functions are called, a local variable pool is allocated, and before returning it must be released.

Because memory management overhead affects execution, and because your target c library may or may not actually reclaim memory that is freed, there is an alternative way to call a global function (“mF”) or a local function (“mf”) that does not create a separate execution context (dynamically allocating and freeing private functions and data.)

When functions are called this way (as a macro), any local variables or functions that are referenced belong to the caller, and in that sense, it is as if the function was expanded in-line at the point of the call. See examples 17 and 18 in Chapter 7.

Memory-resident functions are just the body or content of a global function (everything between the global function name and the enclosing '}'.) They can only be called as macros with the MEMORY-MACRO command (“m@”.) Called in this way, memory-resident functions share the caller`s local variables or functions (current execution context.)

4.13. Storage management operators

Xd `Xg dG qg ql dL dg` dl

Lets say that the "M" function is not needed. Use the X-OUT-DEFINITION command ("XdM") to delete, or cross-out, global function "M". The third letter of this command is the name of the user-defined global function that you want to get rid of. If you want to verify the definition of some global function “H”, use the GLOBAL-DEFINITION command ("dGH") to display something like this:

```
{H "\n whatever!\n" }
```

If you want to erase all user-defined global functions prior to reading in new definitions, use the X-OUT-FUNCTIONS command ("Xg").

The QUERY-GLOBAL (“qg”) and QUERY-LOCAL (“ql”) commands allow you to find out if a specific global or local function exists.

So far not much has been said about user-defined, local functions. They can be used as "scratch" functions directly from the prompt, if, for instance, you need to create a temporary function, that you are using just to reduce keystrokes. Since you don't intend to keep it, and since you already have global functions defined, you might want to define it as a local function. This way, its name doesn't conflict with your global function names, and it cannot be called from anywhere except the prompt. Two commands are available to help you keep up with local functions you use in this way. "dl" and "dL" list all local function names, and the definition of one local function respectively. They work in much the same way as "dg" and "dG".

4.14. Platform-independent values

Il `Ir K v` version

The VERSION command ("v" or “version” macro) displays the version number of the interpreter as a simple digit string of the form "MM.mm.pp", where "MM" is the major release number, "mm" is the minor revision number, and "pp" is the patch number.

As you write more and more programs, they tend to become increasingly sophisticated. You may find that you need access to interpreter-dependent information. The LOOP-DEPTH command ("Il") returns the nesting depth of the loop stack. The CALL-DEPTH command ("Ir") returns the nesting depth of the call return stack.

The SYSTEM-CONSTANT command ("K") allows you to retrieve other interpreter-dependent information, such as the base addresses of the three stacks used by the interpreter (data, function return, and loop stacks), the sizes of the various stack frames, and many other items. These values are described in detail in chapter 5.

4.15. Miscellaneous

```
Te Ts w
```

The TIME-START (“Ts”) and TIME-END (“Te”) commands provide a way to measure the elapsed time in sections of your scripts – the region bracketed by “Ts” and “Te”.

The WAIT-MSEC (“w”) command allows you to temporarily pause a specific number of milliseconds.

4.16. Termination

```
$ quit
```

The QUIT command ("$" or “quit” macro) terminates execution. All functions, variables, and allocated memory are released. Generally, avoid using this command in files that are executed by either input redirection or by the CLI “-f” option flag. Failure to head this advice may cause premature loss of hair when things inexplicably stop working without any warnings or messages. Another common symptom of this problem is interp's sudden termination with a “segmentation fault” error.

5. Alphabetical Listing of Interpreter Commands

The stack usage diagram for each operator shows the stack input on the left side of the '--' and the stack output on the right. For each side, the right-most item is the top of the stack. Taking the '+' operator as an example, n1 and n2 are the inputs, with n2 on top of the stack, and sum is the output. In other words, n1 and n2 are consumed, and replaced with sum.

OPERATOR	STACK USAGE	DESCRIPTION
0x09	(--)	NO-OP: The **tab** character performs no operation but is useful for improving readability.
0x0a	(--)	NO-OP: The **newline** character performs no operation but is useful for improving readability.
0x0c	(--)	NO-OP: The **form feed** character performs no operation but is useful for improving readability.
0x0d	(--)	NO-OP: The **carriage return** character performs no operation but is useful for improving readability.
0x20	(--)	NO-OP: The **space** character performs no operation but is useful for improving readability.
!	(n addr --)	STORE: store a 32-bit value (n) at address (addr)
"	(--)	QUOTE: display the string that follows, up to the closing quote. Special, meta-characters, interpreted during printing: # - print the top-of-stack value in current output base Optional modifiers (only one, follows '#'): b - print value as a notated (0x) hexadecimal byte (4 chars) B - print value as a hex byte (2 characters) c - print value as one ASCII character C - print value as one ASCII character, prints '.' if the character is non-printable d - print value as free-form decimal Dn - display decimal value, right-justified in an n-digit field. The range of n is 0-9 with 0 representing a 10-digit field and all other values representing their own field width. h - print value as a notated (0x) hex halfword (6 chars) H - print value as a hex halfword (4 characters) s - use value as the address of a zero-terminated string of ASCII characters, and print the entire string T - treat the value as elapsed time in microseconds and print it as hh:mm:ss.uuuuuu, where "hh" is hours, "mm" minutes, "ss" seconds, and "uuuuuu" microseconds w - print value as a notated (0x) hex word (10 chars) W - print value as a hex word (8 characters) NOTE: these modifiers (bBhHwW) specify minimum field width. If the value is larger, it still prints properly. !A – store the value on the top of the stack at PRINT_ITERATOR_ADDRESS (K38), reset PRINT_ITERATOR_MODE (K39) to auto-increment (1), and set PRINT_ITERATOR_FIRST_ONE (K40) to TRUE/~0 (only used by auto-decrement) !i - enable auto-increment of the print iterator address after printing the contents. !d - enable auto-decrement of the print iterator address after

