HEWLETT
PACKARD

HP 82441A

FORTH/Assembler ROM

Owner's Manual

For the HP-71

HEWLETT
PACKARD

HP 82441A
FORTH/Assembler ROM

Owner's Manual

For the HP-71

April 1984

82441-90001

Printed in Singapore

Introducing the FORTH/Assembler ROM

The FORTH/Assembler ROM provides an extended software development environment for the HP-71. It contains the following major features:

- A FORTH operating system. This system allows you to write application programs for the HP-71 in FORTH, with a significant advantage in speed over programs written in BASIC. The FORTH operating system coexists with the native HP-71 BASIC operating system, so you can switch between the BASIC and FORTH environments without program or data loss and without having to reconfigure the HP-71. Programs written in either language can execute routines written in the other language. HP-71 FORTH includes string and floating-point operations.
- An assembler. This assembler, written in FORTH, provides nearly the same command set as the assembler used to develop the HP-71 operating system. You can use it to create HP-71 binary files, LEX files to extend the BASIC language, or FORTH primitives.
- A text editor. The editor enables you to create and edit text files, which can be used as source files for BASIC, FORTH, or assembly language programs, or for many purposes unrelated to programming.
- Remote keyboard capability. By using the BASIC keyword KEYBOARD IS (along with the keyword DISPLAY IS provided in the HP 82401A HP-IL Interface), you can use a terminal as an external keyboard and display.

Contents

How To Use This Manual

This manual assumes that you have some experience with FORTH or with assembly language. It documents all operations in the FORTH/Assembler ROM in a reference-oriented manner—you can read the sections that interest you without reading the entire manual.

- If you plan to use FORTH without writing new primitives, read section 2, "The HP-71 FORTH System" and refer to appendix D, "FORTH Words."
- If you plan to create new FORTH primitives, you will also need to read section 4, "The Assembler."
- For an index to FORTH words grouped by function, refer to the inside back cover.
- If you plan to create BIN or LEX files, read section 4, "The Assembler."
- Because both the FORTH system and the assembler use text files for input, read section 3, "The Editor," to learn how to create and edit text files.
- For reference information about any BASIC keyword in the FORTH/Assembler ROM—whether involving FORTH, the editor, or the assembler—refer to appendix C, "BASIC Keywords."

Section 1

Installing and Removing the Module

You can plug the module into any of the four ports on the front edge of the HP-71.

CAUTIONS

- Be sure to turn off the HP-71 (press [f][ON]) before you install or remove any module.
- Whenever you remove one module to make a port available for another module, be sure to turn the HP-71 on and off *while the port is empty* before you install the new module.
- Do not place fingers, tools, or other foreign objects into any port. Such actions can cause minor electrical shocks, interfere with pacemaker devices worn by some persons, and damage port contacts and internal circuitry.

To insert the module, hold the HP-71 with the keyboard facing up and the module with the label facing up, and then push the module into the port until it snaps into place. Be sure to observe the precautions described above.

To remove the module, use your fingernails to grasp the module by the lip on the bottom of its front edge, and then pull the module straight out of the port. Install a blank module in the port to protect its contacts.

Section 2

The HP-71 FORTH System

Introduction

The FORTH/Assembler ROM contains a FORTH system tailored to the HP-71. The advantages of FORTH over BASIC are speed and complete access to the machine. Programs can be written in FORTH, in BASIC, or in both, making use of the best features of each language/system.

FORTH *secondaries* (words constructed from existing FORTH words) can be compiled from keyboard input or from text files created by the editor. The editor is discussed in section 3. In addition, FORTH *primitives* (words written in machine code) can be created by the assembler, which is discussed in section 4.

The word set of the HP-71 FORTH kernel is similar to that defined in the FORTH-83 Standard. This section describes their differences in "Unique Aspects of HP-71 FORTH," which covers enhancements and methods of implementation that are machine-related, and in "FORTH Extentions," which covers enhancements not directly tied to the HP-71. For the complete definition of any FORTH word, standard or nonstandard, refer to appendix D.

References

This section doesn't contain the complete FORTH-83 Standard or tutorial information about FORTH; you can find such material in the following books. You will need to keep in mind the unique aspects of HP-71 FORTH as you read these books.

- Brodie, Leo. *Starting FORTH.* Englewood Cliffs, N.J.: Prentice-Hall, 1981. An effective and entertaining introduction to FORTH.
- *FORTH-83 Standard.* Mountain View, Ca.: FORTH Standards Team, 1983.
- Haydon, Glen B. *All About FORTH: An Annotated FORTH Glossary.* Second edition. Mountain View, Ca.: Mountain View Press, 1983. Some definitions in this manual are borrowed from Dr. Haydon's book.

Using FORTH on the HP-71

Entering and Exiting FORTH. To enter the FORTH environment from the standard HP-71 BASIC environment, type the BASIC keyword `FORTH` and press [END LINE]. The computer then displays the FORTH sign-on message `HP-71 FORTH` and the version. To exit the FORTH environment, type the FORTH word `BYE` and press [END LINE].

The RAM-based portion of the FORTH system, including user-added dictionary words, is contained in an HP-71 file named FORTHRAM. When you exit FORTH, either by executing BYE or by pressing the [OFF] key, the contents of the FORTHRAM file are preserved. Thus the FORTH environment will be in the same state when you reenter as when you exited. If you turn off the HP-71 from the FORTH environment, it will return directly to the FORTH environment when you turn it on. If you purge the FORTHRAM file from the BASIC environment, a new FORTHRAM file will be created when you next execute FORTH.

User Prompts. If you press [END LINE] while the HP-71 FORTH prompt is displayed, FORTH will display OK { 0 }. The OK indicates that FORTH is ready to accept input, and the 0 indicates how many items are on the data stack. If you then type 1 2 3 [END LINE], the FORTH system will display OK { 3 }. You can suppress the OK message by storing a non-zero value into the user variable OKFLG.

Line-editing Keys. All of the HP-71 line-editing keys are functional while in the FORTH environment. Pressing [ATTN] while entering a line will clear the display and leave only the blinking cursor.

Key Redefinitions. The FORTH system duplicates the BASIC method of handling redefined keys. You can switch in and out of user mode while in FORTH, but you must be in the BASIC environment (or use BASICX) to redefine keys.

The Command Stack. The HP-71 command stack is available in FORTH. It operates just as in BASIC, except that in FORTH you can enter the Command Stack just by pressing any of the up- or down-arrow keys—you don't need to press [g][END LINE] first.

Exceptions and the [ATTN] Key. Because the FORTH system can run a program for an indefinite time, it must occasionally check whether a system exception has occurred. FORTH checks for exceptions when it executes ; (semicolon) in a secondary and before it branches in a loop structure. If an exception has occured, FORTH issues the exception poll. An exception can be a service request from the HP-71's internal timers or from other devices, or can result from pressing the [ATTN] key.

Pressing [ATTN] stops the execution of any FORTH word (except HP-IL words, which require pressing [ATTN] twice). Once the FORTH environment recognizes that [ATTN] has been pressed, it executes the system equivalent of ABORT to reset the data and return stacks and to restart the FORTH outer loop (the FORTH system user interface).

Errors. If an error occurs in the FORTH system, all files are closed and an error message is displayed. FORTH error messages sound a tone and preface all errors with FTH ERR:. FORTH error numbers and messages are available through the BASIC keywords ERRN and ERRM$.

If an error occurs in a BASIC O/S subroutine called by the FORTH system, the error message appears as ERR: rather than FTH ERR:.

Advanced FORTH and Assembly Language Programming

This manual contains sufficient information for you to write new FORTH primitives and secondaries. However, if you wish to write FORTH primitives that interact with the native HP-71 operating system, or write HP-71 LEX or binary files, you will need to refer to the *HP-71 Software Internal Design Specification (IDS)*. It comprises three volumes with the following part numbers:

Volume	Description	Part Number
I	*Detailed Design Description*	00071-90068
II	*Module Interface Documentation*	00071-90069
III	*Source Code Listing*	00071-90070

Other detailed documents that you may find useful are:

Description	Part Number
HP-71 Hardware Design Specification	00071-90071
HP-71 HP-IL Detailed Design Description	82401-90023

Unique Aspects of HP-71 FORTH

Twenty-Bit FORTH

Most FORTH systems are implemented on byte-oriented machines with 16-bit addresses. The HP-71, in contrast, is a nibble-oriented machine with 20-bit addresses. To allow access to the entire 1M-nibble address space and to achieve maximum speed, FORTH on the HP-71 is a 20-bit implementation. That is, the data and return stacks are 20 bits wide, and the addresses on those stacks are 20-bit absolute addresses. All quantities on the stacks are 20-bit quantities, regardless of whether a one-byte or 20-bit operation is performed. Unused high-order nibbles are zero or are expected to be zero.

HP-71 FORTH conforms to the FORTH-83 Standard in intent but, because of the nature of the HP-71 CPU, not exactly in effect. The funtionality of the Standard required word set, plus selected words from the extension word sets, are provided in HP-71 FORTH. In most cases, the HP-71 uses the same word names as the Standard. You can determine the behavior of particular HP-71 words compared with their Standard counterparts according to the following general guidelines.

- For operations that deal with *bytes* (such as C@, CMOVE, and FILL), the Standard names are retained for HP-71 FORTH words. Such words will produce the same result as the corresponding Standard words. In several cases analogous words that deal with nibble quantities are also provided; they are listed below in "Nibble and Byte Words."
- For operations that deal with *cells* (such as +, @, and CONSTANT), the Standard names are retained for HP-71 FORTH words. Such words will produce the same result as the corresponding Standard words, except that the quantities manipulated by the words are 20 bits long instead of 16.
- For operations that don't translate well to the HP-71 (with its continuous memory and multiple-file system), the Standard names are replaced for HP-71 FORTH words. For example, LOAD (load from a numbered screen) is replaced by LOADF (load from a named text file), and EXPECT (read up to a specified number of characters) is replaced by EXPECT96 (read up to 96 characters).

The table below lists those words that HP-71 FORTH adds to the Standard word set to perform nibble operations, together with their byte-oriented counterparts.

Nibble and Byte Words

Nibble Word	Action	Byte Word	Action
NALLOT	Allot *n* nibbles.	ALLOT	Allot *n* bytes.
NFILL	Fill *n* nibbles.	FILL	Fill *n* bytes.
N@	Fetch one nibble.	C@	Fetch one byte.
N!	Store one nibble.	C!	Store one byte.
NMOVE	Move *n* nibbles.	CMOVE	Move *n* bytes.
NMOVE>	Move up *n* nibbles.	CMOVE>	Move up *n* bytes.
5+	Increment address by 5 (one cell).	2+	Increment address by 2 (one byte).
5-	Decrement address by 5 (one cell).	2-	Decrement address by 2 (one byte).

Compilation from Files

FORTH compiles new words into the dictionary from "screens" as well as from the keyboard. In traditional versions of FORTH, a screen is a 1K-byte block on a mass storage device (16 lines of 64 bytes each).

Screens. In HP-71 FORTH, a "screen" is a standard HP-71 text file. Each text file consists of a series of text strings of variable length, with each text string preceded by a two-byte length field. The file is terminated by a two-byte marker, FFFF. The editor, described in section 3, can create source screens for FORTH. The name of a screen must be a legal HP-71 file name. The maximum size line that FORTH will process is 96 bytes, which corresponds to the logical display size.

LOADF. The Standard word LOAD is replaced in HP-71 FORTH by LOADF. The inputs to LOADF are two 20-bit numbers: the length of the character string specifying the file to be loaded and the address of this string. LOADF calls HP-71 routines to open, read, and close the file. These routines, in turn, interface to the HP-IL module if it is present, so that screens can reside on HP-IL mass storage devices as well as in HP-71 memory.

FIB Entries. Executing LOADF opens the screen file and creates a *file information block* (FIB) entry in a system buffer called the FIB general purpose buffer. The FIB entry identifies the file and indicates whether the file is in RAM or on mass storage. (If the file is on mass storage, the FIB entry is linked to a system buffer called an I/O buffer that identifies the file.) A *file-information-block number* (FIB#) identifying the FIB entry is stored into the FORTH user variable SCRFIB (*screen FIB#*) to specify the active file.

Mass Memory Buffers. When a file is loaded, its FIB# and the first line of the file are read into a mass memory buffer. There are three mass memory buffers, used in rotation. The contents of the buffer are interpreted until the null at the end of the line (placed there by the FORTH system) is reached. The FORTH word WORD then determines whether this is the end of the active file and, if not, reads the next line from the file into the same mass memory buffer. Each mass memory buffer has the following format.

Format of a FORTH Mass Memory Buffer

FIB#	Line#	Byte count	Data	2 Nulls
1 byte	5 nibbles	2 bytes	Up to 96 bytes	2 bytes

LOADF can save the information necessary to return to the file it is currently interpreting, so LOADF commands can be nested.

Mass Memory. A user can LOADF a file from cassette or disk directly into the FORTH dictionary without first storing the file in RAM. The file will be interpreted a line at a time by reading the line into a FORTH mass memory buffer. However, a file stored on a magnetic card must be read into RAM before it can be loaded into the FORTH dictionary or edited.

File Words

- LOADF accepts input from a specified file rather than the keyboard. Words are executed and definitions are compiled into the user dictionary. The file may exist in RAM or on mass storage.
- BLOCK reads a specified line of the active file into a mass memory buffer and returns the address of the first data byte in the mass memory buffer.
- CLOSEF closes a specified file.
- EOF returns a true flag if the end of the active file has been reached, a false flag if not.
- +BUF returns the address of the next available buffer.
- OPENF opens a FIB entry for a specified file.
- CLOSEALL closes all open files.
- FIRST is a user variable containing the address of the first mass memory buffer in memory.
- LIMIT is a user variable containing the address of the first byte beyond the mass-memory-buffer area.
- PREV is a user variable containing the address of the mass memory buffer last used.
- USE is a user variable containing the address of the mass memory buffer to use next.

- SCRFIB is a user variable containing the either the FIB# of the active file being interpreted by LOADF or else 0.
- BLK is a user variable containing either the line number of the file being interpreted by LOADF or else 0 (input from keyboard).
- LINE# is a user variable containing the line number being loaded from the file specified by SCRFIB.

FORTH/BASIC Interaction

The FORTH/Assembler ROM enables you to temporarily enter the FORTH environment from within the BASIC environment, and vice versa, to take advantage of features of one system while operating from the other. If you press [ATTN] while in a temporary environment, you will be returned to the original environment.

BASIC to FORTH. There are four programmable BASIC keywords that access the FORTH environment.

- FORTHX is a BASIC statement, returning no result.
- FORTHF is a BASIC numeric function that returns the contents of the X-register in the FORTH floating-point stack.
- FORTHI is a BASIC numeric function that returns the number on the top of the FORTH data stack, dropping that value from the stack.
- FORTH$ is a BASIC string function that returns the string specified by the address and character count on the top of the FORTH data stack, dropping those two values from the stack.

FORTHF, FORTHI, and FORTH$ read data from the FORTH environment into BASIC variables without executing any portion of the FORTH system (although FORTHI and FORTH$ alter the data-stack pointer). FORTHX, however, enables you to transfer BASIC data to the FORTH environment and to execute any FORTH words before automatically returning to BASIC.

To execute FORTH operations from the BASIC environment, you use the keyword FORTHX followed by a command string plus up to 14 additional parameters. The optional parameters can be any combination of strings or numeric quantities. The numeric quantities will be pushed onto the FORTH data stack as single-length numbers; strings will be specified on the stack by their addresses and character counts. FORTHX first pushes the optional parameters onto the data stack and then executes the command string. The command string can contain any sequence of FORTH words and parameters, just like input you would enter from the keyboard.

Examples.

```
A$ = FORTH$
X2 = FORTHI
T = TAN (FORTHF)
FORTHX "DROP SWAP TYPE DEPTH .",10,20,X2,"Hello",2*X+6
```

For additional details, refer to appendix C, "BASIC Keywords."

FORTH to BASIC. There are four FORTH words that pass a string (specified on the data stack) to the BASIC system for execution. The string contains BASIC keywords and parameters. The FORTH words call the appropriate BASIC routines to parse and execute the string, as if it were typed to BASIC from the keyboard.

- BASICX passes a string containing BASIC statements to the BASIC system for parsing and execution. It returns no value to the FORTH environment. BASICX can alter the value of BASIC variables. If the string begins with a line number, it will be added to the current BASIC edit file. The string can also call BASIC programs. When the BASIC interpreter finishes, it issues a poll that allows the FORTH system to regain control. If an error occurs, the BASIC system reports the error to the user, and FORTH runs the system equivalent of the ABORT word.
- BASICF passes a string containing a numeric expression to the BASIC system for evaluation. It returns the value of the numeric expression to the X-register in the FORTH floating-point stack.
- BASICI passes a string containing a numeric expression to the BASIC system for evaluation. It returns the value of the numeric expression to the FORTH data stack.
- BASIC$ passes a string containing a string expression to the BASIC system for evaluation. It returns the resulting string to the PAD area and the address and character count to the data stack. The resulting string is truncated to 255 characters if it exceeds this length.

Examples.

```
" BEEP" BASICX
" A(1)" BASICI
" A5/B4*PI" BASICF
" A$" BASIC$
" A6=T(4)*PI" BASICX
" 50 DISP A,B;" BASICX
" A4" BASICF
" STATUS" BASICI
" TIME" BASICF
```

The FORTH/BASIC interface is not reentrant. That is, operations in one environment that are called from the other environment can't exercise the original environment, except to return data. In particular:

- The string passed to the BASIC environment by BASICX can't contain the keyword FORTHX. However, FORTH$, FORTHI, FORTHF are allowed.
- The FORTH command string that is the first argument of FORTHX can't contain the FORTH word BASICX. However, BASIC$, BASICI, and BASICF are allowed.

Applications that respect these two rules will work as long as operations in one environment respect the integrity of the other. For example, don't POKE random data into the FORTHRAM file from BASIC or write over the BASIC environment pointers from FORTH.

HP-IL Operations

To enable controller applications to take advantage of FORTH's speed, the FORTH kernel includes FORTH equivalents of the BASIC statements ENTER and OUTPUT. Additional HP-IL functionality in the FORTH environment can be gained by using the FORTH-to-BASIC words. For example, " STATUS" BASICI returns to the integer data stack a value describing the loop status.

The FORTH word ENTER instructs the HP 82401A HP-IL Interface to receive data from an HP-IL device. The HP-IL module puts the bytes received into a temporary location (the HP-71 math stack). The FORTH system then moves the bytes into an address specified by the user when executing ENTER. The byte count and the address of the data are always returned to the user.

If BASIC system flag −23 is set, ENTER terminates when it receives an End of Transmission message. Otherwise, ENTER continues to request data until its end condition is satisfied. The end condition can be either the reception of a specified number of bytes or of a particular byte value.

The FORTH word OUTPUT instructs the HP 82401A HP-IL Interface to send data to an HP-IL device. The user supplies a byte count and the address of the data to be output.

Two FORTH user variables, PRIMARY and SECONDARY, specify the intended device for OUTPUT and ENTER. Default contents of the variables are 1 for PRIMARY and 0 for SECONDARY. The user must ensure that these variables are properly set up before executing ENTER or OUTPUT.

General Purpose Buffers

Large applications may require blocks of temporary storage that are not a part of the FORTH dictionary space. The HP-71 BASIC O/S provides such temporary storage in the form of general purpose buffers. A maximum of 512 buffers can each contain a maximum of 4095 nibbles, provided that there is enough RAM present to allocate to the buffer. The FORTH/Assembler ROM provides five words to make, find, expand, contract and destory these buffers.

General purpose buffers are maintained at the end of the file chain. The last general purpose buffer is followed by two zero bytes, signifying the end of the general purpose buffer chain. A general purpose buffer has a seven-nibble header field followed by the data space.

Update	Buffer ID	Data length	Data
1 nibble	3 nibbles	3 nibbles	Up to 4095 nibbles

The update nibble is used by the operating system. Refer to the *HP-71 Software IDS* for a description.

Temporary buffers are allocated buffer ID's in the range of E00 to FFF. Because memory contents can move, shifting the position of the buffer, you must use the buffer ID to find the current location of the buffer each time you use it.

General purpose buffers are normally purged by the operating system at coldstart, power on, and during execution of FREE PORT and CLAIM PORT. However, you can mark one buffer to be retained even during these operations by storing its buffer ID into the FORTH user variable VARID. (The assembler uses this variable to save a buffer.)

The following FORTH words deal with general purpose buffers.

- MAKEBF creates a general purpose buffer of a specified size.
- FINDBF finds the current address of a specified general purpose buffer.
- KILLBF deletes a specified general purpose buffer.
- EXPBF expands a specified general purpose buffer by a specified number of nibbles.
- CONBF contracts a specified general purpose buffer by a specified number of nibbles.

Foreign Language Error Messages

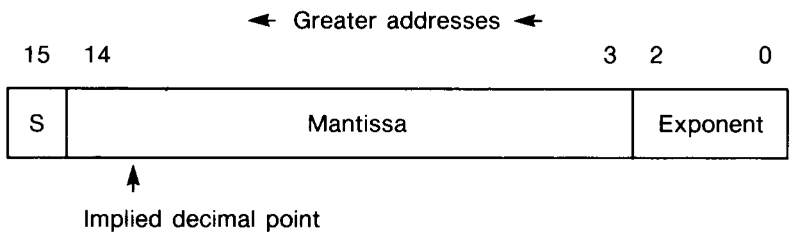
FORTH allows a LEX file to substitute alternative error messages (such as foreign language messages) for its own messages. When a FORTH error occurs, the FORTH system puts together an error number with its LEX ID (47_{10}) and calls the BASIC O/S warning routine. The warning routine allows insertions into the error message. For instance, when FORTH cannot find a word typed in by the user, it gives the message: `FTH ERR: XXX not recognized`. A foreign language LEX file can trap the warning poll and, if the LEX ID of the message is that of the FORTH/Assembler ROM, can substitute its own ID. This causes the message presented to the user to come from the foreign language LEX file rather than from the FORTH/Assembler ROM. Refer to the *IDS* for more information.

FORTH Extensions

Floating-Point Operations

The HP-71 FORTH system includes an HP-RPN-style floating-point stack (X-, Y-, Z-, T-, and LAST X registers). There are FORTH words to manipulate the stack and to use the HP-71 math routines for floating-point operations. There are also FORTH words to create floating-point variables and constants, to fetch and store floating-point numbers, and to display floating-point numbers.

FORTH stores floating-point numbers in the same format as the BASIC system. Each register contains 16 nibbles, as shown below.



Sign. The sign nibble (labeled "S" above) contains 0 for a positive number and 9 for a negative number.

Mantissa. The 12-digit mantissa has an implied decimal point after the most significant digit. The mantissa is not necessarily normalized—that is, it can contain leading zeros to effectively extend the range of the exponent. This field may contain non-numeric data when the register contains an Inf or NaN.

Exponent. The three-digit exponent E is expressed in tens complement, $-499 \leqslant E \leqslant 499$, with the most significant digit in nibble 2. The exponent field is also used to indicate an Inf or NaN: F00 indicates Inf (which may be positive or negative), F01 indicates a quiet NaN, and F02 indicates a signaling NaN.

The following diagram shows how the number −8.23601 E−312 is stored in a register.

15	14										3	2		0
9	8	2	3	6	0	1	0	0	0	0	0	6	8	8

For more information about the formats for floating-point numbers, refer to the *HP-71 IDS*.

A floating-point number is identified in HP-71 FORTH input by the presence of a decimal point. When INTERPRET doesn't identify a character sequence in the input stream as a FORTH word, NUMBER checks the sequence for a decimal point. If there is no decimal point, NUMBER treats it as a potential single- or double-length number. (Many FORTH systems identify double-length numbers by the presence of ., ,, :, /, or a non-leading -. HP-71 FORTH uses only ,, :, and / to identify double-length numbers.)

If the sequence contains a decimal point, the entire sequence is passed to the BASIC O/S routine corresponding to the keyword VAL for evaluation. If the sequence can be evaluated, the result is pushed onto the floating-point stack. "Can be evaluated" means that the character sequence is any valid BASIC numeric expression, which may include literal numbers and BASIC numeric variables. For example, the sequence 10*SIN(30.) entered in the FORTH environment will return the value 5 to the floating-point X-register (assuming that the current HP-71 angular mode is degrees). Similarly, 1.*T1 will return the current value of the BASIC variable T1 to the X-register.

A side effect of the automatic floating-point expression evaluation is that attempted execution or compilation of unrecognized words containing decimal points will result in the BASIC message ERR:Data Type. For example, entering an undefined word XYZABC causes the FORTH message FTH ERR:XYZABC not recognized, but entering the XYZ.ABC will cause the BASIC message ERR:Data Type because of the decimal point.

Floating-point trigonometric functions use the current HP-71 angular mode. FORTH words are provided to switch the mode between degrees and radians. If the mode is set in FORTH, then subsequent BASIC operations will use that mode, and vice versa. Similarly, the floating-point display mode is common to FORTH and BASIC. Floating-point numbers are converted for output (F., FSTR$) in decimal according to the current display mode, which can be set from FORTH or BASIC.

The names of several floating-point operations are prefaced with "F" to distinguish them from operations with similar names. In the following description, x is the contents of the X-register, y is the contents of the Y-register, and so on. All floating-point arithmetic operations return the result to the X-register.

Floating-point Words

- F+ returns $y + x$.
- F- returns $y - x$.
- F* returns $y \times x$.
- F/ returns $y \div x$.

- `X^2` returns x^2.
- `10^X` returns 10^x.
- `SIN` returns the sine of x.
- `COS` returns the cosine of x.
- `TAN` returns the tangent of x.
- `E^X` returns e^x.
- `1/X` returns the reciprocal of x.
- `SQRT` returns the square root of x.
- `Y^X` returns y^x.
- `LGT` returns $\log_{10}$ of x.
- `LN` returns the natural log of x.
- `ATAN` returns the arc tangent of x.
- `ASIN` returns the arc sine of x
- `ACOS` returns the arc cosine of x.
- `RDN` rolls down the stack ("down" in the HP-RPN sense).
- `RUP` rolls up the stack ("up" in the HP-RPN sense).
- `X<>Y` swaps x and y.
- `X`, `Y`, `Z`, `T`, and `L` return the address of the corresponding floating-point register.
- `LASTX` pushes the contents of the LAST X register onto the floating-point stack.
- `FENTER` pushes the contents of the X-register onto the floating-point stack.
- `RCL` fetches a floating-point number from the address on top of the data stack and pushes it onto the floating-point stack.
- `STO` stores x into the address on top of the data stack.
- `F.` displays x without altering the floating-point stack.
- `FVARIABLE` creates a floating-point variable in the FORTH dictionary.
- `FCONSTANT` creates a floating-point constant in the FORTH dictionary.
- `X=0?`, `X>Y?`, `X<Y?`, `X=Y?`, `X#Y?`, `X<=Y?`, and `X>=Y?` perform the specified test and, if true, push a true flag (-1) onto data stack; or if false, push a false flag (0) onto data stack.
- `DEGREES` sets the active angular mode to degrees.
- `RADIANS` sets the active angular mode to radians.
- `STD`, `FIX`, `ENG`, and `SCI` set the display format.

