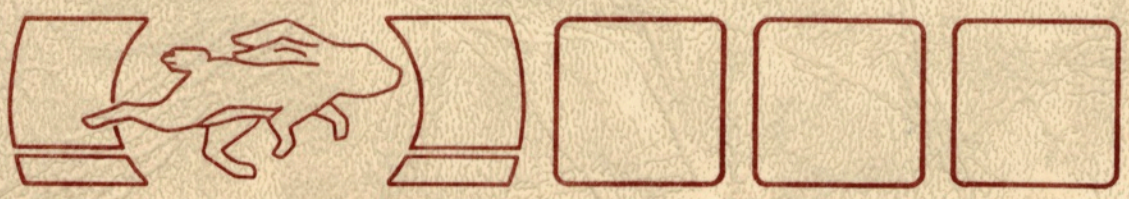
# BASIC
# THE HP-71

Up and Running in
CALC Mode, BASIC and Assembly Language

by Richard E. Harvey



**Preface**

We can thank Hewlett-Packard for the HP-71, but more so, the user community for demonstrating to HP that there is a need for this powerful, compact tool. This book is dedicated to those users.

**Introduction**

In the early 1970's we had 15 pound "portable" calculators like the HP-46. 1972 brought with it the HP-35, the worlds first, handheld scientific calculator. 1974 brought us the programmable portable HP-65 with 100 step memory and built-in card reader. At the close of the 70's the HP-41, an engineering tour de force, became as much at home on a surveyor's belt, a student's desk or floating in zero-G. Each of these machines, and others between them share a design philosophy and RPN (Reverse Polish Notation). Post-fix math has been a way of life for a generation.

The mid-80's saw Hewlett-Packard looking to expand their market and exploit the technological advances of the computer boom. The 71 has retained the HP design philosophy, but traded in RPN for greater speed, memory, and an open, expandable operating system with an advanced dialect of BASIC.

There are two camps of HP supporters: those who think that RPN is the only way to run a calculator, and those who think that HP BASIC is the only way to run a computer. Until the HP-71, those two factions barely knew each other existed.

Well, calculator factions, meet HP BASIC! This is not the checkbook-balancing, Pong-playing, beginner only language found on home computers, but an advanced mathematical tool with several hundred highly optimized functions. And, if you still want it, RPN is just a ROM away.

This book is designed to introduce the novice or experienced user to the HP-71, CALC mode, and HP BASIC, but not leave him there too long. We'll discuss the 71, not as a mystical beast with powers known only by an elite few, but as a learning and working tool. We will also introduce the internal design of the 71 and Assembly Language programming, and support these discussions with a number of tables and charts.

Much of the material covered in volumes one and two of the HP-71 Internal Design Specification is paraphrased here making the purchase of those books (at about $100) unnecessary except for the most devoted Assembly language user.

Personal computers were used for playing Star Wars long before dBase II or Lotus 1-2-3 were even contemplated. It was a well kept secret for quite a while that computers are fun. Let's get some work done, but, let's make it an enjoyable experience. That's the attitude of this book.

This book was formatted and printed by an HP-71 using a program written by the author.


Box 5695
Glendale, Arizona 85312 USA

Second Printing: April 1986

# THE BASIC HP-71

## Up and Running in CALC Mode, BASIC and Assembly Language

## CONTENTS


Introduction
  How to use this book . . . . . 1
  Why Program? . . . . . 1
  Warning about Damaging Your 71 . 2
Getting Started . . . . . 3
  Basic Keyboard Math . . . . . 3
    Mathematical Precedence . . . 4
    Parentheses . . . . . 4
    RES Register . . . . . 4
    Spaces . . . . . 5
    Multi-Statement Lines . . . . 6
    Strings . . . . . 6
    Calculator Variables . . . . . 7
      Variable Names . . . . . 8
      Types of Variables . . . . . 8
The HP-71 . . . . . 11
  Central Processing Unit . . . 11
  Clock Speed . . . . . 11
  Hexadecimal Numbers . . . . . 11
  Memory . . . . . 12
  :PORTS . . . . . 12
Environments . . . . . 13
  Sub-Programs . . . . . 14
    CALL Cautions . . . . . 14
  Modes . . . . . 15
  Command Stack . . . . . 15
Often Used Commands . . . . . 15
Accessories . . . . . 20
  Data Storage . . . . . 20
  Card Reader . . . . . 20
  HP-IL Mass Storage . . . . . 20
  Printers . . . . . 22
  Other HP-IL Devices . . . . . 23
The File Chain . . . . . 24
  Finding Files . . . . . 24
    File Header Structure . . . 25
  Data Files . . . . . 25
    Creating the data File . . . 26
    The File Pointer . . . . . 27
    Storing data . . . . . 27
    Closing the data File . . . 28
  BASIC Files . . . . . 29
  BIN Files . . . . . 29
  FORTH Files . . . . . .30
  LEX Files . . . . . 30
  DATA Files . . . . . 31
  TEXT Files . . . . . 32
  KEY Files . . . . . 33
  SDATA Files . . . . . 34
CALCulating with the HP-71 . . . . . 35
  Long Formulas . . . . . 35
  Inside CALC Mode . . . . . 37
  "CALCAID" Program . . . . . 37
  User Defigned Functions . . . . . 38
Basic BASIC . . . . . 39
  HP BASIC . . . . . 40
HP-71 BASIC Programming . . . . . 43
  "ACYDUCY" Program . . . . . 43
  "INCAT" SubProgram . . . . . 44
  Interpreted BASIC . . . . . 45
  Tokens . . . . . 47
  "DECIDE" Program . . . . . 47
BASIC Programming Hints . . . . . 49
PEEK$s & POKEs . . . . . 58
  Strings in SDATA Files . . . . . 63
Converting from other BASICs . . . . 65
Assembly Language Introduction . . . 71
  Parsing and Decompiling . . . . . 78
  The Math Stack . . . . . 78
  "SAMPLE" LEX File . . . . . 86
Communicating with RS232 . . . . . 88
  RS-232 Cable . . . . . 88
  Setting up the Interface . . . . . 89
  External Keyboards . . . . . 90
  Exchanging Files . . . . . 93
  Display Devices . . . . . 94

  Decimal/Hex/Binary/ASCII Table . . 95
  HP-71B Memory Map . . . . . 98
  System RAM . . . . . 99
  HP-71B Keymap . . . . . 100
  System Flags . . . . . 101
  Assembler Instruction Set . . . . 102
  Minimum LEX File Requirements . . 104
  Assembler Entry Points . . . . . 105

## How to use this book

This book is laid out in a refrence style; it isn't necessary to read it from start to finish. Most of the charts and tables are at the back of the book. While this may cause some page shuffling at first reading, it makes this important refrence material easier to find later without having to wade through several thousand words of flowery prose.

Most subjects are given a cursory introduction first, then in greater detail. To keep from having to dart back and forth, most material on a subject is in the same section. For example, TEXT files are discussed in an introductory manner, then using them in BASIC, then their internal structure (down to the nibble level), in the same section. Each section begins with a main heading in **BOLD** type, and most topics have a layout generally as follows:

```
Main Topic
What it does
Application
Fanatical Detail
```

The obvious disadvantage of this system is that, if you haven't used HP BASIC or your 71 much yet then some place about the middle of the third part it'll look like it's drifting off into a foreign language. The glossary beginning on page 346 of the HP-71 Refrence Manual will aid in the translation to English.

Examples are enclosed in boxes and are discussed in the text immidiately above the box. Since the box may be at the end of a discussion, text which follows the box will not necessarily relate to the example.

### Why Program?

Plug in a ROM, fire it up, and there you have it: instant solutions. Many people buy a 71 with that single intention in mind. Then they think, "what else can I do with the data from the Finance ROM" or "If only my 71 could...". With an understanding of BASIC comes, not only the ability to write programs, but also the ability to make better use of those we already have. This personal computer becomes more personal when we learn to use it our way, not sombody elses; we can adapt the machine to our needs, not the other way around.

A major use of personal programs is to solve little problems: To copy a list of files between Discs, solve those math equations we always use, or set-up the printer for compressed type. These personal solutions are needed every day. And, once we're comfortable with these solutions we can do just about anything, for a large program is a group of smaller solutions.

With this knowledge we can rewrite our favorite HP-41 or BASIC programs from other machines to run on our 71. Or modify distributed programs to suit our own needs.

Lest we forget, programming (and it's associated discoveries) is fun! Working through a program provides as much pleasure as having it done. And, a program is never really finished.

### The Hacker

At one time "Hacker" was a positive term. It meant the person who saw the problem then went at it with dedication, understanding and intuition to solve it. The image is the lone worker with a bag of M&M Peanut (not Plain) candy, spending what should be sleeping hours, coding and testing programs. Romantic perhaps, but there are other ways. We hope to make a case for another work style, a little more sleep, and Hershey Bars.

### Limits

**Warning about Damaging your HP-71**

There are no wrong keystroke combinations. The 71 may beep, scold you with an error message or even display the ominous sounding "Memory Lost" and reset itself, but there is no way to damage it by pressing keys. A controlled crash (INIT 3) is often used to purge unwanted files and restore the machine to start-up conditions. The 71 was designed to perform this operation, there is no damage. INIT 1 will usually free the 71 from any "stuck" situation. INIT 1 is performed by pressing the ON key and / key at the same time, then pressing **ENDLINE** at the prompt.

No operations described in this book will cause a crash (freeze the display, or keyboard or display "that" message) if the directions are followed exactly. The operations least forgiving of improper actions are POKEing and testing Assembly Language routines. It is suggested that you make backup copies of all important files before trying the more esoteric operations.

A very rare crash won't respond to INIT 3. There are two courses of action: Pull out the batteries (and un-plug the AC cord) and modules and hold the down **ON** key for about 30 seconds. If that method doesn't revive the poor confused beast then leave the 71 sitting (without battery) overnight until the circuits are definately discharged. HP has been known to suggest opening the card reader compartment (remove the card reader if present) and shorting together the two taller pins on either end of the row of pin connectors with a paper clip for just a second, though this is not meant to be a recommendation for this method. The one time the author has seen the need for this drastic fix was on a day of 105 degrees and under 10% humidity.

**WorkBook71**

A program package for the HP-71 is available by this author. **WorkBook71** includes Virtual Memory Spreadsheet, File manager, Data Format converter, Full Screen Text Editor and Text Formatter. The Text Formatter was used to print this book. Contact the author for information.

---

When first turned on, the 71 is in BASIC Mode. The LCD display is blank except for the BASIC prompt character ">" and the flashing cursor. Whatever you type will be accepted, without question, until you press **ENDLINE**. Pressing **ENDLINE** tells the computer that you have entered a complete line and now it is time for it to do whatever it is you have entered, and (possibly) display a result. In fact, just about everything we do with the 71 terminates with **ENDLINE**. Since this is "a given", this book will rarely even mention that you should press **ENDLINE**. The **ENDLINE** key has the same effect as **RTN, RETURN, EOL, ENTER** or bent arrow keys on other computers, but NOT the same effect as **ENTER** on RPN calculators.

The first part of this chapter will take advantage of a feature of HP BASIC: If an expression is not explicitly assigned to a variable then it is implied that the result of the expression will be displayed only. This feature of HP BASIC will become self-evident in about four pages. If you would like to have the result of each example printed then preceed each with the PRINT keyword.

### It Just Beeps

It is likely that the first time you turned on your 71 and entered something from the keyboard it displayed an error message then beeped. The messages aren't intended to intimidate, but as an aid in using the computer. As with a calculator, instructions have to be entered exactly as required for the 71 to understand them. Unlike a calculator, the 71 can do several hundred things. A keyboard with hundreds of keys is impractical, hence the command line and it's series of messages.

This set of rules we follow when entering commands is called syntax. The 71 follows these rules of syntax when interpreting commands. This is called parsing. Some ambiguity in syntax is accepted in speech and writing because people can infer the meaning of a sentence from context, inflection, previous knowledge and such. The computer, on the other hand, takes everything pretty much on face value; whatever we tell it, it tries to do. If the 71 can't understand the command (it won't parse) then it will definitely let us know. Understanding syntax and how the computer parses is as important in CALC mode as when writing a program. All commands and syntax guidelines are listed in the **HP-71 Refrence Manual.**

### BASIC Keyboard Math

When doing business as a BASIC mode calculator the 71 works in True Algebraic fashion. Each operation is performed using rules of mathematical precedence, and a final result is returned. The operator is placed between the arguments (numbers). CALC mode displays intermediate values as operations are completed, but let's consider BASIC mode calculations in this chapter.

```
Algebraic Calculator:
      2   X   3   =

Hewlett-Packard RPN Calculator:
      2   ENTER   3   *

HP-71 BASIC Mode:
      2   *   3   ENDLINE
```

Each of these methods has one thing in common: a single keystroke tells the computer that you want it to process the data and return a result. The conventional dollar-ninty-eight Algebraic calculator uses the = equals key and the RPN calculator uses the * key. The 71 is similar to the Algebraic calculator in that the operator (in this case the * which is computerese for multiplication, "X" is just that, an "X") is placed between the operands (the numbers), however **ENDLINE** is used to tell it that we are ready for it to get to work.

All calculators (and computers) have a stack: a place to store intermediate results in a calculation. Without a stack there would be no way to do anything which involves comparing two numbers. In the above example, the Algebraic calculator

places the first number on the stack when we press + then places the second number on the stack and performs the mathematical operation when we press =. The RPN calculator is friendlier to use because we can more readily control the values on the stack and the order in which the operations are performed. An advanced True Algebraic calculator can evaluate an expression using parentheses to designate which of a series of operations will be evaluated first. In this way, the order of calculation determines the intermediate results. We work with mathematical expressions instead of entering operations from the "inside out" to preserve the order of precedence.

## Mathematical Precidence

Algebraic calculations with the 71 are evaluated using the following set of rules regarding precidence (which operation is performed first).

```
( ... )                              Nested Parentheses
^                                    Exponentiation
NOT   unary +   unary -              Operation on one operand (X=-X)
SIN   RND   COS   FACT               Functions
*   /   DIV   %
+   -   &                            Operations on two operands (X=X-Y)
<   =   >   #   ?   <=   >=   <>     Relational operators
AND
OR   EXOR
```

Entered without parentheses, an expression will be performed in the sequence above. Operations of the same level will be completed from left to right.

```
2+3*4-5 = 9
```

Notice that we didn't begin with DISP or PRINT, the 71 assumes that we wanted to display the answer because it wasn't being assigned to a variable (we'll discuss variables in a few minutes). Here, multiplication will be performed first. Since addition and subtraction are on the same level of precidence, they will be evaluated from left to right.

```
3*4 = 12    ,    2+12 = 14    ,    14-5 = 9
```

### Parentheses

Placing parts of an expression within sets of parentheses tells the 71 which operations to perform first. Any number of parentheses may be used. In fact, if you are not sure of how an expression will be evaluated, add extra parentheses for clarity; any extra will be ignored.

```
(2+3)*4-5 = 15    ,    2+3*(4-5) = -1    ,    (2+3)*(4-5) = -5
```

### RES Register

The result of the last mathematical expression is stored in the Result register. This is done regardless of if the value is assigned to a variable or just displayed. This is quite helpful when an intermediate result is needed, then that value is to again be used in the following expression. The RES keyword is used to recall the contents of the register. Remember that RES changes with each expression.

```
2+3*4-5
 9

RES*2
 18
```

**Other Operations**

Not everything we do returns a value, and the flexible way the 71 does these non-calculator operations is what sets it apart from a calculator. Operations return a value are called functions, those which do not are statements. A statement tells the computer to do something. For instance, BYE is a statement which turns off the 71. While the act of turning itself off is a function (as sleeping is a function people perform), it does not return a value so it is deemed a statement. Several system statements and functions are listed below.

**Upper / Lowercase**

Commands can be entered in either upper or lowercase, the 71 automatically converts all commands to uppercase. In this book we will usually demonstrate commands in UPPERCASE to help distinguish them from text. Either of the following is acceptable.

```
beep
BEEP
```

**Spaces**

Spaces are entered between most statements for clarity. As a convenience, these spaces can often be omitted when entering commands. The 71 will use it's dictionary of commands and syntax and try to evaluate the expression. Consider the following:

```
CALLIOPE                    XYZ
BEEPER                      MEMORY
```

In each of these examples the 71 will try to parse (interpret the meaning of the expression) differently based on the context of the expression. Each example will result in a different type of error. In the first case the computer will look for the keyword CALLIOPE. It won't find it so it finds the next closest word which is CALL which is used to call subprograms. So it looks for a subprogram named "IOPE", for our purposes we'll assume that "IOPE" is not the name of a program. This was a complete operation and it just didn't work, so the 71 beeps and displays the error message:

```
ERR:Sub Not Found
```

If you missed the message then press **g** (the blue shift key) then hold down **ERRM** (a shifted function of the **SPC** key) to see it again. A message explaining the most recent error can always be recalled this way.

The second example, BEEPER will be interpreted differently than the first. The closest keyword is BEEP, which can specify a frequency (tone) and number of seconds to beep. BEEPER will cause:

```
ERR:Excess Chars
```

Then the original line will return to the display with the cursor placed at the first character in the line which the 71 did not recognize as part of a valid expression. In this case the cursor is after the third E because E is a valid variable name, and ER is not.

Let's modify BEEPER so that it will parse correctly. BEEP expects either nothing, a single parameter, or two. If there are two parameters then they are separated by a comma. Any one of the three following examples is acceptable.

```
BEEP          BEEP500          BEEP500,1
```

The third problem we foisted on the 71 was XYZ which is not a complete expression, and does not contain a complete keyword. Sinced there was no keyword found the 71 assumes that we really meant to see the value of variable X and whatever followed was a slip of the keyboard. Following the usual beep and error message the original line will be returned to the display with the cursor over the Y because, again, it is the

first character the 71 didn't recognize as part of a valid expression.

In the fourth example, the 71 will return a value without an error (finally). But, is it the result we wanted? MEMORY is not a command in the 71's repetoire. But, it did find MEM a function which returns the amount of memory currently available. Then it looked for an operator that could follow the function; it found OR which does a logical OR of two values. Since OR requires a second operator for it's comparison, the 71 looked for a mathematical expression. It found Y which referred to the variable Y. This is a complete expression, so the computer evaluates it and returns a result. The answer is either the number one, if Y has no value, or zero if Y contains a non-zero value. MEMORY is interpreted as:

```
MEM OR Y
```

This is hardly a useful expression. But it did not cause an error because it evaluated to an expression which the 71 could successfully calculate.

As you can see, the computer will be able to detect and help us with syntatical errors, but is of little help with logical errors.

**Mult-statement lines**

Several expressions can be evaluated in one session by the 71 by separating them with the @ (commercial at sign).

```
2+3*4-5 @ (2+3)*4-5
        9
        15
```

This will cause the first to be evaluated then displayed for the length of the DELAY setting, then the second expression. The two expressions may also be evaluated and displayed at the same time by using the ; semicolon. A semicolon at the end of the expression tells the 71 that we are not finished displaying on that line, more will follow.

```
2+3*4-5;(2+3)*4-5;2+3*(4-5)
    9  15 -1
```

Each number is displayed with either a leading space or a minus sign, and followed by a space. In this way numbers are displayed without runningintoeachother.

**Strings**

Much of computing involves text as well as numbers. A string is a group of characters (letters, numbers, spaces...) enclosed between a pair of quotation marks. Either single (') or double (") quotes may be used, but both ends must match. & adds two strings together, and [ ] square brackets are used to extract just a portion of a string (called a substring). HP BASIC has relatively few string functions (when compared to Microsoft) because the versatility of brackets makes them unnecessary. One or two parameters can be specified within the brackets.

```
[1,4] positions 1 through 4 only
[3,7] positions 3 through 7 only
[5]   from position 5 through the end of the string
```

We'll use the example of VER$ which returns the version of the 71's operating system as well as that of many plug-in accessories. As with all functions which return a string, the last character in VER$ is $ a dollar sign. "$" is usually pronounced as dollar or string. For example, TIME$ is pronounced "time dollar" or "time string". Dollar is probably the preffered pronunciation when describing a problem over the telephone.

Whenever a function contains a $ then it can only be used with string arguments. Try using string expressions with numbers and experience a whole new world of error messages.

```
VER$                VER$[3,4]          "This is the "&VER$[1,4]
HP71:1BBBB          71                 This is the HP71
```

Notice that, unlike number functions, no extra leading or trailing spaces were added. Again, a semicolon can be used to display more than one string, or even strings and numbers, on the same line:

```
MEM;VER$[1,4];2+3*4-5
```

Numbers and strings are two quite different types of information. Therefore it takes an extra step or two to move information between the two. Let's look at some of the string operations.

**CHR$, NUM**

Each character can be expressed as either the character itself, or as a numeric value representing the character. Characters are a single byte, and a byte can have a value of from zero to two hundred fifty five. Therefore, there are 256 possible characters, not all of which can be displayed. NUM takes a string and returns the numeric value of the first character. CHR$ ends with $ since it returns a string. CHR$ is the opposite of NUM in that it will accept a value of 0-255 and returns a single character. A table of ASCII / HEX / DECIMAL / BINARY conversions is listed in the back of this book. Let's use the examples of "A" which is ASCII 65 and "%" which is ASCII 37:

```
NUM("A")            NUM("%")
 65                  37

CHR$(65)            CHR$(37)
A                   %
```

**VAL, STR$**

An incredibly powerful function shared by the 71 and some larger HP computers is VAL. It evaluates a string as a mathematical and returns a numerical result. While not exactly the opposite of VAL (it can't restore a formula once it is gone), STR$ turns a number into it's string equivalent. STR$ follows the current FIX setting, and truncates the fractional part or adds zeros as needed. These two functions are the two main methods for exchanging data between strings and numbers.

```
VAL("2+3*4-5")
STR$(9)
```

## Calculator Variables

Most calculators use data registers for storage of calculator and program data. We'll define a register as a fixed size, pre-defined place to store data. The 71, as with other computers, uses variables. We make this distinction because variables, unlike registers, can be of various sizes and types and possibly move about in memory as ROMs or more memory are added to the computer. Unlike a register, a variable does not exist (and does not use any memory) until it is actually needed.

In BASIC parlance and True Algebraic mathematics a variable is a symbol which represents a value. The symbolic label used for each variable represents actual individual location in memory reserved for that (if you will...) pigeon hole. Let's assign our, now tiring, example to variable X. The keyword LET is optional (and rarely actually used) but is included here for clarity.

```
LET X=2+3*4-5
```

Now X contains the result of the expression. This can be proven by entering:

```
X
 9
```

This statement says "Look up the value of X and display it. We could use a boolean operator to do a comparison and prove that X does contain the result of that formula. In this boolean comparison the value is represented first to help the 71 to interpret the statement; had the variable name been given first then it would merely have been assigned the value. Another alternative to insure that a boolean comparison will be made is to preface the expression with DISP. This consideration is only important when using = because there is no chance for ambiguity with other boolean operators.

```
9=X                    DISP X=9
 1                      1
```

The result of the argument will be either one which means the argument is true, or zero would have been returned had our argument proven false. (whew!).

Previously we have been displaying results using an implied DISP. This means that the results of an expression will be displayed unless the expression begins with a variable assignment, in which case the result will be assigned to that variable and nothing will be displayed. Remembering the optional keyword LET will help.

**Symbolic Variable Names**

The letters A through Z, as you know, are used to designate variable names. Since programs often use more than twenty six variables HP has added the option of adding a single digit suffix to give us A0 through A9, B0 through B9 and so forth. Instead of 26 possible variable names we now have 286.

String variable names, like string functions, end with a $ dollar sign. As with numeric variables they also offer the full 286 possible names. For example, X$, B$, A0$, Z9$. String variable names are separate from numeric variables; both X$ and X can be used at the same time. Unless otherwise specified (with DIM) a string variable can contain a maximum of 32 characters.

**Types of Variables**

The rich library of operations the 71 can perform is further expanded by the ability to use several types of variables. These types specify the mathematical precision and usage of the variable. REAL variables are the full precision variable with twelve significant digits (mantissa) plus three digit exponent. SHORT variables have the same three digit exponent, but use only a five digit mantissa. INTEGER have limited precision (only five digits), and round the fractional part to the nearest whole number. Each of these three variable types use the same amount of memory; there is no savings in using shorter precision. This was done to standardize the way the 71 handles numbers internally, and simplifies things considerably.

**Arrays**

An array consists of a number of elements (either numeric or string) of one type which are represented by a common symbolic label. Numeric arrays may be one or two dimensions, string arrays are limited to a single dimension. A two dimension array is also called a matrix. Unlike regular (scalar) variables, arrays consume less memory per element than individual variables of the same precision. They may be created in any of the three precisions.

**OPTION BASE**

The lower bound of the array is set at either zero or one depending on the current OPTION BASE. Changing the Base setting does not affect the lower bound of arrays previously created. The OPTION BASE setting is a global declaration (same for programs and calculator variables). Either zero or one will be allowed.

```
OPTION BASE 0          OPTION BASE 1
```

Two dimension arrays are refrenced by row then column in the form X(row,col) This table represents an array of (3,4) created with OPTION BASE 0 for a total of twenty elements. If OPTION BASE 1 had been in effect then the zero elements would not exist, it would contain only twelve elements.

```
0,0   0,1   0,2   0,3   0,4
1,0   1,1   1,2   1,3   1,4
2,0   2,1   2,2   2,3   2,4
3,0   3,1   3,2   3,3   3,4
```

Arrays can be implicitly created, though they will be created with elements zero (if OPTION BASE 0) or one (if OPTION BASE 1) through 10. Store something in any element of a non-existant array and it will be created to the default size. If the variable name specified is already being used for a non-array variable then the 71 will cause an error. A more practical method for creating them is demonstrated in a moment.

Arrays are refrenced by an element number following the array name.

```
A(5)        element 5 of the single dimension array A
B(2,3)      element 2,3 of the two dimension array B
X$(3)       element 3 of the string array X$
Y$(5)[2,3] characters [2,3] from element 5 of the array Y$
```

**Using Arrays**

Indirect variable usage has long been a trick used by calculator programmers (STO IND X). Arrays give BASIC this advanced capability as well as extending the number of possible variables far beyond 286 (to a maximum of 65535). The elements may be themselves specified by mathematical expressions.

```
X=A(2*R,3*C)
```

String Arrays can only have a single dimension. As with regular string variables, they are created to a maximum length of 32 characters per element unless otherwise specified. An implicitly created string array will have 10 (or 11 if OPTION BASE 0) elements of a maximum of 32 characters each.

**Re-dimensioning an Array**

Interestingly, if an already existant string or numeric array is re-dimensioned without changing the element size (the number of characters which can be stored per element or the precision of numerical elements) then elements are merely added or eliminated without reseting unchanged elements to null.

Since elements in an array are not destroyed when re-dimensioning (if rules are followed) this can be used in a program where an array will be created of minimal size to conserve memory, then, if conditions change it can be increased in size without losing information.

Variable names can only be used for a single variable type (again, strings and numeric variables are a different case). If a variable name has been used for, for example, a REAL number then it cannot also be used for an array.

The following table lists the available variable types plus memory consumption. COMPLEX variables are included in this table, but are only available with the Math ROM. A complex number can be either REAL or SHORT and has a real and imaginary part represented as (0,0i) where the two parts are represented in parentheses separated by a comma and i represents the imaginary part. Most mathematical expressions can be evaluated using both real and imaginary part of complex numbers when the Math ROM is present.

| Type | Precision | Memory Usage |
|---|---|---|
| REAL | 12 digit, exponent | 9.5 bytes |
| INTEGER | 5 digit, exponent | 9.5 bytes |
| SHORT | 5 digit | 9.5 bytes |
| REAL ARRAY | 12 digit, exponent | 8*(Dim1 - Base +1)*(Dim2 - Base +1)+9.5 |
| INTEGER ARRAY | 5 digit | 3*(Dim1 - Base +1)*(Dim2 - Base +1)+9.5 |
| SHORT ARRAY | 5 digit, exponent | 4.5*(Dim1 - Base +1)*(Dim2 - Base +1)+9.5 |
| STRING | | Max length+11.5 |
| STRING ARRAY | | (Dim - Base +1)*(Max length +2)+9.5 |

```
Additional Data Types with MATH ROM:
      COMPLEX        12 digit, exponent  25.5 Bytes
      COMPLEX SHORT   5 digit, exponent  18.5 Bytes
      COMPLEX ARRAY 12 digit, exponent  16*(Dim1 - Base +1)*(Dim2 - Base +1)+9.5
      COMPLEX SHORT ARRAY 5 digit,exp   9*(Dim1 - Base +1)*(Dim2 - Base +1)+9.5
```

**DIM, SHORT, INTEGER**

These three keywords are used to explicitly create (or declare) variables. The keyword REAL can also be used to create REAL variables, but only DIM can be used for strings. The keyword DESTROY, discussed later, is used to get rid of unwanted variables.

Remember, arrays are not necessarily initilized to zero if they already are in existance. Variables of other types, if they already exist, are set to zero (or null if string variable).

```
Create REAL or string variable
       DIM A , B , C , X$[96] , K$[1]
           REAL X , Y , Z

Create or re-dim a one-dimension array
       DIM A(5) , B$(9)[10]

Create or re-dim two dimensional array
       DIM X1(3,4)

Create a SHORT variable or array
       SHORT V1 , Y(23) , Z(5,8)

Create an INTEGER variable or array
       INTEGER M , R(11) , B(3,2)
```

**ZEN and Variables**

Variable names, as we have discovered, are symbolic. Let's recap some of the other disconcerting realities of variables. A variable does not necessarily exist even if it has been tested, until it has been explicitly (DIM) or implicitly (by storing something in it) created. A variable which has been created as an array cannot be tested as a simple (scalar) variable; nor can a simple variable be tested as an array. A complex variable can only have an imaginary part if it exists; if it does not exist then it cannot be tested for an imaginary part (BEEP, Err).

Strings are always created to zero length, but arrays are merely redimensioned without being zeroed.

In many ways the HP-71 is a hybrid of calculator and computer; Part way between the HP-41 and HP-75. The 41 has a 1-bit CPU which evolved from as far back as the HP-35. The 75, on the other hand, is an 8-bit portable computer which evolved from the desktop bound HP-85. Before introduction, HP had considered calling the 71 the "HP-44" because of the great popularity of the HP-41, then rationalized that, since it is a true computer, it should have a computer name. Hewlett-Packard attaches internal names to products; the HP-75 was the Kangaroo, the HP82161A Cassette Drive was called Filbert (really!) and the 71 is Titan. In addition, the microprocessor (CPU) which controls Titan is called Capricorn.

**Central Processor (CPU)**

The 71 has a custom 4-bit CPU. This means that four address lines carry data into and out of the processor; everything else being equal, this would make the 71 four times as fast as the HP-41, and half as fast as most desktop machines. However, the 71 is a next generation machine with the ability to do full precision floating point math in its 8-byte CPU registers. Registers in the CPU in most desktop computers, by comparison, can only hold two bytes, thus making floating point math on the desktop machine somewhat slower than integer only. These large registers mean that the 71 is optimized for "hard" math and will provide greater accuracy than, for instance, BASIC running on an IBM PC. There are four main working registers and five scratch registers in addition to two 5-nibble (2 1/2 byte) data pointers and nine status registers.

**Clock Speed**

Another unique aspect of the 71 is the low power consumption. Partly responsible for this is the relatively low clock speed. With the ease of doing high precision math, the engineers rationalized that great processor speed would not be necessary. The 71 runs at about 600kz, though this speed varies with the number of devices attached. This is the speed at which the CPU runs, and does not affect the speed of the real time clock or frequency of the beeper. When doing speed comparisons with other computers this compromise will be immediately apparent. In purely mathematical mathematical tests the 71 will keep up with or surpass the desktop machine, while other operations, primarily those dealing with a great deal of memory accessing, the desktop machine will win.

Clock speed is recomputed each time the 71 is reconfigured, which happens when the 71 is turned on or :PORTs are altered. You can see your 71's current clock speed with the following:

```
P$=PEEK$("2F977",5) @ DISP HTD(P$[5]&P$[4,4]&P$[3,3]&P$[2,2]&P$[1,1])*16
```

**Hexadecimal Numbers**

In the above example, "2F977", is a hexadecimal (base 16) number which represents a location in memory. The keyword HTD converts the results of the expression to standard decimal (base 10) format for viewing. The hexadecimal number system (usually called just "hex") is often used to simplify communications with the binary world of the computer.

Everything within the HP-71 is represented in the binary (base 2) numbering system. Binary digits or "bits" (a contraction of parts of both words) can have a value of either zero or one. This limited range is made usable by grouping units of four bits into a nibble, or, as we'll call it throughout this book, a nib (an alternate spelling is "nybble"). Various combinations of zeros and ones in these four bits represent values from zero (all bits clear) through fifteen (all bits set).

| Decimal: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hex: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| Nib 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Nib 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| Nib 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Nib 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

In hex, values between 10 and 15 are represented as a single digit of "A" through "F". Values larger than 15 (or "F" in hex) can be represented by groups of nibs called words. The most common word size is two nibs or one byte. The 71 uses a five nib word size to address memory and can, therefore, use "FFFFF" or 1048575 nibs.

**Memory**

The memory location "2F977", called "CSPEED", is used by the 71 to store the current clock speed setting. Notice that we specified five characters ("2F977") for the location for the location; memory is addressed as five digit hex numbers. Since the HP-71 has a 4-bit CPU, all locations are nibs or 1/2 bytes. This translates to 524,288 bytes, or simply 512K. So, the maximum amount of memory, RAM or ROM, is 512K bytes. A listing of addresses or "memory map" can be found at the back of this book.

A 71 with ostensibly 17.5K bytes only has about 16K available, even when first turned on, nothing in it. This is because the Operating System reserves part of RAM for pointers and other necessary system information (such as "CSPEED"). This seemingly inflated rating of memory is actually less than most computers which can often usurp 4K or more just to be able to power-up.

The Operating System and BASIC live in four 16K ROMs in the first 64K bytes (from address 00000 through 1FFFF) of the memory map. This is known as "hard addressed" or "hard configured". That is, regardless of how memory moves around (and it does, whenever we add memory or plug-in ROMs), the Operating System will always have the same home. In this world of flux, that 64K block of ROM will remain constant. This stability is invaluable for the Assembly Language programmer who wishes to use subroutines from the Operating System. Hewlett-Packard has guaranteed that the official entry points will remain constant, even if the 71's Operating System is changed. An entry point is the address in hex of a machine language subroutine. Aproximately 120 entry points are documented in the back of this book.

**:PORTs**

All of memory, indeed, all devices attached to the 71 are addressed through a bus. That can almost be taken literally: a piece of data can tell the bus where it is going, and it will be delivered to that address. The 71's engineers took advantage of this system to allow portions of RAM to be partitioned from MAIN RAM. When a portion of RAM is designated as independent RAM it is removed from MAIN RAM, and its address is reconfigured to look much as a ROM. The files are still accessable, and may be edited and copied. The advantages to partitioning parts of memory into Independent RAM are many: Eliminate files from the :MAIN file chain to make it easier to find them, or perhaps, to keep them from cluttering up :MAIN RAM. Files in :PORT RAM will usually not be lost in the event of a crash (though, nothing is truely secure).

PORT 0 contains the main RAM as well as the optional HP-IL ROM (if plugged in). Memory in MAIN RAM can be partitioned in 4K blocks from .0 (or, simply 0) through .03. PORTs 1 through 4 are the front module PORTs. :PORT(5) is for the Card Reader. There are also provisions for PORT Extenders to allow more than the standard 5 PORTs, though no PORT Extender is currently available. BASIC keywords for moving files between PORTs are fairly straightforward.

```
CAT :PORT(0)                Inspect file headers of files in :PORT(0)
FREE PORT(0)                Reserve a block of RAM as Independent
MEM(0)                      Return the currently free memory in :PORT(0)
CLAIM PORT(0)               Purge files from :PORT(0), reclaim memory
SHOW PORT                   List :PORTs and sizes
COPY AFILE TO :PORT(0)      Load a file to :PORT(0)
COPY AFILE:PORT(0) TO :TAPE  Copy file from:PORT(0) to mass storage
PURGE AFILE:PORT(0)
```

## Environments

The HP-71 is unusual in the world of personal computers. Its file structure, operating system and advanced BASIC are unique. But another aspect of the computer is often played down or at least taken for granted. An environment, in computer terms at least, consists of a set of programs and data. The 71's proclivity for multiple environments is usually found only on very large computers, often using the incredibly large and complicated UNIX operating system. Let's spend a few minutes discussing the interraction of files and environments in the 71.

The global environment includes the file system (programs, key assignments and such) as well as all flags, the option base setting and calculator variables; actually, pretty much the whole computer. When we RUN a program, the calculator variables are directly accessable to that program. For example, let's assign a value to a variable, then see how it is used in a program. First, give variable X the value of -5.

```
X=-5
```

Now, a program to display the contents of that variable. We'll create a BASIC program file named "TEST", then place a a single line in it (line 10) to display the contents of X.

```
EDIT TEST
10 DISP "Variable X=";X
```

Since it is the current edit file we can run it by pressing the **RUN** key. When the program is run it will display the contents of X (the value -5). After the program is through (that is, almost immediately, since it only runs for a fraction of a second) we can confirm that X still contains -5.

```
X
 -5
```

Now, instead of RUN, let's CALL the program.

```
CALL TEST
 Variable X= 0
```

What happened to X? The value is zero because we CALLed "TEST" which, in effect, made it a sub-program. When a subprogram is CALLed, a temporary program environment is created for it, and the calculator variables are ignored. So, naturally, X had no value in that temporary program environment. When the CALLed program ends it's environment is eliminated and the calculator variables are again active. In fact, now that the CALLed program has ended, we can confirm that X still has the value -5.

The CALLed program can use the same variable names as the main (calculator) environment without fear of altering them. Besides preserving the computer in the state we want it, CALLing a program has the added advantage that all of the memory the program used while it was running is reclaimed. The following is a simplified "Memory Map", that is, a chart of how memory is laid out in the 71. A more detailed map can be found at the back of this book.

```
                  Memory Map
High Memory         Plug-In ROMs & RAM
                    Calculator Variables

Unused Memory

                    Programs & other files
                    Global Enviroment
Low Memory          Operating System
```

As you can see, programs and variables are at opposite ends of memory, there is no

direct relationship between a program and calculator variables. When we RUN a program (with some exceptions we'll discuss in a minute) it will use these same calculator variables. This is how memory is arranged when a program is CALLed.

```
                Memory Map During CALL
High Memory       Plug-In ROM & RAM Modules
                  Calculator Variables (on hold)
                  Active Temporary Program Environment
Unused Memory
                  Programs & other files
                  Global Enviroment
Low Memory        Operating System
```

The calculator variables are left where they are, and new variables, in a separate environment, are created below them in memory. As far as the CALLed program is concerned, this temporary environment is all that exists. While it isn't necessary to remember how memory is arranged (the 71 keeps everything organized for us), it is important to understand environments and the differences between RUN and CALL.

A suspended environment also contains other program information such as the current ON ERROR setting, FOR NEXT and GOSUB stacks. So, for instance, an error in the sub-program nor will RETURN us to the calling environment.

**Sub-Programs**

Many programs begin with a SUB statement (in the program code) which declares the entire program to be a subprogram. Again, the intention is to make it easier to run the program and to preserve the calculator environment. Unless a special purpose dictates otherwise, most commercially available programs for the HP-71 are written as SUBprograms. We'll discuss programming in greater detail later.

A program which has been CALLed may, in turn, call another program (using the CALL command which is programmable). It is conceivable that several environments may be stacked in high memory. Each time a program CALLs another, it's own envronment is saved and the temporary environment is created. Again, when the CALLed program ends, the calling environment is again active.

This unique multiple-environment scheme is even more useful than first impression leads. Up to fifteen data items may be passed between programs, and a program may even CALL itself. The following little program is an example of CALLing programs. The variable X is a counter which is passed from program to program. However, the program CALLs itself, increments the counter, and, if the counter is less than five, will call itself again. Once the program has ended, recall the value of X and you will see that it is now 5 because that is the value returned in the last SUB program CALL.

```
10 X=0 ! initialize X
20 CALL ENVIRON(X) ! call the program
30 DISP "DONE" @ BEEP
40 END ! end of main program
50 SUB ENVIRON(X) ! the sub program
60 X=X+1 @ DISP X;
70 IF X<5 THEN CALL ENVIRON(X) ELSE DISP ! call itself, passing X
80 DISP "END,"; @ BEEP 4000 @ END SUB ! done
```

**CALL Cautions**

When a program is suspended by pressing **ON**, an error in the program, or the keyword PAUSE the program environment is still active. If you then CALL another program, or the same program, this environment will again be active when that program ends. If you repeatedly CALL a program and suspend it, in very short order you will be out of memory. The SUSP annunciator on the right edge of the LCD display is lit whenever a program has been suspended. RUNning a program will automatically end all programs which have been suspended.