OPERATOR	STACK USAGE	DESCRIPTION
" (continued)	(--)	printing the contents. @ - print the value pointed to by the print iterator address (K38) in the current output base. An optional modifier follows the '@'. If no modifier is present, the value is read from memory as a 32-bit word. If a modifier is present, the value is read from memory as indicated below (HALFWORD is 2-bytes, and WORD is 4-bytes): a - (WORD) print the print iterator address as a notated (0x) hexadecimal word (10 characters). Print iterator address is not changed by this modifier. A - (WORD) print the print iterator address as a hexadecimal word (8 characters). Print iterator address is not changed by this modifier. b - (BYTE) print value as a notated (0x) hex byte (4 chars) B - (BYTE) print value as a hex byte (2 characters) c - (BYTE) print value as one ASCII character C - (BYTE) print value as one ASCII character, prints '.' if the character is non-printable d - (WORD) print value as free-form decimal Dn - (BYTE/HALFWORD/WORD) display decimal value right-justified in an n-digit field. The range of n is 0-9 with 0 representing a 10-digit field and all other values representing their equivalent field width. The size of memory read is determined by the specified field width: BYTE - D1, D2, and D3 HALFWORD - D4 and D5 WORD - D6, D7, D8, D9, and D0 h - (HALFWORD) print value as a notated (0x) hex halfword (6 chars) H - (HALFWORD) print value as a hex halfword (4 chars) s - (WORD) use value as the address of a null-terminated string of ASCII characters, and print the entire string or "NULL" if the value (treated as pointer) is NULL. S - (string length+1) use value as a zero-terminated string of ASCII characters, and print the entire string T - (WORD) treat the value as elapsed time in microseconds and print it as hh:mm:ss.uuuuuu, where "hh" is hours, "mm" minutes, "ss" seconds, and "uuuuuu" microseconds w - (WORD) print value as a notated (0x) hexadecimal word (10 chars) W - (WORD) print value as a hex word (8 characters) NOTE: these modifiers (bBhHwW) specify minimum field width. If the value is larger, it still prints properly. NOTE: Each modifier (except aA) increments the print iterator address after reading the value. NOTE: the 'S' modifier is illegal when auto-decrement mode is enabled ("!d") because of the uncertainty associated with searching backwards for the beginning of a variable-length string. \ - display a special character, from this list: n - newline, 0x0A r - carriage return, 0x0D t - tab, 0x09 other - literal character

OPERATOR	STACK USAGE	DESCRIPTION
$	(--)	QUIT: release functions, variables and memory, and exit. Not valid inside a function.
%	(n1 n2 -- remainder)	REM: compute the remainder of n1 / n2
&	(n1 n2 -- result)	AND: and two integers: n1 & n2 = result
'	(-- n)	TICK: push the ASCII code of the following character on the stack. (e.g. 'A places 0x41 on the stack.)
(	(--)	LOOP: marks the beginning of a loop, see \ and)
)	(--)	LOOP-END: marks the end of a loop, see (and \
*	(n1 n2 -- product)	TIMES: multiply two integers: n1 * n2 = product
+	(n1 n2 -- sum)	PLUS: add two integers: n1 + n2 = sum
,	(n --)	PRINT-CHAR: pop and output an ASCII character from the stack.
-	(n1 n2 -- difference)	MINUS: subtract two integers: n1 - n2 = difference
.	(n --)	DOT: pop and print the top stack value in the current output number base, followed by a carriage-return
/	(n1 n2 -- quotient)	DIVIDE: divide two integers: n1 / n2 = quotient
2drop	(m n –)	Macro expands to “SD”
2dup	(m n – m n m n)	Macro expands to “SC”
:	(-- n)	READ-CHAR: read one character and pushes it's ASCII code on the stack.
;	(--)	ELSE: the beginning of the optional else-clause, see [
<	(n1 n2 -- flag)	LESS-THAN: flag is TRUE if n1 less-than n2
<<	(n u -- result)	SHIFT-LEFT: n is left-shifted u bits to obtain result
<=	(n1 n2 -- flag)	LESS-THAN-OR-EQUAL-TO: flag is TRUE if n1 less-than-or-equal-to n2
<>	(n1 n2 -- flag)	NOT-EQUAL: flag is TRUE if n1 not-equal-to n2
=	(n1 n2 -- flag)	EQUAL: flag is TRUE if n1 equal-to n2
>	(n1 n2 -- flag)	GREATER-THAN: flag is TRUE if n1 greater-than n2
>=	(n1 n2 -- flag)	GREATER-THAN-OR-EQUAL-TO: flag is TRUE if n1 greater-than-or-equal-to n2
>>	(n u -- result)	SHIFT-RIGHT: n is right-shifted u bits to obtain result
?	(-- n)	READ-NUM: read an integer number and push it on the stack. If the digit string is illegal, zero is pushed on the stack.
?s	(max addr – n)	READ-STRING: read a string of not more than max characters, including the end-of-string character, store it at addr, and return the number of characters actually read. If max <= 0, reading stops when a carriage return or newline is read. NOTE: The line-end character is not returned with the string, but a zero is substituted (for it) and must be accounted for when you create your receive buffer.
@	(addr -- n)	FETCH: read a 32-bit value from addr and push it on the stack.
F	(--)	CALL: calls a user-defined, global function whose name is the next character following the "F". Global functions have their own context - private local variables and private local functions.
Il	(-- n)	LOOP-DEPTH: returns the loop nesting depth (loop stack item count)
Ir	(-- n)	CALL-DEPTH: returns the function nesting depth (return stack item count)
K	(n -- value)	SYSTEM-CONSTANT: puts the value of the nth system constant on the stack: 0 - address of the data stack list head 1 - size, in bytes, of items on the data stack 2 - address of the loop control stack list head 3 - size, in bytes, of items on the loop control stack 4 - address of the function return stack list head 5 - size, in bytes, of items on the function return stack