String Operations

HP-71 FORTH includes words to create string constants, string variables, and string arrays; to compare strings; to manipulate portions of strings (substrings); and to match string patterns. A string is stored in memory in the following format.

Format of a String in Memory

Maximum length	Current length	Character string (left-justified)
1 byte	1 byte	Maximum-length bytes

A string in memory is usually represented on the stack by a pair of values: an address and a character count (count on top). The address is the location of the first character of the string in memory, and the character count is the current length. This is the format expected by the standard word TYPE.

Occasionally a "counted string" in memory is represented on the stack simply by an address. The address is the location of the string's length byte, which is followed in memory by the string's characters. This is the format expected by the standard word NUMBER.

String constants are created by the word ", which puts the maximum-length byte, the current-length byte, and the string in the pad (system scratch space). String constants are thus very temporary—don't type in two string constants followed by a comparison operator, because the second will have been created on top of the first. String constants are used mainly to set the values of string variables, but you can also use them with other functions as long as you notice when the pad is being overwritten.

String variables are dictionary entries much like numeric variables. At the PFA are the maximum-length and current-length bytes followed by the string. The code field contains the address of code that returns to the stack the address of the first character (PFA + 4) and the current length.

String variable arrays are similar to single variables, but the first two bytes at the PFA indicate the maximum length of each element and the number of elements in the array. Next come the strings, each in the format described above: maximum length, current length, string. The *n*th element is accessed by typing *n array name*; the CFA points to code that returns the address and count of this element, which can be manipulated just like a regular string variable or constant.

String Words

- " creates a temporary string.
- ASC returns the ASCII code for the first character in a string.
- CHR$ returns a temporary string of length 1 for a specified ASCII code.
- END$ creates a temporary substring from the last part of a string.
- FSTR$ converts the number in the X-register to a string.

- LEFT$ creates a temporary substring from the first part of a string.
- MAXLEN returns the maximum allocated length of a string.
- NULL$ creates a temporary string of zero length.
- POS returns the position within a string of a substring.
- RIGHT$ creates a temporary substring of specified length from the last part of a string.
- S= returns a true flag if two strings are equal, a false flag if not.
- S< returns a true flag if $string_1 < string_2$, a false flag if not.
- S! stores $string_1$ into $string_2$.
- S<& adds a copy of one string to the end of another string.
- S>& adds a copy of one string to the beginning of another string.
- SMOVE stores a string at a specified address.
- STR$ converts a double number into a string.
- STRING creates a string variable.
- STRING-ARRAY creates a string-array variable.
- SUB$ creates a temporary substring from the middle part of a string.

Vocabularies

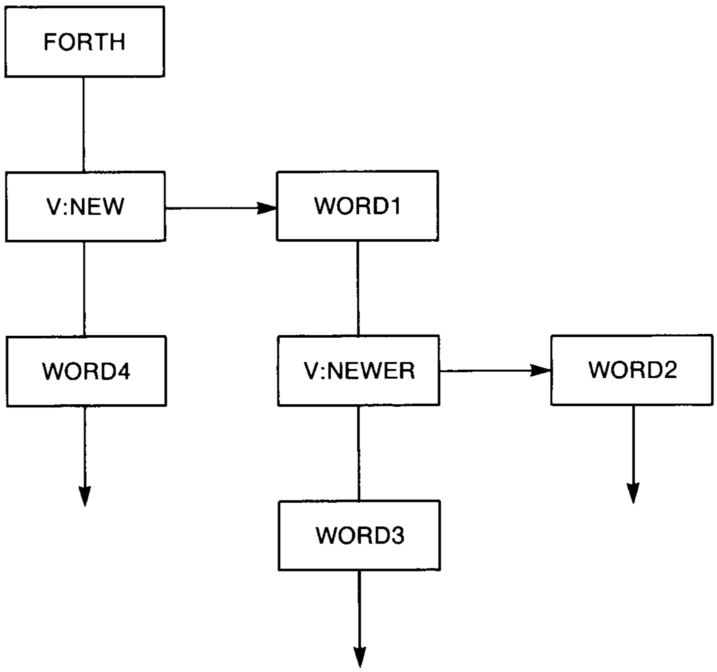
The HP-71 FORTH vocabulary structure is a tree-like structure. Every vocabulary contains the word FORTH, which sets the FORTH vocabulary as the CURRENT vocabulary (to which subsequent new words will be added). This is because FORTH is the first word in the FORTH vocabulary, and all vocabularies eventually chain back to the FORTH vocabulary. The following example creates a vocabulary called NEW.

```
VOCABULARY NEW
NEW DEFINITIONS
```

In the first line, VOCABULARY creates a new vocabulary called NEW. This entry, NEW, is entered into the current vocabulary, which is the FORTH vocabulary. Execution of NEW in the second line makes NEW the CONTEXT vocabulary (in which searches for words begin). DEFINITIONS sets the CURRENT vocabulary to be the same as the CONTEXT vocabulary. To continue the example:

```
: WORD1 ;
VOCABULARY NEWER
NEWER DEFINITIONS
: WORD2 ;
```

Now three vocabularies exist: FORTH, NEW, and NEWER. Suppose that WORD3 is added to the NEW vocabulary, and WORD4 is added to the FORTH vocabulary. The diagram below shows the result.



If either NEW or NEWER is the CONTEXT vocabulary, the word search won't find WORD4 in the FORTH vocabulary. If NEWER is the CONTEXT vocabulary, the word search won't find WORD3 in NEW, but it will find WORD1. In terms of the diagram, the word search proceeds in vocabularies other than the CONTEXT vocabulary by moving leftward and upward, never rightward or downward.

It is important to realize that, while FORTH can be reached from any vocabulary, the converse is not always true. NEW can be found when FORTH, NEW, or NEWER is the CONTEXT vocabulary, but NEWER can be found only when NEW or NEWER is the CONTEXT vocabulary.

Whenever an error occurs, FORTH becomes both the CONTEXT and CURRENT vocabulary.

Error Trapping

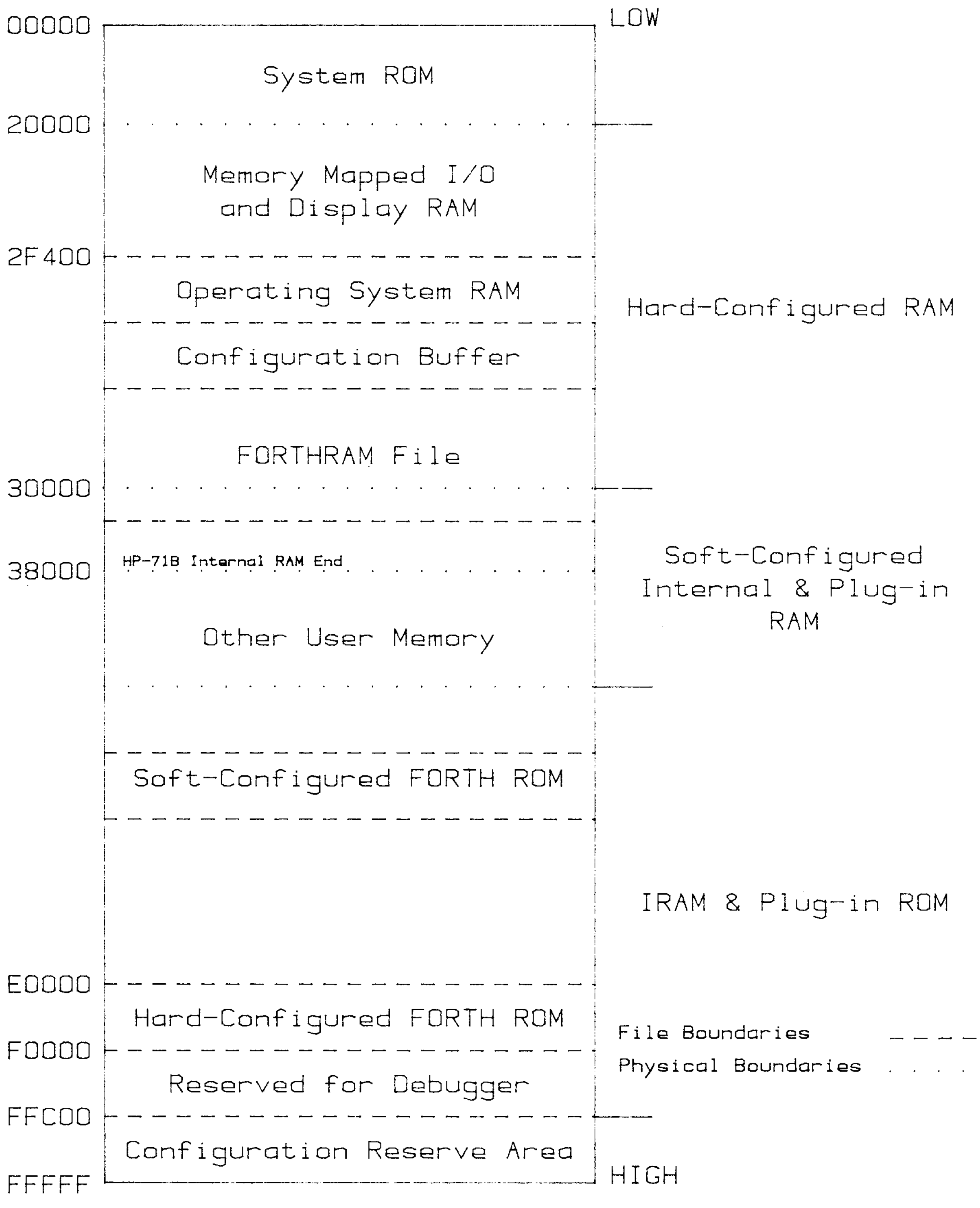
When an error occurs during execution of a FORTH word, a system routine equivalent to ABORT or ABORT" is executed. Normally, these routines will reset the data and return stacks and return to the outer interpreter loop for new input. However, HP-71 FORTH provides an error-trapping facility that can allow FORTH execution to continue after an error.

The user variable ONERR contains the CFA of a word to execute when an error occurs. The system abort routines check the contents of ONERR; if ONERR contains zero, the routines will exit normally through QUIT. If the value of ONERR is non-zero, execution will be transferred to the address contained in ONERR. The stacks are not reset, so the error routine has a chance to recover some or all of the state of the systerm at the time of the error. (The words ABORT and ABORT" don't respect the setting of ONERR.)

FORTH Memory Organization

HP-71 Memory

The diagram below shows a map of the HP-71 memory with the FORTH/Assembler ROM installed.



The FORTH/Assembler ROM uses addresses in three regions:

- Hard-configured ROM, from E0000 to EFFFF. The hard-configured ROM contains the FORTH operating system, the built-in FORTH dictionary, and the assembler.
- Soft-configured ROM. This is a 16K-byte module that contains the editor, all BASIC keywords in the FORTH/Assembler ROM, and the initialization routines for the FORTH environment.
- The FORTHRAM file. This file is stored in user memory and contains the changeable parts of the FORTH system—user variables, user dictionary, and so on. When the FORTH system is active, FORTHRAM will always be the first file in user memory.

The FORTHRAM File

When FORTH or FORTHX is executed, a file called FORTHRAM is created (unless it exists already). FORTHRAM contains both the FORTH system's status information and all words added by users. FORTH has been assigned LIF file types E218 and E219. When the FORTH/Assembler ROM is plugged in and a CATALL is executed, the FORTH system intercepts the file-type poll and displays FORTH instead of the numeric file type for FORTHRAM. Initially FORTHRAM contains about 3K bytes. You can enlarge the file (to expand the dictionary) or contract the file (at the expense of the dictionary), but only after the entire 3K-byte file exists.

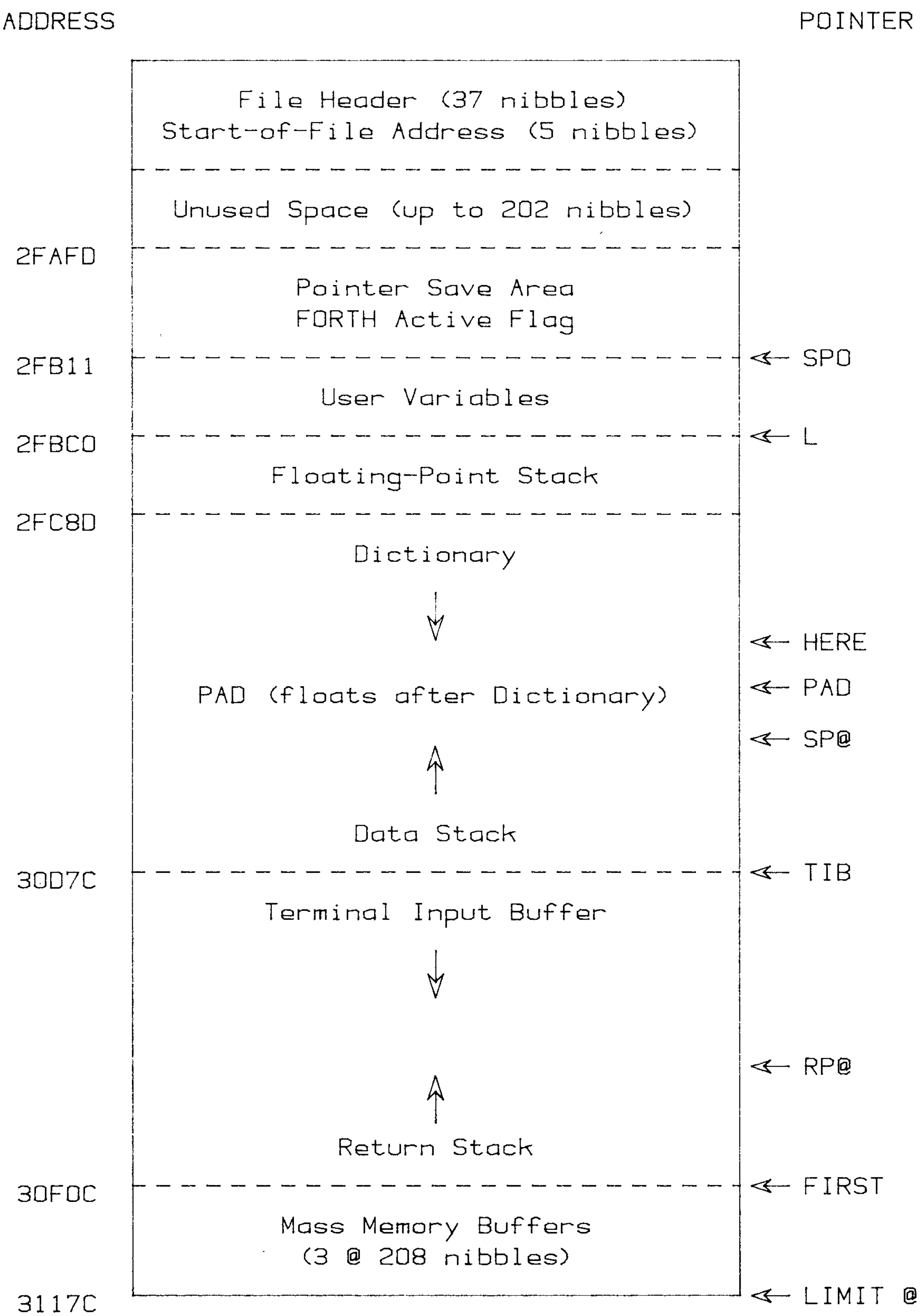
To re-enter FORTH when FORTHRAM is no longer the first file in memory, 37 bytes are required to swap the file back into the first position. If there is not enough memory, an error message is displayed.

Copying FORTHRAM. You can rename, copy, and purge FORTHRAM using HP-71 BASIC file commands. This enables you to have multiple versions of the FORTH system, each containing a different user dictionary. When you have multiple FORTH files, the file currently named FORTHRAM will be the active FORTH file when you enter the FORTH environment. Also, if you make backup copies of your FORTH system, you can restore your system following a memory loss (common when programming in FORTH) by reloading a FORTHRAM file from mass storage rather than by recompiling the dictionary. The FORTH/Assembler ROM is not required to copy the FORTHRAM file out to mass storage, but it is required to copy FORTHRAM back into RAM.

Contents of FORTHRAM. The diagram on the opposite page shows the structure of FORTHRAM. At the beginning of the file are 37 nibbles of system overhead—file name, file type, link to next file, and so on. Next is the address of the FORTHRAM file; when the FORTH system is re-entered, this address indicates whether FORTHRAM has been moved. Next is up to 101 bytes of unused space, depending on FORTHRAM's starting address. Enough space is added to ensure that FORTHRAM's data begins at 2FAFD.

Starting at 2FAFD is the housekeeping information needed to save the FORTH pointers when a system routine alters all of the CPU registers. At 2FB11 starts the block of FORTH system variables called "user variables." The floating-point stack follows the user variables in the file. The user dictionary space starts above the floating-point stack. When the FORTHRAM file is created, 2K bytes (the minimum required by the FORTH standard) are allocated for dictionary entries. The data stack is deep enough to hold a minimum of 40 entries. The return stack and the Terminal Input Buffer share 200 bytes, of which a maximum of 98 bytes can be used by the Terminal Input Buffer (keyboard entry is limited to 96 characters, and FORTH appends 2 null characters for its own use). The mass memory buffers are allocated 312 bytes.

FORTHRAM File Structure



The tables below show the details of a newly created FORTHRAM file. Although the FORTHRAM file is always the first file in user memory, its starting address varies according to the length of the HP-71 configuration buffers, which precede FORTHRAM in memory. The current address of the start of the file can be found by executing

`ADDR$('FORTHRAM')` in BASIC, or
`" FORTHRAM" FINDF` in FORTH.

System Save Area

Address	Contents
2FAFD	Data-stack pointer save.
2FB02	Return-stack pointer save.
2FB07	Instruction pointer save.
2FB0C	FORTH active flag.

User Variables

Address	Contents	FORTH Words To Return Contents
2FB11	Pointer to bottom of data stack.	`S0` or `SP0 @`
2FB16	Pointer to bottom of return stack.	`RP0 @`
2FB1B	Pointer to TIB.	`TIB`
2FB20	Next buffer.	`USE @`
2FB25	Most recent mass storage buffer.	`PREV @`
2FB2A	First mass storage buffer.	`FIRST @`
2FB2F	End of FORTHRAM + 1.	`LIMIT @`
2FB34	Vocabulary link.	
2FB39	Buffer record size.	
2FB3E	Number of characters in TIB.	`#TIB @`
2FB43	Maximum word-name length.	`WIDTH @`
2FB48	Warning mode.	`WARN @`
2FB4D	Enable/disable `OK` in `QUIT`.	`OKFLG @`
2FB52	Line number in current `LOADF` file. (Reset when load error occurs.)	`BLK @`
2FB57	Offset in TIB.	`>IN @`
2FB5C	Number of characters read by `EXPECT96`.	`SPAN @`
2FB61	FIB# of active `LOADF` file.	`SCRFIB @`
2FB66	Address of CONTEXT vocabulary.	`CONTEXT @`

User Variables (continued)

Address	Contents	FORTH Words To Return Contents
2FB6B	Address of CURRENT vocabulary.	CURRENT @
2FB70	Compilation flag.	STATE @
2FB75	Current base.	BASE @
2FB7A	Number type indicator.	
2FB7F	Unused. Available for user programming.	
2FB84	Current position of stack. (Used by compiler.)	
2FB89	Pointer to last character in display string.	
2FB8E	FORGET boundary.	FENCE @
2FB93	Next available nibble in dictionary.	
2FB98	Buffer size in nibbles.	
2FB9D	Line number in current LOADF file. (Preserved after load error.)	LINE# @
2FBA2	Return address for BASIC keywords.	
2FBA7	Reserved for HP-IL use.	
2FBAC	Secondary HP-IL address.	SECONDARY @
2FBB1	Primary HP-IL address.	PRIMARY @
2FBB6	On-error execution address.	ONERR @
2FBBB	Error-occurence flag.	

Floating-Point Stack Registers

Address	Contents	FORTH Words To Return Value to X-register
2FBC0	LAST X register.	L RCL
2FBD0	X-register.	X RCL
2FBE0	Y-register.	Y RCL
2FBF0	Z-register.	Z RCL
2FC00	T-register.	T RCL
2FC10	System use. (Eight bytes for file name.)	

Vectored Execution Addresses

Address	Contents
2FC20	INTERPRET
2FC25	CREATE
2FC2A	NUMBER
2FC2F	, (*comma*)
2FC34	C, (*c-comma*)
2FC39	ALLOT
2FC3E	For *xxx* isn't unique message.

Assembler User Variables

Address	Contents	FORTH Words To Return Contents
2FC43	ID of buffer to preserve.	VARID @
2FC48	Page length.	PAGESIZE @
2FC4D	Name of listing file.	LISTING
2FC79–2FC8C	System use.	

User Dictionary and Above

Address	Contents	FORTH Words To Return Contents
2FC8D	FORTH word.	' FORTH 2- -1 TRAVERSE 5-
2FCB1	Start of first user-defined word. (Addresses above 2FCB1 are variable.)	
2FCB1*	End of dictionary. (Next available nibble.)	HERE
2FD0B*	Pad. (Floats after dictionary.)	PAD
30D7C*	Top of data stack.	SP@
30D7C†	Bottom of data stack = Start of TIB.	S0, SP0 @, or TIB
30F0C†	Bottom of return stack = Start of first mass storage buffer.	RP0 @ FIRST @
3117C†	First nibble after FORTHRAM.	LIMIT @

* Changes when words are compiled or executed.
† Changes when GROW or SHRINK is executed.

The FORTH Dictionary

When you type in a word to be executed or when the system compiles a word from a source file, FORTH must search through its dictionary to find the word and its execution address. HP-71 FORTH searches the RAM part of the dictionary first (the user dictionary) and then the ROM part (the built-in FORTH words). Words in ROM are arranged according to word length to minimize the search time. The length of the target word is used as an index into a jump table so that, for example, only the list of three-character words are searched for a three-character word. A test is also made to ensure that the word is not longer than the longest word in the ROM portion of the dictionary.

As an example of an entry in the dictionary, the structure of a FORTH primitive CMOVE is shown below. Although this word is in the ROM dictionary, its structure is typical of words in either the ROM or RAM parts of the dictionary.

Structure of a Word

Field	Address	Contents
Link	LFA = E3AEE	E3AA6
Name	NFA = E3AF3	5834D4F4655C
Code	CFA = E3AFF	E3B04
Parameter	PFA = E3B04	*code*

Link Field. The contents of the link field (E3AA6) point to the name field of the previous dictionary entry.

Name Field. The first byte of the name field, 85, is 10000101 in binary (note that the byte's two nibbles are reversed, with "5" stored at a smaller address than "8"). The byte's high-order bit is set to indicate the start of the name field, and the second bit is clear to indicate that the word is not immediate. The third bit (the smudge bit, set during compilation of a secondary to prevent the word being used in its own definition) is clear. The five low-order bits have a value of 5 to indicate that the name is five characters long; the maximum length is 31 characters. The second and subsequent bytes in the name field are the ASCII representation of the word's name, with the high bit of the last character is set to indicate the end of the name field. Here the last character is "E" with ASCII value 01000101, so the binary value 11000101 is stored (with nibbles reversed) as 5C.

Code Field. Because CMOVE is a primitive, the code field contains this word's PFA, E3B04, so that the code in the parameter field will be executed. In a secondary, the code field contains the address of the run-time code of :, which nests the FORTH program pointer down one level.

Parameter Field. Because CMOVE is a primitive, the parameter field contains executable code. In a secondary, the parameter field contains the CFAs of the words that make up the secondary.

The ROM-based dictionary contains all of the built-in FORTH words except FORTH, which is always the first word in the RAM-based dictionary. To speed compilation, the FORTH system doesn't search the entire ROM-based dictionary. The ROM-based dictionary is composed of 13 separate linked lists, with each list containing words of a specific length, so the FORTH system searches only the list for the appropriate word length.

At E0000 is a jump table with 13 entries. Each entry contains a pointer to the beginning of the word list for words of a specific length, from 0 through 12 characters. To illustrate this structure, a word VLIST appears below that will display all words in the ROM dictionary. Note that the pointer initially indicates the list of one-character words.

```
HEX
: VLIST E0005
 D 1 DO
  DUP @
   BEGIN DUP
    COUNT 1F AND 1-
    DUP >R 2* SWAP DUP >R
    + C@ 7F AND R> R>
    TYPE EMIT CR 5- @ ?DUP 0=
   UNTIL
  5+ LOOP DROP ;
```

The HP-71 File System

The HP-71 contains a 64K-byte operating system kernel that starts at address 00000. The kernel performs various control functions and contains the BASIC interpreter. External software may be added to the machine in the form of files that the kernal interprets or executes directly. These files may be directly plugged into the machine through ROM or RAM modules, or copied into the machine from external media such as cards or tape.

File Types

The following file types are directly supported by the HP-71 mainframe. OEM software developers may support other file types by first reserving the file type with Hewlett-Packard and then including the appropriate poll handlers in a LEX file. Each file type is identified by a 16-bit value that conforms to Hewlett-Packard's Logical Interchange Format for Mass Media.

When HP-71 files are stored on external media, file security and privacy are encoded, if applicable, in the numeric file type as shown in the chart below. When files are stored in memory, privacy and security are encoded in the flags field of the file header, and the file type stored in the file header is *always* the normal file type.

Numeric File Type

Type	Description	Normal	Secure	Private	Execute Only
BASIC	Tokenized BASIC program.	E214	E215	E216	E217
BIN	HP-71 machine language.	E204	E205	E206	E207
DATA	Fixed data.	E0F0	E0F1	n/a	n/a
LEX	Language extension.	E208	E209	E20A	E20B
KEY	Key assignment.	E20C	E20D	n/a	n/a
SDATA	Stream data.	E0D0	n/a	n/a	n/a
TEXT	ASCII text, in LIF Type 1 format.	0001	E0D5	n/a	n/a
FORTH	FORTHRAM file.	E218	E219	n/a	n/a

Four of these file types are program files: BASIC, BIN (Binary), LEX (Language Extension), and FORTH. BASIC files may be developed on the HP-71 using the built-in BASIC interpreter. BIN, LEX, and FORTH files may be developed on the HP-71 using the FORTH/Assembler ROM.

Types of Program Files

Type	Format	Method of Invocation	Mode of Execution
BASIC	Tokenized BASIC statements.	RUN or CALL command.	Interpretation.
BIN	Machine language (binary).	RUN or CALL command.	Direct execution.
LEX	Language extension file; adds BASIC keywords, messages, and functional extensions; written in machine language.	Through its added BASIC keywords and by polls from operating system.	Direct execution.
FORTH	FORTH vocabulary.	Through FORTH interpreter.	Threaded interpretation.