## Modes

A mode is a state of the computer which controls which operations may be performed. BASIC calculator mode, in which we can enter calculations and edit programs, is only one of many possible modes available in the 71. CALC mode performs purely mathematical operations. FORTH and HP-41 emulation (available with plug-in ROMs) offer other unique modes in which the personality of the computer is changed by the operations which can be performed as well as the methods used to perform them. BASIC, CALC, HP41 and FORTH modes each will remain in effect after turning off then on the computer. BASIC is the most often used and is the "native" operating mode and is always returned to when exiting, for instance FORTH.

```
    >                   BASIC mode
                  <-    CALC. "CALC" annunciator lit. Press f-CALC for BASIC
OK { 0 }                FORTH. Enter "BYE" to return to BASIC
 0.0000                 HP-41 Emulation. "BYE" or "BASIC" to exit

    \                   Command Stack (in any mode)
```

### Command Stack

Pressing **g-CMDS** as you know, enables the command stack. This can be done in any mode. However, when a program is running, the stack is only usable when the program is waiting for you to input a response. When a number is expected in the running program you can alternately input a mathematical expression and the 71 will calculate it and return the result as the number expected. Many programs, such as Text Editors, Spreadsheets, and programs in some ROMs do not use the command stack, instead the arrow keys perform other operations.

Usually the stack contains the five most recent commands, but it can be varied between one and sixteen entries. A program called "CMDSTK" for changing the number of entries is listed in the programming section.

## Often used Commands

These keywords handle most of the day to day, non-programming tasks we ask of our 71. It would take a 400 page book to properly list all of the keywords available. In fact, it did, all of the keywords are listed alphabetically in the HP-71 Refrence Manual.

### STARTUP

When we first turn the 71 on it does some self tests then returns to whatever mode was active at power down. The STARTUP keyword can be used to have the 71 do anything series of BASIC commands. This is helpful to automatically run a program or execute a series of commands. The STARTUP sequence is ignored in CALC mode, but will be performed in FORTH or HP41 modes, though still with BASIC syntax.

```
STARTUP "CAT ALL" Tells the 71 to do a CAT ALL when it turns on
STARTUP "IF TIME$>='13:00:00'THEN RUN LATE"  conditionally run a program
STARTUP ""      Deactivates the startup command
```

### MEM

Available memory is always a concern when running programs, allocating variables, or writing data files. The keyword MEM returns the amount of memory not currently being used in main RAM. A PORT number may specified to show available memory in a PORT. Note that the syntax for this command is different from others dealing with PORTs. MEM(0) will display memory in :PORT(0).

### DELAY

The 71 pauses between displaying a series of lines so that they may be more easily read on the LCD. This delay setting, and the speed at which lines which are too long to fit in the LCD will scroll can be set with the DELAY keyword. A setting of 8 (eight) or greater is interpreted as "INF", the 71 will display the current line until you press a key. A delay of zero will cause the display to change whenever new

data is available. A scroll rate of INF will inhibit long lines from scrolling (the left and right arrow allow you to view the whole line. Many programs alter this setting without restoring it. The default (the way the 71 works when reset) setting is a delay of .5 seconds and a scroll rate of .125

```
DELAY 0          Sets line delay rate to zero
DELAY .5,.125    Sets line delay rate to 1/2 second and scroll to 1/8.
```

**WIDTH, PWIDTH**

We can set the number of characters which the 71 will display or print on a single line with the WIDTH and PWIDTH commands. If a displayed or printed line is longer than the specified length then the 71 will automatically display the remainder on a separate line. The default for both is 96 characters, maximum for both is 255. Normally, when displaying lines longer than the WIDTH setting, pressing a key will show the remainder of the line, However, if WIDTH is set to INF, the end of the line will not be displayed.

```
WIDTH 22         Sets the maximum display line length to 22
PWIDTH 80        Sets the maximum to be printed on one line to 80
```

**STD, FIX, SCI, ENG**

The display format of numbers can be set much as with a calculator. The default is standard BASIC format (STD) which displays in floating point format, supressing decimal point with integers and displaying in scientific notation when the number exceeds 12 digits. This sets the display format but does not affect the numeric precision used in calculations. The keywords set both the number of decimal places and the display mode; valid settings are zero through eleven. The number setting is used in BASIC, CALC, and in most programs, though some programs change this setting.

```
STD              Restores the standard BASIC display format.
FIX 4            Sets the display format to 4 decimal places.
SCI 2            Sets Scientific Notation format to two decimal places.
ENG 3            Sets Engineering Notation to three decimal places.
```

**TIME$, DATE$**

The internal clock on the 71 runs all of the time, even if the computer is shut off. TIME$ and DATE$ recall the current settings of the clock. Current clock settings are also placed on the headers of all files when created or copied to or from mass storage. Time is displayed in 24 hour format and the date is displayed "YY:MM:DD". The clock is set with the statements SETTIME and SETDATE.

```
TIME$            Displays the current time
DATE$            Displays todays date
SETTIME"13:15:00" Sets the clock to 1:15 pm.
SETDATE"86/07/04" Sets the date to July 4,1986.
```

A simple program can be used to constantly display the time and date when other programs are not being used. This can also be assigned to a key or used as the STARTUP string.

```
FOR X=1 TO INF @ DISP TIME$&" "&DATE$ @ NEXT X
```

**DESTROY**

The 71 automatically creates calculator variables as they are used in a program or CALC mode. They do not exist until they are first used. The memory they consume can be reclaimed when they are no longer needed using the DESTROY keyword. This is not as lethal as it sounds; no smoke will rise from the card reader port, all that will happen is the memory used by the variable will be reclaimed.

```
DESTROY ALL      Frees memory used by all calculator variables
DESTROY X,A$     Frees memory used by variables X and A$
```

**CAT, CAT ALL**

Many larger computers use a menu to display files currently available. CAT ALL displays file names, type (BASIC, TEXT, etc.) size in bytes, date and time of creation and the :PORT (if applicable) **UP, DN, g-UP** and **g-DN** keys move us through all files in the chain. In addition, two other keys take on a special meaning during CAT ALL. **f-LINE** moves to the next :PORT, and **f-EDIT** makes that file the current edit file (if it is BASIC). The **ON** key terminates CAT ALL. The catalog entry for individual files can be displayed with the CAT keyword by specifying a file name, :PORT, or mass storage device name.

```
CAT ALL             Catalog of all files in RAM and :PORTs
CAT AFILE           Catalog of file named "AFILE"
CAT :PORT(2)        Catalog of all files in :PORT(2)
CAT                 Catalog of the current file
CAT AFILE:PORT(0)   Catalog of the file "AFILE" in :PORT(0)
CAT AFILE:TAPE      Catalog of the file "AFILE" on Disc or Cassette
CAT CARD            Catalog of a magnetic card track
```

**COPY**

The most often used file command. Used to copy files between RAM, Magnetic Card, :PORTs and Mass storage.

```
COPY AFILE TO BFILE   Copy file named "AFILE" to new file called "BFILE"
COPY AFILE:TAPE       Copy "AFILE" from Disc or Cassette to RAM
COPY AFILE TO :TAPE   Copy a file from RAM to Disc or Cassette
COPY :CARD TO AFILE   Copy a file from magnetic card to RAM
```

**EDIT, PURGE**

These two keywords are used to create new files and eliminate files when they are no longer needed. PURGE can be used on any file type in RAM or on mass storage, unless the file has been SECUREd (see below). The file will be eliminated and any memory it consumed will again be available.

The EDIT keyword can only be used with BASIC file types. If the file specified does not exist it will be created; this is the only way to create new BASIC files. We can only edit files in MAIN RAM or :PORTs, not on mass storage.

```
EDIT              Edit the workfile
EDIT AFILE        Edit a BASIC file called "AFILE"
PURGE             Purge the current file
PURGE AFILE       Purge the file called "AFILE"
PURGE AFILE:TAPE Purge the file "AFILE" on Disc or Cassette
```

**NAME, RENAME**

Since only one file of a given name may exist in RAM or on a given device, some creative file name juggling is in order. RENAME makes it a breeze. NAME is used only to change the name of the "workfile", regardless of what is the current edit file.

```
RENAME TO APROG              Changes name of the current file to "APROG"
NAME APROG                   Name the workfile to "APROG"
RENAME APROG TO BPROG        Changes the name of "APROG" to "BPROG"
RENAME APROG TO BPROG:TAPE Change name of a Disc or Cassette file
```

File names must begin with a letter (A-Z) and may contain numbers (0-9) as long as they are 8 characters or less. While the 71 poses no other restrictions on file names, several names should be avoided because of possible conflicts and confusion. File type names (such as TEXT and KEYS) and device specifiers like TAPE, and MASSMEM as well as ALL, CARD, MAIN and should be avoided.

**SECURE, UNSECURE**

To insure that a file is not accidently PURGEd or otherwise altered, they may be designated as temporarily secured. This is especially useful with BASIC files.

```
SECURE AFILE     Secure "AFILE" against accidental alterations.
UNSECURE AFILE   Make "AFILE" no longer secure and, therefore, alterable
```

**LIST, PLIST**

Print or display the contents of the specified BASIC file. If a printer is assigned and active The EDTEXT LEX file in the Text Editor, HP-41 Emulator and FORTH/Assembler ROMs add the ability to LIST and PLIST TEXT files. File types other than BASIC and TEXT cannot be LISTED or PLISTed.

```
PLIST              List the current file to the PRINTER IS device
LIST AFILE         List all lines in file named "AFILE"
LIST AFILE,10,100 List lines 10 through 100 in file "AFILE"
PLIST AFILE        List "AFIILE" to PRINTER IS device
```

**FETCH**

Moving around the current BASIC file can be speeded by the non-programmable FETCH command. Specify a line number or label and that line will be made the current line. If the line number specified is not found a blank line with that number will be presented to you with the cursor positioned after the line number. However, if a label specified is not found you will be presented with an error (beep, "ERR:Stmt Not Found").

```
FETCH 1200       Make line 1200 in the current BASIC file the edit line
FETCH ZAP        Make the line containing label "ZAP" the edit line.
```

**RUN, CALL, CONT**

```
RUN              Run the current program file
CALL             Call the current file as a subprogram
RUN APROG        Run the program file named "APROG"
CALL APROG(X,Q$) Call "APROG" and pass two parameters
CONT             Continues running the current program where it stopped
f-CONT           Pressing this key continues running the current program
```

**END, END ALL**

Many times we will suspend a program using the **ON** key. The END keyword can be executed from the keyboard to properly terminate the program.

```
END              End the current program and restore calculator variables
END ALL          End all suspended programs
```

**OFF IO, RESTORE IO, RESET HPIL**

The HP-IL Module tries to assign printer and display whenever the 71 is turned on. This can take several seconds each time the computer is turned on if the loop is broken (that's HP for nothing is plugged in). OFF IO disables HP-IL operations and speeds power up considerably. RESTORE IO is used to reinable the HP-IL module. However, if RESTORE IO is used with a broken loop the 71 will hang up for few seconds then issue an error message. When used in this context it is usually better to use RESTORE IO after connecting all devices. If flag -21 is clear then all devices capable of being turned off remotely will be powered down when the 71 is turned off.

```
OFF IO           Disables HP-IL operation
RESTORE IO       Enables HP-IL.
RESET HPIL       Address loop, re-assign all devices.
```

**PRINTER IS, DISPLAY IS**

Normally, HP printers and display devices will be assigned automatically when the 71 is turned on. However, we can reassign these devices as needed. Printer output can be directed to the display to test a print routine without wasting paper or to assign the printer as a display for a kind of super-trace mode. Be cautious about assigning devices; there is no protection against, for instance, assigning a Disc Drive as a Display device (from which no good could possibly come). The easiest way to specify HP-IL devices is with device words such as :DISPLAY and :PRINTER.

```
DISPLAY IS *          Disables external display
DISPLAY IS PRINTER    Establishes the printer as the display device
PRINTER IS DISPLAY    Establishes the display as the printer
PRINTER IS PRINTER    Restores the printer to it's rightful job
```

**INITIALIZE**

Cassettes and Discs must be formatted before use the first time. This means that the media must have a standard directory and data format before the computer can record files on it. The INITIALIZE keyword is used to format the media to the HP-71 format. The INITIALIZE command erases all data previously stored on the media. Unlike files in RAM, a Disc (or Cassette) must allow for a pre-defigned number of files in it's directory. The standard record (also called sector) size is 256 bytes. A record is the fixed physical size set aside for each item or group of items. Files are stored in multiples of records; for example if the file is 512 bytes then it will occupy two records, however a 513 byte file will occupy three records.

The directory is allocated by records, 8 files per record, so the logical size to specify would be a multiple of 8. Determine the maximum number of records the media can contain and the average file size you use before formatting the disc.

Each media may also be given an identifying volume label of up to six characters. When a Disc has a label then you can refrence is by name using a period instead of a colon such as .VOL2 instead of by :TAPE. This is somewhat slower than addressing the device name because the label is placed at the beginning and must be read each time it is refrenced. The media volume label can be ignored, and does not even need to be specified when you INITIALIZE a Disc.

We'll demonstrate the :TAPE device specifier (representing an accessory ID of 16) which can be used for either HP82161A Cassette or HP9114 Disc. Mass storage devices which do not respond to the :TAPE device word can be referred to by :MASSMEM.

```
INITIALIZE :TAPE          Formats mass storage with 128 file entries.
INITIALIZE :TAPE,200      Formats the media with 200 file entries
INITIALIZE VOL2:TAPE,128 Formats media and labels it "VOL2"
```

When a file is stored on mass storage, a contiguous block is reserved. If the file is copied to RAM and subsequently grows, it will no longer fit in the same place on the media. When copied back to Disc the 71 looks for a new block of records large enough to hold the entire file, if one is found then the entire file is placed there and the original location is marked as available for use (in the same way PURGE works with mass storage). If there is no single block of sectors available large enough for the entire file then an "End of Medium" error will be displayed and the file will not be copied. When the file is moved to it's new home, the previous location of that file is now available, the next file to be copied to the media which will fit will be placed within that block of records, even if a considerable number of records within that block are left unused. As you can see, after this scenario is replayed several times, a considerable number of records may be wasted. The PACK command may be used to delete the unused records between files. Since the operation causes considerable media wear and is subject to the vaguaries of battery power, PACKing media should only be done as a last resort.

## Accessories

First there was the calculator and a few ROM's and single density memory modules. As technology (or marketing) progressed larger memory modules, new ROM's and the Cassette Drive showed up. The author bought a Surveying ROM for the HP-41 a few years ago; the only surveying I've ever done was a philosophy class in college. A coined term for this phenomenon is "The Barbie Doll Syndrome"; what you have is nearly as important as what you're going to get. There is a point at which we should be saying "what do I need to get the job done?".

### Data Storage

Unless you use your 71 just as a calculator or only run ROM programs, you will need some sort of data storage medium. A lot of RAM costs (as of February 1986) more than a Card Reader, Cassette or Disc drive, and, regardless of how much memory there is, some sort of mass storage device will still be needed. In practical terms, the only justification for investing in over, perhaps, 64K is if you can't carry a Disc or Cassette because of weight or damage considerations and a great deal of memory is needed. Lets compare the cost of storing 1K on various types of media:

| | |
|---|---|
| 4-K RAM Module | 18.75 |
| 32-K RAM Module | 5.00 - 10.00 |
| Magnetic Card | .55 |
| Cassette | .08 |
| 3 1/2 inch Disc | .01 |

Each of these alternatives have their purposes. If media cost was the only concern then everyone would use a Disc drive. The tradeoffs are in cost of hardware, portability, capacity, reliability, ease of use and speed; each type of storage excels in one or more of these fields.

### RAM Modules

It's been said that there is no such thing as too much memory. That can be easily defended when we are using a large data base or a carefully crafted mathematical model. The other side of the coin is that we can become lackadaisical about backing up files.

### Card Reader

The card reader is a practical investment and a viable trade off compared to a lot of memory. The per kilobyte cost is high, but the initial outlay is reasonable, not even requiring the HP-IL Module. The cards come pre-formatted with a single 650 byte file on each of it's two tracks, and cannot be reformatted, but can be re-written as needed. A file can extend to any number of tracks and recording multiple tracks is as easy as single tracks.

The author has used these magnetic cards with a similar card reader (in the HP-75) for about three years, recording several hundred cards each month; the plastic bezel on the edge of the card reader is worn shiny from the passage of cards. In that time the card reader has never failed and only one track has lost data.

New cards are the most likely to fail, and usually the first time they are used. If a card worked the first time then it is probably safe to assume it will continue to do so. Data is rarely lost because the card will fail when trying to record on it. A very unofficial (and unscientific) survey has found that one card in 350 will have one bad track, and the quality has continued to improve as time goes by. Some precautions are always necessary; don't store cards near your magnet collection, and, if the card won't read, wipe it off by pulling it through a hole in a clean t-shirt.

### HP-IL Mass Storage

The file handling of the 71 was designed with mass storage in mind. The HP-IL Module and a Disc drive could be installed and used without reading any of the manuals. The COPY command and INITIALIZE are the only operations many users will evey need. The 71's efficient use of memory allows a single drive to provide as much

(if not more) utility than larger computers with two or more drives. Currently available are Hewlett-Packard 9114A&B 3 1/2", HP82161A Cassette Drive, and Steinmetz & Brown Ltd 5 1/4" (single & dual available). The S&B drive is less expensive than the 9114, but does not run on batteries.

HP's Disc format is different from most others. While HP series 40,70, and 80 machines can often read the same disc, Apple, IBM (including the HP 110 Portable and 115 Portable Plus) and most other Disk formats will not be readable by the 71. File exchange with non-HP machines is usually done through Modem or by directly connecting the machines together with the HP 82164A HP-IL/RS-232C Interface. The drives are usually used for storage for 71 files, not for sharing with other machines.

**3 1/2" Disc Drive**

HP 9114(A&B) Disc drives offer fast, reliable storage of programs and data. Physically the 9114 is a five and one half pound brick which is more at home in the office than the field. It averages five to ten times as fast as the Cassette drive. The drive averages 6K per second transfer rate; the full memory of the 71 could be exchanged in three seconds. A single Disc holds about 600 K, and the battery runs for about 4 hours of normal use or 40 minutes of continuous use (as with copying an entire medium). Lest the 9114 be considered just another accessory, consider that it has 128K bytes of ROM and 16K bytes of RAM.

The Lead Acid battery pack may be left charging without fear of ruining it. In fact, it works best at a duty cycle of less than 30% of capacity. Continual use (such as duplicating several discs) will discharge the battery faster than the standard charger can recharge it. A light flashes when the battery is getting low, though this is usually not seen until the drive shuts down and refuses to work at all. Many Disc drive users carry a spare battery pack, and third party compainies are developing battery eliminaters and higher capacity chargers.

Early 9114A's had some design problems which have been corrected to large degree in later releases. The two main problems were that the statement PACKDIR didn't work correctly, and battery consumption was excessive. A new ROM is available from HP to upgrade these older machines (as of this writing it is Part#09114-15516) and may be purchased from the Corporate Parts Center. Installation requires a TORX T-9 screw driver and a static free environment and is probably best left to technicians.

The 9114B has been re-engineered for greater battery conservation and has more blatant low battery warnings.

**Disc Media**

Sony manufactures both the drive mechanism and the Discs Media for HP. The drive is designed to work with only double side certified Discs. Sony and HP Discs are preferred, though Maxell, 3M and others also manufacture them. Of the four brands listed, the first three will probably give greater service life. HP Discs have a life in excess of 1,000,000 revolutions; at 600 rpm that is at least 27 hours of continuous use, and normal access is only a few seconds.

Double sided 3 1/2 inch Discs are becomming a standard and are much easier to find than magnetic cards or cassettes. Open the shutter on a single sided Disc and it will look just like a double sided Disc; the difference is that the second side is not warrented to be any good at all. Perhaps the second side failed a test during production or was simply not even finished properly or tested. In fact, a single sided disc can often be formatted and used as double sided, thus saving about a dollar and a half. However, lurking on that side may be head eating rough spots and voids which probably will fail eventually, usually making it so that the drive can't read either side. A rough spot smaller than can be seen may affect the drive: a smoke particle will lower the signal strength to 15% of normal amplitude, imagine what an innexpensive Disc will do.

A more serious problem with using single sided Discs is head wear. Since both heads are in contact, even if only one side is being read, the top head is doing a sand dance whenever the Disc is in the drive. HP suggests reading (not writing) single sided Discs only, and immediately removing them from the drive when done.

**Disc Drive Maintenance**

The recording heads will usually not need cleaning for many years of use. The minor contact the heads have with the Disc surface is enough to keep them free of debris. The only time a recording head should ever be cleaned is if it has been force fed a dirty Disc. HP sells a "Disc" used for head cleaning.

**If you Drop your 9114 Drive**

The manual suggests that if the drive is dropped more than 5 inches the heads will immediately be destroyed. What usually happens is that the top head comes to rest on the Disc, and, if you pull the Disc out of the drive at this time the heads will be pulled out of alignment. If you drop the drive, the first thing to do is push the Disc release button to raise the heads. Do this even if the Disc (or plastic/cardbord shipping protector) is partially ejected. If you pull out the Disc before pushing the release button you will pull the heads out of allignment. In fact, it is a good idea to always push this button whenever the 9114 has been carried around. Using the shipping Disc is also an innexpensive insurance policy.

Several drives may be stacked without interference. A monitor would be the ideal thing to place on the broad flat top, though don't, the 9114 isn't shielded for it. The ThinkJet printer will not interfere, though be cautious about placing other devices there.

**HP 82161A Cassette Drive**

While it is a cassette drive, it acts like a Disc drive, with random data storage and retrieval, though speed which can't keep up with a steady hand and a card reader. The 82161 Cassette drive is a mixture of good portability, relatively long battery life (three hours), reliability (it has been in use since 1981) and reasonable capacity (128K). The cassette drive will put up with fairly hostile environments and work reliably while bouncing around in a car or airplane. A Cassette running 30 i.p.s. searching for a file is much like the scream of a dentists drill.

Cassettes do not have the life expectancy of Discs. As with the Disc, keep the Cassette Drive away from Monitors.

**Cassette Drive Maintenance**

The two enemies of cassettes are stretching and dirt. The tape stretches whenever it is used, until one day either too much of the magnetic surface as flaked off, or it has stretched to the point of unreadibility. Keeping the drive head clean will greatly increase tape and drive life.

The Cassette Drive uses Nickel-Cadmium batteries which have considerably different characteristics than the Lead-Acid battery pack of the HP-9114 Disc Drive. While opinions differ, most people suggest using the Cassette on battery until the battery light has been on for some time (though before it starts acting erratically), then plug it in and recharge it for the full 14-16 hour period. Continual short charge-discharge cycles may not necessarily shorten the batteries life, though it will, in time, limit the time the drive will run on a single charge. Placing the power switch to **ON** instead of **STANDBY** will reduce power consumption considerably.

The small plastic washer used to hold the spindle (which looks like a tiny washing machine agitator) to the metal shafts may, with time and heavy use, work it's way off of older units, leaving the drive useless. Usually the spring, spindle and washer can be slipped back on without necessatating a repair charge. HP Repair Service in Corvallis Oregon will often supply spare washers without charge if requested. A washer can be taped inside of the battery compartment door for that rare need.

**Printers**

The ThinkJet Printer has just about taken over the 71 printer market. It's strong, silent and runs for a long time on batteries.

While some other brands of printers have more features or provide better print quality, the difficulty of connecting them dissuades most people from using them. The two standard printer interfaces are Parallel (Centronics) and Serial (RS-232C). In neither case will the 71 automatically recognize the device as a printer. And

some programs, such as the Finance ROM refuse to recognize non-HP printers. HP printer codes are quite different than, say, Epson's so programs using alternate print styles or graphics may not run on other printers.

**Display Devices**

Most programs are not written especially for use with a monitor, but all programs will be easier to use. The HP 92198A Interface (manufactured by Mountain Computer) provides either 40 or 80 columns. A monitor can be very helpful for writing programs or for testing how a program will format data without having to print it. When used with a monitor the HP-71 becomes the equal to or superior to any desk top computer.

The only Text Editor program currently available which provides full screen Text Editing is a utility in WorkBook71 (the spreadsheet also supports a Video Interface).

**Using a Terminal**

Alone, the 71 is a fine handheld, a terminal provides a display and larger keyboard. Since most non-HP terminals use different display control codes some compatibility problems will be found. The usual problem will be when lines are longer than 80 columns or a program tries to do formatted display. Most compatibility problems can be avoided by using a terminal or computer acting as a terminal for the keyboard, and a Video Interface for the display. Adapting the 71 to a terminal or other computer is discussed later in this book. Many people use a separate computer for program development then transfer the file to the 71 to be transformed into BASIC or Assembled.

**Other HP-IL Devices**

The following is a partial list of available HP-IL devices. Many other third party devices not listed are also available.

HP 1630A/D/G Logic Analyzer
HP 2225B ThinkJet Printer
HP 2671A/G Alphanumeric/Graphics Thermal Printer 8.5" paper
HP 3421A Data Aquisition/Control Unit
HP 3468A Digital Multi-Meter
HP 45643A HP-150 Interface
HP 4945A Transmission Impairment Measuring Set (opt 103)
HP 5006A Signature Analyzer (opt 030)
HP 7470A Graphics Plotter (opt 003) 2 color, 8.5 x 11 paper
HP 9114A Single 3 1/2" Disc Drive
HP 9114B Single 3 1/2" Disc Drive
HP 82160 HP-41 HP-IL Interface
HP 82161A Digital Cassette Drive
HP 82162A Thermal Printer/Plotter 2 1/4" paper
HP 82163A/B 32 Column Video Interface (discontinued)
HP 82164A RS-232C Interface
HP 82165A GPIO Interface
HP 82166C Interface Kit
HP 82168A Acoustic Coupler. 300 baud, battery powered
HP 82169A HP-IB Interface
HP 92198A Mountain Computer 80 col Video Interface
HP 82905B Impact Printer. Similar to Epson MX-80 (discontinued)
HP 82938A Series 80 Interface
HP 82973A IBM PC Interface

Ocean Scientific A/D Interface
SB10161A Single Steinmetz&Brown 5 1/4" Disc Drive
SB10162A Dual Steinmetz&Brown 5 1/4" Disc Drive



## The File Chain

Unlike many larger computers, the 71 can have many files in memory at the same time; they are organized in what is called a file chain. As you know, there are many different types of files, and an understanding of the interraction of these files is a key to getting a "feel" for the 71. We will discuss the file system in several levels in this section.

There is little discussion of memory limitations in the Owner's manuals because there are practically none. The 71 can work with a maximum of 512K bytes of memory, of which relatively little is spoken for by the operating system; the rest is available for add on RAM and ROM. Add to this a Disc or Cassette Drive and there is almost limitless potential for losing things. The 71 uses a file system (or "chain") to keep all of this possible memory organized. This system can be compared to a filing cabinet and each file within it to a, well, a file.

A plug-in ROM can contain from one to several files. For example, the Math ROM contains only one large (LEX) file, while the Finance ROM contains six (sundry type) files. When RAM or plug-in ROM's are added or removed (turn it off first please!) the 71 automatically keeps track of where and how big they are. When we EDIT, CREATE or PURGE a file in RAM, again the 71 keeps everything sorted out.

### Finding Files

The keywords CAT and CAT ALL let us view what files are in the 71 without the chance of accidently mucking about with them. CAT with nothing following it lists the entry for the current BASIC file being edited. CAT followed by a file name (such as CAT KEYS ) displays information about that file. Let's use the catalog entry for the Math ROM. In this example it is in :PORT(2), though it could have been in :PORT(1) amongst others. If we did CAT :PORT(2) then first the standard header would be displayed followed by the first (in this case only) file entry. Had there been other files in the ROM then their information would be displayed when we pressed the DN ARROW key. This same method can be used with CAT :TAPE. If we had entered only CAT MATHROM then only the information about that file would be displayed.

CAT ALL is used to list the catalog information for every file. First the general catalog header is displayed then the information for the first file. The arrow keys are active to allow viewing other files headers. In addition to the arrow keys, f **-LINE** is used to move forward to other :PORTs.

```
NAME     S TYPE   LEN   DATE    TIME PORT
MATHROM  E LEX   32745 11/01/83 12:00 2
```

The first information is, of course, the file name, followed by a space, S, P, or E. A space means a regular file, or "do with it what you will, don't blame me if you ruin it". S means the file has been secured so that it can't be accidently altered or purged. P means "private" the file cannot be edited or altered (even with POKE). E means "execute only" and is a double whammy; you can neither alter nor purge it. In the case of the Math ROM, merely making it PRIVATE would have been sufficient; the only way you can PURGE a ROM file is to pull out the module; nevertheless it's a type E.

Next comes the file type fully spelled out (i.e.: "BASIC", not "B" or "BA" as with some other HP Computers). File types are discussed about two pages down.

Next we have the file size. This is the aproximate number of bytes of RAM (or ROM) without counting variables which the file occupies. The file size does not include the file header information which takes another 18.5 bytes. Even if the file size is ostensibly zero, it still consumes at least those 18.5 bytes. Enter the following line (presuming there isn't a file already called "TEST"). Notice that memory has decreased by 19 bytes (or 18, MEM is often off by a nib).

```
MEM; @ CREATE TEXT TEST @ MEM
```

If the file is larger than 99,999 bytes (pretty unlikely) then the file size is listed in number of kilobytes (or "K" which represents 1024 bytes); for instance 110K means a file of aproximately 110*1024 bytes.

The date and time of creation come next. This information is updated whenever a file is saved to or read from Disc.

The final part of the catalog entry is the number of the :PORT containing the file.

**File Header Structure**

The 71 maintains the file header in a much different format than it displays it. This is to save memory, and speed up many operations. Since the native language of the 71 is not English the file header is stored in "71ese" and is translated into English when we do a CAT. The file header is a contiguous block of 37 nibs:

| | |
|---|---|
| File Name | 16 nibs |
| File type | 4 nibs |
| Flags | 1 nib |
| Copy Code | 1 nib |
| Creation Time | 4 nibs |
| Creation Date | 6 nibs |
| File Chain Length | 5 nibs |

The file name is always eight characters (16 nibs) and is filled with spaces if the name is shorter than 8 characters. The next four nibs are the internal representation of file type (in hex).

| | | | |
|---|---|---|---|
| BASIC | E214 | LEX | E208 |
| BIN | E204 | SDATA | E0D0 |
| DATA | E0F0 | TEXT | 0001 |
| KEY | E20C | | |

The copy code nib is 0 for normal access, 4 for private (P), 8 for secure (S) and B for execute only (E). Note that file types are encoded differently when stored on Disc; this discussion is only for files in RAM.

File creation time and date are stored in BCD (well, kind of). Both fields are stored reversed so that 20:45 86/09/04 (8:45 p.m Sept 4,86) looks like:

| | | | | |
|---|---|---|---|---|
| 5 4 | 0 2 | 4 0 | 9 0 | 6 8 |
| min | hour | day | month | year |

The final 5 nibs are a pointer to the next file in memory. This is the key to the file chain; one file points to the next. The first file contains a pointer to the second, the second to the third and so on.

**ADDR$**

The function ADDR$ returns the location of files in RAM or ROM in hex format. This location is the first nib in the file header; the actual file contents begin after the 37 nibs in the header and, if there is one, any subheader. Since ADDR$ only works with normal, uppercase file names, the only way to find a file with, for instance, a lowercase name is by finding it's position relative to the file preceeding it in the file chain.

**data Files**

Programs, like everything else, are maintained in files. Many programs create or alter data. This information often needs to be retained for later use after the program ends. Obviously, calculator variables are not the place to store this information because any program could change it, and it is difficult to save it to Disc. Information which doesn't grow or change can be kept within a program in "DATA" statements, but separate Data files are more flexible.

Data files (TEXT, DATA or SDATA) are a convenient place to store information in an organized form. In this discussion we will use uppercase "DATA" to designate a file type, and lowercase "data" to designate information to be stored in a file,

regardless of the file type, and to designate a the general class of files which are used to hold information. Since, unlike BASIC programs, we cannot directly edit a data file, their use is more difficult to understand at first. Imagine a filing cabinet which you cannot inspect, and which will only let you have something if you know what you're looking for and exactly where it is. Other than the over-dramatization, that's the superficial look of a data file. With some understanding of BASIC, and planning how we are going to store information they become easier to understand. Remember that programs usually maintain the data files, so, once the program is written, the intrinsics of maintaining the file are automatic.

Three types of data files (DATA, SDATA, TEXT) provide a variety of ways to maintain information. To use this data we must be able to create the file, store something in it, and recall the information. All three file types use the same general methods and the same keywords, the differences in actual use (which are extensive!) will be discussed when we cover each type.

**Creating the Data File**

Before storing or retrieval, the file must created or loaded from mass storage. New data files are all created in the same manner. CREATE, like the other data file keywords to be introduced in this section, is a statement which does not return a value (you can't use X=CREATE...), and will cause an error if the file named already exists, there isn't enough memory, or a funny sounding file name was used (well, at least not a standard HP-71 file name).

```
CREATE <file type> <name>
```

A comma is not used to separate the file type and name, a space should be used. If only the file name is specified then a DATA file will be created with that name. Because of possible ambiguity, files with the same names as file types should are best avoided (imagine a DATA file named "SDATA"). This operation merely creates an empty file of the designated name and type.

Files may also be created to a a specific size. That is, a number of records (standard unit size), or number of bytes of RAM may be reserved for the file:

```
CREATE <file type> <name>, size
```

This method is required for data files on Mass Storage, because Disc (or Cassette) files cannot be expanded once they are created. HP format for files on mass storage requires the entire file to be in one contiguous block, and another file probably exists immediately after the one being used. Files loaded into RAM then copied back to Disc do not have this restriction; if the file no longer fits in it's old resting place when you copy it bact to Disc, then the Disc looks for a new place to put it where it will fit. When using Disc based files, they may be created up to the maximum size of the medium (or whatever room is available in a single block), then read directly without first loading them into RAM; it wouldn't even be a challange to use a, say, 100K file in a 17.5K computer. If a file is of reasonable size, it may at times be used in RAM, and others directly from disc (unless the person who wrote the program being used didn't realize that, and made it so that it couldn't).

Disc based files also have the consideration of the physical size of a record, which is 256 bytes. File record sizes should be established to be either multiples of this size, or easily divisable fractions to minimize the number of file records which are spread between two Disc records. This is done to minimize access time and medium wear because the drive would have to read two physical records to read one file record worth of information.

To be simplistic, imagine a dresser with drawers which can hold six socks each (the physical record size). Now, you've found an odd sock under the bed and placed it in the first drawer, everything shifts down by one sock. Fine, until the third day when you have to open two drawers to match a pair of socks (the logical file

record size). Not a great analogy, but this chapter was begining to get a bit heavy.

**Opening the data File**

Before writing data to, or reading from a file it must be assigned a number. This number is then used then used in data file operations instead of the file name. This is called "opening a file". Files may be on Disc or in RAM. There are some restrictions on the use of these numbers: Valid numbers are 1-255. Only up to 64 files may be open at a time. And, each file can only be assigned one number at a time. If the file specified does not exist, a DATA file of that name, with 256 byte records,will be created.

```
ASSIGN # <variable> TO <filename>
ASSIGN # 1 TO MYFILE
```

Open files require an extra 34 bytes for the open channel. Disc based files require another 256 bytes for a buffer to store the current record being accessed so that the 71 doesn't have to access the Disc constantly.

When a file is opened an entry is added to an invisible system file called the File Information Buffer, or "FIB". The FIB is used by operations, such as PRINT#, to locate the file quickly without having to look through the entire file chain. When a file is opened the information about it is added to this buffer, and, when it is closed, this information is deleted.

**The File Pointer**

When a file is open the FIB contains a pointer to the first item in the file. Each time we read from or write to the file the pointer automatically moves to the next item. This is called sequential access, normally working from the beginning to the end of a file, reading each item in sequence.

The RESTORE statement is used to place the pointer at a specific place in the file for random access. In this way we can move around the file, reading records as desired.

```
RESTORE #1,10
```

This says to the 71 "restore the data pointer to record number ten, regardless of where the pointer is right now". Record number ten is physically the eleventh record in the file because the first record is always zero. Each file type handles the data pointer quite differently. keep the pointer in mind as you experiment with data files.

**Storing Data**

The PRINT# statement merely places the data specified at the current pointer position in the file.

```
PRINT # 2;A
```

This statement would enter the contents of variable A at the pointer position in the file associated with channel #2. If ther had been data at the pointer position then this would replace it. If we had been at the end of the file then a new record would be added at the end with that value in it. This is sequential writing. Strings can be placed in files in much the same way, though be aware of the way strings are handled in each file type.

```
PRINT # 2;A$
```

We can specify the record to which the pointer is placed when writing to a file.

```
PRINT # 2,10;A
```

This is the same as:

```
RESTORE#2,10
PRINT #2;Q$
```

Be sure that your program maintains an accurate count of the number of records when randomly writing a file; moving the data pointer beyond the end of a DATA or SDATA file generates an error.

More than one item may be printed to the file at a time by separating each with a comma.

```
PRINT #2;A,B,C
```

**Recalling Data**

The pointer has the same importance when reading data as it does when writing to the file.

```
READ # 2;A$
```

This statement says "read the next record in file number two and return it's contents to the variable A$". Again, multiple items may be read at the same time

```
READ # 2;A,A$
```

Be sure that the record which is to be read is of the right type for both the data file, and the variable to which it is being read. The following table demonstrates how various types of files handle reads. Note that TEXT files with strings which are formatted to look like numbers (and even complex formulas!) can be interpreted and read to numerical variables.

| Rec.type | Operation | DATA | SDATA | TEXT |
|---|---|---|---|---|
| string | READ#1;A$ | OK | OK | OK |
| string | READ#1;A | ER | ER | OK |
| number | READ#1;A$ | ER | ER | |
| number | READ#1;A | OK | OK | |

**Closing the data File**

When a program is done with a file it should be closed to reclaim the memory used by the FIB entry, and so that the file may be used by other programs. Remember, a file may only assigned to one "channel" at a time. The ASSIGN# statement is also used to close files.

```
ASSIGN # 1 TO *
ASSIGN # 1 TO ""
ASSIGN # 1 TO "*"
```

When we run a program the files it opens are automatically closed when the program ends. However, when exiting a SUB or any program which was CALLed, the files are not automatically closed. If program is CALLed which tries to assign a file which had been left open by another program then an error is generated. The statements CLFLS and CLOSEALL available in some LEX files will close all files.

Files are also automatically closed when the files are purged, though not when the current edit file changes.

**BASIC Files**

The type of file created when you execute the EDIT statement. Consists of one or more BASIC programs. This is the only type of file which can (normally) be edited (inspected and altered) directly using the cursor keys.

The BASIC workfile is the current file whenever you enter EDIT without specifying a file name. If you enter EDIT workfile then a regular file called WORKFILE will be created; lowercase names are used by the 71 to designate files which are maintained by the machine (such as the keys file). The workfile catalog entry, as with the active KEYS file is listed in lowercase; if this file is copied to, for instance, Disc, without specifying a different file name then the name becomes UPPERCASE in the destination file, and is therefore a conventional file.

The BASIC workfile is the scratch file used for experiments and quick solutions to small problems. Since BASIC is always in "program mode", it is suggested that this file be made the current file most often to avoid the chance that an important BASIC program could accidentally be changed.

**BASIC File Format**

In addition to the file header, a BASIC file has a 12-nibble Subheader with the following format:

```
Sub-program Chain Header     5 nibs
Label/DEF FN Chain Header    5 nibs
End of line marker (F0 hex)  2 nibs
```

These headers are pointers to the first sub, label, or DEF FN in the file. Additionally each of the locations pointed to contain a pointer to the next sub, label or DEF FN in the file. In this way these items can be found much more quickly than by sequentially reading the file. This means that it is relatively unimportant where in a file these occur, it will only slow things down when there are a great number of these tokens in the file. This system also makes it possible to place labels anywhere within a line. This method of linking one label, sub or DEF FN to the next is called chaining, and is the same system used in the main file chain.

Each line in a BASIC file begins with a line number (2 bytes,BCD), values to 99 99 followed by a byte representing statement length. If the line has more than one item on it (multi-statement) then an "@" (4F hex) token preceeds the length byte. Each statement in a multi-statement line is separated by an "@" token and yet another statement length byte. Each line ends with an end of line byte (F0). Since the sub-header ends with an end of line byte then each line number, even the first one in the file, is preceeded by an end of line byte. A line in a BASIC file has the following general format:

```
Line# StLen  Stmt  4F StLen  Stmt   0F
```

Line#-The 4 digit BCD number

StLen - The length of the following statement. Adding this value to the current position points to the next "@" token or the EOL byte (0F).

Stmt - The actual tokenized statement. This is the internal representation of the statement, not as it is presented to the user (HP engineers like to call us "users").

**BIN Files**

The least common file found. This is a program written entirely in HP-71 Assembly Language. While they may be RUN and can become the current file, they cannot be edited directly because they have no line numbers and are coded differently than BASIC files. BIN files are used because they are often faster than BASIC and they allow full access to the machine. The speed increase is not as great as might

be expected because they use most of the same utilities as BASIC, they just invoke them directly instead of through the BASIC interpreter.

The disadvantages of BIN files are that they cannot be easily and quickly changed, take much (much!) longer to write than BASIC, and there is a good chance that even the simplest program will "crash" the computer several times during testing. BIN files are written using the FORTH/ASSEMBLER ROM and require at least Vols 1-2 of the HP-71 Internal Documentation (IDS) for a thorough understanding.

**BIN File Format**

Following the file header the BIN file has a 12 nib subheader which is similar to the subheader in a BASIC file, though there are no labels or user defined functions so that field is set to FFFFF.

```
Sub-program Chain Header  5 nibs
Place holder  FFFFF       5 nibs
Filler byte   20          2 nibs
```

**FORTH Files**

Files written in the FORTH language by the FORTH/ASSEMBLY and HP-41 Translator ROM's have their own special file type(s). These files contain the extensible user written dictionary enteries which make up a FORTH program. Programs written in FORTH often run up to twice as fast as those written in BASIC. It is a matter of personal taste whether BASIC or FORTH is easier and more versatile to use.