OPERATOR	STACK USAGE	DESCRIPTION
K (continued)	(n -- value)	6 - base address of input buffer used by outer interpreter 7 - size, in bytes, of one token 8 - base address of the command vector table 9 - size, in bytes, of entries in the command vector table 10 - address of non-operation function 11 - address of unknown-operation function 12 - address of 32-bit value representing output number base 13 - address of the base address of the global data area 14 - size, in bytes, of global variable data area 15 - address of base address of the local variable data area 16 - size, in bytes, of the local variable area 17 - address of base address of system constants data area 18 - size, in bytes, of the system constants data area 19 - address of the base address of the global function table 20 - address of the base address of the local function table 21 – interp's vector table initialization flag (always 1.) 22 – address of the base address of ibuf[] 23 – address of the base address of ebuf[] 24 – argc, number of command line arguments including the command. For CLI builds, the value will always be greater than zero. 25 – argv[], base address of the argument list table. Each 4-byte entry is a pointer to a zero-terminated string, or NULL (last entry.) 26 – pointer to optind, index for the first non-flag argument (no leading "-") on the command line. If there are no arguments of this type, this value will be greater than or equal to argc (24K). For non-CLI builds, this value is NULL. 27 – scratchpad (seconds) for computing elapsed time. The Unix/Linux version (default version) uses this to hold system time in seconds representing the moment when TIME-START ("Ts") was executed. Depending on hardware resources, non-OS ports may/not use this. 28 – scratchpad (microseconds) for computing elapsed time. The Unix/Linux version (default version) uses this to hold system time in seconds representing the moment when TIME-START ("Ts") was executed. Depending on hardware resources, non-OS ports may or may not use this. 29 - TYPE_OF_BUILD (0=OS, 1=NOFLAGS, 2=NOCLI, 3=SIN) 30 - DATA_STACK_DEPTH (default=1024) 31 - LCSP_STACK_DEPTH (default=1024) 32 - FRSP_STACK_DEPTH (default=1024) 33 - VARIABLE_POOL_DEPTH (default=1024) 34 - NO_PRIVATE_DATA (for functions; default=FALSE) 35 - INBUF_SIZE (default=4097) 36 - BIGBUF_SIZE (default=65536) 37 – ENFORCE_ADDRESS_ALIGNMENT (default=~0/TRUE) 38 - PRINT_ITERATOR_ADDRESS (default=NULL; not set) This value shows you where in memory the next QUOTE (") or M-QUOTE (M") operator will print from with the "@" meta-character. 39 - PRINT_ITERATOR_MODE (default=1) This value shows whether the print iterator address auto-increments or auto-decrements to advance to the next location. When one (1), the print iterator address will auto-increment after printing the contents. When zero (0), the print iterator address will auto-decrement after printing the contents. All other values are reserved for future use. 40 - PRINT_ITERATOR_FIRST_ONE (default=TRUE/~0) Used in auto-decrement mode: FALSE/0=decrement

OPERATOR	STACK USAGE	DESCRIPTION
K (continued)	(n -- value)	before printing, and TRUE/~0=print without the pre-decrement. "!A" and "!d" set this to TRUE/~0. Then after the execution of the first following "@" meta-character, it is set to FALSE/0 to enable the pre-decrement of all following “@” meta-characters up to the next “!A” or “!i”. NOTE: all stack list heads consist of 3 4-byte items: address of top-of-stack, address of start of stack, address of end of stack.
L	(index -- addr)	L-VAR: convert index of 32-bit local variable to address
M”	(addr --)	M-QUOTE: print string that follows to addr in memory, up to the closing quote. Special, meta-characters, interpreted during printing: # - print the top-of-stack value in current output base Optional modifiers (only one, follows '#'): b - print value as a notated (0x) hexadecimal byte (4 chars) B - print value as a hex byte (2 characters) c - print value as one ASCII character C - print value as one ASCII character, prints '.' if the character is non-printable d - print value as free-form decimal Dn - display decimal value, right-justified in an n-digit field. The range of n is 0-9 with 0 representing a 10-digit field and all other values representing their own field width. h - print value as a notated (0x) hex halfword (6 chars) H - print value as a hex halfword (4 characters) s - use value as the address of a zero-terminated string of ASCII characters, and print the entire string T - treat the value as elapsed time in microseconds and print it as hh:mm:ss.uuuuuu, where "hh" is hours, "mm" minutes, "ss" seconds, and "uuuuuu" microseconds w - print value as a notated (0x) hex word (10 chars) W - print value as a hex word (8 characters) NOTE: these modifiers (bBhHwW) specify minimum field width. If the value is larger, it still prints properly. !A – store the value on the top of the stack at PRINT_ITERATOR_ADDRESS (K38), reset PRINT_ITERATOR_MODE (K39) to auto-increment (1), and set PRINT_ITERATOR_FIRST_ONE (K40) to TRUE/~0 (only used by auto-decrement) !i - enable auto-increment of the print iterator address after printing the contents. !d - enable auto-decrement of the print iterator address after printing the contents. @ - print the value pointed to by the print iterator address (K38) in the current output base. An optional modifier follows the '@'. If no modifier is present, the value is read from memory as a 32-bit word. If a modifier is present, the value is read from memory as indicated below (HALFWORD is 2-bytes, and WORD is 4-bytes): a - (WORD) print the print iterator address as a notated (0x) hexadecimal word (10 characters). Print iterator address is not changed by this modifier. A - (WORD) print the print iterator address as a hexadecimal word (8 characters). Print iterator address

OPERATOR	STACK USAGE	DESCRIPTION
M" (continued)	(addr --)	is not changed by this modifier. b - (BYTE) print value as a notated (0x) hex byte (4 chars) B - (BYTE) print value as a hex byte (2 characters) c - (BYTE) print value as one ASCII character C - (BYTE) print value as one ASCII character, prints '.' if the character is non-printable d - (WORD) print value as free-form decimal Dn - (BYTE/HALFWORD/WORD) display decimal value right-justified in an n-digit field. The range of n is 0-9 with 0 representing a 10-digit field and all other values representing their equivalent field width. The size of memory read is determined by the specified field width: BYTE - D1, D2, and D3 HALFWORD - D4 and D5 WORD - D6, D7, D8, D9, and D0 h - (HALFWORD) print value as a notated (0x) hex halfword (6 chars) H - (HALFWORD) print value as a hex halfword (4 chars) s - (WORD) use value as the address of a null-terminated string of ASCII characters, and print the entire string or "NULL" if the value (treated as pointer) is NULL. S - (string length+1) use value as a zero-terminated string of ASCII characters, and print the entire string T - (WORD) treat the value as elapsed time in microseconds and print it as hh:mm:ss.uuuuuu, where "hh" is hours, "mm" minutes, "ss" seconds, and "uuuuuu" microseconds w - (WORD) print value as a notated (0x) hexadecimal word (10 chars) W - (WORD) print value as a hex word (8 characters) NOTE: these modifiers (bBhHwW) specify minimum field width. If the value is larger, it still prints properly. NOTE: Each modifier (except aA) increments the print iterator address after reading the value. NOTE: the 'S' modifier is illegal when auto-decrement mode is enabled ("!d") because of the uncertainty associated with searching backwards for the beginning of a variable-length string. \ - display a special character, from this list: n - newline, 0x0A r - carriage return, 0x0D t - tab, 0x09 other - literal character
M?	(addr – n)	M-READ-NUM: read a digit string from addr in memory and push it's integer value on the stack. If the string isn't numeric, zero is pushed on the stack.
MC	(addr1 addr2 n --)	MEM-COPY: copy n 32-bit words from addr1 to addr2 in mem. The areas may overlap.
MR	(portaddr n buffaddr --)	MEM-READ: read the 32-bit I/O port portaddr and write each result in sequence into the n-word area that starts at buffaddr.
MW	(portaddr n buffaddr --)	MEM-WRITE: read n 32-bit words from the area starting at buffaddr and write them to the 32-bit I/O port, portaddr.
Ma	(words -- addr)	MALLOC: allocate words 32-bit words of memory, initialize it to zero, and return the starting address of it.
Mc	(addr1 addr2 size max-err -- total-errors)	MEM-COMPARE: compare two regions of memory. The first region starts at addr1. The second region at addr2. The number of bytes to compare is size. If max-err is greater than zero, it represents the maximum number of