Structure of the File Chain

The HP-71 maintains a file area in main RAM that is composed of a linked list, or chain, of file entries. (Each plug-in ROM module and independent RAM contains its own file chain.) At the beginning of each file entry is a file header. The file header contains identifying information about the file along with the link to the next file entry in the chain. The end of the chain is marked by a zero byte. Each file header contains the following fields:

Fields in a File Header

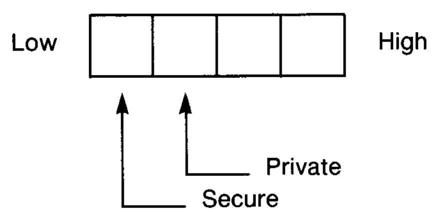
Field	Size
File name	16 nibbles
File type	4 nibbles
Flag	1 nibble
Copy Code	1 nibble
Creation Time	4 nibbles
Creation Date	6 nibbles
Link	5 nibbles

File Name. The file-name field contains the eight-character file name in ASCII, filled with blanks to the right (high memory).

File Type. The file-type field contains a four-digit hex integer, listed in the "File Types" table above.

Flag. The flag field contains four system flags. The two bits in the low end of the flag field indicate file protection. When set, the lower of the two bits indicates a file is SECURE; the higher of the two bits indicates a file is PRIVATE. The remaining two bits of the flag field are unused.

File Header-Flags



Copy Code. The copy-code field indicates the file attributes neccessary for external copying.

Creation Time and Creation Date. The creation-time and creation-date fields represent the time and date in BCD. The time field contains four nibbles; the minutes are in the low byte, and the hour is in the high byte. The date field contains six nibbles; the day is represented in the low byte, the month in the next byte, and the year in the high byte. For example, the internal representation of 03:45 on December 16, 1981, would be as follows:

	Time				Date						
Low	5	4	3	0	6	1	2	1	1	8	High

Link. The link field contains the offset to the next file (header) in memory.

Section 3

The Editor

The FORTH/Assembler ROM editor enables you to create, modify, copy, list, and print text files. These files are suitable source files for the FORTH system and the assembler. This section describes the editor's operation in three parts:

- "Overview of the Editor" describes how to enter and exit the editor, the two types of editor commands, and editor operations other than commands.
- "Editor Commands" describes the specific commands that act on the edit file.
- "Editor Files" describes files used in the editor's operation.

Additional material related to the editor appears in the appendixes. Appendix B, "Error Messages," includes the error messages generated by the editor. Appendix C, "BASIC Keywords," includes the editor keywords DELETE, EDTEXT, FILESZR, INSERT#, MSG$, REPLACE#, SCROLL, and SEARCH, which you can use in your own BASIC programs.

Also in the keyword dictionary is the BASIC keyword KEYBOARD IS. Used in conjunction with DISPLAY IS, KEYBOARD IS allows almost any terminal (or computer acting as a terminal emulator) to be an extension of the HP-71 keyboard and display. Although this keyword isn't strictly a part of the editor, a full-size keyboard can greatly aid text input.

Overview of the Editor

The editor is a BASIC program; when you enter the editor, the HP-71 PRGM annunciator appears. You can enter the editor directly from the FORTH environment by using BASICX:

```
" EDTEXT SCREEN" BASICX
```

will run the editor on a file named SCREEN. When you exit the editor, the HP-71 will automatically return to FORTH. Here is a FORTH word that you might find useful:

```
: EDIT " EDTEXT SCREEN" BASICX " Loading..." TYPE " SCREEN"
LOADF ;
```

When you execute EDIT, the editor will open the file SCREEN for editing. When you exit the editor, the display will show Loading... while the FORTH system compiles the contents of SCREEN into the dictionary.

To enter the editor from BASIC, type EDTEXT *filename* [END LINE]. The editor opens that file for editing or, if *filename* is a new name, creates a new file with that name. The display then shows Line *n*, Cmd:, where line *n* is the current line in the file. Line numbers, which begin with 1, are for reference only; they aren't stored in the file. If you're at the end of the file, the current line is indicated by Eof.

When the Cmd: prompt is displayed, you can:

Display the Current Line. To temporarily display the current line, hold down the [END LINE] key. When you release the key, the Cmd: prompt returns.

Move to A Different Line. There are three methods for moving to a different line:

- To move to any line in a file, enter the line number and press [END LINE]. For example, to move to line 2, enter 2 [END LINE].
- To move to the previous line (smaller line number), press [↑]. To move to the following line (larger line number), press [↓].
- To move to the beginning of a file, press [g][↑]. To move to the end of a file, press [g][↓].

Display the File Name. If you press [↑] when the line 1 is the current line, the editor will display the name of the edit file. To display the file name from any place in the file, hold down [f][+]. When you release [+], the Cmd: prompt returns.

Execute a Command. The editor commands, each of which is described in detail below, fall into two classes:

- The commands T (Text) and I (Insert) are used for entering text. Once you execute the Text or Insert command, the editor remains in Text or Insert mode until you press [RUN] or [ATTN]; only then will the Cmd: prompt return.
- All other editor commands perform specific operations, after which the Cmd: prompt returns automatically.

Exit the Editor. To end the editing session, enter E [END LINE]. The editor closes the edit file and displays Done: *filename*. If you decide not to keep this file, purge it following the instructions in section 6 of the *HP-71 Owner's Manual.*

When you call the editor, a copy of your own redefined keyboard is stored and the editor's key redefinitions are added to yours. Unless the editor keys are the same keys you've redefined, your redefined keys are still available to you while the editor is active. When you exit the editor, the combined redefined keyboard is purged and your own redefined keyboard is restored.

To override a key assignment, use the [g][1 USER] key. This will deactivate USER mode for the next key pressed. Note that if you enter the editor from FORTH, disable USER mode, and then either press [ATTN] or cause any error, the HP-71 will immediately return to the FORTH environment, leaving the current edit file in a corrupted state.

Editor Commands

You can enter the following editor commands whenever the `Cmd:` prompt is displayed. Some editor commands require parameters such as line numbers or a file name. These parameters are identified in syntax diagrams for each command. Any default values for parameters are given after the syntax diagram. In the syntax diagrams:

- Items [enclosed in square brackets] are *optional parameters.* Some optional parameters are nested within others. This indicates that the parameter in the outer pair of brackets must be present *before* the parameter in the inner pair can be included.
- Items shown in `DOT MATRIX` text must appear exactly as shown (although either upper or lower case is acceptable).
- There are two substitute characters that can be used for any *line-number* parameter. A period (.) indicates the *current line*, and the pound sign (#) indicates the *last line in the file.*
- Two adjacent numeric parameters must be separated by a space or comma. No separation is required between a numeric parameter and an alphabetic parameter.

The Text (T) and Insert (I) Commands

[*line number*] T

[*line number*] I

Default value: *line number* = current line

The Text command is your primary means of adding text to the edit file. When you enter Text mode, the current line appears in the display with the cursor at the beginning of the line. Modify the current line as desired (using the standard HP-71 editing keys) and then press [END LINE]. The editor stores these changes to the current line and then makes the following line the current line, displaying it to start the cycle again.

The Insert command permits you to add a line or a series of lines into the middle of a file. When you enter Insert mode, the current line is displayed until you press a key. Type in the text for the new line (using the standard HP-71 editing keys) and press [END LINE]. The editor inserts the new line into the file, just before the current line, and then displays the next line number as the new current line. (The text for the new current line is the same as before; only its line number changes.) Flag one is on to indicate that you are in Insert mode.

Either Text mode or Insert mode work equally well for entering text at the end of a file. In either mode, text is stored in the file only when you press [END LINE]. If you make changes or enter text and then move to another line (by using [↓] or [↑]) *before* you press [END LINE], no changes or text will be stored.

To exit from Text or Insert mode, press [RUN] or [ATTN].

The List (L) and Print (P) Commands

[*beginning line number* [*ending line number*]] L [*number of lines*][N]

[*beginning line number* [*ending line number*]] P [*number of lines*][N]

Default values: *beginning line number* = current line
ending line number = last line

The List and Print commands are similar. List causes the specified lines of text to be displayed consecutively on the current display device (usually the display window or a monitor). If you have an HP 82401A HP-IL Interface installed and a printer assigned, Print causes the specified lines to be printed. When no printer is present, Print responds like List.

After listing or printing, the *current line* will be the line after the ending line number. The following examples show some List and Print commands with parameters:

L	List from the current line to the end of the file.
,L10	List from the current line to the end of the file, or just 10 lines, whichever comes first.
3 9 L N	List from line 3 to line 9 with line numbers.
1 L20N	List, with line numbers, the entire file or the first 20 lines, whichever comes first.
P	Print from the current line to the end of the file.
,P5N	Print five lines starting at the current line, with line numbers.
1P N	Print the entire file with line numbers.

The Copy (C) and Move (M) Commands

[*beginning line number* [*ending line number*]] C [*filename*]

[*beginning line number* [*ending line number*]] M [*filename*]

Default values (Edit file): *beginning line number* = current line
ending line number = beginning line number

(Other file): *beginning line number* = line 1
ending line number = last line

The Copy command permits you to copy one or more lines from one place in the file to another place in the file. You can also copy part of another file into your edit file. Copy always inserts the copied text before the current line. The Move command is similar to the Copy command but deletes the text in the original location.

If no filename is specified, the indicated lines come from the edit file. If a filename is specified, the indicated lines come from the specified file. You can't copy or move a block of text that includes the current line, unless the current line is the first or last line of the block of text.

The Working... message is displayed when you copy or move text.

Here are some examples of the Copy and Move commands:

C	Duplicate the current line.
5 C	Copy line 5 and insert it before the current line.
3 9 M	Move lines 3 through 9 from within the edit file and insert them before the current line, then delete the original lines 3 through 9.
C CAT	Copy the file CAT and insert the lines before the current line.
20 C ABC	Copy lines 20 through the last line of the file ABC and insert the lines before the current line in the edit file.

The Delete (D) Command

[*beginning line number* [*ending line number*]] D [*filename* [+]]

Default values: *beginning line number* = current line
ending line number = beginning line number

The Delete command deletes one or more lines from the edit file. You can place the deleted lines into a new file or, using the + option, append the lines to an existing file. When you execute Delete with line number parameters specifying more than one line, the message OK to delete? Y/N: will appear. You must answer Y before the editor will complete the deletion. If you answer N, the Command Prompt returns.

The Working... message is displayed when you use Delete.

The following examples show some uses of the Delete command:

D	Delete the current line.
12 32D	Delete lines 12 through 32.
4 9 D CACHE	Delete lines 4 through 9 and store them in a new file called CACHE.
2 21D ARCHV+	Delete lines 2 through 21 and append them to the end of a file called ARCHV.

You can not purge a file while you are in the editor, but you can delete all of the text and leave an empty file. Refer to section 6 of the *HP-71 Owner's Manual* for instructions on how to purge a file.

The Search (S) and Replace (R) Commands

[*beginning line number* [ending line number]][?] S/*string1*[/]

Default values: *beginning line number* = current line + 1
ending line number = last line

[*beginning line number* [*ending line number*]][?] R/*string1*/*string2*[/]

Default values: *beginning line number* = current line
ending line number = beginning line

The Search and Replace commands allow you to search through a file for a certain string of characters *string1*. If you use a Search command, the first line containing *string1* becomes the current line. If you use a Replace command, all occurrences of *string1* are replaced by *string2*, and the last line containing *string1* becomes the current line. If either command can't find *string1*, it displays `Not Found`.

These commands search the specified lines in the edit file for the string indicated between the slashes (/). These slashes act as *delimiters*, marking the string's boundaries. If you need / as a normal character in your search string, you can use any other character (except a blank space) as the delimiter. The first non-blank character after the command S or R is the delimiter. The last delimiter is optional unless another command follows this command.

Search and Replace can distinguish between uppercase and lowercase letters. For example, a search for the string `jack` will not find the string `Jack`.

The following examples show some Search commands and Replace commands with parameters:

`S/Jack`	From the next line through the end of the file, search for the first occurrence of the string "Jack."
`3 7 S/Jill`	From line 3 through line 7, search for the string "Jill."
`R/cat/dog/`	Replace all occurences of "cat" with "dog" on the current line.
`4 7R/cat/dog`	On lines 4 through 7, replace all occurences of "cat" with "dog."
`R*3/4*3/8`	On the current line, replace all occurences of "3/4" with "3/8." The character * is used as the delimiter so that slashes may occur in the strings.
`.#R/meet//`	From the current line to the end of the file, replace "meet" with the null string (that is, delete "meet").

If the replacement *string2* causes the line to be longer than 96 characters, the editor will redimension variables, causing a slight delay.

Response Option. You can more closely control the Search and Replace commands by including the ? option in the command string. With this option the editor stops with each match to *string1* and waits for you to respond. The display shows the following information:

- The number of the line containing the matching string.
- The number of the column in which the first letter of the matching string occurs.
- A backslash (\) delimiter.
- Some of the line, beginning with the matching string.
- A slash (/) delimiter.
- A question mark (?) indicating that a response is expected.

Responding to a Search command, your options are:

- Press [Y] to stop the search at this match and make this line the current line.
- Press [N] to search for the next occurrence of the string.
- Press [Q] to quit the search and return to the previous current line.

Responding to a Replace command, your options are:

- Press [Y] to replace this occurence of *string1* with *string2* and search for the next occurrence of *string1*.
- Press [N] to leave this occurence of *string1* intact and search for the next occurrence of *string1*.
- Press [Q] to quit the replacement search and make the last line where replacement occured the current line (or return to the previous current line if no replacements occurred).

If you press any other key (except [ATTN]), the display will show Y/N/Q ? to indicate that only Y, N or Q are permitted as responses. If you press [ATTN], the Cmd: prompt returns.

The Replace command can result in lines longer than 96 characters. If this occurs while you're using the ? option, you can scroll through only a 96-character substring that contains that search string, not through the whole line.

Defining Patterns in Strings. Five characters (., @, &, ^, and ‡) can have special meanings when you're defining strings. To switch these characters to their special meanings, place a backslash (\, assigned to [f][/]) in the string; to return these characters to their normal meanings, place a second backslash in the string. (The string's final delimiter also returns the characters to their normal meanings.) Any of these five characters appearing between the two backslashes will be given their special meaning.

The five characters, their special meanings and some examples of their uses are described in the following paragraphs:

- The period (.) represents any character, and so is called a *wild-card* character. When the editor searches for a matching string, any character can be in those positions where you put a period.

  Example. R/ABC\...\/Recheck ID#/ will replace the occurrences of ABC followed by any three characters, such as ABC999, ABCzyz, or ABC yz, with the string Recheck ID#. R/ABC\.../Recheck ID# has the same effect; the second backslash is not needed because the end of *string1* stops the special-meaning feature, and the ending slash is optional for *string2*.

- The commercial "at" symbol (@) represents any number of wild-card characters. Because the program starts searching for the end of the string at the end of the line, the longest match possible is found.

  Example. `R/ABC\@\CDE/Recheck ID#/` will replace any string that begins with ABC and ends with CDE, such as `ABC123CDE`, `ABCCDE`, or `ABC12 zzzCDE`, with the string `Recheck ID#`.

- The ampersand (&) represents the text that matches *string1*; it is used in a Replace command to insert the actual string that matched *string1* (which may include wild cards) into *string2*.

  Example. `R/\AB./\&DEF/` searches for the string AB*wildcard* and appends the string DEF to it. If ABC is found, the new string will be ABCDEF.

- The up-arrow (^) represents the beginning of a line. As the first character in a string, it specifies that a matching string must be at the beginning of a line. If the up-arrow isn't the first character in the string, it has its normal meaning.

  Example. `R/\^\ABC/CDE/` will search for the string ABC only at the beginning of a line. If ABC appears anywhere else in the line, a match will not be made.

  Example. Suppose you have loaded a text file from the HP-75 into your HP-71. Now you want to delete the four-digit line numbers that the HP-75 put at the beginning of every line. `1#R/\^..../ /` tells your HP-71 to search, from line 1 to the end of the file, for any four characters at the beginning of the line, and replace them with nothing (delete them).

- The dollar sign ($) represents the end of a line. As the last character in a string, it specifies that a matching string must be at the end a line. If the dollar sign isn't the last character in the string, it has its normal meaning.

  Example. `R/ABC\$/CDE` will search for the string ABC only at the *end* of a line. If ABC appears anywhere else in the line, it will be ignored. A second backslash is not needed after the $ because the dollar sign is at the end of *string1*.

If you need to search for a string containing a backslash character as part of the text, you don't want Search and Replace to see the backslash as a switch. The solution is to use two sequential backslashes. The editor will interpret `\\` as a single backslash character, not as a switch.

Editor Files

The editor uses several files in its operation. The names of these files must not be used as the names of files in the HP-71 user memory, because the HP-71 first searches its own memory before searching the plug-in modules. The following list gives the name of each file in the module, along with a brief description of the file.

`EDTEXT`	The editor BASIC language program.
`EDLEX`	A LEX file containing the assembly level support for the editor, including the BASIC keywords.
`EDKEYS`	The editor keys file.
`EDUKEYS`	A temporary keys file created by the editor in main memory to store your user defined keys while the editor is running. When you exit the editor, these keys again become current.

Section 4

The Assembler

The FORTH/Assembler ROM contains an assembler that enables you to write assembly language extensions to the FORTH system or to the BASIC operating system. The assembler provides access to the complete HP-71 CPU instruction set through source code mnemonics that are nearly identical to those of the assembler used to produce the HP-71 BASIC operating system, as listed in the *HP-71 IDS*.

The assembler is invoked from the FORTH environment by the word ASSEMBLE, which is preceded by a string specifying the name of the assembler source file. The source file is an HP-71 text file, which you can create using the editor described in section 3. The output of the assembler can be either new FORTH words, which are placed directly into the FORTH dictionary, or HP-71 language extension (LEX) or binary (BIN) files, which are loaded automatically into the HP-71 file chain. The type of assembler output is specified by pseudo-ops included in the source file. The assembler can also produce an optional assembly listing, which is directed to an HP-71 file or to a listing device on HP-IL.

This section gives the rules for using the assembler, describes the HP-71 CPU, shows some sample source files for the three types of assembly, and finally describes the assembler's mnemonics and pseudo-ops.

Using the Assembler

Running the Assembler

The assembler is run while in the FORTH system by typing:

" *source-file specifier*" ASSEMBLE

The source-file specifier can include a mass storage device specifier. You can't run the assembler from BASIC (using FORTHX) because the assembler uses BASICX.

There is no intermediate link operation. The assembler acts as a loader, creating absolute modules that are ready to execute. New FORTH primitives go directly into the FORTHRAM dictionary. LEX and BIN files go directly into the file chain in RAM.

While the assembler is running, the display will show PASS 1 .... or PASS 2 .... to indicate the assembly's progress. A dot . is added to the display as each source line is processed. If you press [ATTN] while the assembler is active, the assembler will halt and prompt you with the message ABORT [Y/N] ? If you now press [Y], the assembly will terminate, and the message assembler aborted will be displayed. If you press any other key, the assembly will resume.

The Listing File

There are two variables in the FORTH system that control the listing file. The first variable, LISTING, is a string variable containing the listing-file specifier. To set this variable, type:

`" listing-file specifier" LISTING S!`

The listing-file specifier can be the name of an HP-71 text file, the HP-IL device specifier of a printer or display device, or the null string. If you specify a file name, the listing will be output to a RAM file, which you can list or edit using the editor. If you specify an HP-IL device, the listing will be output to that device as the assembly proceeds. If you specify the null string, no listing is created.

The second variable, PAGESIZE, is a numeric variable containing the number of lines per page in the listing file. That is, if PAGESIZE contains the value *n*, a form feed (character code 12) will be sent to the listing file or device after every *n* lines. The default value of PAGESIZE is 56.

Assembler Source Code

The text file containing source code for the assembler, which you create with the editor, must have the following form:

```
Output Pseudo-op
       ⋮
      Code
       ⋮
      END
```

The output pseudo-op must be `FORTH`, `LEX`, or `BIN`, to determine whether the assembler output will be FORTH primitives, a LEX file, or a BIN file. The pseudo-op `END` indicates the end of the source code.

The code portion of the file consists of any number of text lines, each containing one or more of the following items: label, mnemonic, modifier, pseudo-op, expression, comment. These items and the general line format are discussed below.

Line Format

The following template is the recommended column alignment for items in a source-file line. However, the assembler is "free format," requiring only a space to delimit the fields. The maximum length for a label is 6 characters (extra characters are ignored); for a mnemonic, 6 characters; and for a modifier, 50 characters. To distinguish mnemonics generated by pseudo-ops from your source mnemonics, an assembly listing will indent the former to column 3.

label	*mnemonic*	*modifier*	*comments*	
↑ 1	↑ 8	↑ 15	↑ 24	↑ 80

Comments

Text that follows a complete instruction—a mnemonic and any required modifiers—is a comment. If the first non-blank character in a line is a star (*), the entire line is a comment. All other text is considered part of an instruction.

Labels

Labels can contain up to six characters. All alphanumeric characters are allowed, as are all special characters except commas, spaces, and right parentheses. The first character cannot be sharp (#), single quote ('), minus sign (−), left parenthesis, star (*), or the digits 0 through 9. Leading equal signs (=) are ignored, so that =FRED and FRED are the same label. There is no case folding. A label must begin in column 1 or 2; otherwise it will be interpreted as a mnemonic. The restricted label FiLeNd is automatically generated after the last line of source in LEX and BIN files; if you enter this label in your source file, the assembly aborts.

Expressions

Expressions can contain labels, the location-counter value, constants, and operators. Any expression enclosed in parentheses can be nested within a larger expression, with up to three levels of nesting.

Labels. Legal label names are described above. When a label is used within a larger expression, parentheses are required to delineate it: AD1-10 is a label but (AD1)-10 is an expression.

Location-Counter Value. A star (*) in an expression represents the value of the location counter at the beginning of the current instruction.

Constants. The numeric value of a constant can be expressed in decimal, hexadecimal, or ASCII. Some instructions require a constant of a particular type; those instructions are listed under the required type of constant.

- Decimal constants can't exceed 1,048,575. Example: 23434.
- Hexadecimal constants must be preceded by the sharp (#) character and can't exceed FFFFF. Example: #1FF0. Hexadecimal constants are required with LCHEX and NIBHEX. (Leading # is optional when hexadecimal constant is required.)
- ASCII constants must be enclosed within single quotes and can contain one or two characters, . Example: 'AB' (equals 4142_{16}). ASCII constants are required with LCASC and NIBASC.

Operators. There are seven operators, listed below in descending order of precedence. Operators on the same level of precedence are executed left to right in the expression.

− (Unary minus)

& (Logical AND) ! (Logical OR).

* (Multiplication) / (Integer division)

+ (Addition) − (Subtraction)

Overview of the CPU

The HP-71 CPU is a proprietary CPU optimized for high-accuracy BCD math and low power consumption. The data path is four bits wide. Memory is accessed in four-bit quantities, called "nibbles" or "nibs." Addresses are 20 bits, yielding a physical address space of 512K bytes or 1M nibbles.

There are two types of registers on the CPU: arithmetic registers, used for data transfers and arithmetic operations; and control registers, used for program and system control.

Arithmetic Registers

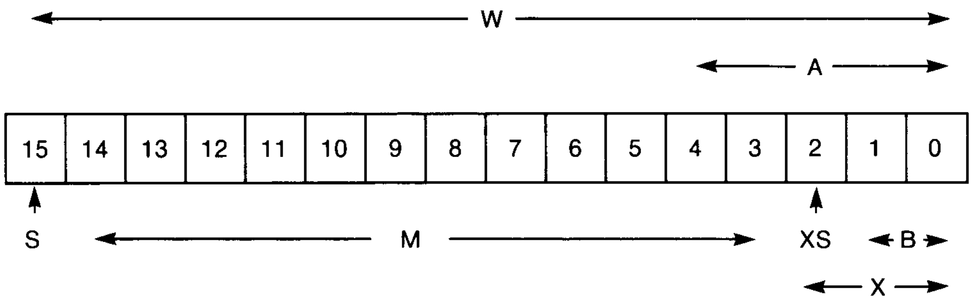
The arithmetic registers comprise the carry flag, the working registers A, B, C, and D, and the scratch registers R0, R1, R2, R3, and R4.

Arithmetic Registers

Name	Description	Size (bits)
Carry	Carry flag, adjusted by calculations and tests. During a calculation the carry flag is set if the calculation overflows or borrows; otherwise the carry flag is cleared. During a test the carry flag is set if the test is true; otherwise the carry flag is cleared.	1
A	Working register, used for shifts, tests, and arithmetic. Also used for memory access and for exchange with scratch registers and data-pointer registers.	64
B	Working register, used for shifts, tests, and arithmetic.	64
C	Most powerful working register, used for shifts, tests, and arithmetic. Also used for memory access, bus access, loading constants, and exchange with scratch registers, data-pointer registers, the pointer register, the hardware return stack, and status bits.	64
D	Least powerful working register, used for shifts, tests, and arithmetic.	64
R0	Scratch register. Used for exchange with A or C register.	64
R1	Scratch register. Used for exchange with A or C register.	64
R2	Scratch register. Used for exchange with A or C register.	64
R3	Scratch register. Used for exchange with A or C register.	64
R4	Scratch register. Used for exchange with A or C register. However, the HP-71 interrupt system uses the five low-order nibbles, effectively making the entire A field unavailable.	64

Subfields of the working registers may be manipulated by field selection. The possible field selections range from the entire register to any single nibble of the register. Certain subfields are designed for use in BCD calculations; others are used for data access or general data manipulation. The following diagram shows the seven fixed fields within a 16-nibble working register.

Fixed Fields within a Working Register



There is a one-nibble CPU pointer (the P register, described under "Control Registers") that can indicate any nibble in a working register. This allows two variable fields to be defined: the indicated nibble alone, or that nibble along with all lower nibbles (to the right). This makes a total of nine fields, listed below.

Fields within a Working Register

Name	Nibbles	Description
B	1-0	Exponent or byte.
X	2-0	Exponent and sign.
XS	2	Exponent sign.
A	4-0	Address.
W	15-0	Full word.
M	14-3	Mantissa.
S	15	Sign.
P	P	At pointer.
WP	P-0	Word through pointer.

Control Registers

The following table describes the CPU's control registers. The two data-pointer registers, D0 and D1, contain pointers to memory used for all memory access.

Control Registers

Name	Description	Size (bits)
PC	Program counter.	20
RSTK	Eight-level subroutine-return stack.	20
ST	Program-status flags.	16
SB	Sticky bit.	1
SR	Service Request bit.	1
MP	Module Pulled bit.	1
XM	External Module Missing bit.	1
P	Pointer register. Points to a nibble in the working registers. Used with field selection and Load Constant mnemonics.	4
D0	Data-pointer register. Used with register A or C during memory access.	20
D1	Data-pointer register. Used with register A or C during memory access.	20
OUT	Keyscan/write-only output register. Used by system; other uses limited.	12
IN	Keyscan/read-only input register. Used by system; other uses limited.	16

Subroutine Return Stack. Return addresses are stored on an eight-level LIFO hardware stack. Subroutine call and return instructions automatically push and pop addresses on this stack. If a ninth address is pushed onto the stack, the oldest address will be lost and will be replaced by zero when it is eventually popped from the stack. Because the memory-reset code of the operating system resides at address 00000, excessive nesting of subroutine calls will cause a memory reset.