**LEX Files**

Language Extension files. They add new BASIC keywords (such as KEYWAIT$) and new operations such as the ability to PLIST TEXT files. LEX files are used when an operation is too slow or difficult to do in BASIC. The typical keyword runs from 15 to 50 times than the equivalent operation in BASIC, and takes from 15 to 50 times as long to write. LEX files are more versatile than BIN files because, instead of providing one-time solutions, they actually extend BASIC so that these solutions may be incorporated time and again. From a programmers point of view, it is preferable to use a LEX with a BASIC file instead of a BIN file because of it's greater versatility. As with BIN files, they are written using the FORTH/ASSEMBLER ROM. This book contains an introduction to LEX files and ASSEMBLY language.

**Using LEX Files**

LEX files are never the current edit file. When they are in the computer their operations are automatically available. Most LEX files contain BASIC keywords which may be incorporated in programs. The use of LEX files is one of the ways HP BASIC is ahead of the pack.

While special keywords should be used when available, keep in mind that if you are going to share the program with another 71 user, the recipient will also have to have that ROM or LEX file.

Should the program be run without the LEX file in the computer an error will result, that is usually the only dammage that will result. If a program is corrupted (trashed) by accidentally running it without the LEX file, then it is probably better to restart the program rather than CONT once the LEX file has been loaded.

Many new capabilities offered by LEX files are introduced for one specific purpose and may not prove reliable in other usage. For instance, a complex number may be placed in RES when the Math ROM is in the 71:

```
DISP (123,456)

     (123,456)
```

However, the boolean operator NOT does not expect a complex number so it is interpreted by the Math ROM, which doesn't have the appropriate operation either so...

```
IF NOT RES THEN BEEP

ERR:XFN Not Found
```

This isn't necessarily a bug in the Math ROM, just beyond it's intended use. The point is that, while the built-in BASIC keywords are meant for general purpose use, keywords in LEX files should be used in context of their intended application. For example, one keyword was written by this author to be used at one place in a single program; it was vital in that application, though virtually useless elsewhere.

**DATA Files**

These files may contain strings or numbers and are the most versatile file type, though probably the least memory efficient. They may be of fixed size or expandable. The file consists of fixed length records of from one to 1,048,575 bytes.

**Using DATA Files**

Numbers always take eight bytes, but strings can be up to the size of one record. Strings which overflow from one record to the next will be difficult to use. If the information to be saved is primarily numbers then an SDATA file is preferred. A TEXT file will often be preferred when working with primarily strings. When the task calls for a mix of text and numbers, or when multiple data items are used for a single purpose (such as name, address, phone), then DATA files are appropriate.

The DATA file can be created to a specific number of records, though if it is a RAM based file, it can be enlarged by writing data past the end of it. The primary reason for specifying file size at creation is to set the individual record length. If not specified, the default record length is 256 bytes, which can be quite difficult to use for random access.

let's create a fixed length DATA file with only two records of twelve bytes each, then assign it to channel #1:

```
CREATE DATA TEST,2,12
ASSIGN #1 TO TEST
```

A number will take eight bytes regardless of it's complexity. Strings take an extra two bytes plus their length. Let's place a number followed by a string (which will overflow into the next record) and another number in the file. The pointer is still at the beginning of the file since we haven't written or read anything, so RESTORE# isn't necessary:

```
PRINT # 1;123,"Synergetics",456
```

Record #0 now has the number 123, the string header and the first character of the string "Synergetics". To prove this let's RESTORE to the beginning of the second record (which is rec#1, remember that records always start at zero), then read the data there:

```
RESTORE # 1,1
READ # 1;Q$,A
DISP Q$;A

ynergetics 456
```

Remember that the record length is 12 bytes, record #1 can only hold "ynergetic". The final "s" (with yet another string header) is in record #2 which the 71 automatically added when it ran out of room in record #1.

```
Rec#0:  123 S
Rec#1:  ynergetic
Rec#2:  s 456
```

The DATA file structure is quite versatile, and, therefore, quite vague at first. The record size will be quite crucial when it becomes time to recall data from the file, because RESTORE places the data pointer at the beginning of a record. If it begins with a string which had overflowed from the previous record then only part of the string will be returned.

When creating a DATA file keep in mind what use it is to have and set the record length to a usable size.

```
Item     unit                    Bytes
NAME:  20 byte string(+hdr)       22
PHONE# 10 byte string(+hdr)       12
DUES   real number                 8
                                  ---
Required record size              42
```

**DATA File Format**

Following the header is the data implementation field. As with many other locations within the 71, the values stored here are byte reversed to make it quicker for the operating system to read them (the CPU is a bit backwards that way, you know).

```
# Records in file    4 nibs
Record length        4 nibs
```

**TEXT Files**

This file type is also known as the HP Logical Interface Format (LIF). The TEXT file can be thought of as two types of files in one. At their simplest, Text files are used to exchange files with other computers. They are meant for sequential access (that is reading or writing one line at a time in sequence). The HP-71 was designed primarily as a number machine, Text was a secondary consideration. Since the record size is variable, TEXT files are valuable for storing free form information.

The EDLEX LEX file adds new operations to make TEXT files a versatile file type. Fortunately, EDLEX is in the HP-41 Translator, FORTH/Assembler, and Text Editor ROMs.

**Using TEXT Files**

Since sequential access was the original intended use for TEXT files. Every time the PRINT# statement is used it writes an end of file marker after it. If we had a ten line file and used RESTORE to move to line five (actually record 5), then PRINT#, that would become the end of the file, all information beyond that point would be lost.

The data pointer can be moved to the end of the file by using the RESTORE# keyword with a number which is greater than the length of the file.

```
RESTORE # 1,9999
```

The following keywords are in EDLEX. They work only on files in RAM, and only with TEXT files.

**DELETE# <chnl#>,<rec#>**

Used to delete a record from the file. All records after the specified record are left unchanged.

**FILESZR("flname")**

Has two uses. It returns the number of records in a TEXT file if the file exists or a negative number representing the reason it didn't. For instance, if it returned -57 ("ERR:File Not Found") the file didn't exist, -58 ("ERR:Invalid Filespec") means a bad file name, or -63 ("ERR:Invalid File Type") for an existant file which isn't TEXT.

The number of records is returned, not the number of the last record in the file. If FILESZR returned 10 then the last record is 9.

**INSERT#<chnl#>,<rec#>,<str>**
The string specified is inserted IN FRONT OF the record specified. No information is lost, everything from the record specified to the end of the file is moved. This operation can be slow on a large file or on a file with several files following it in memory because everything in RAM must be shifted to make room for the new record. It is best to have the TEXT file as the last file in RAM if fast access is needed.

**REPLACE#**
Used instead of PRINT# to replace a current record with a new string. Does not alter the position of the end of the file.

**SEARCH#(<str>,<col>,<start rec>,<end rec>,<chnl#>)**
A function which returns a numerical result of the search performed. The required parameters are the search string, POS within the record to start the search, the starting place (record) and final record to search, then the file's channel number. The string is compared literally to the file; upper and lowercase of the same character will be treated as different characters.

If the file is empty or the string is not found then zero is returned. Otherwise it returns a very calculator-like result. In the example, R stands for record number, C is the position within the record (the column number) where the match was made, and L is the length of the string. For most uses only the integer (IP) portion of the number is used.

```
RRR.CCCLLL
```

**TEXT File Format**

The file does not begin with an implementation field therefore to find a record within a file a utility must read each record sequentially. The file consists of variable length lines (records) which do not begin with a line number. If files are to be exchanged with an HP-75 then a four digit, sequentially numbered, line number must be added. Each record begins with two bytes (NOT byte reversed) which state record length followed by the actual data. Each record has an even number of bytes; if a record is written with an uneven number of bytes then an extra byte is added to the end (which could be of any value since it is never used). Unlike TEXT files written by some other computers, there is no carriage return (ASCII 13) at the end of each record.

Adding the current location to the record length points to the next record. The end of a file is marked by a record length of FFFF (which would not be a valid record length).

**KEY Files**

Contain re-definitions for the keyboard. There may be more than one key file type in RAM at a time. The key file is active only when USER mode is set and the file is named keys.

**KEY File Format**

There is no sub header. Each entry in the key file is formatted as follows:

| | |
|---|---|
| Keycode | 2 nibs |
| Entry length | 2 nibs |
| Assignment type | 1 nib |
| String assigned to key | varies |

The keycode is the actual keycode represented in hex. The next byte designates the length of this entry or the end of the file. The assignment type is represented as a nib with one of the following three values:

```
0  DEF KEY ___      immediate execute
1  DEF KEY ___;     typing aid
2  DEF KEY ___:     direct execute
```

The actual key definition string follows the assignment type nib.

**SDATA Files**

This type of data file has the same format as HP-41 "DA" files and is used primarily for storing numbers. This is a very efficient, flexible and easy to use file for the storage of numbers, though somewhat less so for strings. It is the recommended file type for those times when you have a whopping lot of REAL precision numbers to store and quickly recall. This is one case when you can ignore BASIC's pedantic insistance on calling everything a variable, and call the records in SDATA files "registers".

**Using SDATA Files**

Each record can hold one full precision number or a six letter string. RESTORE#, PRINT#, and READ# work very smoothly with SDATA files. Multiple variables or even arrays and complex numbers can be stored to SDATA files. The same keywords are used for an array, it is not necessary to designate it as an array.

```
PRINT # 1;A
READ # 1;A
```

The following is how a 2x2 array with option base zero is stored in an SDATA file.

| Record# | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Element | 0,0 | 0,1 | 1,0 | 1,1 | 1,2 | 2,0 | 2,1 | 2,2 |

There is no mainframe method for storing strings in SDATA files. The end of the BASIC Programming Hints chapter in this book describes a method for (fairly) easy use of six byte (or shorter) strings in RAM based SDATA files.

**SDATA File Format**

This is physically the simplest file type. There is no subheader, each record is eight bytes long and holds a BCD number (though in a modified format at times). The first register begins on the 37th nib after the ADDR$ of the header.

## CALCulating with the HP-71

Many people graduate to the 71 from Hewlett-Packard calculators such the HP-41 or HP-12 and sometimes have an initial disappointment because the ease of use seems to be gone. After using one for several years, it is easy to forget the learning curve required to master RPN calculators; understanding how to evaluate an expression and translate it to the particular syntax required. It's a common situation: a friend borrows your HP calculator to figure, for instance, sales tax and stares at the keyboard for about three times as long as it would have taken to figure out the tax in their head.

Hand the 71 to that friend and they will be able to at least compute sales tax without a ten minute philosophical discussion about post-fix notation. CALC can be a powerful, expandable computational tool, or it can be used for simple, spontaneous calculations, just punch in an expression and watch the 71 earn it's keep. We've already been discussing calculating with the HP-71, CALC mode is an extension of BASIC mode calculating. Let's spend just a few minutes reviewing some of the basics of CALC mode.

Unlike BASIC mode calculations, in which the 71 waits to evaluate the entire expression until after it gets an **ENDLINE**, then, bang, an answer, CALC mode is always on it's guard. As soon as an intermediate result can be evaluated the 71 replaces that portion of the expression in the display with the results. A closing parenthesis, comma or mathematical operator (such as / or + ) signals the end of an expression, so the 71 tries to evaluate the intermediate result. Pressing **ENDLINE** designates the end of the formula.

There are actually two levels of CALC mode: the CALC editor during which the insert cursor is at the right edge of the display, or editing a line in the stack, during which the BASIC editing keys are active. String operations, including HTD and DTH$ are not allowed in either CALC level.

Editing an expression with the command stack is much like BASIC mode calculations. Intermediate results are not displayed. You can recall previous expressions and modify them in two ways: by altering the entry as needed then pressing **ENDLINE**, or by deleting the **ENDLINE** (crooked arrow) character at the end of the line then pressing **RUN** to again be able to use the CALC editor with that expression. Key assignments are allowed except immediate execute keys which don't cause an error while editing the stack, they are merely ignored.

### Rules of Precidence

As with BASIC mode calculations, the rules of precidence dictate how intermediate results will be evaluated. Our earlier example of 2+3*4-5 will be evaluated exactly as it is in BASIC mode. Notice, again, that multiplication is evaluated before addition so the intermediate result will be based first on the multiplication, then the addition. The 2+3 hangs there while the 71 waits for the next expression to be completed.

```
      2+3*  ,  2+3*4  ,  14-5 ,  9
```

Again, parentheses can be used to insure that the expression will be evaluated as intended. As with BASIC mode calculations, extra sets of parentheses may be used to insure that an expression will work the way desired. The 71 will automatically enter a closing right parenthesis for each left parenthesis entered. Entering a closing parenthesis, even though one is already displayed, will move the cursor outside of that set.

```
      (2+3)  ,  5*4  ,  20-5  ,  15
```

### Long Formulas

CALC, as with all types of "input", is limited to 96 characters per line. Enter a long, complicated mathematical expression, or just add a column of numbers and when the 96'th character is reached the 71 will beep and send you back to the beginning of

the line. This leaves your formula in mid-number and mind in mid-thought. Instead of entering the numbers until the error occurs, find a mid-point and do an intermediate answer.

```
23.7+64.2+44.2+33.1+32.0+27.9+26.3+19.7+24.6+29.4+34.6+33.2+28.4+21.6
()+23.2+31.0+35.1+38.6+46.7+51.5+63.1+74.8+61.3+59.2+52.6+43.1+31.6+26.1
```

**RES, ()**

In the above formula the first portion of column of numbers was terminated with ENDLINE. When we went to the second line we used () to recall the results of the earlier calculation. Even more important than in BASIC mode, RES provides a continuity between mathematical expressions. Note that RES does not change until pressing **ENDLINE** so that intermediate results may again use () within the formula without it having been changed by the intermediate result. The RES keyword, an empty pair of parentheses or just **ENDLINE** without anything displayed will return the result of the previous calculation.

```
2+3  ,  ()*4  ,  ()-5  ,  15
```

**ENDLINE**

Terminates the expression, closing necessary parentheses and completing any partially evaluated expression. **ENDLINE** also places the result of the calculation in the RES register.

**ENDLINE** actually places the carriage return character on the end of the line. It is possible to exit CALC without having pressed **ENDLINE**. In that case, if you edit that stack entry in BASIC mode the last character on the line will not be displayed. This is because the BASIC editor assumes that the final character on the line is a carriage return, which is not normally displayed.

**ON**

The "cancel" key. The formula being evaluated is cleared from the display. The formula is not entered into the command stack nor is the intermediate result placed in RES. The expression will be placed on the stack if, for instance, a long line had been entered and returned "ERR:Line Too Long".

**RUN**

Terminates an expression much like pressing an operator key. Evaluates as much of the formula as it can and displays intermediate results. This does not suspend the rules of precidence. Using our above example, press **RUN**

```
2+3  RUN
*    "ERR:Precidence"
```

Since the entire expression is maintained, regardless of intermediate results displayed, the 71 still expects the expression to be evaluated in sequence.

The real value of the **RUN** key in CALC mode is to partially evaluate an expression with the intention of possibly altering it before obtaining a final result.

**f-BACK**

The "undo" key. Before pressing **ENDLINE** to terminate the expression. **F-BACK** used in conjunction with **RUN** allows us to step backwards through an expression then alter it as needed.

**Arrow Keys**

The **UP** and **DOWN**, g-UP and **g-DOWN** keys move us through the command stack. In CALC mode, the left and right arrow keys are ignored unless the command stack is enabled.

**Command Stack**

Many jobs which might otherwise require writing a program can be performed within

CALC using the entries on the stack. Nominally five, but up to the last sixteen calculations are available in the stack. You can look through the previous entries to make sure that calculations were entered correctly, revise an expression, or recall an entry to the CALC editor. If you delete the carriage return character then press **RUN**, the expression can again be used with the CALC Editor; adding and deleting characters to obtain intermediate results.

**Variables**

As with BASIC mode calculations, values may be exchanged with the calculator environment just by refrencing them in an expression. Unlike BASIC mode, even if a value is destined to go to a variable it is still displayed. The value is not assigned to the variable until the end of the expression, so intermediate results may still be performed, and the expression may be altered until you are satisfied with it before pressing **ENDLINE.**

```
X=TIME/3600      Assigns the result of the expression to variable X
TIME/3600-X      Uses the value previously stored in X in an expression
```

Since the calculator variables are used, they may be shared with programs.

**Inside CALC Mode**

When a BASIC program is running there is somewhat increased power consumption; the computer is constantly working. Lest you think that the ever attentive state of CALC mode will end with increased battery consumption, the 71 performs these tasks quite fast and actually has time for a catnap (called "light sleep") between keystrokes. It wakes up every now and again (well, in computer time) to flash the cursor, then nods off again. In fact, given ten minutes of waiting for us, the computer will shut down completely (called "deep sleep"). The 71 is eminently patient with us slow humans, though less so when we press the wrong keys.

As with the rest of the HP-71's Operating System, all of CALC mode is written in Assembly Language. The three main modules include the editor, decompiler (which tries to interpret our keystrokes, and a group of utilities. Surprisingly, much of CALC mode is contained within this 961 byte block of ROM. Part of the reason for this efficiency is the use of utilities from the BASIC interpreter which helps explain the similarity to BASIC. As an example of this efficency, when the Command stack is enabled, CALC uses the very same editor as the BASIC keyword INPUT.

The environment is altered considerably while in CALC. Because of the complexity of CALC, only single line FN's can be used, and an active display device is ignored, though an active external KEYBOARD IS is allowed. This is probably because the display device would have to be actively supported, thus slowing things down as the insert cursor was constantly being moved around. Try to visualize how CALC mode would look on a monitor and you can see the problem; the engineers dropped the issue all together. Exernal keyboards, on the other hand, require no extra work from CALC. They are interpreted using an interrupt which handles the key, then returns it to CALC so that CALC receives the key without knowing where it came from.

**"CALCAID" Program**

This little program is used for continuous addition without worrying about getting over 96 characters. The current value of variable X is displayed followed by a question mark and the flashing cursor.

```
0.00 ?
```

Enter a mathematical expression then press **ENDLINE.** The results are saved to variable L (easy to remember - think of it as Last x) and the value is added to variable X (as with RPN calculators). Any mathematical expressions which evaluate to a single numerical result (including string functions which return a number) are allowed, and the command stack is active. After each input the program continues to prompt for further data. Re-enter CALC mode by pressing **ENDLINE** without any data. The small "trick" in the program is that it will branch to line 100 with any

mathematical error (which no data at all certainly qualifys as), and line 100 "presses" the CALC mode key and ends the program. Any errors in data entry will also put us back in CALC. The program should always be RUN, never called, so that the calculator variables will be common with those used within CALC mode. A variation of this program can be done to enter data into a statistical array, or any series of variables (preferably using MAT INPUT from the MATH ROM), then return to CALC mode to evaluate the data.

So, we're in CALC mode and want to run a column of numbers which will probably exceed 96 characters so we exit CALC (press **f-CALC** again), and press **RUN** with this as the current program.

```
  9 ! "CALCAID" program
 10 ON ERROR GOTO 100
 20 DISP X; @ INPUT L
 30 X=X+L @ GOTO 20
100 PUT "f," @ END ! go to calc mode
```

**User Defigned functions**

In addition to the power of adding new keywords using LEX files, we can write functions "on the run" in BASIC to use in formulas and save time entering often used or complex formulas. In the context of CALC mode a user defined function can be thought of as a mini program or "macro".

A user function is created with the BASIC statement DEF FN followed immediately by a name which can be letters A-Z and optionally followed by an number. A single line function evaluates an expression and returns a single numerical result. The function must be in the current BASIC file, not in a SUB program.

```
10 DEF FNT=TIME
```

A FN may consist of any number of nested parentheses incorporating the most convoluted logic desired as long as it is a single expression which evaluates to a numerical result.

While it may seem limiting that we may only use single line functions, remember that they may also call other single line functions (or themselves!). Another advantage of using FN's is that they may use strings within the formulas. Let's defign FNT2 which uses FNT twice, to see how long it takes to evaluate a function.

```
20 DEF FNT2=FNT-FNT
```

Compare this to the time it would take to enter the expression directly in CALC to see the actual time savings.

```
TIME-TIME
```

The "hook" of allowing user defigned functions to be recursive as well as calling other functions brings us a new application of BASIC as a threaded language to be interpreted in CALC mode. These FNs can be added to a single file, such as at the end of the "CALCAID" program for a custom function set.

## Basic BASIC

Unlike most personal computers, the 71 is a useful tool without programs. With programming, BASIC expands the 71 to help us solve problems with speed and simplicity. This section introduces BASIC and simple programming concepts and is intended for those who have not programmed in BASIC, or would like a refresher discussion of how BASIC is implemented on the 71.

You've turned on your 71 and entered CALC mode to convert five centimeters to inches. In algebraic logic (which is how BASIC works) you might enter:

```
5 * .3937      [ENDLINE]
```

Now, suppose you have a whole list of numbers (cms's?). You could wander up and down the command stack replacing one number with another until the job was done, or you could write a program to help you.

At it's simplest, a program is a series of keystrokes which have been recorded so that they can be used time and again without having to re-enter them each time. To differentiate between entering commands for immediate exectution and writing programs, programs are stored as a series of lines each beginning with a line number of from one to four digits. When we run the program the computer reads each line and interprets the instructions.

Let's make a program of our metric problem. We have the math part (X*.3937), let's add a program structure to it and have the computer do the work. To begin with, we need to have the computer ask for the number, we do this with an INPUT statement. INPUT, in it's various forms, is used to display a prompt, wait for you to input something, then store the information where the program tells it to.

```
10 INPUT "cm:";X
```

Line 10 tells the 71 to display the prompt ("cm:") then wait for the user to enter the number and press **ENDLINE**. The number is then automatically assigned to variable X, though we could have used any variable. Now that we have the number (safely stored in X) we need to do the math with it:

```
20 Y=X*.3937
```

The solution is in variable Y; let's display the results. INPUT gave us an input, the math operations worked pretty much as they do in CALC mode, so it would figure to display the results we would use something like DISP:

```
30 DISP Y
```

Fine, a simple solution. We can embellish the results to be a little more informative:

```
30 DISP Y;"in."
```

The semicolon tells the 71 that more is to follow on the line, then we add the "in.". Note that the quotation marks (either single or double, but both ends must be the same) are required. Otherwise there would be no way for the 71 to distinguish the end of the prompt from the rest of the universe.

### GOTO

We are writing the program to save some time converting a lot of numbers. So let's modify our program so that it will continue working until we press **ATTN** to stop it. We'll do this by adding a line to tell the computer to go back and start over when it's done with the program:

```
40 GOTO 10
```

What will happen now is that the result will stay in the display for as long as the delay setting (usually a half second), then the program will GOTO line 10 and start over. Remember that ending DISP with a semicolon tells the computer that it is not done with displaying on the current line. We can modify line 30 so that the results will stay on the left of the display when the program goes back to line 10. DISP formats numbers with a leading space if the number is positive or a minus sign if it is negative; in either case it is followed by a space so it isn't necessary to add extra spaces to separate the numbers from the words displayed. With the final modifications the program will run until you press **ATTN** to halt it.

```
10 INPUT "centimeters";X
20 Y=X*.3937
30 DISP X;"in.,";
40 GOTO 10
```

The main difference between a four line program and a five hundred line program is four hundred ninty-six lines. This is not as casual a statement as it sounds. A large program can be thought of as a problem or group of problems which can be broken down into a series of small solutions. These small solutions, like the one above, are then combined to form the whole. A plan, whether in the form of a list of necessary tasks, a flowchart, shopping list, or any other form which you find comfortable makes the difference between a programming nightmare and a problem brought down to size.

**HP BASIC**

BASIC was born in the 60's as a learning tool for computer novices. As such, it used easy to remember commands and simple syntax. Hewlett-Packard has taken the approach of a multitude of commands with an editor which checks syntax when you enter the code. HP BASIC is an excellent environment for learning programming because of it's completeness and consistancy of operation. While you will probably need to refer often to the HP-71 Reference Manual and will hear your share of beeps, the only damage likely to occur to the 71 is from physical abuse to the keyboard.

Let's take the concept from our centimeter to inch converter program and give it another function. This will look a little less like generic BASIC, and more like HP's. We will begin with a prompt which is in the form of a simple menu. When you run the program all you will have to do is press the appropriate key to select a function.

In order to save a little space we will have more than one statement on a line; each statement is separated by a commercial at "@" sign; this is called concatenation. The lines which begin with an exclamation mark are remarks and are ignored when the program runs.

```
  1 ! inch/cm converter
  9 ! set the keyboard to uppercase
 10 LC OFF @ DISP "metric converter" @ WAIT.5
 19 ! the main menu prompt
 20 "MENU": DISP "convert to:Cm/In"
 29 ! wait for a key
 30 IF KEYDOWN("C") THEN "CM"
 39 ! if the wrong key then go back for another
 40 IF NOT KEYDOWN("I") THEN 30
 99 ! inch--> cm
100 "IN": DISP "cm:"; @ GOSUB "INP"
110 Y=X*.3937 @ GOTO "DSP"
200 "CM": DISP "in:"; @ GOSUB "INP"
210 Y=X*2.54
498 !
499 ! input subroutine
500 "INP": INPUT X @ RETURN
598 !
```

```
599 ! display routine
600 "DSP": DISP Y @ WAIT 1
619 ! we're done, return to menu
620 GOTO "MENU"
```

**GOSUB, RETURN**

GOTO, as you recall, is used to go to a portion of the program. Often we will want to use a portion of a program several times to save memory or link together separate modules to keep from having to write similar code several times. In these situations we use a subroutine. To get to the subroutine we use GOSUB and to return the subroutine ends with RETURN. When GOSUB is executed by the computer the address (location in memory) of the GOSUB statement in the program is stored. When RETURN is next encountered the computer looks up where it came from and execution continues from the statement following the GOSUB.

Lines 100 and 200 both have GOSUB "INP" These refer to the label on line 500. We could have just as easily used GOSUB 500, but when programs begin to get complex it is often helpful to use labels to refer to subroutines. Line 500 does an INPUT then RETURNs. Labels will be discussed in more detail later.

A subroutine may call another subroutine by again using GOSUB. Each additional GOSUB adds another entry to the list. This list is called, conveniently enough, the gosub stack and it is maintained automatically by the 71. As you can see, for every GOSUB there must be a pending RETURN, although there may be some time and quite a bit of program between them. The gosub stack works as last in first out; whenever a RETURN is encountered the last entry is "popped" from the stack and program execution continues at the next statement following that last GOSUB. Every GOSUB adds one to the stack, every RETURN pops one. If the stack is empty and the program encounters a RETURN then an error message is displayed because the 71 doesn't have any place to return to.

**FOR, NEXT**

While GOTO can be used to loop through a series of statements. FOR and NEXT provide a more elegant method for looping for a predetermined number of times. Consider the following. Line 10 does a piece of business, the program falls through to line 100 for some more work, then the end of line 100 sends us back to line 10; the problem is that line 500 never gets executed.

```
 10 DISP "HP-71"
100 BEEP @ GOTO 10
500 DISP "end of loop"
```

In addition to having the 71 run around in circles we can have it loop for, say, 10 times:

```
 10 FOR L=1 TO 10 @ DISP "HP-71"
100 BEEP @ NEXT L
500 DISP "end of loop"
```

Now the 71 will loop for only ten times then it will fall through to line 500. FOR and NEXT are paired like GOSUB and RETURN except that there is an association between the two so that you cannot have multiple NEXT's for one FOR, though any amount of program may exist between the two and you can jump in and out of the loop at will.

When the computer comes upon the NEXT statement it increments the value by one then compares it to the final value designated. If the resulting value is less than or equal to the final value (the TO value) then the computer goes back to the statement following the FOR. In the above example the value of L after completing the loop is eleven since it was incremented before comparing it to the final value.

Each FOR - NEXT loop is associated with a numerical variable (like X1 or L) and the initial loop value may be set in any way. Since the variable is a regular program variable you may use it or change it within the loop. HP BASIC allows us to exit a loop and even start another loop using the same variable without causing an error.

Loops are always incremented by one each time through them unless we specify the STEP value:

```
10 FOR L=10 TO 100 STEP 10
```

This would still result in ten times through the loop, however, at the termination of the loop L would equal 110. Positive or negative STEPs may be specified. The advantage of using a negative number is that we can have the counter decrement each time instead of increment. You can create an endless loop by specifying FOR L=1 TO INF since when the counter gets above twelve significant digits adding one to it doesn't increment it at all. The same effect can be seen with FOR L=0 TO 0 STEP 0 but it takes more memory and isn't as clean looking.

**IF, THEN**

BASIC has the ability to alter program flow based on whether or not IF [expression] THEN [do something].
The expression can be a mathematical or string comparison which evaluates to a non-zero (true) value boolean argument. Note that negative values are non-zero. If the expression evaluates true (that is it is not zero) then the statement after THEN is performed, otherwise it is ignored.

The statement to be evaluated (after THEN) may be almost any BASIC statement or series of statements. The exceptions are DIM, FOR, NEXT, DATA and DEF FN. A traditional usage of IF THEN is to be followed by GOTO or GOSUB; for this reason if a line number or label follows THEN then it is interpreted as an implied GOTO. This is saying to the 71 "if the following expressing returns a non-zero result then goto this line, if not then forget that I ever brought up the subject and continue with the next line". The syntax allowed is very liberal and as such usage is incredibly versatile.

```
 10 IF X>Y THEN 2000 ! implied GOTO
500 IF TIME>64800 THEN GOSUB "LATE" @ GOTO 100
 30 IF (A+B)*(X-Y)>Z THEN R=R-12.75
```

We can evaluate the expression as false by using the keyword NOT. This is saying "if the expression is not true then...":

```
 20 IF NOT (X-Y)*Z THEN 500
 70 IF NOT M THEN X=Y
500 IF X OR NOT Y THEN BEEP @ GOTO 100
```

Multiple comparisons may be made using AND or OR:

```
10 IF X AND Y OR MEM<2000 THEN BEEP @ DISP X$
```

**ELSE**

If the expression evaluates as false then the program ignores the remainder of the line and continues on the next line. We can redirect the program to continue on the same line but after the statements which would have been evaluated had the argument been true. A disadvantage to using ELSE is reduced readibility of the program listing.

```
100 IF X=1 THEN 400 ELSE IF X THEN 500
110 BEEP
120 DISP "end of chapter" @ END
```

The BASIC line editor used in keyboard calculations serves several purposes. If we give it a mathematical expression then it tries to evaluate it and return a result. Enter a command like CAT ALL and the 71 will do just that. However, begin the expression with a line number and you have entered a line of HP-71 BASIC. Same syntax requirements, same beeps and error messages. Since BASIC uses English like words and algebraic logic, once we are familiar with it's workings, the logic of a program can usually be figured out just by reading the listing. An advantage of BASIC over FORTH or Assembly language is that we can edit and modify programs at a moments notice.

Some of the material presented here is informational only, it has little to do with the act of programming in BASIC. It is included to give you a feeling for what happens when you press **ENDLINE** in BASIC.

Let's begin by sharing a little program called "ACYDUCY". The program presents two cards and asks the user to guess if he thinks a third card will fall between the first two. The user can pass either by betting zero or by clearing the input. It uses an infinite deck and maintains the users bank. The program ends if you go broke, but, then, that's only fair.

```
 10 STD @ B=100 @ Q=5 @ RANDOMIZE @ ON ERROR GOTO 30 ! initialize variables
 20 'START': DISP "Bank:$";STR$(B) @ WAIT 1
 30 'BET': C=IP(11*RND)+2 @ IF C<2 OR C>11 THEN "BET" ! get first cards
 40 D=IP(14*RND)+2 @ IF D<5 OR D>14 THEN 40
 50 IF C>=D-2 THEN "BET" ! an implied goto
 60 E=C @ GOSUB "CARD" @ E=D @ GOSUB "CARD"
 70 INPUT "bet:$",STR$(Q);Q @ IF Q<=0 THEN "BET" ! an implied GOTO
 80 IF Q>B THEN DISP "Bank:$";STR$(B); @ GOTO 70
 90 F=IP(14*RND)+2 @ IF F<2 OR F>14 THEN 90
100 E=F @ GOSUB "CARD"
110 IF F>C AND F<D THEN DISP "WIN!,"; @ B=B+Q @ GOTO "START"
120 DISP "Sorry..."; @ BEEP 500,.2 @ WAIT 1
130 IF Q<B THEN B=B-Q @ GOTO "START" ! any money left?
140 DISP "You're Broke!" @ END
150 'CARD': IF E=10 THEN DISP "TEN"; ELSE IF E=11 THEN DISP "JACK";
160 IF E=12 THEN DISP "QUEEN"; ELSE IF E=13 THEN DISP "KING";
170 IF E=14 THEN DISP "ACE";
180 IF E<10 THEN DISP STR$(E);
190 DISP ","; @ RETURN
```

Line 10 follows standard BASIC practice of initializing variables to a known value before starting. ON ERROR is used to trap bad input if the user had, for instance, entered no bet at all. This is necessary because the program assigns the input to a numeric variable and a null string is not a valid numeric expression (beep, "Err:Numeric Input"). RANDOMIZE is used to place a new seed in the random number generator. No parameter has been specified, the 71 automatically will use the current clock setting for the new seed. For programs which need a very random sampling specify at least a 12 digit number.

Line 20 (label "START") is the greeting; many programs begin with a prompt to let the user know he has run the right program. Label "BET" finds the two cards and makes sure they are at least two cards apart. Line 70 prompts the user for a bet. The default bet is the same as the previous bet (variable Q). While only a string may be furnished as the default input, the result may be returned to numeric variables. In fact anything which will evaluate to a string may be used.

```
INPUT "numbers:","1,2,"&STR$(IP(C));A,B,C
```

The subroutine "CARD" is used to display the string equivalent of the current card. It is a subroutine to save some memory because it is used three times within the

program.

There are no string variables used within the program. Each number is displayed as a string (using STR$) to suppress the leading and trailing spaces in the LCD window.

While "ACYDUCY" is relatively compact for what it does, it does have several weaknesses. It uses calculator variables without reason, uses more variables than absolutely necessary and it will run until the user either runs out of money or presses **ATTN**.

**Sub-Programs**

As we've discussed, entire programs are often used as subprograms in order to preserve the calculator environment. An equally important usage of subprograms is to make commonly used modules available to several programs; Thus saving memory and simplifying programming. "INCAT" is an example of a subprogram which has little value as a stand alone program, it has no user interface at all, but quite helpful in programs dealing with data files.

The program returns a value which represents the file type of a specified file. It requires two parameters: a string with the file name in question, and a numeric variable in which to return the result. Syntax for use is CALL INCAT("filename",Q) Possible results returned in the numeric variable can be:

```
 0  File is nonexistent          6 It is KEY
 1  It is a TEXT file            7 It is BASIC
 2  It is SDATA                  8 It is FORTH
 3  It is DATA
 4  It is BIN                   20 Unknown type
 5  It is LEX                   21 Invalid name
```

A BASIC file will return the value seven. If you have entered "ACUDUCY" then you can test "INCAT":

```
CALL INCAT("ACYDUCY",Q) @ DISP Q
```

We could have used ADDR$ to find a file, then PEEK$ed out it's file type from the data in the file header, but that wouldn't have helped if the file resides on Disc. Instead we keep in on a very high level: display the CAT entry for the file, then read the information using DISP$. The problem with this method is that DISP$ is intended to recall information after an INPUT, it usually will return a null string otherwise. This is because displayed data is not accumulated into the input buffer unless the cursor is on. The escape sequence CHR$(27)&">" on line 9610 turns on the cursor then is followed immediately by CAT. Since the HP-IL Module uses two extra spaces in the CAT entry, line 9620 looks for those spaces and trims them if found. Finally, line 9630 compares the first two characters of known file types to this file. Using "INCAT" is demonstrated in the **Communicating with RS232** section of this book in a program called "NEC".

```
 9600 SUB INCAT(F$,T) @ ON ERROR GOTO 9660
 9610 DISP CHR$(27)&">"; @ CAT F$ @ T$=DISP$[1,32]
 9620 IF NUM(T$[12])=32 THEN T$=T$[3]
 9630 T=POS("TESDDABILEKEBAFO",T$[12,13])
 9640 IF NOT MOD(T,2) THEN T=20 ELSE T=(T+1) DIV 2
 9650 GOTO 9670
 9660 T=21 @ IF ERRN=57 OR ERRN=255022 THEN T=0
9670 DISP CHR$(27)&"<" @ END SUB
```

The sub program receives values, evaluates them, and returns the result in the same variables. When a sub is called the parameters passed point to the actual variable in the calling program's environment, though a different variable name may be used. Thus saving some time and memory because a copy of the variable does not have to be present in both environments. If actual values are passed as with CALL

SUBPGM(1,2,"test"), then, of course, nothing is returned. Of course a SUB can be called without passing any data; the maximum number of parameters we can pass between programs is 15. Variables which the sub creates are in it's own environment and do not affect the CALLing program.

Line numbers in "INCAT" begin with 9600 because it is expected that it will be added at the end of a BASIC file. Any number of SUBs may be placed in a file.

**Interpreted BASIC**

BASIC on the HP-71 is an interpreted language. What that means is that the computer reads each line, looks up the meaning of the current statement then goes to the part of the operating system which contains the actual instructions for that operation. Consider the following:

```
10 BEEP 2000 @ DISP "I love my [(HP-71)]"
```

The computer skips past the beginning of the line until it finds the word "BEEP" (actually, a token representing beep). It then looks up the location in memory of the machine language code which makes the beep happen, calls that routine, which itself interprets the line to see if frequency and duration had been specified, and, finally, the computer beeps.

With all of that done the 71 returns to the next statement on the line, in this case is an "@"; which it interprets as saying there is another statement following on the line, and so on. This may seem like a long and involved process but it is streamlined and happens hundreds of times per second. An extra benefit to us is that HP-71 BASIC will not allow us to do something really stupid, while Assembly code or FORTH will blindly run the computer off of a cliff if we tell it to.

**Tokens**

To both save memory and speed things up, each statement on the line is "tokenized". This means that when we enter the line the computer substitutes a code of from one to four bytes for the actual keyword. Because of this DISP and PRINT take the same amount of memory even though they have a different number of characters. The spaces separating items do not take any memory; they are actually part of the keywords themselves. This explains why we can't insert extra spaces between statements for clarity, they are not part of the keyword, so were never entered into the line.

When we edit a line in a BASIC program the computer looks up each token and displays the keywords and other data associated with it. And, if you press **ENDLINE** while editing the line, the whole process of tokenization starts again.

**Statements versus Functions**

All tokens can be summed up into these two categories. In general, a statement is free standing and does not necessarily require any parameters. An example of a statement requiring no parameters is OFF. BEEP is a statement which may optionally have up to two parameters. The 71 lumps together statements and commands which some BASICs treat separately.

A function may require zero or more parameters, does something with the data and returns either a numerical or string result. An example of a function is MAX which requires two numeric parameters and returns a number. One of the strengths of functions is that they may be used within expressions containing several other arguments as long as the final result is a single number or string. For example:

```
10 X=MAX(FP(Y),ABS(FP(Z)))
```

The distinction between statements and functions is often blurred, and the HP-71 has a penchant for allowing liberal BASIC syntax. The only blanket statement to made about functions and statements is that functions may be preceeded with the keyword LET and statements may not. If you are in a crowd of sticklers and don't wish to misrepresent a function as a statement then you can usually call it a "token" or "keyword" without anyone ever catching on that you are not sure what it is. In this chapter we'll usually say "token" or "statement"; you can nod with a knowing smile.

**Writing Readable BASIC**

BASIC program lines are numbered in order from 1 through 9999. Because of the way the line numbers are stored, each line number requires the same memory regardless of whether the line number has one, two, three or four digits. For readibility you may want to begin different modules within a program with different groups of line numbers. For instance main input may be on 1000-2000 and subroutines may be on 8000-9999. It is generally a good idea to number lines by 10's or 100's so that the inevitable changes may be inserted without having to renumber the program. While the following is by no means etched in stone, it gives an example of how a single purpose program might be organized.

```
    1 -  999  initialize vars
 1000 - 1999  input data
 2000 - 4999  process data
 5000 - 7999  output results
 8000 - 9999  subroutines
```

When you later go back to make changes It will be easier to find the problem code.

Multiple items may appear on a line if you separate each item with an "@" (commercial at sign). This is called concatenation. The advantages of placing a lot of code on one line are a savings of two bytes per item, (theorically) somewhat faster operation, saving paper when listing the program and because it is easier and more efficient to have a conditionally executed series of statements follow on the same line instead of using a GOSUB. The disadvantage is that it is nearly impossible to keep in mind the layout of a program with long lines with a 22 character window view of it.

Be sure that the code you are writing has the appropriate syntax for the long line; for instance, the 71 does not allow NEXT after ELSE. Perhaps the ideal way to work is to write and debug the program using many short lines. Then, when things seem to be going well, see how much can be fit on a line.

To make programs even easier to understand we use remarks. REM is a statement which tells the computer to ignore whatever else follows on the line. A more elegant looking way to say REM is with ! (exclamation mark), it also takes up less of our little window.