OPERATOR	STACK USAGE	DESCRIPTION
Mc (continued)	(addr1 addr2 size max-err – total-errors)	mismatches that will be displayed on the screen. Total-errors is the number of bytes that don't match. If max-err was greater than zero, then the count of mismatches will be displayed on the screen. In other words, if max-err is zero, MEM-COMPARE doesn't display anything on the screen.
Mf	(addr --)	M-FREE: free the memory located at addr.
Ml	(addr – length)	M-LENGTH: returns the length of the zero-terminated string that starts at addr. The returned length does not include the terminator.
Mm	(count hardware-address -- mapped-address)	MAP-MEMORY: Map count bytes of hardware memory, starting with hardware-address, to system memory. Mapped-address is the system memory address (>0) that is equivalent to hardware-address in the hardware address space. Non-OS builds return hardware-address.
Ms	(addr1 size1 addr2 size2 -- result)	M-SEARCH: search the region of memory starting at addr1, with a size of size1 bytes, for a string of size2 bytes that start at addr2. If the byte string is found, the result is the memory address where the matching string of bytes begin. If the byte string is not found, the result is -1.
N	(n -- -n)	NEGATE: change the sign of the number
Pd	(--)	PRINT-DECIMAL: change the output number base to decimal
Po	(--)	PRINT-OCTAL: change the output number base to octal (base 8)
Px	(--)	PRINT-HEX: change output number base to hexadecimal (base 16)
SC	(m n – m n m n)	STACK-2COPY: duplicate the top two numbers on the stack
SD	(m n --)	STACK-2DROP: drop the top two numbers from the stack
Sc	(n -- n n)	STACK-COPY: duplicate the top number on the stack
Sd	(n --)	STACK-DROP: drop the top number from the stack
Sn	(-- n)	STACK-NUM: returns the number of items on the stack
So	(n1 n2 -- n1 n2 n1)	STACK-OVER: duplicate the second number on the stack
Sp	(... n -- ... n)	STACK-PICK: copy the nth stack entry to the top of the stack.
Sr	(n1 n2 n3 -- n2 n3 n1)	STACK-ROT: move the third stack entry to the top of the stack.
Ss	(n1 n2 -- n2 n1)	STACK-SWAP: swap the top two numbers on the stack
Te	(-- time)	TIME-END: stop measuring elapsed time and place the resulting timing value, in microseconds, on the top of the stack. If the elapsed time is too large (>2146 seconds or whatever the the hardware allows) then the value returned is -1. If the value returned is zero (0) then this command is not implemented for your version of interp.
Ts	(--)	TIME-START: begin measuring elapsed time (microseconds)
V	(index -- addr)	VAR: convert index of 32-bit global variable to its address
Xd	(--)	X-OUT-DEFINITION: remove the definition of the user-defined, global function whose name is the next character following the 'd'. Not valid inside a function.
Xg	(--)	X-OUT-FUNCTIONS: remove the definitions of all of the user-defined, global functions. Not valid inside a function.
[	(flag --)	IF: if flag is TRUE execute the commands following the [up to the matching ; or]. If flag is FALSE, skip to the matching ; or] and begin executing at that point.
\	(flag --)	WHILE: if flag is FALSE, execution resumes after the matching), otherwise, execution continues up to the matching)
]	(--)	END-IF: marks the end of the conditional statement, see [and ;
^	(n1 n2 -- result)	EXCLUSIVE-OR: XOR two integers: n1 ^ n2 = result
`	(--)	LOCAL-FUNCTION: marks the beginning of the definition of a local user function whose name is the next character after the "`". A second "`" marks the definition's end.
a	(n -- |n|)	ABSOLUTE-VALUE: return the absolute value of the number on top of the