Note: Because interrupt processing requires one level of the hardware return stack, code that executes *with interrupts enabled* must not use more than *seven* levels of return addresses on that stack. Otherwise, an interrupt may eventually result in a memory reset.

Loading Data from Memory

When memory is read into a register, the CPU places the lowest-addressed nibble in the lowest-order nibble of the register. The nibbles in a CPU register are numbered right-to-left, from least significant to most significant. For example, if the data in memory shown below is read into a CPU register, the data in the register will be arranged as shown.

Address	Contents
1000	5
1001	6
1002	7
1003	8

CPU Register

		8	7	6	5
15	...	3	2	1	0

When data is written to memory from a register, the CPU places the least significant nibble of the register in the lowest nibble of the addressed memory location. For example, if the data in the register shown above is written to memory, the data in memory will be arranged as shown.

Types of Assembly

To indicate whether to assemble a FORTH primitive, a LEX file, or a BIN file, the first line of the source file must contain a `FORTH`, `LEX`, or `BIN` pseudo-op. The sample files below illustrate each type of assembly.

FORTH Primitives

FORTH primitives must maintain three FORTH-system pointers. These pointers are the instruction pointer (different from the CPU hardware program counter), the data-stack pointer, and the return-stack pointer. They are maintained in the following CPU registers.

FORTH-System Pointers in CPU Registers

CPU Register	FORTH-System Pointer
D0	Instruction Pointer
D1	Data-Stack Pointer
A field in B	Return-Stack Pointer

Because the FORTH return stack is a software stack, it isn't limited to the seven levels of the CPU hardware stack .

In FORTH, stacks grow down in memory. Therefore, to push an item onto the data stack, you should decrement the data-stack pointer by 5 before storing the item on the stack:

```
 *             Stack push:
 *
D1=D1- 5       Decrement stack pointer.
DAT1=C A       Store item from A field of the C register onto the
 *              data stack.
 *
 *
 *             Stack pop:
 *
C=DAT1 A       Read top item from data stack into A field of C
 *              register.
D1=D1+ 5       Increment stack pointer.
```

End all FORTH primitives with RTNCC (return and clear the carry flag).

Sample FORTH Assembly.

```
       FORTH            Declare an assembly of FORTH primitives.
       WORD    '+'      Create a link field, name field, and code field for
        *                a primitive called "+."
       A=DAT1  A        Copy into the A field of register A the contents of
        *                memory pointed to by D1. This copies the first
        *                parameter on the FORTH data stack.
       D1=D1+  5        Increment the D1 data pointer. This increments the
        *                FORTH data-stack pointer.
       C=DAT1  A        Copy the second parameter on the FORTH data stack
        *                into the A field of register C.
       A=A+C   A        Add the two parameters.
       DAT1=A  A        Copy the result to the stack.
       RTNCC            Return to inner loop.
DSPCNT EQU     #2E3FE   Define a label for the system location that controls
        *                the display contrast. A nibble 0 gives minimum
        *                contrast and a nibble 15 gives maximum contrast, as
        *                with the BASIC command CONTRAST.
       WORDI   'DISP'   Create a link field, name field, and code field for
        *                an immediate primitive called "DISP."
       A=DAT1  A        Pop the first parameter into the A field of
       D1=D1+  5         register A.
       P=      0        Set the pointer register to 0 for subsequent Load
        *                Constant instruction.
       LC(5)   DSPCNT   Load the A field (low-order five nibbles) of
        *                 register C with the system location DSPCNT.
       CD0EX            Exchange the A field of register C with the data
        *                pointer D0.
       DAT0=A  1        Copy one nibble of register A to memory pointed to
        *                by D0.
       CD0EX            Exchange the A field of register C and D0 back to
        *                original values.
       RTNCC            Return to the inner loop.
       END              Mark the end of the source file (optional).
```

LEX Files

Although LEX files usually define new BASIC keywords, they can also answer system polls or define message tables. After assembling a LEX file you must turn the HP-71 off and then on again. This registers the LEX file in the system's LEX entry buffer for keyword checking and poll handling. For a full description of LEX files and their uses, refer to the *HP-71 IDS*.

Two sample LEX files appear below. The first is a poll handler, the second defines a keyword. Note that both files begin with the pseudo-ops `LEX`, `ID`, `MSG`, and `POLL`; this sequence is required for all LEX files.

Sample Poll-Handler LEX File. The following LEX file will intercept the configuration poll and save the general purpose buffer whose ID is #E01.

```
        LEX     'POLL'   Declare an assembly of a LEX file named "POLL."
        ID      #5C      This LEX file has an ID of 5C.
        MSG     0        There is no message table in this LEX file.
        POLL    POLHND   Our poll handler begins at the label POLHND.
        ENDTXT           Mark the end of the BASIC keyword tables. In this
         *                case there are no tables, but the ENDTXT pseudo-op
         *                is still required.
BUFNUM  EQU     #E01     Define a label for the ID# of the buffer to save.
I/ORES  EQU     #118FF   Define a label for the entry point of a system
         *                routine. This system routine will prevent the
         *                system from reclaiming the buffer indicated in the
         *                X field of register C.
pCONFG  EQU     #FB      Define a label for the configure poll.
POLHND                   Define a label for the start of the poll-answering
         *                routine.
        SETHEX           Set the arithmetic mode to hexadecimal.
        P=      0        Set the pointer register to 0 for subsequent Load
         *                Constant instructions.
        LC(2)   pCONFG   Load B field (low-order two nibbles) of register C
         *                with the poll we want to handle.
        ?B=C    B        Test whether the current poll is the configure poll.
        GOYES   CONFIG   If so, branch to our routine (carry flag is set).
        RTNSXM           If not, exit (carry flag is clear).
CONFIG                   Define a label for the start of our routine to
         *                answer the configure poll.
        LC(3)   BUFNUM   Load X field (low-order three nibbles) of register C
         *                with the ID# of the buffer to save.
        GOSBVL  I/ORES   Call the system routine to prevent system from
         *                reclaiming this buffer. The routine clears the
         *                carry flag.
        RTNSXM           Exit and set External Module Missing bit.
        END              Mark the end of the source file (optional).
```

Sample Keyword LEX File. The following LEX file defines a BASIC function ONE that returns the number 1.

```
        LEX     'KEYWORD' Declare an assembly of a LEX file named "KEYWORD."
        ID      #5D       This LEX file has an ID of 5D.
        MSG     0         There is no message table in this LEX file.
        POLL    0         There is no poll handler in this LEX file.
FNRTN1  EQU     #0F216    Define a label for the entry point of a system
        *                  routine. This system routine returns a numeric
        *                  parameter to the math stack.
        ENTRY   FNCT      This keyword is coded at the label FNCT.
        CHAR    #F        This keyword is a BASIC function, indicated by a
        *                  characterization nibble of F.
        KEY     'ONE'     This keyword is called "ONE."
        TOKEN   1         This keyword has token 1. The LEX ID# and token
        *                  uniquely define each BASIC keyword.
        ENDTXT            Mark the end of the BASIC keyword tables.
        NIBHEX  00        The minimum and maximum number of parameters for
        *                  this function is zero.
FNCT                      Define a label for the start of the code for the
        *                  keyword ONE.
        *                 Put a floating-point "1" into register C:
        C=0     W         Clear all digits in register C.
        P=      14        Set the pointer register to the most-significant
        *                  digit in the mantissa.
        LCHEX   1         Load the most-significant digit in register C's
        *                  mantissa field with a hex 1.
        GOVLNG  FNRTN1    Send the result back to the system.
        END               Mark the end of the source file (optional).
```

Note the pseudo-ops ENTRY, CHAR, KEY, and TOKEN; these are required for each keyword in a LEX file. When there are multiple keywords in an assembly, the ENTRY and CHAR pseudo-ops for the first keyword come first, followed by the ENTRY and CHAR pseudo-ops for the second keyword, and so on. After the ENTRY and CHAR pseudo-ops for the final keyword come the KEY and TOKEN pseudo-ops for the first keyword, followed by the KEY and TOKEN pseudo-ops for the second keyword, and so on.

Binary Files

Binary files are program files coded in assembly language. They can be executed like BASIC programs by using RUN, CHAIN, or CALL. They usually run faster than comparable BASIC programs and, unlike BASIC programs, can refer to system entry points.

Sample Binary Program. This binary program displays HELLO.

```
        BIN     'HELLO'  Declare an assembly of a binary file called "HELLO."
        CHAIN   -1       There are no subprograms in this file. Binary
         *                subprograms are described in the HP-71 IDS.
BF2DSP  EQU     #01C0E   Define a label for the entry point of a system
         *                routine. The system routine displays the string in
         *                memory that starts at DAT1 and ends with a
         *                character #FF.
ENDBIN  EQU     #0764B   Define a label for the entry point of the system
         *                routine that ends a binary program.
         *               The code immediately follows the pseudo-ops.
        GOSUB   POP      This line, combined with C=RSTK (labeled POP), puts
         *                the address of the following string into the A
         *                field of register C.
        NIBASC  'HELLO'  The string HELLO.
        NIBHEX  D0A0FF   Carriage return, line feed, end-of-string mark.
POP     C=RSTK           Pop the return address (which is the address of the
         *                preceding string) into the A field of register C.
        D1=C             Copy the string's address to D1.
        GOSBVL  BF2DSP   Call the system routine to display the string
         *                pointed to by D1.
        GOVLNG  ENDBIN   The correct way to exit a binary program.
        END              Mark the end of the source file (optional).
```

Assembler Mnemonics

The assembler mnemonics are listed below in condensed form, grouped by function. A list of all mnemonics (listed in ASCII order) with their opcodes and cycle times appears in the *HP-71 Software IDS*.

Branching Mnemonics

GOTO Mnemonics. In the following mnemonics,

- *offset* is the distance in nibbles to the specified label.

GOTO *label*	Short goto ($-2047 \leqslant$ *offset* $\leqslant 2048$).
GOC *label*	Short goto if carry ($-127 \leqslant$ *offset* $\leqslant 128$).
GONC *label*	Short goto if no carry ($-127 \leqslant$ *offset* $\leqslant 128$).
GOLONG *label*	Long goto ($-32766 \leqslant$ *offset* $\leqslant 32769$).
GOVLNG *label*	Very long goto (to absolute address).
GOYES *label*	Short goto if test true ($-128 \leqslant$ *offset* $\leqslant 127$). (Used only with test mnemonics.)

GOSUB Mnemonics. In the following mnemonics,

- *offset* is the distance in nibbles to the specified label.

GOSUB *label*	Short gosub ($-2044 \leqslant$ *offset* $\leqslant 2051$).
GOSUBL *label*	Long gosub ($-32762 \leqslant$ *offset* $\leqslant 32773$).
GOSBVL *label*	Very long gosub (to absolute address).

Return Mnemonics.

RTN	Return.
RTNSC	Return and set carry.
RTNCC	Return and clear carry.
RTNSXM	Return and set External Module Missing bit.
RTI	Return from interrupt (enable interrupts).
RTNC	Return if carry set.
RTNNC	Return if no carry set.
RTNYES	Return if test true. (Used only with test mnemonics.)

Test Mnemonics

Each test mnemonic must be followed with a GOYES or RTNYES mnemonic. The test mnemonic and the GOYES or RTNYES mnemonic combine to generate a single opcode. Each test will set the carry flag if true, or clear the carry flag if false.

Register Tests. In the following mnemonics,

- (r, s) = (A, B), (A, C), (B, A), (B, C), (C, A), (C, B), (C, D), or (D, C).
- fs = A, P, WP, XS, X, S, M, B, or W.

?*r*=*s* *fs*	Is *fs* field of *r* equal to *fs* field of *s*?
?*r*#*s* *fs*	Is *fs* field of *r* not equal to *fs* field of *s*?
?*r*=0 *fs*	Is *fs* field of *r* equal to zero?
?*r*#0 *fs*	Is *fs* field of *r* not equal to zero?
?*r*>*s* *fs*	Is *fs* field of *r* greater than *fs* field of *s*?
?*r*<*s* *fs*	Is *fs* field of *r* less than *fs* field of *s*?
?*r*>=*s* *fs*	Is *fs* field of *r* greater than or equal to *fs* field of *s*?
?*r*<=*s* *fs*	Is *fs* field of *r* less than or equal to *fs* field of *s*?

Pointer Tests. In the following mnemonics,

- *n* is an expression whose hex value is from 0 through F.

`?P= n`	Is P register equal to *n*?
`?P# n`	Is P register not equal to *n*?

Program-Status Tests. In the following mnemonics,

- *n* is an expression whose hex value is from 0 through F.

`?ST=0 n`	Is bit *n* in ST equal to 0?
`?ST=1 n`	Is bit *n* in ST equal to 1?
`?ST#0 n`	Is bit *n* in ST not equal to 0?
`?ST#1 n`	Is bit *n* in ST not equal to 1?

Hardware-Status Tests.

`?XM=0`	Is the External Module Missing bit clear?
`?SB=0`	Is the Sticky bit clear?
`?SR=0`	Is the Service Request bit clear?
`?MP=0`	Is the Module Pulled bit clear?

P Register Mnemonics

In the following mnemonics,

- *n* is an expression whose hex value is from 0 through F.

Note that the C register is the only working register used with the P register. During those operations that involve a calculation, the carry flag is set if the calculation overflows or borrows; otherwise the carry flag is cleared.

`P= n`	Set P register to *n*.
`P=P+1`	Increment P register.
`P=P-1`	Decrement P register.
`C+P+1`	Add P register plus one to A field in C. Arithmetic is hexadecimal.
`CPEX n`	Exchange P register with nibble *n* in C.
`P=C n`	Copy nibble *n* in C to P register.
`C=P n`	Copy P register to nibble *n* in C.

Status Mnemonics

In the following mnemonics,

- *n* is an expression whose hex value is from 0 through F.

`ST=0 n`	Set bit *n* in ST to 0.
`ST=1 n`	Set bit *n* in ST to 1.
`CSTEX`	Exchange X field in C and bits 0 through 11 in ST.
`C=ST`	Copy bits 0 through 11 in ST into X field in C.
`ST=C`	Copy X field in C into bits 0 through 11 in ST.
`CLRST`	Clear bits 0 through 11 in ST.
`SB=0`	Clear Sticky bit (SB).
`SR=0`	Clear Service Request (SR) bit.
`MP=0`	Clear Module Pulled (MP) bit.
`XM=0`	Clear External Module Missing (XM) bit.
`CLRHST`	Clear SB, SR, MP, and XM bits.

System-Control and Keyscan Mnemonics

The first four mnemonics below are useful for most programmers. The remaining mnemonics are used by the system and have limited general use; for details, refer to the *HP-71 IDS* and the *HP-71 Hardware Specification.*

`SETHEX`	Set arithmetic mode to hexadecimal.
`SETDEC`	Set arithmetic mode to decimal.
`C=RSTK`	Pop return stack into A field in C.
`RSTK=C`	Push A field in C onto return stack.
`CONFIG`	Configure.
`UNCNFG`	Unconfigure.
`RESET`	Send Reset command to system bus.
`BUSCC`	Send Bus Command C to system bus.
`SHUTDN`	Stop here.
`C=ID`	Request ID (A field in C).
`SREQ?`	Sets service request bit if service has has been requested. Nibble 0 in C shows what bit(s) are pulled high.
`INTOFF`	Disable interrupts (doesn't affect ON-key or module-pulled interrupts).
`INTON`	Enable interrupts.
`OUT=C`	Copy X field in C into OUT.
`OUT=CS`	Copy nibble 0 of C into OUT.
`A=IN`	Copy IN into nibbles 0 through 3 in A.
`C=IN`	Copy IN into nibbles 0 through 3 in C.

Scratch Register Mnemonics

In the following mnemonics,

- *r* = A or C.
- *ss* = R0, R1, R2, R3, or R4.

*rss*EX	Exchange *r* and *ss*.
r=*ss*	Copy *ss* into *r*.
ss=*r*	Copy *r* into *ss*.

Memory-Access Mnemonics

Data-Pointer Mnemonics. In the following mnemonics,

- *r* = A or C.
- *ss* = D0 or D1.
- *n* is an expression whose hex value is from 0 through F.
- *nnnnn* is an expression whose hex value is from 0 through FFFFF.

During those operations that involve a calculation, the carry flag is set if the calculation overflows or borrows; otherwise the carry flag is cleared.

*rss*EX	Exchange A field in *r* with *ss*.
*rss*XS	Exchange nibbles 0 through 3 in *r* with *ss*.
ss=*r*	Copy A field in *r* into *ss*.
ss=*r*S	Copy nibbles 0 through 3 in *r* into *ss*.
ss=*ss*+ *n*	Increment *ss* by *n*.
ss=*ss*- *n*	Decrement *ss* by *n*.
ss=(2) *nnnnn*	Load *ss* with two nibbles from *nnnnn*.
ss=(4) *nnnnn*	Load *ss* with four nibbles from *nnnnn*.
ss=(5) *nnnnn*	Load *ss* with *nnnnn*.

Data-Transfer Mnemonics. In the following mnemonics,

- *r* = A or C.
- *fs* = A, P, WP, XS, X, S, M, B, W (or a number *n* from 1 through 16).

r=DAT0 *fs*	Copy data at address contained in D0 into *fs* field in *r* (or into nibble 0 through nibble $n - 1$ in *r*).
r=DAT1 *fs*	Copy data at address contained in D1 into *fs* field in *r* (or into nibble 0 through nibble $n - 1$ in *r*).
DAT0=*r* *fs*	Copy data in *fs* field in *r* (or in nibble 0 through nibble $n - 1$ in *r*) to address contained in D0.
DAT1=*r* *fs*	Copy data in *fs* field in *r* (or in nibble 0 through nibble $n - 1$ in *r*) to address contained in D1.

Load-Constants Mnemonics

In the following mnemonics,

- *h* is a hex digit.
- *i* is an integer from 1 through 5.
- *nnnnn* is an expression with hex value from 0 through FFFFF.
- *c* is an ASCII character.

`LCHEX` *h . . . h*	Load up to 16 hex digits into C. The least significant digit is loaded at the pointer position; more significant digits are loaded into higher positions, wrapping around from nibble 15 to nibble 0.
`LC(`*i*`)` *nnnnn*	Load *i* hex digits from the value of *nnnnn* into C. The least significant digit is loaded at the pointer position; more significant digits are loaded into higher positions, wrapping around from nibble 15 to nibble 0.
`LCASC '`*c . . . c*`'`	Load up to eight ASCII characters into C. The least significant nibble of the low-order character is loaded at the pointer position; more significant nibbles are loaded into higher positions, wrapping around from nibble 15 to nibble 0. For example, `LCASC 'AB'` is equivalent to `LCHEX 4142`.

Shift Mnemonics

In the following mnemonics,

- *r* = A, B, C, or D.
- *fs* = A, P, WP, XS, X, S, M, B, or W.

Non-circular shift operations shift in zeros. If any shift-right operation, circular or non-circular, moves a non-zero nibble or bit from the right end of a register or field, the Sticky bit SB is set. The Sticky bit is cleared only by a `SB=0` or `CLRHST` instruction.

r`SRB`	Shift *r* right by one bit.
r`SLC`	Shift *r* left by one nibble (circular).
r`SRC`	Shift *r* right by one nibble (circular).
r`SL` *fs*	Shift *fs* field in *r* left by one nibble.
r`SR` *fs*	Shift *fs* field in r right by one nibble.

Logical Mnemonics

These mnemonics are summarized below, using the following variables:

- (*r*, *s*) = (A, B), (A, C), (B, A), (B, C), (C, A), (C, B), (C, D), or (D, C).
- *fs* = A, P, WP, XS, X, S, M, B, or W.

r`=`*r*`&`*s* *fs*	*fs* field in *r* AND *fs* field in *s* into *fs* field in *r*.
r`=`*r*`!`*s* *fs*	*fs* field in *r* OR *fs* field in *s* into *fs* field in *r*.

Arithmetic Mnemonics

Arithmetic results depend on the current arithmetic mode. In hexadecimal mode (set by `SETHEX`), nibble values range from 0 through F. In decimal mode (set by `SETDEC`), nibble values range from 0 through 9, and arithmetic is BCD arithmetic.

There are two groups of arithmetic mnemonics. In the first group (general), almost all combinations of the four working registers are possible; in the second group (restricted), only a few combinations are possible. During those operations that involve a calculation, the carry flag is set if the calculation overflows or borrows; otherwise the carry flag is cleared.

General Arithmetic Mnemonics. In the following mnemonics,

- (*r*, *s*) = (A, B), (A, C), (B, A), (B, C), (C, A), (C, B), (C, D), or (D, C).
- *fs* = A, P, WP, XS, X, S, M, B, or W.

r=0 *fs*	Set *fs* field in *r* to zero.
r=*r*+*r* *fs*	Double *fs* field in *r* (shift left by one bit).
r=*r*+1 *fs*	Increment *fs* field in *r* by 1.
r=*r*-1 *fs*	Decrement *fs* field in *r* by 1.
r=-*r* *fs*	Tens complement or twos complement, depending on arithmetic mode, of *fs* field in *r*. Clears Carry if *argument* =0; sets Carry otherwise.
r=-*r*-1 *fs*	Nines complement or ones complement, depending on arithmetic mode, of *fs* field in *r*. Clears Carry.
r=*r*+*s* *fs*	Sum *fs* field in *r* and *fs* field in *s* into *fs* field in *r*.
s=*r*+*s* *fs*	Sum *fs* field in *r* and *fs* field in *s* into *fs* field in *s*.
r=*s* *fs*	Copy *fs* field in *s* into *fs* field in *r*.
s=*r* *fs*	Copy *fs* field in *r* into *fs* field in *s*.
*rs*EX *fs*	Exchange *fs* field in *r* and *fs* field in *s*.

Restricted Arithmetic Mnemonics. In the following mnemonics,

- (*r*, *s*) = (A, B), (B, C), (C, A), or (D, C).
- *fs* = A, P, WP, XS, X, S, M, B, or W.

r=*r*-*s* *fs*	Difference of *fs* field in *r* and *fs* field in *s* into *fs* field in *r*.
r=*s*-*r* *fs*	Difference of *fs* field in *s* and *fs* field in *r* into *fs* field in *r*.
s=*s*-*r* *fs*	Difference of *fs* field in *s* and *fs* field in *r* into *fs* field in *s*.

No-op Mnemonics

`NOP3`	Three-nibble no-op.
`NOP4`	Four-nibble no-op.
`NOP5`	Five-nibble no-op.

Pseudo-ops

Control Pseudo-ops

`EJECT` — Generate a form feed in the assembly listing.

`END` — Mark the end of the assembly source file. Any characters in the file following `END` are ignored by the assembler. This pseudo-op is optional.

label `EQU` *expression* — Define *label* to have the value of *expression*. All references to *label* will have this value; *label* can't be redefined in a later part of the program.

`LIST ON/LIST OFF` — Send/suppress output to the listing file. (Limited RAM may require a shortened listing file.)

`STITLE` *subtitle* — Force a new page and put *subtitle* at the top of each page of the listing file, just underneath the title. The text for *subtitle* can contain up to 40 characters.

`TITLE` *title* — Put *title* at the beginning of each page of the listing file. The text for *title* can contain up to 40 characters.

Constant-Generating Pseudo-ops

`BSS` *expression* — Evaluate *expression* and generate that number of zero nibbles.

`CON(`*i*`)` *expression* — Evaluate *expression* and generate an absolute constant of length i nibbles, $1 \leqslant i \leqslant 5$.

`NIBASC` '*chars*' — Generate the specified ASCII characters, with the two nibbles within each byte reversed. The modifier field may specify up to eight characters. (The result is the same if each character is placed in its own `NIBASC` pseudo-op.)

`NIBHEX` *h . . . h* — Generate up to sixteen hexadecimal nibbles.

`REL(`*i*`)` *expression* — Evaluate *expression* and generate a constant (relative to the current location-counter value) of length i nibbles, $1 \leqslant i \leqslant 5$.

Macro-Expansion Pseudo-ops for FORTH Words

`FORTH` — Assemble a new FORTH primitive. This pseudo-op must be the first line in the file.

`WORD` '*name*' — Define a FORTH primitive called *name*. The assembly code that defines *name* should directly follow the `WORD` pseudo-op.

`WORDI` '*name*' — Define an immediate FORTH primitive called *name*. The assembly code that defines *name* should directly follow the `WORDI` pseudo-op.

Macro-Expansion Pseudo-ops for LEX Files

`LEX` *'name'*
: Assemble a new LEX file called *name*. This pseudo-op must be the first line in the source file. The LEX file will have the correct header. The intial data for this file is defined by the `ID`, `MSG`, and `POLL` pseudo-ops, which must be present in that order.

`ID` *byte*
: Define the LEX ID of this LEX file. The byte is placed in the appropriate data field. This pseudo-op is required when the `LEX` pseudo-op is used.

`MSG` *label*
: Define the beginning of this LEX file's message table. `MSG` will place *label* in the appropriate field. This pseudo-op is required when the `LEX` pseudo-op is used. If there is no message table, enter `MSG 0`.

`POLL` *label*
: Define the beginning of this LEX file's poll-handling routine. `POLL` will place *label* in the appropriate field. This pseudo-op is required when the `LEX` pseudo-op is used. If there is no poll-handling routine, enter `POLL 0`.

`ENTRY` *label*
: Begin the definition of a BASIC keyword. Each keyword requires four pseudo-ops: `ENTRY`, `CHAR`, `KEY`, and `TOKEN`.

Because of the structure of the LEX file's keyword tables, these pseudo-ops require a particular order. For example, the following assembly-language header defines two keywords, `KEY1` and `KEY2`.

`ENTRY` *label1*	The code for the first keyword begins at *label1*.
`CHAR 5`	The first keyword is legal for keyboard execution and after `THEN...ELSE`.
`ENTRY` *label2*	The code for the second keyword begins at *label2*.
`CHAR #F`	The second keyword is a function.
`KEY 'KEY1'`	The first keyword is invoked with "KEY1" in BASIC.
`TOKEN 1`	The first keyword has token 1.
`KEY 'KEY2'`	The second keyword is invoked with "KEY2" in BASIC.
`TOKEN 2`	The second keyword has token 2.
`ENDTXT`	Mark the end of the keyword tables.

`CHAR` *h*
: Describe the type of BASIC keyword. Each `ENTRY` requires a corresponding `CHAR`, which places a "characterization nibble" in the keyword tables. The characterization nibble defines BASIC keywords as follows.