Remarks are used during program development to annotate code which may be confusing to read or designate parts of the program which haven't been written yet. If memory is at a premium (isn't it always?), the remarks may be deleted when the program is finished, but, a copy of the program with remarks should probably be kept to make it easier to change the code later. A method to help keep programs readable while conserving memory is to separate different modules in programs with a line with just a ! on it; memory cost is minimal.

```
 100 !                                  510 IF X THEN !
 700 ! FOR X=1 TO INF                   940 X=NOT X !
```

Readabiality also means entering code using easily understandable syntax. For example, line 9620 in "INCAT" could have been entered as:

```
9620 T$=T$[(NUM(T$[12])=32)*3]
```

There is a savings of a couple of bytes. But which line is more readable?

**The "DECIDE" Program**

This is not so much a program as a demonstration of programming. "DECIDE" is a decision making aid. Enter a series of up to 9 items and up to 9 factors. Press ENDLINE without any input when through entering and it will move to the next prompt. When the final data is input it will display the results one by one; press any key to move to the next item. The main purpose of the program is to demonstrate some HP BASIC principles.

```
  10 CALL DECIDE @ SUB DECIDE !                    create separate environment
  20 L=FLAG(-16) !                              recall current option base setting
  30 D=  (HTD(PEEK$("2F949",1)&PEEK$("2F948",1))/32  !     recall current DELAY
  40 OPTION BASE 0 @ INTEGER K(9,9) !              two dimension int. array
  50 OPTION BASE 1 @ DIM E$[7],T$(9)[8],F$[72] !     simple, array, long string
  60 L=FLAG(-16,L) !                                    restore option base
  70 E$=CHR$(27)&"H"&CHR$(27)&"J"&CHR$(27)&"<-" !       display clear string
     ! input loops ------------------------------------------------------------
1000 DISP E$&"Decide-" !                              start of program display
1010 Q$="item" @ GOSUB 'PROMPT' @ T=F !                 get items to evaluate
1020 FOR L=1 TO 9 @ T$(L)=F$[L*8-7,L*8] @ NEXT L !       move string to array
1030 F$="" @ Q$='factor' @ GOSUB 'PROMPT' !                   get the factors
1040 DISP "Repeat ratings Y/N";
1050 GOSUB 'WKEY' @ L=POS("NY",K$) @ IF NOT L THEN 1040 !    go get a keystroke
1060 L=FLAG(0,L-1) !                              clear flag 0 if repeating ratings
     ! rate each item
1070 DISP E$&"Rate:" @ FOR L=1 TO T @ ON ERROR GOTO 1080 @ FOR L2=1 TO F
1080 DISP FNT$(T$(L))&"("&FNT$(F$[L2*8-7,L2*8]);")="; !            display item
1090 GOSUB 'WKEY' @ K1=VAL(K$)
1100 IF FLAG(0) THEN 1130 !          if flag 0 set then don't check for repeats
     ! see if rating has already been used
1110 FOR L3=1 TO L-1 @ IF K(L3,L2)=K1 THEN BEEP @ DISP K1;"USED" @ GOTO 1080
1120 NEXT L3
1130 K(L,L2)=K1 @ NEXT L2 @ NEXT L
     ! calculate results -------------------------------------------------------
2000 FOR L=1 TO T @ FOR L2=1 TO F !                              calculate ratings
2010 K(L,0)=K(L,0)+K(L,L2) @ K(0,L2)=K(0,L2)+K(L,L2) @ NEXT L2 @ NEXT L
2020 K1=0 @ FOR L=1 TO T @ IF K1<K(L,0) THEN K1=K(L,0) @ L3=L
2030 NEXT L
     ! display results ---------------------------------------------------------
4000 DISP E$&"Results-" @ DELAY INF !           set delay to wait for keystroke
4010 DISP "Highest is "&T$(L3)
4020 FOR L=1 TO T @ DISP FNT$(T$(L))&' rated';K(L,0) @ NEXT L !      list items
4030 FOR L=1 TO F @ K1=0 @ FOR L2=1 TO T @ IF K(L2,L)>K1 THEN K1=K(L2,L)
4040 Y=L2 @ NEXT L2
4050 DISP "Top "&FNT$(F$[L*8-7,L*8])&" is "&FNT$(T$(Y)) @ NEXT L ! list factors
4060 DELAY 0 @ DISP "New data, Results, End"; !                  done, go again?
4070 GOSUB 'WKEY' @ ON POS("RNE",K$)+1 GOTO 4060,4000,50,8300
     ! subroutines -----------------------------------------------------------
     ! wait for a key --> replace with KEYWAIT$  or WTKEY$ <--
8000 'WKEY': DISP CHR$(27)&'>'; !                                 turn on cursor
8010 K$=UPRC$(KEY$) @ IF NOT LEN(K$) THEN 8010 !                  wait for a key
8020 DISP CHR$(27)&"< "&K$ @ RETURN !                turn off cursor, return
     ! main input routine
8100 'PROMPT': Q$=FNT$(Q$) !                              trim trailing spaces
8110 FOR F=1 TO 9 !                                        loop through inputs
8120 DISP "What is "&Q$&"#";F; @ INPUT K$
8130 IF NOT LEN(K$) AND F>2 THEN F=F-1 @ RETURN !         two items are enough
8140 IF NOT LEN(K$) THEN BEEP @ GOTO 8120 !                 need at least two
8150 F$[F*8-7,F*8]=UPRC$(K$) @ NEXT F @ F=9 @ RETURN
     ! user FN to trim trailing spaces from prompts
8200 DEF FNT$(K$)=K$[1,POS(K$&" "," ")-1] !          FNT$ trims trailing spaces
8300 DELAY D @ PUT "#43" @ END SUB !              restore DELAY, press ATTN, bye
```

**Strings**

Unless DIMmensioned at the outset, strings can contain up to 32 characters. The four strings used by "DECIDE" each have a different purpose and are created in a different way. K$ is not dimensioned, so it defaults to 32 characters. E$ is a constant, containing an escape sequence, and never changed in the program. T$ is a string array with 9 elements (since OPTION BASE 1 is set), each element can contain up to eight characters. F$ is a traditional (for HP) string, DIMmed quite large and used exactly the same as T$. Each item in F$ is eight characters, the same as an element in the T$ array. Let's display element n in F$ and array T$:

```
F$[n*8-7,n*8]                    T$(n)
```

A single string is often used instead of an array for applications which would seem natural for an array when we know that we will be doing string comparisons.

```
POS(A$,"!")      Finds the first ! in A$
POS(A$,"!",8)    Finds the first ! in A$ starting at position 8
LEN(A$)          Returns the length of the string.
```

Or extracting substrings. By giving us fewer but more versatile string functions, HP has enabled some incredibly complex string operations to be done almost automatically

```
A$[2,7]              Positions 2 through 7 of A$
A$[2,7][2]           Positions 2 through 7 of A$, then position
                     2 through then end of the resulting string
A$[POS(A$,"?",3)+1] Return substring following ?
```

```
  5 CALL FROG @ SUB FROG !              leap frog game. Jump only 1 char at a time
 10 A$='XXXX 0000' @ FOR T=1 TO 99 !     99 tries to reverse order of characters
 40 DISP USING 'ZZX">>>",9A,"<<<"';T,A$ !                  display current board
 50 Q$=KEY$ @ IF NOT POS("123456789",Q$) THEN 50 !      get position to jump to
 60 L=POS(A$," ") @ S=VAL(Q$) @ IF S=L THEN BEEP @ GOTO 50 ELSE DISP S;
 70 IF ABS(S-L)>2 THEN BEEP @ DISP 'Too far!' @ GOTO 40 ELSE DISP "-->";L
 80 B$=A$[S,S] @ A$[S,S]=A$[L,L] @ A$[L,L]=B$ @ IF A$="0000 XXXX" THEN 100
 90 NEXT T @ BEEP @ BEEP @ T=99 !                                 end of loop
100 DISP T;"moves" @ WAIT 1 @ DISP "go again Y/N" !     A good score is under 10
110 Q$=UPRC$(KEY$) @ IF NOT LEN(Q$) THEN 110 !                        replay?
120 DISP @ IF Q$#'N' THEN 10
```

## BASIC Programming Hints

There are usually several ways to perform the same task in BASIC. We can go from sloppy to concise to sacrificing clarity to save a few bytes and still reach the same goal. This section is about the point between "wow, it works!" and "I wonder what's on TV"; The program is functional, but let's conserve memory, make it a little more elegant looking and try to make it run faster. The two aspects of memory conservation are minimizing code used and economical allocation of variables; we will discuss both together. Using multi-statement lines (concatenated) is not discussed here; you should be comfortable using them before using the techniques discussed in this chapter.

There are a few negative aspects of using memory conservation methods. Remarks, for example, are invaluable for following program flow but use a lot of memory. And, since all BASICs differ, even between HP computers, some of these techniques will make it difficult to adapt programs to (or from) other machines. A bigger problem, even if you are using just the 71, is that programs can become nearly unreadable and therefore difficult to debug with the more extreme methods.

### Planning

We begin writing efficient code and saving memory before writing the first line of code. Flowcharts and pseudo-code are discussed elsewhere, but let's recap to say that organization and anticipating subroutines early on are critical to efficient programming.

### Variable Lists

As programs grow so does the number of program variables. A written list can be invaluable to see that X1 is X1 throughout and doesn't become X0 some place. The added benefit is that the list helps spot unnecessary vars or over-DIMmed strings.

### Saving Scratch

As an example, consider writing a spreadsheet program. We need a pointer for the current column, current row, cell format and so forth. The problem is that the program does so many different things that we can't afford to use separate variables for each section. We will reserve some variables to be used within the various modules and for passing information between them. Since simple variable refrences (like A,B,C) use one byte less each time they are refrenced than if they have an optional numeral (like A1,B2,C3) we try to use single letter names. In our spreadsheet we might use:

```
X= temporary X coordinate
Y= temporary Y coordinate
Z= general purpose scratch
L= temporary loop counter
```

Designating these variables as scratch means that they may be used by any routine within the program because they can generally be assumed to contain nothing important or "trash". They may also be used to pass information between modules; we pass the coordinates in X&Y, the subroutine does it's business then returns the results in, perhaps, Z. By the use of scratch variables we conserve memory (by reusing) and assure ourselves that important variables within the program are not accidently corrupted.

Regardless of the program being worked on, you might use the same names for various types of usages. This will make the program easier to read without having to constantly refer to the variable list. For instance, the author often uses Q and Q$ for inputs and A and A$ for results; L for loop counter and X,Y,Z for scratch. Usage of variables I (the letter eye) and O (the letter oh) look similar to 1 and 0 on many printers and are best avoided, though many people use I to represent Index or Iteration.

**Variable Names**

A numeral suffix in variable names costs 1 extra byte each time the variable is refrenced. X1=1 uses one byte more than X=1.

**OPTION BASE, DESTROY ALL, RESET**

Programs written for distribution should avoid selecting OPTION BASE 1 because it is a global setting and affects the operation of the entire variable chain. Even at the expense of wasting the possible zero element.

DESTROY ALL (which also destroys calculator variables) and RESET (which resets all user flags-including LC) should be avoided when possible for the same reason. An example is EDTEXT in the FORTH/Assembler ROM which sets uppercase mode; nice for Assembly source files but not for writing letters.

**String Arrays**

When first refrenced, if they haven't been DIMmed, string arrays are created to 11 elements (or 10 if OPTION BASE 1) of 32 characters each. They also use 3 bytes per element and 9.5 bytes for the array.

The following is excerpted from a commercially available HP-71 program. The line numbers have been changed to protect the innocent:

```
910 A$(1)="n "
920 A$(2)="i% " @ Y4=15
930 A$(3)="PV "
940 A$(4)="PMT "
950 A$(5)="FV "
```

These are the only values stored in A$(). Since OPTION BASE 0 had been established earlier on and the array had not been DIMensioned, it has, by default, used 32*11 bytes for the strings plus 31.5 bytes for the array; a total of 383.5 bytes to store 15 characters of information. If it had been DIMmed to 5 elements of 4 bytes each it would have required about 39.5 bytes plus 7 bytes for the DIM statement. As you can see, it is important to properly DIM string arrays before use. Now, about those short lines...

**DIM Strings**

It requires 5-7 bytes to DIM a string; There is some memory savings by not DIMming a string which will contain from 25-28 characters. The exception is ROM based software where it is worth spending 6 bytes of ROM to save 2 bytes of RAM.

**INTEGER, SHORT**

The 71 uses 9 1/2 bytes to store simple variables, regardless of their precision. Unless INTEGER is required to round a number, specifying INTEGER or SHORT precision will not save memory.

**Clearing an Array**

Often we may want to "zero out" a numeric or string array from within a program. Usually this is done with a loop:

```
1000 FOR L=0 TO 20 @ X(L)=0 @ X$(L)="" @ NEXT L
```

DESTROY followed by recreating the array(s) with DIM is much faster (the example is 5 times as fast) and is 46 bytes shorter. This second method is more work for the computer but, then, that's it's job.

```
100 DESTROY X,X$ @ DIM X(20),X$(20)
```

**User Defined Functions**

User Functions (DEF FN's) make it easy to pass variables to subroutines and use the results within a mathematical expression. Two disadvantages are that it is considerably slower than GOSUB and there is memory overhead for the environment in addition to the extra code required in the program. Instead of using:

```
1000 Z=FNX(Y)
```

We would assign the values to "scratch" variables (in this case A) and use GOSUB:

```
1000 A=Y @ GOSUB 9200 @ Z=B
```

If we have a commonly used routine, need to pass several parameters, want the advantages of nesting the FN or just want elegant looking code, then by all means use user functions. If execution speed and memory conservation are important then GOSUB is recommended.

**Nesting Mathematical Expressions**

While BASIC is an algebraic language, the 71 is internally a stack oriented machine vaguely reminscent of HP's RPN calculators. When an expression is interpreted the 71 passes parameters to the functions and places intermediate results in a section of memory called the Math Stack (although it is also used for strings). Let's start with a simple set of expressions. We want the current time in 12 hour (but decimal fraction) format; not the most useful value, but easy to explain. TIME returns the number of seconds since midnight.

```
10 X=TIME @ IF X>43200 THEN X=X-43200
20 DISP X/60/60
```

Besides saving some code, the following is considerably faster than this example. The speed increase is from simplified code and from doing it in one expression which keeps the numbers "floating" on the stack. Whenever a value is assigned to a variable then the expression is completed and the stack is not used to the greatest efficiency.

```
10 DISP MOD(TIME,43200)/3600
```

The way the HP-71 engineers use the math stack is also responsible for versatile handling of string subscripts. Since strings all go on the stack, they can be trimmed as we like them; extra parentheses may be added to designate a new string expression for which subscripts apply. This example takes the substring [2,4] of element 5 of array A$ adds the entire length of B$ to it then displays the results starting with the second character.

```
10 DISP (A$(5)[2,4]&B$)[2]
```

**TRANSFORM**

Lines as long as 120 characters are allowed by the 71 even though we can edit only edit lines up to 96 characters long. In addition to saving memory by allowing more information (such as after THEN), there is greater security of the code because the lines are uneditable. Unfortunately, they are also uneditable by you. Somewhat longer lines may be entered by deleting spaces between statements. A useful method is to write the program as a TEXT file then transform it to BASIC. Some other computers allow long lines so it might be useful to use another machine to write the program. Be forewarned that many people find program security methods disgusting.

**Quoted Strings**

Enter GOTO's to labels and HP-IL device specifiers without quotation marks.

```
GOTO label
GOSUB label
CAT :tape
CALL pgmname
```

A token representing quote will be entered by the 71 instead of the actual quotes so that (single) quotes will be displayed when you edit the line but a byte will be saved because there is no literal quote. If you later edit the line and press ENDLINE then the actual quotes will be entered and the savings will be lost so be sure to eliminate the quotes again.

**IF, THEN, ELSE**

GOTO is implied following THEN and ELSE if followed by a label or line number so the keyword GOTO is optional for a savings of 3 bytes. In this example the program will branch to line 800 if X#0, otherwise it will GOTO the label "start":

```
610 IF X THEN 800 ELSE START
```

An implied GOTO to a label may take any form which evaluates to a string. Note that this only applies to labels, calculated line numbers won't work (alas). In the two following examples the program will branch to label "A1" if X=1, to "B1" if X=2 and so forth.

```
340 IF X THEN CHR$(X+64)&"1"
440 IF X THEN "ABCDE"[X,X]&"1"
```

This syntax may be used with GOSUB and GOTO outside of the context of a conditional to provide highly flexible (and compact) branching.

```
90 GOSUB "A1B1C1D1E1F1"[X,X+1]
```

Instead of using a subroutine which may only be refrenced once, follow the conditional with the actual code, presuming it will fit.

```
500 IF NOT X THEN DISP "No value" @ BEEP @ X=.0001 @ Y=NOT Y
```

A disadvantage of IF, THEN and ELSE is that they restrict what may follow on the same line and therefore limit concatenation. One of the strengths of BASIC is that boolean arguments may be nested within mathematical expressions. In the example we will replace IF THEN with an argument which uses the same amount of memory but places no restrictions on what may follow on the line.

```
1000 IF X THEN Y=Y+100
```

If X=zero then we want to add nothing, and, since one hundred times nothing is nothing we can use the following:

```
1000 Y=Y+(100*X#0)
```

Other comparisons including NOT, MAX, MIN and MOD may be used in the same form.

If a variable is to be toggled between two values based on a comparison then assign the number to the second condition first and make only one comparison:

```
10 IF X THEN Y=1 ELSE Y=2
```

Replacing this with the following will save three bytes:

```
10 Y=2 @ IF X THEN Y=1
```

**NUM**

When using NUM with a substring, only the first character in the string will be used. NUM(Q$[5]) will suffice, the second subscript (as with Q$[5,6]) is unnecessary.

**ON ERROR**

One of the primary usages of ON ERROR is during an INPUT to trap bad data. Since any error which occurs will cause the branch then we can purposfully allow errors to happen to eliminate a series of IF THEN's.

```
 10 IF NOT X THEN 10
110 ON X GOTO "X1","X2","X3"
```

We could have used ON X+1 GOTO... but this wouldn't have helped if X=37. The simplest approach is to change the program to trap anything which may cause problems. In this example line 10 would contain code to interpret the error or provide the equivalent of an ELSE.

```
100 ON ERROR GOTO 20
110 ON X GOTO "A1","B1","C1"
```

We can also use ON ERROR to allow branching to different routines at an intentionally caused error, for instance:

```
100 IF NOT X THEN "" ! implied goto to a null string
```

**Optional Parameters**

Several statements may optionally be entered without parameters or in an abreviated form.

**CLEAR** HP-IL statement. Operates the same as CLEAR LOOP. Saves 5 bytes.

**DEGREES/RADIANS** OPTION ANGLE is optional, DEGREES or RADIANS is sufficient. Saves 4 bytes.

**RUN** Without parameters re-starts the same program.

**Recalling a displayed line**

Normally un-recoverable data may be assigned to a variable by turning on the cursor before displaying the data. Display the escape character (chr$(27)) followed by ">" and terminated by a semicolon to supress the CR/LF to turn on the cursor. The example assigns the CAT to C$. Line 520 is optional and is used to turn off the cursor again to keep from having the cursor (occasionally not even flashing!) present at odd times. The INCAT subprogram listed earlier is a practical application of this operation.

```
500 DISP CHR$(27)&">";
510 CAT @ C$=DISP$
520 DISP CHR$(27)&"<";
```

**String Operations**

These routines are not necessarily the most memory efficient way to do the operations, though they have been tested for speed.

**Filling a string with spaces**

A previously unused string (or one nulled by using X$="") can be filled with spaces to any length needed by placing a single space on it's extreme right.

```
10 DIM X$[200] @ X$[200]=" "
```

**Trimming Leading Spaces**

```
100 IF Q$[1,1]=" " THEN Q$=Q$[2] @ GOTO 100
```

**Trimming Trailing Spaces**

```
100 IF Q$[LEN(Q$)]=" " THEN Q$[LEN(Q$)]="" @ GOTO 100
```

**Centering a String**

Where Q$ is the string to center W is the width of the finished line and X$ is scratch. Fills the left of the string with spaces.

```
10 X$="" @ X$[(W-LEN(Q$))/2]-" " @ Q$=X$&Q$
                -or-
10 PRINT TAB(MAX((W-LEN(Q$))/2,1));Q$
```

**Replacing one Character with Another**

```
10 X=POS(Q$,S$) @ IF X THEN Q$[X,X+LEN(S$)-1]=S$ @ GOTO 10
```

**Reversing a String**

This routine is demonstrated as both a SUB and a DEF FN. The operation is the same in both versions. This function is designed to use only one string variable and is slower than the same operation using two strings. A SUB, which is somewhat faster, follows these two examples.

```
10 SUB REV(R$)  @ FOR L=1 TO LEN(R$)/2 @ P=LEN(R$)+1-L
20 R=NUM(R$[L]) @ R$[L,L]=R$[P,P]
30 R$[P,P]=CHR$(R) @ NEXT L @ END SUB

10 DEF FNR$(R$) @ FOR L=1 TO LEN(R$)/2 @ P=LEN(R$)+1-L
20 R=NUM(R$[L]) @ R$[L,L]=R$[P,P]
30 R$[P,P]=CHR$(R) @ NEXT L @ FNR$=R$ @ END DEF
```

The following has the same operation as the string reversing method listed above, but uses two string variables

```
10 SUB REV(R$) @ DIM A$[LEN(R$)]
20 FOR L=LEN(R$) TO 1 STEP -1 @ A$=A$&R$[L,L] @ NEXT L
30 R$=A$ @ END SUB
```

**Rotating Left**

While this is presented as a DEF FN, a SUB or a simple subroutine could use the same basic operation. The FN rotates the string left by the number of characters specified by X.

```
100 DEF FNL$(A$,X)
110 FOR L=1 TO X @ A$=A$[2]&A$[1,1] @ NEXT L
120 FNL$=A$ @ END DEF
```

**Rotating Right**

```
100 DEF FNR$(A$,X)
110 FOR L=1 TO X @ A$=A$[LEN(A$)]&A$[1,LEN(A$)-1] @ NEXT L
120 FNR$=A$ @ END DEF
```

**The Alternate Character Set**

Characters above ASCII 127 are displayed in inverse video on a monitor or using the alternate character set on the built-in LCD. The function HGL$, found in some LEX files, sets the high bit on all characters in a string about 50 times as fast as can be done in BASIC. The following method assumes that each character is below ASCII 128 to begin with.

```
FOR X=1 TO LEN(Q$) @ Q$[X,X]=CHR$(128+NUM(Q$[X])) @ NEXT X
```

**Creating the Character Set**

The two options of the following routine are Underlined and Inverse (white on black). It uses the KEYWAIT$ function. The program is 329 bytes long and builds the character set in blocks of 8 characters.

Note line 110 which uses GDISP to display alternate characters starting at uppercase "A". Since each character takes 6 bytes for the definition and "A" is CHR$(65), we begin displaying at 65*6, or position 390 in the 768 byte string.

```
 10 CALL CHARSET @ SUB CHARSET
 20 DIM C$[768],Y$[48] @ DISP "Underline/Inverse")
 30 T=POS("UI",KEYWAIT$) @ IF NOT T THEN 30
 40 DELAY 0 @ CHARSET "" @ FOR X=128 TO 255 STEP 8
 50 FOR Y=0 TO 7 @ DISP CHR$(X+Y): @ NEXT Y @ Y$=GDISP$[1,48]
 60 IF T=1 THEN 80
 70 FOR Y=1 TO 48 @ Y$[Y,Y]=CHR$(255-NUM(Y$[Y])) @
    NEXT Y @ GOTO 100
 80 FOR Y=1 TO 48 @ Z=NUM(Y$[Y]) @ IF Z<128 THEN
    Y$[Y,Y]=CHR$(Z+128)
 90 NEXT Y
100 C$=C$&Y$ @ DISP @ NEXT X @ CHARSET C$
110 GDISP CHARSET$[390] @ BEEP
```

**Alternate Character Set in Programs**

Prompts for input, title lines, and warnings have more impact when displayed in inverse video. The easiest way to enter these characters into a program is to assign them to keys. The easiest way is with HGL$, but it can be done (with a few more keystrokes) without that keyword. Don't forget the ";" to designate it as a typing aid key assignment.

```
DEF KEY "?",HGL$("?");
              -or-
DEFKEY "?",CHR$(128+NUM("?"));
```

**Waiting for a Key**

KEYDOWN and KEY$ give us the ability to wait for single keystrokes and use the key within the program. The program would look something like:

```
100 IF KEYDOWN("A") THEN 500
110 IF KEYDOWN("B") THEN 600
120 IF NOT KEYDOWN("C") THEN 100
```

In addition to the extra memory used, the 71 stays in the battery-slurping program running state regardless of how long it waits for the proper keystroke. INPUT goes to low power between keystrokes but requires **ENDLINE** to terminate the input, doesn't automatically qualify input and can't be used within mathematical expressions since it is a statement. Several LEX files contain the keyword KEYWAIT$ which does go to low power, waits for a keystroke and, unlike KEY$, always returns a string. Many plug-in ROMs have the function, you may already have it since it isn't always documented. KEYWAIT$ is a function which returns a string we can use in conjunction with POS or NUM to do some fancy branching. This will work best with single character key definitions.

```
10 DISP "Option :A/B/C/D" @ GOSUB CHR$(POS(T$,UPRC$(KEYWAIT$))+65)
```

Sometimes we may want to trap shifted or control keys; this is a little more complicated. Most shifted keys return two character strings and control keys return three or four characters. This example assumes there is an ON ERROR trap in case the user presses the wrong key.

```
500 K$=UPRC$(KEYWAIT$)
510 IF LEN(K$)=1 THEN GOSUB CHR$(POS(T$,K$)+65) @ GOTO 500
520 IF LEN(K$)=2 THEN GOSUB CHR$(POS(T2$,K$)+65)&"2" @ GOTO 500
530 GOTO 500
```

Another version of KEYWAIT$ is called WTKEY$ which returns the ASCII character of the key instead of the keymap value. It always returns a single character; for instance, ENDLINE returns CHR$(13) instead of "#38".

If one of these keywords isn't available (or we don't want to use any LEX files), we can define a user function to simulate it. Remember that this does not place the 71 in low power mode, but it does allow it to be used like KEYWAIT$. A KEYWAIT$ LEX file is about 55 bytes (WTKEY$ is somewhat larger) while the DEF FN is 42; the main savings is in the convenience of not needing the LEX file. The first example returns a null string if the user presses one of the shift keys. The second example (FNK1$) allows shifted keystrokes, but requires a string variable.

```
9000 DEF FNK$
9010 IF NOT KEYDOWN THEN 9010
9020 FNK$=KEY$ @ END DEF

9100 DEF FNK1$
9110 K$=KEY$ @ IF NOT LEN(K$) THEN 9110
9120 FNK1$=K$ @ END DEF
```

**Constants**

PI (3.14 etc) is a constant, and is part of BASIC because it is used so frequently. Programs often use other values time and again, often it will save memory to assign the number to a variable. Let's take the memory required to create a single digit constant variable:

```
Variable     9.5
Statement    5.0
            14.5
```

Entering a single digit number in a program (for instance: X=1) takes the 5 bytes, the same as recalling a variable (X=C). When we go to a two digit constant the "break even" point for using the variable is 16 usages. It is only when multi-digit numbers are used frequently that it becomes practical to use variables for constants.

We can thank the engineers for entering numbers in the program line with only as much precision as needed. The HP-75, for instance, enters numbers in a program in full precision even if they have a single digit. In fact, the most memory efficient way to enter the constant 1 on the HP=75 is with X=SGN(EPS).

**Inverting a Flag**

Flags are often used to represent a certain state which may change as the program runs. For this reason an efficient method to toggle between set and clear is a very high priority. The function FLAG sets or clears the flag accordingly but also returns one if the flag was not inverted, and zero if it was. Two common methods of inverting a flag are shown. The first method uses no variables, but limits what may follow on the line. The second method is four bytes shorter but requires a scratch variable to store the results of the FLAG function (unless we want to display an occasional spurious zero or one while the program runs, in which case DISP can be used).

```
IF FLAG(1) THEN CFLAG 1 ELSE SFLAG 1

X=FLAG(1,NOT FLAG(1))
```

**Flag Variables**

In the same respect we can invert a variable used as a flag quite simply. If a flag is to be inverted often in several different locations within a program and it isn't important that it be displayed (flags 0-4) then it will be more efficient to use a variable as the flag.

```
X=NOT X
```

**GOTO, GOSUB to a "label"**

Refrencing labels of four characters requires the same memory as refrencing line numbers. If a line number is refrenced several times then there is some memory savings in using three character (or shorter labels).

**END, ENDSUB**

The END and END SUB statements are not required if the program flows to the last line in the file or if the program is followed by another SUB in the same file. While a SUB may only have one END SUB, END may be used within the SUB to terminate it without the necessity of branching to the last line in the SUB.

```
10 SUB TEST
20 IF [expression] THEN END
30 GOTO 20
40 SUB TEST2
```

```
 5 CALL MATHQUIZ @ SUB MATHQUIZ @ E$=CHR$(27)&"H"&CHR$(27)&"J"
10 DISP E$&"Math quiz" !                                    start of program
20 ON ERROR GOTO 20 @ T=0 @ INPUT 'Largest number? ','10';M !    max unit?
30 M=ABS(IP(M)) @ IF NOT M OR M>999 THEN 20 !          make sure it is in range
40 DISP "function:  +  -  *  /" !                    prompt for type of function
50 F=POS("+-*/",KEY$) @ IF NOT F THEN 50 !            wait for proper keystroke
60 FOR Q=1 TO 10 !                                    loop through 10 questions
70 X=IP(RND*M) @ Y=IP(RND*M) @ IF X<=Y OR NOT Y THEN 70 !       get the numbers
80 IF F=4 AND FP(X/Y) OR Y=1 THEN 70 !          integer only answer if dividing
90 C=0 @ DISP E$&"Question number";Q @ ON ERROR GOTO 100 !     ask the question
100 IF F=1 THEN DISP X;"plus";Y; @ B=X+Y
110 IF F=3 THEN DISP X;"times";Y; @ B=X*Y
120 IF F=2 THEN DISP X;"minus";Y; @ B=X-Y
130 IF F=4 THEN DISP X;"divided by";Y; @ B=X/Y
140 INPUT A @ IF A=B AND NOT C THEN T=T+1 @ DISP "Very good!";B @ GOTO 180
150 IF A=B AND C=1 THEN DISP "Yes,";B @ GOTO 180
160 IF A#B AND NOT C THEN DISP "Sorry, not";A @ C=1 @ GOTO 100
170 DISP "The answer is:";B
180 NEXT Q !                                                  end of the loop
190 IF T=10 THEN DISP "GREAT! PERFECT SCORE" @ GOTO 220 !      display results
200 IF T>7 THEN DISP "Good!";
210 DISP T;"of 10 correct"
220 WAIT 2 @ INPUT "quiz again? ","Y";E$ @ IF UPRC$(E$)="Y" THEN 10 !    replay?
```

The 71 is a nibble oriented machine; many operations which take a full byte on some other computers can be done in a nib on the 71. Any location within the full memory address range of the 71 (which isn't in a private file) may be inspected using PEEK$. In addition, RAM locations may be altered using POKE. Remember that PEEK$ and POKE work with nibs which are a "half-byte", or four bits. A nib can have a (HEX) value of "0-F" which translates to 0-15 in decimal.

Many of the routines in this section use REV$. If you do not have this function available in your 71 it can be simulated with the REV subprogram in the String Operations section earlier in this book.

Let's look at System RAM. A chart at the back of this book lists most of system RAM. This is information the computer needs (such as the current PWIDTH) and some scratch space used by Assembly Language routines. Since the CPU reverses data when it reads and stores it, just about everything is stored nib-reversed or the whole location reversed. Since most of the system functions work in HEX internally, most information is in HEX.

For these reasons DTH$ and HTD aren't going to give an accurate conversion when the nibs are reversed. Remember that DTH$ fills with leading "0" so it will have to be trimmed to the proper size before POKEing. REV$ is almost a necessity when working with system RAM. Let's use the example of the location **DWIDTH**, two nibs which contain the current WIDTH setting. First let's set the WIDTH to 96:

```
WIDTH 96
```

Now, let's PEEK the two nibs at **DWIDTH(2F94F):**

```
PEEK$("2F94F",2)
      06
```

Converting 96 (decimal) to hex gives us "60". As you can see **DWIDTH** is backwards, or "nibble reversed". REV$ reverses the order of characters so that our "06" becomes "60". Now HTD (Hex To Decimal) can convert it to a Decimal value:

```
HTD(REV$(PEEK$("2F94F",2)))
       96
```

There is a problem going the other way (Decimal To HEX) because a five character, right justified, string is always returned:

```
DTH$(96)
      00060
```

We need two nibs. POKE always uses the full length of the string furnished to it; if we had just POKE'd the above then **DWIDTH** would have gotten the first two nibs ("00") and the next three nibs would have gone to the next higher addresses. The following shows what would happens if we POKE DTH$(96) to any (unnamed) location in RAM:

```
--> THIS IS AN EXAMPLE, DON'T DO IT <--
POKE DTH$(X),DTH$(96)          DTH$(X)   = "0"
                               DTH$(X+1)= "0"
                               DTH$(X+2)= "0"
                               DTH$(X+3)= "6"
                               DTH$(X+4)= "0"
```

As you can see, POKE places the first nib at the location specified, the second at the next nib higher in memory, and so forth, for the full length of the string.

**System RAM Chart**

The first item on each line of the System RAM chart is it's five digit (hex) location in RAM. Each location has a four to six character symbolic name. These names are used to identify the locations in documentation and for equate tables used by the Assembler. The third item is the number of nibs reserved for that utility, and finally, some remarks.

**ROWDVR(2E350)** These 16 nibs control the appearance of the LCD display. Slight modifications can give us bold characters which will stay in effect until we do an INIT 1 when the normal display returns. Be careful changing some of the nibs or the display can become pretty much unintelegible and various display flags (like PRGM and the shift annunciators) may be lit at odd times. The following are are reasonable looking character sets. They enhance the horizontal lines which is not as attractive as fatter vertical lines. Experiment with various combinations to give your 71 a "free form" look.

```
POKE "2E357","22"  Normal
                "63"  Bold
                "23"  Bold Top
                "62"  Bold Bottom
```

**DCONTR (2E3FE)** contains the current contrast setting as set by the CONTRAST keyword. The value is in hex so the possible contrast range is from 0-15. We can recall the current setting:

```
HTD(PEEK$("2E3FE",1))
```

The first value in the example is "2F3FE" which is the memory location, the second is the number one, the number of nibs we wish to see. Again, the function HTD is used to turn the hex number into decimal. In the same manner we can set the contrast.

```
POKE "2E3FE","F"
```

This set the contrast setting to 15 (which is "F" in hex). PEEK$ and POKE always use strings because all memory locations are five digit hex numbers which must be represented as strings.

**ATNDIS(2F441)** Used to disable the **ATTN** key so that it will not stop a program. The normal value for this location is "0". POKE an "F" here and **ATTN** will be treated like any other key by KEYWAIT$ and KEY$. Also, INPUT cannot be suspended. This is helpful when a program uses the **ATTN** key to, for instance, enter a command level (such as EDTEXT, the HP Text Editor), or if a program is doing a critical operation which could cause problems if interrupted. KEYWAIT$ returns "#43" and WTKEY$ returns CHR$(14) when you press **ATTN.**

```
POKE "2F441","F"   Disable ATTN key
POKE "2F441","0"   Enable ATTN key
```

**The Key Buffer**

Up to the previous 15 keystrokes can be saved in a location called the key buffer. This is why you can type ahead before an INPUT occurs and press keys faster than they can be displayed. Remember that the key buffer is emptied after each DISP unless the current DELAY is zero. The buffer is filled from low memory to high memory, the oldest key down is the one at the beginning of the buffer.

**KEYPTR(2F443)** Tells us how many keys are in the key buffer.

**KEYBUF(2F444)** is the beginning of the 30 nib (fifteen byte) key buffer. The first key is at "2F444", each additional key is plus "2" hex. Enter the following, without extra spaces, exactly as written for an example of how the key buffer is filled:

```
POKE"2F443","E"
```

The "P" is gone, but the other characters remain. If we had used "F" instead of "E" then the CHR$(13) entered when we pressed **ENDLINE** would have caused an "Excess Chars" error when the 71 tried to interpret:

```
OKE"2F443","F"
```

**Setting CMMD stack size**

The command stack usually has five levels, though it can be altered between one and sixteen entries. The 71 does not have a function for this operation, though it has been done in several LEX files. The stack can be altered between one and sixteen entries. This program limits it to fifteen entries because of some peculiar things which can happen with the full sixteen. CALL the program with a parameter, for instance: CALL CMDSTK(10) to set it to ten entries.

The following BASIC program is quite similar to SETCMDST (in the HP-71 Utilities Solution Book), but is listed here for those who do not have that book. Since the program alters system pointers, it cannot be interrupted during operation without a dreadful belly-up crash, so the **ATTN** key is disabled. Enter the program exactly as written (without remarks) and double check it before running.

```
 9 ! disable ATTN key
10 SUB CMDSTK(X) @ POKE "2F441","F"
19 ! make # cmmds 0< X <16,   find the command stack
20 X=MIN(15,MAX(1,IP(X))) @ A=HTD(REV$(PEEK$("2F576",5)))
29 ! create empty cmmd entries
30 DIM S$[X*6] @ FOR Y=1 TO X @ S$=S$&"000300" @ NEXT Y
39 ! set stack pointers
40 E$=REV$(DTH$(A+X*6)) @ POKE "2F580",E$&E$&E$
49 ! place empty commands in buffer
50 POKE DTH$(A),S$ @ POKE "2F976",DTH$(X-1)[5]
59 ! enable the ATTN key,  bye
60 POKE "2F441","0" @ END SUB
```

**Display Devices**

If there is a display device active then a program may want to support it. The usual (and HP recommended) method to find the device is by checking the loop for one. The disadvantages of this are that if there is no HP-IL module then it will cause an error when the function is encountered. Another problem is that if the loop is broken then everything will get hung up waiting for the HP-IL ROM to realize it. Regardless, the operations are quite time consuming. The following method is quite fast and cannot cause an error. Two locations are used by the 71 when dealing with display devices.

**DSPCHX(2F674)** contains five nibs which relate to various aspects of the device. If the most significant bit of the first nib is set (the PEEK$ is "8" or greater) then a device is active. If a non-HP display device is being used then generally the 71 will clear bit 2 which will cause insert and delete to not re-display the remainder of the line.

```
Bit 3  Set if Device is active
Bit 2  Set if HP82163 Video Interface is active
Bit 1  Set if output is to a printer (send full lines only)
Bit 0  Set if display is on
```

**IS-DSP(2F78D)** is used by the HP-IL ROM to describe the device. This is much like IS-PRT for the printer, and **IS-PLT** for the plotter (the what?). The first two nibs are the address of the device; values of "10" (decimal 1) to "E1" (nib-reversed hex for decimal thirty) mean a valid address. Outside of that range means not a good device.

We can take these two pieces of information and determine if a display is active, and where it is. This routine has also been done in Assembly language as a function called TVIS (issued to the author). In the following example variable D will contain either the address of the display or zero if there is none, it isn't active, there is no HP-IL module, or the device is assigned with extended addressing.

```
10 D=HTD(REV$(PEEK$("2F78D",2))) @ IF D>30 THEN D=0
20 D=D*(HTD(PEEK$("2F7B1",1))>=8)
```

**PEEKing at Flags**

User and system flags are stored in system RAM in a contiguous block. One of the nicest uses for PEEKing and POKEing flags is to save current configuration, alter the machine as needed, then restore the former conditions when through. While SFLAG and CFLAG won't work with system flags, there is no restriction when using PEEK$ and POKE. **Be forewarned that altering some system flags (as with POKEing anywhere) will** lock up the computer beyond INIT 3.

**SYSFLG(2F6D9)** System flags.
**FLGREG(2F6E9)** User flags.

Flags are stored in order with four flags per nib, therefore the smallest unit we can alter is four flags (one nib). Clearing a block of user flags by POKEing is much more flexible than using RESET and faster than individually altering several flags. It is more memory efficient only when altering more than about 12 flags in a statement (assuming the flags could be changed using SFLAG or CFLAG). This example will set flags 0-3:

```
POKE "2F6E9","F"
```

Now, to clear those same four flags enter:

```
POKE "2F6E9","0"
```

Since we are dealing with locations in RAM, the next four flags (4-7) are located plus 1 nib at 2F6EA flags 8-11 are at 2F6EB and so forth. A chart in the back of this book lists the locations of system flags and their uses.

**LOCKWD(2F7B2)** The security password set by the LOCK statement is stored in eight bytes, nib reversed with the most significant byte in lower memory. Be sure not to POKE any characters which cannot be entered from the keyboard (such as several CHR$(10)'s) into this location because you will not be able to turn the 71 back on. To view the current password:

```
100 Q$=CHR$(126) @ FOR X=HTD("2F7B2") TO X+14 STEP 2
110 Q$=Q$&CHR$(HTD(REV$(PEEK$(DTH$(X),2))))
120 NEXT X @ DISP Q$&CHR$(126)
```

**RESREG(2F7C1)** The results register can contain a REAL number, stored backwards in BCD (Binary Coded Decimal) in the first 16 nibs at **RESREG.** A complex number (if there is a Math ROM) uses the full 34 nibs. The internal representation of a 16 nib BCD number is as follows:

```
S M M M M M M M M M M M M X E E E
```

From left to right, the "S" stands for the sign. Following this is the 12 digit mantissa ("M"), the sign of the exponent("X"), then the three digit exponent("E"). The decimal place is implied to be after the first digit (from the left) in the mantissa (though it doesn't really exist). This standard is followed with any REAL number. For more information on numerical fields refer to the section on Assembly programming.