OPERATOR	STACK USAGE	DESCRIPTION
a (continued)	(n -- |n|)	stack – if n is negative, it is converted to its positive equivalent, otherwise it is unchanged.
b!	(n addr --)	BYTE-STORE: store an 8-bit value
b@	(addr -- n)	BYTE-FETCH: fetch an 8-bit signed value, sign-extension occurs
bC	(addr1 addr2 n --)	BYTE-COPY: copy n bytes from addr1 to addr2 in memory. The areas may overlap.
bR	(portaddr n buffaddr --)	BYTE-READ: read the 8-bit I/O port portaddr and write each result in sequence into the n-byte area that starts at buffaddr.
bW	(portaddr n buffaddr --)	BYTE-WRITE: read n 8-bit bytes from the area starting at buffaddr and write them to the 8-bit I/O port, portaddr.
bL	(index -- addr)	L-BYTE-VAR: convert index of 8-bit local variable to its address
bV	(index -- addr)	BYTE-VAR: convert index of 8-bit global variable to its address
dG	(--)	GLOBAL-DEFINITION: list definition of the user-defined, global function whose name is the next character following the 'G'. Not valid inside a function..
dL	(-- n)	LOCAL-DEFINITION: list definition of the user-defined, local function whose name is the next character following the 'L'. Not valid inside a function..
depth	(-- n)	Macro expands to “Sn”
dg	(--)	LIST-GLOBALS: list the names of all user-defined, global functions. Not valid inside a function.
dl	(--)	LIST-LOCALS: list the names of all user-defined, local functions. Not valid inside a function.
drop	(n --)	Macro expands to “Sd”
dup	(n -- n n)	Macro expands to “Sc”
else	(--)	Macro expands to “;”
endif	(--)	Macro expands to “]”
endloop	(--)	Macro expands to “)”
f	(--)	LOCAL-CALL: calls a user-defined, local function whose name is the character following the "f". Local functions share their parent's private local functions, but have their own, private local variables
free	(addr --)	Macro expands to “Mf”
h!	(n addr --)	HALFWORD-STORE: store a 16-bit value
H@	(addr -- n)	HALFWORD-FETCH: read a 16-bit signed value
hC	(addr1 addr2 n --)	HALFWORD-COPY: copy n halfwords from addr1 to addr2 in memory. The areas may overlap.
hR	(portaddr n buffaddr --)	HALFWORD-READ: read the 16-bit I/O port portaddr and write each result in sequence into the n-hword area that starts at buffaddr.
hW	(portaddr n buffaddr --)	HALFWORD-WRITE: read n 16-bit halfwords from the area starting at buffaddr and write them to the 16-bit I/O port, portaddr.
hL	(index -- addr)	L-HALFWORD-VAR: convert index of 16-bit local variable to its address
hV	(index -- addr)	HALFWORD-VAR: convert index of 16-bit global variable to its address
if	(flag --)	Macro expands to “[“
l	(-- addr)	L-BASE: returns base address of caller's local variables. Only valid inside a function.
loop	(--)	Macro expands to “(“
m@	(addr --)	MEMORY-MACRO: calls a user-defined function starting at addr in memory. When called this way, the function shares the caller's local variables and local functions (the current execution context.) NOTE: Error messages will refer to a function named zero (0x00), because it has no “name.”

OPERATOR	STACK USAGE	DESCRIPTION
malloc	(words -- addr)	Macro expands to “Ma”
mF	(--)	GLOBAL-MACRO: calls a user-defined, global function whose name is the next character after the "F". When called this way, global functions share the caller's local variables and local functions (no dynamic memory allocation.)
mf	(--)	LOCAL-MACRO: calls a user-defined, local function whose name is the next character after the "f". When called this way, local functions share the caller's local variables and local functions (no dynamic memory allocation.)
over	(n1 n2 -- n1 n2 n1)	Macro expands to “So”
p	(-- n)	POLL: returns non-zero (true) when input is waiting to be read, and 0 (false) otherwise.
pick	(... n -- ... n)	Macro expands to “Sp”
qg	(-- flag)	QUERY-GLOBAL: return true (~0) if the global function whose name is the next character following the 'g' is defined, or false (0) if it is not defined.
ql	(-- flag)	QUERY-LOCAL: return true (~0) if the local function whose name is the next character following the 'l' is defined, or false (0) if it is not defined.
quit	(--)	Macro expands to “$”
rot	(n1 n2 n3 -- n2 n3 n1)	Macro expands to “Sr”
swap	(n1 n2 -- n2 n1)	Macro expands to “Ss”
v	(--)	VERSION: display version information. Not valid inside a function.
version	(--)	Macro expands to “v”
w	(n --)	WAIT-MSEC: delay execution for AT LEAST n milliseconds
while	(flag --)	Macro expands to “\”
{	(--)	FUNCTION: marks the beginning of the definition of a global user function whose name is the next character after the "{"
|	(n1 n2 -- result)	OR: OR two integers: n1 | n2 = result
}	(--)	FUNCTION-END: marks the end of the definition of a global user function
~	(n -- result)	INVERT: invert the bits of an integer (logical NOT)

6. Error Messages

NOTE: The following table uses '*c*' to represent both printable and non-printable ASCII characters, although in actual error messages, the non-printable ASCII characters are rendered in hex notation (0x*nn*.) Typically, this character is either the operator or sub-operator being executed at the time the error occurred.

If verbose-style error messages are enabled, greater error context is provided by displaying a fragment of the code string containing the error (bad opcode.) The line below that is an "arrow" pointing at the token where the error occurred. Other information is also displayed. See example 20 in Chapter 7.

Error Code	Error Message Comments or Description
1	***c*: stack empty.** This error can be reported by any operator that consumes stack items. In general terms it indicates that the command '*c'* failed because it did not find enough data on the stack.
2	***c*: system constant is out of range.** Only SYSTEM-CONSTANT issues this error. The top stack item represents the particular system constant desired, and that value is too large, or negative. In other words, you've requested a non-existent system constant.
3	**Literal string too long.** The QUOTE (") and M-QUOTE (M") operators will not process strings longer than 80 characters
4	**I don't know what '*c*' means.** This token is not recognized as a valid command.
5	***o*: don't know what sub-operator '*c*' means.** Only fourteen tokens have sub operators: < > ? I M P S T X b d h m q. Check your work to see which sub-operator you forgot to type. '*o*' is the operator (character preceding the unknown sub operator) that issued this error.
6	**The '*c*' command attempted to divide by zero.** Only DIVIDE ('/') and REM ('%') issue this error. Both operators take two integers from the stack. The top stack item is the divisor and cannot be zero.
7	***c*: the memory address is misaligned.** Eight operators optionally test their address arguments for proper alignment: addresses of halfwords (16-bit) must be divisible by two and addresses of words (32-bit) must be divisible by four. Check the address(es) passed to one of the following operators, as indicated in the error message: MEM_WRITE ("MW"), MEM_READ ("MR"), HALFWORD_WRITE ("hW"), HALFWORD_READ ("hR"), HALFWORD_FETCH ("h@"), HALFWORD_STORE ("h!"), FETCH ("@"), and STORE ("!"). Check System Constant 37. Also the print iterator feature of QUOTE (") and M-QUOTE (M") optionally test the print iterator address for proper alignment.
8	**The global function '*c*' is not defined.** The global function name referenced by CALL ("F"), GLOBAL-MACRO ("mF"), or GLOBAL-DEFINITION ("dG") does not exist. Either you have a typo, or the function has not yet been created.

Error Code	Error Message Comments or Description
1	***c*: stack empty.** **The local function '*c*' is not defined.**
9	The local function name referenced by LOCAL-CALL (“f”), LOCAL-MACRO (“mf”), or LOCAL-DEFINITION (“dL”) does not exist. Either you have a typo, or the function has not yet been created.
10	***c*: Sub-operator is missing.** The last token, just before the end-of-string (zero), is one of the fourteen tokens that have sub-operators. Either you typed the wrong token, or forgot to type the sub-operator.
11	***c*: stack full.** You have invoked an operator or sub operator that pushes one or more items onto the stack, but the stack is already full. You have a programming error to find and correct.
12	***c*: function return stack overflow.** You have too many nested function calls and have filled up the function return stack. This is a programming error that you'll have to fix.
13	***c*: function return stack underflow.** Either CALL (“F”) or LOCAL-CALL (“f”) is trying to return from your function but the function return stack is empty. Since this is a very abnormal condition, you may have been manipulating the function return stack, and created a programming problem that you have to fix.
14	***c*: loop control stack overflow.** You have too many nested LOOP (“(“) operators and have filled up the loop control stack. This is a programming problem that has to be fixed.
15	***c*: loop control stack underflow.** The LOOP-END (“)”) operator is trying to find the beginning of the current loop, to start the next iteration, but the loop control stack is empty. This is a very abnormal error, and may indicate that you have improperly manipulated the loop control stack.
16	***c*: could not allocate local variables for function call.** You have run out of memory in the computer and could not allocate enough space for the CALL (“F”) or LOCAL-CALL (“f”) operator to initiate your function call. This could be caused by deeply nested functions, a memory leak, or a non-working implementation of free() in your target's c library.
17	***c*: is only valid inside a function.** The L-BASE (“l”) operator is only valid inside of a function.
18	***c*: I can't understand this number.** Hypothetically, this error can occur, but it has never been observed in the “wild.”
19	***c*: global function name is missing.** Hypothetically, this error can occur only three ways: the FUNCTION (“{“) operator, GLOBAL-MACRO (“mF”) operator, or the CALL (“F”) operator is at the end of the string, and the next character after the “{“, “mF” or the “F” is the end-of-string (zero.)
20	***c*: local function name is missing.** Hypothetically, this error can occur only three ways: the LOCAL-FUNCTION (“`“) operator, LOCAL-MACRO (“mf”), or the LOCAL-CALL (“f”) operator is at the end of the string, and the next character after the “`“, “mf”, or the “f” is the end-of-string (zero.)

Error Code	Error Message Comments or Description
21	***c*: is not valid inside a function.** The QUIT ("$") operator cannot be used inside a function.
22	***c*: the stack entry number cannot be negative.** The STACK-PICK ("Sp") operator requires that the stack entry number be greater than or equal to zero. In other words, executing the "Sp" operator with a negative number on top of the stack causes this error.
23	***c*: the port read count cannot be negative.** The BYTE-READ ("bR"), HALFWORD-READ ("hR"), and MEM-READ ("MR") operators require a transfer count that is greater than or equal to zero.
24	***c*: the port write count cannot be negative.** The BYTE-WRITE ("bW"), HALFWORD-WRITE ("hW"), and MEM-WRITE ("MW") operators require a transfer count that is greater than or equal to zero.
25	***c*: can't open "*map-path*" for memory mapping.** OS-builds only (embedded Linux.) Typically indicates a permissions problem accessing "*map-path*". Both read and write access is required. The most common value for "*map-path*" is "/dev/mem".
26	***c*: can't map the requested hardware region.** OS-builds only (embedded Linux.) Your only indication that the hardware address is incorrect in some unspecified (undocumented) way.
27	***c*: print iterator address is not set.** The print control string associated with the QUOTE (") or M-QUOTE (M") operator is using the "@" meta-character (print iterator) before setting the base address ("!A") with a value from the data stack.
28	***c*: can't print embedded strings in auto-decrement mode.** The print control string associated with the QUOTE (") or M-QUOTE (M") operator is attempting to print an embedded string via the "@S" sequence but PRINT_ITERATOR_MODE (39K) is set for auto-decrement mode via a previously occurring "!d". This is not allowed because of the uncertainty associated with searching backwards for the beginning of a variable-length string.
9999	***c*: unrecognized error code:** *nn* Hypothetically, this error can only occur if interp's maintainers makes a mistake. As mentioned previously – find them; kill them.

7. Examples (Try them yourself – in the order presented.)

1. `2 1 - .`
   will print 1 as the result
   NOTE: spaces between operators are optional except to separate numbers.

2. `2 1-.`
   same as above, with the unnecessary spaces removed.

3. `3 2 + 7 *.`
   equivalent to: (3 + 2) * 7.

4. `4 dup * "The answer is #\n"`
   equivalent to: 4 * 4
   prints "The answer is 16" on the screen (if you haven't changed the output number base, the default is decimal.) Remember, “dup” is the macro equivalent of “Sc”.

5. `3 2 + 7 * Px.`
   same as example 3. except the number base is changed to hex
   before the result is printed on the screen.

6. `Pd 5 ScSc ** .`
   equivalent to: 5 * 5 * 5
   the output number base is changed back to decimal before printing. If you convert this example to use the “dup” macro in place of “ScSc” you have to write it as “dup dup” with a space in between.

7. `0x100 .`
   prints the decimal equivalent, 256, of the hex number 100.

8. `{a SCSs " # + # = " +. }`
   defines a global function named "a" that adds two numbers and displays the equivalent algebraic expression including the result. “SCSs” can be rewritten as “2dup swap” using the default macros.

9. `3 2 Fa`
   executes the global, user-defined function "a" that was defined
   in the previous example. It will display: " 3 + 2 = 5".

10. `{a So + Ss ( SCSs < \ Sc b@ , 1+ ) SD }`
    redefines the global function "a" to print a string of ASCII characters from memory. It needs a memory address and the number of characters to display. (See example 16.)

11. `0x404 8 Fa "\n"`
    uses the redefined "a" function from the previous example to display an 8-character ASCII string that starts at memory location 0x404.

12. `{M SC < [ Ss ] Sd }`
    defines the function "M" which expects two numbers and returns the one that is larger.

13. `32 33 FM .`
    Uses the "M" function directly, and prints 33.

14. `{t FM " # is the largest.\n"}`
    defines a new function "t" that calls the function "M" and prints the largest of the two numbers on the stack.

15. `33 34 Ft`
    uses the "t" function and displays " 34 is the largest."

16. This example uses some of the outer interpreter features, including the standard predefined macros in Appendix A., to define the "a" function from example 10 in a much more readable form, that you are more likely to use if you create PC files containing your interpreter programs:

```
__ Fa ( addr count -- ) print a string of ASCII characters
__    usage example: 0x404 8 Fa "\n"
#BUFFER
{a
   over + swap              __ convert addr count to end-addr start-addr
   loop 2dup swap < while   __ while start-addr is < end-addr
     dup b@ , 1+            __ display next byte and increment start-addr
   endloop
   2drop                    __ remove start-addr end-addr from stack
}
#EXECUTE
```

17. This example expects a memory address on top of the stack and uses the new print iterator to print 256 bytes of memory in hex, as 16 rows of 8 16-bit values:

```
__ FD ( addr -- ) print 256 bytes of memory in hex,
__                as 16 rows of 8 16-bit values
__    usage example: 4K FD
#BUFFER
{D
   1~&                 __ force address to be halfword-aligned
   255 over + swap     __ convert addr to end-addr start-addr
   loop 2dup swap <= while __ while start-addr is <= end-addr
      dup              __ copy the row start-addr
      "!A @a: @h @h @h @h @h @h @h @h\n" __ print a row
      16+              __ advance to next row start
   endloop
   2drop               __ drop start-addr end-addr from stack
}
#EXECUTE
```

18. An expanded version of the previous example, with local functions called as macros.

```
__ FD ( addr -- nxtaddr ) Print 256 bytes of memory in hex, as 16 rows of 4
__                        32-bit values with ASCII equivalents. The value
__                        left on the stack, nxtaddr, is the address of the
__                        next location following the last dumped location.
__                        This helps with repeated FD commands.
__    usage example: 8K FD FD .
#BUFFER
{D

   __   ff ( addr -- ) print ASCII equiv. of the word, in left-to-right order
   `f                __ assume little-endian machine
     3+              __ start at the end of the word
     "!A!d@C@C@C@C" __ print 4 characters (decrementing)
   `

   __   fe ( addr -- ) print the ASCII equivalent of the 4 words
   `e
     "  "               __ output two blanks
     15 over + swap     __ convert addr to end-addr start-addr
     loop 2dup swap <= while   __ while start-addr <= end-addr
       dup mff          __ display the ASCII equivalent of a word
       4+               __ advance to next word
     endloop 2drop      __ remove start-addr end-addr from stack
   `

   __    fd ( addr -- ) dump the 4 words in hex followed by the ASCII
   `d
     dup                __ preserve a copy of addr for fe to use later on
     "!A @a:"           __ print the row address
     " @W @W @W @W"     __ display 4 words
     mfe                __ display ASCII equiv.
     "\n"               __ print newline
   `

  3~&                   __ force addr to be word aligned
  255 over + swap       __ convert addr to end-addr start-addr
  loop 2dup swap <= while   __ while start-addr <= end-addr
    dup mfd 16+         __ display a row and advance start-addr to next row
  endloop
  drop 1+               __ remove start-addr, adjust end-addr to next byte
}
#EXECUTE
```

19. This example illustrates one way to use SYSTEM-CONSTANTS 24-26 to access the command line information passed by the CLI build (default & embedded Linux builds.)

```
# FA ( -- ) list CLI argument information

#BUFFER
{A
  24K "\n#d arguments were on the command line:\n"
  "ndx  argument\n"
  "---  --------\n"
  24K 0 loop          __ look at each of the arguments
  2dup swap < while
    dup 26K@ = if
      "=>"            __ indicates the first script argument
    else
      "  "
    endif dup "#d"            __ print the argument index
    dup 4* 25K+@ " \"#s\"\n" __ then print the argument string
    1+                        __ advance the argument index
  endloop 2drop       __ end the loop and drop the indices

  __ if optind (26K@) is out of range (>= 24K) tell us all about it
  26K@ 24K >= if
    26K@ "=>#d"

    __ the string pointer is *supposed* to be NULL in this case ...
    26K@ 4* 25K+@ dup 0= if
      drop " NULL\n"        __ normal (NULL) case
    else
      " \"#s\"\n"           __ abnormal (non-NULL) case
    endif
  endif

  26K@ "optind=#d\n"

  "\nNOTE: \"=>\" indicates the first script argument.\n\n"

}
#EXECUTE
```

20. This example illustrates verbose-style error messages:

```
interp> #verbose
interp> {a (.) "that's it!\n"} Fa
.: stack empty.
in global function "a":
 (.) "that's it!\n"
  ^
  caller's local function pointer = 0x8068450
  caller's local variables pointer = 0x806e8a8
  caller's opcode pointer = 0x8056d98 (a)
  function call depth = 1
  loop depth = 1
  data stack depth = 0
  bad opcode pointer = 0x8070dea
  error code = 1
```

Appendix A.

A.1 Predefined macros (in alphabetical order)

Name	Definition
2drop	SD
2dup	SC
depth	Sn
drop	Sd
dup	Sc
else	;
endif	]
endloop	)
free	Mf
if	[
loop	(
malloc	Ma
over	So
pick	Sp
quit	$
rot	Sr
swap	Ss
version	v
while	\

Appendix B.

B.1. Macro Processor Error Messages

Error Num	Macro Processor Error Messages Comments or Description
1	**putbak: too many characters pushed back.** In general terms, this error message is telling you that you have created a recursive macro definition (e.g. #define thing1=thing1) Don't do that. It is also remotely possible that your macro definition is simply too long.
2	**gettok: token too long.** The alphanumeric character string representing the name of your macro definition is simply too long. Shorten it.
3	**getdef: missing space.** This indicates a syntax error in your define statement: there must be one space between “#define” (or “#assign”) and the name of your macro.
4	**getdef: non-alphanumeric name.** This indicates a syntax error in your define (or assign) statement: there must be one space, no other characters allowed, between “#define” (or “#assign”) and the name of your macro. The name itself must be letters and digits only. No other characters allowed.
5	**getdef: missing equal sign in define/assign.** This indicates a syntax error in your define (or assign) statement: there must be an “=” between the macro name and its definition. No other characters are permitted.
6	**getdef: definition too long.** In practical terms, if you see this message, there has been a change to the macro processor code that makes it seem that the definition of your macro is too long. Find the programmer who did this and insist on an immediate bug fix.
7	***name*: too many definitions in insert_macro.** The table that holds the macro name/definition pairs is a fixed size, determined at compile-time. You have created so many macros, you need that table enlarged. Find the programmer and ask for help. If you decide to put your macro definitions on a diet instead, *name* refers to the macro that couldn't be added to the table.
8	**getundef: missing space.** There must be one space between #undefine and the name of the macro to delete.
9	**getundef: non-alphanumeric name.** The name of the macro to delete is probably missing.
10	**getundef: spurious characters(s) after name.** The only character allowed after the name of the macro to delete, is a carriage return.
11	**assign_macro: missing or invalid assignment symbol.** The only legal text after the '=' is either “%n” or “%s” (without the quotes, of course.) Also, no spaces or other characters are allowed between the '=' and the '%'.
12	**assign_macro: data stack is empty.** You probably forgot to put something on the data stack on the line preceding the “#assign” statement.
13	**assign_macro: NULL pointer.** The “%s” assignment symbol will test the value on the top of the data stack. In this case the value is zero.

Error Num	Macro Processor Error Messages
	Comments or Description
14	**assign_macro: empty string.**
	The “%s” assignment symbol thinks the value on the top of the data stack points to a byte containing zero.
15	**assign_macro: string too long.**
	The “%s” assignment symbol thinks the value on the top of the data stack points to a very, very long string. Try printing out the string with this : “#s\n” to see for yourself.
16	**assign_macro: error converting value.**
	In practical terms, if you see this message, the “sprintf” function failed to process the string pointed to by the value on the top of the data stack. This is probably a programmer-error. Find him, and kill him.

Appendix C.

C.1. Embedded Linux Features.

The default interp build (“make” with no arguments, or “make clean all”) is feature-for-feature identical to the embedded Linux build. This statement's wording takes into account the fact that you must edit the original Makefile and change the compiler pathname to make an embedded Linux build for your target – unless your target also hosts gcc, and you build on the target.

What are these “features?” I'm glad you asked. Every command (in this document) works as described. If there are build-specific features, the “embedded Linux” description applies. Interp (imain.c actually) will automatically suppress the prompt (“interp> “) when the input has been redirected to a file (e.g. “./interp <myscript.int” .) You may pass option flags to interp, and you may pass arguments through to your script.

C.2. Buffered Input With Embedded Linux.

Most operating systems, including Linux/Unix systems buffer your terminal input for you. Operationally, this means that the device driver – not your own code is actually gathering up the typed input character-by-character, and it isn't until either the end of the file (EOF) or the end of the line (NEWLINE) that the driver signals that input is available to be read. This is true whether you read lines of input or characters.

If you use POLL (“p”) or READ-CHAR (“:”) with –most-- operating systems, they will behave slightly differently than they do on an embedded system with an RTOS, or that is stand-alone (run-at-reset.) On these two types of embedded systems, POLL responds to each individual key press, and READ-CHAR does too.

Testing with Linux shows that POLL continues to return false – even while you are typing characters, until you press the ENTER key. At that moment, the next call to POLL returns true, and it's your responsibility to read everything available. You won't get another true result from POLL until the next time the ENTER key is pressed.

The READ-CHAR command will block, even while you are typing keys, until the ENTER key is pressed, then it will return one character. The next call will get the next character, and so forth, until the NEWLINE is read. Then READ-CHAR blocks again awaiting the next ENTER key.

Interp doesn't try to circumvent this problem. The recommendation is don't use POLL or READ-CHAR on Linux/Unix-based systems. If you can't or choose not to avoid them, write your code and operational procedures accordingly.

C.3. Command Line Interface.

If you type this command: `interp -h`

You'll get this usage information:

```
interp Version 01.07.00

USAGE:
  interp [ options ] [ script args ]

Where:
  options is any combination of:

     -c commands - Execute the interp commands as if they
                   had been typed at the keyboard, with
                   this restriction: no outer interpreter
                   directives ("#" in column one.)

                   If the interp commands include embedded
                   whitespace the whole command string must
                   be surrounded with quotation marks.

                   If the command string includes special
                   characters usually interpreted by the
                   shell, they must be escaped with "\".
     -C          - Continue executing after errors when
                   reading redirected input or when reading
                   from a file (see "-f" option below.)
     -q          - Stop executing after errors when reading
                   redirected input or when reading from
                   a file (see "-f" option below.)
     -f filename - Read from the specified file until EOF.
     -h          - Display usage and quit.
     -l          - List lines from the redirected input.
                   Does not list interactive mode input.
     -n          - Disable verbose-style error messages
     -N          - Non-interactive mode. Because the program
                   doesn't quit until all the options have
                   been processed, if this is the first
                   option, it won't block script arguments.
     -V          - Display version and build info and quit.
     -v          - verbose-style error messages attempt to
                   show where the error occurred.

  script args is a list of whitespace-separated arguments.
              Individual args may be quoted on linux/unix.
              Args beginning with a dash are not supported.
              Args with embedded whitespace must be quoted.
```

NOTES:

- The command line is processed as written, from left-to-right without being reorganized.
- *options*, if present, must precede *script* *args*.
- The "-c", "-f" "-n", "-C", "-q" and "-v" options may be used repeatedly in a command, in any order or sequence.
- Lines echoed by "-l" are preceded by "-->".

Appendix D.

D. Emulation Features.

If you have a build (or make one yourself) that is intended for a emulator-based execution environment, the “Mm” (MEMORY-MAP) command will effectively be a no-op (it returns the HW address.) All other documented commands work. There will not be any prompt suppression for input redirection because this environment (probably) doesn't support it. You will not be able to pass optional flags to interp, but you will be able to pass arguments to your script.

Appendix E.

E.1. Stand-Alone Features.

If you have a build (or make one yourself) that is intended for a stand-alone execution environment, the “Mm” (MEMORY-MAP) command will effectively be a no-op (it returns the HW address.) All other documented commands work. There will not be any prompt suppression for input redirection or CLI support because this environment doesn't support it, and you will not be able to pass arguments to your script (same reason.)