Values for the Characterization Nibble

Value	Type of keyword
1	Keyboard execution.
4	Legal after THEN . . . ELSE.
8	Begin BASIC (legal as first keyword in a statement).
15	Function.

Other values for the characterization nibble define combinations of the above types. For example, a characterization nibble of 5 defines a keyword that is legal for keyboard execution and after THEN . . . ELSE. For details about the characterization nibble, refer to the *HP-71 IDS*.

KEY *'name'*
Define the name that will evoke the keyword in BASIC. When there are multiple keywords in one LEX file, the names of the keywords must be in alphabetic order. There is one exception: the name 'abc' is not before the name 'abcd'. If the first characters are the same, the *longer* text must come first. Otherwise, the BASIC operating system will never find the longer keyword.

TOKEN *number*
Define the token number of the keyword most recently named (by KEY). When there are multiple keywords in one LEX file, their token numbers must be in ascending order. TOKEN places the token number in the keyword tables.

ENDTXT
Mark the end of the keyword tables. This pseudo-op follows the ENTRY, CHAR, KEY, and TOKEN pseudo-ops when a keyword is defined, or it marks their absence if no keyword is defined.

Macro-Expansion Pseudo-ops for BIN Files

BIN *'name'*
Assemble a BIN file called *name*. This pseudo-op must be the first line in the source file. BIN creates the file header; the user must create the subheader using the CHAIN pseudo-op.

CHAIN *'label'*
Create a 12-nibble subheader containing a subprogram and label chains. If there are no subprograms, enter CHAIN -1.

Appendix A

Care, Warranty, and Service Information

Care of the Module

The HP-71 FORTH/Assembler ROM does not require maintenance. However, there are several precautions, listed below, that you should observe.

CAUTIONS

- Do not place fingers, tools, or other objects into the plug-in ports. Damage to plug-in module contacts and the computer's internal circuitry may result.
- Turn of the computer (press [f][OFF]) before installing or removing a plug-in module.
- If a module jams when inserted into a port, it may be upside down. Attempting to force it further may result in damage to the computer or the module.
- Handle the plug-in modules very carefully while they are out of the computer. Do not insert any objects in the module connector socket. Always keep a blank module in the computer port when a module is not installed. Failure to observe these cautions may result in damage to the module or the computer.

Limited One-Year Warranty

What We Will Do

The HP 82441A FORTH/Assembler ROM is warranted by Hewlett-Packard against defects in materials and workmanship affecting electronic and mechanical performance, but not software content, for one year from the date of original purchase. If you sell your unit or give it as a gift, the warranty is transferred to the new owner and remains in effect for the original one-year period. During the warranty period, we will repair or, at our option, replace at no charge a product that proves to be defective, provided you return the product, shipping prepaid, to a Hewlett-Packard service center.

What Is Not Covered

This warranty does not apply if the product has been damaged by accident or misuse or as the result of service or modification by other than an authorized Hewlett-Packard service center.

No other express warranty is given. The repair or replacement of a product is your exclusive remedy. **ANY OTHER IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS IS LIMITED TO THE ONE-YEAR DURATION OF THIS WRITTEN WARRANTY.** Some states, provinces, or countries do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. **IN NO EVENT SHALL HEWLETT-PACKARD COMPANY BE LIABLE FOR CONSEQUENTIAL DAMAGES.** Some states, provinces, or countries do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state, province to province, or country to country.

Warranty for Consumer Transactions in the United Kingdom

This warranty shall not apply to consumer transactions and shall not affect the statutory rights of a consumer. In relation to such transactions, the rights and obligations of Seller and Buyer shall be determined by statute.

Obligation to Make Changes

Products are sold on the basis of specifications applicable at the time of manufacture. Hewlett-Packard shall have no obligation to modify or update products once sold.

Warranty Information

If you have any questions concerning this warranty, please contact an authorized Hewlett-Packard dealer or a Hewlett-Packard sales and service office. Should you be unable to contact them, please contact:

- In the United States:

Hewlett-Packard
Personal Computer Group
Customer Support
11000 Wolfe Road
Cupertino, CA 95014

Toll-Free Number: (800) FOR-HPPC (800 367-4772)

- In Europe:

Hewlett-Packard S.A.
150, route du Nant-d'Avril
P.O. Box CH-1217 Meyrin 2
Geneva
Switzerland
Telephone: (022) 83 81 11

Note: Do *not* send units to this address for repair.

- In other countries:

Hewlett-Packard Intercontinental
3495 Deer Creek Rd.
Palo Alto, California 94304
U.S.A.
Telephone: (415) 857-1501

Note: Do not send units to this address for repair.

Service

Hewlett-Packard maintains service centers in most major countries throughout the world. You may have your unit repaired at a Hewlett-Packard service center any time it needs service, whether the unit is under warranty or not. There is a charge for repairs after the one-year warranty period.

Hewlett-Packard products are normally repaired and reshipped within five (5) working days of receipt at any service center. This is an average time and could vary depending upon the time of year and the work load at the service center. The total time you are without your unit will depend largely on the shipping time.

Obtaining Repair Service in the United States

The Hewlett-Packard United States Service Center for battery-powered computational products is located in Corvallis, Oregon:

Hewlett-Packard Company
Service Department

P.O. Box 999
Corvallis, Oregon 97339, U.S.A.
or
1030 N.E. Circle Blvd.
Corvallis, Oregon 97330, U.S.A.

Telephone: (503) 757-2000

Obtaining Repair Service in Europe

Service centers are maintained at the following locations. For countries not listed, contact the dealer where you purchased your unit.

AUSTRIA
HEWLETT-PACKARD Ges.m.b.H.
Kleinrechner-Service
Wagramerstrasse-Lieblgasse 1
A-1220 Wien (Vienna)
Telephone: (0222) 23 65 11

BELGIUM
HEWLETT-PACKARD BELGIUM SA/NV
Woluwedal 100
B-1200 Brussels
Telephone: (02) 762 32 00

DENMARK
HEWLETT-PACKARD A/S
Datavej 52
DK-3460 Birkerod (Copenhagen)
Telephone: (02) 81 66 40

EASTERN EUROPE
Refer to the address listed under Austria.

FINLAND
HEWLETT-PACKARD OY
Revontulentie 7
SF-02100 Espoo 10 (Helsinki)
Telephone: (90) 455 02 11

FRANCE
HEWLETT-PACKARD FRANCE
Division Informatique Personnelle
S.A.V. Calculateurs de Poche
F-91947 Les Ulis Cedex
Telephone: (6) 907 78 25

GERMANY
HEWLETT-PACKARD GmbH
Kleinrechner-Service
Vertriebszentrale
Berner Strasse 117
Postfach 560 140
D-6000 Frankfurt 56
Telephone: (611) 50041

ITALY
HEWLETT-PACKARD ITALIANA S.P.A.
Casella postale 3645 (Milano)
Via G. Di Vittorio, 9
I-20063 Cernusco Sul Naviglio (Milan)
Telephone: (2) 90 36 91

NETHERLANDS
HEWLETT-PACKARD NEDERLAND B.V.
Van Heuven Goedhartlaan 121
NL-1181 KK Amstelveen (Amsterdam)
P.O. Box 667
Telephone: (020) 472021

NORWAY
HEWLETT-PACKARD NORGE A/S
P.O. Box 34
Oesterndalen 18
N-1345 Oesteraas (Oslo)
Telephone: (2) 17 11 80

SPAIN
HEWLETT-PACKARD ESPANOLA S.A.
Calle Jerez 3
E-Madrid 16
Telephone: (1) 458 2600

SWEDEN
HEWLETT-PACKARD SVERIGE AB
Skalholtsgatan 9, Kista
Box 19
S-163 93 Spanga (Stockholm)
Telephone: (08) 750 2000

SWITZERLAND
HEWLETT-PACKARD (SCHWEIZ) AG
Kleinrechner-Service
Allmend 2
CH-8967 Widen
Telephone: (057) 31 21 11

UNITED KINGDOM
HEWLETT-PACKARD Ltd
King Street Lane
GB-Winnersh, Wokingham
Berkshire RG11 5AR
Telephone: (0734) 784 774

International Service Information

Not all Hewlett-Packard service centers offer service for all models of HP products. However, if you bought your product from an authorized Hewlett-Packard dealer, you can be sure that service is available in the country where you bought it.

If you happen to be outside of the country where you bought your unit, you can contact the local Hewlett-Packard service center to see if service is available for it. If service is unavailable, please ship the unit to the address listed above under Obtaining Repair Service in the United States. A list of service centers for other countries can be obtained by writing to that address.

All shipping, reimportation arrangements, and customs costs are your responsibility.

Service Repair Charge

There is a standard repair charge for out-of-warranty repairs. The repair charges include all labor and materials. In the United States, the full charge is subject to the customer's local sales tax. In European countries, the full charge is subject to Value Added Tax (VAT) and similar taxes wherever applicable. All such taxes will appear as separate items on invoiced amounts.

Computer products damaged by accident or misuse are not covered by the fixed repair charges. In these situations, repair charges will be individually determined based on time and materials.

Service Warranty

Any out-of-warranty repairs are warranted against defects in materials and workmanship for a period of 90 days from date of service.

Shipping Instructions

Should your unit require service, return it with the following items:

- A completed Service Card, including a description of the problem.
- A sales receipt or other proof of purchase date if the one-year warranty has not expired.

The product, the Service Card, a brief description of the problem, and (if required) the proof of purchase date should be packaged in adequate protective packaging to prevent in-transit damage. Such damage is not covered by the one-year limited warranty; Hewlett-Packard suggests that you insure the shipment to the service center. The packaged unit should be shipped to the nearest Hewlett-Packard designated collection point or service center. Contact your dealer for assistance. (If you are not in the country where you originally purchased the unit, refer to "International Service Information" above.)

Whether the unit is under warranty or not, it is your responsibility to pay shipping charges for delivery to the Hewlett-Packard service center.

After warranty repairs are completed, the service center returns the unit with postage prepaid. On out-of-warranty repairs in the United States and some other countries, the unit is returned C.O.D. (covering shipping costs and the service charge).

Further Information

Circuitry and designs are proprietary to Hewlett-Packard, and service manuals are not available to customers. Should other problems or questions arise regarding repairs, please call your nearest Hewlett-Packard service center.

When You Need Help

Hewlett-Packard is committed to providing after-sale support to its customers. To this end, our customer support department has established phone numbers that you can call if you have questions about this product.

Product Information. For information about Hewlett-Packard dealers, products, and prices, call the toll-free number below:

(800) FOR-HPPC
(800 367-4772)

Technical Assistance. For technical assistance with your product, call the number below:

(503) ~~754-6666~~
757-2004

For either product information or technical assistance, you can also write to:

Hewlett-Packard
Personal Computer Group
Customer Communications
11000 Wolfe Road
Cupertino, CA 95014

Appendix B

Error Messages

The error messages listed in the following tables relate only to FORTH/Assembler ROM operations. For other error or warning messages, refer to the *HP-71 Reference Manual.*

This appendix contains four listings:

1. An alphabetical listing of FORTH error messages with their corresponding error numbers. You can use the error's number to look up the error in the next listing.
2. A numerical listing of FORTH error messages with a description of each error condition.
3. An alphabetical listing of assembler messages with a description of each message.
4. An alphabetical listing of editor messages with a description of each message.

FORTH Messages

Alphabetical Listing of FORTH Messages

Message	Number
address not inside a file	47043
argument < 1	47006
attempted to redefine null	47011
bad parameters	47046
BASIC not re-entrant	47063
cannot load	47055
compile only	47009
conditionals not paired	47017
Configuration	47052
definition not finished	47007
dictionary full	47008
empty stack	47014
FORTH not re-entrant	47064
FORTHRAM file not in place	47018
full stack	47015
HP-IL error	47010
illegal CASE structure	47060
in protected dictionary	47013
Invalid Filespec	47019
no DO before LEAVE	47059
no ending "	47005
no ending)	47004
no ending ;	47003
____ Not Found	47002
not in current vocabulary	47054
____ not recognized	47016
string won't fit	47053

Numerical Listing of FORTH Messages with Descriptions

Error Number	Message and Condition
47002	____ Not Found The argument to ' (tick) isn't in the dictionary. Check the spelling of the word.
47003	no ending ; The definition being compiled from a text file is unfinished. Put in an ending semicolon.
47004	no ending) .(or (isn't matched by an ending parenthesis. Put in an ending parenthesis.
47005	no ending " ." or " isn't matched by an ending double quote. Put in an ending double quote.
47006	argument < 1 A word that expects positive integers finds negative numbers or zero on the stack. Ensure the proper values on the stack.
47007	definition not finished The stack's size at the end of a word doesn't equal its size at the start. Review the control structures and immediate words used in the definition.
47008	dictionary full The dictionary space in FORTHRAM is used up. Use FORGET or GROW.
47009	compile only A compile-time word is used at run time. Check word usage in definitions.
47010	HP-IL error Something is wrong related to the HP-IL interface. Check that the HP-IL interface is plugged into the HP-71; check the integrity of the loop.
47011	attempted to redefine null A colon (starting a colon definition) is the only input received from the keyboard; or WORD '' or WORDI '' appears in a primitive assembly. Fatal to assembly. You can't redefine the null word in FORTH.
47013	in protected dictionary The argument for FORGET is below FENCE (or in ROM). Reset FENCE.
47014	empty stack A word expecting stack parameters finds the stack empty. Provide stack parameters.
47015	full stack The space in FORTHRAM for the data stack is used up. Use GROW to enlarge FORTHRAM or use FORGET to make space in FORTHRAM.
47016	____ not recognized The input is neither an existing word nor a number. Check the spelling of the word; check the CONTEXT vocabulary.
47017	conditionals not paired A control-structure word (such as THEN) appears without the preceding word (such as IF). Supply the missing word.

Error Number	Message and Condition
47018	FORTHRAM file not in place FORTHI, FORTHF, or FORTH$ is attempted when the FORTHRAM file hasn't been created or has moved. Use FORTH or FORTHX to enter FORTH and then exit.
47019	Invalid Filespec The argument to FINDF is an illegal file specifier. Supply a valid file specifier.
47043	address not inside a file ADJUSTF is given an address not properly within a file, such as the address of a file header. Check the address of the file.
47046	bad parameters A string word finds an out-of-range value on the stack, such as a character-position parameter of 20 for a string only 10 characters long. Check the stack value.
47052	Configuration An oversized configuration buffer or an erroneous pointer to that buffer prevents the FORTHRAM file from occupying its required location. This will never occur under normal circumstances. Remove a LEX file from RAM or remove a module.
47053	string won't fit A string is too long for the specified variable. Check the size of the variable.
47054	not in current vocabulary The argument for FORGET isn't in the CURRENT vocabulary. Check the spelling of the word and the CURRENT vocabulary.
47055	cannot load The file is open, doesn't exist, etc. Check the file's status.
47059	no DO before LEAVE LEAVE is used outside a DO-loop. Use LEAVE only inside a DO-loop.
47060	illegal CASE structure ENDCASE isn't preceded by valid CASE . . . OF . . . ENDOF structure. Check the complete control structure.
47063	BASIC not re-entrant BASICX is used in an argument to FORTHX. Eliminate such usage.
47064	FORTH not re-entrant FORTHX is used in an argument to BASICX, or in a program or user-defined function executed from BASICX, BASICI, or BASICF. Eliminate such usage.

Assembler Messages

`assembler aborted`
User has aborted the assembler.

`attempted to redefine null`
A colon (starting a colon definition) is the only input received from the keyboard; or `WORD ' '` or `WORDI ' '` appears in a primitive assembly. Fatal to assembly. You can't redefine the null word in FORTH.

`cannot open source file`
The argument to `ASSEMBLE` is invalid, missing, or the file is open. Fatal to assembly. Check that the source file is a text file in RAM or on tape.

`cannot resolve equate`
The evaluation of the equate differs between the first and second passes. Check that all parts of the expression can be evaluated during the first pass.

`dictionary full`
The dictionary space in FORTHRAM is used up. Use `FORGET` or `GROW`.

`duplicate label`
An existing label name is used again. Recall that labels of more than six characters are defined by the first six characters. Rename the duplicate label.

`excess characters in expression`
An expression contains too many characters. Check that the expression is stated correctly.

`GOYES or RTNYES required`
A test instruction isn't followed by a `GOYES` or `RTNYES` instruction. The test and branch instructions appear to be separate but combine to form one instruction. Supply the missing `GOYES` or `RTNYES` instruction.

`illegal dp arithmetic value`
An illegal value is used in data-pointer arithmetic. Check that the value of the modifier field is from 1 through 16.

`illegal expression`
An expression has illegal syntax or is too complicated. Check the syntax, the levels of parentheses, and the number of operations.

`illegal pointer position`
The pointer register is set to or tested for an illegal value. Check that the value of the modifier field is from 0 through 15.

`illegal status bit`
The status bits are set to or tested for an illegal value. Check that the value of the modifier field is from 0 through 15.

`illegal transfer value`
An illegal value is used in data transfer. Check that the modifier field contains a valid word select or a number from 1 through 16.

`illegal word select`
The modifier field isn't a valid word select. Valid entries are: A, B, M, P, S, W, WP, X, and XS.

`invalid filename specifier`
The filename specifier following a `LEX` or `BIN` pseudo-op isn't a valid filename. Fatal to assembly. Refer to the *HP-71 Owner's Manual* for valid filenames.

`Invalid Filespec`
The argument to `FINDF` is an illegal file specifier. Supply a valid file specifier.

`invalid listing argument`
The modifier field of `LIST` is neither `ON` nor `OFF`. Check that the modifier is uppercase.

`invalid listing file`
The contents of LISTING are invalid, or listing file equals source file. Fatal to assembly. Set LISTING to `NULL$` for no listing, to an HP-IL device for a listing to that device, or to a RAM filename for a listing to that text file.

`invalid quoted string`
One or both single quotes are missing from a quoted string or quoted constant.

`jump or value too large`
A relative jump is too great, or the value of a constant requires more nibbles than the instruction can generate. Use a mnemonic for a longer jump, or check the value of the constant.

`listing file full`
There is no space in RAM for more of the listing file. Move the listing and source files out of RAM, or move other files to external storage.

`listing file not TEXT`
The file specified in LISTING already exists and isn't an HP-71 text file. Fatal to assembly. Provide a different file specifier.

`mismatched parentheses`
A right parenthesis is missing. Supply right parenthesis.

`missing/illegal label`
Illegal characters appear in a label, or a label required for an `EQU` instruction is missing. Check that a legal label is present.

`missing/multiple file type`
The first line in the source file isn't `LEX`, `BIN`, or `FORTH` (fatal to assembly); or one of these pseudo-ops appears a second time; or any pseudo-op of the wrong type appears (such as `WORD` within a LEX file). Check that the source file begins with a `LEX`, `BIN`, or `FORTH` pseudo-op and contains pseudo-ops suitable for that type of file.

`needs previous test instruction`

A `GOYES` or `RTNYES` instruction appears without a preceding test instruction. The test and branch instructions appear to be separate but combine to form one instruction. Supply the missing test instruction.

`non-hexadecimal digit present`

The modifier field contains illegal characters. Use only hex digits 0 through F.

`not enough memory for assembler`

There is insufficient space in RAM for the required assembler variables, files, or operations. Fatal to assembly. Put the listing file to an HP-IL device; move the source file (or other files) to external storage.

`pagesize too small`

PAGESIZE is set to less than 8. Fatal to assembly. Set PAGESIZE to 8 or greater.

`restricted label FiLeNd exists`

The user has placed this restricted label in the source file. Fatal to assembly. Choose a different label.

`symbol table full`

There is no space in RAM for more symbols. Fatal to assembly. Move listing file or source file out of RAM.

`too many ASCII chars present`

The modifier field contains more than eight ASCII characters. Use no more than eight ASCII characters.

`too many hex digits present`

The modifier field contains more than 16 hex digits. Use no more than 16 hex digits.

`unknown opcode`

The opcode isn't recognized. Check that the opcode is spelled correctly, in uppercase letters, and properly placed in an opcode field.

`unrecognized label`

An undefined label appears within an expression. Check whether parentheses are required to separate the label from an operator.

`warning: word not unique`

name in `WORD` '*name*' is already present in the FORTH dictionary.

Editor Messages

DONE

The editor has been exited.

File Exists: ____

The file specified to receive deleted lines already exists. Use the ± option, or choose a different filename.

Insufficient Memory

There is insufficient memory for the operation being performed. If other operations requiring less memory can be performed, the Cmd: prompt returns to the display. If no further operations are possible, the editor is exited. Purge a file or execute DESTROY ALL.

Invalid File Type: ____

The file specified in the command string must be a text file.

Invalid Param: ____

The editor doesn't recognize the parameter portion of a command string. Review the command's syntax.

Line Too Long

The line of text is longer than 96 characters, which is not allowed in text mode.

? Cmd: ____

The editor doesn't recognize the letter as a valid command. The valid commands are c, d, e, f, h, i, l, m, p, r, s, and t.

Working...

The editor is executing a command.

Appendix C

BASIC Keywords

Introduction

This Appendix describes the BASIC keywords added to the HP-71 when the FORTH/Assembler ROM is plugged in. The keywords fall into three categories:

BASIC-to-FORTH	Editor	Remote Keyboard
FORTH	DELETE#	ESCAPE
FORTH$	EDTEXT	KEYBOARD IS
FORTHF	FILESZR	RESET ESCAPES
FORTHI	INSERT#	
FORTHX	MSG$	
	REPLACE#	
	SCROLL	
	SEARCH	

Organization

Entries in this appendix are arranged in alphabetical order. The same format is used for every keyword entry so that you can quickly find the information you need. The format is similar to that used in the *HP-71 Reference Manual*—refer to that manual for additional details.

Each keyword entry provides the following information for the keyword:

- **Keyword name.** Shows the basic keyword.
- **Purpose.** Gives a one-line summary of the operation that the keyword performs.
- **Keyword type.** Identifies the keyword as a *statement* or as a *function.* (None of the keywords are operators.)
- **Execution options.** Indicates situations in which you can execute the keyword:
  - From the keyboard.
  - In CALC mode.
  - After THEN or ELSE in an IF ... THEN ... ELSE statement.
  - While the HP-71 is operating as an HP-IL device (not as controller). This is given only for HP-IL words.

- **Syntax diagram.** Defines the required and optional components within the statement or function for proper syntax. Parameters shown within brackets are optional. Parameters shown in a vertical stack are alternatives.
- **Examples.** Illustrates and explains some ways that the keyword can be used, and shows some possible syntax variations.
- **Input parameters.** Defines the parameters used in the syntax diagram, gives their default values (if applicable), and lists restrictions on parameter values or structure. (This heading isn't included for keywords that use no parameters.)
- **Operation.** Gives a detailed description of the keyword's operation and other information that's useful for learning and using the keyword.
- **Related keywords.** Lists other keywords that either influence the results of the subject keyword or else are similar in function.

DELETE#

Deletes one record from a text file.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE

`DELETE#` *channel number* , *record number*

Example

`DELETE# 5,14` — Deletes the 14th record from the text file currently assigned to channel #5.

Input Parameters

Item	Description	Restrictions
channel number	Numeric expression rounded to an integer.	1 through 255.
record number	Numeric expression rounded to an integer.	

Operation

The `DELETE#` keyword deletes the specified record from the text file assigned to the specified channel number. Record numbers always begin at 0, so line number 1 is record number 0.

The channel number and the record number can be expressions. `DELETE#` rounds each of the resulting values to an integer.

`DELETE#` returns an error message if the assigned file is external, protected, or not a text file.

Related Keywords

`ASSIGN#`, `INSERT#`, `REPLACE#`, `FILESZR`

EDTEXT

Invokes the text editor.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE

EDTEXT *file specifier*[, *command string*]

Examples

`EDTEXT SCREEN`	Runs the editor program, with SCREEN as the edit file.
`EDTEXT SCREEN, L`	Runs the editor program, with SCREEN as the edit file. Begins by listing the file to the display device.

Input Parameters

Item	Description	Restrictions
file specifier	String expression or unquoted string.	File must be in RAM or IRAM.
command string	See description of editor command strings in section 3.	

Operation

The EDTEXT keyword starts the editor program. The optional command string permits you to have the editor begin immediate execution of editor commands that appear in the command string.

An error can cause the editor program to terminate without going through its normal exit path. If you are running the editor from another BASIC program, or from the FORTH environment, you can check for this situation by using DISP$ to read the display contents. If the result is other than Done: <filename>, then you will know that the editor has encountered a fatal error, the edit file may be in a corrupt state, and the editor key assignments may still be active. For example, from the FORTH environment, you can type the sequence

```
" EDTEXT SCREEN" BASICX " DISP$" BASICX DROP @ -102588 =
```

to edit the file SCREEN. When the editor terminates, a true flag will be pushed on the stack if the editor terminated normally (here we are checking the numerical equivalent of the first three characters on the display to see if they match "Don", which translates to −102588).

Related Keywords

ASSIGN#, DELETE#, REPLACE#, FILESZR

ESCAPE

Adds or modifies an escape-sequence key specification in the current KEYBOARD IS key map buffer.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE
	■ Device Operation

ESCAPE *string*, *key number*

Example

`ESCAPE "A",43`	Specifies that the escape sequence (ESC)A received from the KEYBOARD IS device will be changed to key code 43.
`ESCAPE "A",0`	Cancels the (ESC)A assignment.

Input Parameters

Item	Description	Restrictions
string	String expression.	Only the first character is used.
key number	Keycode.	0 through 168.

Operation

ESCAPE specifies that a particular one-character escape sequence (the escape character ASCII 27 followed by a single character) received by the HP-71 from the current KEYBOARD IS device will be replaced by an HP-71 keycode in the key buffer input. ESCAPE requires two parameters, a one-character string and a numeric keycode. The string specifies the escape sequence; the number indicates the corresponding keycode.

The first execution of ESCAPE creates a special HP-71 buffer that specifies the mapping of escape sequences received from a KEYBOARD IS device to HP-71 keycodes. Each subsequent use of ESCAPE will add a new character/key code mapping, or modify an existing one, in the buffer. You can clear the buffer completely by executing RESET ESCAPE. The buffer will be cleared if you turn on the HP-71 when the FORTH/Assembler ROM is not installed.

ESCAPE (continued)

A mapping of an escape sequence created with ESCAPE can be cancelled by assigning keycode 0 to the character:

ESCAPE "*character*",0

removes the entry for *character* from the keymap buffer.

As an example of the use of ESCAPE, suppose that you have connected a terminal to the HP-71 through the HP 82164A HP-IL/RS232 interface. On many terminals the cursor up, down, right, and left keys transmit the escape sequences (ESC)A, (ESC)B, (ESC)C, and (ESC)D, respectively. The following program will cause these sequences to map to the corresponding cursor keys on the HP-71, when the terminal is the KEYBOARD IS device:

10 RESET ESCAPE	Purges any former key map buffer.
20 ESCAPE "A",50	Maps (ESC)A to cursor-up key (50).
30 ESCAPE "B",51	Maps (ESC)B to cursor-down key (51).
40 ESCAPE "C",48	Maps (ESC)C to cursor-right key (48).
50 ESCAPE "D",47	Maps (ESC)A to cursor-left key (47).
60 END	

Related Keywords

KEYBOARD IS, RESET ESCAPE

FILESZR

Returns the number of records in a text file.

☐ Statement	■ Keyboard Execution
■ Function	☐ CALC Mode
☐ Operator	■ IF...THEN...ELSE

```
FILESZR (filename)
```

Example

`X=FILESZR("SCREEN")` Sets the variable `X` equal to the number of records in the text file `SCREEN`.

Input Parameters

Item	Description	Restrictions
file name	String expression or quoted string.	Can not include a device specifier or CARD.

Operation

The `FILESZR` keyword returns the number of records in the file specified, if that file exists. If the file does not exist, or the operation fails for any other reason, a negative number is returned. The absolute value of the negative number is the error number of the error that caused the function to fail.

Related Keywords

`INSERT#`, `DELETE#`, `REPLACE#`

FORTH

Transfers HP-71 operation to the FORTH environment.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	□ IF...THEN...ELSE

```
FORTH
```

Operation

Keyboard execution of FORTH (it is not programmable) causes the HP-71 to exit the BASIC operating system environment and transfer control to the FORTH environment. The message HP-71 FORTH 1A is displayed. Subsequent keyboard input is interpreted by the FORTH outer interpreter.

If the HP-71 is turned off while FORTH is active, it will automatically reenter the FORTH environment when the HP-71 is turned back on.

Execution of the FORTH word BYE will return the HP-71 to BASIC.

Because of the complete access to the HP-71 memory space provided by FORTH, it is quite possible for a FORTH program to store inappropriate data into HP-71 operating system RAM. In many cases, this will cause a memory lost condition. Following a memory loss, the HP-71 will return to the BASIC environment.

Related Keywords

FORTH$, FORTHF, FORTHI, FORTHX

FORTH$

Returns to a BASIC string variable the contents of a string defined in the FORTH environment by an address and character count on the FORTH data stack.

☐ Statement	■ Keyboard Execution
■ Function	☐ CALC Mode
☐ Operator	■ IF...THEN...ELSE

```
FORTH$
```

Examples

`A$=FORTH$`	Returns the value of the FORTH string to the BASIC variable `A$`.
`C$=C$&FORTH$`	Concatenates the FORTH string to `C$`.

Operation

`FORTH$` reads a string specified by the address and character count on the FORTH data stack and returns its value to a BASIC string variable. The contents of the FORTH data stack must already have been established prior to execution of `FORTH$`. If there are fewer than two values on the data stack when `FORTH$` is executed, an error will occur, producing the message `FTH ERR:empty stack`.

When `FORTH$` is executed, two values are dropped from the top of the FORTH data stack. There is no other effect on the FORTH environment. If the FORTHRAM file does not exist, the message `FTH ERR:FORTHRAM not in place` will be displayed.

Related Keywords

`FORTH`, `FORTHF`, `FORTHI`, `FORTHX`

FORTHF

Returns the contents of the FORTH floating-point X-register to a BASIC numeric variable.

☐ Statement	■ Keyboard Execution
■ Function	■ CALC Mode
☐ Operator	■ IF...THEN...ELSE

```
FORTHF
```

Examples

`X=FORTHF`	Copies the contents of the FORTH X-register to the BASIC variable `X`.
`X=SIN(FORTHF)`	Computes the sine of the contents of the X-register and places the result in the BASIC variable `X`.
`FORTHX'" A" BASICF FWORD'` `B=FORTHF`	Copies the BASIC variable `A` to the FORTH X-register, then executes a FORTH word `FWORD`, and returns the resulting value from the X-register to the BASIC variable `B`.

Operation

`FORTHF` allows floating-point numeric data in the FORTH environment to be accessed from the BASIC environment. `FORTHF` copies the contents of the FORTH floating X-register to a BASIC numeric variable. The contents of the FORTH floating-point stack remain unchanged, and there is no other effect on the FORTH environment.

The FORTH environment can be configured prior to execution of `FORTHF` through the keyword `FORTHX`. If the FORTHRAM file does not exist, the message `FTH ERR:FORTHRAM not in place` will be displayed.

Related Keywords

`FORTH`, `FORTH$`, `FORTHI`, `FORTHX`

FORTHI

Returns the top value from the FORTH data stack to a BASIC numeric variable.

☐ Statement	■ Keyboard Execution
■ Function	■ CALC Mode
☐ Operator	■ IF...THEN...ELSE

```
FORTHI
```

Examples

`I=FORTHI`	Moves the top value from the FORTH data stack to the BASIC variable I.
`I=FORTHI^2`	Computes the square of the value on the FORTH data stack and places the result in the BASIC variable I.
`FORTHX'" I" BASICI FWORD'` `B=FORTHI`	Copies the BASIC variable I to the FORTH data stack, then executes a FORTH word `FWORD`, and returns the resulting top value from the data stack to the BASIC variable `B`.

Operation

`FORTHI` allows values contained on the FORTH data stack to be accessed from the BASIC environment. `FORTHI` moves the value on the top of the FORTH data stack to a BASIC numeric variable. The value is dropped from the data stack, but there is no other effect on the FORTH environment.

If there are no values on the data stack when `FORTHI` is executed, an error will occur, producing the message `FTH ERR:empty stack`. The FORTH environment can be configured prior to execution of `FORTHF` through the keyword `FORTHX`. If the `FORTHRAM` file does not exist, the message `FTH ERR:FORTHRAM not in place` will be displayed.

Related Keywords

`FORTH`, `FORTH$`, `FORTHF`, `FORTHX`

FORTHX

Executes a FORTH command string.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE

`FORTHX"` *command string*`"`[`,`*parameter list*]

Example

```
FORTHX "DROP + . TYPE CR",
"Hello",1,2,3
```

Push onto the FORTH data stack the address and character count of the string "Hello," and the values 1, 2, and 3; then execute the FORTH words `DROP`, `+`, `.`, `TYPE`, and `CR`.

Input Parameters

Item	Description	Restrictions
command string	String expression.	Contains valid FORTH words.
parameter list	Numeric expressions and string expressions, separated by commas.	Maximum of 14 parameters.

Operation

The `FORTHX` keyword allows you to execute FORTH routines from the BASIC environment. The optional parameter list is a list of up to 14 string or numeric expressions, separated by commas. Each item in the list is pushed onto the FORTH data stack: numbers as single length numbers, and strings each as two numbers representing the address and character count of the string. After the parameters are placed on the stack, the sequence of FORTH words specified in the command string is executed, following which control is returned to the BASIC environment.

`BASICX` can not be included in the command list—the FORTH/BASIC interface does not permit reentrant execution.

The strings passed to FORTH in the parameter list are created in temporary memory. FORTH words can copy those strings to FORTH string variables, or concatenate them to existing strings, but you should not attempt to write other strings to the addresses of the temporary `FORTHX` strings.

Related Keywords

`FORTH`, `FORTH$`, `FORTHF`, `FORTHI`

INSERT#

Inserts one record into a text file.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE

`INSERT#` *channel number* , *record number* ; *new record*

Example

`INSERT# 5,14;"Hello there"`

Inserts the string "Hello there" into the file currently assigned to channel #5, as record 14. The former record 14 becomes record 15.

Input Parameters

Item	Description	Restrictions
channel number	Numeric expression rounded to an integer.	1 through 255.
record number	Numeric expression rounded to an integer.	
new record	String expression.	

Operation

The `INSERT#` keyword inserts the new record at the record number in the file assigned to the specified channel number. The new record is an HP-71 string expression. The channel number and the record number can be expressions. Record numbers always begin at 0, so line number 1 is record number 0. `INSERT#` rounds each of the resulting values to an integer.

The new record is inserted ahead of the record previously numbered at the record number. The former record, and all subsequent records, have their records numbers incremented incremented by 1.

`INSERT#` returns an error message if the assigned file is external, protected, or not a text file.

Related Keywords

`ASSIGN#`, `DELETE#`, `REPLACE#`, `FILESZR`

KEYBOARD IS

Assigns one HP-IL device to be used as an external keyboard.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE
	□ Device Operation

```
                 device specifier
                 [:]NULL
KEYBOARD IS      [:]*
                 "[:]NULL"
                 "[:]*"
```

Examples

`KEYBOARD IS RS232(2)`	Assigns the second HP-IL/RS232 Interface to be the `KEYBOARD IS` device.
`KEYBOARD IS *`	Deactivates any `KEYBOARD IS` assignment.

Input Parameters

Item	Description	Restrictions
device specifier	See standard description in HP-IL Interface Owner's Manual.	None

Operation

The `KEYBOARD IS` statement assigns one HP-IL device to act as a remote keyboard for the HP-71. That is, whenever the HP-71 is expecting keyboard input, it will check the `KEYBOARD IS` device to determine if the device has data available. If so, the data will be read into the HP-71 key buffer, and executed as if it had been entered from the HP-71 keyboard. The HP-71 keyboard continues to function normally. Input can be mixed from the HP-71 keyboard and the remote keyboard.

`KEYBOARD IS` is deactivated by either of the statements `KEYBOARD IS NULL` or `KEYBOARD IS *`.

While `KEYBOARD IS` is active, the HP-71 is continually transmitting on HP-IL. This results in an increase in power consumption, even while the HP-71 is apparently idle. It is recommended that you connect the AC adapter to the HP-71 to conserve battery life while you are using `KEYBOARD IS` for remote input.

KEYBOARD IS (continued)

If the loop is broken while KEYBOARD IS is active, press [ATTN] twice to restore HP-71 operation. When the loop is restored, execute RESET HPIL, reinitialize the keyboard device, and execute KEYBOARD IS again.

By making DISPLAY IS and KEYBOARD IS assignments to the same HP-IL device (usually an interface class device), almost any terminal, or computer acting as a terminal emulator, can be used as an extension of the HP-71 keyboard and display. Most HP-71 operations can be executed from the terminal just as if they were keyed in directly on the HP-71. If you set Flag −21, the automatic loop power down that occurs when the HP-71 turns itself off will be disabled, so that the KEYBOARD IS device can turn the HP-71 on remotely.

For proper operation of KEYBOARD IS, the designated device must be enabled to set HP-IL service requests when it has data available. You can refer to the owner's manual for an HP-IL device to determine how to enable the device. For example, the following sequence will set up the HP 82164A HP-IL/RS232 Interface for use as the KEYBOARD IS device:

```
REMOTE @ OUTPUT RS232;"SE0;SE3;" @ LOCAL @ DTH$(MOD(SPOLL
("RS232"),2^20))
```

The remote mode command SE0 disables any current service request mode on the interface; SE3 sets the interface for service request on data available. The status read (SPOLL) shows any error condition—the DTH$ formats the device status in hexadecimal. A normal status will show the friendly "A1" as the last byte.

All characters received from the KEYBOARD IS device are placed directly into the key buffer, with the following two exceptions:

1. "Control characters," i.e., characters corresponding to ASCII codes from 0 through 31, are generated on the HP-71 by pressing the [g][CTRL] combination followed by another character. The latter character determines the output character according to its ASCII code: the control character will have the ASCII value 64 less than the keyed character. For example, character 1 is generated by pressing [g][CTRL] A (A=ASCII 65). KEYBOARD IS makes the same translation of control characters to keyboard characters. Control characters received from the KEYBOARD IS device are replaced in the key buffer by two keycodes—key 158 ([g][CTRL]) plus an additional keycode to specify the control character according to the mapping just described.
2. One-character escape sequences (the escape character ASCII 27 followed by one additional character), which can optionally be replaced in the input stream by user-specified HP-71 keycodes. Through use of the ESCAPE keyword, the user can map such escape sequences into arbitrary HP-71 keys (such as ON or the command stack) from the remote keyboard. (Notice, however, that remote execution of the ON key will not interrupt the HP-71 unless it is expecting keyboard input.) For a complete explanation of this feature, refer to the documentation of the ESCAPE keyword.

Related Keywords

ESCAPE, RESET ESCAPE, DISPLAY IS, PRINTER IS

MSG$

Returns the message string corresponding to a specified error number.

☐ Statement	■ Keyboard Execution
■ Function	☐ CALC Mode
☐ Operator	■ IF...THEN...ELSE

```
MSG$(error number)
```

Example

`A$=MSG$(58)`	Places the message string associated with error #58 into the string variable `A$`.

Input Parameters

Item	Description	Restrictions
error number	Numeric expression.	Valid error number.

Operation

The `MSG$` keyword provides access to the error message strings generated by the HP-71 operating system, the FORTH/Assembler ROM, or any other LEX file. `MSG$(`*n*`)` returns the string corresponding to the *n*th error.

`MSG$` is a generalization of the keyword `ERRM$`, which returns the message string associated with the most recent error.

Related Keywords

`ERRN`, `ERRL`, `ERRM$`

REPLACE#

Replaces one record in a text file.

■ Statement	■ Keyboard Execution
☐ Function	☐ CALC Mode
☐ Operator	■ IF...THEN...ELSE

REPLACE# *channel number* , *record number* ; *new record*

Example

`REPLACE# 5,14;"Hello there"`	Replaces the 14th record in the text file currently assigned to channel #5, with the string "Hello there".

Input Parameters

Item	Description	Restrictions
channel number	Numeric expression rounded to an integer.	1 through 255.
record number	Numeric expression rounded to an integer.	
new record	String expression.	

Operation

The REPLACE# keyword replaces a specified record, in the text file assigned to the specified channel number, with a new record. The new record is an HP-71 string expression. The channel number and the record number can be expressions. Record numbers always begin at 0, so line number 1 is record number 0. REPLACE# rounds each of the resulting values to an integer.

REPLACE# returns an error message if the assigned file is external, protected, or not a text file.

Related Keywords

ASSIGN#, DELETE#, INSERT#, FILESZR

RESET ESCAPE

Purges any existing key-map buffer created by the ESCAPE keyword.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE
	■ Device Operation

```
RESET ESCAPE
```

Related Keywords

KEYBOARD IS, ESCAPE

SCROLL

Scrolls the display to a position and waits for a key to be pressed.

■ Statement	■ Keyboard Execution
□ Function	□ CALC Mode
□ Operator	■ IF...THEN...ELSE

```
SCROLL position
```

Example

`DISP "Hello there" @ SCROLL 4`	Display the string "Hello there," with the fourth character in the string as the first character in the display, so that the display shows "lo there."

Input Parameters

Item	Description	Restrictions
position	Numeric expression rounded to an integer.	1 through 96.

Operation

The SCROLL keyword enables you to display a string, under program control, that can be scrolled from the keyboard. Execution of SCROLL causes the current display string to shift so that the character in the position specified by the numeric expression is the leftmost character in the display. Execution halts, so that a user can press the left- and right-arrow keys to scroll the display. Execution resumes when any other key is pressed (the pressed keycode is placed in the key buffer). The number input with SCROLL must be greater than zero.

SEARCH

Finds a string in a text file.

☐ Statement	■ Keyboard Execution
■ Function	☐ CALC Mode
☐ Operator	■ IF...THEN...ELSE

```
SEARCH(search string, column number, begin line, end line, channel)
```

Example

`X=SEARCH("Hello",5,1,99,2)`	Searches the file assigned to channel #2 for the string "Hello." The search starts in column 5, line 1, and extends through line 99.

Input Parameters

Item	Description	Restrictions
search string	String expression.	1 through 9999.
column number	Numeric expression rounded to an integer.	1 through 9999
begin line	Numeric expression rounded to an integer.	1 through 9999
end line	Numeric expression rounded to an integer.	1 through 9999
channel	Numeric expression rounded to an integer.	1 through 255

Operation

The `SEARCH` keyword enables you to determine the location of a specified string within an HP-71 text file. If the search is successful, `SEARCH` returns a value in the format *nnn.cccll*, where *nnn* is the record number, *ccc* is the column number, and *lll* is the length of the matched string. If the search is unsuccessful, zero is returned.

The search string can be any string expression, and the other parameters can be any numeric expression. Each input value is rounded to an integer. A zero is returned for an empty file.

Related Keywords

`INSERT#`, `DELETE#`, `REPLACE#`

Appendix D

FORTH Words

This appendix describes all FORTH words in the FORTH/Assembler ROM. The words appear in ASCII order. For a discussion of nonstandard FORTH operations, refer to section 2, "The HP-71 FORTH System." For a listing of all FORTH words grouped by functional category, refer to the inside back cover of this manual.

Each entry shows the word, its pronunciation, its use of the data stack, and a brief description of the word's operation. A word EXAMPLE might have the following entry:

EXAMPLE	*(Example)*	n_1 n_2 → n_3

Perform the specified operation on n_1 and n_2, replacing them on the data stack with the result n_3. (Before EXAMPLE is executed, n_2 is on the top of the stack. After EXAMPLE is executed, n_3 is on the top of the stack.)

Some descriptions begin with "COMPILE" or "IMMEDIATE." These indicate the following:

- COMPILE indicates that the word is intended for use only during compilation. Direct execution of the word can give meaningless or dangerous results; where appropriate, a FTH ERR: compile only error occurs.
- IMMEDIATE indicates that the word is executed, rather than compiled, when encountered during compilation.

Notation

The stack-use diagrams use the following variables to represent various types of data.

Definition of Stack Variables

Variable	Type of Data
n	A signed (twos complement) 20-bit integer.
un	An unsigned 20-bit integer.
d	A signed (twos complement) 40-bit integer.
ud	An unsigned 40-bit integer.
flag	A signed (twos complement) 20-bit value, either −1 (true) or 0 (false).
c	A 20-bit value whose two low-order nibbles represent an ASCII character.
addr	A 20-bit address.
count	A 20-bit value whose two low-order nibbles represent the number of characters in a string.
str	A 40-bit value comprising *addr* and *count*. *Count* is on top and tells how many characters are to be found at *addr*.

Errors

Many FORTH words require one or more parameters on the data stack. When a word is executed with too few parameters on the stack, unpredictable errors will occur. The error message `FTH ERR: empty stack` might be displayed, but only after the operation is carried out on spurious parameters. These spurious parameters come from the terminal input buffer (TIB), which resides above the data stack. If a result is returned, it will be written into the TIB, and an error message like `FTH ERR: xYzgt not recognized` occurs when FORTH tries to interpret this result as a character string containing FORTH words and data.

FORTH is similar to assembly language in its lack of user protection. In most cases FORTH will attempt to perform the specified operation, even if the operation will cause a Memory Lost condition. For instance, it is easy to write a FORTH loop that pushes a value onto the data stack 1,000,000 times. Execution of this loop will overwrite the user dictionary, the FORTH system variables, and the BASIC O/S variables. Eventually the machine will be too confused to continue and will perform a cold start. In other cases you might need to perform an `INIT 3` to recover normal HP-71 operation.

FORTH Glossary

!	*(Store)*	*n addr* →

Store *n* at *addr*.

"	*(Quote)*	→ *str*

Used in the form: " *ccc*"

IMMEDIATE. In execute mode: Take the characters *ccc*, terminated by the next ", from the input stream, and store them in a temporary string variable at the PAD. The string variable's header shows a maximum length of 80 characters or the current length, whichever is greater. Any other word that returns another temporary string will wipe out the first string.

In compile mode: Compile into the dictionary the runtime address of ", two bytes for the length of the string *ccc* (maximum length = current length), and the string itself. A string must be contained on a single line of a source file.

#	*(Sharp)*	ud_1 → ud_2

Used in the form: <# ### #>

Divide ud_1 by BASE, convert the remainder to an ASCII character, place this character in an output string, and return the quotient ud_2. Used in pictured output conversion; refer to <#.

#>	*(Sharp-greater)*	*ud* → *addr n*

End pictured output conversion. #> drops *ud* and returns the text address and character count. (These are suitable inputs for TYPE.)

#S	*(Sharp-s)*	*ud* → *0 0*

Convert *ud* into digits (as by repeated execution of #), adding each digit to the pictured numeric-output text until the remainder is zero. A single zero is added to the output if *ud* = 0. Used between <# and #>.

#TIB	*(Number-t-i-b)*	→ *addr*

Return the address of the variable #TIB, which contains the number of bytes in the terminal input buffer. Set by QUERY.

'	*(Tick)*	→ *addr*

Used in the form: ' *name*

Return the CFA of *name*.

'STREAM	*(Tick-stream)*	→ *addr*

Return the address of the next character in the input stream.

(	*(Paren)*	→

Used in the form: (*ccc*)

IMMEDIATE. Consider the characters *ccc*, delimited by), as a comment to be ignored by the text interpreter. The blank following (is not part of *ccc*. (may be freely used while interpreting or compiling. A comment must be contained on a single line of a source file.

*****	*(Times)*	n_1 n_2 → n_3

Return the arithmetic product of n_1 and n_2.

***/**	*(Times-divide)*	n_1 n_2 n_3 → n_4

Multiply n_1 and n_2, divide the result by n_3, and return the quotient n_4. The product of n_1 and n_2 is maintained as an intermediate 40-bit value for greater precision in the division.

***/MOD**	*(Times-divide-mod)*	n_1 n_2 n_3 → n_4 n_5

Multiply n_1 and n_2, divide the result by n_3, and return the remainder n_4 and the quotient n_5. The product of n_1 and n_2 is maintained as an intermediate 40-bit value for greater precision in the division.

+	*(Plus)*	n_1 n_2 → n_3

Return the arithmetic sum of n_1 and n_2.

+!	*(Plus-store)*	*n addr* →

Add *n* to the 20-bit value at *addr*.

+BUF	*(Plus-Buff)*	$addr_1$ → $addr_2$ *flag*

Advance the mass-storage-buffer address ($addr_1$) to the address of the next buffer ($addr_2$). +BUF returns a false flag if $addr_2$ is the address of the buffer currently pointed to by PREV; otherwise, +BUF returns a true flag.

,	*(Comma)*	*n* →

Used in the form: 1234 ,

Allot five nibbles and store *n* in the dictionary.

−	*(Minus)*	n_1 n_2 → n_3

Subtract n_2 from n_1 and return the difference n_3.

−TRAILING	*(Dash-trailing)*	*addr* $count_1$ → *addr* $count_2$

Adjust the character count of the text beginning at *addr* to exclude trailing blanks.

.	*(Dot)*	*n* →

Convert *n* according to BASE and display the result in a free-field format with one trailing blank. Display a minus sign if *n* is negative.

."	*(Dot-quote)*	→

Used in the form: ." *ccc*"

COMPILE, IMMEDIATE. Compile the characters *ccc*, delimited by ", so that later execution will transmit *ccc* to the current display device. The blank following ." is not part of *ccc*. A string must be contained on a single line of a source file.

.(	*(Dot-paren)*	→

Used in the form: .(*ccc*)

IMMEDIATE. Display the characters *ccc*, delimited by). The blank following .(is not part of *ccc*. A string must be contained on a single line of a source file.

.S	*(Dot-S)*	→

Print the contents of the stack as unsigned integers, starting with the top of the stack. .S doesn't alter the stack.

/	*(Divide)*	n_1 n_2 → n_3

Divide n_1 by n_2, and return the quotient n_3. Division by 0 always yields 0.

/MOD	*(Divide-mod)*	n_1 n_2 → n_3 n_4

Divide n_1 by n_2, and return the remainder n_3 and quotient n_4.

0	*(Zero)*	→ *0*

Return the constant 0.

0<	*(Zero-less)*	*n* → *flag*

Return a true flag if $n < 0$; otherwise, return a false flag.

0=	*(Zero-equals)*	*n* → *flag*

Return a true flag if $n = 0$; otherwise, return a false flag.

0>	*(Zero-greater)*	*n* → *flag*

Return a true flag if $n > 0$; otherwise, return a false flag.

1	*(One)*	→ *1*

Return the constant 1.

1+	*(One-plus)*	*n* → *n+1*

Increment n by 1.

1−	*(One-minus)*	*n* → *n−1*

Decrement n by 1.

1/X	*(Reciprocal-of-X)*	→

Divide 1.0 by the contents of the X-register. `1/X` places the result in the X-register and the original value of x in the LAST X register.

10^X	*(10-to-the-X)*	→

Raise 10 to the power contained in the X-register. `10^X` places the result in the X-register and the original value of x in the LAST X register.

2	*(Two)*	→ *2*

Return the constant 2.

2*	*(Two-times)*	$n \rightarrow 2n$

Return the product of n and 2.

2+	*(Two-plus)*	$n \rightarrow n+2$

Increment n by 2.

2−	*(Two-minus)*	$n \rightarrow n-2$

Decrement n by 2.

2/	*(Two-divide)*	$n \rightarrow n/2$

Divide n by 2 and return the result. 2/ produces $n/2$ by shifting n one bit to the right and extending the sign bit.

2DROP	*(Two-drop)*	$d \rightarrow$

Drop the double number (or two single numbers) on the top of the data stack.

2DUP	*(Two-dup)*	$d_1 \rightarrow d_1\ d_1$

Duplicate the double number (or pair of single numbers) on the top of the data stack.

2OVER	*(Two-over)*	$d_1\ d_2 \rightarrow d_1\ d_2\ d_1$

Make a copy of the second double number (or third and fourth single numbers) on the data stack.

2SWAP	*(Two-swap)*	$d_1 \ d_2 \rightarrow d_2 \ d_1$

Reverse the order of the two double numbers on the top of data stack.

3	*(Three)*	$\rightarrow 3$

Return the constant 3.

4N@	*(Four-n-fetch)*	*addr* → *n*

Return the four-nibble (two-byte) quantity located at *addr*.

5+	*(Five-plus)*	$n \rightarrow n+5$

Increment *n* by 5.

5−	*(Five-minus)*	$n \rightarrow n-5$

Decrement *n* by 5.

:	*(Colon)*	→

Used in the form: `:` *name* . . . `;`

Create a word definition for *name* in the compilation vocabulary and set compilation state. The search order is changed so that the first vocabulary in the search order is replaced by the compilation vocabulary. The compilation vocabulary is unchanged. The text from the input stream is subsequently compiled. *name* is called a *colon definition.* The newly created word definition for *name* cannot be found in the dictionary until the corresponding `;` is successfully processed.

;	*(Semicolon)*	→

Used in the form: `:` *name* . . . `;`

IMMEDIATE, COMPILE. Stop compilation of a colon definition. `;` compiles `EXIT` into the dictionary, clears the smudge bit (so that this colon definition can be found in the dictionary), and sets execute state.

<	*(Less-than)*	n_1 n_2 → *flag*

Return a true flag if $n_1 < n_2$; otherwise, return a false flag.

<#	*(Less-sharp)*	→

Initialize pictured numeric output. The words `<#`, `#`, `#S`, `HOLD`, `SIGN`, and `#>` can specify the conversion of a double number into an ASCII-character string stored in right-to-left order.

<>	*(Not-equal)*	n_1 n_2 → *flag*

Return a true flag if $n_1 \neq n_2$; otherwise, return a false flag.