Since the RES register changes whenever a number is displayed or assigned to a variable, this location will constantly change. Try some values with this little program:

```
10 DISP REV$(PEEK$("2F7C2",16))

INF:        0999999999999F00
EPS:        0100000000000501
 PI:        0314159265359000
-PI:        9314159265359000
```

**ERR#(2F7E4)** These four nibs are the hex (reversed) equivalant of the value ERRN returns.

**ERRL#(2F7EC)** This is the same value as returned by the ERRL function. The line number is stored reversed in four BCD (not hex!) nibs.

**Display/Print settings**

SCROLT(2F946) The 2 nib representation of the scroll setting. This is the second parameter of the DELAY statement. The value is the number of 1/32nd's of a second (.03125 sec. for decimal fans) in nib reversed hex. INF is stored as "FF".

**DELAYT(2F948)** The first parameter in the DELAY statement. Stored in the same format as **SCROLT.**

**DWIDTH(2F94F)** Current WIDTH setting in two reversed hex nibs. INF is "00".

**PWIDTH(2F958)** The current PWIDTH setting in the same format as **DWIDTH.**

**Determining program size**

As you know, the 71 works with nibbles so that an operation which may take full bytes on some machines can be done in multiples of nibs. We can access how changes have affected program size by beginning the program with:

```
1 DISP MEM @ END
```

This tells us available memory. But that may have changed because of other causes besides editing the program. We could also use CAT, but that is often off by one nib. The following routine returns the size of the program to the nib:

```
1 DISP (HTD(REV$(PEEK$("2F567",5)))-HTD(REV$(PEEK$("2F562",5))))/2-49 @ END
```

What this line does is subtract the end of the file from the beginning, then divide it by two to turn it into bytes, then it subtracts 49, the size of this line. This line can be changed into a remark when not needed and, or course, should be removed when the program is finished.

**Program Memory usage**

Another routine can be used to determine the amount of memory used by variables (which are not counted in the program size). Of course, we should already know what free memory was available before the program was run.

```
1 DISP  (HTD(REV$(PEEK$("2F599",5)))-HTD(REV$(PEEK$("2F594",5))))/2-106.5
```

**Finding the Card Reader**

If there is no Card Reader then the following will return "0", other values mean there is one. This is not part of System RAM, but memory allocated for the Card Reader.

```
PEEK$("2C014",1)
```

**Strings in SDATA Files**

We can read either numbers or strings from SDATA files, however, the 71 only allows writing numbers to these files. Since a string can be read there is no reason we can't write strings, hence this section. The format used is, to say the least, unusual looking; physically seven characters can be used, though READ# will only recognize six (for HP-41 compatibility); a byte is wasted. We'll discuss the actual register (record) format in a moment.

The SDATA file has a 37 nib header followed by as many 8-byte registers as specified. The file can be expanded by RESTOREing to the end of the file then using PRINT#n;n. To store a string in an SDATA file you must first place a number (pick a number, any number) in that register, then POKE a string over top of that number. The reason for first entering a number is to make sure the file is large enough for the string. Remember that registers begin with register# 0. The file doesn't have to be open (It isn't necessary to use ASSIGN# ) in order to replace a current register with a new string. First let's create an SDATA file and give it three registers:

```
CREATE SDATA TEST @ ASSIGN #1 TO TEST
PRINT #1;0,0,0
```

Next turn "QUACK!" into the SDATA Text format:

```
Q$="QUACK!" @ CALL SDTEXT(Q$)
```

Third, find out where the first register in the SDATA file is:

```
A=HTD(ADDR$("TEST"))+37
```

Now put "QUACK!" in register 0, which begins at the first nib past the header, then read the file to confirm it's there:

```
POKE DTH$(A),Q$
RESTORE#1 @ READ #1;X$ @ DISP X$

       QUACK!
```

Let's print "HP-71" to register #1, the second register, which is +16 nibs from the beginning of the file.

```
Q$="HP-71" @ CALL SDTEXT(Q$)
POKE DTH$(A+16),Q$
RESTORE#1 @ READ#1;Q$,A$,Q
DISP Q$,A$,Q

       QUACK!    HP-71     0
```

The SDATA file now has three records (OK, registers), the first two are strings and the third still has the value of zero.

**The SDTEXT SUB**

The string passed to SDTEXT must be DIMmed at least 16. Only up to the first six characters will be returned in a sixteen character string suitable for POKEing into an SDATA file. Of course, this can be used as a subroutine in a program to save some time and memory. If this routine is added to a program instead of being used as a SUB, be sure that X=0 before entering. SDTEXT always returns a 16 byte string which is the correct length for POKEing into a single register. The program uses REV$, as usual.

```
10 SUB SDTEXT(Q$) @ Q$=Q$[1,6] @ A$="0010000000000000"
20 X=2+X @ C=NUM(Q$) @ Q$=Q$[2] @ IF NOT C THEN 60
30 A$[16-X,17-X]=REV$(DTH$(C)[4]) @ IF X<8 THEN 20
40 IF LEN(Q$) THEN A$[4,4]=DTH$(NUM(Q$))[5] @ A$[7,7]=DTH$(NUM(Q$))[4,4]
   @ Q$=Q$[2]
50 IF LEN(Q$) THEN A$[1,2]=REV$(DTH$(NUM(Q$))[4])
60 Q$=A$ @ END SUB
```

**SDATA Register Format**

Let's use the register as it appears in memory (which is backwards, as usual). The HP-41 has 10 digit accuracy and a two digit exponent, while the HP-71 has 12 digit mantissa and three digit exponent, so some differences are found in the handling of SDATA files by both machines. Physically, first come the last two digits of the exponent then the sign of the mantissa (which is also the flag for a string: 0=positive, 9=negative, 1=string), then the first digit of the exponent (which is not used by the HP-41). The last byte of the mantissa (lower case "m"'s) is not recognized by the HP-41 so a string byte there would be ignored).

Strings are stored nib reversed, but also the two nibs of character number 5 in the string are separated by a null byte. The first digit of the mantissa is zero and the sign is one for strings.

```
          low mem                    high mem

          E E S E m m M M M M M M M M M M
    CHR#  6 6   5      5 4 4 3 3 2 2 1 1


          PI           ØØØØ953562951413
          PI(HP-41)    ØØØØØ04562951413
          9E27         72ØØØØØØØØØØØØØ9
          .23456       99Ø9ØØØØØØ65432
          -1Ø          1Ø9ØØØØØØØØØØØØ1
         "HP-71"       ØØ1100373D205840
         "QUACK!"      121BØØ4341455150


           12   1  B  ØØ  4 34 14 55 15 Ø
           || sign |      | || || || || ||
           ||       \    /  || || || || ||
           ||        \  /   || || || || ||

           !          K     C  A  U  Q
```

```
10 CALL GRAPHIX @ SUB GRAPHIX @ DIM C$[132] !              GDISP demonstration
20 FOR L=1927 TO 2027 STEP 2 !    lets look at ROM characters, including cursors
30 C$=C$&CHR$(HTD(REV$(PEEK$(DTH$(L),2)))) !                accumulate the pattern
40 GDISP C$&GDISP$ !    display the pattern plus what is already being displayed
50 NEXT L @ C$="" @ GOTO 20 !   Program will loop forever, don't get hypnotized!
```

## Converting from other BASICS

As BASIC has grown it has followed divergent paths. Microsoft has been the dominant force because of shear number of units sold. There are probably more computers running Microsoft BASIC (registered Trademark of Microsoft Corp.) than any other implementation or any other language. In this discussion we will lump BASIC together into two camps: HP, and everybody else; Apologies to Dartmouth (where BASIC originated) for this generalization. Most of the discussions here apply to IBM PC, APPLE, TRS-80, Atari and Commodore. Sinclair and other propriatary variations are intentionally "included out". We will also discuss the HP-75 to compare BASICs evolution within HP's own walls.

HP has taken other paths to the point that most published BASIC programs won't run on HP computers without considerable changes. Northstar computers have the most HP-like BASIC. BASIC, even MS BASIC, is not a standardized language. It can be said that Microsoft BASIC and HP BASIC are two different languages with similar syntax. This is not a tutorial; every example will not work in all cases.

Don't expect any PEEK$ or POKE from any other computer to work on the 71. Try to find out what the operation does then look for either an HP keyword or try to find an equivalent PEEK in the chart in the back of this book. Not all PEEKs will have an HP-71 counterpart.

### Microsoft BASIC

The differences between Hewlett Packard and Microsoft BASIC are as much philosophy as code; generally speaking, HP has more keywords than MS because of a desire to make programming easier. Where MS will require a POKE, HP will have a keyword; in fact, while HP usually does a very complete version of BASIC, some HP's don't even have PEEK and POKE. Both languages use tokenized code, but HP tokenizes and checks syntax as code is entered, thus, while there may be a short delay after pressing ENDLINE, most typing errors are caught immediately, instead of while the program runs. If you rave about MS BASIC then you aren't familiar with HP BASIC. If we had a computer with both BASICs running a similar program, the HP language version would operate considerably faster. Thus, while the 71 actually runs fairly slow (for fuel efficiency), programs run relatively fast because of the slick way they are coded.

Numerical precision and round off error are usually quite extreme with MS, while the 71 uses quite accurate BCD routines. Be sure that a given line is not expecting the sloppiness of MS to limit number of loops or to induce an error purposfully.

Varible names are often quite long with MS. Expect variables named "LOOP" or maybe "DAYS". These are not keywords, even though they often look like them. Replace them with the usual character or character and a number variable names (A, A1...) the 71 can accept.

With all of that aside, let's concentrate on similarities and converting programs. The two most obvious differences that can be seen at first glance are string functions and the use of the "@" (commercial at) instead of ":" (colon) to delimit statements on concatenated lines.

HP also uses "&" (amperstand) to concatenate stringes where MS uses "+" (plus), the same operator used with numbers.

```
MS:  "str1"+"str2"
HP:  "str1"&"str2"
```

### Graphics and Escape Sequences

HP-71 BASIC does not have graphics functions (except GDISP and GDISP$, but they are quite unlike Microsoft). Most escape sequences are quite similar with one exception which will require major surgery. Placing the cursor at a specific location on the display is done with CHR$(27) followed by a three character command.

MS specifies CHR$(27) then uppercase Y followed by a character specifing the row coordinate and a character for the column coordinate. The col/row coordinates are

offset by 31 so that the first row is CHR$(32), the second is CHR$(33) and so forth.

HP uses CHR$(27) followed by the percent character (%) followed by col then row specifyier characters. The coordinates begin at zero for col 1 and row 1. The first row is CHR$(0), the second is CHR$(1).

The scheme used by MS assures that most coordinates can be represented by displayable characters, so the program will often contain literal strings. In the example C is the column and R is the row coordinate.

```
MS:  CHR$(27)+"Y"+CHR$(R+31)+CHR$(C+31);
HP:  CHR$(27)&"%"&CHR$(C-1)&CHR$(R-1);
```

**CLEAR**

This does not clear the display. It is used by MS to free an area of RAM for strings. Unless otherwise DIMmed, strings share a common buffer which is usually about 256 characters by default. This statement is used to increase the size of this buffer. Check the maximum string length used in the program and DIM strings individually.

**CLS, CHR$(27)+"E"**

These are two alternative methods of clearing a display device by MS and is one instance where HP has not included a keyword!

The simplest method would be to use CHR$(27)&"E" which resets both display device and the LCD. The problem with this method is that it also turns the cursor on. When you clear the display using CHR$(27)&"E" and follow it by an INPUT then the prompt string will be included within the default input. Even if an INPUT does not follow, the flashing cursor can be a distraction, so a more useful solution is to use CHR$(27)&"H"&CHR$(27)&"J" which homes the cursor then clears the display from that point; if the cursor had been off (quite likely) then this operation will not turn it on.

```
MS:  CLS
MS:  CHR$(27)+"E";
HP:  CHR$(27)&"H"&CHR$(27)&"J";
```

**CHR$(27)+"p", CHR$(27)+"q"**

MS uses escape plus lowercase "p" to enable inverse video (black characters on white background), then escape plus lowercase "q" to restore the normal display. There is not an exact counterpart with HP.

HP uses characters above CHR$(127) for inverse video on a monitor or for the alternate character set on the LCD. We can display any character in inverse video by adding 128 to it's ASCII value. For instance, "A" is CHR$(65) and inverse "A" is CHR$(65+128). This must be done on a character by character or string by string basis (unlike MS which displays everything inverse until told not to).

HGL$ is in some LEX files (including the one in WorkBook71) and may be used to quickly "set the high bit" (add 128 to each character). As an alternative this simplified line may be used:

```
FOR X=1 TO LEN(S$) @ S$[X,X]=CHR$(128+NUM(S$[X])) @ NEXT X
```

**!  '  REM**

The use of REM to designate a remark is universal in BASIC. While MS uses a single quote (') as an alternative, HP uses the exclamation mark (!).

```
MS: REM this is a remark
HP: REM this is a remark
MS: ' this is a remark
HP: ! this is a remark
```

" " ' ' (Quoted String)

Microsoft accepts only the double quote character (") to delimit strings while HP allows single or double quotes. As discussed above, a single quote (') in MS designates a remark.

If a string is either the only or last item on a line then MS allows the closing quote to be optional, while it is required by HP.

```
MS:  PRINT "a string
MS:  PRINT "a string"
HP:  PRINT "a string"
HP:  PRINT 'a string'
```

**ASC**

Used to recall the ASCII value of a character.

```
MS:  N=ASC(X$)
HP:  N=NUM(X$)
```

**EOF**

This function tests to see if you have reached the end of a file; a true result means you have. There is no equivalent function for the 71. Use an ON ERROR trap to branch when the computer generates ERRN 32 ("No data").

**FRE**

An example of a function which really doesn't need a parameter, but MS requires one anyway because of the limited parser. This is the same as MEM.

```
MS:  FRE(0)
HP:  MEM
```

**INSTR**

Compares two strings and returns the a number representing the position within the match occured or zero if there was no match. Note that the POS keyword is also used by MS but for a different purpose.

```
MS:  P=INSTR(<start at>,"abcde","cd")
HP:  P=POS("abcde","cd",<start at>)
```

**KILL**

Hardly a term GreenPeace would appreciate. HP is much more humaine in their usage of PURGE:

```
MS:  KILL "filename"
HP:  PURGE filename
```

**LEFT$**

Returns the specified number of characters from left most portion of the string.

```
MS:  X$=LEFT$(X$,5)
HP:  X$=X$[1,5]
```

**MID$**

Returns a substring beginning at the starting position and including the specified number of characters. MS will usually pad with spaces on the right if the number of characters specified is longer than the string, HP-71 will not add the spaces.

```
MS: X$=MID$(X$,S,7)
HP: X$=X$[S,S+7]
```

**OPEN "filename" FOR APPEND AS #n**
**OPEN "filename" FOR INPUT AS #n**
**OPEN "filename" FOR OUTPUT AS #n**

Many dialects of MS BASIC require you to designate what you are going to do with a file (read from or write to it) when it is assigned. In the case of all of the examples above use:

```
HP:  ASSIGN #n TO "filename"
```

**PRINT, LPRINT, ?**

Keywords for displaying information stem from large machines with terminals connected to them, therefor you would PRINT to your Terminal. "?" is an abreviation of "PRINT" and is used by most dialects of BASIC (the 71 uses "?" for a numerical comparison). HP has added the keyword DISP to simplify writing and understanding things, and eliminate the ambiguity of where the the information is to go. HP uses PRINT to mean send this to a printer while MS may use LPRINT (meaning "Line Print" because it is destined to be printed on a line printer). Occasionally a software switch is used to designate that the PRINT information is to be sent to a printer (less common).

```
MS:  ? "display this"
HP:  DISP "display this"
MS:  PRINT "display this"
HP:  DISP "display this"
MS:  LPRINT "print this"
HP:  PRINT "print this"
```

**PRINT AT, PRINT @**

Print at is used to locate the cursor then print a string. Read the section on Graphics and escape sequences for conversion.

**RIGHT$**

Returns the specified number of characters starting from the right most portion of the string.

```
MS:  X$=RIGHT$(X$,5)
HP:  X$=X$[LEN(X$)-5]
```

**READ, OPEN, INPUT#, INPUT$ CLOSE#**

With the exception of INPUT$, file functions have direct counterparts. The 71 must create a file before using ASSIGN# while MS (and many variations of HP BASIC) implictly creates one if it doesn't exist.

INPUT$ (syntax is Q$=INPUT$(<channel>,<#chars>) is used in MS to read Text files which may or may not have a carriage return at the end of each line, it usually reads a single character at a time then adds it to an output string. A file could conceivably not have a single carriage return, in which case a string could not contain the entire line read by INPUT#. Since each line in a 71 file is of a finite length this function is not needed so it is safe to "blunder in" and read a whole line.

```
MS:  OPEN "file" FOR <INPUT,OUTPUT> AS # 1
HP:  ASSIGN # 1 TO "file"
MS:  INPUT#1,X$
HP:  READ #1,X$
MS:  CLOSE #1
HP:  ASSIGN #1 TO *
```

**HP-75 BASIC**

The HP-75 is a sibling which came into production about 18 months before the 71. It has BASIC in ROM, 32 character single line display, ROM and RAM ports, HP-IL interface and a Card Reader. While the HP-71 can be traced back to the HP-41, the 75 descended from the series 80 (specifically the HP-85) and a different engineering team. HP-85 programs (such as found in solutions books) are similar to HP-75 programs. HP-75 is most similar to HP-83 and HP-85 BASIC, while the 71 is most similar to HP-86 and HP-87.

BASIC as furnished with the 75 has fewer keywords than the 71 and can be thought of as a sub-set of it, with conversion being fairly easy for primarily mathematical programs. Most HP-75 owners have the I/O ROM (with 150 keywords) and VisiCalc ROM (VisiCalc is a registered trademark of VisiCorp 83 keywords). Programs using either of these ROMs will be more difficult to convert, in the latter case often nearly impossible though WorkBook71 by this author has a LEX file which makes it easier). As a side benefit, the converted program will take 20-30% less memory on the 71 than it did on the 75.

Many 75 programs use CHR$(27)&"E" to clear the display device. To avoid unnecessarily turning on the cursor, this should be changed to:

```
CHR$(27)&"H"&CHR$(27)&"J";
```

**Transferring files**

Connect the 71 and 75 together using HP-IL. Set CONTROL OFF and edit a new file on the 71. Designate the 71 as the PRINTER IS device on the 75, place the 71 in REMOTE mode, then PLIST the program. It is suggested that the program be edited on the 75 first to look for lines which will not be interpreted properly, and place an exclamation mark at the beginning of these lines. The 71 will beep and respond with an error message whenever it cannot interpret a line, and the line will be lost so it is best to "comment" these possible offenders first.

```
PRINTER IS ":C1" @ REMOTE ":C1" @ PLIST "filename"
```

Programs may also be transferred using Cassette, Disc, or even Card Reader. The common file type is called LIF1 on the 75, which corresponds to TEXT files on the 71. TRANSFORM the file to the specified type then merely save it to the medium. When sending a TEXT file to the 75 be sure each line begins with a 4 digit line number (leading zeros for <1000) or the 75 won't be able to TRANSFORM it.

**File handling**

The file chain on both machines is operationally the same (although internally they are totally different). On the 75 all editing commands (including EDIT and FETCH!) are programmable and the current edit file is often not the file being run. While READ (without specifing a file number) refers to the running program, not edit file, DELETE, RENUMBER, PURGE etc refer to the current edit file. These operations can usually be simulated with ASSIGN#, PRINT# and READ#. On the 75, new files are created using EDIT or ASSIGN#; if they do not exist they will be created as the same type of file as the current edit file if the type is not specified; the default is not DATA as on the 71.

File types that may be edited are BASIC and TEXT (with line numbers). On the 75 a DATA file is actually a BASIC program file in which each line begins with DATA. DATA and TEXT files may be read and printed to randomly by specifying line numbers. Intermediate lines are not required, you may, for instance, enter data on line 9000 without there being data on any other line. Files automatically grow and shrink as needed. If data is read from a non-existant line an error is generated. Since TEXT lines begin with a number, most programs writing to TEXT files insert a leading space to separate the line number from possible following ASCII numerical characters, these are obviously not needed on the 71. Random read/write may be simulated by using the INSERT#, DELETE# and REPLACE# commands in the EDLEX file furnised with the 71 FORTH and Text Editor ROMs. Unlike the 71, the 75 will not place an EOF marker at the end of the current PRINT# line in a TEXT file. PRINT#n,n;"" will erase a record on the

75 while it will make a blank (though still existant) record on the 71. DATA files would be easier to use if created to the maximum size the program will need then filled with place holding data (null strings, spaces or zeros) so that random read/write will work properly.

```
HP-75: ASSIGN #1 TO "filename", TEXT
HP-71: CREATE "file" TEXT @ ASSIGN #1 TO "filename"
```

**Interpreting keystrokes**

KEY$ operates as with the 71 except that it returns a single ASCII for any keystroke; **RTN** ( **ENDLINE** on the 71) returns CHR$(13) while the 71 will return "#38" and **UP ARROW** returns CHR$(132) instead of "#50". The 75 has several keys without counterparts on the 71 (TIME, APPT, FET, CLR, TAB). SKEY$, WKEY$ and KEYWAIT$() from the two previously mentioned ROMs may be substituded by KEYWAIT$ or a user function (described elsewhere). Read the program documentation for the program and alter the keys accordingly.

**HP-IL**

Devices (display, Disc, etc) are always refrenced by a quoted string or variable name, the same as used by ASSIGN IO on the 71, device words (such as ":MASSMEM") are never used. Generic device names (again, such as ":MASSMEM") are easy to substitute for assignments the 75 gives to devices. The following are the default names given devices by the HP-75 I/O ROM and the HP-71 equivalent. Additional devices of a given type are incremented to the next number; beyond number 9 they next move letters beginning with A.

```
 75                71
:A1(analytical)
:B1                :HPIB
:C1                :HP71
:D1                :DISPLAY
:E1(elect.Inst)    :INSTRMT
:G1                :GRAPHIC
:I1                :INTRFCE :GPIO :RS232
:M1                :TAPE :MASSMEM
:O1(general)
:P1                :PRINTER
:U1(unknown)
:X1(extended)
```

## Assembly Language Introduction

This is an introductory discussion of some of the concepts of Assembly Language programming on the 71. Even if BASIC solves all of your programming problems you might find this section to be interesting reading to get a feel for the interraction between interpreted BASIC and the actual Machine language which it invokes. This section will deal with Assembly Language at a fairly high level, and should get the average user started writing Functions without having to purchase any of the IDS.

At it's lowest level, a computer is not even a very good calculator; it can't even multiply or divide properly. It can add, subtract, make comparisons and shift data around. This is the level at which machine code operates. To make it even more of a challenge, we can't even edit machine code directly.

To overcome what must seem like an insurmountable task we have an Assembler. This is a program which read a Text file and creates machine code from commands within that file. And, so we don't have to re-invent the computer every time we sit down to write even the simplest thing, the HP-71 has a library of utility programs within it's 64k operating system. These utilities operate much like keywords in BASIC. With these utilities, inspiration and some ingenuity comes LEX and BIN (binary) files. For clarity, it might help to refer to the finished code as Machine code (or language) and the Text file as Assembly Source code. Because Functions are most usable for most users, we will discuss writing Functions and leave statements, polls and interrupts to the more adventuresome users with access to all volumes of the IDS (Internal Design Specifications).

HP-71 LEX files are used to either respond to polls (which the operating system issues when, for instance, errors occur), or extend BASIC with new keywords, or both. With LEX files, HP-71 BASIC remains a living and growing language. When new concepts in programming are discovered or often used or tedious routines are found, new BASIC keywords can be created to implement them.

BIN files are RUN or CALLed in the same manner as BASIC programs. While BASIC programs are interpreted, BIN files contain executable code rather than BASIC tokens. Advantages of BIN files are faster execution, keeping private code private and doing things which are difficult to do in BASIC. You may not find extreme speed gains over BASIC in many operations because you usually call the same code that BASIC uses. Direct access to hardware such as beeper, display, keyboard and all aspects of I/O are primary reasons for using BIN files. While some argue that BIN files are easier to write than LEX, the following discussion is directed towards LEX files because of their versatility.

Assembly language is written as Text files which are then interpreted by the Assembler which directly creates executable BIN and LEX files. The FORTH/Assembler ROM is the usual method for creating these files. Although there is an Assembler available for use on the HP series 200 machines, don't expect to walk into your dealer and buy it.

Since the source files are Text they may be written on any machine which can be made to communicate with the HP-71. Whatever machine you use be sure that the text editor does not imbed control codes within the text. For instance, some computers use CTL-I (ASCII 9) for the TAB character instead of accumulating spaces.

Since comments are a vital part of writing Assembly code, the source text file is often quite large. One known LEX file of about 800 bytes has a source file of approximately 9k without remarks and 18k with remarks. If memory is at a premium, write and debug the file in sections then do the final assembly from a Disc based file containing all of the completed modules. The Assembler can take twenty minutes or more for a large file, be sure that all of the batteries involved aren't waiting to surprise you.

### The HP-71

The 4-bit Capricorn processor is a decendent of the 1-bit CPU which has powered over a million HP-41's since 1979. The four bit data path combined with higher clock speed give a considerable speed gain over the 41. It is a highly evolved processor with four 64-bit working registers enabling us to do BCD (binary coded decimal) math

with reasonable speed. five 64-bit scratch registers and two 20-bit data-pointers help enforce the look that the processor might be as much math co-processor as CPU. These big floating-point registers help insure that HP-71 BASIC won't be subject to the rounding errors of most versions of Microsoft BASIC.

The address space is 1024K nibs, or 512K bytes. The first 64K bytes are the operating system and BASIC. Addresses are always referred to in 5 digit hex, either in BASIC or Assembly Language. After operating system ROM comes memory-mapped I/O and the display RAM. Following that area is the hard addressed (can't be moved) System RAM. A memory map and listing of System RAM are at the back of this book.

**CPU Registers**

Assembly Language operations involve moving data into or out of or altering data in the CPU. Before understanding movement of RAM it is necessary to be familiar with the usage of the CPU Registers.

**Working Registers**

Registers A&C are the most versatile because they are also used for memory access. A is often used to pass hex values between subroutines. By far the most operations are available for the C register.

There are operations to use the B register with C and A. B can be used for shifts and tests as well as other math. Except for lack of memory access, B is nearly as useful as C or A.

D has the fewest operations available and is used for tests and some arithmetic. D is only accessable through the C register; there are no operations to move data between D and A. It is the most cumbersome register to use because of the limited number of instructions for it.

**Carry**

This is a flag which is set or cleared to signify the result of an operation is true (using RTNCC, RTNSC). In calculations, CARRY is set if the calculation overflows or borrows. CARRY is useful in subroutines for situations like if a string is over a certain length then use RTNSC, else use RTNCC. The associated tests are GOC (goto if carry set) and GONC (goto if no carry).

**Scratch Registers**

R0, R1,R2,R3 and R4 are used for temporary storage of information from C or A. Moving data into and out of is slower than moving between the main arithmetic registers, though much faster and easier and safer than placing the information in RAM. Relatively few system entry points use these registers (except for R4) so they are the ideal place to store intermediate results when calling a subroutine.

The A field of R4 is used whenever an interrupt occurs, which can be at any time. Don't expect to be able to place anything in the A field and have it still be there.

**Control Registers D0, D1**

D0 (dee zero) and D1 are data pointers used for memory access. D0 or D1 would be set to point at a location in RAM, then the apporprate instruction (such as C=DAT0) loads C or A with the data. Since they can be incremented and decremented quite easily they are also used as counters in loops.

When execution is passed to a Function, D0 points to the next instruction in a program, and D1 points to the top of the Math Stack. While the Function may (and usually will) alter both of these registers, they MUST be restored to the proper value when the Function terminates so that the BASIC interpreter can keep track of program flow and memory. This usually means D0 and D1 are copied to a scratch register (usually R2 or R3) or a "safe" location in RAM, then copied back to D0 and D1 later.

**Return Stack (RSTK)**

Each GOSUB, GOSUBL or GOSBVL leaves an address on the Return Stack. When the next RTN instruction is found the address is popped from the stack and execution continues

from that address. RSTK has eight levels available. When one level is popped, the rest move down and "00000" is placed at the top. If more than eight addresses are pushed on the stack then the oldest entry is lost.

Interrupts require one level on the stack, thus leaving seven for most operations. Statememts may use the full seven levels.

Because functions may be nested, the operating system may have several pending operations when it turns control over to each function. For this reason functions are usually further limited to four levels. C=RSTK can be used to save the top address to (the A field of) C temporarily before using an operation which will use more than four levels. RSTK=C is used later to return it to the stack.

**Fields Within Registers**

Each working register can contain a single 8 byte floating point value. Many operations require less than full mathematical precision, and can be done in hex or as simple numbers. Each register can be addressed by fields within the register, thus adding to it's capacity, or speeding operations.

The most memory efficient way to use registers is in 5-digit hex in the A field of a register. Even if smaller values are used (such as string length), using the full A field is the most common method used by entry points.

**P Pointer**

Fields within a register may be specified by field name or by the value of the pointer P, or by a combination of the register from nib zero through the pointer value (WP). P is also useful as a flag and can contain values of up to 15. P may be tested for any value using ?P= or ?P#. P is more versatile and useful than you could imagine 4-bits being. Most system entry points exit with P=0 and others require that P=0.

```
                Fields in Working Registers
  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0

  B    1-0   Exponent or single byte
  X    2-0   Exponent and sign
  XS    2    Exponent sign only
  A    4-0   Address. Full
  W   15-0   Whole word
  M   14-3   Mantissa only
  S    15    Sign nib only
  P          As pointed to by P
  WP   P-0   Word through to P

        WORKING REGISTERS
Carry   1 Carry Flag
  A    64
  B    64
  C    64
  D    64
  R0   64 Scratch
  R1   64 Scratch
  R2   64 Scratch
  R3   64 Scratch
  R4   64 Scratch.  A field used for interrupts

  P     4 Register Pointer
  D0   20 Program pointer
  D1   20 Math stack pointer
  PC   20 Program counter
 RSTK  20*8 Return Stack
  ST   16 Program status flags
  SB    1 Sticky Bit
```

```
      MP    1 Module Pulled Bit
      XM    1 External Module Missing Bit
     OUT   12 Keyscan use
      IN   16 Keyscan use
```

**LEX File Requirements**

Following the file header are many requirements for LEX file construction.

```
      LEX ID               2 nibs
      Lowest Token #       2 nibs
      Highest Token #      2 nibs
      Next LEX Table link  5 nibs
      Speed Table flag     1 nib  \
      Opt. Speed Table     78 nibs > 1 nib if no Speed Table
      Speed Table flag     1 nib  /
      Text Table Offset    4 nibs
      Message Table Offset 4 nibs
      Poll Handler Offset  5 nibs
      Main Table           9*(#keywords)
      TEXT Table           3*(#keywords) + 2*total #chars + 3
      Message Table
      Poll Handler Code
      Exception Code
      Next LEX Table (optional)
```

If that table didn't thwart any desire to write a LEX file then you should know that much of the above is written automatically by the Assembler. All we have to do is provide the appropriate OP Codes (operations for the machine to perform) and Assembler Pseudo-OPs (operations the Assembler interprets) and the Assembler will generate a complete LEX file for us. It's not as simple as BASIC is, but, then neither was BASIC the first time. Let's look at what is actually required by the Assembler to write a LEX file, and introduce the worlds simplest LEX file in the process. If you have the FORTH/Assembler ROM then enter the following into a Text file as listed. Use the Text file name "REVTEXT" to differentiate from the resulting LEX file (which is called "REVLEX").

```
      LEX    'REVLEX'
      ID     #5C       5C,5D,5E allocated for testing
      MSG    0         no message table
      POLL   0         we will ignore polls
REV$  EQU    #1B38E    entry point to reverse a string on the stack
EXPR  EQU    #0F23C    exit here because results are already on stack
      ENTRY  revstr
      CHAR   #F
      KEY    'REV$'    the "$" signifies that we will return a string
      TOKEN  1         values from 1-255 available for scratch
      ENDTXT           the end of the text table
      NIBHEX 4         this paramater is a string
      NIBHEX 11        will accept min 1 and max 1 parameters
* execution code
revstr GOSBVL REV$     reverse the string on the math stack
      GOVLNG EXPR      everything is still in order, exit
```

As with any Assembly Language operations, copy every important file in your 71 to Disc in case an error during testing crashes the 71. The 71 will not crash during Assembly. Now, go to FORTH environment and assemble the file.

```
      " REVTEXT" ASSEMBLE
```

When the Assembler is done...

```
BYE
```

Now, turn off the 71 then turn it back on. This adds "REVLEX" to the LEX file chain. Enter the following to test your new keyword.

```
REV$("ABCDEFG")
```

REVLEX is listed in the form used for source files by the Assembler. The first seven spaces are reserved for labels; lines without labels are filled with spaces up to the mnemonic (the OP or psudo-OP code). Modifiers (such as field specifiers) are separated from the mnemonic by at least a space, and usually begin at the 15th column. Anything following a modifier, or after an OP such as SETHEX which doesn't call for a modifier, is ignored. Remarks usually begin at column 24, though a space is all that is needed. In addition, lines beginning with "*" are also remarks. Only one operation is allowed per line. There is no such thing as an optional parameter with Assembly language.

Marking the display at 8, 15 and 24 columns with a piece of tape (or a felt tip pen if you're adventuresome) makes it easier to maintain column alignment. As a coincidence of this, most of the example program steps in this book are indented to the eight column because of the markings on the display.

Any display format wide enough to allow for the mnemonic and modifier on the same line is sufficient. Since line numbers are not used, it can be difficult to keep track of where we are in a fairly large file. Source code (the TEXT file) often exceeds 500 lines. A screen oriented Text Editor with a width of at least 40 columns is the easiest way to write LEX files. When using just the built-in LCD stop often and PLIST the file (EDLEX adds PLISTing of TEXT files).

The FORTHRAM file does not have to have free room for the Assembler. About 2K of free memory can be gained using a newly created FORTHRAM file (one without any user words defigned in it) reduced to about 1K.

```
3800 SHRINK
```

If only the 71 is used, and with limited memory, edit the file in sections using EDTEXT (the HP TEXT Editor) or TED (the Screen Oriented TEXT Editor in WorkBook71), then merge the files on Disc for Assembling.

The Assembler is sensitive to upper and lowercase. The entire instruction set must be entered in uppercase. Labels may be from one to six characters and cannot begin with equals, sharp, single quote, left parentheses or digits 0-9
= # ' ( 0123456789 . Again, the Assembler recognizes upper and lowercase as different characters. An advantage to this system is that local labels (within the file) can be entered in lowercase, and UPPERCASE labels would designate calls to the Operating System. Single quotes or the backslash (\) may be entered when quotes are needed. The \ must be assigned to a key for use. Labels may be refrenced as often as needed. However, labels must only exist one time in the file.

**The LEX File**

**LEX**

The first line is the name of the LEX file to be created. It cannot be the same name as the source file. The source file cannot begin with a remark.

**ID #**

Each file is identified by a hex byte. ID "00" and "01" are used by the mainframe. Of the total 256 possible ID's three are allocated for experimentation: 5C,5D & 5E. Distributed products should not use these ID's. These are the ones we usually use as we are developing new files or those for personal use.

If two different LEX files were in the 71, both using the same token numbers, a conflict will exist. The proper keyword will be tokenized when entered, but whichever one that came first in the file chain would be the one executed in a program. The 71 would probably go down in flames as the parameters of a function were passed to another, or worse yet, to a statement. To be safe, even for personal

use, keep a list of the usage of ID's, token numbers, and keywords.

Once a LEX file has been tested, you may want to distribute it. HP (Corvallis) will allocate an ID for the file and token numbers for each keyword to eliminate possible conflict. Write Systems Engineering Support in the HP Portable Computer Division Product Support Group in Corvallis Oregon for an application. Be aware that the allocation of ID's can take some time (several months!) because HP also checks for conflict with keywords.

**MSG**

This line refers to an error message table. In the examples given here this will always be "0" because of the expense of memory and bulk for RAM based LEX files. The usual errors of bad parameters are handled by the mainframe when entering the keyword in the BASIC program, and by the entry points when getting the data off of the stack. The most likely error to occur with strings is not enough memory to move them to a temporary buffer (usually at the beginning of free memory "AVMEMS"). A function which helps BASIC by qualifying data instead of balking whenever an error occurs can actually make BASIC programming easier. For instance, if a numerical paramater must be in the range 1-15, then zero could default to 1 and 1E27 could default to 15. Some entry points for error messages are listed in the back of this book, they may be used in lieu of creating new error messages.

**POLL**

At various times (such as when certain errors occur) the operating system polls LEX files to see if they want to intercept them. An example of this is PLIST when the file type is TEXT. This is not a mainframe valid operation, so before the operating system generates an error to the user it polls to see if anyone wants a shot at PLISTing, which EDLEX does. Another example is the VER$ poll during which everybody gets a chance at displaying their revision number. The use of these and other polls is discussed in Vol I of the IDS.

**EQU**

All refrences to locations are done using labels. The equate table is usually (though not necessarily) at the beginning of the file. It lists all of the entry points and System RAM points used in the LEX file. In REVLEX we used the entry "REV$" with GOSBVL and EXPR after GOVLNG.

**ENTRY**

This is a refrence to the label designating the location within the file where the actual code for the function lives.

**CHAR**

The charactarization nib tells the operating system what kind of keyword lives at the above refrenced label. "F" is the most common and refers to a function which returns either a string or a number and may be used either from the keyboard or program. If a function is restricted to not being usable in a program (such as EDIT) it would be "5" the bits in the nib mean the following:

| | |
|---|---|
| Legal from keyboard | bit 0 |
| Unused | bit 1 |
| Legal after THEN/ELSE | bit 2 |
| Programmable | bit 3 |

**KEY**

Each keyword is listed quoted following the psudo-OP "KEY". Only uppercase letters and numbers from 0-9. The keyword must begin with a letter and be no longer than eight characters. Functions which return a string have the dollar sign ($) as their final character; this is the only way that the Assembler can tell a function which returns a string from one which returns a number. Be sure that function names do not conflict with program variable names (don't call a function "A1").

The list of keywords is called a Text Table. Entries in this table must be

listed in alphabetical order. If a shorter keyword is contained within the beginning of a longer keyword then the longer keyword must be listed first, though alphabetically it would not be. The keyword "ABCD" would appear BEFORE the keyword "ABC" or the second keyword would not be found.

**TOKEN**
Each keyword within the file has an exclusive token number. As with LEX ID's, keywords may be 1-255. The scratch ID's have all of the tokens available. As with LEX ID numbers, tokens are issued by HP. Again, be sure to maintain a list of LEX ID's as well as tokens used in files for your own use.

**Parsing Functions**
The primary reason we are discussing functions exclusively and not statements is that the mainframe automatically qualifys the operation when the user enters the keyword. With statements the LEX file must contain code to make sure the user input the proper number and type of parameters, and insure parentheses, spaces and commas are in the correct position. This takes longer to write, is less reliable, and adds considerably to the size of the LEX file. If you wish to write statements refer to BOTH VOLs I&II of the IDS.

Often what one would think of as a statement can be written as a function. It could return a flag to signify that the operation went as it should. An example of a statement as a function is the mainframe FLAG.

In order to help the mainframe parse our functions we must give it several pieces of information:

```
Does it return a number or string
How many parameters are required
Are parameters numbers or strings
```

The keyword itself tells the parser if it returns a string (if the keyword ends with a "$") or number (no "$"). Several nibs at the beginning of the actual code (before the entry point) signify what type of parameters are required. In REVLEX the actual code began with:

```
      NIBHEX 4
      NIBHEX 11
revstr GOSBVL REV$
```

The NIBHEX pseudo-op tells the Assembler to place the following hex nibs (up to 16 maximum) in the file. The first two nibs preceeding the actual code designate the minimum and maximum number of parameters the function will accept. The minimum is, obviously, "0", maximum "F", so a function can have between zero and fifteen parameters. In the example we used NIBHEX 11 which says that the function will accept a minimum and maximum of 1 parameter. The mainframe function FLAG can use either one or two parameters.

**Types of parameters**
In addition to telling the 71 how many parameters, we can tell it what kind. The nib(s) preceeding the two describing the number of parameters tell the 71 if they are strings or numbers. "8" designates a number, "4" is a string. The code for FLAG looks something like:

```
      NIBHEX 88
      NIBHEX 12
FLAG   GOSUB  PARMCT
```

The "12" designates that FLAG will accept either one or two parameters. The "88" means that both paramaters are numbers. Let's create a function which takes a string, but can optionally use a second parameter of a number:

```
       NIBHEX 8          the second param (a number)
       NIBHEX 4          the first param (a string)
       NIBHEX 12         we'll accept either one or two params
entry
```

No problem, so far, but what do we do with the parameters when we get them, and what happens if the user supplies a quoted string or mathematical expression or sticks my function in the middle of a bunch of other functions...

**PARSING and DECOMPILING**
**Or: "The Lexical Analyzer to the rescue"**

The Lexical Analyzer is the one responsible for deciphering the users code, making sure parameters and syntax are correct, and turning it into tokens. It searches the TEXT tables until it finds the correct keyword; the keyword indexes it into the main table and from there it finds the execution code and looks up the parameters requested. If all goes well the code is accepted and it either is entered as a line of BASIC or immediately executed. If something is wrong it issues a rude comment to the user (an error message). Parentheses and hierarchy are it's task, a function never worrys about it.

**Decompiling**

After a successful parse of a keyboard operation or during BASIC program execution, the 71 gathers up the requested parameters, places them on the math stack and transfers execution to the function at the label designated as the entry point by the main table in the LEX file.

Since all functions are interpreted from the innermost parentheses out, all our function will ever see is complete numbers or strings. Functions can, at times, be supplied with pointers to arrays, but all of the functions we'll deal with here will use real values.

**Entry Conditions**

When the 71 turns things over to our function several registers reflect the conditions:

```
       D0 points to next expression
       D1 points to top (low memory end) of Math Stack
       C(S)= number of actual parameters (if variable)
       B(B)= function table entry#
```

If there are optional parameters then C(15, the sign field) contains the actual quantity. In our example from above we'll expand it to see how we will handle the situation with either one or two parameters by loading the quantity into P because it is a fast and memory efficient way to do a 1 nib comparison:

```
       NIBHEX 8
       NIBHEX 4
       NIBHEX 12
functn P=C    15        load C(S) into P
       ?P=    2         if P=2 then two params
       GOYES  param2
param1
```

In all other cases we can assume that the proper number and type of parameters specified were supplied, or it never would have gotten this far (BEEP! ERROR!, without us having even done anything).

Now, the 71 is ours, all qualified, everything in it's place, all we have to do is... Wait a minute! where is everything...

**The Math Stack**

Up in high memory, hanging upside down, is the Math Stack. Intermediate results are

stored there during function execution. The oldest entry is in higher memory and grows towards low memory as items are added and shrinks again as items are dropped.

The location of the top (low memory, first item) of the Math Stack is pointed to by MTHSTK, the bottom of the stack (high memory, end of oldest item on the stack) is pointed to by FORSTK. The location of the stack changes with the amount of user memory and the number of environments (suspended up there in limbo). In fact, FORSTK points to the FOR/NEXT Stack (which itself can grow and shrink), just above it (higher address) in memory.

**Format of data on Math Stack**

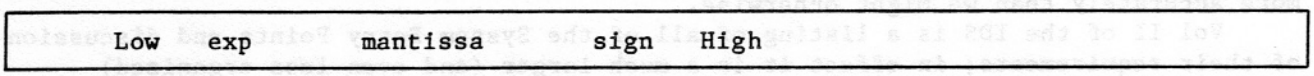
While ostensibly a "math" stack, it contains each type of intermediate result a function can handle. Notice that all non-array numbers are REAL; there is no speed or memory savings by using SHORT or INTEGER.

The first nib in the item on the Math Stack is the data type code. Usually a system routine is used to recall data from the stack; they test this nib to insure the proper data type then issue an error if it doesn't match the type needed.

| | | | |
|---|---|---|---|
| 0-9 | REAL | D | COMPLEX SHORT |
| A | INTEGER | E | COMPLEX REAL |
| B | REAL SHORT | F | STRING |
| C | REAL ARRAY | | |

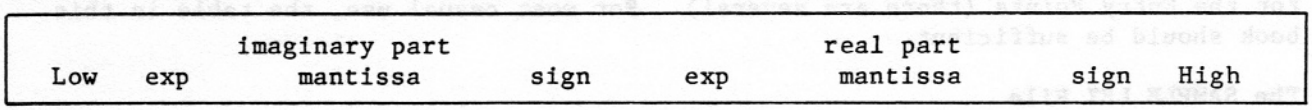
**Real Numbers**

There is no header, but it can be easily seen that the value on the stack is a real number because the first nib (low memory) is "9" or smaller. Add 16 to D1 to move it past this item on the stack. Note that most number-popping routines do not move D1 past the number.




**Complex Numbers**

A complex number is exactly like the format of two simple REAL numbers, but preceeded by "E0". The imaginary part is in lower memory. Add the length of the two numbers plus the "E0", a total of 34 nibs, to D1 to move it past this item on the stack.

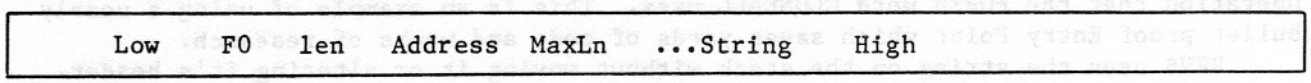



**Strings**

Strings begin with "F0", followed by five nibs representing it's length (in nibs, NOT bytes). Beyond that are nine nibs which refer to the destination of the string and the maximum length allowed, but both of these values have no meaning for functions and can be ignored, in fact, they may prove unreliable. Once it has been determined that a string is there then the important parts are the five nibs representing string length and the string itself. Add the length of the string header (16) plus the length of the string to D1 to move it past the string.

The actual string is stored backwards with the beginning in high memory, and it's end at the end of the string header. System entry REVPOP is often used to reverse a string and return it's length to A(A), so this format is nearly transparent. Be sure a string needs reversing, because it will have to again be reversed when placing the results back on the stack.

A non-existant string array will begin with "F8" and will have zero length, but will still have 16 nibs for the header




**Array Descriptors**

Arrays do not actually get placed on the Math Stack, they may not even fit, so why even try. The address is the actual location in RAM of the array.

```
                1  1     8          5
   Low          t  #  b Dim Lengths  Address High

                t = type of array
                # = number of dimensions (1 or 2)
                b = option base (0 or 1), if "8" then a STAT array
```

The first 4 nibs of DIM lengths are the second dimension. The second are for the first dimension. The pointer points to the array. To calculate actual data address of the variable, subtract the relative pointer value from the address of the relative pointer.

**System Entry Points**

The entry points can be thought of as subroutines or ready made functions, used much as in BASIC. Shifting a register left or right can be used for simple multiplication and division, but more complicated operations require several instructions to accomplish. The Entry Points greatly simplify writing LEX files. Several Entry Points are listed in the back of this book. There are several important things to look for when deciding to use a System Entry Points: What does it require for entry (D0,D1,P,HEX/DEC,data) and how does it leave the CPU when done. A major consideration is the number of stack levels it requires. Remember than Functions are restricted to four levels under most circumstances. Also, how does it handle errors, and will control never come back if some requirement isn't met. Be sure when using these subroutines that they don't take more memory to set up for than they save by using them. Many routines, such as those to pop data from the Math Stack, are helpful whether they save memory or not, because they handle the routine more accurately than we might otherwise.

Vol II of the IDS is a listing of all of the System Entry Points and discussion of their requirements; in effect it is a much larger (and even less organized) version of the Entry Point Table in the back of this book. Vol III includes the information in Vol II plus complete listings of the Operating System (and it costs four times as much as Vol II). Vol II is much better organized and easier to use. The advantage of having Vol III is to check for errors and ommisions in the write-ups for the Entry Points (there are several). For most casual use, the table in this book should be sufficient.

**The SAMPLE LEX File**

The four operations in this file demonstrate some fairly simple uses of LEX files. These functions could be written several ways, they are just an example of one way they can be done. Each keyword has been researched by Hewlett-Packard to be compatible. Without this research there could be conflict with other similar words in which perhaps neither keyword would work properly. For personal use any keywords may be used, in fact, mainframe keyword names may be given new meanings; you could, for instance, have PASSWORD not work at all to thwart people with a strange sense of humor. To be sure of compatibility, keywords which are spelled slightly differently. For example use "CEL" instead of "CELL" or "NXT" for "NEXT", or a contraction such as "CLFLS" for CLOSE FILES".

REV$ and CLFLS are mainframe routines, "SAMPLE" does little more than add keywords. CLFLS is a statement which requires no parameters and returns nothing; it is the only statement we'll demonstrate. CLFLS calls the routine to close all files. This is valuable when exiting a program which was CALLed, because the files are not automatically closed (as they are when RUNning a program). This is the same operation that the FORTH word CLOSEALL uses. This is an example of using a nearly bullet proof Entry Point which saves yards of code and weeks of research.

REV$ uses the string on the stack without moving it or altering it's header. For that reason it exits through EXPR, which assumes that D0 is in the same condition as when entered, D1 points to the first nib of the string header on the Math Stack, and the string already has a proper header on it. REV$ is often used as a subroutine within string functions.

HGL$ calls for a single string paramater then sets the high bit on each character, then it too exits through EXPR. The method used is to do a logical OR on the high nib on each character using the "C=C!B" OP code and hex "8". This is one case where it is easier to write the code in Assembly language than it is in BASIC. REV$ and HGL$ are demonstrated in BASIC in the section on BASIC programming for comparison. HGL$ as listed does not have a bug, but shows an alternative, and less efficient, method to perform the function. The "morehi" loop contains two D1=D1+1 instructions. As you know, a loop which is to be repeated several times should be written as efficiently as possible; a cleaner method would have D1 decremented by one nib before entering the loop, then a single D1=D1+2 instruction could have been used. As with BASIC, programs in Assembly Language can be written in several ways.

WTKEY$ is an alternative to KEYWAIT$ (which is in the Finance ROM and several other LEX files). Unlike KEYWAIT$ and KEY$, WTKEY$ returns the actual ASCII code for the key pressed, not the key code. For instance, ENDLINE returns CHR$(13), not "#38", and g-CMDS returns CHR$(25), not "#150". The advantage of this keyword is that all keys return a single character making it easier to respond to a variety of keystrokes with the use of POS or NUM. WTKEY$ is a modification of a similar, copyrighted, function being marked by this author. It is not being released into the public domain. However, it sure is nicer to use than KEYWAIT$. A BASIC version of KEYWAIT$ is demonstrated in the BASIC section.

**Crashes**

Assembly language programming is full of crashes, even when the code is written properly errors occur. Entry points can be entered incorrectly (a common one for this author) or conditionals could be reversed. D0 and D1 are constantly being changed so it is easy to forget to restore them. Regardless of the reason, the 71 will occasionally crash. Maybe taking with it all of your :PORT's. Always be sure that everything in the computer is backed up on Disc or Card.

The crashes will only happen after Assembly, when testing the file. Unlike common belief, the 71 can't sense that you are making a mistake in the source file, though it sometimes seems like it.

Most common mistakes will be caught by the Assembler, which will issue a warning at appropriate times. At that point you can stop Assembly (press **ON**, then **Y** at the prompt) and call the Text Editor to see what was wrong with the line mentioned. If a part of the file is known to have bugs you might complete Assembly, and test only those keywords which are completed.

A monitor, listing file, or "DISPLAY IS PRINTER" is invaluable to watch the Assembler because it's messages will dissapear from the LCD when the next line in the file is read.

**Instruction Set**

A listing of the Assembler Instruction Set is included at the back of this book. While all of the instructions are also listed in the FORTH/Assembler Manual, this listing will probably prove easier to use because it requires little page flipping. Many operations will take one nib less memory when specifying the A field, this is noted in the nib column. If two values separated by a comma are listed, then the A field version will be one nib shorter. For example "A=0 A" takes two nibs, while "A=0 X" would require three. The third page also lists the Psudo-OPs (that is, instructions for the assembler), and an example of the bare minimum required for the source file. "fs" stands for field select, "n" means a hex nib, and "fsd" means Field select fs or d (number of digits).

**Why is Everything Backwards?**

In the case of SDATA files, it's inside out and backwards. BASIC PEEK$s data from RAM directly, left to right, or low memory to high. The CPU works in the same way for Assembly language. However, when loading from RAM the lowest addressed nib is loaded to the lowest nib of the CPU register, the next nib in the next higher location, and so on. In effect, the data wraps into the register. The same is true when writing data from the CPU registers to RAM. While this method of working is often confusing in BASIC, in Assembly language it is transparent, and can often be ignored.

```
High                          <--                    Low
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 S  M  M  M  M  M  M  M  M  M  M  M  X  E  E  E
```

Only the C and A registers may be used for accessing memory. The following codes load data from or store data to RAM using D0 to point to the lowest nib. The equivalent operations are also available using D1 as the pointer.

```
C=DAT0 A        Copy 5 nibs to C(A) from location pointed to by D0
A=DAT0 W        Copy 16 nibs to A(W) from location pointed to by D0
DAT0=C A        Copy C(A) to RAM location pointed to by D0
DAT0=A W        Copy whole A register to RAM pointed to by D0
```

The instructions for loading constants work in the same manner. Both of the following will load C with "HP-71". The first load the hex code, the second uses the ASCII representation. Both are byte reversed because, again, the 71 reads and writes from low memory to high and from lower nib of the CPU towards the higher.

Use the P pointer to designate which nib in the CPU is to receive the first nib.

```
LCHEX  31372D5048
LCASC  '17-PH'
```

**The RETURN Stack**

The stack is referred to in the same way FORTH uses it. The last entry goes on the top of the stack and the oldest is on (or at least nearer) the bottom. When a RTN in encountered the newest address is popped off the top, the others move up one, and, instead of using an arbitrary number, "00000" is added at the bottom. Then the 71 goes back to the address it just popped off the stack, and reads the next command.

Each time a new entry is pushed onto the stack (by using a variation of GOSUB) the other seven entries drop one level to make room for it. If there already had been eight then the oldest one would have been pushed off the bottom of the stack never to be seen again.

When the 71 gets to our function there are already as many as three entries on the stack (pending RTNs). There is a real number waiting for us on the Math Stack, so we call a subroutine in our LEX file which itself the calls the system routine "POP1R". Now there are five entries. "POP1R" itself calls another routine which makes six. That routine does a RTN to bring us back to five, then "POP1R" RTNs to our subroutine which brings it to four levels. Our subroutine finishes it's business and returns it to the main code and our original three levels.

It is important to keep in mind the operation of the stack as subroutines are called because of the limit of seven levels, or actually four when using Functions. Whenever writing code you should be able to ask yourself at any moment "how deep is the stack?".

Remember that every time we pop an address (using RTN) a zero entry is added at the bottom. There must be an absolute relationship between GOSUBs and RTNs. If a RTN is encountered when the stack has nothing but zeros then the 71 will return to the address it finds on the stack, which is "00000". "00000" is the address of the code to reset the computer. And it will. If nothing else, that's enough reason to re-read this section.

There are times when four levels just aren't enough. To the rescue come "C=RSTK" and "RSTK=C". Respecitvely they pop the top item off of the stack to the A field of C, or push whatever is in C(A) onto the top of the stack. If a routine is going to require five levels then save one to, for instance, R3:

```
C=RSTK
R3=C
```

And, when done...

```
C=R3
RSTK=C
```

Another use of these two codes is to temporarily store a five nib value when it can be assured that the stack has room. Used carefully, this can save a byte or two, and this operation is faster than saving it elsewhere.

**GOSUB, GOSUBL, GOSBVL GOTO, GOVLNG**

Unlike BASIC, there are three commands for gosub, and two for goto. The reason is distance. GOTO and GOSUB are for distances within +2047 or -2048 nibs from the current location. GOSUBL is for distances within +32767 and -32768 nibs. GOSBVL and GOVLNG are for anywhere within the address range of the computer. The shorter versions save a few clock cycles, and a nib or two. The Assembler Instruction Set listing in the back of this book includes number of nibs for each instruction, but for files of about 1K the short form can be used for subroutines within the file, and the long form elsewhere. If a jump is greater than allowed the Assembler will issue a warning.

**Strings From the Math Stack**

As discussed earlier, strings live on the Math Stack upside down. At the end of the header is the tail end of the string. Operations which leave the string shorter can often use the string on the stack without moving it first. Since strings can often be larger than the scratch allowed for functions, the easiest place to work with them is at the opposite end of free memory from the Math Stack: at AVMEMS. If a numerical result is to be returned then the remains of the string are not important, and will get tromped the next time memory moves.

The two usual methods for getting information are "POP1S" which returns A(A) with the length of the string and D1 pointing to the tail (low memory). If the string needs to be reversed then we usually use "REVPOP", which is the same as "POP1S", except it calls "REV$" first. Either of these entry points will return A(A) with the number of NIBs; since ASCII characters are all two nibs each, this can be assumed to be an even number.

Assume that we are writing a function which trims just spaces from both ends of a string. The string will either be shorter or remain the same length when we're done, so there is no concern about checking for enough memory. First, spaces at the back end of the string could be removed, reverse the string, then trim spaces from the beginning, then reverse it again and exit. The same thing could be done by moving the middle of the string to AVMEMS then exit through BF2STK, which will move the truncated string back again. A third, and faster way to trim both leading and trailing spaces is to find the first non-space on the front end of the string (high memory), then start at the other end and find the first non-space in that direction, then shift what's left up a byte at a time, then exit through ADHEAD which will put a new string header on it.

**Numbers From the Math Stack**

Regardless of if the number is REAL, INTEGER, or SHORT, it is always on the Math Stack as a REAL number. "POP1N" checks the number on the Math Stack then returns the number in A. It exits in decimal mode. The D1 pointer is not changed, you will have to increment it past the number if more data is to be read or if a string is returned. Of course, if a number is returned then D1 is at the proper location. This routine exits in decimal mode; since various routines alter hex/dec modes often, be sure the proper math mode is in effect. To get a hex integer from a floating point number the easiest way is to use "POP1R" (which returns a 12 form number to A) followed by a call to "FLTDH" to turn it into hex in A(A). If the number doesn't round to 0-FFFFF then "FLTDH" returns with carry set. So, these three routines can give us real, floating point numbers, 12 forms, or 5 digit hex.

**Temporary Scratch**

Several parts of System RAM are reserved for function use. One caution should be made first, if a location is alocated for functions, then a routine that expects to be called from a statement may use it. For example, CHEDIT, the mainframe character editor, uses FUNCR0 and FUNCD0, because input type of routines are nearly always statements.

The System RAM chart at the back of this book lists all of the buffers which are reserved for functions. Remember, since this RAM is available for all functions, don't expect it to remain unchanged between uses. This RAM can be used as one block or split up in 5-nib sections (plus one) as needed. The transform buffer is used during execution of the TRANSFORM keyword. For that reason it is unavailable for any parse, decompile or transformation routine, but fine for regular functions.

```
FUNCR0 16 nibs divided as follows:
       F-R0-0 5 nibs
       F-R0-1 5 nibs
       F-R0-2 5 nibs
       F-R0-3 1 nib
FUNCR1 16 nibs divided as follows:
       F-R1-0 5 nibs
       F-R1-1 5 nibs
       F-R1-2 5 nibs
       F-R1-3 1 nib
TRFMBF (the transform buffer) 60 nibs
```

**Leeway**

There has been determined a minimum amount of memory for the 71 to be able to operate. It needs enough memory to be able to at least beep and say "ERR: Insufficient Memory". That amount has been determined to be 106 bytes because that is the minimum that it would take to copy a file to Disc. They worked it out as follows:

```
 25 bytes for CMMD stack to enter COPY command
 25 bytes to move the tokenized statement to statement buffer
 25 bytes to save COPY file information on the Save Stack
 31 bytes to issue COPY poll to external device
106 bytes = leeway
```

What this means to us is that we may execute any routine which uses all of free memory from AVMEMS to the top of the Math Stack while a function operates. But, when the function terminates, there must be at least 106 bytes free after the result is on the stack. All of the function returns listed will make sure that leeway is preserved. Remember that most functions return an equal amount or less than they started with as a result. The only operations which need be concerned with maintaining memory are those which either could return a large string, or which affect the size of a file in memory (which most functions don't).

**Exiting the Function**

When the function has completed it's business either a numerical or string result must be returned to the stack and D1 must point to the header. Additionally D0 needs to be restored (if altered) to the value it had when our function began. BF2STK, EXPR, FNRTN1, FNRTN2, FNRTN3, FNRTN4 are system utilities to return values to the Math Stack. It is the responsibility of the function to see that the results conform to the proper data type. Numbers with F's in them and seven and one half byte strings are sure to cause problems. Be sure to read the requirements for these routines and then GOVLNG. We're gone.

**Subroutines**

"SAMPLE" uses "getD1" and "saveD1" in most operations. Subroutines are very important to saving memory and keeping the number of errors to a minimum. Let's

demonstrate with some routines which would be used by a file with several keywords. "moveit" is a variation on several mainframe utilities, but was written because the mainframe required the CPU registers to be quite different from how they were being used in the rest of the keyword. The other subroutine called by "moveit" is "getavm" which places D0 at the beginning of free RAM, as with other movable locations, D0 should not contain AVMEMS, but instead the location it points to. The main use of "moveit" is to move a string from one location to the end of free RAM so that it may be manipulated. Usually this would add an "FF" byte to designate the end of the string, but the assumption is that this string might be added to an existing one already at the end of memory. "movblk" is an alternate entry point if D0 already points to the destination. You might use "moveit" to move a large block, then "movblk" to drop another string in the middle of it. There is no memory check, if the string pointed to is on the top of the stack then it will still be moved without worry of overwriting itself since it starts at the low memory end and there will always be at least a string header worth of space between them.

```
* move A(A) bytes to (AVMEMS)
* D1 points to the string, no memory check
AVMEMS EQU    #2F594
moveit GOSUB  getavm    place D0 at (AVMEMS)
movblk ?A=0   A         any more bytes?
       RTNYES
       C=DAT1 B         move a byte
       DAT0=C B
       D1=D1+ 2         increment pointers
       D0=D0+ 2
       A=A-1  A         decrement string length
       A=A-1  A
       GOTO   movblk
*
getavm D0=(5) AVMEMS    place D0 at the beginning of available RAM
       C=DAT0 A         read pointer
       D0=C             place it in D0
       RTN
```

The following routine is used to make sure there is sufficient free memory for creating a buffer.

```
* Assumes that D1 has been saved because MEMCKL uses D1
* if not enough memory then it restores D1, then exits
MEMCKL EQU    #012A5
MEMERR EQU    #0944D
memck  P=     0         will add leeway
       GOSBVL MEMCKL
       RTNNC            no carry= OK
       C=RSTK           pop the pending return, we're not going back
       GOSUB  getD1     restore D1
       GOVLNG MEMERR    BEEP! Err:Insufficient Memory
```

Assembler Bugs

VER$ "FTH:1A" of the Forth/Assembler ROM has at least two known bugs, the first of which can be fatal. When Assembling a file with more than nine keywords be sure that you first use the FORTH word "DECIMAL" to set the FORTH environment to decimal mode. The code for "B=B+B" does not always work correctly. If you have trouble with "B=B+B" contact HP for the current FORTH language fix.

```
        LEX     'SAMPLE'
        ID      #5C         a scratch ID.  DON'T USE FOR DISTRIBUTION
        MSG     0           no message table
        POLL    0           ignore polls
CLOSEA  EQU     #120E4      closes ASSIGN#'s
NXTSTM  EQU     #08A48
OUTELA  EQU     #05303
POP1S   EQU     #0BD38      pop a string from math stack
EXPR    EQU     #0F23C      return from expression, doesn't change stack
FUNCRO  EQU     #2F89B      scratch buffer for functions
BF2STK  EQU     #18663      move buffer to math stack
SLEEP   EQU     #006C2      power down, wait for a key
POPBUF  EQU     #010EE      pop last key from key buffer
KEYCOD  EQU     #1FD22      look-up table for keycode to ASCII
FNRTN1  EQU     #0F216      return from a function
CKSREQ  EQU     #00721      used by WTKEY$ & KEYWAIT$ to ckeck KEYBOARD IS
REV$    EQU     #1B38E      reverse string on stack
CSLC5   EQU     #1B435      shift C Left 5 nibs circular
CSRC5   EQU     #1B41B      shift C Right 5 nibs circular
        ENTRY   clfls
        CHAR    #D
        ENTRY   hgl
        CHAR    #F
        ENTRY   rev
        CHAR    #F
        ENTRY   wtkey
        CHAR    #F
        KEY     'CLFLS'     closes all open files ASSIGN#n TO * on everything
        TOKEN   2           start with token #2
        KEY     'HGL$'      sets high bit on all chars in a string
        TOKEN   3
        KEY     'REV$'      reverse chars in a string (mainframe function)
        TOKEN   4
        KEY     'WTKEY$'    wait-for-a-key, return ASCII, not the keycode
        TOKEN   5
        ENDTXT              ****  end of the text table  ****
        REL(5)  decom       point to decompile routine for statement
        REL(5)  par         point to parse routine
clfls   P=      0           ****  CLFLS - does ASSIGN#n TO * on everything ****
        GOSUB   saveD1      copy D0,D1 to R2 (sub is at end of this file)
        GOSBVL  CLOSEA      call mainframe
        GOSUB   getD1       restore D0,D1 from R2
        GOVLNG  NXTSTM      bye
decom   GOVLNG  OUTELA      decompile routine for clfls. Pointed to by REL(5) decom
par     RTNCC               parse routine for clfls. Pointed to by REL(5) par
        NIBHEX  4           param is a string
        NIBHEX  11          minimum and maximum of one paramater
hgl     SETHEX              *** HGL$($)  Set high bit on all chars in a string ****
        GOSUB   saveD1      save D0 & D1 pointers to R2
        GOSBVL  POP1S       get string stats D1 points at start of string, A=str len
        P=      15          pointer to S field
        LCHEX   8           load constant "8" into S field of B
        B=C     S
morehi  ?A=0    A           any chars left in string
        GOYES   hibye       no, then exit
```

```
         D1=D1+ 1      move pointer to hi nib of current character
         C=DAT1 S      load the nib to S field of C
         C=C!B  S      logical OR of B into C
         DAT1=C S      restore the nib to string
         A=A-1  A      decrement string length counter
         A=A-1  A
         D1=D1+ 1      move to next nib
         GOTO   morehi go back for next character
hibye    GOSUB  getD1  restore DO,D1 to beginning of string header, then exit
         GOVLNG EXPR   bye
         NIBHEX 00     no params so no nib to describe param type
wtkey    SETHEX        **** WTKEY$  wait for a key, return ASCII ****
         GOSUB  saveD1 save DO,D1 to R2
nxtkey   GOSBVL SLEEP  nod off and wait for a keystroke
         GOC    cksreq carry set if KEYBOARD IS pressed the key
         GOSBVL POPBUF keycode is in B(A), (B=B+B bug in assembler)
         A=B    A      move key to A(B), save a nib by using A field
         A=A+A  A
         D1=(5) KEYCOD look-up ASCII from keycode table (index by key#)
         CD1EX
         C=C+A  A      add this key to keycode to offset into list
         CD1EX         key pos back to D1
         C=DAT1 B      read ASCII to C(B)
         D1=(5) FUNCRO point D1 to scratch buffer
         DAT1=C B      put key in buffer
exit     D1=D1+ 2      enter here to move anything at FUNCRO then exit
         P=     0      just to make sure P=0
         LCHEX  FF     place "FF" after key in buffer for end of string char
         DAT1=C B
         GOSUB  getD1  restore pointers from R2
         LC(5)  FUNCRO needed by BF2STK
         ST=0   0
         GOVLNG BF2STK move this buffer to math stack, exit
cksreq   GOSBVL CKSREQ used by WTKEY$ to see if KEYBOARD IS pressed a key
         GOTO   nxtkey
         NIBHEX 4      param is a string
         NIBHEX 11     min and max of one param
rev      GOSBVL REV$   **** REV$ - same thing used everywhere else ****
         GOVLNG EXPR
saveD1   RO=C          copy DO,D1 to R2
         CD1EX         uses RO for scratch so it doesn't trash C
         D1=C
         GOSBVL CSLC5
         CDOEX
         DO=C
         R2=C
         C=RO
         RTN
getD1    RO=C          restore DO,D1 from R2, uses RO for scratch
         C=R2
         DO=C
         GOSBVL CSRC5
         D1=C
         C=RO
         RTN
```



## Communicating with RS-232C

There are times when you may want to connect your 71 to something other than the standard HP offering of accessories which plug directly into the HP-IL Interface. Most often these devices will be modems (telephone hookups), printers, terminals (used as a keyboard and display for the 71), BSR controllers (have your 71 turn on lights or water the garden), other computers (to share information) or lab equipment (so it can check if the plants need watering). RS-232C is called "serial" because data is transferred using a single set of wires, one bit at a time.

HP-IB, also known as IEEE-488, is often used for controller applications, and is usually expensive to use. For instance, HP-IB cables can cost ten times as much as HP-IL cables. HP-IB can be thought of as the big brother of HP-IL. We also have Parallel (also known as "Centronics" after the printer company who standardized on it). The parallel interface is most often used for printers. To communicate with non-HP-IL devices we most often use an HP-IL/RS-232C Interface. Much of the world (with the exception of parallel printers) accepts RS-232C so we will discuss a few applications using this device.

The Electronic Industry Association (EIA) has designated RS-232 as a standard. The standard establishes protocol, connectors and other specifications to insure that products from different manufacturers will be able to work together.

Most new computers either come with it or offer it as an option. Be sure that whatever you want to connect to your HP-71 has an "RS-232C port" or "serial port".

### The Black Box

In order to "speak RS-232" you will need an HP-82164A HP-IL/RS-232C Interface (in addition to the 82401A HP-IL Interface). Unfortunately it cannot run on batteries or a car adapter which makes using it on the run difficult. The first things you will notice when you plug it into the loop are that it doesn't do anything and the manual was written for electronics engineers and people with an intimate relationship with the lower level commands of HP-IL. The next thing you will see is that there isn't a clue about using it with the HP-71.

The HP-82164A RS-232 Interface does not have the innate ease of set-up of most HP-IL devices. This is because the standard is somewhat flexible and many companies have taken it upon themselves to create the ultimate RS-232 standard, usually slightly different than the other guy. The HP-IL/RS-232 Interface is designed to adapt to most of these quirks. Before deciding to spend the rest of your life communicating with pencil and paper, remember that you only have to go through this one time for each RS-232 device, so read on.

### The RS-232 Cable

Once you've determined what you want to hook up to your 71 you must next physically connect them, so we will first discuss acquiring a cable.

"RS" stands for "Revised Standard" and, as such, it is (usually) fairly easy to connect things together using it, though even under the best of circumstances it will be much more work than connecting, for instance, a 9114A Disc drive. Many companies (HP included) have used the "standard" 25 pin connector (called DB-25) for other purposes, or other connecters for RS-232. For instance, IBM has used the 25 pin connector for parallel, and HP has used a 9-pin connector for RS-232 on the larger HP-110 and HP-115 Portable Computers, as has Apple for the Macintosh. This 9-pin connector has even been used for power cords (imagine the zap!). For these reasons it is often necessary to have custom cables made. The alternative is to make your own cable.

Dealers will sell you cables for $50 or so plus, perhaps, $20 consultation fee during which they will glance at your RS-232 owners manual, tell you that HP makes a good product, and, pointing to the door, nod and say "Yup, it'll work... course, ya might have ta switch two and three". You leave not knowing which two and three things he's talking about. It might take a few go-arounds with the dealer to make sure it works properly.

You can save time and money if you go to Radio Shack or other local electronics hobbiest store and explain your problems and ask for the wires and connectors to make

your own cable. Make sure you get the appropriate male and female ends. This will cost about $15 and the ubiquitous electronics jock employee there will probably offer free consultation. I've made several cables using a large Weller soldering gun which is more suited for car radiators. They will sell you a little soldering pen more suited for the job, and less likely to vaporize wires.

When you look at the front end of a male connector (the kind that is on the HP-IL/RS-232 Interface itself) you will see that the pins are numbered from 1-13, left to right, on the first row and from 14-25 on the second row. A female connector mates directly so that the pins are a mirror image of the male connector with pin 1 in the upper right corner. There is never any reason to use pins other than 2-8 and 20. The simplest circuit will use only 2 and 3 (that's what the salesman was telling you to switch).

Look inside of most devices and you will see that the other pins don't do anything, often the pins have been left off of a male connector. The chart on the back of the interface is usually all that you will have to refer to when making a cable. Pin 2 generally transmits data while pin 3 receives it, and this is the problem: many companies reverse the use of 2 and 3. There are other variations. For instance Okidata printers use pin 11 for what HP uses pin 8; hook it up 8 to 8 and the printer will loose data when the computer gets ahead of it. Your printer or modem manual will probably have some criptic chart to show if all is normal.

If it appears that everything is going to be pretty much standard, create the cable with pins 2-8 and 20 connected straight through. If it doesn't work, "switch two and three". If it still doesn't work or works only marginally (loses data or occasionally garbles it) then re-read the sections below and the manual for the device.

A switch inside of the interface comes set for DTE (Data Terminal Equipment). If you are going to use a modem, or hooking to another computer, leave it that way. For some devices (like an Okidata printer) you have to open the interface and turn the plug around. This is illustrated on pages 21-22 of the interface manual.

If you are going to use two or more types of devices, a separate cable for each will save you constantly opening up the interface to change that plug, or cable to change a wire.

Switch boxes are available to allow connecting several devices through the same interface. The price range is $50 to $160 for a box with little more in it than some wires, plugs and switches. However, (in the case of the $50 unit) this box is a time and equipment saver when working with several devices and only one RS-232 Interface. Of course, another option is to have a separate RS-232 interface for each device.

**Setting up the Interface**

Once hooked up, it's time to configure the interface to the 71 and device. Since RS-232 is versatile, it can interpret data in many ways. The interface does not save it's configuration when you shut it off. You need a set-up program (or key assignment) to run each time you turn it on. Since the whole loop can slow down with some set-ups, it may also be helpful to have a program to clear the Interface.

The 14 control registers and 12 character registers in the RS-232 Interface give complete control over what is sent through the Interface. For example you can set a printer to use 7 bit data instead of the usual 8 bits so that characters above ASCII 127 will be printed as the equivalent minus 128. This is useful if you PLIST a program which has characters from the alternate character set so that the characters will print as normal characters instead of the weird graphics cartoon characters that many printers provide for characters above 127. Lets use an example of a program for a printer which runs at 2400 Baud. It is listed as a program, but if the extra spaces were removed it could be assigned to a key.

```
 5 ! A printer set-up program
10 REMOTE :RS232
20 OUTPUT :RS232 ;"SE2;SE4;SE5;SE7;LI0;LI5;SL2;SL4;SW1;SBA;P0;C0"
30 LOCAL @ PRINTER IS :RS232
```

The easiest way to change registers is in REMOTE mode during which anything received by the RS-232 is assumed to be a command. We entered REMOTE mode on line 10; be sure to designate which device is to be used. Line 20 sends various commands as found on pages 37-40 of the RS-232 Manual. Look up the nemonic for the command, then enter them in the string, each separated by a semicolon, as many commands as are needed. Line 30 restores everything to LOCAL mode then assigns the printer. In the example we used "PO" to set even parity and "SBA" to set the interface to 2400 baud.

So, we've used the REMOTE command to tell the RS-232 how to communicate with the printer, now the RS-232 becomes pretty much invisible and sends whatever we tell it to to the printer; at this point we can tell the printer how to work with it's own brand of commands.

Printers use special commands to change character style, size and such. This is usually done with characters below CHR$(32) or with a series of characters which begins with CHR$(27). CHR$(27) is called the "escape" character, and signifies that the next character to follow is to be interpreted as a command to do something, and neither the CHR$(27) or the command following it will be printed. In fact, CHR$(27) is almost never displayed or printed; if you want to see what it looks like then enter DISP CHR$(27+128). These control codes are sent to the printer in a PRINT statement and may be within a string containing data to print. Normally, when anything is printed, a line is advanced each time. End the PRINT statement with a semi-colon to keep from having this line printed.

Sending an escape code only:

```
PRINT CHR$(27)&"1";
```

In this example nothing will be printed, however the printer will receive the escape sequence (assuming that escape "1" is a valid code). The second example will send the escape code but also print data.

```
PRINT CHR$(27)&"1>>> HP-71 <<<"
 >>> HP-71 <<<
```

The interface's default (how it works when first turned on) configuration will allow you to connect a HP Terminal with only minimal configuration (see the Zenith program below). The major concerns when writing the configuration are Baud rate and handshake.

Baud rate refers to the speed at which data will be transferred. Basically it represents the number of bits per second (plus some overhead) the device can send or receive data. Terminals will usually operate at 2400 or 9600, modems at 300 or 1200. An elusive problem can often be worked around by slowing the Baud rate; if the computer you are using starts throwing garbage characters at your 71 at 9600 Baud ("SBE") then try 4800 ("SBC"), or, as a last resort (because it's kind of slow for everyday life) try 2400 or less. The maximum selectable Baud rate is 19200 and, on a loop with several devices, it is unlikely that you would see any effective speed gain over 9600. If you just can't get it to work consistently then go back and check the cable.

Protocol refers to how the device expects data to look and the messages to be used by the interface to tell it when it is ready for data. Check the conditions your device requires and, again, look up the appropriate codes on pages 37-40 to send it in REMOTE mode.

**External Keyboards**

The 71 can use an external keyboard to aid in entering long programs. One method of using a keyboard is to have the 71 designated as a device (CONTROL OFF) and have the controller place the 71 in REMOTE mode (the 71 can't make itself REMOTE). The controller then sends complete lines of data (such a program lines) which the 71 will accept without displaying them unless there is a syntax error. This can be done using an HP-75 as the controller. In fact, if the 75 designates the 71 as the printer and PLISTS a program then the 71 will try to enter each line as a program

line. This is all well and good if you have a 75, but a more elegant solution is actually easier.

A LEX file called "KEYBOARD IS" is available in the FORTH/Assembler ROM and from the Users Library as LEX file #03194-71-5. It adds the keyword KEYBOARD IS which operates like PRINTER IS and and ESCAPE which is used to trap the escape sequences sent out by the keyboard and turn them into keystrokes the 71 can understand. The ESCAPE keyword creates a buffer (like a file, but it doesn't show up with CAT) of the characters which you tell it to look for. The 71 does not go to low power state when waiting for keystrokes when an accessory keyboard is active, so it is usually best to have a wall plug handy. The interface is set-up to tell the 71 when it has data (to "interrupt"), unlike other usages when the 71 tells the interface when it has information or perhaps polls it to see if it has data to send (which is why the external keyboard doesn't slow the 71 down. With this method of using the interface the device can also turn the 71 on by "pressing" the **ON** key if flag -21 is set. This is the flag which disables the 71 from automatically powering down devices when it turns off, so be sure to clear it when not using a keyboard.

The two examples below use this LEX file. The first uses a NEC PC-8201A as the keyboard. The second uses an innexpensive Zenith terminal.

Both examples use the RS-232 at it's default 9600 Baud rate. This is done for simplicity, but also because slow transmission speeds cause a delay in response to keystrokes. The 71 has a built-in buffer for keys which have been pressed, but the 71 has been too busy to notice them. What often happens if you type very fast is that the keys will get bogged down in the RS-232 and may still be processed after a DISP statement (which wouldn't happen normally because the key buffer is emptied by the 71 during a DISP).

**A computer as a Keyboard**

The advantages of using a NEC PC-8201A, Radio Shack Model 100 or Olivetti M-10 as a terminal are many. They have reasonable Text Editors, built-in communications programs, and give us an exposure to Microsoft Basic. These machines have been superseeded by more advanced (and expensive) models with larger displays and better software, but we'll concentrate here on the less expensive models. Of the three machines, the NEC might be preffered for our uses; it comes with more memory, better keyboard, is usually cheaper, and the Basic goes faster. Since we are going to write a communications program in Basic to map the keys on the big computer to our purposes, fast Basic is important. The program is used so that the cursor keys and such can be made to work properly on the 71 instead of merely entering the code for the key which TELCOM (or a real terminal) would do.

The program below was written for use with the NEC PC-8201A and may require modification to run on the Radio Shack or Olivetti. Since it is written in Microsoft Basic, it should also be possible to modify the program to run on other machines which use that dialect, presuming you know how the cursor keys and RS-232 port are handled. The program reads keystrokes individually and maps them to escape sequences if they are below ASCII 32. The 71 traps to the escape sequences and turns them back into 71 keystrokes.

Our sample program adds some new editing features to the 71. The TAB key does 6 cursor rights to aid in moving across the screen. The ESC key works as ATTN or ON. CTL UP ARROW and CTL DN ARROW scroll a display device one line up or down; helpful for looking at a line when it has scrolled off the display, though it does not alter what is on the LCD so be aware that the monitor and LCD may not display the same thing. INS toggles insert mode on/off as with [I/R]. BS is the same as [BACK]. DEL works like [-CHAR] the vertical bar key (¦) works the same as [CMDS]. Back slash (\) adds a clear key which erases the current line from the display. The function keys are not displayed, but they still operate and may be used and reassigned as desired; for instance, f-5 enters "Run". The cut and paste buffer is not altered, if you press PAST then the entire contents of that buffer will be sent to the 71. The other keyboard characters work as on the 71, however, remember, any key which does not have a counterpart on the 71 will be ignored. The [LC] key on the 71 inverts the way the 71 interprets case, so that if you are set to lower case on the 71 then uppercase characters from the NEC will be turned into lowercase, and vice versa.