=	*(Equals)*	n_1 n_2 → *flag*

Return a true flag if $n_1 = n_2$; otherwise, return a false flag.

>	*(Greater-than)*	n_1 n_2 → *flag*

Return a true flag if $n_1 > n_2$; otherwise, return a false flag.

>BODY	*(To-body)*	$addr_1$ → $addr_2$

Return the PFA ($addr_2$) of the word whose CFA is $addr_1$. ($addr_2 = addr_1 + 5$.)

>IN	*(To-in)*	→ *addr*

Return the address of the variable >IN, which contains the current offset within the input stream. The offset is expressed in nibbles and points to the first position past the first blank.

>R	*(To-R)*	*n* →

COMPILE. Transfer *n* to the return stack.

?	*(Question-mark)*	*addr* →

Used in the form: `HEX 2FCC5 ?`

Display the number at *addr* using the current `BASE` and the `.` (dot) format.

?COMP	*(Query-comp)*	→

COMPILE. Issue a `FTH ERR: compile only` message if not in compile mode.

?DUP	*(Query-dup)*	*n* → *n (n)*

Duplicate n if $n \neq 0$.

?STACK	*(Query-stack)*	→

Issue a `FTH ERR: empty stack` message if the stack pointer is above the bottom of the stack; or issue a `FTH ERR: full stack` message if the stack pointer has grown into the pad.

?TERMINAL	*(Query-terminal)*	→ *flag*

Return a true flag if a key has been pressed and placed in the key buffer; otherwise, return a false flag.

@	*(Fetch)*	*addr* → *n*

Return the number stored at *addr*.

ABORT	*(Abort)*	→

Reset the data and return stacks, close all files, set execution mode, set `FORTH` as the current and context vocabulary, and return control to the terminal.

ABORT"	*(Abort-quote)*	*flag* →

Used in the form: : *name* . . . ABORT" *ccc*" . . . ;

COMPILE, IMMEDIATE. If *flag* is true, display the character string *ccc* (delimited by ") and execute ABORT; otherwise, drop the flag and continue execution. The character string must be contained on a single line of a source file.

ABS	*(Absolute)*	*n* → |*n*|

Return the absolute value of n.

ACOS	*(A-cos)*	→

Calculate the arc cosine of the contents of the X-register, according to the currently active angular mode. ACOS places the result in the X-register and the original value of x in the LAST X register.

ADJUSTF	*(Adjust-f)*	*addr n* → *flag*

Adjust a file by n nibbles, starting at *addr* and moving toward greater addresses, and return a true flag if successful or a false flag if not. ADJUSTF enlarges the file for positive n or shrinks the file for negative n.

ALLOT	*(Allot)*	*n* →

Add n bytes to the parameter field of the most recently defined word (regardless of the CURRENT and CONTEXT vocabularies).

AND	*(And)*	n_1 n_2 → n_3

Return the bit-by-bit AND of n_1 and n_2.

ASC	*(Ascii)*	*str* → *n*

Return the ASCII value of the first character in the string specified by *str*.

ASIN	*(A-sine)*	→

Calculate the arc sine of the contents of the X-register, according to the currently active angular mode. `ASIN` places the result in the X-register and the original value of x in the LAST X register.

ASSEMBLE	*(Assemble)*	*str* →

Assemble the file whose name is specified by *str*. `ASSEMBLE` uses `BASICX`, so you can't call `ASSEMBLE` from BASIC.

ATAN	*(A-tan)*	→

Calculate the arc tangent of the contents of the X-register, according to the currently active angular mode. `ATAN` places the result in the X-register and the original value of x in the LAST X register.

BASE	*(Base)*	→ *addr*

Return the address of the variable BASE, which contains the current numeric-conversion base.

BASIC$	*(Basic-dollar)*	str_1 → str_2

Used in the form:

```
" A$" BASIC$
" A$[1,3]" BASIC$
```

Return the current value of a BASIC string expression (specified by str_1) to the pad as a FORTH string (specified by str_2.)

BASICF	*(Basic-f)*	*str* →

Used in the form:

```
" A" BASICF
" A1/A6" BASICF
" A9*PI" BASICF
" TIME" BASICF
```

Return the current value of a BASIC numeric expression (specified by *str*) to the FORTH X-register, lifting the floating-point stack.

BASICI	*(Basic-i)*	*str* → *n*

Used in the form: `" A" BASICI`
`" A1/A6" BASICI`

Return the current value of a BASIC numeric expression (specified by *str*). An overflow error occurs if the variable's value exceeds FFFFF.

BASICX	*(Basic-x)*	*str* →

Used in the form:

```
" RUN 'JOE'" BASICX
" BEEP" BASICX
" A=PI" BASICX
" 10 DISP A" BASICX
```

Pass a string (specified by *str*) to the BASIC system for parsing and editing/execution, and then return to FORTH.

BEGIN . . . UNTIL	→

Used in the form: . . . `BEGIN` *actions flag* `UNTIL` . . .

IMMEDIATE, COMPILE. Execute *actions* and test *flag*; if *flag* is false, repeat; if *flag* is true, skip to the word following `UNTIL`.

BEGIN . . . WHILE . . . REPEAT	→

Used in the form: . . . `BEGIN` *actions*$_1$ *flag* `WHILE` *actions*$_2$ `REPEAT` . . .

IMMEDIATE, COMPILE. Execute *actions*$_1$ and test *flag*; if flag is true, execute *actions*$_2$ and repeat; if *flag* is false, skip to the word following `REPEAT`.

BL	*(Blank)*	→ *c*

Return 32_{10}, the ASCII value for a space or blank.

BLK	*(B-l-k)*	$\rightarrow$ *addr*

Return the address of the variable BLK, which contains the number of the line being interpreted from the active file. The value of BLK is an unsigned number; if it is zero, the input stream is taken from the keyboard device.

BLOCK	*(Block)*	*n* $\rightarrow$ *addr*

Return the address of the first byte in the mass-storage-buffer copy of line *n* in the active file. If line *n* hasn't already been copied from the file (in RAM or on mass storage) into a mass storage buffer, `BLOCK` does so.

BYE	*(Bye)*	$\rightarrow$

Exit the FORTH environment and return control to the BASIC environent.

C!	*(C-store)*	*n addr* $\rightarrow$

Store the two low-order nibbles of *n* at *addr*.

C,	*(C-comma)*	*n* $\rightarrow$

`ALLOT` one byte and store the two low-order nibbles of *n* at `HERE`.

C@	*(C-fetch)*	*addr* $\rightarrow$ *byte*

Return the contents of the byte at *addr*. The three high-order nibbles of the five-nibble stack entry are 0.

C@+	*(C-at-plus)*	$str_1 \rightarrow str_2\ c$

Return *c*, the first character in the string specified by str_1, and str_2, where $addr_2 = addr_1 + 2$ and $count_2 = count_1 - 1$. If $count_1 = 0$, $c = 0$ and $str_2 = str1$.

CASE . . . OF . . . ENDOF . . . ENDCASE	*(Case Statements)*	$n \rightarrow$

Used in the form:

```
. . . CASE
n1 OF actions1 ENDOF actions1'
n2 OF actions2 ENDOF actions2'
n3 OF actions3 ENDOF actions3'
. . .
ENDCASE . . .
```

IMMEDIATE, COMPILE. Starting with the first case statement ($i = 1$):

- If $n = n_i$, drop n, execute $actions_i$, and skip to the word following `ENDCASE`.
- If $n \neq n_i$, execute $actions_i'$ and examine the next case statement. (If there are no more case statements, drop n and skip to the word following `ENDCASE`). Note that each optional $actions_i'$ can alter the value of n (the number on the top of the stack) tested by the next case statement.

CHR$	*(Char-dollar)*	$n \rightarrow str$

Convert the two low-order nibbles of n into an ASCII character and place it in a string specified by *str*. The string is a temporary string of length 1, located on the pad.

CHS	*(Change-sign)*	$\rightarrow$

Replace x, the contents of the X-register, with $-x$.

CLOSEALL	*(Close-all)*	$\rightarrow$

Close all open files (that is, files with an open FIB entry).

CLOSEF	*(Close-f)*	$n \rightarrow$

Close the file whose FIB# is n.

CMOVE	*(C-move)*	$addr_1\ addr_2\ un \rightarrow$

Move un bytes, first moving the byte at $addr_1$ to $addr_2$ and finally moving the byte at $addr_1 + 2(un - 1)$ to $addr_2 + 2(un - 1)$. If $un = 0$, nothing is moved.

CMOVE>	*(C-move-up)*	$addr_1$ $addr_2$ un →

Move *un* bytes, first moving the byte at $addr_1 + 2(un - 1)$ to $addr_2 + 2(un - 1)$ and finally moving the byte at $addr_1$ to $addr_2$. If $un = 0$, nothing is moved.

COMPILE	*(Compile)*	→

Used in the form: `:` $name_1$ . . . `COMPILE` $name_2$ . . .

COMPILE. Compile the CFA of $name_2$ when $name_1$ is executed. Typically $name_1$ is an immediate word and $name_2$ is not; `COMPILE` ensures that $name_2$ is compiled, not executed, when $name_1$ is encountered in a new definition.

CONBF	*(Con-buff)*	n_1 n_2 → *flag*

Contract by n_1 nibbles the general-purpose buffer whose ID# is n_2, and return a true flag; or return a false flag if such a buffer doesn't exist. If the specified buffer contains fewer than n_1 nibbles, `CONBF` contracts it to 0 nibbles. n_1 must not exceed FFF.

CONSTANT	*(Constant)*	*n* →

Used in the form: *n* `CONSTANT` *name*

Create a dictionary entry for *name*, placing *n* in its parameter field. Later execution of *name* will return *n*.

CONTEXT	*(Context)*	→ *addr*

Return the address of the variable CONTEXT, which specifies which vocabulary to search first during interpretation of the input stream. (Word searches through successive parent vocabularies are discussed in section 2.)

CONVERT	*(Convert)*	d_1 $addr_1$ → d_2 $addr_2$

Accumulate the string of digits beginning at $addr_1 + 2$ into the double number d_1, and return the result d_2 and the address $addr_2$ of the next non-digit character. For each character that is a valid digit in `BASE`, `CONVERT` converts the digit into a number, multiplies the current double number (initially d_1) by `BASE`, and adds the converted digit to the current double number. When `CONVERT` encounters a non-digit character, it returns the current double number and the non-digit character's address.

COS	*(Cos)*	→

Calculate the cosine of the contents of the X-register, according to the currently active angular mode. COS places the result in the X-register and the original value of x in the LAST X register.

COUNT	*(Count)*	$addr_1$ → $addr_2$ n

Return the address ($addr_2$) of the first character, and the character count (n), of the counted string beginning at $addr_1$. The first byte at $addr_1$ must contain the character count n. The following diagram shows the parameters for a three-character text string:

	Address	**Contents**	
$addr_1$ →	1000	3	← n
$addr_2$ →	1002	A	
	1004	B	
	1006	C	

CR	*(C-r)*	→

Send a carriage-return and line-feed to the current display device.

CREATE	*(Create)*	→

Used in the form: CREATE *name*

Create a standard dictionary entry for *name* without allotting any parameter-field memory. Later execution of *name* will return *name*'s PFA. Words that use CREATE directly are called *defining words.*

CREATEF	*(Create-f)*	*str* *n* → *addr* *str* *n* → false

Create a text file in RAM whose name is specified by *str* and that contains *n* nibbles. If successful, CREATEF returns the address of the beginning of the file header (which contains the file name); otherwise, it returns a false flag. If the specified string exceeds eight characters, the file name will be the first eight characters.

CRLF	*(C-r-l-f)*	→ *str*

Return *str* specifying the two-character string constant containing the ASCII characters carriage-return and line-feed. This string can be concatenated with other strings for use with words such as `OUTPUT`.

CURRENT	*(Current)*	→ *addr*

Return the address of the variable CURRENT, which specifies the vocabulary to receive new word definitions.

D+	*(D-plus)*	d_1 d_2 → d_3

Return the arithmetic sum of d_1 and d_2.

D−	*(D-minus)*	d_1 d_2 → d_3

Subtract d_2 from d_1 and return the difference d_3.

D.	*(D-dot)*	*d* →

Display *d* according to BASE in a free-field format, with a leading minus sign if *d* is negative.

D.R	*(D-dot-R)*	*d* *n* →

Display *d* (according to BASE) right-justified in a field *n* characters wide.

D<	*(D-less-than)*	d_1 d_2 → *flag*

Return a true flag if $d_1 < d_2$; return a false flag otherwise.

DABS	*(D-abs)*	d_1 → $|d|$

Return the absolute value of *d*.

DECIMAL	*(Decimal)*	→

Set the input-output numeric conversion `BASE` to ten.

DEFINITIONS	*(Definitions)*	→

Set the `CURRENT` vocabulary to match the `CONTEXT` vocabulary.

DEGREES	*(Degrees)*	→

Select `DEGREES` angular mode.

DEPTH	*(Depth)*	→ n

Return n, the number of items on the data stack (not counting n itself).

DIGIT	*(Digit)*	c n_1 → n_2 true c n_1 → false

If c is a valid digit in base n_1, return that digit's binary value (n_2) and a true flag; otherwise, return a false flag.

DLITERAL	*(D-literal)*	d →

COMPILE, IMMEDIATE. Compile d into the word being defined, such that d will be returned when the word is executed.

DNEGATE	*(D-negate)*	d → $-d$

Return the twos complement of a double number d.

DO . . . +LOOP (Do, Plus-loop) n_1 n_2 →

Used in the form: . . . `DO` *actions n* `+LOOP` . . .

COMPILE, IMMEDIATE. Execute a definite loop, each time incrementing the loop index by n. `DO` moves n_1 (the loop limit) and n_2 (the initial value of the loop index) to the return stack, with n_2 on top, and then executes *actions*. `+LOOP` increments the index by n (which can be negative) and repeats *actions*, until the index is incremented across the boundary between $n - 1$ and n. For example,

`10 1 DO` *actions* `1 +LOOP`

will execute *actions* nine times, with values of the index from 1 through 9; and

`-10 -1 DO` *actions* `-1 +LOOP`

will execute *actions* ten times, with values of the index from -1 through -10. `DO . . . +LOOP` may be nested within control structures.

DO . . . LOOP n_1 n_2 →

Used in the form: . . . `DO` *actions* `LOOP` . . .

COMPILE, IMMEDIATE. Execute a definite loop, each time incrementing the loop index by 1. `DO` moves n_1 (the loop limit) and n_2 (the initial value of the loop index) to the return stack, with n_2 on top, and then executes *actions*. `LOOP` increments the index by 1 and repeats *actions*, until the index is incremented from $n - 1$ to n. `DO . . . LOOP` may be nested within control structures.

DOES> (Does)

Used in the form: `:` *name* . . . `CREATE` . . . `DOES>` . . . `;`

COMPILE, IMMEDIATE. Define the run-time action of a word created by a defining word. `DOES>` marks the termination of the defining part of the defining word *name* and begins the definition of the run-time action for words that will later be defined by *name*.

DROP (Drop) n →

Drop the top number from the stack.

DUP (Dup) n → n n

Return a second copy of the top number on the stack.

EMIT	*(Emit)*	$c \rightarrow$

Transmit the character c to the current display device.

ENCLOSE	*(Enclose)*	$addr\ c \rightarrow addr\ n_1\ n_2\ n_3$

Examine the string that begins at *addr*, and return:

- n_1, the nibble offset from *addr* to the first character that doesn't match the delimiter character c.
- n_2, the nibble offset from *addr* to the first delimiter character c that follows non-delimiter characters in the string.
- n_3, the nibble offset from *addr* to the first unexamined character.

An ASCII null is treated as an unconditional delimiter.

END$	*(End-dollar)*	$str_1\ n \rightarrow str_2$

Create a temporary string (specified by str_2) consisting of the nth character and all subsequent characters in the string specified by str_1. (RIGHT$ is similar but takes substring length, not character position, for a parameter.)

ENG	*(Engineering)*	$n \rightarrow$

Select engineering display mode with $n + 1$ significant digits displayed, for $0 \leqslant n \leqslant 11$.

ENTER	*(Enter)*	$addr\ n_1 \rightarrow addr\ n_2$ $addr\ n_1\ c\ 0 \rightarrow addr\ n_2$

Receive up to n_1 bytes of data from the HP-IL device whose address is specified by PRIMARY and SECONDARY, and store the data at *addr* and above (greater addresses). ENTER leaves *addr* on the stack and returns n_2, the actual number of characters received. Executing ENTER requires the HP 82401A HP-IL Interface.

There are two options for termination in addition to the limit of n_1 characters:

- If system flag −23 is set, ENTER will terminate when an End Of Transmission message is received.
- If the argument on the top of the stack is 0, ENTER interprets the second argument on the stack to be a character and will terminate when an incoming character matches this character. This option is effective only when system flag −23 is clear.

EOF	*(E-o-f)*	→ *flag*

Return a true flag if there are no more records in the active file; otherwise, return a false flag. EOF examines the record length of the next record in the file specified by the FIB# in SCRFIB. It assumes that the current pointer into the file is pointing at the next record length and that the file is a text file.

EXECUTE	*(Execute)*	*addr* →

Execute the dictionary entry whose CFA is on the stack.

EXIT	*(Exit)*	→

COMPILE. Terminate execution. Don't use EXIT within a DO loop.

EXPBF	*(Expand-buff)*	n_1 n_2 → *flag*

Expand by n_1 nibbles the general-purpose buffer whose ID# is n_2, and return a true flag; or return a false flag if such a buffer doesn't exist, if the resulting size would exceed 2K bytes, if there is insufficient memory, or if n_1 is negative. n_1 must not exceed FFF.

EXPECT96	*(Expect-96)*	*addr* →

Accept 96 characters from the keyboard (or fewer characters followed by [ENDLINE]), append two null bytes, and store the result at *addr* and above (greater addresses). EXPECT96 also copies the text into the Command Stack.

E^X	*(E-to-the-x)*	→

Raise *e* to the power contained in the X-register. E^X places the result in the X-register and the original value of *x* in the LAST X register.

F*	*(F-times)*	→

Multiply the contents of the X- and Y-registers. F* drops the stack (duplicating T into Z), then places the result in the X-register and the original value of x in the LAST X register.

F+	*(F-plus)*	→

Add the contents of the X- and Y-registers. F+ drops the stack (duplicating T into Z), then places the result in the X-register and the original value of x in the LAST X register.

F-	*(F-minus)*	→

Subtract the contents of the X-register from the contents of the Y-register. F- drops the stack (duplicating T into Z), then places the result in the X-register and the original value of x in the LAST X register.

F.	*(F-dot)*	→

Display the contents of the X-register according to the currently active display format. F. doesn't alter the contents of the X-register.

F/	*(F-divide)*	→

Divide the contents of the Y-register by the contents of the X-register. F/ drops the stack (duplicating T into Z), then places the result in the X-register and the original value of x in the LAST X register.

FABS	*(F-abs)*	→

Take the absolute value of the contents of the X-register. FABS places the result in the X-register and the original value of x in the LAST X register.

FCONSTANT	*(F-constant)*	→

Used in the form: *floating-point number* FCONSTANT *name*

Create a dictionary entry for *name*. When *name* is later executed, the value that was in the X-register when *name* was created is placed in the X-register, lifting the floating-point stack.

FDROP	*(F-drop)*	→

Copy the contents of the Y-register into the X-register, the contents of the Z-register into the Y-register, and the contents of the T-register into the Z-register. The previous contents of the X-register are lost.

FENCE	*(Fence)*	→ *addr*

Return the address of the variable FENCE, which contains the address below which the dictionary is protected from `FORGET`.

FENTER	*(F-enter)*	→

Copy the contents of the Z-register into the T-register, the contents of the Y-register into the Z-register, and the contents of the X-register into the Y-register. The previous contents of the T-register are lost.

FILL	*(Fill)*	*addr un byte* →

Fill memory from *addr* through *addr* + (2*un* − 1) with *un* copies of *byte*. `FILL` has no effect if *un* = 0.

FIND	*(Find)*	$addr_1$ → $addr_2$ *n*

Search the dictionary (in the currently active search order) for the word contained in the counted string at $addr_1$. If the word is found, `FIND` returns the word's CFA (= $addr_2$) and either $n = 1$ (if the word is immediate) or $n = -1$ (if the word isn't immediate). If the word isn't found, `FIND` returns $addr_2 = addr_1$ and $n = 0$.

FINDBF	*(Find-buff)*	*n* → *addr* *n* → false

Return the start-of-data address in the general-purpose buffer whose ID# is *n*, or return a false flag if such a buffer doesn't exist.

FINDF	*(Find-f)*	*str* → *addr* *str* → false

Search main RAM for the file whose name is specified by *str*, and return either the address of the beginning of the file header (if successful) or a false flag (if not). If the specified string exceeds eight characters, `FINDF` considers only the first eight characters.

FIRST	*(First)*	→ *addr*

Return the address of the variable FIRST, which contains the address of the first (lowest addressed) mass storage buffer in the FORTHRAM file.

FIX	*(Fix)*	*n* →

Select fixed-point display mode with n decimal places, $0 \leqslant n \leqslant 11$.

FLITERAL	*(F-literal)*	→

IMMEDIATE, COMPILE. Compile the value x (the contents of the X-register) into the dictionary. When the colon definition is later executed, x will be placed in the X-register, lifting the floating-point stack.

FLUSH	*(Flush)*	→

Unassign all mass storage buffers.

FORGET	*(Forget)*	→

Used in the form: `FORGET` *name*

Delete from the dictionary *name* (which must be in the search order that begins with the `CURRENT` vocabulary) and all words added to the dictionary after *name* (regardless of their vocabulary). Failure to find *name* in the search order that begins with the `CURRENT` vocabulary is an error condition.

FORTH	*(Forth)*	→

Set the CONTEXT vocabulary to FORTH, the name of the first vocabulary in RAM. Because all vocabularies ultimately chain to the FORTH vocabulary, the word `FORTH` can be found regardless of the CONTEXT vocabulary.

FP	*(F-p)*	→

Take the fractional part of the contents of the X-register. `FP` places the result in the X-register and the original value of x in the LAST X register.

FSTR$	*(F-string-dollar)*	→ *str*

Create a string (specified by *str*) that represents the contents of the X-register.

FTOI	*(F-to-i)*	→ *n*

Convert x (the contents of the X-register) to an integer and return it to the data stack. If $|x| >$ FFFFF, an overflow error occurs. `FTOI` takes the absolute value of x, rounds it to the nearest integer, and converts it to a five-nibble value. If x was positive, `FTOI` returns this result; if x was negative, `FTOI` returns the twos complement of this result.

FVARIABLE	*(F-variable)*	→

Used in the form: `FVARIABLE` *name*

Create a dictionary entry for *name*, and allocate eight bytes for its parameter field. Subsequent execution of *name* will return *name*'s PFA. This parameter field will hold the contents of the variable, which must be initialized by the application that creates it.

GROW	*(Grow)*	*n* → *flag*

Enlarge the user dictionary by *n* nibbles and return a true flag; or if there is insufficient memory, return a false flag (without enlarging the dictionary).

H.	*(H-dot)*	*un* →

Display *un* in base 16 as an unsigned number with one trailing blank.

HERE	*(Here)*	→ *addr*

Return the address of the next available dictionary location.

HEX	*(Hex)*	→

Set `BASE` to sixteen.

HOLD	*(Hold)*	*c* →

Insert character *c* into a pictured numeric output string. Used between `<#` and `#>`.

I	*(I)*	→ *n*

Used in the form: `... DO ... I ... LOOP ...`

COMPILE, IMMEDIATE. Return the current value of the `DO`-loop index.

IF . . . THEN	*flag* →

Used in the form: `... IF` *actions* `THEN ...`

COMPILE, IMMEDIATE. Execute *actions* if and only if *flag* is true. `IF ... THEN` conditionals may be nested.

IF . . . THEN . . . ELSE	*flag* →

Used in the form: `... IF` $actions_1$ `ELSE` $actions_2$ `THEN ...`

COMPILE, IMMEDIATE. Execute $actions_1$ if and only if *flag* is true; execute $actions_2$ if and only if *flag* is false. `IF ... ELSE ... THEN` conditionals may be nested within control structures.

IMMEDIATE	*(Immediate)*	→

Mark the most recent dictionary entry as a word to be executed, not compiled, when encountered during compilation.

INTERPRET	*(Interpret)*	→

Interpret the input stream to its end, beginning at the offset contained in `>IN`. The input stream comes from the TIB (if `BLK` contains 0) or from the mass storage buffer containing the *n*th line of the active file (if `BLK` contains *n*.)

IP	*(I-p)*	→

Take the integer part of the contents of the X-register. `IP` places the result in the X-register and the original value of *x* in the LAST X register.

ITOF	*(I-to-f)*	*n* →

Convert *n* into a floating-point number and place it in the X-register, lifting the floating-point stack.

J	*(J)*	→ *n*

Used in the form: `... DO ... DO ... J ... LOOP ... LOOP ...`

COMPILE, IMMEDIATE. Return the index of the next outer loop. Used within nested `DO ... LOOP` structures.

KEY	*(Key)*	→ *c*

Return the low-order seven bits of the ASCII value of the next key pressed. If the key buffer is empty, wait for a key to be pressed.

KILLBF	*(Kill-buff)*	*n* → *flag*

Delete the general-purpose buffer whose ID# is *n*, and return a true flag; or return a false flag if no such buffer exists.

L	*(L)*	→ *addr*

Return the address of the floating-point LAST X register.

LASTX	*(Last-x)*	→

Lift the floating-point stack and copy the contents of the LAST X register into the X-register.

LATEST	*(Latest)*	→ *addr*

Return the NFA of the most recent word in the `CURRENT` vocabulary.

LEAVE	*(Leave)*	→

COMPILE, IMMEDIATE. Skip to the word after the next `LOOP` or `+LOOP`. `LEAVE` terminates the loop and discards the control parameters. Used only within a `DO . . . LOOP` or `+LOOP` construct.

LEFT$	*(Left-dollar)*	str_1 *n* → str_2

Create a temporary string (specified by str_2) consisting of the first *n* characters in the string specified by str_1.

LIMIT	*(Limit)*	→ *addr*

Return the address of the variable LIMIT, which contains the first address beyond the mass-storage-buffer area.

LINE#	*(Line-number)*	→ *addr*

Return the address of the variable LINE#, which contains the number of the line being loaded from the active file (specified by SCRFIB).

LISTING	*(Listing)*	→ *str*

Return *str* specifying the contents of the string variable LISTING, which identifies the file or device to which the assembler will direct its output. `LISTING` can contain up to 20 characters.

LITERAL	*(Literal)*	*n* →

COMPILE, IMMEDIATE. Compile *n* into the word being defined, such that *n* will be returned when the word is executed.

LN	*(Natural log)*	→

Calculate the natural log (base *e*) of the contents of the X-register. `LN` places the result in the X-register and the original value of *x* in the LAST X register.

LOADF	*(Load-f)*	*str* →

Interpret the entire file specified by *str*. If the file cannot be opened for any reason (doesn't exist, wrong type, already opened, etc.), `LOADF` will give the error message `FTH ERR:` *filename* `cannot load`.

LGT	*(Log-ten)*	→

Calculate the common log (base 10) of the contents of the X-register. `LGT` places the result in the X-register and the original value of *x* in the LAST X register.

M*	*(Mixed-multiply)*	n_1 n_2 → d

Return the double-number product *d* of two single numbers n_1 and n_2. All numbers are signed.

M/	*(Mixed-divide)*	d n_1 → n_2 n_3

Divide the double number *d* by the single number n_1, and return the single-number remainder n_2 and the single-number quotient n_3. All numbers are signed.

M/MOD	*(Mixed-divide-mod)*	ud_1 un_1 → un_2 ud_2

Divide the double number ud_1 by the single number un_1, and return the single-number remainder un_2 and the double-number quotient ud_2. All numbers are unsigned.

MAKEBF	*(Make-buff)*	n → $addr$ $ID\#$ true n → false

Create a buffer n nibbles long and return a true flag, the buffer ID#, and the address of the beginning of data area in the buffer; or if unsuccessful (not enough memory, no free buffer ID#s), return a false flag. n cannot exceed 4095_{10}.

MAX	*(Max)*	n_1 n_2 → n_3

Return the greater of n_1 and n_2.

MAXLEN	*(Max-length)*	str → n

Return the maximum length (that is, bytes of memory allotted in the dictionary) for the string specified by *str*.

MIN	*(Min)*	n_1 n_2 → n_3

Return the smaller of n_1 and n_2.

MOD	*(Mod)*	n_1 n_2 → n_3

Divide n_1 by n_2, and return the remainder n_3 with the same sign as n_1.

N@	*(N-fetch)*	$addr$ → n

Return the contents of the nibble at *addr*. The four high-order nibbles of n are zeros.

N!	*(N-store)*	*n addr* →

Store at *addr* the low-order nibble of *n*.

NALLOT	*(N-allot)*	*n* →

Add *n* nibbles to the parameter field of the most recently defined word (regardless of the `CURRENT` and `CONTEXT` vocabularies).

NEGATE	*(Negate)*	*n* → −*n*

Return the twos complement of *n*.

NFILL	*(N-fill)*	*addr un n* →

Fill memory from *addr* through *addr* + (*un* − 1) with *un* copies of the low-order nibble in *n*. `NFILL` has no effect if *un* = 0.

NMOVE	*(N-move)*	$addr_1$ $addr_2$ *un* →

Move *un* nibbles, first moving the nibble at $addr_1$ to $addr_2$ and finally moving the nibble at $addr_1$ + (*un* − 1) to $addr_2$ + (*un* − 1). `NMOVE` has no effect if *un* = 0.

NMOVE>	*(N-move-up)*	$addr_1$ $addr_2$ *un* →

Move *un* nibbles, first moving the nibble at $addr_1$ + (*un* − 1) to $addr_2$ + (*un* − 1) and finally moving the nibble at $addr_1$ to $addr_2$. `NMOVE>` has no effect if *un* = 0.

NOT	*(Not)*	n_1 → n_2

Return the ones complement (true Boolean NOT) of n_1.

NULL$	(Null-dollar)	→ str

Create a temporary string (specified by *str*) in the pad, with maximum length = 80 and current length = 0.

NUMBER	(Number)	addr → d addr →

Examine the counted string at *addr* and convert it into a double number *d*.

- If the string contains a decimal point, NUMBER tries to convert it into a floating-point number and place it in the X-register, lifting the floating-point stack. If the string contains a decimal point but is not a legal floating-point number, a Data Type error occurs.
- If the string does not contain a decimal point, NUMBER tries to convert it into an integer number and return it to the data stack. If the string isn't a legal integer, a FTH ERR: NUMBER not recognized error occurs.

OKFLG	(Okay-flag)	→ addr

Return the address of the variable OKFLG. If the value of OKFLG is 0, the OK { *n* } message is shown when the FORTH system is ready for input; otherwise, the message is suppressed.

ONERR	(On-error)	→ addr

Return the address of the variable ONERR, which contains the CFA of the user's error routine. The value of ONERR is checked when a FORTH-system error occurs. If the value of ONERR is zero, the error is processed by the system's error routine. If the value of ONERR is not zero, control is transferred instead to the user's error routine. The stacks are not reset. The BASIC keywords FORTH and FORTHX set the value of ONERR to zero.

OPENF	(Open-f)	str → t str → str f

Open an FIB for the file whose name is specified by *str*, and store the FIB# into SCRFIB. If successful, OPENF returns a true flag. If the file was empty or there was a problem in opening the file, OPENF returns *str* and a false flag.

OR	*(Or)*	n_1 n_2 → n_3

Return the bit-by-bit inclusive OR of n_1 and n_2.

OUTPUT	*(Output)*	*addr* *n* →

Send *n* bytes, stored at *addr* through *addr* + 2(n − 1), to the HP-IL device whose address is specified by `PRIMARY` and `SECONDARY`. Executing `OUTPUT` requires the HP 82401A HP-IL Interface.

OVER	*(Over)*	n_1 n_2 → n_1 n_2 n_1

Return a copy of the second number on the stack.

PAD	*(Pad)*	→ *addr*

Return the address of the pad, which is a scratch area used to hold character strings for intermediate processing.

PAGESIZE	*(Pagesize)*	→ *addr*

Return the address of the variable PAGESIZE, which contains the number of printed lines per page for the assembler listing. The default value is 56; the minimum value is 8.

PICK	*(Pick)*	n_1 → n_2

Return a copy of the n_1-th entry on the data stack (not counting n_1 itself). For example, `1 PICK` is equivalent to `DUP`, and `2 PICK` is equivalent to `OVER`.

POS	*(Pos)*	str_1 str_2 → *n* str_1 str_2 → false

Search the string specified by str_2 for a substring that matches the string specified by str_1, and return the position of the first character in the matching substring (or a false flag if there is no matching substring).

PREV	*(Prev)*	→ *addr*

Return the address of the variable PREV, which contains the address of the most recently referenced mass storage buffer.

PRIMARY	*(Primary)*	→ *addr*

Return the address of the variable PRIMARY, which specifies an HP-IL address. The valid range for PRIMARY is 0 through 31, and the default value is 1. (The contents of PRIMARY and SECONDARY specify which HP-IL device to use with ENTER and OUTPUT. If system flag −22 is clear, the contents of PRIMARY alone specify a simple address; if system flag −22 is set, the contents of PRIMARY and SECONDARY specify an extended address.)

QUERY	*(Query)*	→

Accept characters from the current keyboard until 96 characters are received or an [END LINE] character is encountered, and store them in the TIB. QUERY sets #TIB to the value of SPAN.

QUIT	*(Quit)*	→

Clear the return stack, set execution mode, and return control to the keyboard. No message is displayed.

R>	*(R-from)*	→ *n*

COMPILE. Remove *n* from the top of the return stack and return a copy to the data stack.

R@	*(R-fetch)*	→ *n*

COMPILE. Return a copy of the number on the top of the return stack.

RADIANS	*(Radians)*	→

Select RADIANS angular mode.

RCL	*(Recall)*	*addr* →

Lift the floating-point stack and place in the X-register the floating-point number found at *addr*.

RDN	*(Roll-down)*	→

Roll down the floating-point stack. `RDN` copies from the T-register into the Z-register, from the Z-register into the Y-register, from the Y-register into the X-register, and from the X-register into the T-register.

RIGHT$	*(Right-dollar)*	str_1 n → str_2

Create a temporary string (specified by str_2) consisting of the last (rightmost) n characters in the string specified by str_1. (`END$` is similar but takes character position, not substring length, for a parameter.)

ROLL	*(Roll)*	n →

Move the nth entry on the data stack (not counting n itself) to the top of the stack. For example, `2 ROLL` is equivalent to `SWAP`, and `3 ROLL` is equivalent to `ROT`.

ROT	*(Rote)*	n_1 n_2 n_3 → n_2 n_3 n_1

Rotate the top three entries on the data stack, bringing the deepest to the top of the stack.

RP!	*(R-p-store)*	→

Reset the return stack to 0 addresses.

RP@	*(R-p-fetch)*	→ *addr*

Return the current value of the return-stack pointer.

RP0	*(R-p-zero)*	→ *addr*

Return the address of the system variable RP0, which contains the address of the bottom of the return stack. (The bottom of the return stack has a greater address than the top.)

RUP	*(Roll-Up)*	→

Roll up the floating-point stack. `RUP` copies from the X-register into the Y-register, from the Y-register into the Z-register, from the Z-register into the T-register, and from the T-register into the X-register.

S!	*(S-store)*	str_1 str_2 →

Store the contents of the string specified by str_1 into the string specified by str_2.

S–>D	*(Sign-extend)*	*n* → *d*

Return a signed double number *d* with the same value and sign as the signed single number *n*.

S0	*(S-zero)*	→ *addr*

Return the address of the bottom of the data stack.

S<	*(S-less)*	str_1 str_2 → *flag*

Return a true flag if the string specified by str_1 is "less than" the string specified by str_2, or a false flag if not. `S<` first compares the ASCII values of the first characters; if they are equal, it then compares the second characters, and so on. `ABC` is defined to be less than `ABCD`.

S<&	*(S-left-concatenate)*	str_1 str_2 → str_3

Append the contents of the string specified by str_2 to the end of the string specified by str_1, and return str_3, the address and length of the resulting string. The address of str_3 is the address of str_1; the length of str_3 is the combined length of str_1 and str_2. If the concatenation would exceed str_1's maximum length, no concatenation occurs and $str_3 = str_1$. Either str_1 or str_2 can specify a temporary string in the pad. The `<` sign indicates that the left string will contain the result of the concatenation.

S=	*(S-equals)*	str_1 str_2 → *flag*

Return a true flag if the two strings are equal, or a false flag if not. S= compares only the current length and contents of the strings, not the maximum length or old contents stored beyond current length.

S>&	*(S-right-concatenate)*	str_1 str_2 → str_3

Append the contents of the string specified by str_2 to the end of the string specified by str_1, and return str_3, the address and length of the resulting string. The address of str_3 is the address of str_2; the length of str_3 is the combined length of str_1 and str_2. If the concatenation would exceed str_2's maximum length, no concatenation occurs and $str_3 = str_2$. Either str_1 or str_2 can specify a temporary string in the pad. The > sign indicates that the right string will contain the result of the concatenation.

SCI	*(Scientific)*	*n* →

Select scientific display mode with $n + 1$ significant digits displayed, $0 \leqslant n \leqslant 11$.

SCRFIB	*(Screen-f-i-b)*	→ *addr*

Return the address of the variable SCRFIB, which contains the FIB# of the currently active file (or 0 if no file is being loaded).

SECONDARY	*(Secondary)*	→ *addr*

Return the address of the variable SECONDARY, which specifies the extended portion of an HP-IL address. The valid range for SECONDARY is from 0 through 31, and the default value is 0. (The contents of PRIMARY and SECONDARY specify which HP-IL device to use with ENTER and OUTPUT. If system flag −22 is clear, the contents of PRIMARY specify a simple address; if system flag −22 is set, the contents of PRIMARY and SECONDARY specify an extended address.)

SHRINK	*(Shrink)*	*n* → *flag*

Shrink the user's dictionary space (and consequently the FORTHRAM file) by *n* nibbles, and return a true flag; or return a false flag if there are fewer than *n* free nibbles in the dictionary.

SIGN	*(Sign)*	*n* →

Insert the ASCII minus sign - into the pictured numeric output string if n is negative. Used between `<#` and `#>`.

SIN	*(Sine)*	→

Calculate the sine of the contents of the X-register, according to the currently active angular mode. `SIN` places the result in the X-register and the original value of x in the LAST X register.

SMOVE	*(S-move)*	*str addr* →

Store at *addr* and above (greater addresses) the characters in the string specified by *str*.

SMUDGE	*(Smudge)*	→

Toggle the smudge bit in the latest definition's name field.

SP!	*(S-p-store)*	→

Reset the data stack to 0 items.

SP0	*(S-p-zero)*	→ *addr*

Return the address of the system variable SP0, which contains the address of the bottom of the data stack. (The address of the bottom of the data stack is greater than the address of the top.)

SP@	*(S-P-fetch)*	→ *addr*

Return *addr*, the address of the top of the data stack before `SP@` was executed.

SPACE	*(Space)*	→

Transmit an ASCII space to the current display device.

SPACES	(Spaces)	n →

Transmit n spaces to the current display device. Take no action for $n \leqslant 0$.

SPAN	(Span)	→ addr

Return the address of the variable SPAN, which contains the count of characters actually read by the last execution of EXPECT96.

SQRT	(Square-root)	→

Calculate the square root of the contents of the X-register. SQRT places the result in the X-register and the original value of x in the LAST X register.

STATE	(State)	→ addr

Return the address of the variable STATE, which contains a non-zero value if compilation is occurring (or zero if not).

STD	(Standard)	→

Select the BASIC standard display format.

STO	(Store)	addr →

Store the contents of the X-register at *addr*.

STR$	(String-dollar)	d → str

Convert the number d into a temporary string in the pad, specified by *str*.

STRING	*(String)*	$n \rightarrow$

Used in the form: *n* `STRING` *name.*

Create a dictionary entry for *name*, allotting one byte for a maximum-length field (value = n), one byte for a current-length field (value = 0), and n bytes for the string characters.

STRING-ARRAY	*(String-array)*	$n_1\ n_2 \rightarrow$

Used in the form: n_1 n_2 `STRING-ARRAY` *name*

Create a dictionary entry for *name*, allotting one byte for the maximum-length field (value = n_1), one byte for the dimension field (value = n_2), and (n_1 + 2) bytes each for n_2 string-array elements. `STRING-ARRAY` fills in the maximum-length (value = n_1) and current-length (value = 0) fields for each string-array element.

Later execution of *n name* will return str_n, the address and current length of the *n*th element of the string array.

SUB$	*(Sub-dollar)*	$str_1\ n_1\ n_2 \rightarrow str_2$

Create a temporary string (specified by str_2) consisting of the n_1th through n_2th characters in the string specified by str_1.

SWAP	*(Swap)*	$n_1\ n_2 \rightarrow n_2\ n_1$

Exchange the top two entries on the data stack.

SYNTAXF	*(Syntax-f)*	*str* → *flag*

Return a true flag if the string specified by *str* is a valid HP-71 file name, or return a false flag if not. If the specified string exceeds eight characters, `SYNTAXF` checks only the first eight characters.

T	*(T)*	→ *addr*

Return the address of the floating-point T-register.

TAN	*(Tan)*	→

Calculate the tangent of the contents of the X-register, according to the currently active angular mode. TAN places the result in the X-register and the original value of x in the LAST X register.

TIB	*(T-i-b)*	→ *addr*

Return the address of the terminal input buffer. The terminal input buffer can hold up to 96 characters.

TOGGLE	*(Toggle)*	*addr* n_1 →

Replace n_2 (the contents at *addr*) with the bit-by-bit logical value of (n_1 XOR n_2).

TRAVERSE	*(Traverse)*	$addr_1$ n → $addr_2$

Return the address of the opposite end (length byte or last character) of a definition's name field.

- If $n = 1$, $addr_1$ is the address of the length byte, and $addr_2$ is address of the last character.
- If $n = -1$, $addr_1$ is the address of the last character, and $addr_2$ is the address of the length byte.
- If n doesn't equal 1 or -1, $addr_1 = addr_2$.

TYPE	*(Type)*	*addr* n →

Transmit n characters, found at *addr* through $addr + (2n - 1)$, to the current display device. TYPE transmits no characters for $n \leqslant 0$.

U.	*(U-dot)*	*un* →

Display *un* (according to BASE) as an unsigned number in a free-field format with one trailing blank.

U<	*(U-less-than)*	un_1 un_2 → *flag*

Return a true flag if $un_1 < un_2$, or return a false flag if not.

UM*	*(U-m-times)*	$un_1 \ un_2 \rightarrow ud$

Return the double-number product *ud* of two single numbers un_1 and un_2. All numbers are unsigned.

UM/MOD	*(U-m-divide-mod)*	$ud_1 \ un_1 \rightarrow un_2 \ un_3$

Divide the double number ud_1 by the single number un_1, and return the single-number remainder un_2 and the single-number quotient un_3. All numbers are unsigned.

USE	*(Use)*	$\rightarrow addr$

Return the address of the variable USE, which contains the address of the next mass storage buffer available for use.

VAL	*(Val)*	$str \rightarrow d$ $str \rightarrow$

Convert the string specified by *str* into a number.

- If the string contains a decimal point, `VAL` tries to convert it into a floating-point number and place it in the X-register, lifting the floating-point stack. If the string contains a decimal point but is not a legal floating-point number, a `Data Type` error occurs.
- If the string does not contain a decimal point, `VAL` tries to convert it into an integer number and return it to the data stack. If the string is not a legal integer, a `FTH ERR: VAL not recognized` error occurs.

VARIABLE	*(Variable)*	$\rightarrow$

Used in the form: `VARIABLE` *name*

Create a dictionary entry for *name*, allotting five nibbles for its parameter field. Later execution of *name* will return *name*'s PFA. This parameter field will hold the contents of the variable, which must be initialized by the application that created it.

VARID	*(Var-i-d)*	$\rightarrow addr$

Return the address of the variable VARID, in which the assembler stores the ID# of the general-purpose buffer that it uses. If the value of VARID is non-zero, the FORTH system will preserve the buffer with that ID#.

VOCABULARY	(Vocabulary)	→

Used in the form: `VOCABULARY` *name*

Create (in the `CURRENT` vocabulary) a dictionary entry for *name* that begins a new linked list of dictionary entries. Later execution of *name* will select *name* as the `CONTEXT` vocabulary. (Vocabularies are discussed in section 2.)

WARN	(Warn)	→ addr

Return the address of the variable WARN. If WARN contains a non-zero value, compiling a new word whose name matches an existing word causes a *name* `isn't unique` message to be displayed; if WARN contains 0, the message is suppressed.

WIDTH	(Width)	→ addr

Return the address of the variable WIDTH, which determines the maximum allowable length for the name of a word. The valid range for WIDTH is from 1 through 31.

WORD	(Word)	c → addr

Receive characters from the input stream until the non-zero delimiting character *c* is encountered or the input stream is exhausted, and store the characters in a counted string at *addr*. `WORD` ignores leading delimiters. If the input stream is exhausted as WORD is called, a zero-length string results.

X	(X)	→ addr

Return the address of the floating-point X-register.

X<>Y	(X-exchange-y)	→

Exchange the contents of the X- and Y-registers.

X#Y?	**X<=Y?**	Floating-point Comparisons	→ *flag*
X<Y?	**X=0?**		
X=Y?	**X>=Y?**		
X>Y?			

Compare the contents of the X- and Y-registers, and return a true flag if the test is true or a false flag if not. The tests don't alter the contents of the X- and Y-registers.

XOR	*(X-or)*	n_1 n_2 → n_3

Return the bit-by-bit exclusive OR of n_1 and n_2.

X^2	*(X-squared)*	→

Calculate the square of the contents of the X-register. `X^2` places the result in the X-register and the original value of x in the LAST X register.

Y	*(Y)*	→ *addr*

Return the address of the floating-point Y-register.

Y^X	*(Y-to-the-x)*	→

Raise the contents of the Y-register to the power contained in the X-register. `Y^X` places the result in the X-register and the original value of x in the LAST X register.

Z	*(Z)*	→ *addr*

Return the address of the floating-point Z-register.

[	*(Left-bracket)*	→

IMMEDIATE. Suspend compilation. Subsequent text from the input stream will be executed.

[']	*(Bracket-tick)*	→

Used in the form: `:` *name*$_1$ . . . `[']` *name*$_2$ . . .

COMPILE, IMMEDIATE. Compile the CFA of *name*$_2$ as a literal. An error occurs if *name*$_2$ is not found in the currently active search order. Later execution of *name*$_1$ will return *name*$_2$'s CFA.

[COMPILE]	*(Bracket-compile)*	→

Used in the form: . . . `[COMPILE]` *name* . . .

IMMEDIATE, COMPILE. Compile *name*, even if *name* is an `IMMEDIATE` word.

]	*(Right-bracket)*	→

Resume compilation. Subsequent text from the input stream is compiled.

Subject Index

Page numbers in **bold** type indicate primary references; page numbers in regular type indicate secondary references.

BASIC Keywords by Category

This list shows all BASIC keywords by functional category. All BASIC keywords and their defininitions appear in appendix C, in alphabetic order.

Keyword	Description
BASIC to FORTH	
FORTH	Transfers HP-71 operation to the FORTH environment.
FORTH$	Returns to a BASIC string variable the contents of a string in the FORTH enivronment.
FORTHF	Returns to a BASIC numeric variable the contents of the FORTH floating-point X-register.
FORTHI	Returns to a BASIC numeric variable the value on the top of the FORTH data stack.
FORTHX	Executes a FORTH command string.
Editor	
DELETE#	Deletes one record from a text file.
EDTEXT	Invokes the text editor.
FILESZR	Returns the number of records in a text file.
INSERT#	Inserts one record into a text file.
MSG$	Returns the message string corresponding to a specified error number.
REPLACE#	Replaces one record in a text file.
SCROLL	Scrolls the display and waits for a key to be pressed.
SEARCH	Finds a string in a text file.
Remote Keyboard	
ESCAPE	Adds or modifies an escape-sequence key specification in the key-map buffer.
KEYBOARD IS	Assigns one HP-IL device to be used as an external keyboard.
RESET ESCAPE	Purges any existing key-map buffer created by the ESCAPE keyword.

FORTH Words by Category

This list shows all FORTH words by functional category. Some words appear in more than one category. All FORTH words and their defininitions appear in appendix D, sorted by name in ASCII order.

General

Dictionary Management

ALLOT
CONTEXT
CURRENT
DEFINITIONS
FENCE
FORGET
FORTH
GROW
HERE
NALLOT
PAD
SHRINK
VOCABULARY

System

>BODY
?STACK
ABORT
ABORT"
ASSEMBLE
BYE
DECIMAL
DEGREES
DEPTH
EXECUTE
FIND
HEX
LATEST
QUIT
RADIANS
TIB
TOGGLE
TRAVERSE

BASIC System Access

BASIC$
BASICF
BASICI
BASICX

Control Structures

BEGIN . . . UNTIL
BEGIN . . . WHILE
. . . REPEAT
CASE . . . OF. . . ENDOF
. . . ENDCASE
DO . . . +LOOP
DO . . . LOOP
IF . . . THEN
IF . . . THEN
. . . ELSE
LEAVE

Memory

!
+!
4N@
?
@
C!
C@
C@+
CMOVE
CMOVE>
FILL
N!
N@
NFILL
NMOVE
NMOVE>
RCL
S!
SMOVE
STO

Interpretation

'
(
INTERPRET

Return Stack

>R
I
J
R>
R@
RP!
RP0
RP@

Defining Words

:
;
CONSTANT
CREATE
EXIT
FCONSTANT
FVARIABLE
STRING
STRING-ARRAY
VARIABLE

Compilation

,
?COMP
C,
COMPILE
DLITERAL
DOES>
FLITERAL
IMMEDIATE
LITERAL
SMUDGE
STATE
[
[']
[COMPILE]
]

Assembler

ASSEMBLE
LISTING
PAGESIZE

Files

File Manipulations

+BUF
ADJUSTF
BLOCK
CLOSEALL
CLOSEF
CREATEF
EOF
FINDF
FLUSH
LOADF
OPENF
SYNTAXF

General Purpose Buffers

CONBF
EXPBF
FINDBF
KILLBF
MAKEBF

Input/Output

Constants

- 0
- 1
- 2
- 3
- BL

Numeric-Input Conversion

- BASE
- CONVERT
- DIGIT
- NUMBER

Numeric Output

- .
- .S
- BASE
- D.
- D.R
- F.
- H.
- U.

Number Formatting

- #
- #>
- #S
- <#
- ENG
- FIX
- HOLD
- SCI
- SIGN
- STD

Character Input

- ?TERMINAL
- COUNT
- ENCLOSE
- EXPECT96
- KEY
- QUERY
- WORD
- 'STREAM

Character Output

- -TRAILING
- ."
- .(
- CR
- EMIT
- SPACE
- SPACES
- TYPE

HP-IL

- ENTER
- OUTPUT
- PRIMARY
- SECONDARY

Arithmetic

Single Length

- *
- */
- */MOD
- +
- -
- /
- /MOD
- 1+
- 1-
- 2*
- 2+
- 2-
- 2/
- 5+
- 5-
- ABS
- FTOI
- MOD
- NEGATE

Double Length

- D+
- D-
- DABS
- DNEGATE
- S->D

Mixed Length

- M*
- M/
- M/MOD
- UM*
- UM/MOD

Floating Point

- 1/X
- 10^X
- ACOS
- ASIN
- ATAN
- CHS
- COS
- E^X
- F*
- F+
- F-
- F/
- FABS
- FP
- IP
- ITOF
- LGT
- LN
- SIN
- SQRT
- TAN
- X^2
- Y^X

Logical

- AND
- NOT
- OR
- XOR

Stack

Single Length

- DROP
- DUP
- OVER
- PICK
- ROLL
- ROT
- S0
- SP!
- SP0
- SP@
- SWAP

Double Length

- 2DROP
- 2DUP
- 2OVER
- 2SWAP

Floating Point

- FDROP
- FENTER
- LASTX
- RDN
- RUP
- X<>Y

Comparisons

Single Length

- 0<
- 0=
- 0>
- <
- <>
- =
- >
- ?DUP
- MAX
- MIN
- U<

Double Length

- D<

Floating Point

- X#Y?
- X<=Y?
- X<Y?
- X=0?
- X=Y?
- X>=Y?
- X>Y?

String

- S<
- S=

User Variables

- #TIB
- >IN
- BASE
- BLK
- CONTEXT
- CURRENT
- FIRST
- L
- LIMIT
- LINE#
- LISTING
- OKFLG
- ONERR
- PAGESIZE
- PREV
- PRIMARY
- SCRFIB
- SECONDARY
- SPAN
- STATE
- T
- USE
- VARID
- WARN
- WIDTH
- X
- Y
- Z

String Words

- "
- ASC
- CHR$
- CRLF
- END$
- FSTR$
- LEFT$
- MAXLEN
- NULL$
- POS
- RIGHT$
- S<
- S<&
- S=
- S>&
- STR$
- SUB$
- VAL

HEWLETT PACKARD

Portable Computer Division
1000 N.E. Circle Blvd., Corvallis, OR 97330, U.S.A.

European Headquarters
150, Route Du Nant-D'Avril
P.O. Box, CH-1217 Meyrin 2
Geneva-Switzerland

HP-United Kingdom
(Pinewood)
GB-Nine Mile Ride, Wokingham
Berkshire RG11 3LL

82441-90001 English

Printed in Singapore 4/84