The program runs over 40 words per minute. In fact, there are some delays built in because of the problem of repeating keys all being interpreted. An alternative way to write the program would be to allow only one key pressed at a time, but this would slow it needlessly. Without the delays you could easily have over 50 UP ARROW keys waiting in the RS-232 buffer when you inadvertently left your finger on the key. With the delay, the most you could get is about 5 keys. Line 800 of NECKBD sets the NEC for 9600 baud.

```
  5 ! KBD program for the HP-71 using NEC PC-8201A:
 10 RESTORE IO @ CONTROL ON @ REMOTE
 20 OUTPUT :RS232;"SE0;SE3;" @ LOCAL @ KEYBOARD IS :RS232
 30 RESET ESCAPE @ ESCAPE "!",43 @ ESCAPE "/",47
 40 ESCAPE "0",48 @ ESCAPE "2",50
 50 ESCAPE "3",51 @ ESCAPE "g",103
 60 ESCAPE "i",105 @ ESCAPE CHR$(150),150
 70 ESCAPE CHR$(159),159 @ ESCAPE CHR$(160),160
 80 ESCAPE "&",38 @ ESCAPE CHR$(162),162
```

```
  5 REM "NECKBD" HP-71 KEYBOARD for the NEC PC-8201A
 10 E$=CHR$(27):PRINT E$+"U":CLS:PRINT
100 PRINT" [ESC] = ATTN          [¦] = CMMD
200 PRINT" [INS] = I/R           [\] = CLR
400 PRINT:PRINT"[STOP] , [SHIFT]+[f.5] to exit";
800 OPEN "COM:8N81XN" FOR OUTPUT AS #1
6010 K$=INKEY$:IF K$="" THEN 6010
6020 K=ASC(K$):IF K<32  THEN 7000
6025 IF K=127 THEN K$=E$+"P" 'S-del
6026 IF K= 92 THEN K$=E$+CHR$(159)+E$+"J" ' \ (backslash)
6027 IF K=124 THEN K$=E$+CHR$(150) '¦ cmd
6030 PRINT#1,K$; : GOTO 6010
7000 IF K= 13 THEN K$=E$+"&" 'rtn
7010 IF K= 29 THEN K$=E$+"/" 'left
7020 IF K= 30 THEN K$=E$+"2":GOTO7500 'up
7030 IF K= 31 THEN K$=E$+"3":GOTO 7500 'down
7040 IF K= 28 THEN K$=E$+"0" 'right
7050 IF K=  9 THEN K$=E$+"0"+E$+"0"+E$+"0"+E$+"0"+E$+"0"+E$+"0" 'tab
7060 IF K= 18 THEN K$=E$+"i" 'ins
7070 IF K=  1 THEN K$=E$+CHR$(159) 'S-left
7080 IF K=  6 THEN K$=E$+CHR$(160) 'S,ctl-right
7090 IF K= 26 THEN K$=E$+"T" 'ctl-down
7100 IF K= 20 THEN K$=E$+CHR$(162) 'S-up
7110 IF K= 23 THEN K$=E$+"S" 'ctl-up
7120 IF K=  2 THEN K$=E$+CHR$(163) 'S-down
7130 IF K=  8 THEN K$=E$+"g" 'back
7140 IF K= 27 THEN K$=E$+"!" 'esc
7200 GOTO 6030
7500 PRINT#1,K$; : FOR K=1 TO 450:NEXT: GOTO 6010
```

**Using a Terminal as a Keyboard**

The KEYBOARD LEX file is easier to use with a terminal than another computer, though innexpensive terminals have few special keys to use for the 71's special keystrokes. We have mapped the QUIT key (escape+CHR$(124)) to [ATTN] and HELP (escape+chr$(126)) to [CMDS]. If your terminal has other dedicated keys they may be used for [I/R], [-CHAR] and others.

```
 10 RESTORE IO @ CONTROL ON @ REMOTE
 20 OUTPUT :RS232 ; "SE0;SE3;" @ LOCAL @ KEYBOARD IS :RS232
 30 RESET ESCAPE @ ESCAPE CHR$(124),43 @ ESCAPE CHR$(126),150
 40 ESCAPE "A",50 @ ESCAPE "B",51
 50 ESCAPE "C",48 @ ESCAPE "D",47
```

```
60 ESCAPE "S",162 @ ESCAPE "T",163
70 ESCAPE "U",160 @ ESCAPE "V",159
```

**Exchanging Files**

Since we're using an external keyboard at times, why not exchange Text files with the other computer? The following program, in conjunction with the TELCOM program in the NEC and the "NECTALK" program below allow fairly easy exchange. Use the built-in program TELCOM to receive files and "NECTALK" to send files. When transferring files to the 71 the program will display the length of each line and will end automatically. When transferring files to the NEC in TELCOM mode watch the file as it is displayed; the 71 will beep when it is done, and a second or two later the last line of that file will be displayed on the NEC. At that point press [SHIFT] [f.5] on the NEC to stop TELCOM. The main incompatibilities between the 71 and the NEC are that lines do not necessarily end with a carriage return, and when the NEC sends a line it does not add a line feed character after it, which the 71 (and most of the world) expects, so we use these small BASIC programs to adapt.

Since the NEC and Radio Shack do not add a line feed (CHR$(10)) at the end of each line, and do not tell the "host" (the 71) when the file is done, we usually use a program written in BASIC. These machines can be made to add a line feed while in the TELCOM program by a simple POKE. The program on the 71 will have to be stopped manually when the NEC has completed it's transfer, or enter "!END" then press **RTN** then press **CTL J** (for line feed) while still in TELCOM. To enable the automatic line feed after carriage return enter BASIC on the NEC RS or Olivetti machines then one of the following:

```
NEC PC-8201A:  POKE 62469 , 1          Olivetti M-10:  POKE 63069,1
RS Model 100:  POKE 63066 , 1
```

Be sure to restore the location to zero for normal use.

For conformity with HP-71 file structure, each line, including the last line in the file, must end with a carriage return. Otherwise the resulting lines could be longer than the 71 allows. Be also aware that the TAB will be transferred to the 71 as a tab character [CHR$(9)], not as a series of spaces.

The options are to send a file, receive a file replacing any existing data in that file or receive a file appending data to the end of the file. Files may be in RAM or on Disc. If you specify a Disc file then the file size is not limited to available RAM in the 71. When transferring files to the 71 run NECTALK first to find the length of the Text file to create on the 71. If you have specified a file size and it is to be in RAM then the size will automatically expanded as needed; Disc based files must have their size specified accurately (or too large).

The INCAT subprogram and KEYWAIT$ function used in this program are discussed in the programming section in this book. The program assumes that the computer with which you are exchanging data is also the KEYBOARD. Delete the KEYBOARD IS on line 80 and CALL KBD on line 1000 if this is not the case.

```
 5 ! HP-71 file transfer program
10 CALL NEC @ SUB NEC @ DIM Q$[256]
20 INPUT "text file:";F$ @ IF NOT LEN(F$) THEN CAT ALL @ GOTO 20
30 CALL INCAT(F$,X) @ IF X AND X#1 THEN 20
40 DISP "Receive/Send" @ F=POS("RS",UPRC$(KEYWAIT$)) @ IF NOT F THEN 20
50 IF NOT X AND S=2 THEN 20
60 IF NOT X OR F=2 THEN 80
70 DISP "Append/New" @ X=POS("AN",UPRC$(KEYWAIT$)) @ IF NOT X THEN 20
80 KEYBOARD IS * @ CLEAR :RS232 @ IF X THEN 110
90 INPUT "size:";L @ CREATE TEXT F$,L
100 DISP "start transfer"
110 DISP "working" @ ASSIGN #1 TO F$ @ ON ERROR GOTO 1000
    @ IF 1=F THEN 130
120 READ #1;Q$ @ OUTPUT :RS232 ;Q$ @ GOTO 120
130 IF X=1 THEN RESTORE #1,9999
```

```
 140 ENTER :RS232 ;Q$ @ IF Q$="!END" THEN 1000
 150 DISP LEN(Q$) @ PRINT #1;Q$ @ GOTO 140
1000 ASSIGN #1 TO * @ CALL KBD @ BEEP @ DISP "done" @ END
9600 SUB INCAT(F$,T) @ ON ERROR GOTO 9640 @ DISP CHR$(27)&">";
     @ CAT F$ @ T$=DISP$[1,32]
9610 IF NUM(T$[12])=32 THEN T$=T$[3]
9620 T=POS("TESDDABILEKEBAFO",T$[12,13]) @ IF NOT MOD(T,2) THEN T=20
     ELSE T=(T+1) DIV 2
9630 END
9640 T=21 @ IF ERRN=57 OR ERRN=255022 THEN T=0

   5 REM "NECTALK" program for the NEC PC-8201A
  10 MAXFILES=2:PRINT CHR$(27)+"U":CLS:FILES:INPUT"file";F$
  40 OPEN F$FOR INPUT AS #1:X=0:M=0:PRINT"checking len
  60 INPUT#1,Q$:X=X+LEN(Q$):M=M+1:IF NOT EOF(1)THEN 60
  70 CLS:CLOSE:PRINT"Len:";X;",Lines:";M:PRINT"min file size:";(M*3)+X
  80 INPUT"press [RTN]";Q$
 100 OPEN F$ FOR INPUT AS #1
 110 OPEN "COM:"FOR OUTPUT AS #2
 150 Q$=INPUT$(1,1):PRINT#2,Q$;:IF NOT EOF(1)THEN 150
 200 PRINT#2,"!END":CLOSE:MAXFILES=1
```

**Display Devices**

The HP 82163A (32 columns, 16 rows) and HP 92198A (80 columns, 24 rows) are the preferable Video Interfaces. HP terminals are preferred over other terminals because of compatibility. If other computers or other brands of terminal are used be aware that HP uses unique escape sequences which make formatted display and word wrap at the end of the line an iffy situation. Programs which offer formatted display (such as the spreadsheet in WorkBook71) may not display properly due to different escape code interpretation. If possible, check the Terminal manual before purchase to insure that it can interpret HP escape sequences or can be programmed to do so. Some of the most problematic escape sequences are listed below.

```
      HP                                Common usage
Cursor to Address: % col row            Y row col
Insert Cursor:          Q
Replace Cursor:         R
Cursor on:              >
Cursor off:             <
```

If you are using a terminal as both keyboard and display and find that yyoouu ggeett ttwwoo of each character on the display then the terminal is echoing the transmitted data, set FULL DUPLEX on the terminal to eliminate the double vision.

If a display device ignores insert and delete modes, the 71 can be fooled into thinking that an HP display is being used. This will not enable the insert cursor (presuming the terminal has one). As a last resort the 71 can be made to only display a line after you press **ENDLINE.** This POKE will have to be repeated each time the HP-71 is turned on or devices are re-configured.

```
POKE"2F7B1","D"  ! Mimic a Hewlett-Packard display device
POKE"2F7B1","B"  ! Mimic a printer used as a display
```

## Decimal / Hex / Binary / ASCII Conversions

| Dec | Hex | Binary | Asc | Dec | Hex | Binary | Asc |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 00000000 | | 128 | 80 | 10000000 | |
| 1 | 1 | 00000001 | | 129 | 81 | 10000001 | |
| 2 | 2 | 00000010 | | 130 | 82 | 10000010 | |
| 3 | 3 | 00000011 | | 131 | 83 | 10000011 | |
| 4 | 4 | 00000100 | | 132 | 84 | 10000100 | |
| 5 | 5 | 00000101 | | 133 | 85 | 10000101 | |
| 6 | 6 | 00000110 | | 134 | 86 | 10000110 | |
| 7 | 7 | 00000111 | | 135 | 87 | 10000111 | |
| 8 | 8 | 00001000 | | 136 | 88 | 10001000 | |
| 9 | 9 | 00001001 | | 137 | 89 | 10001001 | |
| 10 | A | 00001010 | | 138 | 8A | 10001010 | |
| 11 | B | 00001011 | | 139 | 8B | 10001011 | |
| 12 | C | 00001100 | | 140 | 8C | 10001100 | |
| 13 | D | 00001101 | | 141 | 8D | 10001101 | |
| 14 | E | 00001110 | | 142 | 8E | 10001110 | |
| 15 | F | 00001111 | | 143 | 8F | 10001111 | |
| 16 | 10 | 00010000 | | 144 | 90 | 10010000 | |
| 17 | 11 | 00010001 | | 145 | 91 | 10010001 | |
| 18 | 12 | 00010010 | | 146 | 92 | 10010010 | |
| 19 | 13 | 00010011 | | 147 | 93 | 10010011 | |
| 20 | 14 | 00010100 | | 148 | 94 | 10010100 | |
| 21 | 15 | 00010101 | | 149 | 95 | 10010101 | |
| 22 | 16 | 00010110 | | 150 | 96 | 10010110 | |
| 23 | 17 | 00010111 | | 151 | 97 | 10010111 | |
| 24 | 18 | 00011000 | | 152 | 98 | 10011000 | |
| 25 | 19 | 00011001 | | 153 | 99 | 10011001 | |
| 26 | 1A | 00011010 | | 154 | 9A | 10011010 | |
| 27 | 1B | 00011011 | | 155 | 9B | 10011011 | |
| 28 | 1C | 00011100 | | 156 | 9C | 10011100 | |
| 29 | 1D | 00011101 | | 157 | 9D | 10011101 | |
| 30 | 1E | 00011110 | | 158 | 9E | 10011110 | |
| 31 | 1F | 00011111 | | 159 | 9F | 10011111 | |
| 32 | 20 | 00100000 | | 160 | A0 | 10100000 | |
| 33 | 21 | 00100001 | ! | 161 | A1 | 10100001 | ! |
| 34 | 22 | 00100010 | " | 162 | A2 | 10100010 | " |
| 35 | 23 | 00100011 | # | 163 | A3 | 10100011 | # |
| 36 | 24 | 00100100 | $ | 164 | A4 | 10100100 | $ |
| 37 | 25 | 00100101 | % | 165 | A5 | 10100101 | % |
| 38 | 26 | 00100110 | & | 166 | A6 | 10100110 | & |
| 39 | 27 | 00100111 | ' | 167 | A7 | 10100111 | ' |
| 40 | 28 | 00101000 | ( | 168 | A8 | 10101000 | ( |
| 41 | 29 | 00101001 | ) | 169 | A9 | 10101001 | ) |
| 42 | 2A | 00101010 | * | 170 | AA | 10101010 | * |
| 43 | 2B | 00101011 | + | 171 | AB | 10101011 | + |
| 44 | 2C | 00101100 | , | 172 | AC | 10101100 | , |
| 45 | 2D | 00101101 | - | 173 | AD | 10101101 | - |
| 46 | 2E | 00101110 | . | 174 | AE | 10101110 | . |
| 47 | 2F | 00101111 | / | 175 | AF | 10101111 | / |
| 48 | 30 | 00110000 | 0 | 176 | B0 | 10110000 | 0 |
| 49 | 31 | 00110001 | 1 | 177 | B1 | 10110001 | 1 |
| 50 | 32 | 00110010 | 2 | 178 | B2 | 10110010 | 2 |

## Decimal / Hex / Binary / ASCII Conversions

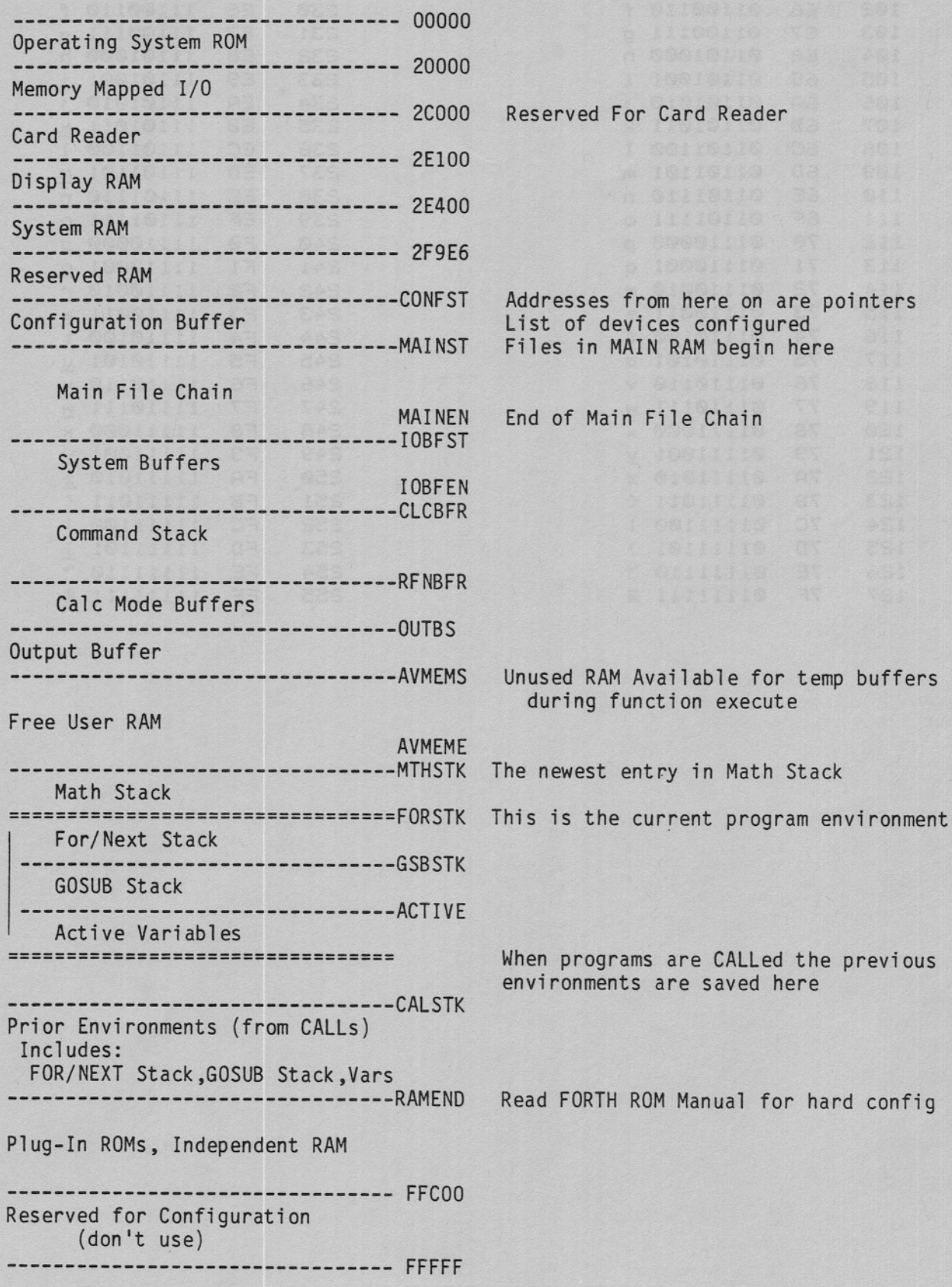
| Dec | Hex | Binary | Asc | Dec | Hex | Binary | Asc |
|---|---|---|---|---|---|---|---|
| 51 | 33 | 00110011 | 3 | 179 | B3 | 10110011 | 3 |
| 52 | 34 | 00110100 | 4 | 180 | B4 | 10110100 | 4 |
| 53 | 35 | 00110101 | 5 | 181 | B5 | 10110101 | 5 |
| 54 | 36 | 00110110 | 6 | 182 | B6 | 10110110 | 6 |
| 55 | 37 | 00110111 | 7 | 183 | B7 | 10110111 | 7 |
| 56 | 38 | 00111000 | 8 | 184 | B8 | 10111000 | 8 |
| 57 | 39 | 00111001 | 9 | 185 | B9 | 10111001 | 9 |
| 58 | 3A | 00111010 | : | 186 | BA | 10111010 | : |
| 59 | 3B | 00111011 | ; | 187 | BB | 10111011 | ; |
| 60 | 3C | 00111100 | < | 188 | BC | 10111100 | < |
| 61 | 3D | 00111101 | = | 189 | BD | 10111101 | = |
| 62 | 3E | 00111110 | > | 190 | BE | 10111110 | > |
| 63 | 3F | 00111111 | ? | 191 | BF | 10111111 | ? |
| 64 | 40 | 01000000 | @ | 192 | C0 | 11000000 | @ |
| 65 | 41 | 01000001 | A | 193 | C1 | 11000001 | A |
| 66 | 42 | 01000010 | B | 194 | C2 | 11000010 | B |
| 67 | 43 | 01000011 | C | 195 | C3 | 11000011 | C |
| 68 | 44 | 01000100 | D | 196 | C4 | 11000100 | D |
| 69 | 45 | 01000101 | E | 197 | C5 | 11000101 | E |
| 70 | 46 | 01000110 | F | 198 | C6 | 11000110 | F |
| 71 | 47 | 01000111 | G | 199 | C7 | 11000111 | G |
| 72 | 48 | 01001000 | H | 200 | C8 | 11001000 | H |
| 73 | 49 | 01001001 | I | 201 | C9 | 11001001 | I |
| 74 | 4A | 01001010 | J | 202 | CA | 11001010 | J |
| 75 | 4B | 01001011 | K | 203 | CB | 11001011 | K |
| 76 | 4C | 01001100 | L | 204 | CC | 11001100 | L |
| 77 | 4D | 01001101 | M | 205 | CD | 11001101 | M |
| 78 | 4E | 01001110 | N | 206 | CE | 11001110 | N |
| 79 | 4F | 01001111 | O | 207 | CF | 11001111 | O |
| 80 | 50 | 01010000 | P | 208 | D0 | 11010000 | P |
| 81 | 51 | 01010001 | Q | 209 | D1 | 11010001 | Q |
| 82 | 52 | 01010010 | R | 210 | D2 | 11010010 | R |
| 83 | 53 | 01010011 | S | 211 | D3 | 11010011 | S |
| 84 | 54 | 01010100 | T | 212 | D4 | 11010100 | T |
| 85 | 55 | 01010101 | U | 213 | D5 | 11010101 | U |
| 86 | 56 | 01010110 | V | 214 | D6 | 11010110 | V |
| 87 | 57 | 01010111 | W | 215 | D7 | 11010111 | W |
| 88 | 58 | 01011000 | X | 216 | D8 | 11011000 | X |
| 89 | 59 | 01011001 | Y | 217 | D9 | 11011001 | Y |
| 90 | 5A | 01011010 | Z | 218 | DA | 11011010 | Z |
| 91 | 5B | 01011011 | [ | 219 | DB | 11011011 | [ |
| 92 | 5C | 01011100 | \ | 220 | DC | 11011100 | \ |
| 93 | 5D | 01011101 | ] | 221 | DD | 11011101 | ] |
| 94 | 5E | 01011110 | ^ | 222 | DE | 11011110 | ^ |
| 95 | 5F | 01011111 | _ | 223 | DF | 11011111 | _ |
| 96 | 60 | 01100000 | ` | 224 | E0 | 11100000 | ` |
| 97 | 61 | 01100001 | a | 225 | E1 | 11100001 | a |
| 98 | 62 | 01100010 | b | 226 | E2 | 11100010 | b |
| 99 | 63 | 01100011 | c | 227 | E3 | 11100011 | c |
| 100 | 64 | 01100100 | d | 228 | E4 | 11100100 | d |

## Decimal / Hex / Binary / ASCII Conversions

| Dec | Hex | Binary | Asc |
|---|---|---|---|
| 101 | 65 | 01100101 | e |
| 102 | 66 | 01100110 | f |
| 103 | 67 | 01100111 | g |
| 104 | 68 | 01101000 | h |
| 105 | 69 | 01101001 | i |
| 106 | 6A | 01101010 | j |
| 107 | 6B | 01101011 | k |
| 108 | 6C | 01101100 | l |
| 109 | 6D | 01101101 | m |
| 110 | 6E | 01101110 | n |
| 111 | 6F | 01101111 | o |
| 112 | 70 | 01110000 | p |
| 113 | 71 | 01110001 | q |
| 114 | 72 | 01110010 | r |
| 115 | 73 | 01110011 | s |
| 116 | 74 | 01110100 | t |
| 117 | 75 | 01110101 | u |
| 118 | 76 | 01110110 | v |
| 119 | 77 | 01110111 | w |
| 120 | 78 | 01111000 | x |
| 121 | 79 | 01111001 | y |
| 122 | 7A | 01111010 | z |
| 123 | 7B | 01111011 | { |
| 124 | 7C | 01111100 | \| |
| 125 | 7D | 01111101 | } |
| 126 | 7E | 01111110 | ~ |
| 127 | 7F | 01111111 | ■ |

| Dec | Hex | Binary | Asc |
|---|---|---|---|
| 229 | E5 | 11100101 | e |
| 230 | E6 | 11100110 | f |
| 231 | E7 | 11100111 | g |
| 232 | E8 | 11101000 | h |
| 233 | E9 | 11101001 | i |
| 234 | EA | 11101010 | j |
| 235 | EB | 11101011 | k |
| 236 | EC | 11101100 | l |
| 237 | ED | 11101101 | m |
| 238 | EE | 11101110 | n |
| 239 | EF | 11101111 | o |
| 240 | F0 | 11110000 | p |
| 241 | F1 | 11110001 | q |
| 242 | F2 | 11110010 | r |
| 243 | F3 | 11110011 | s |
| 244 | F4 | 11110100 | t |
| 245 | F5 | 11110101 | u |
| 246 | F6 | 11110110 | v |
| 247 | F7 | 11110111 | w |
| 248 | F8 | 11111000 | x |
| 249 | F9 | 11111001 | y |
| 250 | FA | 11111010 | z |
| 251 | FB | 11111011 | { |
| 252 | FC | 11111100 | \| |
| 253 | FD | 11111101 | } |
| 254 | FE | 11111110 | ~ |
| 255 | FF | 11111111 | ■ |

# HP-71 MEMORY MAP



# HP-71 SYSTEM RAM

| Address | Name | #nibs | |
|---|---|---|---|
| ---------- Display Driver ---------- | | | |
| 2E100 | ANNAD1 | 1 | |
| 2E101 | ANN1.5 | 1 | |
| 2E102 | ANNAD2 | 2 | |
| 2E104 | DD3ST | | |
| 2E160 | DD3END | | |
| 2E1F8 | TIMER3 | | |
| 2E1FF | DD3CTL | 1 | |
| 2E200 | DD2ST | | |
| 2E260 | DD2END | | |
| 2E2F8 | TIMER2 | | |
| 2E2FF | DD2CTL | 1 | |
| 2E300 | DD1ST | | |
| 2E34C | ANNAD3 | 2 | |
| 2E34E | ANNAD4 | 2 | |
| 2E350 | ROWDVR | | |
| 2E3F8 | TIMER1 | | |
| 2E3FE | DCONTR | 1 | Contrast nib |
| 2E3FF | DD1CTL | | |
| ---------- Interrupt RAM ---------- | | | |
| 2E400 | INTR4 | 16 | |
| 2F410 | INTA | 16 | |
| 2F420 | INTB | 16 | |
| 2F430 | INTM | 8 | |
| 2F438 | CMOSTW | 4 | CMOS test #168F |
| 2F43C | VECTOR | 5 | Interrupt vector |
| 2F441 | ATNDIS | 1 | ATTN key disable |
| 2F442 | ATNFLG | 1 | ATTN key hit? |
| 2F443 | KEYPTR | 1 | Key buf pointer |
| 2F444 | KEYBUF | 15*2 | Key buffer |
| 2F471 | WINDST | 2 | Disp window start |
| 2F473 | WINDLN | 2 | Window length |
| 2F475 | DSPSTA | 6 | Display status |
| 2F47B | ESCSTA | 1 | Escape status |
| 2F47C | FIRSTC | 2 | Buf pos of 1st chr |
| 2F480 | DSPBFS | 2*96 | Input buffer |
| 2F540 | DSPMSK | 96/4 | Input mask |
| ---------- System Pointers ---------- | | | |
| 2F558 | MAINST | 5 | Main Pgm mem start |
| 2F55D | CURRST | 5 | Current file start |
| 2F562 | PRGMST | 5 | Current Program |
| 2F567 | PRGMEN | 5 | Current Pgm end |
| 2F56C | CURREN | 5 | Current file end |
| 2F571 | IOBFST | 5 | System buffers |
| 2F576 | IOBFEN | 5 | System buf end |
| 2F576 | CLCBFR | 5 | Calc Mode ptrs |
| 2F57B | RFNBFR | 5 | |
| 2F580 | RAWBFR | 5 | |
| 2F585 | CLCSTK | 5 | |
| 2F58A | SYSEN | 5 | End of system ram |
| 2F58F | OUTBS | 5 | Output buffer |
| 2F594 | AVMEMS | 5 | Free memory start |
| 2F599 | AVMEME | 5 | Free memory end |
| 2F599 | MTHSTK | 5 | Math stack |
| 2F59E | FORSTK | 5 | FOR/NEXT stack |
| 2F5A3 | GSBSTK | 5 | GOSUB stack |
| 2F5A8 | ACTIVE | 5 | Active vars space |
| 2F5AD | CALSTK | 5 | CALL stack |
| 2F5B2 | RAMEND | 5 | End of memory |
| 2F5B7 | PRMPTR | 5 | |
| 2F5BE | CHNLST | 26*7 | |
| 2F674 | DSPCHX | 5 | External display |
| 2F679 | PCADDR | 5 | Pgm counter |
| 2F67E | CNTADR | 5 | Cont address |
| 2F683 | ERRSUB | 5 | ON ERROR addr |
| 2F688 | ERRADR | 5 | ON ERR stmt addr |
| 2F68D | ONINTR | 5 | ON INTR stmt addr |
| 2F692 | DATPTR | 5 | DATA stmt pointer |
| 2F697 | TMRAD1 | 5 | TMR#1 stmt addr |
| 2F69C | TMRAD2 | 5 | TMR#2 stmt addr |
| 2F6A1 | TMRAD3 | 5 | TMR#3 stmt addr |
| 2F6A6 | TMRIN1 | 8 | TMR#1 interval |
| 2F6AE | TMRIN2 | 8 | TMR#2 interval |
| 2F6B6 | TMRIN3 | 8 | TMR#3 interval |
| 2F6C1 | LDCSPC | 5 | Space after line# |
| 2F6C6 | INBS | 5 | Input buf start |
| 2F6CB | AUTINC | 4 | Increment for AUTO |
| 2F6CF | LEXPTR | 5 | Temporary RESPTR |
| 2F6D4 | CMDPTR | 5 | CMMD stack ptr |
| 2F6D4 | INADDR | 5 | Stmt len parse/dec |
| 2F6D9 | SYSFLG | 16 | System flags |
| 2F6E9 | FLGREG | 16 | User flags |
| -------- Math Exception traps --------- | | | |
| 2F6F9 | INXNIB | 1 | Inexact result |
| 2F6FA | UNFNIB | 1 | Underflow |
| 2F6FB | OVFNIB | 1 | Overflow |
| 2F6FC | DVZNIB | 1 | Divide by zero |
| 2F6FD | IVLNIB | 1 | Invalid result |
| 2F6FE | RNSEED | 15 | Random# seed |
| ---------- Alarm Clock ---------- | | | |
| 2F70D | NXTIRQ | 12 | Next SREQ |
| 2F719 | ALRM1 | 12 | Timer #1 |
| 2F725 | ALRM2 | 12 | Timer #2 |
| 2F731 | ALRM3 | 12 | Timer #3 |
| 2F73D | ALRM4 | 12 | Timeout timer |
| 2F749 | ALRM5 | 12 | WAIT timer |
| 2F755 | ALRM6 | 12 | External alarm |
| 2F761 | PNDALM | 2 | Bitmap pending alm |
| ---------- Clock Accuracy ---------- | | | |
| 2F763 | TIMOFS | 12 | Time error offset |
| 2F76F | TIMLST | 12 | Time last set |
| 2F77B | TIMLAF | 12 | Last AF corection |
| 2F787 | TIMAF | 6 | Accuracy factor |
| HP-IL Device Assignments | | | |
| 2F78D | IS-DSP | 7 | Display |
| 2F794 | IS-PRT | 7 | Printer |

# HP-71 SYSTEM RAM

| | | | |
|---|---|---|---|
| 2F79B | IS-INP | 7 | Keyboard |
| 2F7A2 | IS-PLT | 7 | |
| 2F7A9 | MBOX^ | 3 | HP-IL Mailbox ptr |
| 2F7AC | LOOPST | 1 | HP-IL loop status |
| 2F7AD | STATAR | 3 | STAT array name |
| 2F7B0 | TRACEM | 1 | TRACE mode(0,2,4,6 |
| 2F7B1 | DSPSET | 1 | Display status |
| 2F7B2 | LOCKWD | 8*2 | Password |
| 2F7C2 | RESREG | 34 | RES register |
| 2F7E4 | ERR# | 4 | ERRN |
| 2F78E | CURRL | 4 | Current line # |
| 2F7EC | ERRL# | 4 | ERRL |
| ---------- Scratch RAM ---------- | | | |
| 2F871 | STMTR0 | 16 | |
| 2F871 | S-R0-0 | 5 | |
| 2F876 | S-R0-1 | 5 | |
| 2F87B | S-R0-2 | 5 | |
| 2F880 | S-R0-3 | 1 | |
| 2F881 | STMTR1 | 16 | |
| 2F881 | S-R1-0 | 5 | |
| 2F886 | S-R1-1 | 5 | |
| 2F88B | S-R1-2 | 5 | |
| 2F890 | S-R1-3 | 1 | |
| 2F891 | STMTD0 | 5 | Temp D0 |
| 2F896 | STMTD1 | 5 | Temp D1 |
| ---------- Function Scratch ---------- | | | |
| 2F89B | FUNCR0 | 16 | |
| 2F89B | F-R0-0 | 5 | |
| 2F8A0 | F-R0-1 | 5 | |
| 2F8A5 | F-R0-2 | 5 | |
| 2F8AA | F-R0-3 | 1 | |
| 2F8AB | FUNCR1 | 16 | |
| 2F8AB | F-R1-0 | 5 | |
| 2F8B0 | F-R1-1 | 5 | |
| 2F8B5 | F-R1-2 | 5 | |
| 2F8BA | F-R1-3 | 1 | |
| 2F8BB | FUNCD0 | 5 | Temp D0 |
| 2F8C0 | FUNCD1 | 5 | Temp D1 |
| 2F8C5 | TRFMBF | 60 | TRANSFORM scratch |
| 2F901 | SCRST0 | 4*16 | Scratch stack(Mant |
| 2F941 | SCREX0 | 5 | Scratch stack(exp |
| ---------- Display/Print ---------- | | | |
| 2F946 | SCROLT | 2 | Disp scroll rate |
| 2F948 | DELAYT | 2 | Disp delay rate |
| 2F94A | NEEDSC | 1 | Scroll needed |
| 2F94D | DPOS | 2 | Current DISP col |
| 2F94F | DWIDTH | 2 | Disp width |
| 2F956 | PPOS | 2 | Current PRINT col |
| 2F958 | PWIDTH | 2 | Print width |
| 2F95A | EOLLEN | 1 | Len of ENDLINE |
| 2F95B | EOLSTR | 2*3 | ENDLINE string |
| 2F976 | MAXCMD | 1 | #CMMD stack entries |
| 2F977 | CSPEED | 5 | Clock Speed(Hz/16) |
| ---------- HP-IL ---------- | | | |
| 2F97C | ERRLCH | 1 | |
| 2F97D | TERCHR | 2 | ENTER term chr |
| 2F97F | HPSCRH | 7 | Reserved |
| 2F986 | RESERV | 48*2 | Reserved |

# HP-71B KEYMAP

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| g- | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 |
| f- | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| g- | q | w | e | r | t | y | u | i | o | p | ' | { | } | ^ |
| f- | IF | THEN | ELSE | FOR | TO | NEXT | DEF | KEY | ADD | LR | PRED | MEAN | SDEV | SQR |
| | Q | W | E | R | T | Y | U | I | O | P | 7 | 8 | 9 | / |
| g- | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 |
| f- | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 |
| | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| g- | a | s | d | f | g | h | j | k | l | ; | $ | % | & | : |
| f- | CALL | GOSU | RETU | GOTO | INPU | PRIN | DISP | DIM | BEEP | FACT | SIN | COS | TAN | EXP |
| | A | S | D | F | G | H | J | K | L | = | 4 | 5 | 6 | * |
| g- | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 |
| f- | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 |
| | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 |
| g- | z | x | c | v | b | n | m | [ | ] | CMDS | ! | " | # | @ |
| f- | EDIT | CAT | NAME | PURG | FETC | LIST | DELE | AUTO | COPY | RES | ASIN | ACOS | ATAN | LOG |
| | Z | X | C | V | B | N | M | ( | ) | END | 1 | 2 | 3 | - |
| g- | 155 | | | 158 | 159 | 160 | 161 | 162 | 163 | L | 165 | 166 | 167 | 168 |
| f- | 99 | | | 102 | 103 | 104 | 105 | 106 | 107 | I | 108 | 109 | 110 | 111 |
| | 43 | | | 46 | 47 | 48 | 49 | 50 | 51 | N | 53 | 54 | 55 | 56 |
| g- | | | | CTRL | ⏮ | ⏭ | ERRM | ⏫ | ⏬ | E | USR | ( | ) | ? |
| f- | OFF | | | SST | BACL | -CHR | I/R | LC | -LIN | | USER | VIEW | CALC | CONT |
| | ON | f- | g- | RUN | ◀ | ▶ | SPC | ▲ | ▼ | | 0 | . | , | + |

# HP-71 SYSTEM FLAGS

Addr # If set means:

2F6D9 -1 Suppress warning messages
-2 Beeper is off
-3 Continuous power on
-4 Inexact result trap (INX)
2F6DA -5 Underflow trap (UNX)
-6 Overflow trap (OVF)
-7 Divide by zero trap (DVZ)
-8 Invalid operation trap (IVL)
2F6DB -9 USER mode set
-10 OPTION ANGLE RADIANS

Round Off Setting

| | near zero | pos | neg |
|---|---|---|---|
| -11 to INF | 0 | 1 | 1 |
| -12 Neg Round | 1 | 0 | 1 |

Display Format

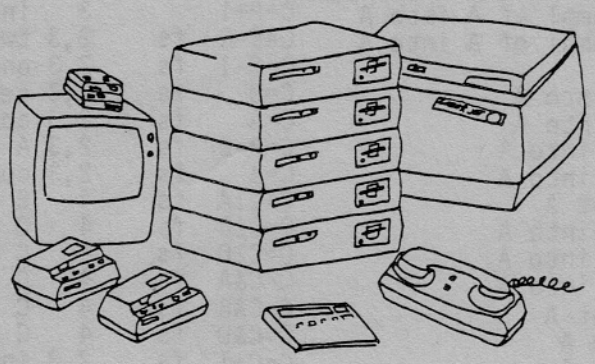
| | STD | FIX | SCI | ENG |
|---|---|---|---|---|
| 2F6DC -13 | 0 | 1 | 0 | 1 |
| -14 | 0 | 0 | 1 | 1 |

-15 Lowercase mode (LC ON)
-16 OPTION BASE 1

Display Digits

| fix | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2F6DD -17 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| -18 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| -19 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| -20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

2F6DE -21 Auto loop power down off
-22 Use extended HP-IL addressing
-23 HP-IL ENTER terminated by EOT
-24 Don't re-assign dev RESTOREIO
2F6DF -25 Beep loud
-26 Don't show BASIC prompt
-27 Alt. error message language
-28 !TEST ONLY
2F6E0 -29 !TEST ONLY
-30 !TEST ONLY
-31 !TEST ONLY
-32 !TEST ONLY
2F6E1 -33 !TEST ONLY
-34 !TEST ONLY
-35 !TEST ONLY
-36 !TEST ONLY
2F6E2 -37 !TEST ONLY
-38 !TEST ONLY
-39 !TEST ONLY
-40 !TEST ONLY
2F6E3 -41 !TEST ONLY
-42 Plug-in module was pulled
-43 HP-71 is dormant
-44 Always return from MEMERR
2F6E4 -45 Clock mode (1 second update)
-46 Clock EXACT
-47 Command stack active
-48 Control key hit
2F6E5 -49 DSLEEP from power down
-50 Req set TRNOF in MAINLP
-51 Turnoff at MAINLP
-52 VIEW key pressed
2F6E6 -53 !Future use
-54 !Future use
-55 !Future use
-56 !Future use
2F6E7 -57 "AC" Annunciator lit
-58 USER suspended (see flag -9)
-59 Key repeated
-60 "((*))" Alarm annunciator lit
2F6E8 -61 "BAT" annunciator lit
-62 "PRGM" annunciator lit
-63 "SUSP" annunciator lit
-64 "CALC" annunciator lit



# ASSEMBLER INSTRUCTION SET

| inst | field | nibs | |
|---|---|---|---|
| ?A#0 | fs | 5 | test A#0 |
| ?A#B | fs | 5 | test A#B |
| ?A#C | fs | 5 | test A#C |
| ?A<=B | fs | 5 | test A<=B |
| ?A<B | fs | 5 | test A<B |
| ?A=0 | fs | 5 | test A=0 |
| ?A=B | fs | 5 | test A=B |
| ?A=C | fs | 5 | test A=B |
| ?A>=B | fs | 5 | test A>=B |
| ?A>B | fs | 5 | test A>B |
| ?B#0 | fs | 5 | test B#0 |
| ?B#A | fs | 5 | test B#A |
| ?B#C | fs | 5 | test B#C |
| ?B<=C | fs | 5 | test B<=C |
| ?B<C | fs | 5 | test B<C |
| ?B=0 | fs | 5 | test B=0 |
| ?B=A | fs | 5 | test B=A |
| ?B=C | fs | 5 | test B=C |
| ?B>=C | fs | 5 | test B>=C |
| ?B>C | fs | 5 | test B>C |
| ?C#0 | fs | 5 | test C#0 |
| ?C#A | fs | 5 | test C#A |
| ?C#B | fs | 5 | test C#B |
| ?C#D | fs | 5 | test C#D |
| ?C<=A | fs | 5 | test C<=A |
| ?C<A | fs | 5 | test C<A |
| ?C=0 | fs | 5 | test C=0 |
| ?C=A | fs | 5 | test C=A |
| ?C=B | fs | 5 | test C=B |
| ?C=D | fs | 5 | test C=D |
| ?C>=A | fs | 5 | test C>=A |
| ?C>A | fs | 5 | test C>A |
| ?D#0 | fs | 5 | test D#0 |
| ?D#C | fs | 5 | test D#C |
| ?D<=C | fs | 5 | test D<=C |
| ?D<C | fs | 5 | test D<C |
| ?D=0 | fs | 5 | test D=0 |
| ?D=C | fs | 5 | test D=C |
| ?D>=C | fs | 5 | test D>=C |
| ?D>C | fs | 5 | test D>C |
| ?MP=0 | | 5 | test module pulled bit |
| ?P# | n | 5 | test pointer#n |
| ?P= | n | 5 | test pointer=n |
| ?SB=0 | | 5 | test sticky bit=0 |
| ?SR=0 | | 5 | test service requst bit=0 |
| ?ST#0 | n | 5 | test status bit n#0 |
| ?ST#1 | n | 5 | test status bit n#1 |
| ?ST=0 | n | 5 | test status bit n=0 |
| ?ST=1 | n | 5 | test status bit n=1 |
| ?XM=0 | | 5 | test external mod missing |
| A=-A | fs | 3 | two's compl of A into A |
| A=A-1 | fs | 3 | one's compl of A into A |
| A=0 | fs | 2,3 | set A=0 |
| A=A!B | fs | 4 | A OR B into A |
| A=A!C | fs | 4 | A OR C into A |
| A=A&B | fs | 4 | A AND B into A |
| A=A&C | fs | 4 | A AND C into A |
| A=A+1 | fs | 2,3 | increment A |
| A=A+A | fs | 2,3 | sum A+A into A |
| A=A+B | fs | 2,3 | sum A+B into A |
| A=A+C | fs | 2,3 | sum A+C into A |
| A=A-1 | fs | 2,3 | decrement A |
| A=A-B | fs | 2,3 | A-B into A |
| A=A-C | fs | 2,3 | A-C into A |
| A=B | fs | 2,3 | copy B to A |
| A=B-A | fs | 2,3 | B-A into A |
| A=C | fs | 2,3 | copy C to A |
| A=DAT0 | fsd | 3,4 | load A from mem using D0 |
| A=DAT1 | fsd | 3,4 | load A from mem using D1 |
| A=IN | | 3 | load A(0-3)=input reg |
| A=R0 | | 3 | copy R0 to A |
| A=R1 | | 3 | copy R1 to A |
| A=R2 | | 3 | copy R2 to A |
| A=R3 | | 3 | copy R3 to A |
| A=R4 | | 3 | copy R4 to A |
| ABEX | | 2,3 | exchange A and B |
| ACEX | | 2,3 | exchange A and C |
| ADOEX | | 3 | exchange A(A) and D0 |
| ADOXS | | 3 | exchange A(0-3) and D0 |
| AD1EX | | 3 | exchange A(A) and D1 |
| AD1XS | | 3 | exchange A(0-3) and D1 |
| AR0EX | | 3 | exchange A and R0 |
| AR1EX | | 3 | exchange A and R1 |
| AR2EX | | 3 | exchange A and R2 |
| AR3EX | | 3 | exchange A and R3 |
| AR4EX | | 3 | exchange A and R4 |
| ASL | fs | 2,3 | shift A left 1 nib |
| ASLC | | 3 | shift A left 1 nib circ |
| ASR | fs | 2,3 | shift A right 1 nib |
| ASRB | | 3 | shift A right 1 bit |
| ASRC | | 3 | shift A right 1 nib circ |
| B=-B | fs | 2,3 | two's compl of B into B |
| B=B-1 | fs | 2,3 | one's compl of B into B |
| B=0 | fs | 2,3 | set B=0 |
| B=A | fs | 2,3 | copy A to B |
| B=B!A | fs | 4 | B OR A into B |
| B=B!C | fs | 4 | B OR C into B |
| B=B&A | fs | 4 | B AND A into B |
| B=B&C | fs | 4 | B AND C into B |
| B=B+1 | fs | 2,3 | increment B |
| B=B+A | fs | 2,3 | sum B+A into B |
| B=B+B | fs | 2,3 | sum B+B into B |
| B=B+C | fs | 2,3 | sum B+C into B |
| B=B-1 | fs | 2,3 | decrement B |
| B=B-A | fs | 2,3 | B minus A into B |
| B=B-C | fs | 2,3 | B minus C into B |
| B=C | fs | 2,3 | copy C to B |
| B=C-B | fs | 2,3 | C minus B to B |
| BAEX | fs | 2,3 | exchange B and A |
| BCEX | fs | 2,3 | exchange B and C |
| BSL | fs | 2,3 | shift B left 1 nib |
| BSLC | | 3 | shift B left 1 nib circ |
| BSR | fs | 2,3 | shift B right 1 nib |
| BSRB | | 3 | shift B right 1 bit |
| BSRC | | 3 | shift B right 1 nib circ |
| BUSCC | | 3 | enters bus command "C" |
| C+P+1 | | 3 | increment C plus P pointer |
| C=C-C | fs | 2,3 | two's compl of C into C |
| C=C-1 | fs | 2,3 | one's compl of C into C |
| C=0 | fs | 2,3 | set C=0 |
| C=A | fs | 2,3 | copy A to C |
| C=A-C | fs | 2,3 | A minus C to C |
| C=B | fs | 2,3 | copy B into C |
| C=C!A | fs | 4 | C OR A into C |
| C=C!B | fs | 4 | C OR B into C |
| C=C!D | fs | 4 | C OR D into C |
| C=C&A | fs | 4 | C AND A into C |
| C=C&B | fs | 4 | C AND B into C |
| C=C&D | fs | 4 | C AND D into C |
| C=C+1 | fs | 2,3 | increment C |
| C=C+A | fs | 2,3 | sum C+A into C |
| C=C+B | fs | 2,3 | sum C+B into C |

# ASSEMBLER INSTRUCTION SET

| | | | |
|---|---|---|---|
| C=C+C | fs | 2,3 | sum C+C into C |
| C=C+D | fs | 2,3 | sum C+D into C |
| C=C-1 | fs | 2,3 | decrement C |
| C=C-A | fs | 2,3 | C minus A into C |
| C=C-B | fs | 2,3 | C minus B into C |
| C=C-D | fs | 2,3 | C minus D into C |
| C=D | fs | 2,3 | copy D into C |
| C=DAT0 | fsd | 3,4 | load C from mem using D0 |
| C=DAT1 | fsd | 3,4 | load C from mem using D1 |
| C=ID | | 3 | request chip ID to C(A) |
| C=IN | | 3 | load C(0-3)with input reg |
| C=P | n | 4 | copy P pointer to C(n) |
| C=R0 | | 3 | copy R0 to C |
| C=R1 | | 3 | copy R1 to C |
| C=R2 | | 3 | copy R2 to C |
| C=R3 | | 3 | copy R3 to C |
| C=R4 | | 3 | copy R4 to C |
| C=RSTK | | 2 | pop return stack to C(A) |
| C=ST | | 2 | copy status reg to C(X) |
| CAEX | fs | 2,3 | exchange C and A |
| CBEX | fs | 2,3 | exchange C and B |
| CD0EX | | 3 | exchange C(A) and D0 |
| CD0XS | | 3 | exchange C(0-3) and D0 |
| CD1EX | | 3 | exchange C(A) and D1 |
| CD1XS | | 3 | exchange C(0-3) and D1 |
| CDEX | fs | 2 | exchange C and D |
| CLRHST | | 3 | clear hardware status bits |
| CLRST | | 2 | clear program status bits |
| CONFIG | | 3 | configure(see hardwar spec) |
| CPEX | fs | 4 | exchange C(n) and P |
| CR0EX | | 3 | exchange C and R0 |
| CR1EX | | 3 | exchange C and R1 |
| CR2EX | | 3 | exchange C and R2 |
| CR3EX | | 3 | exchange C and R3 |
| CR4EX | | 3 | exchange C and R4 |
| CSL | fs | 2,3 | shift C left 1 nib |
| CSLC | | 3 | shift C left 1 nib circ |
| CSR | fs | 2,3 | shift C right 1 nib |
| CSRB | | 3 | shift C right 1 bit |
| CSRC | | 3 | shift C right 1 nib circ |
| CSTEX | | 2 | exch C(X)and status reg |
| D0=(2) | nn | 4 | load 2 nibs into D0 |
| D0=(4) | nnnn | 6 | load 4 nibs into D0 |
| D0=(5) | nnnnn | 7 | load 5 nibs into D0 |
| D0=A | | 3 | copy A(A) to D0 |
| D0=AS | | 3 | copy A(0-3) to D0 |
| D0=C | | 3 | copy C(A) to D0 |
| D0=CS | | 3 | copy C(0-3) to D0 |
| D0=D0+ | n | 3 | add n to D0 (0<n<=16) |
| D0=D0- | n | 3 | subtract n from D0 |
| D0=HEX | hh | 4 | load D0 with hex const |
| D0=HEX | hhhh | 6 | load D0 with hex const |
| D0=HEX | hhhhh | 7 | load D0 with hex const |
| D1=(2) | nn | 4 | load 2 nibs into D1 |
| D1=(4) | nnnn | 6 | load 4 nibs into D1 |
| D1=(5) | nnnnn | 7 | load 5 nibs into D1 |
| D1=A | | 3 | copy A(A) to D1 |
| D1=AS | | 3 | copy A(0-3) to D1 |
| D1=C | | 3 | copy C(A) to D1 |
| D1=CS | | 3 | copy C(0-3) to D1 |
| D1=D1+ | n | 3 | add n to D1 (0<n<=16) |
| D1=D1- | n | 3 | subtract n from D1 |
| D1=HEX | hh | 4 | load D1 with hex const |
| D1=HEX | hhhh | 6 | load D1 with hex const |
| D1=HEX | hhhhh | 7 | load D1 with hex const |
| D=-D | fs | 2,3 | two's compl of D into D |
| D=-D-1 | fs | 2,3 | one's compl of D into D |
| D=0 | fs | 2,3 | set |
| D=C | fs | 2,3 | copy C to D |
| D=C-D | fs | 2,3 | C-D into D |
| D=D!C | fs | 4 | D OR C into D |
| D=D&C | fs | 4 | D AND C into D |
| D=D+1 | fs | 2,3 | increment D |
| D=D+C | fs | 2,3 | sum D+C to D |
| D=D+D | fs | 2,3 | sum D+D to D |
| D=D-1 | fs | 2,3 | decrement D |
| D=D-C | fs | 2,3 | D minus C to D |
| DAT0=A | fsd | 3,4 | copy to mem from A using D0 |
| DAT0=C | fsd | 3,4 | copy to mem from C using D0 |
| DAT1=A | fsd | 3,4 | copy to mem from A using D1 |
| DAT1=C | fsd | 3,4 | copy to mem from C using D1 |
| DCEX | fs | 2,3 | exchange D and C |
| DSL | fs | 2,3 | shift D left 1 nib |
| DSLC | | 3 | shift D left 1 nib circ |
| DSR | fs | 2,3 | shift D right 1 nib |
| DSRB | | 3 | shift D right 1 bit |
| DSRC | | 3 | shift D right 1 nib circ |
| GOC | | 3 | go label if carry set(<=127) |
| GOLONG | | 6 | go long to label (<-32767) |
| GONC | | 3 | go label if no carry (<=127) |
| GOSBVL | | 7 | gosub very long to label |
| GOSUB | | 4 | gosub to label (<=2047) |
| GOSUBL | | 6 | gosub long label (<=32767) |
| GOTO | | 4 | goto label (<=2048) |
| GOVLNG | | 7 | go very long to label |
| GOYES | | 2 | jump if test is true |
| INTOFF | | 4 | interrupt off |
| INTON | | 4 | interrupt on |
| LC(m) | | 3+m | load C with const.(0<m<=6) |
| LCASC | | 3+n | load C with ASCII using P |
| LCHEX | | 3+n | load C with hex using P |
| MP=0 | | 3 | clear module pulled bit |
| NOP3 | | 3 | three nib no-op |
| NOP4 | | 4 | four nib no-op |
| NOP5 | | 5 | five nib no-op |
| OUT=C | | 3 | load output reg with C(X) |
| OUT=CS | | 3 | load output reg with C(0) |
| P=C | n | 4 | copy C at nib n to P |
| P=P+1 | | 2 | increment P pointer |
| P=P-1 | | 2 | decrement P pointer |
| P= | n | 2 | set P pointer to n |
| R0=A | | 3 | copy A to R0 |
| R0=C | | 3 | copy C to R0 |
| R1=A | | 3 | copy A to R1 |
| R1=C | | 3 | copy C to R1 |
| R2=A | | 3 | copy A to R2 |
| R2=C | | 3 | copy C to R2 |
| R3=A | | 3 | copy A to R3 |
| R3=C | | 3 | copy C to R3 |
| R4=A | | 3 | copy A to R4 |
| R4=C | | 3 | copy C to R4 |
| RESET | | 3 | system bus reset command |
| RSTK=C | | 2 | push C(A) onto rtn stack |
| RTI | | 2 | return from interrupt |
| RTN | | 2 | return |
| RTNC | | 3 | return if carry set |
| RTNCC | | 2 | clear carry, return |
| RTNNC | | 3 | return if carry clear |
| RTNSC | | 2 | set carry, return |
| RTNSXM | | 2 | set ext mod missing,return |
| RTNYES | | 2 | return if test is true |
| SB=0 | | 3 | clear sticky bit |
| SETDEC | | 2 | set cpu to decimal mode |

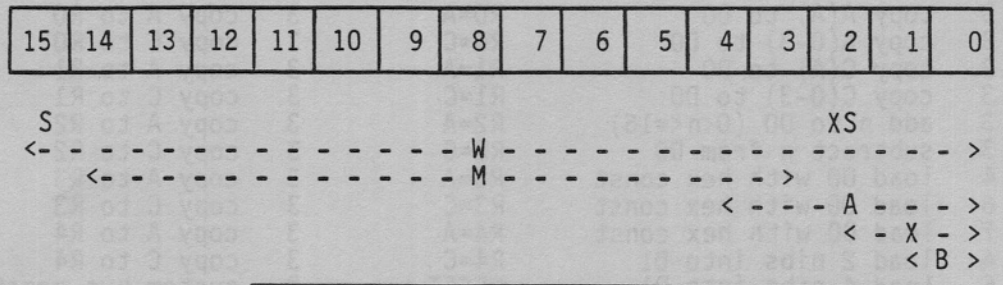
# ASSEMBLER INSTRUCTION SET

```
                                        e - expression      l - label
                                        b - byte            '' - quoted string
SETHEX       2   set cpu to hexadecimal mode  n - nib or nibs
SHUTDN       3   shut down bus and cpu
SR=0         3   clear service request bit   BIN    ''  assemble a BIN file
SREQ?        3   poll for service request    BSS    e   evaluate then enter n nibs of '0'
ST=0   n     3   clear program status bit n  CHAIN  ''  subheader for BIN files
ST=1   n     3   set program status bit n    CHAR   n   type of BASIC Keyword
ST=C         2   copy C(X) to status reg     CON(i) e   evaluate expr, enter (i) nibs
UNCNFG       3   unconfigure                 EJECT      form feed in the listing
XM=0         3   clear ext mod missing bit   END        mark end of file (optional)
                                             ENDTXT     mark end of keyword table
                                             ENTRY  l   begin def of BASIC keyword
                                             EAU    l   define a label for entry point
                                             FORTH      assemble a FORTH primitive
                                             ID     b   LEX ID of the file
                                             KEY    ''  define a keyword name
                                             LEX    ''  assemble a LEX file
                                             LIST OFF   disable use of listing file
                                             LIST ON    enable use of listing file
                                             MSG    l   point to message tbl (or 0)
                                             NIBASC ''  enter up to 8 ASCII chars
                                             NIBHEX n   enter up to 16 hex nibs
                                             POLL   l   define poll handler (or 0)
                                             STITLE ''  formfeed, add subtitle
                                             TITLE  ''  title the listing
                                             TOKEN  n   token number of keyword
                                             WORD   ''  define FORTH primitive
                                             WORDI  ''  define immediate FORTH primitive
```

## MINIMUM LEX FILE REQUIREMENTS

```
        LEX     'FILENAME'
        ID      #nn
        MSG     0
        POLL    0
?????? EQU     #nnnnn
        ENTRY   lbl         keyword entry
        CHAR    #n          F= all uses
        KEY     'KEYWORD'   $ for string functions
        TOKEN   b           1-255
* beginning of actual code
lbl
```

## FIELDS WITHIN A WORKING REGISTER




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

# HP-71 ENTRY POINTS

**A-MULT 1B349** Multiply two 20 bit Hex integers. Entry: integers in A(A),C(A)
Exit: P preserved. A(A)=product. Carry if ok. Carry clear if overflow; A(A)=FFFFF.
Levels: 0. Uses: A(A),B(A),C(A),C(14)

**ADHEAD 181B7** Add string header to string on stack.
Entry: R1(A)=start of item (high mem). D1=end of item (low mem). S0 set if RTN needed else jumps to EXPR. P=0
Exit: D1 points at str hdr.
Levels: 2. Uses: A(A),C(W),D1

**ARGPRP 0E8EF** Pop & normalize a REAL. As ARGPR+ but user modes not checked.
**ARGPR+ 0E8EB** Read user modes, pops & normalize real. Split and normalizes arg to 15 digits.
Entry: Number on top of stack. D1 points to top of stack.
Exit: DECMODE, A/B=15 digit form of arg. If signaling Nan then carry set, XM=1 else carry clear.
Levels: 2. Uses: A,B,C(A),D(A),SB,XM,S8-11

**ARGSTA 0E90C** Read user modes. Pop & test real number for array or complex.
Entry: Number on top of stack.
Exit: DECMODE. A 12 digit number. Carry clear if finite, set if INF. Fatal err if array, complex, NaN.
Levels: 2. Uses: A,B(X),D(A),P,SB,XM,S8-11

**ASCII 0079B** Bit pattern tables. Each char has 10 nibs. 2 nibs per display column. Least significant bit of byte (nib pair) is top row. Read using ASCII + 10 * (chr#). Table only not an entry point. Entry: DON'T!

**ASLW3 0ED21** Shift A left 3 nibs.
**ASLW4 0ED1E** Shift A left 4 nibs.
**ASLW5 0ED1B** Shift A left 5 nibs.
**ASRW3 0ED10** Shift A right 3 nibs.
**ASRW4 0ED0D** Shift A right 4 nibs.
**ASRW5 0ED0A** Shift A right 5 nibs.

**ATNCLR 00510** Clear ATTN flags to inhibit effect of ATTN key. Entry: Doesn't matter.
Exit: Carry clear if ATNFLG was set.
Levels: 0. Uses: A(A),D1

**AVS2DS 09708** Send buffer at AVMEMS to display.
Entry: Buffer of chars at AVMEMS terminated with FF byte. P=0.
Exit: P=0, Carry clear.
Levels: 3. Uses: P,A,B,C,D,D1,R0(10-5),R2,STMTR0

**BF2DSP 01C0E** Send a buffer to display.
Entry: D1 points to chars terminated with an FF byte. Otherwise same as BF2DSP.
**BF2STK 18663** Push a string buffer onto math stack.
Entry: D1=math stack,P=0. S0=0 to go to EXPR when done or S0=2 to return when done.
Exit: P=0,D1 adjusted for string, D0 unchanged.
Levels: 1. Uses: A(A),B(A),C(A),D(A),R1,D0,D1

**CHIRP 0EC5A** Does a short "error" beep. Entry: HEXMODE. Exit: HEXMODE.
Levels: 2. Uses: A,B,C,D,P,D0

**CK"ON" 076AD** Check if ON/ATTN key has been pressed. Needs called after each statement. Use within operations which you may want to be able to interrupt in process. Entry: Any.
Exit: Carry set if ATTN not hit. Carry clear, S14 (no cont) set if ATTN has been hit.
Levels: 0. Uses: A(S),D1,S14

**CLRFRC 0C6FC** Clear fractional part of 15 form in A/B. Entry: A/B=15 digit form.
Exit: DECMODE, A/B=digit with fractional part cleared. Carry set if no FP, clear otherwise.
Levels: 2. Uses: A(A),B,C(A),P

**CMPT 125B2** Return current time in 512ths second since Jan 1,0000 in hex.
Entry: Any.
Exit: HEXMODE, P=0, Carry clear, C and R1 have current time, R0=timer value corresponding to current time.
Levels: 1. Uses: A,B,C,D,P,R0,R1,D0,D1,S0-S11

**COLLAP 091FB** Collapse math stack. Entry: Not important.
Exit: D1=MTHSTK, C(A)=new (MTHSTK), Carry clear.
Levels: 0. Uses: C(A),D1

**CRLFND 0229E** Send CR/LF to display ignoring current delay setting.
Entry: P=0. Exit: P=0.
Levels: 5. Uses: A,B,C,D,D0,D1

**CSLC15 1B427** C shift left circular.

| | |
|---|---|
| **CSLC14 1B424** | **CSLC8 1B42C** |
| **CSLC13 1B421** | **CSLC7 1B42F** |
| **CSLC12 1B41E** | **CSLC6 1B432** |
| **CSLC11 1B41B** | **CSLC5 1B435** |
| **CSLC10 1B418** | **CSLC4 1B438** |
| **CSLC9 1B415** | **CSLC3 1B43B** |

**CSRC15 1B441** C shift right circular.

| | |
|---|---|
| **CSRC14 1B43E** | **CSRC8 1B42C** |
| **CSRC13 1B43B** | **CSRC7 1B415** |
| **CSRC12 1B438** | **CSRC6 1B418** |
| **CSRC11 1B435** | **CSRC5 1B41B** |
| **CSRC10 1B432** | **CSRC4 1B41E** |
| **CSRC9 1B42F** | **CSRC3 1B421** |

**D1C=R3 03047** Restore C(A),D1 from R3. Opposite of R3=D1C. Entry: Any.
Exit: C(A)=R3(A), A(A)=R3(5-9),D1=R3(5-9). Carry not affected.
Levels: 0. Uses: A,C(A),D1

**DAY2JD 13407** Convert #days since Jan 1,0000 to Julian date (year and day of year).
Entry: Day#
Exit: SETDEC, A(W)=year(BCD), B,C=day of year (BCD).
Levels: 1. Uses: A,B,C,D,P

**DAYYMD 13335** Convert day# to year,month,day. Entry: C=day# (in hex).
Exit: A=year (BCD decimal), B=month(BCD decimal), D=day (BCD decimal).
Levels: 1. Uses: A,B,C,D,P

**DCHX=C 1B2D0** **DECHEX 1B2D2** Convert dec integer to hex integer.
Entry for DCHX=C: C(W)= dec integer.
Entry for DECHEX: A(W)= dec integer.
Exit: P=0, HEXMODE, A(A)=hex integer. Carry if good number, clear if overflow; XM= not carry.
Levels: 1. Uses: A,B,C,P,XM

**DCHXF 1B223** Convert 12 digit floating point to 5 digit hex integer.
Entry: A(W)= floating point number.
Exit: P=0, HEXMODE, A(A)=hex integer, Carry set if number is in range and positive. Carry clear if out of range. If carry clear and XM=1 then number is out of range and FFFFF is returned. If carry clear and XM=0 then number is negative and is returned in 2's compliment.
Levels: 1. Uses: A,B,C,P,XM

**DCHXW 0ECDC** Convert full word decimal to hex. Entry: P=0, HEXMODE, C=number.
Exit: A,B,C= hex number, carry clear.
Levels: 0. Uses: A,B,C,P

**DRANGE 1B076** Verify a byte is in range ASCII "0"-"9".
**RANGE 1B07C** Verify a byte is in specified range.
Entry for DRANGE: P=0,A(B)=Byte to check.
Entry for RANGE: P=0, A(B)=byte to check, C(B)=lower boundry, C(3-2)=upper boundry.
Exit: P=0, Carry clear if the byte was in the range.
Levels: 0. Uses: C(A)

**DSPBUF 09723** Send buffer of chrs to display. Versatile routine; send until terminator byte is found or a specified number of chars. Can observe or ignore width.
Entry: D0 points to buffer.
P=0 to send until terminator byte (specified in A(B)) is found.
P=2 to send number of chars as specified in A(A), Ignore WIDTH.
P=4 As with P=0 but observe width. B(A) must be zero.
Exit: P=0, Carry clear.
Levels: 3. Uses: A,B,C,D,D0,D1,R0,R1,R2,STMTR0

**ESCSEQ 023C1** Send escape (ASCII 27 decimal) followed by one other char to display.
Entry: P=0,C(B)=chr to follow the esc.
Exit: P=0
Levels: 4. Uses: A,B,C,D,D0,D1. If an interrupt occurs also uses S<RSTK / RSTK<R - be aware that this uses some RAM.

**EXPR 0F23C** Function return. Assumes D0 and D1 are in order and stack is free of trash.
Entry: D0=pgm counter, D1=stack pointer.
Exit: Back to BASIC.
Levels: 4. Uses: Everything.

**FILEF 09FB0** Find a file in MAIN file chain only.
**FINDF 09F77** Find a file in specified chain.
**FINDF+ 09F63** As with FINDF but checks for bad data.
Entry: File name in A(W). For FINDF also D(S)=F for main & plug-ins or D(S)=0 for main only.
Exit: P=0, If Carry clear then file was found and D1=file start, A(W),B(W)=file name, D(S)=device type. (Current device types: 0=main RAM, 1=IRAM, 2=ROM, 3=EEPROM) D(B)= extender#/port#.
Levels: 2. Uses: A,B,C,D,D1,S6,S8,R1,R2(if outside of main search), R3(if single PORT search).

**FLOAT 1B322** Convert dec integer to 12 digit float point. Entry: A(W)=unsigned integer
Exit: DECMODE,A(W)=floating point number, Carry set.
Levels: 0. Uses: A(W),P

**FNRTN1 0F216** Return to BASIC from a function.
**FNRTN2 0F219 FNRTN3 0F235 FNRTN4 0F238**
Pushes results of a function onto the math stack and go to expression controller after evaluation. These are the easiest ways to exit a function which returns a number.
Entry for FNRTN1: A(A)=Pgm counter, D1=stack ptr, C(W)=number.
Entry for FNRTN2: A(A)=PC, D1=stack ptr,C(W)=number.
Entry for FNRTN3: A(A)=PC, D1 already adjusted for number, C(W)=number.
Entry for FNRTN4: D0=PC, D1 already adjusted for number, C(W)=number.

**HDFLT 1B31B** Convert hex integer <=FFFFF to decimal float point. Entry: A(A)=hex integer.
Exit: DECMODE, P=0, A(W)=floating point number, Carry set.
Levels: 1. Uses: A(W),B(W),C(W),P

**HEXASC 17148** Convert up to 7 hex digits to ASCII. Returns the string reversed.
Entry: P=0,A(W)=hex digits, C(S)=number of nibs to convert (<=7).
Exit: P=0,A(W),B(W)=converted string, C(S)=F, Carry set
Levels: 0. Uses: A(W),B(W),C(S)

**HXDCW 0ECB4** Convert full word hex to decimal.
**HEXDEC 0ECAF** Convert A field hex to decimal.
Entry for HXDCW: C(W)=full hex word.
Entry for HEXDEC: A(A)= 5 hex digits.
Exit: DECMODE, A,B,C=result in decimal, Carry clear.
Levels: 0. Uses: A(W),B(W),C(W)

**HMSSEC 13274** Convert decimal hours,mins,sec to hex seconds since midnight.
Entry: A(W)=hours (BCD integer), B(W)=minutes (BCD), D(W)=seconds (BCD).
Exit: P=0,HEXMODE, A,B,C=seconds since midnight in hex, Carry clear.
Levels: 1. Uses: A,B,C,D,P

**INFR15 0C73D** Integer/Fraction Split 15 digits.
Returns position of decimal point in P. If the exponent is 14 (representing a 15 digit integer) then C(A)=0. If the exponent is >14 (but a finite number) then C(A)=50000.
Entry: -. Exit: A/B=split number.
Levels: 1. Uses: A,B,C,P

**I/OAL+ 1197B** Allocate I/O buffer without leeway check.
**I/OALL 1197D** Allocate I/O buffer with leeway check.
If buffer already exists will adapt it to size specified.
Entry: C(X)=buffer ID#. I/OALL needs P=0.
Exit: If Carry set then buffer allocated and D1 points past buffer header, D0 points 1 nib past buffer header front (at buf ID), B(A)=buf size or amount it was changed in size. C(6-0)=header info, C(0)=# addresses to update, C(1-3)=ID#,C(4-6)=buffer length. If buf already exists and was expanded then A=D1, D(A)= point from which expanded (from bottom). If carry clear then no room, C(4)=err#, P=0.
Levels: 3. Uses: A,B,C,D,D0,D1

**I/OCOL 11979** Collapse I/O buffer to zero length. Leaves header. If buf doesn't exist then 6 nibs of RAM will be used without checking leeway. Use I/ODAL to eliminate header.
Entry: C(X)=buffer ID#
Exit: If Carry clear then zero len buffer created. If Carry set then D1= past header, P=0,D0=past header at ID#

Levels: 2. Uses: A,B,C,D,DO,D1

**I/ODAL 11A41** Deallocate an I/O buffer. Entry: C(X)= buffer ID#.
Exit: Carry set if buffer deallocated. Carry clear if buffer wasn't found.
Levels: 2. Uses: A,B,C,D,DO,D1

**I/OFND 118BA** Find an I/O buffer and set high bit on buffer ID# so that it will be deallocated at next configuration.
**IOFNDO 118C1** Find an I/O buffer.
Entry: C(X)=Buffer ID#.
Exit: If Carry set then C(X)=buffer ID#,D1 points past buffer header, A(A)=buffer length field, C(S)=number of addresses within buffer to update. If not carry then buf was not found.
Levels: 0. Uses: A,C(A),D1

**IDIVA OEC6E** Integer divide in hex or decimal mode. Zeros nibs 5-15 of A&C then goes to IDIV.
**IDIV OEC7B** Hex or decimal full word integer divide. Caution: if denominator=0 then this routine will loop indefinitely.
Entry: hex or dec mode, A=dividend, C=divisor.
Exit: mode not changed, P=15, quotent in A, remainder in B and C, Carry clear.
Levels: 0. Uses: A,B,C,P

**IVAERR 02920** Report an "Invalid Arg" Error. Doesn't return.

**KEY$ 1ACA8** The KEY$ function. Pops the last key from key buffer. Can be used after SLEEP to implement KEYWAIT$ function. Entry: P=0.
Exit: Pops the key and places it on the stack then returns to BASIC. Does not return, be sure DO,D1 are accurate.
Levels: 3. Uses: A,B,C,D(A),R0,R1,R2,S0-S2,DO,D1

**KEYCOD 1FD22** Keycode map. NOT AN ENTRY POINT. Maps the keycode to the definition. Example: ENDLINE is #38 which maps to 0D hex (ASCII 13). Entry: DON'T!

**MEMBER 1B098** Check if a byte is a member of a set of up to 8 bytes.
Entry: A(B)=byte to be compared. C(P-0)=set of bytes in the set starting at nib 0 of C and extending to P pointer.
Exit: P=0, Carry set if byte was found and in set.
Levels: 0. Uses: C(WP),P

**MEMCKL 012A5** Check available mem with or w/o leeway. Useful before creating temporary buf.
Entry: C(A)=amount of mem to check for. P=0 if leeway to be added to amount to be checked.
Exit: P=0. If carry clear then enough memory, B(A)=amount to chec, A(A)=AVMEMS, D1=(AVMEMS), C(A)=available memory minus requested amount If Carry set then not enough memory, B(A)=amount to check for, C(A)=eMEM.
Levels: 0. Uses: A(A),B(A),C(A),D1

**MPOP1N 0BD8D** Pop one number from stack, give Signaled Op message if necessary.
Entry: D1=stack pointer.
Exit: DECMODE, A(W)=number. Carry set if number is complex, imaginary part in R0.
Levels: 3. Uses: A,B,C,D,R3,S8-11

**MPY OECBB** Multiply hex*hex or hex*dec.
Entry: If hex*hex then SETHEX, arguments in A,C. If hex*dec then SETDEC, hex argument in C, dec argument in A.
Exit: P unchanged, Mode not changed, Carry clear. Result in A,B,C. If hex*hex then result is hex. If hex*dec then result is dec.
Levels: 0. Uses: A,B,C

**POP1N 0BD1C** Pop one real number from math stack. Will error out if non-numeric data. Note: does not move D1 past the number on the stack.
Entry: D1=math stack pointer.
Exit: P=0, DECMODE. If Carry clear then result is real and in A(W). If Carry set then number is complex, real part in A(W), imaginary part in R0.
Levels: 0. Uses: A,B(0). If Carry then uses R0

**POP1N+ 0BD91** Pop 1 number from stack, check for NaN. Signal if appropriate.
**POP1R 0E8FD** Pop 1 number from stack, check for NaN.
Entry: D1=top of math stack.
Exit: DECMODE, A=12 digit form of number, Carry clear. Doesn't return if bad data.
Levels: 1. Uses: A,B(X),P

**POP1S 0BD38** Pop 1 string from math stack. Exits with D1 pointing past header at end of string

(low mem). Errors out if bad data.
Entry: SETHEX, D1 points at string header.
Exit: P=0, A(A)=string length in nibs, D1 points at last char (low memory end) in string.
Levels: 0. Uses: A(W),D1,P

**POPBUF 010EE** Pop last key from key buf. Interrupts disabled during this routine. Entry: Any
Exit: Carry set if buffer was empty. If Carry clear then key is in B(A).
Levels: 0. Uses: B(A),C(W),D0

**POPMTH 1B3DB** Skip past first item on math stack. Useful for counting items or skipping stuff. Works with strings and complex numbers.
Entry: P=0,D1=top of math stack.
Exit: P- ,D1 moved past item.
Levels: 0. Uses: A,C,D1

**PUTRES 18115** Put a number in the RES register. Entry: D1 points to number.
Exit: HEXMODE, P=0, D1 unchanged. Carry clear if real, set if complex number.
Levels: 1. Uses: A(W),B(0),D0,R0 (If number is complex).

**R3=D10 03526** Save D0,D1 in R3. Entry:
Exit: R3(A)=D0,R3(9-5)=D1,A(A)=C(A), Carry unchanged.
Levels: 0. Uses: A,C(A),R3

**RDTEXT 17489** Read a line from a TEXT file to output buffer. File must have a FIB# (using ASSIGN# in BASIC).
Entry: R4(15-14)=file FIB#, OUTBS= start of output buffer, AVMEMSD=(OUTBS).
Exit: P=0, AVMEMS= after last nib read. If Carry then C(3-0)=error code. If Carry clear then S7 set if file positioned at EOF, C(A)=length of line including header, or zero if no EOF marker at end of file. Line length header or EOF marker not copied to output buffer.
Levels: 5+1 on RSTKBF. Uses: A,B,C,D,D0,D1,R0,R1,R2,R3,P,S11-S9,S7,S6,S4-S0

**REV$ 1B38E** Reverse characters in a string on math stack.
Entry: HEXMODE, D1=points at string header.
Exit: D1 unchanged, C(A),D(A)=D0. Will error out if item on stack doesn't begin with proper string header.
Levels: 1. Uses: A,B,C,D,P

**REVPOP 0BD31** Does a REV$ then POP1S. See POP1S for details.

**RNDAHX 136CB** Pop, test, round a decimal number from stack. Entry: D1=top of math stack.
Exit: HEXMODE, P=0, A(A)=hex integer, XM=0. If Carry set then it is non-negative (incl -0). If Carry clear then negative. Will error out if array, complex, or NaN.
Levels: 3. Uses: A,B(S),B(A),C(A),D(A),P,SB,XM

**RPLLIN 013F7** Replace a line in a file in memory. Used to insert, delete or replace.
Entry: P=0, OUTBS=start of replacement line, end of line is at AVMEMS, A(A)=address of last nib+1 of file, R3(A)=length of old line in nibs (use zero to insert).
Exit: P=0, R3(A)=offset to move (destination end minus source end). A(A)=end of replaced line in file plus one, B(A)=length of replacement line in nibs, C(A)=(OUTBS). If Carry clear then output buffer is collapsed. If Carry set then unsuccessful and C(3-0)=err#.
Levels: 3. Uses: A,B,C,D,D0,D1,R1,R2,R3

**SECHMS 13252** Convert hex seconds time of day to decimal hours, minutes, seconds.
Entry: C(W)=time of day in hex.
Exit: HEXMODE, P=15, A(W)=hours (BCD integer), B(W),C(W)=minutes(BCD), D(W)=seconds (BCD). Carry clear.
Levels: 1. Uses: A,B,C,D,P

**SETALR 12917** Set alarm relative to current time. Details same as SETALM.
**SETALM 1290D** Set absolute alarm time.
Entry: A(11-0)=number of 512ths second since Jan 1,0000, C(0)=alarm number (0-5).
Exit: P=0, R1=current time in 512ths since year zero, R0=timer value corresponding to current time. Carry clear.
Levels: 2. Uses: A,B,C,D,P,D0,D1,S0-S11,R0,R1.

**SFLAG? 1364C** Test a system flag.
Entry: HEXMODE, P-0, C(B)=flag number in hex (FF= flag-1).
Exit: HEXMODE, P=0, D(A)=D0. Carry clear if flag clear, set if flag set.
Levels: 1. Uses: A(A),C,D(A)

**SFLAGS 135FA** Set a system flag and update annunciator. Details same as SFLAGC.

**SFLAGT 13608** Toggle a system flag and update annunciator. Details same as SFLAGC.
**SFLAGC 13601** Clear a system flag and update the annunciators.
Entry: HEXMODE, P=0, C(B)=flag number in hex (FF=flag -1).
Exit: HEXMODE, P=0, D(A)=D0, flag cleared, Carry clear.
Levels: 2. Uses: A(A),B(A),C,D(A),P, plus RAM at ANNAD1-4, SYSFLG.

**SLEEP 006C2** Scan the keyboard, go to light sleep if key buffer empty. Wakes up when key pressed. Leaves the key in the buffer. Since it debounces it will not recognize a key that was down when it was entered (won't repeat). Entry:
Exit: P=0. Carry clear if keys in buffer. Carry set if no keys in buffer.
Levels: 1. Uses: A,B,C,D0.

**SPLITA 0C6BF** Split 12-form in A into A/B. If carry then we have NaN or INF.
NaN= A(A)=00F01, B(XS)=F. Inf= A(A)=00F00, B(XS)=F.
Entry: A=number to split.
Exit: A/B= split number.
Levels: 0. Uses: A,B

**STR$00 1815C** Convert a number on the stack into a string back on the stack using current display settings.
Entry: D1=top of math stack. S0 set if return when done, else jums to EXPR. S1 set if leading and trailing blanks are to be added.
Exit: P=0, D1 points to string on stack, returns if S0 was set. Errors to MEMERR if memory overflows.
Levels: 2. Uses: A,B,C,D(A),R0,R1,R2,D1,S0,S1.

**STRTST 1B1C7** Test two strings for equality. Returns position within strings where equality test failed.
Entry: D0 and D1 at high memory end of the two strings. C(A)= number of nibs to compare.
Exit: B(A)=(block comparison length -1)/16, P=(block comparison length) mod 16. D0,D1 set at first words not equal. If comparison length= zero then Carry clear, XM=1. If strings equal then Carry clear, XM=0. If strings not equal Carry set, XM=0.
Levels: 0. Uses: A,B(A),C,P,D0,D1.

**STUFF 1B0B2** Fill memory with 16-nibble pattern of "stuff".
**WIPOUT 1B0AF** Fill memory with "0".
Entry: HEXMODE, D1=start of area to be filled, C(A)=length in nibs of area to stuff. For STUFF entry A(W)=pattern to stuff, WIPOUT presets A(W) to zero.
Exit: P=0, D1=past last nib stuffed, Carry clear.
Levels: 0. Uses: P,C,D1. WIPOUT also uses A.

**TODT 13229** Convert time in hex seconds since Jan 1,0000 to time of day, day#.
Entry: HEXMODE, C(W)=hex seconds.
Exit: HEXMODE, P=15, A= days since zero in hex, B,C= of seconds since midnight. Carry set.
Levels: 0. Uses: A,B,C,P

**WFTMDT 085DD** Zero flags(including nib 2), write time,date to file header.
**WFTMD- 085D6** As WFTMDT but does not zero nib 2 (nib 2 is copy code).
Entry: D0=start of file.
Exit: P=0, R1=file start, D0=time field header.
Levels: 3. Uses: A,B,C,D,P,D0,D1,R0,R1,S0-S7, plus 32 nibs at SCRTC.

**YMDDAY 13304** Convert year, month, day to absolute day number.
Entry: A=year (BCD), B=month (BCD), D=day (BCD).
Exit: HEXMODE, P=0, A,B,C= number of days since Jan 1,0000 in hex.
Levels: 1. Uses: A,B,C,D,P

**YMDHMS 130DB** Return current time and date. Entry: Doesn't matter
Exit: through YMDH01
**YMDH01 130E5** Convert a time to 0000YYMMDDHHMMSS.
Entry: C(W)=time in seconds since Jan 1,0000.
Exit: HEXMODE, C=0000YYMMDDHHMMSS, A(B)=HH, B(B)=MM, D(B)=SS, Carry clear.
Levels: 2. Uses: A,B,C,D,P,D0,D1,R0,R1,S0-S11.
s: