The BASIC HP-71

VERSION 2.1

Up and Running in
CALC Mode, BASIC and Assembly Language

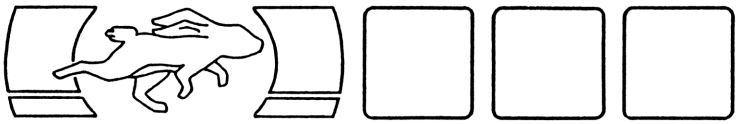
by Richard E. Harvey

The BASIC HP-71

Up and Running in
CALC Mode, BASIC and Assembly Language

Version: 3.0

by: Richard E. Harvey



Contents

Introduction 1
How to Use This book 2
Why Program? 3
Limits 4
Cautions 4
2 The HP–71 5
Central Processor (CPU) 5
Clock Speed 5
Hexadecimal Numbers 5
Memory 6
Environments 7
Sub-Programs 9
CALL Cautions 10
3 Command Performance 11
4 Getting Started 21
The Parser at Work 22
BASIC Keyboard Math 23
Mathematical Precedence 24
Parentheses 25
Strings 26
Calculator Variables 28
ZEN and Variables 32
5 CALC Mode 33
6 Basic BASIC 39
HP BASIC 41
7 HP–71 BASIC Programming 47
Sub-Programs 48
Interpreted BASIC 50
Strings 52
Control Codes 53
8 HP–71 Files 55
The File Chain 55
BASIC Files 57
BIN Files 58
FORTH Files 58
LEX Files 58
DATA Files 59
TEXT Files 61
KEY Files 64
SDATA Files 64
9 HP–71 Data Files 66
Creating the Data File 66
Opening the Data File 67
The File Pointer 68
Storing Data 68
Recalling Data 69
Closing the Data File 69
10 BASIC Programming Hints 71
11 PEEK$s & POKEs 84
System RAM 85
Strings in SDATA Files 90
12 Convert From other BASICs 93
HP-75 BASIC 98
13 Assembly Language 101
Source Files 101
The HP-71 102
CPU Registers 102
Return Stack (RSTK) 103
Assembling the LEX File 105
The LEX File 107
Parsing Functions 109
Types of Parameters 110
Decompiling 111
Entry Conditions 111
The Math Stack 111
System Entry Points 113
Crashes 117
Instruction Set 117
The RETURN Stack 118
Strings From Math Stack 120
Numbers From Math Stack 120
Temporary Scratch 120
Exiting the Function 121
Assembler Bugs 123
14 Non–obfuscating Programs 125
Modular Programming 126
User Friendly Programming 128
Menus and Command Lines 128
15 Accessories 130
Data Storage 130
How Much RAM Can I Add? 130
HP-IL Mass Storage 132
Printers 135
Display Devices 135
Other HP-IL Devices 135
16 Communicating 137
Communicating with RS-232C 142
HP-IL to PC Interface Card 145
Tables 148
System RAM 148
Memory Map 149
Assembler Instruction Set 150
Minimum LEX File Requirements 152
Fields in Working Registers 153
System Entry Points 154
System Flags 160
Display Escape Codes 160
Dec/Hex/Oct/Bin/ASCII Table 161
Keyboard Map 164

The BASIC HP-71

We can thank Hewlett-Packard for the HP-71, but also the user community, for demonstrating to HP that there is a need for this powerful, compact tool, and for following though in the years since its introduction. This book is dedicated to those users.

Introduction

In the early 1970's we had 15 pound "portable" calculators like the HP-46. In 1972 we saw the introduction of 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.

Personal computers were used for playing Star Wars long before word processors or spreadsheets were even contemplated. It was a well kept secret for years that computers are fun. Let's get some work done, but, let's make it an enjoyable experience. That's the attitude we'll share in this book.

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'll also describe the internal design of the 71 and Assembly Language programming, supporting these discussions with several tables and charts. Much of the material covered in volumes one and two of the HP-71 Internal Design Specification is paraphrased, making the purchase of those books (at about $100) unnecessary except for the most devoted Assembly language user.

Richard E. Harvey
Box 5695
Glendale, Arizona 85312 USA

How to Use This book 1

Much of this book is reference 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 reference material easier to find later without having to wade through several thousand words of flowery prose.

Most subjects are given a cursory introduction, followed by greater detail. We've tried to include most material on a subject in the same section to minimize darting back and forth. 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, 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 its drifting off into a foreign language. The glossary in *The HP-71 Reference Manual* will aid in the translation to English, or you could reserve reading those sections in this book until a later date.

Examples are indented from the left margin, and are explained in the immediately preceding text. Since an example may be at the end of a discussion, text which follows will not necessarily have anything to do with it. Numbers are printed in STD display format. Remarks in examples are preceded by and exclamation mark, even when we're illustrating Calc Mode; this is the standard HP BASIC way to indicate remarks, so there's no time like the beginning to become comfortable with it. Boxes with a simulated flashing cursor (actually, a "▒" character) simulate the LCD display on the 71, though they occasionally overflow quite a bit on the right. Nested boxes along the left side of the page demonstrate a series of steps to follow. When relevant, a second box on the right shows likely results of the experiment (usually without the flashing cursor).

Listings of Assembly Language follow the standard Assembly indentation format: Labels are left flush and code is indented eight spaces. Programs supporting a topic are listed with the topic. Other sample programs and subroutines are scattered throughout the book, and many are not mentioned in the text at all; we'll leave it to you to make these discoveries.

The 71 understands commands written in either upper or lowercase. In this book, we'll usually show commands in UPPERCASE to help distinguish them from text. Either of the following is acceptable.

```
beep                                    BEEP
```

We'll also often say "Now, create a file called TEST" though it isn't always necessary, and it really doesn't matter what you call the file. If you want to follow the example and you already have a file

called TEST, feel free to use another file name. Computer technical journals, for example, often call things FOOBAR; these are personal computers, after all.

While we'll cover quite a bit of information quickly, there are no prizes for speed reading. Take your time and explore your 71, you may be surprised about how much you'll find. If you are new to your 71, Chapter 3, "Command Performance" should help you get started without having to wade too deeply through the *HP-71 Reference Manual*.

Version 3?

There's little you can't do with a 71. To help prove that point, the first version of this book was printed with one. Files were either written on the 71 connected to a terminal, or transferred to the 71 over RS-232 and stored on the 9114 Disc Drive (on a single disc). All file management, text formatting, and printing tasks were handled by an HP-71B with 17.5K of RAM using programs written by the author. Because of the long battery life of the HP82161A Cassette Drive, most printing was done from Cassette based files. While there's much to be said for the right tool for the job, who can define what is the right tool in all cases? Most often, the job is made to fit available tools. Our 71 is a more portable, personal, and versatile computer than the desk-bound; what can it do? Whatever you want! While version 3.0 was edited with a desk-top computer, the 71 was always at its side, usually connected by HP-IL cable. Changes in this latest revision include more consistent organization, clearer illustrations, and an index.

Handheld computing has changed considerably in the nearly two years since this book was first printed. The HP-75 has been discontinued, leaving the 71, and the singular HP-94, alone in data collection and system monitoring. New HP calculators, including the HP-19B business calculator and HP-28S scientific calculator, share some of the technology, though not the expandability or BASIC language of the 71. And the amazing 71 itself has undergone changes to provide more immunity to static energy. Perhaps the most important development is the variety of new memory modules, opening up the 71 to tasks previously requiring many pounds of hardware.

Why Program?

Plug in a ROM, fire it up, and there you have it: Instant solutions. Many people buy a 71 with that single objective 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 skill to make better use of those we already have. This personal computer becomes more personal when we learn to use it our way, not somebody else's; we can adapt the machine to our needs, not the other way around.

A principal 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. 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 (along with its associated discoveries) is fun! Working through a program provides as much pleasure as having it done. And, a program is never truly finished.

Limits

The material presented herein is informational only and not warranted for any application. While every effort has been made to assure the accuracy of the information, no liability is assumed. Determination of suitability and implementation are the users responsibility. These materials are the property of the author. By receipt of these materials the user agrees to abide by applicable copyright laws.

Cautions

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. No operations described in this book will cause a crash (freeze the LCD, or keyboard or display *that* message) if the directions are followed exactly. The operations least forgiving of mishaps 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. Don't assume that files in a PORT are safe.

A controlled crash (INIT-3) is often used to purge unwanted files and restore the machine to start-up conditions. INIT-1 will usually free the 71 from any "stuck" situation. The 71 was designed to perform this operation, there is no damage. You can do an INIT-1 by pressing the `ON` key and **/** key simultaneously, then pressing `ENDLINE` at the prompt. For INIT-3, press the 3 key before pressing `ENDLINE`.

A very rare crash won't respond to INIT-3. There are two courses of action: Pull out modules and batteries (and un-plug the AC cord), 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 completely 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. 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

WorkBook71 is a software package for the HP-71 including Virtual Memory Spreadsheet, File Manager, Data Format Converter, Full Screen Text Editor and Text Formatter. The ROM version includes an RPN calculator and several other utilities. Please contact the author or dealer from whom you purchased this book for information.

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 the HP-35 in the early 70's, and the newer HP-19B and HP-28S calculators use a processor similar to the 71. The 75, on the other hand, is an 8-bit portable computer which evolved from the desk-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 under development; the HP-75 was the Kangaroo, and the HP82161A Cassette Drive was called Filbert (really!), and the 71 is Titan. In addition, the microprocessor (CPU) which controls Titan is called Saturn, and the internal bus structure is Capricorn.

Central Processor (CPU)

The 71 has a custom 4-bit CPU. Four bits 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, handling 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 or four 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, interpreted BASIC running on an IBM PC.

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 HP engineers rationalized that great processor speed would not be necessary. Most 71's run at a little over 600kz, though this speed varies with the number of devices attached (combined city and highway, your mileage may differ). This is the speed at which the CPU runs, and does not affect the speed of the real time clock or tone of the beeper. When doing speed comparisons with other computers, this compromise will be immediately apparent. In purely 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 re-computed 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:

```
                                    ! Find the current CPU speed.
P$=PEEK$("2F977",5)                 !   Then display it.
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 location in memory expressed in hexadecimal (base 16). Numbers are generally handled in standard decimal (base 10) format, so the 71 keeps hexadecimal

(usually called just "hex") in strings. The keyword HTD converts the results of the expression to standard decimal (base 10) format, and DTH$ converts decimal numbers to hex strings. The hexadecimal number system 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 internally. 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). So, hex is a convenient way to deal with binary numbers.

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

In hex, "9" plus "1" is "A"; values between 10 and 15 are represented as a single digit of "A" through "F". The 71 represents values larger than 15 (or "F" in hex) in groups of nibs called words. The most common word size is two nibs, or one byte, though word size differs with other machines.

Memory

To keep things sorted out, the 71 assigns an individual number, called an address, to each nib of RAM and ROM. Memory location "2F977", named CSPEED, is where the 71 stores the current clock speed setting. Notice that we specified five hex digits for the location; since the HP-71 has a 4-bit CPU, all locations are nibs or 1/2 bytes. A five nib word size, therefore, addresses "FFFFF" or 1048576 nibs (including nib "00000") of memory. This translates to 524,288 bytes, or simply 512K. So, the maximum amount of memory, RAM or ROM, is 512K bytes.

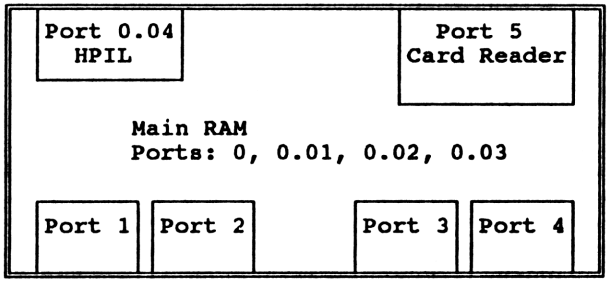
A 71 with ostensibly 17.5K bytes of RAM has only about 16K available, even when first turned on, nothing in it. This is because the Operating System reserves part of RAM for pointers and other necessary system information, like CSPEED. This seemingly inflated rating of memory is actually less than most portable computers which often usurp 4K or more just to be able to power-up. And desktop computers can squander as much as 80K or more before loading any programs.

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 of the beginning of a machine language

subroutine. While these addresses won't change, the code following may; for this reason you should only use documented entry points. Over 100 entry points are listed 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 the 71 delivers it 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 like a ROM. The files are still accessible; you can edit and copy them like files in MAIN RAM. Files in PORT RAM will usually not be lost in the event of a crash (though, nothing is truly secure). There are also provisions for PORT Extenders to address beyond the standard 5 PORTs, though no PORT Extender is currently available.



0 This is main RAM as well as the optional HP-IL ROM. Memory in MAIN RAM can be partitioned in 4K blocks from .0 (or, simply 0) through .03, and is followed by the HP-IL Module. If any RAM is added internally, it becomes an extension to PORT 0, and the HPIL Module moves up one (for example: 0.05).

1-4 The front module PORTs.

5 The Card Reader port, though it's just as likely to have a large RAM module addressed there.

Environments

The HP-71 is unique in the world of personal computers. Its file structure, operating system and advanced BASIC are brilliant. But environments are 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 interaction of files and environments in the 71, and what that has to do with us.

The global environment includes the file system (programs, key assignments and such) and 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 accessible 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
```

```
! Assign the value -5 to
!   the variable X.
```

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

```
EDIT TEST
```

```
! Create a BASIC program
!   file.
```

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

```
! Write a line to the file.
```

Since it's the current edit file, we can run it by pressing the RUN key. When the program runs, it displays 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 pressing 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 sub-program is CALLed, a temporary program environment for its own variables is created, and the calculator variables are ignored. So naturally, X had no value in that temporary program environment. When the CALLed program ends, its environment is eliminated and the calculator variables are again active. In fact, now that the CALLed program has ended, its environment is gone and we can confirm that X still has the value -5 in the main environment.

```
X
```

```
-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 when the program ends. Now that we're done, you can erase the file to reclaim memory:

```
PURGE TEST
```

```
! Delete the BASIC program
!   file from memory.
```

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	What's There
High Memory	Plug-In ROMs & RAM.
	Calculator Variables.
Unused Memory	
	Programs & other files.
	Global Environment.
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 When a program is CALLed	
High Memory	Plug-In ROM & RAM Modules.
	Calculator Variables (on hold).
	Active Temporary Program Environment.
Unused Memory	
	Programs & other files.
	Global Environment.
Low Memory	Operating System.

When the program is CALLed, 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; it couldn't find the calculator variables with a Bloodhound. 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, neither an error in the sub-program nor RETURN will return us to the calling environment; it requires END or END SUB.

Sub-Programs

Many programs begins with a SUB statement to declare the entire program to be a sub-program. 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 sub-programs.

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, its own environment 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 under 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. To get a feel for how it runs, press ATTN to suspend the program while it runs, then single-step through it by repeatedly pressing the f-SST key to watch it loop.

```
10 X=0  ! A recursive program.
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, 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.

A second problem with suspending programs is open data files. Suspend a program which has opened a file or two, then CALL that program again, and those files will still be open. The 71, not liking files opened more than once at a time (especially with the same file number), will cause an error in the program. You could end up with a nasty mess when the program has an error trap that keeps asking you for a different file name and keeps trying to give it the same channel number.

Command Performance 3

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 Reference Manual. We'll introduce keywords used in programs later when we talk more about programming.

ASSIGN IO *Assign HP-IL Devices.*

When the 71 powers-up, it assigns devices found on the loop. Though you may use ASSIGN IO to assign device codes if you want, it isn't necessary. The problem is that the device code is associated with the address on the loop, not the device, and you will have to use ASSIGN IO again each time you re-configure the loop. It's much easier to address a device by its device name such as :TAPE or :DISPLAY or the location on the loop. For these reasons, most 71 users don't assign device codes. The HP-IL Interface Manual notes syntax and usage of ASSIGN IO. If you've experimented with device codes and wish to eliminate them enter:

```
ASSIGN IO *  ! Cancel HPIL device code assignments.
```

CALL, CONT, RUN *Running Programs.*

When you RUN a program (by pressing the RUN key or entering RUN filename), that program becomes the current edit file when the program ends. If you often use a single program, you might RUN it once, then it will only be necessary to press RUN the next time you need the program because it will become the current edit file. On the other hand, after a CALL the current edit file does *not* change. You can use these two features together: CALL for the utilities and programs in ROM, RUN for the workhorse program you often use. RUNning and CALLing programs is covered in greater detail Chapter 2, "The HP-71."

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

CAT, CAT ALL *View Catalog Listings.*

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  ! Display the catalog entry of the current file.
CAT AFILE  ! Catalog of file named "AFILE".
CAT :PORT(2)  ! Catalog of all files in :PORT(2).
CAT AFILE:PORT(0)  ! Cat of the file "AFILE" in :PORT(0).
CAT ALL  ! Catalog of all files in RAM and :PORTs.
CAT :TAPE  ! Cat of the entire medium; similar to CAT ALL
CAT AFILE:TAPE  ! Cat of file "AFILE" on Disc or Cassette.
CAT CARD  ! Catalog of a magnetic card track.
```

COPY *File Utility.*

The most useful file command. It copies files between RAM, Magnetic Card, :PORTs and Mass storage. How it works depends on where you are copying the file to, and if a file by that name already exists at that destination. If you are copying to RAM from a mass storage device, you cannot overwrite a file already in RAM by that name. However, copy a file from RAM to Disc and the copy will be made, possibly overwriting a file already there with that name. The idea is to make it easier to make a backup copy of a new version of a file without first purging the old one.

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

CLAIM PORT, FREE PORT, SHOW PORT

The most readily apparent PORTs are those four along the front edge of the computer, and the Card Reader. When we plug RAM modules into the 71, they become part of main RAM, and files there are part of the main file chain. To organize things, we can partition memory into separate PORTs. In addition to the front PORTs (numbered 1-4 from left to right), we can separate main RAM into PORTs of 4K each, called PORT(0) through 0.04. PORT(0) is the first 4K, PORT(0.01) is the second, and so forth; the HPIL Module is the last part of PORT(0). Commands to reserve and reclaim PORTed RAM are not programmable; if you are writing a program requiring PORTed RAM, be sure the memory is set aside before the program begins. BASIC keywords for moving files between PORTs are fairly straightforward, though the names may be difficult to remember:

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

DEF KEY *Customize the Keyboard.*

Press f-USER and the "USER" annunciator in the LCD lights (or turns off) and little else happens. Unless you want to count letting you create your own custom 71. How about running a program by

pressing a single key, a keyboard full of custom typing aids, or a completely redesigned DVORAK-style keyboard?

The three types of key assignments (aside from those the 71 was born with) are typing aids, immediate execution, and direct execution. The character following the key assignment string tells the 71 which type of assignment:

;	Semicolon for a typing aid.
:	Colon for direct execution. These commands are directly executed without entering them in the display or onto the Command stack. These assignments do their business without entering anything in the display.
""	No punctuation (or a space). These are called Immediate Execute key assignments. Like a typing aid, but also automatically "presses" ENDLINE at the end of the string. The command is entered onto the Command stack.

We won't even try to list a key map here; if you'd like to see one, you'll find one at the back of this book. While you could use the keymap codes to define keys, why would anyone want to try to remember those codes? Instead, we usually leave the bottom row editing keys as they are, and assign to the f- and g- shifted keys on the top three rows. Since the g- shifted keys are already assigned for useful typing aids (such as the lowercase alphabet), the f- shifted keys are the most convenient to re-assign. For example, to assign the character "~" (which is not normally on the keyboard) to the f- shifted Q key as a typing aid:

```
DEF KEY "fQ",CHR$(126);  ! Assign ~ to the f- shifted Q key.
```

These examples show the "f" in lowercase for readability, though it's not really necessary. Now, restore the key to its original definition:

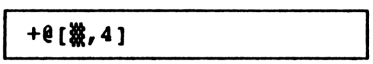
```
DEF KEY "fQ"  ! Removes key assignment from f- Q key.
```

If you have the KEYWAIT$ (*not* WTKEY$) keyword, you can simplify key assignments a bit. Instead of the assignment key, specify KEYWAIT$ and the 71 will wait for you to press any key and assign the string to that key. You can tell if you have KEYWAIT$ by trying it: Type the command and press ENDLINE; if the computer seems to "hang" until you press the next key, then displays that key's code, it's there.

```
DEF KEY KEYWAIT$,CHR$(96);  ! Assign ` to the next key pressed.
```

Key definitions may be any number of characters which will fit on a single line. You can eliminate the spaces between commands if your string is quite long. This next example is also a typing aid, but it includes four escape codes to move the cursor left over the "1" in "+@[1,4]". Since it's a typing aid, we could have used CHR$(8), the backspace character, instead of CHR$(27)&"D", because the 71 interprets them the same. These typing aids are often called "boilerplate text;" that is, they contain standard information useful for things like entering data in a spreadsheet.

```
DEF KEY "fS","+@[1,4]"&CHR$(27)&"D"&CHR$(27)&"D"&CHR$(27)&"D"&CHR$(27)&"D";
```



Direct Execute keys do exactly that; press the key and whatever is assigned to it is performed without displaying anything, and without altering the contents of the Command Stack. Many programs ignore Direct Execute key assignments, though others which use INPUT will accept them and the automatic ENDLINE they "press." This key assignment calls a program called RPN:

```
DEF KEY "f3","CALL RPN":  ! Directly execute a program.
```

If you often use complex formulas which really aren't complex enough to require programs, you can assign them to keys instead of writing those programs. The trick is the keyword DISP$, which reads whatever you've typed onto the command line. DISP$ returns a string, so if your formula requires a number, you can use VAL to interpret the string as a mathematical expression and return the number to the function. These examples do decimal to hex and hex to decimal conversions. Simply type a number (or a mathematical expression), then press the key; the number you typed is passed to the function via DISP$, and the answer is displayed. The stack is unchanged.

```
DEF KEY "fH","DTH$(VAL(DISP$))":  ! Reads display as a number.
DEF KEY "fD","HTD(DISP$):  ! Enter a hex value and press the key.
```

The third type of key assignment, Immediate Execute, appends the string to whatever you've already typed in the display, then surreptitiously presses the ENDLINE key for you. If you haven't typed anything, they operate like Direct Execute keys, except that the string is added to the Command Stack. Less versatile and useful than the other two types, they seem almost to be a throwback from an earlier computer.

Once you are comfortable with them, you'll probably find yourself often using temporary key assignments instead of writing programs. A key assignment might even be a little program in itself, one long line with FOR... NEXT loops and such. You can even assign regular keys to special tasks and use f-USER as g-1USER as extended shift keys. Though watch for programs which set user mode (like the HP Text Editor), because you're stuck in user mode, with whatever you've assigned to the un-shifted keys in the way, while the programs run. g-1USER is an interesting key. If you are in User mode, pressing it will cause the 71 to ignore the key assignment of next keystroke and will instead return the regular key assignment. However, if you are *not* in User mode, pressing g-1USER will return whatever is assigned to the next keystroke. It's a clever key which can get you out of situations like accidentally re-assigning the ENDLINE key.

So, how do you recall what's assigned to a key without pressing it, and how do you preserve these carefully hewn jems for posterity? If you want to look at a key assignment without changing it, press f-VIEW followed by a key which has an assignment, and the definition will remain in the display for as long as you hold the key down. FETCH KEY brings the key back for editing, and KEYDEF$ returns the definition as a string.

```
FETCH KEY "f1"  ! Recall the string for editing.
K$=KEYDEF$("f1")  ! Recall the key as a string to K$.
```

Key assignments are maintained in a file called "keys" of the file type called "KEY" (no confusion there?). When you create a key assignment, you are writing to this file. It's a real file, so you can treat it like one. You can have multiple key files, though only one at a time can be called "keys".

```
COPY keys TO MYKEYS  ! Creates a spare copy of the keys.
COPY MYKEYS TO keys  ! Copy, not rename, a file to the key file.
MERGE MYKEYS  ! Add the keys to the current definitions.
PURGE KEYS  ! Eliminates key assignments.
```

DELAY *Display Delay Rate.*

The 71 pauses between displaying a series of lines to make sure we have time to read them on the LCD. The amount of time for the delay, and the speed at which lines which are too long to fit in the LCD will scroll, are set with the DELAY keyword. A setting of 8 (eight) or greater is interpreted as "INF" (infinity), the 71 will display the current line until you press a key. A delay of zero causes 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 keys let 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. Many people prefer zero delay with INF scroll rate.

```
DELAY 0  ! Line delay rate to zero.
DELAY .5,.125  ! Line delay 1/2 sec, scroll 1/8.
DELAY 0,INF  ! Zero delay, don/t scroll.
```

DESTROY *Erase Unwanted Variables.*

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 these variables 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. You can use DESTROY with a variable name, even if the variable doesn't exist, without causing an error. It isn't necessary to include the parentheses when you specify an array to be trashed.

```
DESTROY ALL  ! Frees memory used by calculator vars.
DESTROY X,A$  ! Frees memory used by vars X and A$.
```

EDIT, PURGE *File Utilities.*

These two keywords 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. There is no undo; when you've PURGEd a file, it would be difficult to POKE it back into shape.

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 BASIC files in MAIN RAM or :PORTs, not on mass storage.

The BASIC workfile is the current file whenever you enter EDIT without specifying a file name. If you enter EDIT workfile, a regular file called WORKFILE will be created. The workfile catalog entry is, like the active KEYS file, listed in lowercase. If this file is copied to, for instance, Disc without specifying a different file name, 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's suggested that this file be made the current file most often to avoid the chance that an important BASIC program could accidentally be changed.

```
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.
PURGE ALL  ! Careful with this one: it deletes everything not secured!
```

END, END ALL Ends Suspended Programs.

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. The main reasons for doing this are to close any possibly open data files and to regain the memory used by variables in these programs. If a SUB program has been suspended, your calculator variables won't be available until the program ends. You can tell that you need to enter END ALL if you notice the "SUSP" annunciator lit and you aren't planning on continuing the program. Another use for END is to save a few keystrokes closing data files opened from the keyboard (as opposed to in a program).

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

FETCH Finds a Line Number or Label.

Moving around the current BASIC file can be speeded by the non-programmable FETCH command. Specify a line number or label, and the computer will jump to that line, displaying it, ready for editing. 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 without a line number or label will bring back the last you edited or, if you've just moved to the file, will bring up the first line in the file.

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

INITIALIZE Format a Disc.

If you've ever seen the "ERR:Invalid Medium" message, you know that Cassettes and Discs must be formatted before you first use them. This means that the media must have a standard directory and data format before the computer can record files on it. The INITIALIZE keyword formats the media to the HP-71 format and erases all data previously stored on the media. Unlike files in RAM, a Disc (or Cassette) must allow for a pre-defined number of files in its 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, 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.

We can give each Disc an identifying volume label of up to six characters. When a Disc has a label, you can reference is by name using a period instead of a colon such as ".VOL2" instead of ":TAPE". This is somewhat slower than addressing the device name because the label is placed at the beginning of the Disc, and must be read each time it's referenced. 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 (which are quite rare) can be referred to by :MASSMEM.

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

Now, let's look at how the computer handles the mass storage media when we COPY files. When the 71 stores a file is on mass storage, a contiguous block of Disc space 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 the 71 copies the file back to Disc, it looks for a new block of records large enough to hold the entire file. If one is found, 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, an "End of Medium" error will be displayed and the file will not be copied, though there may be more than enough sectors scattered throughout the Disc. When the file is moved to its 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.

After this scenario is replayed several times, a considerable number of records may be wasted. The PACK command reorganizes files, squeezing out the unused records between files, and crunching the file allocation table. When it's done, the unused records are at the end of the media, ready to be used again. Since PACK causes considerable media wear and is subject to the vagaries of battery power, PACKing media should only be done as a last resort. Usually it's better to copy the files to a new Disc and start over. Earlier versions of the HP-9114A Disc Drive did not handle the PACK command correctly, so avoid using it on those early machines. If you do decide to use PACK, make sure you have spare copies of all important files on another Disc before starting.

LIST, PLIST *List a File.*

Prints or displays the contents of the specified BASIC file. If a printer is assigned and active, PLIST will send the data to it, otherwise it acts like LIST and displays the data on the LCD and, if there is one, the monitor. The LEX file in the Text Editor, HP-41 Emulator, FORTH/Assembler, and WorkBook71 ROMs add the ability to LIST and PLIST TEXT files. File types other than BASIC, KEY, and TEXT cannot be LISTED or PLISTed.

```
PLIST  ! List current file to PRINTER IS device.
LIST AFILE  ! List all lines in file named "AFILE".
LIST AFILE,10,100  ! List lines 10 through 100 in "AFILE".
PLIST AFILE  ! List "AFILE" to PRINTER IS device.
LIST KEYS  ! List the key assignments.
```

MEM How Much Available Memory?

Available memory is always a concern when running programs, allocating variables, or writing data files. The MEM keyword returns the amount of unused main RAM memory. Note that the syntax for this command is unlike others dealing with PORTs; MEM(0) will display memory in :PORT(0).

```
MEM  ! Display amount of RAM currently available.
MEM(0)  ! Display available RAM in :PORT(0).
```

NAME, RENAME File Utilities.

Since only one file of a given name may exist in RAM or on a storage 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 file you are currently editing.

```
RENAME TO APROG  ! Rename the current file to "APROG".
NAME APROG  ! Name the workfile to "APROG".
RENAME APROG TO BPROG  ! Change name of "APROG" to "BPROG".
RENAME APROG TO BPROG:TAPE  ! Change name of a Disc file.
```

File names must begin with a letter (A-Z) and may contain numbers (0-9) as long as the total length is 8 characters or fewer. 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, LOOP, and MASSMEM, as well as ALL, CARD and MAIN should be avoided.

OFF IO, RESTORE IO, RESET HPIL

The HP-IL Module tries to assign a printer and display whenever the 71 is turned on; this can take several seconds if the loop is broken (that's HP's way of saying nothing is plugged in). OFF IO disables HP-IL operations and speeds power-up significantly. RESTORE IO is used to re-enable 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's usually better to use RESTORE IO after connecting all devices. If flag -21 is clear, all devices capable of being turned off remotely (such as the Cassette and ThinkJet printer) 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.
SFLAG -21  ! Enable auto power-down of loop devices.
```

PRINTER IS, DISPLAY IS

Normally, the 71 automatically assigns HP printers and display 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 device.
DISPLAY IS PRINTER  ! Establishes printer as the display.
PRINTER IS DISPLAY  ! Establishes the display as the printer.
PRINTER IS PRINTER  ! Restores the printer to its rightful job.
PRINTER IS NULL  ! Throw away all printer output.
```

SECURE, UNSECURE File Utilities.

To insure that a file is not accidentally PURGEd or otherwise altered, they may be designated as temporarily SECUREd. This is especially useful with BASIC files; it's easy to forget which file you're in and accidentally change the wrong one.

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

SETDATE, SETTIME Set the Clock.

These commands set the system clock. When setting the time, remember that the 71 uses a 24 hour clock; be sure to add 12 for hours past noon. Usually we set time and date with strings; numeric expressions are better handled within programs.

```
SETDATE "88/09/04"  ! Set date with a string to September 4,1988.
SETDATE 1988366  ! Set date to day number 366 of 1988 (a leap year).
SETTIME "20:30:00"  ! Set the clock to 8:30 pm.
SETTIME 28800+(30*60)  ! Set clock using seconds since midnight.
```

STARTUP

When we first turn the 71 on, it does some self-tests then returns to whatever mode was active at power down. With the STARTUP keyword, you can make your 71 do any BASIC commands when you turn it on. This is helpful to automatically run a program or execute a series of commands. You might use it to set certain status like delay rate or display machine status. Halt the STARTUP routine by pressing ENDLINE. Think of the STARTUP sequence as a direct execution key assignment that the 71 presses whenever it turns on. The STARTUP sequence is ignored in CALC mode or when a program turned off the computer, but will be performed in FORTH or HP41 modes, though still with BASIC syntax.

```
STARTUP "CAT ALL"  ! Do a CAT ALL when it turns on.
STARTUP "IF TIME$>='13:00:00'THEN RUN LATE"  ! A conditional.
STARTUP "FOR X=1 TO60@TIME$@WAIT.5@NEXT X" ! Display time for awhile.
STARTUP ""  ! Deactivate startup command.
```

STD, FIX, SCI, ENG Number Format.

Much like a calculator, these statements set the number display format. The default is standard BASIC (STD) which displays in floating point format, showing only as many decimal places as necessary, and displaying in scientific notation when the number exceeds 12 digits. This sets the display format rounding, but does not affect the numeric precision used in calculations. These keywords set both the number of decimal places and the display mode; valid settings are zero through eleven. The number display format setting is used in BASIC, CALC, and in most programs, though some programs change this setting (and may not restore it when they are done).

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

TIME$, DATE$, SETTIME, SETDATE

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

```
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. There is a short delay built into this program to reduce power consumption. This routine could also be assigned to a key or used as the STARTUP string.

```
FOR X=1 TO INF @ DISP TIME$&" "&DATE$ @ WAIT.5 @ NEXT X  ! Display time.
```

WIDTH, PWIDTH Set Line Length.

We can set the maximum number of characters which the 71 will display or print on a single line with the WIDTH and PWIDTH commands; both may be set to different values. If a displayed or printed line is longer than the specified length, 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 a long line will not be displayed (ever). Again, many programs alter these settings.

```
WIDTH 22  ! Set max display line length to 22.
PWIDTH 80  ! Set max to be printed on one line to 80.
```

Getting Started 4

When you first turn on the 71, it's in BASIC Mode; the LCD display is blank except for the BASIC prompt character ">" and the flashing cursor. It's not really a blank stare, but a blank slate with which to work. The 71 accepts whatever you type, without question, until you press ENDLINE. Pressing ENDLINE tells the computer that you've entered a complete line and now it's time for it to do whatever it is you have entered, and (possibly) display an answer. In fact, just about everything we do with the 71 ends with ENDLINE. Since this is "a given," we'll often forgo even mentioning it. ENDLINE is the same as RTN, RETURN, EOL, ENTER or bent arrow (↵) keys on other computers, but it is *not* the same as ENTER ^ on RPN calculators.

We'll take advantage of an interesting feature of HP BASIC: If the result of an expression is not explicitly assigned to a variable then it is implied that the 71 should only display the result of the expression, and do nothing else with it. What this means is that when you enter a formula, the 71 will display the result; you could include the DISP keyword, but why bother. This trait of HP BASIC will become self-evident in a few pages. If you would like to have the result of each example printed, just precede each with the PRINT keyword.

It Just Beeps

More than likely, the first time you picked up your 71 and typed something, it displayed an error message then beeped; it happens to everyone. These messages are an aid in learning and using the computer; instead of curtly not doing what you've asked, it tells you why it can't do it. The 71 isn't trying to harass, just be helpful. Unlike a calculator, the 71 can do several hundred things. A keyboard with hundreds of keys is impractical, hence the command line and its series of messages. By the way, if you're tired of hearing the beep, just set flag -2.

```
SFLAG -2
CFLAG -2

SFLAG -25
CFLAG -25
```

```
! Set flag -2 to turn off beeper.

! Clear flag -2 to let the
! beeper work.

! Set flag -25 to beep even louder.

! Clear flag -25 to beep quietly.
```

Syntax

We humans accept some ambiguity in syntax in speech and writing because we can infer the meaning of a sentence from context, inflection, previous knowledge, body language, and such. A 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 certainly let us know. Other times, the computer will carry out the command but the results might not be what we had expected. Understanding syntax and how the computer parses is as important while working in CALC mode as when writing a program. Once you have a feel for how a new computer or programming language expects commands to look, all that's left to mastering the machine is learning its commands.

As time goes on, you'll find that your 71 beeps at you less and less. That's because you're learning the syntax rules for 71 commands. The 71 follows these rules of syntax when interpreting commands we type. This is called parsing.

The Parser at Work

We usually type spaces between BASIC statements for clarity, and a space separates each statement when you look at a BASIC program. As a convenience, these spaces can often be omitted when we enter commands. The 71 will use its 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 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 calls sub-programs. So it looks for a sub-program named "IOPE", and for our purposes we'll assume that "IOPE" is not the name of a program. This is a complete operation and it just didn't work, so the 71 beeps and displays the error message:

```
! The 71 displays an error message
! without the flashing cursor.
```

```
ERR:Sub Not Found
```

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

The second example, BEEPER, is a different kind of problem, and the 71 "logics it out" with another set of rules. The closest keyword is BEEP, which works alone, or we could optionally specify a frequency (tone) and number of seconds to beep. BEEPER will cause:

```
! Another error message when the
! 71 tries to interpret BEEPER.
```

```
ERR:Excess Chars
```

After a short pause, the original line returns to the display, with the cursor pointing at the first character in the line which the 71 didn't recognize as part of a valid expression. The computer placed the cursor after the third E because E is a valid variable name, and ER is not.

Let's modify BEEPER so that the 71 can parse it correctly. If there are two parameters, BEEP, like most commands, expects them to be separated by a comma. Any one of the following is acceptable:

BEEP BEEP 500 BEEP 500,1 BEEP E,R

The third problem we foisted on the 71 was XYZ, which is not a complete expression, and does not contain a complete keyword. Since 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 71 returns the original line to the display. Again, the cursor points to the first character the computer didn't recognize as part of a valid expression.

In the fourth example, the 71 returns a value without an error (finally). But is it the result we wanted? MEMORY is not a command in the 71's repertoire, 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 its 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. The computer interprets MEMORY as:

```
DISP 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 syntactical errors, but is of little help with logical errors.

Multi-Statement Lines

We can evaluate several expressions in one session by separating them with the "@" (commercial at sign). Often these will be system commands like DELAY 0 or OFF IO, but just about anything is fair game. This is the same format we use when writing programs, so we're actually writing a mini-program; add a line number and it's a program.

The 71 evaluates and displays the two expressions, one at a time, for the duration of the DELAY setting. If the DELAY is short you might miss the first number as it flashes by. A more useful method is to evaluate and display the formulas at the same time. The semicolon (;) concatenates (joins together) the two expressions so that they will display together on one line. A semicolon at the end of the expression tells the 71 that we are not finished displaying on that line, more will follow.

The 71 displays each number with either a leading space or a minus sign, and followed by a space so that they don't runintoeachother.

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 (which type of function the 71 will perform first), and a solution is returned. CALC mode displays intermediate values as operations are completed, but let's consider BASIC mode calculations in this chapter. Let's look at how various types of calculators handle multiplication:

Keystrokes	Calculator
2 X 3 =	Algebraic Calculator.
2 ENTER 3 *	Hewlett-Packard RPN.
2 * 3 ENDLINE	HP-71 BASIC Mode.

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 an answer. The conventional dollar-ninety-eight Algebraic calculator uses the = equals key, with the RPN calculator, it's the * key. Algebraic logic places the operator (+, -, etc.) between the arguments (numbers). The 71 is similar to the Algebraic calculator in that the operator ("*" is computer-ese for multiplication, "X" is just that, an "X") is placed between the operands. However, ENDLINE tells the computer that we are ready for it to get to work.

Most computers (and calculators) have a stack: A place to store intermediate results in a calculation or data passed between functions. 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 enters 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 evaluated. 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 intermediate results. We work with mathematical expressions instead of entering operations from the "inside out" to preserve the order of precedence. The True Algebraic calculator gives us complete control of operations, without concern for the stack.

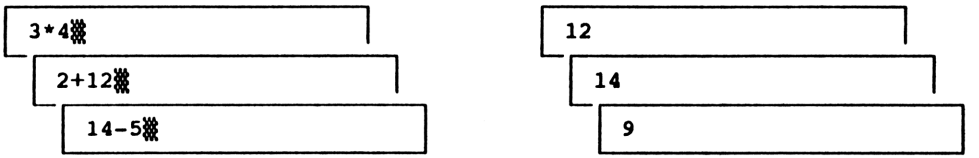
Mathematical Precedence

To avoid voluminous parentheses in everything we do, it pays to be conversant in precedence. Precedence, simply put, is the order in which mathematical expressions are evaluated. This order of priority probably differs a bit from what you remember from Algebra 101, but it's fairly consistent between various versions of BASIC.

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

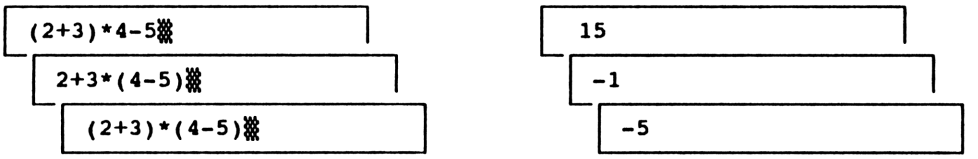
Enter them without parentheses, and the 71 evaluates mathematical expressions in the sequence above. Operations of the same level are interpreted from left to right.

Notice that we didn't begin with DISP or PRINT; the 71 assumes that we wanted to display the answer because we didn't assign it to a variable (we'll discuss variables in a few minutes). Here, multiplication is performed first. Since addition and subtraction are on the same level of precedence, the 71 interprets from left to right as they appear in the expression.



Parentheses

Enclosing parts of an expression within sets of parentheses tells the 71 which operations to perform first. If you're not sure of how an expression will be evaluated, add extra parentheses for clarity. As long as you include complete opening and closing pairs or parentheses, the 71 ignores leftover sets.



RES Register

The result of each mathematical expression is stored in the RESult register, regardless of if the value is assigned to a variable or just displayed. This is quite helpful when you need to see an intermediate result, then use that value again in the following expression. The RES keyword, assigned to the `f-ENDLINE` key, recalls the contents of the register. Remember, RES changes with *each* expression.

Other Operations

Not everything we do returns a value, and the flexible way the 71 does these sophisticated non-calculator jobs is what sets it apart from a calculator. Operations returning an answer are called functions, those which do not are statements. A statement is just that, a complete statement which 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. In fact, following English rules, it is a complete statement.

Strings

Computers work with text as often as with numbers. A string is a group of characters (letters, numbers, spaces...) enclosed in a pair of quotation marks. Either single (') or double (") quotes may be used, but both ends must match. A numeric variable with no value is zero by default, while a string with no value is null, or "". Yes, it's possible for a string to exist without any value at all. String "math" operators are the ampersand (&) to add two strings together, and Square brackets [ ] to tell the computer 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. You may designate one or two parameters within the brackets.

	Substring returned
[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. Try using string expressions with numbers and experience a whole new world of error messages.

"$" is usually pronounced as dollar or string. For example, TIME$ is pronounced "time dollar" or "time string." Dollar is probably the preferred pronunciation when describing a program over the telephone.

```
VER$
```

```
VER$[3,4]
```

```
VER$[1,4]&" Computer"
```

```
HP71:1BBBB
```

```
71
```

```
HP71 Computer
```

Notice that, unlike number functions, the 71 adds no extra leading or trailing spaces to the results of string functions. There is no equivalent "RES$" function to recall the last string result. Again, we can use a semicolon to display more than one string, or even strings and numbers, on the same line:

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

```
16438 HP71 9
```

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 string operations.

CHR$, NUM

The 71 expresses each character 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 are displayable. NUM takes a string and returns the numeric value of the first character. CHR$ is the *opposite* of NUM, accepting a value of 0-255 and returning 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")
 CHR$(65)
```

```
65
 A
```

```
NUM("%")
 CHR$(37)
```

```
37
 %
```

VAL, STR$

An incredibly powerful function shared by the 71 and a few larger HP computers is VAL. In most dialects of BASIC, it simply extracts a number from a string; the 71 carries this to the extreme, and evaluates a string as a mathematical expression (parentheses, variable references, the whole nine yards), and returns a numerical result. While not exactly the opposite of VAL (it can't restore a formula once it's gone), STR$ turns a number into its string equivalent. STR$ follows the current FIX setting, and truncates the fractional part or adds zeros as needed. These two functions are the main methods for exchanging data between the otherwise incompatible world of strings and numbers.

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

```
9
 Total=9
```

ASCII Codes

No computer is an island. Nor is a terminal, modem, or printer. ASCII is an acronym for American Standard Code for Information Interchange, the common format used for text by most computers. This standard, developed by the American National Standards Institute (ANSI), standardized assures that ASCII 65 is an uppercase "A", regardless of the machine. The computer drawing the television weather map, your bank's all-night-teller machine, an HP 1000, and the 71 could, given a way to connect them, have a reasonable conversation. Though it's hard to say what they'd have in common to talk about.

The standard is 8 bits (one byte) representing each character, with the eighth bit (the high bit in codes above 127) reserved for parity. A parity bit is a checksum of one kind or another which is tagged onto the data to assure that, when the byte gets to where it's going, it hasn't been garbled by a gremlin along the way. Add-up the bits when the byte gets to the destination and see if the checksum has changed. Computerdom has become much more reliable in the last 20 years, so the parity bit is rarely used today in data communications. Which left the engineers at each company to decide how to use the bit; while one might display hieroglyphics, another might print italics.

As are most "standards", ASCII is only partially recognized as standard; codes 32 through 126 are usually displayed or printed the same on any machine. While the 71 displays many of them as special characters (Greek to me), ASCII Codes zero through 31 are often reserved for printer or

display or file handling codes. For example, printing CHR$(15) sets many printers in condensed type mode, and CHR$(7) usually rings a bell (probably a squeaky little beeper), and isn't printed or displayed at all when you send it to a printer or terminal. Most HP 70 and 80 series computers display CHR$(7) as a pictograph of a bell; the IBM PC displays it as a left arrow character.

A byte is a byte, and since there are only 256 variations, the characters serve a second purpose beyond displaying ASCII codes. Programs and data are also represented with the same codes; you can tell the purpose of a byte by context and data structure. Regardless of how complex things seem, you can eventually reduce them down to one of those 256 numbers, and beyond that to the 8-bits (the one's and zero's) which make up the byte.

Calculator Variables

This book makes a distinction between calculator and program variables, though there really is none. So, why this extra complication when we usually try to make things easier on ourselves? Recall that the 71 can have multiple environments at the same time; often a running program will use an entirely separate set of variables. That's why we'll define the calculator variables as those in the main environment, used in CALC mode and BASIC mode, and in utility programs you write for your own solutions. A polite program will either use the main environment only with your permission, or not use it at all. This insures that your calculator variables are just that.

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. It's not just a philosophical difference, 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 needed.

We've been calling RES a register because it technically is: It's of a fixed size, type, and location, and can neither be created nor destroyed.

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, then read the variable to verify the answer. The keyword LET is optional (and rarely actually used) but is included for clarity.

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

X
```

```
! Assign the results to a variable.
! Then display the variable.

 9
```

This statement says "Look up the value of X and display it." We can use a boolean (logical) operator to do a comparison and prove that X does contain the result of that formula. A boolean returns a true (the number 1) or false (the number zero) result; any non-zero result is true. This boolean comparison lists the value on the left to help the 71 interpret the statement; had the variable name been given first, the 71 would have assigned it 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's no chance for ambiguity with other boolean operators.

The result of this argument is either one which means the argument is true, or zero is returned if our argument proves false. (whew!).

We'll get back to boolean comparisons later, but you may want to try this little program now to see how the 71 uses logical operators. Substitute the keyword PRINT for DISP if you'd like a printed table.

```
10 DISP " X  #  =  NOT AND OR  EXOR"  ! Boolean truth table.
20 FOR X=0 TO 1  ! Loop through zero and one comparison.
30 DISP X;X#1;X=1;" ";NOT X;" ";X AND 1;" ";X OR 1;" ";X EXOR 1
40 NEXT X  ! The end of the loop.
```

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

Symbolic Variable Names

Variable names, as you know, are represented by the letters A through Z. Since programs often need 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; we can use both X$ and X 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 71 operations is further expanded by the ability to store strings and numbers in several types of variables. These types specify the mathematical precision and usage of the variable. REAL variables are full precision, 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); the 71 rounds the fractional part to the nearest whole number when you store in an INTEGER. Each of these three variable types uses the same amount of memory; there is no memory savings in using shorter precision as there is with some other computers. This was done to standardize the way the 71 handles numbers internally, and simplifies things considerably.

DIM, SHORT, INTEGER

These three keywords explicitly create (or declare) variables. The keyword REAL can also be used to create REAL variables, though DIM works just as well.

Remember, arrays are not necessarily initialized to zero if they already are in existence. Variables of other types, if they already exist, are set to zero (or null if a string variable). The keyword DESTROY is used to get rid of unwanted variables.

```
DIM A,B,C,X$[96],K$[1]  ! Create REAL or string variable.
REAL X,Y,Z  ! This creates a REAL variable, just like DIM.
DIM A(5),B$(9)[10]  ! Create or re-dim a 1-dimension array.
DIM X1(3,4) ! Create or re-dim 2-dimension array.
SHORT V1,Y(23),Z(5,8)  ! Create a SHORT variable or array.
INTEGER M,R(11),B(3,2)  ! Create an INTEGER variable or array.
DIM S$(4)[10] ! 4 element string array, maxi 10 characters per element.
```

Arrays

An array consists of a group of elements of a single type, 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. We can create variables in any of the three precisions.

Two dimension arrays are referenced 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 the zero elements would not exist, it would contain twelve elements.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	0,0	0,1	0,2	0,3	0,4
Row 1	1,0	1,1	1,2	1,3	1,4
Row 2	2,0	2,1	2,2	2,3	2,4
Row 3	3,0	3,1	3,2	3,3	3,4

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).

```
OPTION BASE 0  ! Start arrays with element zero.
OPTION BASE 1  ! The first element is one.
```

Using Arrays

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 nonexistent 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; when in doubt, use DESTROY before creating the array. We'll talk about more practical method for creating them in a moment.

The MATH ROM is invaluable if you often use arrays. The MATH ROM function TYPE returns the type of the specified variable, and UBND and LBND return the bounds (number of elements in one direction or the other). If you do not have a MATH ROM, the most practical way to deal with arrays is to always explicitly DIM them, then keep track of their usage, and DESTROY them immediately when they are no longer needed.

Since we usually use single character names for scalar variables, it's often less confusing to use less common names for arrays. Arrays are referenced by an element number following the array name.

```
A(5)  ! Element 5 of the single dimension array A.
B6(2,3)  ! Element 2,3 of the two dimension array B6.
X0$(3)  ! Element 3 of the string array X0$.
Y9$(5)[2,3]  ! Characters [2,3] from element 5 of the array Y9$.
```

Indirect variables have long been a trick used by calculator programmers (STO IND X). Arrays give BASIC this advanced capability and extend the number of possible variables far beyond 286 (to a maximum of 65535). Elements may be themselves specified by mathematical expressions.

```
X=A(2*R,3*C)                    ! Read an element of the array.
```

As we said earlier, string arrays can only have a single dimension. Like regular string variables, the 71 creates string arrays 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. A single element may be null (empty) or filled to the maximum number of characters for which it is DIMmed without affecting other elements in the array.

Re-Dimensioning an Array

Interestingly, if an already existent 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 resetting unchanged elements to null.

Moreover, we can use re-dimensioning in a program, conserving memory by DIMing the array small. Then, if conditions change during the program run, we can increase the number of elements without losing information.

The 71 allows variable names be used for a single variable type at one time (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.

Variable Memory Use

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+exp	9.5 bytes
INTEGER	5 digit+exp	9.5 bytes
SHORT	5 digit	9.5 bytes
REAL ARRAY	12 digit+exp	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+exp	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:

Type	Precision	Memory Usage
COMPLEX	12 digit+exp	25.5 Bytes
COMPLEX SHORT	5 digit+exp	18.5 Bytes
COMPLEX ARRAY	12 digit+exp	16*(Dim1-Base+1)*(Dim2-Base+1)+9.5
COMPLEX SHORT ARRAY	5 digit+exp	9*(Dim1-Base+1)*(Dim2-Base+1)+9.5

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 it cannot be tested for an imaginary part (BEEP, Error).
- Strings are always created to zero length, but arrays are merely redimensioned without being zeroed.

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

CALC Mode 5

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's 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.

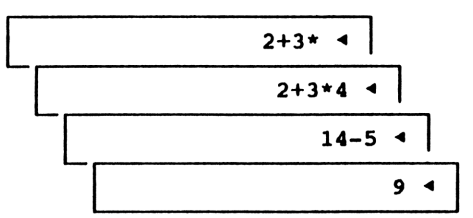
Hand the 71 to that friend and they will be able to at least compute sales tax without a ten-minute philosophical discussion about post-fix notation. CALC can be a powerful, expandable computational tool, or we can use it for simple, spontaneous calculations; just punch in an expression, and watch the 71 earn its 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 guard. As soon as the 71 can evaluate an intermediate result, it 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 really two levels of CALC mode: The CALC editor, during which the insert cursor is at the right edge of the display, and 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 in CALC mode (press g-CMDS or one of the up or down arrow keys) 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 end of the line character (crooked arrow ←┘) at the end of the line then pressing RUN to again use the CALC editor with that expression. USER keyboard assignments work while editing the stack, except immediate execute keys which don't cause an error, they are simply ignored.

Rules of Precedence

Like back in BASIC mode, rules of precedence dictate how the 71 evaluates intermediate results. Our earlier example of 2+3*4-5 is evaluated exactly as it is in BASIC mode. Notice again, multiplication is evaluated before addition so the intermediate result is based first on the multiplication, then the addition. The 2+3 hangs there while the 71 waits for us to enter the next expression to be evaluated.



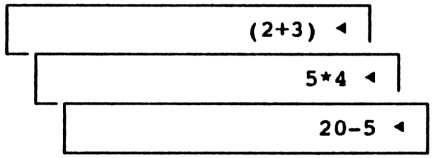

```
! With the next keystroke:

! The addition "hangs" there.

! The multiplication is evaluated
! followed by the addition.

! Finally, press ENDLINE.
```

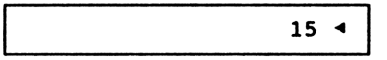
Again, we can use parentheses to insure that the expression will be evaluated as intended. The 71 will automatically enter a closing right parenthesis for each left parenthesis entered while in CALC, though BASIC mode requires matching sets. Entering a closing parenthesis, even though one is already displayed, will move the cursor outside of that set. Just in BASIC mode, the computer ignores extra sets of parentheses; and they're free, when in doubt use several.




```
! Evaluate the addition.

! Now, the multiplication.

! The answer when we press ENDLINE.
```



Long Formulas

CALC is, as are all types of input, 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 beeps and sends 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
  ()+23.2+31.0+35.1+38.6+46.7+51.5+63.1+74.8+61.3+59.2+52.6+43.1
```

RES, ()

We split the list above into two portions, pressing ENDLINE at a comfortable point (somewhere before the 96th character). 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 changes ***only*** when you press ENDLINE, so that intermediate results may again use () within the formula without changing RES. The RES keyword, an empty pair of parentheses, or just ENDLINE without anything displayed, will return the result of the previous calculation.

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.

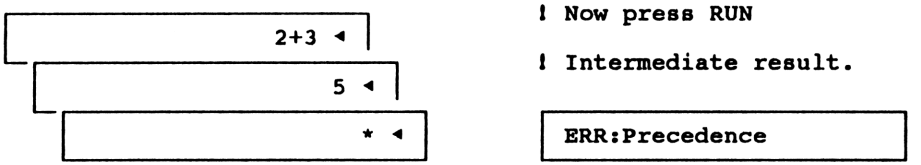
ENDLINE places the carriage return character on the end of the line. It's possible to exit CALC without pressing ENDLINE. In that case, if you edit the latest stack entry in BASIC mode, the last character on the line is not displayed. This is because the BASIC editor assumes that the final character on the line is probably a carriage return, which is not normally displayed.

ON

The cancel key. The formula is cleared from the display. The formula is *not* entered into the command stack nor is any 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 precedence. Using our above example, press RUN.



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 the final answer.

f-BACK

The "undo" key. Before pressing ENDLINE to terminate the expression. f-BACK, used with RUN, allows us to step backwards through an expression and 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. If you'd like more than the standard five entries, run the program CMDSTK, listed later.

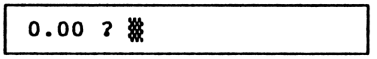
Variables

As with BASIC mode calculations, values may be exchanged with the calculator environment just by referencing 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 you may still evaluate intermediate results, changing formulas until you are satisfied, 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.
```

CALCAID Program

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




```
! CALCAID program display.
```

Enter a mathematical expression then press ENDLINE. The program saves results to variable L (easy to remember--think of it as LastX) 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. The program continues through this little loop, prompting for input and adding the result to X, until you end it. 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 qualifies 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. We exit CALC (press f-CALC again), and press RUN with this as the current program.

```
 10 ON ERROR GOTO 100  ! "CALCAID" program.
 20 DISP X; @ INPUT L  ! don't forget the semi-colon (;).
 30 X=X+L @ GOTO 20    ! loop until we get an error.
100 PUT "f," @ END     ! go to calc mode.
```

User Defined functions

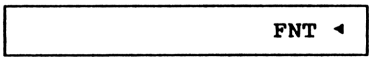
In addition to the power of adding new keywords using LEX files, we can write functions "on the run" in BASIC to use in formulas and save time entering often used or complex formulas. In the context of CALC mode, a user defined function can be thought of as a mini-program or "macro."

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 numerical result. The function must be in the current BASIC file, not in a SUB program.

```
200 DEF FNT=TIME  ! A simple one line function.
```

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.

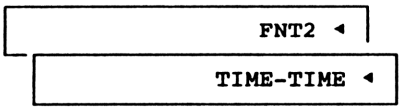



```
! Using a single line DEF FN in
! calc mode.
```

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 FNs is that they may use string functions. Let's define FNT2, which uses FNT twice, to see how long it takes to evaluate a function. Exit CALC mode and type the following:

```
210 DEF FNT2=FNT-FNT  ! Single line FN referring to another function
```

Go back to CALC mode and compare this to the time it would take to enter the expression directly in CALC to see the actual time savings.




```
! Call the nested DEF FN.

! Compare this result to the time
! the FN takes to run.
```

The "hook" of allowing user defined 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.

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 drain, 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 is the rest of the HP-71's Operating System, 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; not very much code space considering the complexity and versatility. 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 efficiency, 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 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. External 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.

```
  5 CALL MATHQUIZ @ SUB MATHQUIZ @ E$=CHR$(27)&'H'&CHR$(27)&'J'
 10 DISP E$&"Math Quiz" @ RANDOMIZE TIME
 20 ON ERROR GOTO 20 @ T=0 @ INPUT "Largest number? ","10";M
 30 M=ABS(IP(M)) @ IF NOT M OR M>999 THEN 20 !     make sure it's in range
 40 DISP "function:  +  -  *  /" !             prompt for type of function
 50 F=POS("+-*/",KEY$) @ IF NOT F THEN 50 !          wait for a proper key
 60 FOR Q=1 TO 10 !                               loop through 10 questions
 70 X=INT(RND*M) @ Y=INT(RND*M) @ IF X<=Y OR NOT Y THEN 70 !   get numbers
 80 IF F=4 AND NOT FP(X/Y) OR Y=1 THEN 70 !    integer answer for division
 90 C=0 @ DISP E$&"Question number";Q @ ON ERROR GOTO 100
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
200 IF T>7 THEN DISP "Good!";
210 DISP T;"of 10 correct" !          the lazy way to wait for a keystroke
220 WAIT 2 @ INPUT "quiz again? ","Y";E$ @ IF UPRC$(E$)="Y" THEN 10
```

Basic BASIC 6

This section introduces BASIC and simple programming. If you're new to programming in BASIC, or would like a refresher in the 71 dialect, read on.

BASIC is a child of the 60's, created at Dartmouth College, and raised as a learning tool for computer novices. It has easy to remember commands, simple syntax, and is probably the easiest computer language with which to learn programming. The acronym, Beginner's All-purpose Symbolic Instruction Code, is an understatement; BASIC has grown into a powerful programming tool.

Unlike most personal computers, the 71 is a useful tool without programs. BASIC programming expands the 71, helping us solve problems with speed and simplicity. Any complex pursuit, be it programming, surgery or farming, requires patience and a willingness to get your hands dirty. Back-up everything in RAM onto Disc, and let's get started.

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
```

```
1.9865
```

Now, suppose you have a whole list of numbers (cms's?) to convert. 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.

A program is a series of keystrokes we've recorded to use time and again without having to re-enter them. To differentiate between entering commands for immediate execution 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.

The 71 reads a program file a line at a time, and each line from left to right, a statement at a time. Each mathematical expression is evaluated using the same rules of precedence we use in BASIC or CALC mode. Contrary to popular myth, a computer does not wander up and down a program in a relentless quest, looking for what's most logical to do next. It does what we tell it to.

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 its various forms, displays a prompt, waits for you to input something, then stores the information where the program code tells it to. The 71 knows that we are entering a line in a BASIC file because we've begun it with a line number; without the line number, the computer evaluates it in immediate mode and displays the answer.

```
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
```

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

```
30 DISP Y
```

Fine, a simple solution. Let's embellish the answer to be more informative.

```
30 DISP Y;"inches"
```

The semicolon tells the 71 that more is to follow on the line, then we add the "inches". 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 string from the rest of the universe.

GOTO

We're writing the program to save time converting a lot of numbers. 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 71 to go back and start over when it's done.

```
40 GOTO 10
```

The answer stays in the display for as long as the delay setting (usually a half-second), then the program will GOTO line 10 and start over. Ending the 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 answer 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's followed by a space, so it's not necessary to add extra spaces to separate the numbers from the words displayed. With the final modifications, the program runs 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 ninety-six lines. This is not as casual a statement as it sounds. A large program can be thought of as a group of problems which can be broken down into a series of small solutions. Small solutions like the one above combine to solve big problems. 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

Hewlett-Packard has taken the approach of a multitude of commands with an editor which checks syntax when you enter a line of code. HP BASIC is an excellent programming environment because of its completeness and consistency of operation. And it's a language to grow with because of its speed and power. While you will probably need to refer often to the *HP-71 Reference Manual*, and you'll 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 idea 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'll begin with a prompt 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'll enter more than one statement on a line. Each statement is separated by "@", the commercial at sign; this is called concatenation. Lines beginning with an exclamation mark are remarks which are ignored when the program runs. Line 50 gives us the opportunity to exit the program by pressing E; this is usually better than suspending with ATTN.

```
  1 ! inch/cm converter.
 10 LC OFF @ DISP "metric converter" @ WAIT.5
 20 'MENU': DISP "convert to: Cm/In"  ! the main menu prompt.
 30 IF KEYDOWN("C") THEN 'CM'  ! wait for a key.
 40 IF KEYDOWN("E") THEN END  ! the escape hatch.
 50 IF NOT KEYDOWN("I") THEN 30  ! if wrong key, go back for another.
100 'IN': DISP "cm:"; @ GOSUB 'INP'  ! inch--> cm
110 Y=X*.3937 @ GOTO 'DSP'
200 'CM': DISP "in:"; @ GOSUB 'INP'
210 Y=X*2.54
500 'INP': INPUT X @ RETURN  ! input subroutine.
600 'DSP': DISP Y @ WAIT 1  ! display routine.
620 GOTO 'MENU'  ! we're done, return to menu.
```

Labels

As the program name describes the program, labels identify the purpose of a subroutine. 'ERRTRAP' handles errors, 'ASKYN' might handle input, and 'MAINLOOP' is probably just that; a descriptive label makes the program easier to write and maintain. While long labels consume a few extra bytes, a line which says GOSUB 'READFILE' won't need a remark to remind us what it does, or hours in debugging if line numbers are changed. Since the label is independent of the line number, we can insert, delete, and re-number lines without affecting the logic or flow when

the major modules are distinguish by program labels. Within 5 minutes of using labels, you'll wonder how you ever got along without them.

There are few restrictions on labels; anything may follow on the line without an "@", and the allowed names follow the same rules as program names. Some rules of thumb with labels are:

- Don't use function names for label names (it's confusing).
- Don't use same label twice (only the first one will be found when the program searches for it).
- Enter label references without quotes. When you enter the label or label reference without quotation marks, the 71 tokenizes the label with an implied quote, thus saving a byte over entering literal quotes.

Labels are especially useful with DATA statements. Begin each line or series of DATA statements with a label, then use RESTORE to place the pointer at the beginning of that sequence.

```
200 RESTORE 'NAMES' @ READ X$ @ DISP "Salesperson: "&X$
210 RESTORE 'DISTRICT' @ READ X$ @ DISP "District: "&X$
220 RESTORE 'SALES' @ READ X @ DISP "Total Sales:";X
500 'NAMES': DATA Bob,Mary,Vern,Carol
510 'CITY': DATA San Jose,Palo Alto,Livermore,Mountain View
520 'DISTRICT': DATA Valley,Downtown,East Bay,South Bay
530 'SALES': DATA 5600,6700,3400,5100
```

GOSUB, RETURN

GOTO, as you know, tells the 71 to go-to a line or label in the program. This is unconditional branching. Often we use a portion of a program several times to save memory, link modules, or 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 the 71 executes the GOSUB, the address (location in memory) of the GOSUB statement is stored in a list. When the 71 encounters the next RETURN, it looks up where it came from, and the 71 continues from the statement after the GOSUB.

Subroutines may call other subroutines by again using GOSUB, adding new entries to the list. This list is called, conveniently enough, the gosub stack, and it's maintained automatically by the 71 while the program runs. As you can see, for every GOSUB there's a pending RETURN, though there may be quite a bit of time and 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 gosub stack is empty and the program encounters a RETURN, an error message is displayed because the 71 doesn't have any place to return to.

- Statements following a GOSUB on the same line will not be executed if you press ATTN to suspend the program, then f-CONT to continue. It's good programming practice to account for this by placing GOSUBs as the last statement on the line.

An oversight even experienced programmers make is to follow a GOSUB with RETURN. When a subroutine calls another, that routine will also end with a RETURN, giving the 71 two RETURNs to execute one after the other, plus the overhead of an extra entry on the gosub stack. It will save some time and code if you place the call to the nested subroutine at the end of your routine and use GOTO. Be sure to note in a remark that the GOTO is in lieu of nesting the subroutine. Overuse of GOTOs in this or any manner results in "spaghetti code," which will be difficult to unravel later.

```
2000 GOSUB 3900 @ RETURN  ! Readable, but wastes code and time.
2000 GOTO 3900  ! Shorter and faster.
```

POP

If a RETURN statement following a GOSUB is never encountered, the pending RETURN remains on the gosub stack. Sometimes this is because of an error in program design; often conditions change and you may not want to return at all. The POP keyword cancels the RETURN from the *most recent* GOSUB. When the 71 POPs, the newest entry on the gosub stack is deleted, and execution continues from the POP statement. If you're using an all-purpose error trap, and the number of pending RETURNs will differ depending on how you got there, use the maximum number of POPs you will possibly need. Unlike extra RETURNs which cause an error, the 71 ignores extra POPs. The HELP subroutine, listed later, uses POP.

Computed GOTO, GOSUB

The 71 can help us write compact, readable, and quick programs, with program structures like ON...GOTO and ON...GOSUB. These are very advanced forms of conditional branching. The computer evaluates the expression, then uses the result to decide which line number or label to branch to. An answer of 1 sends the computer to the first line, 2 sends it to the second, and so forth, for as many line numbers (or labels) you've included. While superficially looking like the ON ERROR statement, syntax and use are quite different.

```
 10 FOR L=1 TO 3
 20 ON L GOSUB 100,200,300
 30 NEXT L @ DISP "end of program" @ END
100 DISP "subroutine #1" @ RETURN
200 DISP "subroutine #2" @ RETURN
300 DISP "subroutine #3" @ RETURN
```

Line 20 says "IF X=1 THEN GOSUB 100 ELSE IF X=2 THEN GOSUB 200 ELSE IF X=3 THEN GOSUB 300." If the expression evaluates to a number outside of the number of program lines (or labels), the 71 will cause an error; used carefully, this is a versatile program structure.

There's a bug in HP71:1BBBB which requires OPTION ROUND NEAR or OPTION ROUND ZERO for computed gosub and computed goto to work correctly. Use only simple numerical operators (+, -, *, /, DIV).

FOR, NEXT

While GOTO jumps between a series of lines, FOR and NEXT provide a more elegant and structured method for looping for a predetermined number of times. In the next example, 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 RND*1000,.1 @ GOTO 10
500 DISP "end of loop"
```

Besides running around in circles, we can have it loop for, say, 10 times.

```
 10 FOR L=1 TO 10 @ DISP "HP-71"
100 BEEP FP(TIME)*1000,.1
200 NEXT L
500 DISP "end of loop"
```

Now the 71 will loop for only ten times, then fall through to line 500. FOR and NEXT are paired like GOSUB and RETURN, except that you should not have multiple NEXTs for one FOR; any amount of program may exist between the two and you can jump in and out of the loop at will. A few BASICs (including this one) allow multiple NEXTs for a single FOR, but it is poor programming practice because of the dilemma of following the programs logic.

When the computer comes upon the NEXT statement, it increments the counter by one, *then* compares it to the TO value. If the resulting value is less than or equal to the TO value, the computer goes back to the statement after the FOR. In the above example the value of L is eleven after completing the loop, since it was incremented *before* comparing it to the TO 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 loop counter variable is a standard 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, unless we specify the STEP value.

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

This will still loop ten times. However at the termination of the loop, L equals 110, because the computer incremented it by 10 each time.

You can create an endless loop with FOR L=1 TO INF. Infinity is an unreachable goal for the 71; when it 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.

We can use positive or negative STEPs. The advantage of a negative number is to decrement the counter each time, instead of increment. A program called NOREMS, in the communications section, uses a negative loop counter to read a TEXT file backwards from end to beginning. This next little program shows the effect of negative and positive steps. This program prints a list of system and user flags which are set. Unlike the HP-41 function which prints the status of all flags, this program prints only flags which are set. If a flag isn't set, its status isn't printed. If no flags are set, nothing is printed.

```
10 CALL PRFLAGS @ SUB PRFLAGS  ! print system flags, then user flags.
20 FOR L=-1 TO -64 STEP -1  ! a negative step value.
30 IF FLAG(L) THEN PRINT "system flag";L;"is set"  ! print it if set.
40 NEXT L  ! the end of the first loop.
50 FOR L=0 TO 63  ! the default step is +1.
60 IF FLAG(L) THEN PRINT "user flag";L;"is set"
70 NEXT L @ END SUB
```

IF, THEN

The IF statement begins a powerful and easy to use method to conditionally execute a passage of program code.

- If the expression evaluates true (it's not zero), then the statement after THEN is performed, otherwise it's ignored. That's the same syntax as the previous sentence.
- 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.
- 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 use of IF...THEN is followed by GOTO or GOSUB, called conditional branching. For this reason if a line number or label follows THEN, it's interpreted as an implied GOTO. This is saying to the 71 "if the next 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."

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

We can evaluate the expression as false (zero) with 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
```

And we can do multiple comparisons with 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 nested Else's is reduced readability.

```
1600 IF X=1 THEN 400 ELSE IF X THEN 500
150 IF P=46 THEN PRINT CHR$(12); @ P=0 ELSE P=P+1 @ GOTO 80
```

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

The BASIC line editor does several jobs. If we give it a mathematical expression, it tries to evaluate it and return an answer. Enter a command like CAT ALL and the 71 will do just that. However, begin the expression with a line number and you've entered a line in a BASIC program. Same syntax requirements, same beeps and error messages. Since BASIC uses English-like words and algebraic logic, once we're familiar with its workings, the logic of a program can usually be figured out by reading the listing. An advantage of BASIC over FORTH or Assembly language is that we can edit and modify programs on-the-fly.

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

```
 9 ! ACYDUCY
10 STD @ B=100 @ Q=5 @ RANDOMIZE @ ON ERROR GOTO 30  ! initialize.
20 'START': DISP "Bank:$";STR$(B) @ WAIT 1
30 'BET': C=IP(11*RND)+2 @ IF C<2 OR C>11 THEN 'BET'  ! get 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 traps bad input if the user, for instance, enters 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 places a new seed in the random number generator. We've specified no parameter for RANDOMIZE, the 71 automatically uses 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 that 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, we can use anything which evaluates to a string as the default.

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

The subroutine 'CARD' displays the string equivalent of the current card. It's used three times within the program, so we've made it a subroutine to save memory.

There are no string variables used within the program. Each number is displayed as a string (using STR$) to suppress displaying extra leading and trailing spaces. 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 stressed, entire programs are often used as sub-programs in order to preserve the calculator environment. ACYDUCY is not a SUB, and uses the calculator variables, a problem if you've created the calculator variables to other types. For example, if B is an array, and line 20 tries to use it as a scalar variable, the program will halt (ok, "crash") with an error message. An equally important use of sub-programs is to make commonly used modules available to several programs, thus saving memory, and simplifying programming.

- While a subroutine begins with a local label and we enter it with GOSUB, a sub-program begins with a SUB statement, then we CALL it; separate program, separate variables.

INCAT is a sub-program which has little value as a stand alone program, it has no user interface at all, but is helpful in programs dealing with data files. INCAT returns a value representing the type of a specified file. It requires two parameters: A string with the file name, and a numeric variable in which to return the result. Possible answers 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 ACYDUCY, you can test INCAT.

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

```
! Call the program with a file name
! and a variable for the results.
```

```
7
```

We could use ADDR$ to find a file, then PEEK$-out its file type from the data in the file header, but that won't help if the file resides on Disc or it doesn't exist. Instead we keep in on a very high level: First we display the CAT entry for the file, then read the file type from the display using DISP$. 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 adds 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 most file types to this file. If you use APPT or ROM files, change line 9630 to recognize them. A program called NEC, in the communications section, uses INCAT.

```
9600 SUB INCAT(F$,T) @ ON ERROR GOTO 9660  ! trap filename error.
9610 DISP CHR$(27)&">"; @ CAT F$ @ T$=DISP$[1,32]  ! turn on cursor.
9620 IF NUM(T$[12])=32 THEN T$=T$[3]  ! trim spaces if Disc file.
9630 T=POS("TESDDABILEKEBAFO",T$[12,13])  ! look for the file name.
9640 IF NOT MOD(T,2) THEN T=20 ELSE T=(T+1) DIV 2  ! file type #.
9650 GOTO 9670  ! good data
9660 T=21 @ IF ERRN=57 OR ERRN=255022 THEN T=0  ! error trap.
9670 DISP CHR$(27)&"<" @ END SUB  ! turn off cursor, exit.
```

The sub-program receives values, evaluates them, and returns the answer in the same variables. When a sub is called, the parameter list points to the actual variable in the calling program's environment, though a different variable name may be used (more confusing for the 71, easier for us). 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 MYPGM(1,2,"test"), nothing is returned, because there were no variable names passed to return the data in. Of course a SUB can be called without passing any data; the maximum number of parameters we can pass between programs is 15.

- The number and type of parameters must match in both the SUB and CALL statements.
- Variables which the sub creates are in its own environment and do not affect the CALLing program.
- You can have any number of SUBs in a file.
- A sub-program can only be CALLed, never RUN; we can only RUN program files.
- The first sub-program in a file usually has the same name as the file.

To share the advantages of a SUB program, files often begin with a line which CALLs the SUB within the file; if it did not, the program would end when it ran into the SUB statement.

```
10 CALL MYPGM @ SUB MYPGM  ! a separate program environment
20 ! the program begins here...
```

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 machine language instructions for that operation. Consider this fragment:

```
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 that done, the 71 returns to the next statement on the line. It finds 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's 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 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, though they have a different number of characters. Spaces separating items do not take any memory. 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 71 looks up each token and displays the keywords and parameters and such. When we press ENDLINE while editing the line, the whole process of tokenization starts again.

Statements versus Functions

Tokens can be summarized in 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 the keyword BYE. BEEP is a statement which may optionally have up to two parameters. The 71 lumps together statements and commands, which some BASICs treat separately. Operators (like + or DIV) are a special case.

Functions may require zero or more parameters, do something with the data, and return either a numerical or string result. MAX, for example, is a function 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 result is a single number or string.

```
10 X=MAX(FP(Y),ABS(FP(Z)))  ! Nesting functions.
```

The distinction between statements and functions is often blurred, and the HP-71 has a penchant for allowing liberal BASIC syntax. One blanket statement about functions and statements is that functions may be preceded with the keyword LET and statements may not. In the strictest sense, a function, its operators, and the variable assignment, form a complete statement, though we'll let

that pass. If you are in a crowd of sticklers and don't wish to misrepresent a function as a statement, you can usually call it a token or keyword without anyone ever catching on that you are not sure what it is.

Spread among computer languages are a number of names for commands, parameters and data structures. Among them are (in alphabetical order) atoms, conditionals, definitions, directives, functions, keywords, library routines, macros, operations, operators, predicates, procedures, routines, statements, utilities, and words. In this chapter we'll usually say command or statement, and you can nod with a knowing smile.

The DECIDE Program

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 displays the answers one by one; press any key to move to the next item. DECIDE doesn't do very much, and what it does, it does elaborately; much ado about nothing. The 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=IP(HTD(PEEK$("2F949",1)&PEEK$("2F948",1))/32)  ! find DELAY.
  40 OPTION BASE 0 @ INTEGER K(9,9)  ! two dimension int. array.
  50 OPTION BASE 1 @ DIM E$[7],T$(9)[8],F$[72]  ! simple,array,long str.
  60 L=FLAG(-16,L)  ! restore option base.
  70 E$=CHR$(27)&"H"&CHR$(27)&"J"&CHR$(27)&"<-"  ! display clear string.
 999 ! input loops ----------------------------------------------------
1000 DISP E$&"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  ! get a key.
1060 L=FLAG(0,L-1)  ! clear flag 0 if repeating ratings.
1069 ! 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.
1109 ! see if rating has already been used.
1110 FOR L3=1 TO L-1
1115 IF K(L3,L2)=K1 THEN BEEP @ DISP K1;"USED" @ GOTO 1080
1120 NEXT L3
1130 K(L,L2)=K1 @ NEXT L2 @ NEXT L
1999 ! 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
```

```
3999 ! 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 ! ratings.
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 ! facts
4060 DELAY 0 @ DISP "New data, Results, End";  ! done. Go again?.
4070 GOSUB 'WKEY' @ ON POS("RNE",K$)+1 GOTO 4060,4000,50,8300
7998 ! subroutines --------------------------------------------------
7999 ! wait for a key --> replace with KEYWAIT$ 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.
8099 ! 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$) @ NEXT F @ F=9 @ RETURN
8199 ! user FN to trim trailing spaces from prompts.
8200 DEF FNT$(K$)=K$[1,POS(K$&" "," ")-1]  ! FNT$ trims trailing spaces.
8300 DELAY D @ PUT "#43" @ END SUB  ! restore DELAY, press ATTN, bye.
```

Strings

The four strings used by DECIDE are created differently. Unless dimensioned at the outset, strings can hold up to 32 characters. K$ is not dimensioned, so it is implicitly DIMmed to 32 characters. E$ is a constant containing an escape sequence and never changes. 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 large and used the same as T$. Each item in F$ is eight characters, the same as an element in the T$ array; F$ and T$ have similar uses. Let's display element 3 in F$ and array T$:

```
DISP F$[3*8-7,3*8]  ! An element from a simple string.
DISP T$(3)  ! An element in a string array.
```

A single string is often used instead of an array for applications which would seem natural for an array. One reason to avoid string arrays is for compatibility with versions of HP BASIC which do not have them. The second is that an "element" in a single string can overflow or be shifted easily. A single long string surpasses arrays when we know that we will be doing comparisons. It's easier to search a single string than use a FOR-NEXT loop to check individual elements in an array. The 71 makes a temporary copy of the string in free memory whenever it does string manipulation; be sure that you have enough memory for the string and its clone.

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

By giving us fewer but versatile string functions, HP makes some incredibly complex string operations almost automatic. These extract sub-strings:

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

Control Codes

Control codes are ASCII characters 0-31, which form yet another language to grapple with the built-in LCD, external displays, printers, and some other HP-IL devices. When we display or print these codes, the device performs the task instead of displaying the character. If the device can't understand the code, it's usually ignored. The ASCII table in the back of this book lists control codes. The 71's built-in LCD only understands four control codes; others are displayed as special characters.

CHR	Purpose	HP-71 response
8	Backspace.	Move the cursor one character position to the left.
10	Linefeed	Advance paper or display one line w/o homing cursor.
13	Carriage Return	Returns the cursor to the beginning of the line.
27	Escape	The beginning of a multi-character control sequence.

Escape codes are a special breed of control code, beginning with the escape character CHR$(27), and followed by one or more other characters. Escape sequences multiply the number of available control sequences and opportunities for confusion. If the device can't understand the escape sequence, it'll ignore the escape character plus one following character; the remainder of alien escape codes longer than two characters will be displayed or printed. We used display escape codes in INCAT to turn the cursor on then off, and in DECIDE to clear the display; let's look at some ThinkJet printer codes.

```
810 SUB NORMAL @ PRINT CHR$(27)&"&k0S";  ! normal printer font.
820 SUB EXPAND @ PRINT CHR$(27)&"&k1s";  ! expanded width printer font.
830 SUB BOLDON @ PRINT CHR$(14);  ! print characters in bold type.
840 SUB BOLDOFF @ PRINT CHR$(15);  ! turn off bold printing.
850 SUB LINES6 @ PRINT CHR$(27)&"&l6D";  ! print 6 lines per inch.
860 SUB LINES8 @ PRINT CHR$(27)&"&l8D";  ! print 8 lines per inch.
```

The SUB NORMAL prints a five-character sequence, while BOLDON prints a single character. Each ends with a semicolon; if they had not, the 71 would add it's own control codes (carriage return and line feed), which would make the printer print a blank line (control codes are invisible).

Akin to its workings with printers, the 71 speaks to display devices in a constant stream of escape sequences; moving the cursor left, right, word-wrapping, and occasionally displaying something. Whatever the computer displays on the LCD is echoed to the external display device. The HP-IL command OUTPUT is useful when we want to send data to an external display without it also

showing up on the 71's LCD. Line 10 in this program fragment clears the LCD and display device then displays "Hewlett-Packard HP-71". Remember, as with printing or displaying anything, control codes apply from the current cursor position. Line 20 uses OUTPUT to move the cursor to the column 20, row 12, then display a logo, without changing what's on the LCD display.

```
10 DISP CHR$(27)&"H"&CHR$27)&"JHewlett-Packard HP-71"
20 OUTPUT :DISPLAY ; CHR$(27)&"%"&CHR$(19)&CHR$(11)&"[(hp)]"
```

HP-71 Files 8

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 understanding workings of these files is a key to getting a *feel* for the 71.

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 one or 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 ROMs are added or removed (turn the computer off first please!) the 71 automatically keeps track of where and how big they are. When we EDIT, CREATE, PRINT# or PURGE a file in RAM, again the 71 keeps everything sorted out.

Finding Files

The keywords CAT and CAT ALL, discussed in Chapter 3, "Command Performance," let us view what files are in the 71 without mucking about. Let's use the catalog entry for the Math ROM. In this example it's in :PORT(2), though it could have been in :PORT(1) amongst others. If we did CAT :PORT(2), the standard header would be displayed followed by the first (in this case only) file entry. Were there other files in the ROM, the 71 would display their catalog information when we pressed the DN ARROW (↓) key. If we had entered only CAT MATHROM, only the information about that file would be displayed. CAT ALL lists the catalog information for every file. First the general catalog header is displayed then the information for the first file.

```
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 accidentally 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 one page down.

Next we have the file size. This is the approximate 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▒  ! How much memory for a file header?
```

Sizes for files larger than 99,999 bytes (pretty unlikely) are listed in number of kilobytes (or "K", which represents 1024 bytes); for instance 110K means a file of approximately 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:

Size	Description
16 nibs	File Name.
4 nibs	File type.
1 nib	Flags.
1 nib	Copy Code.
4 nibs	Creation Time.
6 nibs	Creation Date.
5 nibs	File Chain Length.

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; we'll only cover files in RAM here. File creation time and date are stored in BCD (well, kind of). Both fields are stored reversed so that 20:45 89/09/04 (8:45 p.m Sept 4,89) looks like:

54	02	40	90	98
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 ADDR$ function returns the location of files in RAM or ROM in hex strings. This location is the first nib in the file header; the file contents begin after the 37 nibs in the header and, if there is one, any sub-header. Since ADDR$ only works with normal, uppercase file names, the only way to find a file with, for instance, a lowercase name (such as the key file or LEX files in some ROMs) is by finding its position relative to the file preceding it in the file chain.

BASIC Files

The type of file created when you execute the EDIT statement. A BASIC file contains 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.

BASIC File Format

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

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

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), an "@" (4F hex) token precedes 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, each line number, even the first one in the file, is preceded 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").

0F The end of the line.

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, BIN files cannot be edited directly because they have no line numbers, and are coded differently than BASIC files. BIN files often run faster than BASIC, and 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 created with the FORTH/ASSEMBLER ROM and a thorough understanding of Vols 1-2 of the HP-71 Internal Documentation (IDS).

BIN File Format

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

Size	Description
5 nibs	Sub-program Chain Header.
5 nibs	Place holder FFFFF.
2 nibs	Filler byte 20.

FORTH Files

Files written in the FORTH language with the FORTH/ASSEMBLY and HP-41 Translator ROMs have their own file type(s). While files created with these ROMs share the same name, they are incompatible with each other. These files contain the extensible user written dictionary entries which make up a FORTH program. Programs written in FORTH often run up to twice as fast as those in BASIC. It is a matter of personal taste whether BASIC or FORTH is easier and more versatile to use. Commercial programs for the 71 are rarely in FORTH because they would require the user to own one of the two ROMs. One custom 71 ROM written in FORTH includes a FORTH interpreter.

LEX Files

Language Extension files add new BASIC keywords (such as KEYWAIT$) and new operations like 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 *at least* 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's preferable to use a LEX with a BASIC file instead of a BIN file because of its greater versatility. Like BIN files, they are usually written using the FORTH/ASSEMBLER ROM.

Using LEX Files

LEX files are never the current edit file. When they are in the computer their operations are automatically available. 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.

Trying to run a BASIC program without a required LEX file in the computer will cause an error; that's usually the only damage. If a program is corrupted (trashed) by accidentally running it without the LEX file, it's probably better to restart the program once the LEX file has been loaded than to CONT.

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)
```

```
! Display complex number and
! enter it into RES.
```

However the boolean operator NOT doesn't expect a complex number, so it's 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 its 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.

When significant events happen (like an error or even turning on the computer), the 71 polls LEX files to see if they have anything special to do. Some LEX files use keyboard or clock-tick interrupts to add a running clock display or add special keyboard features. This latter group, especially if you have several of them, will clearly slow down the computer. If your computer seems to have gotten sluggish lately responding to keystrokes, a LEX file may be the culprit.

DATA Files

These files may contain strings or numbers, and are the most versatile data 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 (unless they are complex), but strings can be up to the size of one record. When information to be saved is primarily numbers, an SDATA file is preferred. A TEXT file is best for 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), DATA files are appropriate.

The DATA file can be created to a fixed number of records. If it's RAM based, you can enlarge it by writing data past the end. 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.

Strings which overflow from one record to the next will cause problems which will be hard to straighten out. For example, 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▒
```

```
! Create a fixed-length file.
! Open the file.
```

As we said earlier, a number takes eight bytes, however 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), then 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 what's there:

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

```
! Move the pointer to rec#1
! Read then display the record.
```

```
ynergetics 456
```

Since 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.

Contents	Record
123 S	Record # 0.
ynergetic	Record # 1.
s 456	Record # 2.

If a string or number overflows into a record containing data, that data is lost. The DATA file structure is quite versatile, and therefore vague. 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, 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.

Bytes	Item	Unit
22	Name	20 byte string(+hdr).

12	Phone#	10 byte string(+hdr).
8	Dues	Real number.
42		Record size needed.

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).

Size	Description
4 nibs	Number of Records in file.
4 nibs	Record length.

TEXT Files

This file type is also known as the HP Logical Interface Format (LIF). TEXT files are designed for sequential access (that is reading or writing one line at a time in sequence). The TEXT file can be thought of as two types of files in one: Sequential and random. Since the record size is variable, TEXT files are valuable for storing free form information such as used by a Text Editor. TEXT files are also used to exchange files with other computers. The HP-71 was designed primarily as a number machine, Text was a secondary consideration at the outset, so random file writing requires an additional LEX file.

Using TEXT Files

PRINT# writes an end of file marker after it. If we had a ten line file and used RESTORE to move to line five, then used PRINT#, line five would become the end of the file; information beyond that point would be lost.

The EDLEX and TEXTUTIL LEX files add new operations to make TEXT files a versatile file type. Fortunately the HP-41 Translator, FORTH/Assembler, WorkBook71, and Text Editor ROMs each contain one of these files.

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          ! Move to the end of the text file.
```

Special TEXT File Keywords

As you can probably tell from the preceding, the author is fond of TEXT files, though not when they are used with PRINT#. The following keywords are in EDLEX (and TEXTUTIL) and make TEXT the most versatile files. These keywords work only on files in RAM, and only with TEXT files. These keywords are documented here because they are rarely explained fully in owner's manuals. These operations can be slow on a large file or on a file with several files following in memory because everything in RAM must be shifted when a record is inserted or deleted.

DELETE *Delete Record.*

```
DELETE# <channel>,<record>
```

Deletes a single record from an open TEXT file. Specify first the channel number, then the record number. All records after the specified record are left unchanged. Remember, the first record in the file is record number zero; to delete line 10 from a file which is assigned to channel 3 we enter:

```
DELETE# 3,9
```

FILESZR *Text File Size.*

```
X=FILESZR("textfile")
```

A function returning the number of records in the specified TEXT file if the file exists, or a negative number representing the reason it didn't. 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.

To differentiate the error message from a possible number of records, the function returns a negative number for the error number. Use the absolute value of the number to look-up the error number; obviously a file cannot contain a negative number of records, so, in the case of a negative result, you will know that the file is not a valid RAM-based Text file. 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 existent file which isn't TEXT.

INSERT *Insert Text File Line.*

```
INSERT#<channel>,<record>,"new record data"
```

Adds a record to an open TEXT file. The new record is inserted in front of the specified record. No information is lost, everything from the record specified to the end of the file is moved. It is best to have the TEXT file as the last file in RAM if fast access is needed. We usually use this statement instead of PRINT# because INSERT# does not write an end of file marker.

REPLACE *Replace Text File Line.*

```
REPLACE#<channel>,<record>,"data to replace old record"
```

Used instead of PRINT# to replace a current record with a new string. Does not alter the position of the end of the file marker. The new string does not have to be the same length as the one replaced.

SEARCH *Text File String Search.*

```
X=SEARCH("str",<column>,start record>,<end record>,<channel>)
```

A function which returns a numerical result of the search performed. As the names imply, use SEARCH# and REPLACE# to for a quick and effective search & replace. Required parameters are the search string, POS within the record to start the search, the starting place (record) and final record to search, followed by 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. Keep in mind that the searches are usually made from the end of the line, so the longest occurrence of any search string on each line will most likely be found.

If the file is empty or the string is not found, zero is returned. Otherwise it returns a very calculator-like result. For most uses only the integer (IP) portion of the number is used.

RRR.CCCLLL

R Record number within the file.
C Position within the record (the column number) where the match was found.
L The length of the string.

The backslash character (\) is the switch to enable special functions. If you do not want the following options to be performed, do not begin the search string with the backslash. The control options are:

\ (backslash) Start or stop a search feature.
. (period) Wild card character.
@ (at) Multiple wild cards.
^ (up arrow) Search from the beginning of the line.
$ (dollar) Search from the end of the line.

SEARCH# period (.) option
The wild card. A period represents *any* single character. The example could find "quick" or "quack" or any word with that four character sequence and an unknown character.

```
^qu.ck
```

SEARCH# at (@) option
The commercial at sign (@) represents any number of unknown characters on a single line. Since any number of characters may be represented by the @, the string should be specified carefully. The following could find "quack" or "quarterback".

```
\q@k
```

SEARCH# up arrow (^) option
The up arrow following a backslash specifies that the search will be conducted from the beginning of the line. In the example we will look for the string "quack" beginning at the start of the line. The word will only be found if it occurs at the beginning of a line.

```
\^quack
```

SEARCH# dollar ($) option
While up arrow (^) specifies a search starting at the beginning of the line, the dollar sign specifies that the comparison is made from the *end* of the line. The dollar must appear at the end of the line to designate this switch; a dollar sign anywhere else in the line is interpreted as the character itself.

```
\quack$
```

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, each line must begin with a four digit, sequentially numbered, line number.

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, an extra byte is padded 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. If you are exchanging files with other computers, be sure to transfer the TEXT file using their file format; this will happen automatically if you use a communications program. 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:

Size	Description
2 nibs	Key code.
2 nibs	Entry length.
1 nib	Assignment type.
varies	String assigned to key.

The key code is the actual key code 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 number storage, 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 insistence on calling everything a variable, and call the records in SDATA files "registers."

Using SDATA Files

Each record holds 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. The standard 8-byte length makes indexing into the file efficient in Assembly language, and a breeze in BASIC.

```
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 Chapter 10, "BASIC Programming Hints" 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 sub-header, each record is eight bytes long and holds a BCD number (though in a modified format at times). The first register begins at the 37th nib after the ADDR$ of the header.

The Data File

Many programs create or alter data which often needs to be retained for later use after the program ends. Obviously, calculator variables are not the place to store this data 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. Since, unlike BASIC programs, we cannot directly edit a data file, their use may be 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 overall look of a data file. Pre-planning how we are going to store information makes data files easier to understand. Remember that programs usually write the data; once a program is written, the intrinsics of maintaining the data file are automatic.

Three types of data files (DATA, SDATA, TEXT) offer 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. Differences in how these keywords work with each file type (which are extensive!) will be covered when we cover each type.

Creating the data File

Before storing or retrieving information, the file must created or loaded from mass storage. New data files are all created in the same manner. CREATE, like the other keywords 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 <FlType> <name>▓          ! Create a data file.
```

A space, not a comma, separates the file type and name. Most keywords require a comma between all parameters, so keep this in mind. If only the file name is specified, 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.

We can also create files 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. As you'll see in a minute, it's often necessary to declare the file size up front.

```
CREATE FlType name,size▓         ! Create a fixed-length file.
```

This method is required for data files on Mass Storage, because Disc (or Cassette) files cannot be expanded once they are created unless you load them into RAM. You cannot increase a data file on Disc because HP's format for files on mass storage requires the entire file to be in one contiguous block, and another file probably lives 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 its old resting place when you copy it back to Disc, the 71 looks for a new place to put it where it will fit. Disc based files 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 challenge 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 didn't realize that, and made it so that it couldn't).

The HP-IL module uses a 256 byte buffer when accessing Disc based files. As you move through the file, the module grabs 256 byte chunks into the buffer. You'll notice the slight delay every 256 bytes if you edit a Disc based file with, for example, a spreadsheet program. There is no simple way to increase this buffer size short of writing new file I/O code, a task to be avoided by the faint of heart.

Disc based files also have the consideration of the physical size of a record, which is 256 bytes (the same size as the buffer). While this isn't a fixed rule, we usually create file record sizes in either multiples of this size, or easily divisible fractions, to minimize the number of file records which are spread between two Disc records. This minimizes access time and medium wear because the drive would otherwise have to read two physical records to read one file record worth of information.

To be simplistic, imagine a dresser with drawers holding 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 beginning 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 in data file operations instead of the file name. This is called opening a file.

```
ASSIGN # 1 TO MYFILE▒          ! Open the file.
```

There are a few things to remember when opening files:

- Valid file channel numbers are 1-255.
- A maximum of 64 files may be open at a time.
- Each file can only be assigned one number at a time.
- If the file specified with ASSIGN# does not exist, a DATA file of that name, with 256 byte records, will be created.
- Open files require an extra 34 bytes for the open channel.

- Disc based files need another 256 bytes for a buffer to store the current record being accessed, so that the 71 doesn't have to read the Disc constantly. The record in this buffer is written to the file when you move to a different record or when the file is closed.
- You should never remove a Disc from the drive when there are open Disc-based files.

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's 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 sequential access, working from the beginning to the end of a file, reading each item in sequence.

The RESTORE statement moves the pointer at a specific place in the file for random access. We can move around the file, pointing to records, then reading them as desired.

```
RESTORE #1,10
```

```
! Move the file pointer to
! record#10 of file #1.
```

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
```

```
! Write a record (the value of A)
! to file #1.
```

This statement enters the contents of variable A at the pointer position in the file associated with channel #2. If there had been data at the pointer position, this would replace it. If we had been at the end of the file, 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$
```

```
! Write string record to file #1.
```

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

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

```
! Move to record 10
! then write to file.
```

This is the same as:

```
RESTORE #2,10▒                    ! Move to record.
  PRINT #2;Q$▒                    ! Separate write.
```

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 causes an error.

Print more than one item at a time to the file by separating them with commas:

```
PRINT #2;A,B,C▒                   ! Write three values sequentially
                                  ! to file# 2.
```

Recalling Data

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

```
READ # 2;A$▒                      ! Read a record from file# 2 to A$.
```

This statement says "Read the next record in file number two and return its contents to the variable A$." Again, we can read multiple items:

```
READ # 2;A,A$▒                    ! Read records to variable A
                                  ! then A$ in sequence.
```

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.

Record type	Operation	DATA	SDATA	TEXT
String	READ#1;A$	OK	OK	OK
String	READ#1;A	ERROR	ERROR	OK
Number	READ#1;A$	ERROR	ERROR	
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 also closes files.

```
ASSIGN # 1 TO *▒                  ! Close file associated with #1.
```

You can also furnish a quoted string to ASSIGN#; if it's null or contains an asterisk ("*") it will close the file.

When a program we RUN ends, the files it opens are automatically closed. However, when exiting a SUB or any program which was CALLed, the files are not automatically closed. If a CALLed program tries to assign a file left open by another program, an error is generated. The statements CLFILS and CLOSEALL, available in some LEX files, will close all files. Files are automatically closed when they're purged, though not when the current edit file changes.

```
 10 CALL BIOR @ SUB BIOR ! biorythm printer.  With TAB and PRINT USING
 20 DIM E$[51],S$[51],M$[36] ! date format: month$, day$, 4-digit year
 30 DEGREES @ M$="JanFebMarAprMayJunJulAugSepOctNovDec"
500 LINPUT "What is your name? ";E$
520 INPUT "Birthdate(m,dd,yyyy): ";M,D,Y @ IF Y<1880 THEN 520
530 PRINT TAB(20);"Birthdate   ";M$[M*3-2,M*3];D;",";Y
540 GOSUB 9000 @ X1=J @ S$=DATE$ ! don't concatenate DATE$.
550 INPUT "Biodate(m,dd,yyyy):",S$[4,5]&","&S$[7]&",19"&S$[1,2];M,D,Y
560 GOSUB 9000 @ X2=J @ R=X2-X1
570 INPUT "Plot number of days:","30";B @ DISP "print instructions Y/N"
580 Q=POS("YN",UPRC$(KEY$))-1 @ IF Q<0 THEN 580 ! wat for a good key
590 IF LEN(E$) THEN PRINT "Biorythm plot for: ";E$
600 PRINT TAB(20);"Age ";R;" Days"
610 PRINT TAB(20);"Biorythm for";B;" days from ";M$[M*3-2,M*3];D;",";Y
620 PRINT TAB(20);"p=Physical   s=Sensitivity   c=Cognitive" @ PRINT
630 PRINT USING "14X,'-100',10X,'Low',10X,'0',9X,'High',9X,'+100'"
640 S$=CHR$(124)&"----------------------+----------------------"&CHR$(124)
650 PRINT TAB(17);S$
1000 FOR L=1 TO B @ E$=CHR$(124) @ E$[26]=E$ @ E$[26]=E$
1010 X=ABS(25*SIN(360*R/23)+26) @ E$[X,X]="p"
1020 X=ABS(25*SIN(360*R/28)+26) @ E$[X,X]="s"
1030 X=ABS(25*SIN(360*R/33)+26) @ E$[X,X]="c"
1050 X=J-1721119 @ N1=X+2 @ Y1=IP((N1-.2)/365.25)
1060 N2=N1-IP(365.25*Y1) @ M1=IP((N2-.5)/30.6) @ D=IP(N2-30.6*M1+.5)
1080 IF M1<=9 THEN Y=Y1 @ M=M1+3 ELSE Y=Y1+1 @ M=M1-9
1090 PRINT USING "6X,3A,3D,4X,51A";M$[M*3-2,M*3],D,E$
1100 R=R+1 @ J=J+1 @ NEXT L @ PRINT TAB(17);S$ @ PRINT @ IF Q THEN END
2000 PRINT '"P" = The 23 day Physical cycle.';
2010 PRINT ' Relates to Vitality, Endurance and Energy.'
2020 PRINT '"s" = The 28 day Sensitivity cycle.';
2030 PRINT ' Sensitivity, Intuition and Cheerfulness.'
2040 PRINT '"c" = The 33 day Cognitive cycle.';
2050 PRINT ' Mental Alertness and Judgment.' @ PRINT TAB(41);'-+-'
2060 PRINT '     Greater than "0", high values, Energetic and Dynamic.'
2070 PRINT '     Less than "0", low values, Recuperative periods.'
2080 PRINT '     "0" values, critical days, Accident prone days,'
2090 PRINT '         especially for the physical and sensitivity cycles.'
2100 END
9000 Y1=Y+(M-2.85)/12
9010 J=IP(IP(IP(367*Y1)-IP(Y1)-.75*IP(Y1)+D)-1.5)+1721115 @ RETURN
```

BASIC Programming Hints 10

There are usually several ways to do 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 chapter 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'll cover both together.

Keep in mind that these techniques form the modules of efficient programs, and efficiency is the real bottom line. There are a few disadvantages in using memory conservation methods:

- Remarks are invaluable for understanding program flow, but use up memory.
- Convoluted programs can be difficult to maintain.
- Since BASICs differ, even in the HP camp, many of these techniques will make it difficult to adapt programs to (or from) other machines.

Nesting Mathematical Expressions

While BASIC is an algebraic language, the 71 is internally a stack oriented machine vaguely reminiscent of HP's RPN calculators. When an expression is interpreted, the 71 passes parameters to the functions, and places intermediate answers 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 code, the next solution is considerably faster than this code. 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, 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 designed the math stack is also responsible for versatile handling of string subscripts. Since strings usually 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 fragment 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]
```

Testing Execution Speed

Code which is to be repeated often should be optimized for speed over size. For example, MIN(2,1) is faster than MIN(1,2); placing the largest likely argument first, with either MIN or MAX, is most efficient. The only way to know this by testing.

A fair test of any procedure is 500 iterations; this typically runs under five seconds, and is small enough that you can run it two or three times for each combination. Change the arguments or order of variable assignment and run the test again. You might find that some code runs faster not concatenated! If you want to use the results of the test again, don't begin it with a SUB statement.

```
    1 X=1 @ Y=2  ! assign some values to test.
    9 T9=TIME @ FOR L9=1 TO 500
  100 Z=MIN(X,Y)  ! the code we're testing for speed.
 9999 NEXT L9 @ DISP (TIME-T9)/(L9-1)-.01  ! iteration minus correction.
```

The core of the program uses normally unused line numbers so that we can copy lines directly from the program we're developing. We're using odd variable names (T9, L9) which are unlikely to be found in the code you're testing. Line 9999 divides the time by the loop counter and subtracts a fudge-factor (some overhead) to get a reasonably un-distorted time for a single iteration. Test the loop with a simple variable assignment (X=1) and correct the fudge factor to zero-out the result to make the differences in actual runs more flagrant. Watch for machine differences between runs; the run file or a data file could move, distorting file access times. Don't use remarks within the loop; everything counts. And, if you work late, watch for time roll-over at midnight.

OPTION BASE, DESTROY ALL, RESET

Programs written for distribution should avoid selecting OPTION BASE 1, even at the expense of wasting the possible zero element, because it is a global setting, and affects the operation of the entire variable chain. BUBLSORT, in Chapter 14, "Non-obfuscating Programs," ignores zero elements (if they exist). DECIDE shows how to save the current OPTION BASE, change it as needed, then restore it when done; if you must set OPTION BASE, this is the preferred method.

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. Representative is EDTEXT, in the FORTH/Assembler ROM, which sets uppercase mode; nice for Assembly source files, but not for writing letters.

Variable Names

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

DIM Strings

It requires 5-7 bytes to DIM a string. There is some memory savings by not DIMing a string which will contain from 25-28 characters. An exception is ROM 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. Specifying INTEGER or SHORT precision will not save memory; use INTEGER to round a number.

FOR-NEXT Loops and numeric comparisons are slower with integers.

Constants

PI (3.14 etc) is a constant, and is part of BASIC because it is used so often and never changes. Programs often use other values time and again, many times it will save memory to assign the number to a variable. Let's take the memory required to create a single digit constant variable.

9.5	Variable
5.0	Statement
14.5	Total

A single-digit number in a program (for instance X=1) takes the 5 bytes, the same as recalling a constant (X=C). When we go to a two-digit constant, the break even point for using the variable is 16 uses. Only when multi-digit numbers are used frequently does it become practical to use vars for constants.

We can thank the HP 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 state which may change as the program runs. The function FLAG sets or clears the flag accordingly but also returns one if the flag was not inverted, or 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 var for the result of the FLAG function; use DISP to save a variable if you don't mind displaying an occasional spurious zero or one.

```
IF FLAG(1) THEN CFLAG 1 ELSE SFLAG 1  ! invert a flag.
X=FLAG(1,NOT FLAG(1))  ! invert a flag using a scratch var.
```

Flag Variables

In the same respect, we can invert a variable used as a flag quite simply. When a flag is to be inverted often, and it isn't important that it be displayed (flags 0-4), it's more efficient to use a variable than a real flag.

```
X=NOT X  ! invert a variable used as a flag.
```

Clearing an Array

Usually we zero out numeric or string arrays in a loop:

```
1000 FOR L=0 TO 20 @ X(L)=0 @ X$(L)="" @ NEXT L  ! clear an array.
```

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

```
100 DESTROY X,X$ @ DIM X(20),X$(20)[10]  ! clear array quickly.
```

User Defined Functions

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

```
1000 Z=FNX(Y)  ! Calling a used defined function.
```

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

```
1000 A=Y @ GOSUB 9200 @ Z=B  ! in lieu of a DEF FN.
```

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.

END, END SUB

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 FRED
300 IF <expression> THEN 100  ! the implied end of the program.
400 SUB BARNEY ...  ! new program. implied END SUB.
500 SUB WILMA ...  ! another new program.
```

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'  ! implied goto.
```

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). The two fragments will branch to label 'A1' if X=1, to 'B1' if X=2 and so forth.

```
340 IF X THEN CHR$(X+64)&'1'  ! calculated goto to a label.
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 referenced 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 limits 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.

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

Other comparisons including NOT, MAX, MIN and MOD are usable 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
```

ON ERROR

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

```
 10 IF NOT X THEN 10  ! if X=0
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. The code at label 'ERRTRAP' would contain code to interpret the error or provide the equivalent of an ELSE.

```
100 ON ERROR GOTO 'ERRTRAP'  ! set the error trap.
110 ON X GOTO 'A1','B1','C1'
```

We can also use ON ERROR to branch to an all purpose routine following an intentionally caused error. This is an implied GOTO to a nonexistent label.

```
200 IF NOT X THEN ''  ! implied goto to a null label to cause an error.
```

GOTO, GOSUB to a Label

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

Enter Labels Without Quotes

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.

Optional Parameters

Several statements may optionally be entered without parameters or in an abbreviated 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 unrecoverable displayed data may be assigned to a variable by turning on the cursor before displaying the information. Display the escape character (chr$(27)) followed by ">" and ended by a semicolon to suppress the CR/LF to turn on the cursor. The example assigns the CAT to C$. Line 520 is optional and turns off the cursor to keep from having it (occasionally not even flashing!) present at odd times. The INCAT sub-program listed earlier is a practical application of this operation.

```
500 DISP CHR$(27)&">";  ! turn on the cursor.
510 CAT @ C$=DISP$  ! display the data.
520 DISP CHR$(27)&"<";  ! turn the cursor off.
```

Passing Parameters

DISP$ offers a unique way to pass parameters to programs. You can start the program in three ways: First, CALL or RUN the program normally, it will pause and ask for the parameter. assign the program to a key using DEF KEY "?","CALL PASS":, then enter the parameter on the edit line and press the key assignment. The third method is to CALL it from another program which displayed the parameter with the cursor on (which is how INCAT, in Chapter 7, "HP-71 BASIC Programming," works).

```
10 CALL PASS @ SUB PASS
20 X$=DISP$  ! read the input line.
30 IF NOT LEN(X$) THEN INPUT "parameter:";X$  ! input if none passed.
```

Checking For a LEX File

Most larger LEX files answer to the version poll. Therefore you can see if they are in the computer by looking for their name in the string returned by VER$. Don't look for an exact match unless it is important to know which version of the LEX file is there (such as working around a bug). Except for the beginning of the line ("HP71:"), each LEX file is separated by a space; the revision

number follows a colon. Include both the space and colon to insure that another LEX file name doesn't contain the one you want.

```
IF POS(VER$," HPIL:") THEN DISP "HPIL Module present"
IF NOT POS(VER$," MATH:") THEN DISP "No MATH ROM present"
```

ON TIMER

Timers are an under used feature; every ten minutes the program might save to disc, or twice a minute it will update the time on the monitor, or every other minute it'll take a reading from an HP-IL device. ON TIMER GOSUB is more useful than the GOTO variety; we can have two or more processes effectively running concurrently. Since the timer could interrupt at any time, be sure to designate separate variables for use only within the timer subroutine. If your program uses INPUT, KEYWAIT$, or calls another program, the timer won't "go off" until that procedure ends. So, while the 71's clock is very accurate, the time the timer subroutine was entered may be off by the amount of time it took for any of these unrelated tasks. Be sure to set timers for one second or longer; a bug causes VER:BBBB 71's to occasionally "hang" when timers are set for under about .7 seconds. Chapter 11, "PEEK$s and POKEs," has a program fragment to check if a timer is active.

```
ON TIMER #1,5 GOSUB 'INTERUPT' ! set-up a 5-second timer interrupt
OFF TIMER #1 ! disable the timer
```

- The three timers are global. Change them in a sub-program and you've changed them in the main program.
- Use variables in timer subroutines which are not used in the main program.
- Avoid setting timers recursively.
- Don't forget to turn them off!
- Set times longer than one second.
- Timer interrupts will be delayed until after INPUT or sub-program calls.
- Remember, they won't interrupt an INPUT or KEYWAIT$.
- END or STOP de-activates timers.

Displaying a Help Line Using a Timer

One HP-41 innovation is on-line help; press a key to see what it does without executing the function. This routine is generally called from a menu driven sub-routine to display a line of help if the user continues to hold down the key. Since your subroutine was entered when the user pressing a key, this routine checks to see if a key is still down. If the user releases the key immediately it returns without displaying anything. If the user holds down the key for more than one second, it displays "cancel" and never returns, but instead goes to the label 'MAINLOOP'. From your subroutine, place a help message in X$, then GOSUB HELP. The second version does not use a timer and can be used for duration shorter than one second.

```
6800 'HELP': IF KEYDOWN THEN DISP X$  ! display prompt if key down.
6810 ON TIMER #1,1 GOTO 6820  ! set the timer .
6820 GOSUB 6850 @ OFF TIMER #1 @ RETURN
6830 DISP "cancel" @ OFF TIMER #1 @ GOSUB 6850
6840 POP @ POP @ POP @ GOTO 'MAINLOOP'  ! timeout, don't return.
6850 IF KEYDOWN THEN 6850 ELSE RETURN  ! wait for no keys down.

8150 'HELP2': T=TIME @ IF KEYDOWN THEN DISP X$  ! doesn't use a timer.
8160 IF NOT KEYDOWN THEN RETURN  ! exit if no key is down.
8170 IF TIME<1+T THEN 8160
8180 DISP "cancel" @ GOSUB 8190 @ POP @ POP @ GOTO 'MAINLOOP'
8190 IF KEYDOWN THEN 8190 ELSE RETURN  ! wait for no keys down.
```

Input without INPUT

Many of the commands in high level languages like BASIC are complete utilities, almost sub-programs. INPUT and LINPUT are among the most powerful, automatic and complex statements, with machine language subroutines nested fourteen levels or deeper (amazing on a machine with an 8-level hardware return stack). There are a few cautions in using INPUT:

- Limited to 96 characters.
- ON TIMER will not interrupt.
- Cursor keys enable the command stack.
- Require a key file to trap special keys.
- Will time-out and turn off the computer after 10 minutes.

Many programs work around INPUT by re-assigning the cursor keys to immediate execute keys, and use DISP$ to recall the input string. The following routine is primitive, slow, and limited it its simulation of INPUT, but gives an idea of how complex INPUT really is. It displays a default string, allows minimal editing, and returns a string. If flag -3 is set, the computer won't time-out while waiting for keys. It assumes delay is set to 0 and displays one character longer than fits in the LCD so the arrow annunciator will light when the string exceeds 21 characters. The routine can be enhanced to use overstrike mode and trap the arrow keys.

X	Maximum number of characters allowed.
X$	Input string (dim it as large as necessary).
K$	Current keystroke (dim at least 4 characters).
ENDLINE	Enter the line and return.
LEFT	Backspace.
g-ERRM	display the latest machine error.

```
4000 'MEMO': DISP CHR$(27)&"E"&X$[MAX(1,LEN(X$)-20)];  ! cursor on.
4010 K$=KEY$ @ IF NOT LEN(K$) THEN 4010  ! wait for a key.
4020 DISP CHR$(27)&"<"  ! turn off the cursor.
4029 ! see if the user pressed g-ERRM
4030 IF K$="#161" THEN DISP ERRM$ @ GOSUB KEYUP @ GOTO 'MEMO'
```

```
4039 ! flush the key buffer. if ATTN then cancel and return.
4040 POKE "2F442","00" @ IF K$="#43" THEN RETURN
4049 ! if left-arrow then delete a char from X$, go back for more.
4050 IF K$="#47" THEN X$[LEN(X$)]="" @ GOTO 'MEMO'
4060 IF K$="#38" THEN RETURN ! is the key ENDLINE? if so then exit.
4069 ! if K$ >1 char, we have an unwanted control key, ignore it.
4070 IF LEN(K$)>1 THEN GOTO 'MEMO'
4079 ! is the string under max # chars? then add this char.
4080 IF LEN(X$)<X THEN X$=X$&K$ @ GOTO 'MEMO'
4099 ! the char was valid but the string was too long.
4100 DISP @ DISP "<<<string too long>>>"
4110 BEEP 900,.2 @ BEEP 1200,.2 @ GOTO 'MEMO'
4499 ! wait for no keys down. has no effect on any variable.
4500 KEYUP: IF KEYDOWN THEN 8060 ELSE POKE "2F442","00" @ RETURN
```

Quoted Strings

Enter GOTOs to labels and HP-IL device specifiers without quotation marks.

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

String Arrays

When first referenced, 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 next example is excerpted from a commercially available HP-71 program. We've changed the line numbers 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 was 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...

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$ (or HI$), 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; use it on a string with the high-bits already set, and it will clear them.

```
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 next 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  ! create alternate charset.
15 DIM C$[768],Y$[48] @ DISP "Underline/Inverse"
20 T=POS("UI",KEYWAIT$) @ IF NOT T THEN 20  ! wait for a keystroke.
25 DELAY 0 @ CHARSET "" @ FOR X=128 TO 255 STEP 8
30 FOR Y=0 TO 7 @ DISP CHR$(X+Y): @ NEXT Y @ Y$=GDISP$[1,48]
35 IF T=1 THEN 45
40 FOR Y=1 TO 48 @ Y$[Y,Y]=CHR$(255-NUM(Y$[Y])) @ NEXT Y @ GOTO 55
45 FOR Y=1 TO 48 @ Z=NUM(Y$[Y]) @ IF Z<128 THEN Y$[Y,Y]=CHR$(Z+128)
50 NEXT Y
55 C$=C$&Y$ @ DISP @ NEXT X @ CHARSET C$
60 GDISP CHARSET$[390] @ BEEP  ! Display the character set.
```

If you have HGL$ or HI$, this second version is faster. It is initially slower because it appends to the character set as it goes, instead of maintaining it in a string variable. The speed gain is using HI$ on line 35 to set the high-bit on each character; since the high-bit is the bottom, this underlines the character.

```
10 CALL CHARSET2 @ SUB CHARSET2  ! this version uses HI$.
15 DIM Y$[48] @ DISP "Underline/Inverse"
20 T=POS("UI",WTKEY$)) @ IF NOT T THEN 20  ! WTKEY$ similar to KEYWAIT$
25 CHARSET "" @ DELAY 0 @ FOR X=128 TO 255 STEP 8
30 FOR Y=0 TO 7 @ DISP CHR$(X+Y); @ NEXT Y @ Y$=GDISP$[1,48]
35 IF 1=T THEN Y$=HI$(Y$) @ GOTO 55
40 FOR Y=1 TO 48 @ Y$[Y,Y]=CHR$(255-NUM(Y$[Y])) @ NEXT Y
55 CHARSET CHARSET$&Y$ @ DISP @ NEXT X
60 GDISP CHARSET$[390] @ BEEP
```

Alternate Character Set in Programs

Prompts for input, title lines, and warnings have more impact when displayed in inverse video. Once the characters are in the program, they will be displayed in the alternate character set whenever one is assigned. The easiest way to enter these characters into a program is to assign them to keys. The easiest way is with HI$ (or 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 "?",HI$("?");
```

Or...

```
DEFKEY "?",CHR$(128+NUM("?"));
```

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 its extreme right or a null after the end. This does not work with all other HP BASICs; on the HP-75, for example, it brings back the old string.

```
10 DIM X$[200] @ X$[200]=" "
10 DIM X$[200] @ X$[201]=""
```

Centering a String

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

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

DATE$

There are several ways to extract the day or month number from DATE$. However, 71's with VER$ "1BBBB" or earlier have a random and rare bug which may cause an unexpected error when concatenating DATE$. The problem is time-related, and may not show up in hundreds of tests, only to happen repeatedly other times. For this reason, you should assign the DATE$ to a string variable before concatenation. Even if you have a later 71, in order to assure that your program will run on earlier machines, it's best not to concatenate DATE$.

```
X$=DATE$[4,5]  ! sure to cause an error occasionally.
X$=DATE$ @ X$=X$[4,5]  ! correct.
```

Lowercasing a String

```
10 SUB LWRC(X$) @ FOR L=1 TO LEN(X$) @ X=NUM(X$[L])
20 IF X>64 AND X<90 THEN X$[L,L]=CHR$(X+32)  ! is it in the range?
30 NEXT L @ END SUB
```

NUM

Only the first character in a string or substring is significant to NUM. NUM(X$[5]) will suffice, the second subscript (as with X$[5,6]) is unnecessary.

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 listed as both a SUB and a DEF FN; a BASIC keyword called REV$ is listed in the Assembly language section. The operation is similar in all versions. These first examples use only one string variable and are slower than the same operation using two strings. REV2 is similar, but uses two string variables.

```
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

70 SUB REV(R$) @ FOR L=1 TO LEN(R$)/2 @ P=LEN(R$)+1-L
80 R=NUM(R$[L]) @ R$[L,L]=R$[P,P]
90 R$[P,P]=CHR$(R) @ NEXT L @ END SUB

10 SUB REV2(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

This FN rotates the string left by the number of characters specified by X.

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

Trimming Leading Spaces

```
100 IF Q$[1,1]=" " THEN Q$=Q$[2] @ GOTO 100  ! trim leading spaces.
```

Trimming Trailing Spaces

```
100 IF Q$[LEN(Q$)]=" " THEN Q$[LEN(Q$)]="" @ GOTO 100 ! trailing spcs.
```

Waiting For a Key

KEYDOWN and KEY$ give us the ability to wait for single keystrokes without using INPUT, then use the key within the program. KEYDOWN returns one if the specified key is down or zero otherwise. Used without a parameter, KEYDOWN checks to see if any key is being pressed. The program is 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. By the way, you can tell when your 71 is in the low-power state by placing an AM radio near it and tuning it until the 71 causes interference. When the 71's waiting for a keystroke, you can hear a little blip of a heart beat when the cursor

flashes. And when it's busy, such as in a continuous loop, the beat changes to a hummingbird-like pace.

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. KEY$ is easier to use than KEYDOWN, but will return a null string for no key presses; we still have to use it in a loop. Several LEX files contain the keyword KEYWAIT$ which does go to low power, waits for a keystroke and, unlike KEY$, always returns a string. Many plug-in ROMs have the function, you may already have it since it isn't always documented. KEYWAIT$ returns a string we can use with POS or NUM for fancy branching. This will work best with single character key definitions.

```
20 DISP "Option :X/Y/?" @ GOSUB CHR$(POS("XY?",UPRC$(KEYWAIT$))+65)
```

This routine branches to label 'A' for the wrong keystroke, or labels 'B', 'C', or 'D' for the correct key; no error trap is needed. Since the program label need not be the same as the keystroke, any key can be used.

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 'MAINLOOP': K$=UPRC$(KEYWAIT$)
510 IF LEN(K$)=1 THEN GOSUB CHR$(POS(T$,K$)+65) @ GOTO 'MAINLOOP'
520 IF LEN(K$)=2 THEN GOSUB CHR$(POS(T2$,K$)+65)&"2"
530 GOTO 'MAINLOOP'
```

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
```

If you are using ON TIMER# interrupts, you'll want to be sure not to use KEYWAIT$ or WTKEY$ because the timer won't interrupt the function. If the timer goes off during KEYWAIT$ (or even INPUT), it will be ignored until the function terminates.

PEEK$s & POKEs

The 71 is a nibble oriented machine; many operations which take a full byte on 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.

```
X$=PEEK$("2E3FE",1)  ! Returns value of 1 nib from specified address.
POKE "2E3FE","A"  ! Sets the nib at that address to "A".
```

Many of the routines in this section use REV$. If you do not have this function available in your 71, it can be simulated with the REV sub-program in Chapter 10, "BASIC Programming Hints." REV$ is called REVR$ or REVRS$ in some LEX files.

Let's look at System RAM. This is information the computer needs (such as the current PWIDTH) and 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. A chart at the back of this book lists most of system RAM.

For these reasons DTH$ and HTD aren't going to give an accurate conversion when the nibs are reversed. 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 set the WIDTH to 96, then PEEK the two nibs at DWIDTH(2F94F).

```
WIDTH 96                          ! Set the display width.
PEEK$("2F94F",2)                  ! Then PEEK at it.
                                  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 HTD$ always returns a five character, right justified, string.

```
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, corrupting (as they say) the data that lives there. The following shows what would happens if we POKEd DTH$(96) to location x in RAM.

Action	Returns
DTH$(PEEK$(x, 1))	"0"
DTH$(PEEK$(x+1,1))	"0"
DTH$(PEEK$(x+2,1))	"0"
DTH$(PEEK$(x+3,1))	"6"
DTH$(PEEK$(x+4,1))	"0"

POKE places the first nib at the address, the second at the next nib higher in memory, and so forth, for the full length of the string. For a POKE-able 96 in HEX, we first trim leading zeros, then reverse what's left, something like:

```
rev$(DTH$(96)[4])
```

System RAM

The first item on each line of the System RAM chart is its five-digit (hex) location in RAM. Each location has a four to six character symbolic name. These names identify the locations in documentation, and are used in equate tables for the Assembler. The third item is the number of nibs reserved for that utility, and finally, a few 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 unintelligible, and various display flags (like PRGM and the shift annunciators) may be lit at odd times. The following 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.
POKE "2E357","63"▒                ! Bold.
POKE "2E357","23"▒                ! Bold Top.
POKE "2E357","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.

```
HTD(PEEK$("2E3FE",1))▒            ! Read the contrast setting.
```

The first value in the example is "2F3FE", 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. This sets the contrast setting to 10 ("A" in hex).

```
POKE "2E3FE","A"▒
```

```
! Set contrast to 10.
```

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

The 71 stores up to the last 15 keystrokes 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 at the beginning of the buffer.

KEYPTR (2F443) Tells us how many keys are in the key buffer. If your program has been busy for awhile, the user might think it has died, and may start pressing keys. Later, the program stops for an INPUT and those keystrokes come back to haunt. You can keep this from happening by POKEing a zero at KEYPTR before the INPUT statement.

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 example exactly as written, without extra spaces, to see how the key buffer fills.

```
POKE"2F443","E"▒
```

```
OKE"2F443","E"▒
```

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

Setting CMD 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 is in several LEX files. This program limits it to fifteen entries because of peculiar things which can happen with the full sixteen. CALL the program with, for instance, CALL CMDSTK(10) to set it to ten entries.

This program is 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 inter-

rupted 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.

```
10 SUB CMDSTK(X) @ POKE "2F441","F" ! disable ATTN key.
19 ! make max cmds 0< X <16, find the command stack.
20 X=MIN(15,MAX(1,IP(X))) @ A=HTD(rev$(PEEK$("2F576",5)))
29 ! create empty cmd entries
30 DIM S$[X*6] @ FOR Y=1 TO X @ S$=S$&"000300" @ NEXT Y
40 E$=rev$(DTH$(A+X*6)) @ POKE "2F580",E$&E$&E$  ! set stack pointer.
50 POKE DTH$(A),S$ @ POKE "2F976",DTH$(X-1)[5]  ! blank commands in buf.
60 POKE "2F441","0" @ END SUB  ! enable the ATTN key, bye.
```

Display Devices

If there is a display device active, 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, it will cause an error when the function is encountered, and if the loop is broken, everything will get hung up waiting for the HP-IL ROM to realize it. Regardless, these operations are quite time consuming. This following 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 active the 71 will generally clear bit 2. This causes keeps the 71 from re-displaying the line when inserting and deleting characters, a unkempt looking affair when the 71 is connected to a terminal.

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

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

We can use this information to determine if a display is active, and where it is. In the 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. The information won't be accurate if nothing has been displayed since the loop has changed, though this is very unlikely.

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

Checking ON TIMER

You can see if timer #1 is set by checking if the address at TMRAD1 (2F697) is non-zero, then TMRIN1 (2F6A6) has the setting. Usually it's enough to see if a timer is set; the examples return non-zero values if a timer is set.

```
'CKTIMER1': X=HTD(PEEK$("2F697",5)) @ RETURN  ! timer # 1.
'CKTIMER2': X=HTD(PEEK$("2F69C",5)) @ RETURN  ! Timer # 2.
'CKTIMER3': X=HTD(PEEK$("2F6A1",5)) @ RETURN  ! Timer # 3.
```

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 some 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:

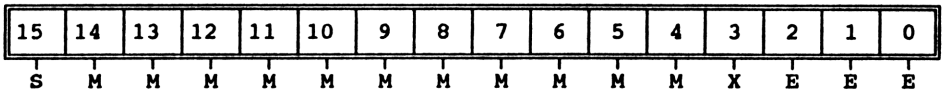
```
POKE "2F6E9","F"▒  ! Set flags 0-3.
POKE "2F6E9","0"▒  ! Clear flags 0-3.
```

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:



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 various 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) equivalent 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. 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

Since the 71 works with nibbles, an operation which may take full bytes on other 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. This 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 Use

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 this PEEK returns "0", other values mean there is one. This is not part of System RAM, but memory allocated for the Card Reader.

```
PEEK$("2C014",1)▒                    ! Is there a Card Reader?
```

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 36 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▒                   ! Create the file.
ASSIGN #1 TO TEST▒                   ! Open it.
PRINT #1;0,0,0▒                      ! Write to it.
```

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

```
Q$="QUACK!"▒                         ! The string.
CALL SDTEXT(Q$)▒                     ! Call the program.
```

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 that it's there:

```
POKE DTH$(A),Q$                    ! Replace the old record.
  RESTORE #1 @ READ #1;X$          ! Read then display the record.
    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$="HP-71" @ CALL SDTEXT(Q$)       ! Call the program.
  POKE DTH$(A+16),Q$               ! Write the string to SDATA file.
    RESTORE #1                     ! Move pointer to start of file.
      READ #1;Q$,A$,Q              ! Read the string and a number.
        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

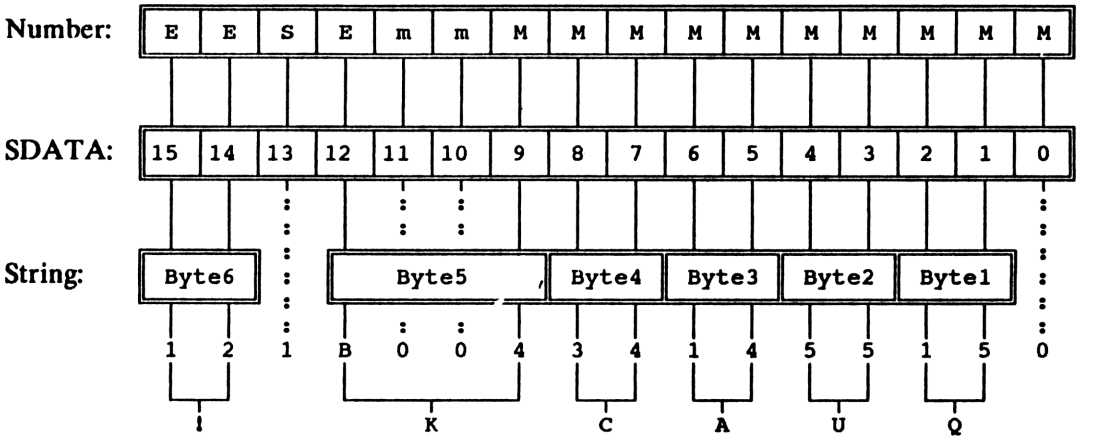
The string passed to SDTEXT must be DIMmed at least 16. Only up to the first six characters will be returned, in a sixteen character string suitable for POKEing into an SDATA file. Of course, this can be used as a subroutine in a program to save time and memory. If this routine is added to a program 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 the HP-41 uses for sign of the exponent). 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.



"QUACK!"	121B004341455150
PI	0000953562951413
PI(HP-41)	0000004562951413
9E27	7200000000000009
.23456	9909000000065432
-10	1090000000000001
"HP-71"	001100373D205840

Converting From Other BASICs 12

As BASIC has grown, it has followed divergent paths. Microsoft (registered trademark of Microsoft Corp.) is the dominant force because of shear number of units sold. Microsoft introduced BASIC to small computers (recorded on paper tape). There are probably more computers running Microsoft BASIC than any other implementation, or any other language. In this discussion we'll lump BASIC together into two camps: HP, and everybody else. Apologies to Dartmouth (where BASIC originated) for this generalization. A third category, modern BASIC compilers, resembles Microsoft, but without line numbers. Most of this section applies to IBM PC, APPLE, TRS-80, Atari and Commodore. Sinclair and other proprietary variations are intentionally "included out." We will also cover 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.

Examples 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. HP BASIC is easier to read because of the consistent program formatting, and because of the highly mnemonic keyword names (some MS keywords are just plain goofy). 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.

Round-off errors are often extreme with MS, while the 71 uses quite accurate BCD routines. Be sure that a program is not expecting the sloppiness of MS to limit number of loops or to intentionally induce an error. Programs compiled in Borland TurboBASIC (trademark Borland) and Microsoft QuickBASIC version 3.0 (and later) may use full 80-bit precision for floating point.

Variable names often end with a character signifying the variable type. Interpreters generally allow a single variable of a given name, while compilers accept multiple variables with the same name as long as the suffix is included.

X$	A string.	X!	Single precision.
X%	Integer.	X#	Double precision.
X&	Long Integer.		

Variable names are often quite long with MS. Expect variables names like COUNTER or DAYS. These are not keywords, 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. Starting with version 4, QuickBASIC no longer requires user defined functions to begin with "FN".

HP also uses "&" (ampersand) to concatenate strings where MS uses "+" (plus), the same operator used with numbers. Even the plus is often optional.

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

The two most obvious differences are string functions, and the use of the "@" (commercial at) instead of ":" (colon) to delimit statements on concatenated lines.

Graphics and Escape Sequences

HP-71 BASIC does not have graphics functions (except GDISP and GDISP$, but they are quite unlike Microsoft). COLOR and SCREEN can often be ignored, though the rest can cause problems. When you run into programs with the following graphics keywords, it's probably best to send up a white flag.

CIRCLE	COLOR	GET	PRESET	SCREEN
CLEAR	DRAW	PCOPY	PSET	VIEW

Most escape sequences are similar with one exception which will require major surgery. Moving the cursor to 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 specifying 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$(27) followed by the percent character (%) followed by col then row specifier. The coordinates begin at zero for col 1 and row 1. The first row is CHR$(0), the second is CHR$(1). The scheme used by MS assures that 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.

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

'' (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. The use of REM to designate a remark is universal in BASIC.

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

ASC

Used to recall the ASCII value of a character.

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

BEEP, SOUND

The BEEP statement takes no parameters and does a short beep of about 800 Hz. The two parameters for SOUND are frequency and duration. Frequency is expressed like HP, but duration is usually representing as clock ticks. These units are often 1/18th second for desk-top machines, or 0.02 second for portables.

```
MS:   SOUND 500,9 ' this is 9/18th's of a second.
HP:   BEEP 500,.5 ! this is 1/2 second.
```

CHR$(27)p, CHR$(27)q, COLOR

Some versions of MS BASIC use 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. The COLOR statement also changes the color of the current character.

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. On larger machines, CLEAR requires no parameter, since 64K is allocated for vars. In those cases CLEAR works like DESTROY ALL on the 71.

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, the prompt string is included within the default input. Even if an INPUT does not follow, the flashing cursor can be a distraction. 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";
```

CSRLIN, POS

CSRLIN returns the current vertical cursor position (row), and POS returns the horizontal cursor position (column) on the display. There is no equivalent. HP's POS keyword is the INSTR keyword in Microsoft BASIC.

DEFINT, DEFLNG, DEFSNG, DEFDBL, DEFSTR

Used in lieu of DIM. Create integer, long integer, single precision, double precision, and string variables. All vars beginning with the designated letter are of the specified type unless dimmed individually later.

Be cautious of programs using DEFSTR to create variables because the string variables do not require the dollar sign on many versions of MS BASIC.

EOF, LOF

EOF 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"). LOF returns the number of bytes in a file.

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. Many MS programs use FRE() even when they don't care about current memory, because it performs a garbage collection in the limited string space.

Some BASIC compilers running on computers with segmented architecture (like the IBM PC) use the parameter to tell it what kind of memory (string space, for example) you want to know about. Since the 71 has linear memory addresses, you can ignore the param type.

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

INSTR

Compares two strings and returns the a number representing the position within the match occurred 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 humane 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]
```

LOCATE, PRINT AT, PRINT

Print at is used to locate the cursor at a column and row coordinate. Read the section on Graphics and escape sequences for conversion.

NEXT

The loop counter variable in FOR-NEXT loops is often implied; this implicitly refers to the most recent FOR. If more than one variable is listed in a single NEXT statement, they are listed from the most deeply nested loop to the least.

```
MS:   FOR X=1 TO 10 : NEXT
MS:   NEXT A,B,C
HP:   FOR X=1 TO 10 @ NEXT X
HP:   NEXT A @ NEXT B @ NEXT C
```

Most versions of BASIC loop faster when using integers for the loop counters. The 71 is faster with REAL variables; check the program initialization routines to see if the variable has been declared an integer and is used only within loops.

RIGHT$

Returns the number of characters starting from the right-most end of the string.

```
MS:   X$=RIGHT$(X$,5)
HP:   X$=X$[LEN(X$)-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, therefore you would PRINT to your Terminal. "?" is an abbreviation 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 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"
MS:   PRINT "display this"
MS:   LPRINT "print this"
HP:   DISP "display this"
HP:   PRINT "print this"
```

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) implicitly creates one if it doesn't exist.

MS uses INPUT$ (syntax is Q$=INPUT$(<channel>,<#chars>) 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 lines in a 71 Text file is a finite length this function is not needed, it's 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.

HP-75 BASIC has fewer keywords than the 71, and can be thought of as a sub-set, conversion being fairly easy for primarily mathematical programs. Variable names may be used for scalar and arrays at the same time. 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 are more difficult to convert, and usually require STRINGLX or CUSTUTIL for functions not in standard 71 BASIC.

Transferring Files

Connect the 71 and 75 together using HP-IL. Edit the program on the 75 first to look for lines which will not be interpreted properly, then 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 turn these possible offenders into remarks first. 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.

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

```
! 71 is the printer.
! Put 71 in remote mode.
! Send file to 71.
```

Programs may also be transferred by 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 desired type, then 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. As a side benefit, the converted program will take 20-30% less memory on the 71 than it did on the 75.

File Handling

The file chain on both machines is operationally similar (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 specifying a file number) refers to the running program, not edit file, DELETE, RE-NUMBER, 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 nonexistent 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 furnished 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 existent) 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$() from the two previously mentioned ROMs may be substituted by KEYWAIT$ or a user function. The most 75-like key reading routine is WTKEY$. Read the documentation for the program and alter the keys accordingly.

HP-IL

The 75 I/O ROM solved several problems on the 75. If you find an older HP-75 program which doesn't use that ROM, a goodly portion of the program will do things like turning on or off the loop, setting up error traps, waiting for the loop to error when addressing it, and assigning devices. Pre-I/O ROM 75 owners were known to spend some time waiting for the loop to hang-up and issue the beep and "ERROR: Loop Timeout." Once you've determined the purpose of the code in these programs, you can most likely eliminate it.

Devices (display, Disc, etc) are always referenced 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 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 the number 9 they next use letters beginning with A (:M1,:M2...:MA,:MB...).

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

Assembly Language 13

Assembly Language Introduction

This chapter introduces Assembly Language programming on the 71. Even if BASIC solves your programming needs, you might find this section to be interesting reading to get a feel for the interaction between interpreted BASIC, and the actual Machine language it invokes. We'll deal with Assembly language at a fairly high level, which should get the average user started writing functions without having to purchase any of the IDS (Internal Design Specifications).

At its lowest level, a computer isn't even a very good calculator: It can't even multiply or divide properly. It adds, subtracts, makes comparisons and shifts 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 reads a Text file and creates machine code from commands within that source 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 its 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. We'll introduce writing functions for BASIC, and leave statements, FORTH primitives, polls and interrupts to the more adventuresome users with access to all volumes of the IDS (Internal Design Specifications).

HP-71 LEX files 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, we'll focus LEX files because of their versatility.

Source Files

We write Assembly language as Text files, then feed them to an Assembler which interprets them, and 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.

Comments are a vital part of writing Assembly code, so the source text file is often quite large. One LEX file of about 800 bytes has a source file of approximately 18k with remarks, which trims to 9k without remarks. If memory is at a premium, write and debug the file in sections then do the final assembly from a Disc (or RAM Disc) based file containing 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 Saturn processor is a descendent 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, gives a considerable speed gain over the 41. It's a highly evolved processor, with four 64-bit working registers handling BCD (binary coded decimal) math with reasonable speed. There are four main working registers and five 64-bit scratch registers in addition to two 20-bit (5-nibble) data pointers and nine status registers. It looks as much like a math co-processor as a 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. 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. Addresses are always referred to in 5 digit hex in BASIC and Assembly language; the 71 uses linear addressing, not the segmented architecture found on many desktop machines. 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 moving data around, it's necessary to be familiar with the CPU Registers.

Working Registers

Registers A and C are the most versatile because they are also used for memory access. A is often used to pass hex values (usually in the A field of A) 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 is 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 accessible through the C register; there are no operations to move data between D and A. It's 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 these scratch registers 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. Remember, interrupts occur when you least expect them, such as when the user presses a key.

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 appropriate instruction (such as C=DAT0) loads C or A with the data. Since they can be incremented and decremented quite easily, they are also useful as loop counters.

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 when you no longer need them.

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. Statements 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 the address 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 its 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.

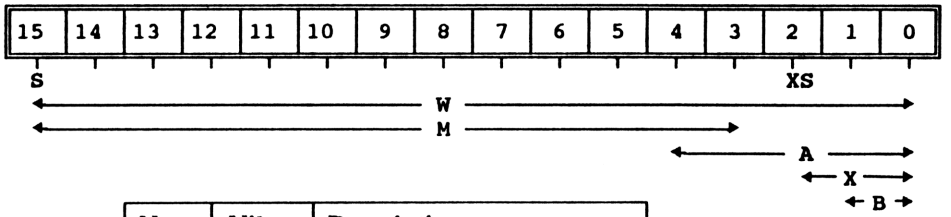
P Pointer

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.

WORKING REGISTERS

Reg	Size	Purpose
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.

FIELDS IN WORKING REGISTERS



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.

LEX FILE REQUIREMENTS

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

Remarks	Size	
LEX ID	2 nibs	
Lowest token #	2 nibs	
Highest token #	2 nibs	
Next LEX table link	5 nibs	
Speed table flag	1 nib	] 1 nib if no Speed Table.
Opt. speed table	78 nibs	
Speed table flag	1 nib	
Text table offset	4 nibs	
Message table offset	4 nibs	
Poll handler offset	5 nibs	
Main table	9*(total_keywords)	
Text table	3*(total_keywords) + 2*total_chars + 3	
Message table		
Poll handler code		
Exception code		
Next LEX table (optional)		

Assembling the LEX File

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 easy as BASIC is to you now, 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, 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' the LEX file name
       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 parameter 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 before Assembly, 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 then return to BASIC.

```
" REVTEXT" ASSEMBLE▒
```

```
! Assemble the LEX file.
```

```
BYE▒
```

```
! Get out of FORTH.
```

Turn your 71 off then back on. Now test our keyword.

```
REV$("ABCDEFG")▒
```

```
GFEDCBA
```

REVLEX is listed in the form used for source files by the Assembler. The first seven spaces on each line are reserved for labels; lines without labels are filled with spaces up to the mnemonic (the OP or pseudo-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.

Any display format wide enough to allow for the mnemonic and modifier on the same line is sufficient. The source code may exceed 500 lines, and there are no line numbers; it can be difficult to keep track of where we are in a large file. A screen oriented Text Editor with a width of at least 40 columns is the easiest way to write source files. When using the built-in LCD, PLIST the file often.

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 defined in it) reduced to about 1K.

```
3800 SHRINK
```

```
! Reclaim some memory used
! by the FORTH environment.
```

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 numbers.

```
= # ' ( 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 referenced 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 parameter 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 covered in Vol I of the IDS.

EQU

All references 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 reference to the label designating the location within the file where the actual code for the function lives.

CHAR

The characterization nib tells the operating system what kind of keyword lives at the above referenced 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:

	Description
Bit 0	Legal from keyboard.
Bit 1	Unused.
Bit 2	Legal after THEN/ELSE.
Bit 3	Function is Programmable.

KEY

Each keyword is listed quoted following the pseudo-OP KEY. Only uppercase letters and numbers from 0-9 are allowed. 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 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). While they don't cause conflicts, HP will not allocate keywords ending with a question mark (because it's confusing to read).

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, 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.

Each keyword for commercial distribution must be 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 keyword names may be used. In fact, mainframe keywords 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, use only keywords which won't be confused with mainframe keywords. For example use "RTN" instead of "RETURN" or "NXT" for "NEXT", or a contraction such as "CLFLS" (which is an allocated keyword in WBLEX) for "CLOSE FILES." To minimize conflicts, HP restricts two-character keywords, and requests that the keyword be as long as possible. If you're keyword does the same thing as a simple PEEK$ or POKE, HP will probably not allocate it.

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 qualifies the operation when the user enters the keyword. With statements, the LEX file must contain code to make sure the user inputs 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 71 parse our functions, we must answer three questions:

- Does it return a number or a 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 code begins 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 preceding 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.

Types of Parameters

In addition to telling the 71 how many parameters, we can tell it what kind. The nib(s) preceding the two nibs describing the number of parameters tell the 71 if parameters are strings or numbers. "8" designates a number, "4" is a string. The mainframe function FLAG can use either one or two parameters. 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 parameters 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 its task, a function never worries about it. What a relief!

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:

Register/Field	Description
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, C(15, the sign field) contains the actual quantity. We'll expand our example from above to see how we will handle the situation with either one or two parameters. Loading the quantity into P 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 its 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 toward 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 in BASIC.

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. This first nib represents:

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.

Low Memory		High Memory
Exp	Mantissa	Sign

Complex Numbers

A complex number is exactly like the format of two simple REAL numbers, but preceded 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.

Low Memory					High Memory
Imaginary Part			Real Part		
Exp	Mantissa	Sign	Exp	Mantissa	Sign

Strings

Strings begin with "F0", followed by five nibs representing its 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 its end at the end of the string header. The System entry REVPOP reverses a string and return its 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-existent string array will begin with "F8" and will have zero length, but will still have 16 nibs for the header

Low Memory				High Memory
"F0"	Len	Address	MaxLn	...The String

Array Descriptors

Arrays do not get placed on the Math Stack; they may not even fit, so why even try. The address is the location in RAM of the array.

Low Memory				High Memory
1	1	1	8	5
t	#	DIM	Lengths	Address

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 address pointer points to the array. To calculate data address of the variable, subtract the relative pointer value from the address of the relative pointer. See entry point RECADR.

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, functions are restricted to four levels under most circumstances. Also, how does it handle errors, and control will 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 registers 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 contains a listing of their requirements. In effect it is a much larger (and even less organized) version of the Entry Point list 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 better organized and easier to use. The advantage of having Vol III is to check for errors and omissions 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 common uses of LEX files. These functions could be written several ways, they are just an example of one method.

REV$ and CLFLS are untapped mainframe routines, SAMPLE does little more than add keywords to them. 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 its header. For that reason it exits through EXPR. EXPR 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$ (or HI$ in WorkBook71) calls for a single string parameter 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 shown 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's often more convenient to use than KEYWAIT$. A BASIC version of KEYWAIT$ is demonstrated in the BASIC section.

```
        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 key code to ASCII.
FNRTN1 EQU    #0F216   Return from a function.
CKSREQ EQU    #00721   Used by WTKEY$ to check 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.
       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$' Wait-for-a-key, return ASCII, not the key code.
       TOKEN  5
       ENDTXT          * End of the text table *
       REL(5) decom    Point to decompile routine for statement.
       REL(5) par      Point to parse routine.
clfls  P=     0        * CLFLS - do ASSIGN#n TO * on everything *
       GOSUB  saveD1   Copy D0,D1 to R2 (sub is at end of this file).
       GOSBVL CLOSEA   Call the mainframe routine.
       GOSUB  getD1    Restore D0,D1 from R2.
       GOVLNG NXTSTM   Bye.
decom  GOVLNG OUTELA   Decompile for clfls. Pointed to by REL(5) decom.
par    RTNCC           Parse for clfls. Pointed to by REL(5) par.
       NIBHEX 4        Param is a string.
       NIBHEX 11       Minimum and maximum of one parameter.
hgl    SETHEX          * HGL$($) Set high bit on all chars.
       GOSUB  saveD1   Save D0 & D1 pointers to R2.
       GOSBVL POP1S    Get string stats D1 points at start of string,
*                      A now =string length.
       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          else move pointer to hi nib of current char.
```

```
        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 D0,D1 to beginning of string header,
        GOVLNG EXPR       then exit.  bye.
        NIBHEX 00       No params so no nib to describe param type.
wtkey   SETHEX          * WTKEY$   wait for a key, return ASCII.
        GOSUB  saveD1   Save D0,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  Key code 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 key code table.
        CD1EX           Index by key#
        C=C+A  A        Add this key to key code to offset into list.
        CD1EX           Key position back to D1.
        C=DAT1 B        Read ASCII character to C(B).
        D1=(5) FUNCR0   Point D1 to scratch buffer.
        DAT1=C B        Put key in buffer.
exit    D1=D1+ 2        Enter here to move anything at FUNCR0 then exit.
        P=     0        Just to make sure P=0.
        LCHEX  FF       "FF" after key in buffer for end of string char.
        DAT1=C B
        GOSUB  getD1    Restore pointers from R2.
        LC(5)  FUNCR0   This is needed by BF2STK.
        ST=0   0
        GOVLNG BF2STK   Move this buffer to math stack, exit.
cksreq  GOSBVL CKSREQ   See if KEYBOARD IS pressed a key.
        GOTO   nxtkey
        NIBHEX 4        Parameter is a string.
        NIBHEX 11       Min and max of one param.
rev     GOSBVL REV$     * REV$  same thing used everywhere *
        GOVLNG EXPR
saveD1  R0=C            Copy D0,D1 to R2
        CD1EX           Uses R0 for scratch so it doesn't trash C.
        D1=C
        GOSBVL CSLC5
        CD0EX
        D0=C
```

```
        R2=C
        C=R0
        RTN
getD1   R0=C            Restore D0 and D1 from R2; uses R0 for scratch.
        C=R2
        D0=C
        GOSBVL CSRC5
        D1=C
        C=R0
        RTN
```

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's 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 to back up everything in the computer on Disc or Cards.

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), then call your Text Editor from FORTH to see what was wrong with the line mentioned. If a part of the file is known to have bugs. you might complete Assembly, then test only those keywords which are completed.

A monitor, listing file, or DISPLAY IS PRINTER is invaluable to watch the Assembler because its messages will disappear from the LCD when the next line in the file is read.

Instruction Set

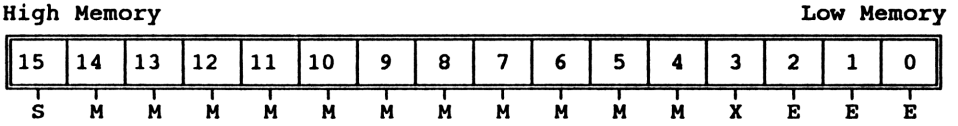
A listing of the Assembler Instruction Set is included at the back of this book. While all of the instructions are also listed in the FORTH/Assembler Manual, this listing will probably prove easier to use because it requires little page flipping. Many operations will take one nib less memory when specifying the A field, this is noted in the nib column. If two values separated by a comma are listed, the A field version will be one nib shorter; often we use the entire A field of a register when we really need only a few nibs, in order to save this code. For example, A=0 A takes two nibs, while A=0 X would require three. We've also listed the Pseudo-OPs (that is, instructions for the assembler), and an example of the bare minimum required for the source file. Field specifiers are:

fs	Stands for field select.
n	Means a hex nib.
fsd	Means field select fs or d (number of digits).

Why is Everything Backward?

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.



Only the C and A registers are 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). Imagine there's 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 its 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. Respectively, 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, save one to, for instance, R3:

```
C=RSTK          * pop first return from stack to C(A).
R3=C            * copy C to R3 for safe storage.
```

And, when done...

```
C=R3            * recall R3 back to C.
RSTK=C          * push C(A) on return stack.
```

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 under 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. This table lists the variations (including conditional jumps) and number of nibs used to record the jump distance; the instruction itself requires one additional nib.

2 Nibs	3 Nibs	4 Nibs	5 Nibs
GOC	GOSUB	GOLONG	GOSBVL
GONC	GOTO	GOSUBL	GOVLNG
GOYES			

As you can see, there's memory to be saved by using the shortest jump possible. Conditional jumps require short jumps. At an average of 4 nibs per line of source code, a conditional jump can't leap any farther than about 30 lines of source code. GOTO and GOSUB are suitable any jump within the average LEX file, while GOVLNG and GOSBVL are used for calls to system entry points.

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. Since strings are often too large to fit in the scratch RAM 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, the remains of the string are not important and will get trampled the next time memory moves.

The two usual methods for getting information are POP1S and REVPOP. POP1S 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 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. Then we reverse the string, 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. We then shift what's left up a byte at a time, exiting 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 a REAL number on the Math Stack. 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. The easiest way to get a hex integer from a floating point number 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. To recap, 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 allocated 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 routines are nearly always statements.

The System RAM chart lists 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 scratch use by 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 **60 nibs the transform buffer)**

Leeway

There has been determined a minimum amount of memory for the 71 to be able to operate. The computer 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. The HP engineers 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	Total required 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 its 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, and 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

The SAMPLE LEX file 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. Usually we verify how Entry Points we'll call use the CPU registers, and try to use them similarly.

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, 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
```

This routine makes sure there is sufficient free memory to create a buffer:

```
MEMCKL EQU    #012A5   Assumes that D1 has been saved because MEMCKL uses D1
MEMERR EQU    #0944D   if not enough memory then it restores D1, then exits
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. Also, 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     'CKSUM'  * This file includes the keyword 'CKSUM$'        *
        ID      #5C      *(c) Copyright 1987  Richard E. Harvey   Ver:1.0 *
        MSG     0        * Does a summation of all of the set bits in a   *
        POLL    0        *   string of up to 31 characters and returns a  *
AVMEMS  EQU     #2F594   *   one char string.                              *
POP1S   EQU     #0BD38   * Syntax:   X$=CKSUM$(X$)                        *
STKCHR  EQU     #18504   * This file demonstrates the slippery sticky bit *
ADHEAD  EQU     #181B7   * experiment with other ways you can do a bit    *
        ENTRY   cksum$   * summation in 71 Assembler. Hint: Most checksums*
        CHAR    #F       * use carry instead of the sticky bit.           *
        KEY     'CKSUM$' This is not an allocated keyword name!!!
        TOKEN   6        Scratch token number in decimal.
        ENDTXT
        NIBHEX  4        The parameter required is a string.
        NIBHEX  11       A single parameter is required.
cksum$  SETHEX           Make sure the machine is in HEX mode.
        GOSBVL  POP1S    Pop the string from the stack.
* we now have: A(A)=string len in nibs, D1 points at end of string, P=0
        CD1EX            Move D1 past the first character in the string.
        C=C+A   A          add the length of the string
        D1=C               place the number back in D1
        R1=C             Save this pointer to R1 to use when we exit.
* if the string is longer than 62 nibs then use just the first 62.
        LCHEX   0003E    Place hex equivalent of 62 in C(A)
        ?C<A    A          if string len is over 62
        GOYES   undr62       then use 62 (the value of C(B))
        C=A     B            else use the whole string (the value of A(B)).
undr62  D=C     B          move string length to D(B).
        A=0     B          zero the checksum counter.
loop    ?D=0    B        If no string is left then exit
        GOYES   exit
        D1=D1-  2        Decrement the string address by two nibs.
        D=D-1   B        Decrement the string length counter.
        D=D-1   B
        C=DAT1  B        Read two nibs into C(B).
* calculate the number of 1's.
        P=      8        P is our loop counter, set to 8 for 8 bits.
bit     SB=0               clear the sticky bit.
        CSRB               shift C(W) right one bit.
```

```
       ?SB=0              is the sticky bit clear?
       GOYES  clear         then ignore this bit.
       A=A+1  B             else add one to the checksum.
clear  P=P-1              decrement the counter.
       ?P#    0           any more bits left?
       GOYES  bit           go back for more
       GOTO   loop          else exit with P=0 and A incremented by set bits.
exit   D1=(5) AVMEMS    Get current available memory start.
       C=DAT1 A         Read current (avmems) to C(A).
       D=C    A         Copy (AVMEMS) to D(A) for STKCHR and ADHEAD.
       C=R1             Restore D1 which was saved in R1 when we started.
       D1=C
       C=A    B         Copy our checksum byte to C(B).
* R1=start of string. D(A)=(AVMEMS), D1=points to stack, C(B)=character.
* this routine decrements D1 (the stack pointer) by 2 nibs.
       GOSBVL STKCHR    15-35 place the character on the stack.
       P=     0         Set the pointer to zero (tho it already is).
       ST=0   0         Means don't return from the following routine.
       GOVLNG ADHEAD    14-13 add the string header, return to basic.
```

Non-obfuscating Programs 14

This chapter is seemingly at odds with much of the rest of the book; endorsing form over function, the difference between deftly pounding form into raw marble and creating mortar. First, let's look at an approach to program planning, then spend a few minutes on the user interface.

BASIC, many argue, encourages plan-as-you-write programming; start on line 10 and continuing, akin to a novel, to the thrilling conclusion. Then do the re-write until there's a happy ending. The happy ending can come a little faster and smoother with planning. Surprisingly, many programmers who wouldn't think of assembling a child's toy without the instruction manual, begin multi-hundred line programs without a plan. The longer you spend planning, the less time you'll need to spend programming and correcting errors. We start by asking ourselves questions like:

- What is the purpose of the program? What are the tasks? Try to state the program's job in a single sentence.
- What data do we need? How will we get the data?
- What will we do with the data? How will we store the data?
- How can we maintain that data in the most efficient and flexible manner? Remember, as the project develops, the data requirements may change.
- Who is the user and what user interface is most suitable?
- What are the individual tasks the program will do?
- Have we planned for program maintenance? The program will go through revisions and may be the basis for another program.

Your project may have other considerations like hardware requirements. After answering these questions, we're ready to attack the project, dividing and conquering. Many programmers next write an outline or flow chart, others use pseudo-code, which is simply a series of remarks describing what the code (when it is finally written) will do. At this point, you might review the original questions to insure that they've been answered. Then begin to assemble the program, not worrying that modules are missing or incomplete. There are as many program development styles as there are programmers; once you've developed your own style, programming becomes more efficient--and restful.

You may want to begin modules within a program with different groups of line numbers; main input might be on 1000-2000, and subroutines on 8000-9999. It's generally a good idea to number lines by 10's or 100's so the inevitable changes may be inserted without having to re-number the program.

0001 - 0999	Initialize
1000 - 1999	Input data
2000 - 4999	Process data
5000 - 7999	Output results
8000 - 9999	Subroutines

While this table is by no means etched in stone, it's one way to organize a single purpose program. When you later make changes, it will be easier to find the problem code.

Modular Programming

A module is just a segment of code which you often use. We'll define a program module as any one of the following:

- Subroutines beginning with a label within current program file.
- SUB within the current program file.
- SUB in a separate program file.
- DEF FN user function in the current file.
- LEX or BIN file.

Call them procedures, atoms, routines, or functions if you will, the logic is the same: Divide the job into manageable pieces. Many examples in this book, such as the REV sub-program and the REV$ keyword form complete statements and utilities.

Depending on your application, modules within the main program file, or in an external file, may be preferred. More than likely you'll use a combination of both. Programs for distribution usually maintain all modules within the same file to simplify use. A computer used for a dedicated purpose, say data analysis, is the ideal home for a separate library program file. Creating a library of sub-programs is similar to writing new LEX files with the Assembler; the procedures become instantly available for other programs, or easy to customize and add to a program.

For example, you may often need to sort small amounts of data, so you place a bubble sort routine in a program file:

```
900 SUB BUBLSORT(E,X(),X$())  ! bubble sort; ignores zero elements.
910 FOR J=1 TO E @ FOR I=E TO J+1 STEP -1 @ IF X$(I)>X$(J) THEN 930
920 T=X(I) @ T$=X$(I) @ X(I)=X(J) @ X$(I)=X$(J) @ X(J)=T @ X$(J)=T$
930 NEXT I @ NEXT J @ END SUB
```

We've written and carefully debugged the module; now, other programs can CALL BUBLSORT(TotElements,PointerArray,SortStrArray) with little development time and few worries about bugs. And, this routine ignores the zeroth element in arrays, so OPTION BASE is unimportant; it's often best to leave global machine status as it is in these routines.

To make modular programming easier, we begin each internal module with a label, and only reference line numbers within the module. In this way, we can re-number a module to make it fit in the next program wherever a contiguous group of line numbers allow.

Not all modules are called as subroutines. Your library might contain convenient forms of INPUT or IMAGE statements or formulas which are tedious to write from scratch. With modular programming, the last program you wrote becomes the seed for the next. Though one might have been a game and the next an accounting program, you'll find that most programs share many elements.

Remarks

The REM statement, or the more readable exclamation mark (!), make programs easier to understand and maintain. If memory is at a premium (isn't it always?), the remarks may be deleted when

the program is completed, but a spare copy with remarks will simplify later changes. An easy to use, though very non-standard, type of remark is the un-numbered line in a BASIC source Text file. We'll list techniques for using this special type of remark in Chapter 16, "Communicating with Other Computers."

A method to help keep programs readable while conserving memory is to separate modules in programs with a REM-line; memory cost is minimal. If you are planning to remove remarks later, they will be easier to locate if they are whole lines, not tagged on the end of a line of code. Be sure that no GOTOs or GOSUBs refer to remarked lines or you'll have to re-write these references before you can delete these remarks.

```
110 IF X THEN !
700 ! FOR X=1 TO INF
```

Variable Lists

As programs grow so does our use of program variables. A written list is invaluable to insure that X1 is X1 throughout, and doesn't become X0 some place. The list also helps spot unnecessary vars or over-DIMmed strings.

Saving Scratch

Consider variables we would use in a spreadsheet program. We need a pointer for the current column, current row, cell format and so forth. Since the program does so many different things, we can't afford to use separate variables for each section. We'll reserve scratch vars for use within the various modules, and for passing information between them. Simple var names (like A,B,C) use one byte less each time they are referenced than if they have an optional numeral (like A1,B2,C3), so we try to use single letter names. In our spreadsheet we might use:

C	Temporary column coordinate.
R	Temporary row coordinate.
L	Temporary loop counter.
X	General purpose scratch.

Designating these variables as scratch means that they may be used by any subroutine as local variables because they can generally be assumed to contain nothing important or "trash." Scratch vars also are used to pass information between modules; we pass the coordinates in C&R, the subroutine does its business, then returns the answer in, perhaps, X. Scratch vars conserve memory (by reusing), and guarantee that important variables aren't accidentally trampled.

Regardless of the program, you might use the same names for various classes of use. 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. Variable names I (the letter eye) and O (the letter oh) look similar to 1 (one) and 0 (zero) on many printers, and are best avoided, though many programmers use I to represent index or iteration.

User Friendly Programming

We've been talking about programs from a programmers point of view, now it's the users turn. The objective is to get the job done and provide a usable tool, not to show how clever we are. The most complex program often is the easiest to use; it places few restrictions on the user. Here are a few common courtesies.

- Leave modes alone or restore them. If your program alters key files, flags, FIX, OPTION BASE, startup sequence, or other global setting, restore them when your program ends.
- Don't make the users run their machine your way. Begin your program with a SUB statement and leave it open to custom configuration.
- Let the user decide file names and locations. Let the user use files on Cassette, even if testing proves that time or media wear are problems; technology moves on, and that Cassette may really be a RAM Disc.
- Minimize hardware requirements. Use error traps to insure that the program will run even without the HP-IL Module.
- Conserve memory to speed up program loading, allocation, and data manipulation. If applicable, offer small and large memory versions.
- Try to support a monitor, but don't require one.
- Only beep if you have to. Flashing lights and strange noises are strictly from the bad science fiction movies of the 50's. The computer is a tool; the last thing the average user wants is a nagging machine.
- Expect the worst and plan for it. Things happen. People press the wrong keys, the loop may be configured wrong or batteries may die.
- If a state is toggled and the user knows the current state, then just toggle; Y/N is annoying except as a confirmation. An otherwise friendly HP-75 Spreadsheet program asks if you'd like video updating enabled or disabled; obviously, one of those states is currently active, and the user is undoubtedly trying to change to the other or he wouldn't be in that menu.
- Ask the user for confirmation on major decisions. "erase everything:Y/N".
- If something is going to take a long time, keep the user entertained so that he doesn't think the machine has died.
- Anticipate execution time. Plan to process data on-the-fly or wait until after all inputs: Spread out the lulls, or give the user a coffee break.
- Be helpful. Offer default inputs and try to look-up the data for the user. When practical, use mnemonics for options.
- Be consistent. If you are using a command line or a menu "user interface," maintain it throughout the program.

Menus and Command Lines

The BASIC language is a command driven program; you're presented with a blank line and a flashing cursor, and you're expected to know what to do with it. It's worth the effort to learn, so you've read the owner's manuals, this book, and other books, magazines, and manuals on the subject. Now you can type complex commands and the 71 only occasionally gives you a bad time

about them. Many programs work with a command line; one Text Editor enters "command level" when you press ATTN, then waits for you to type a command. The advantage of this system is that you can accomplish several things in a long line of abbreviated command names, parameters, and such; merging files or search & replace are only a few cryptic commands away. Providing that you can remember all of the commands.

The antithesis is a Text Editor which offers a menu of available options whenever you press the ATTN key. You then press a single key representing a mnemonic of available options. The program immediately carries out the task, or perhaps falls into a second menu level. File utilities could, for example, be in the second level deep. This is an easy way to remember what to do in a multi-level program, one with, for instance, edit and command modes.

A single level program, like the Finance ROM, is an ideal example of a menu driven program; enter commands assigned to keys and labeled with a keyboard overlay. When the task is completed, the menu returns.

An easy to use program is preferable to a fantastic manual.

Accessories 15

First there was the calculator; only a few ROMs and single density memory modules were available. With technological (or marketing) progress, larger memory modules, new ROMs, and the Cassette drive showed up; every year or two, a new generation. 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

The 71 data storage picture has changed in the year and a half since the first version of this book was released: RAM is less than half the cost, though a lot of RAM still costs more than a Card Reader, Cassette or Disc Drive. Unless you use your 71 just as a calculator or only run ROM programs, you will need some variety of media for archival storage. In practical terms, the primary justification for investing in over perhaps 64K is if you can't carry a Disc or Cassette because of weight or damage considerations, and you need to carry several programs or a large data base. Since needs expand to fill available RAM, how much memory you need is a individual consideration. The more memory you have, the more personal and portable the 71 becomes. And unlike HP-IL devices, it's in the machine, not dangling from wires. Lets compare the cost of storing 1K on various types of media. These are approximate list prices as of January, 1988.

per K	List	Device
18.75	75.00	4K RAM Module.
5.00	160.00	32K Front port.
4.60	295.00	64K Card Reader port or front port.
4.12	395.00	96K Card Reader port.
3.87	495.00	128K Card Reader port.
.55	.70	Magnetic Card.
.08	9.50	Cassette.
.01		3 1/2 inch Disc.

Each of these alternatives have their advantages. If media cost were the only concern, everyone would use a Disc drive, and nobody would use a 4K front port module. 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 criteria. 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 may become lackadaisical about backing up files. The first version of this book was written with a standard 17.5K HP-71, as was the RAM version of WorkBook71; obviously a disc drive was more important than monster-memory.

How Much RAM Can I Add?

Everything within the 71 shares the same 512K linear address space. The Operating System, HP-IL module, System RAM, and an area reserved in high-memory used during configuration, account for less than 100K. Therefore, about 400K is the addressable limit for add-on RAM and

ROM. If you were to manage to connect more than 400K worth of "things" (ok, devices), the 71 would only recognize the first 512K.

Each of the six ports (including PORT 0) supports up to 16 devices, inviting RAM-cram. The author has used a 71 with 256K wired internally, with 128K in ROM modules in the front ports, for a total of 464K of RAM/ROM. If a 128K RAM module had been added in the Card Reader port, the 71 wouldn't have been able to find it; it would have exceeded the address space.

A 256K 71 takes little extra battery power and doesn't degrade performance for most uses. Deleting files in the middle of the file chain or inserting data in a file or gigantic array can take a minute or more. Building a string 100,000 characters long requires the 71 to have 200,000 bytes available to later use the string. This is because most operations will require the string to be copied onto the Math Stack for use (see Chapter 13, "Assembly language," for discussion of the Math Stack). A string array requires, at most, a single element to be on the stack for most operations. When working with large amounts of data, it's generally faster to use an array or fixed-size Data file than scalar variables or a Text file. This is because, in arrays and Data files, the memory has already been allocated, an there is less data movement.

RAM Modules

Currently, RAM is available in 4K, 32K and 64K front port modules, 32-128K Card Reader port modules, and 128-512K RAM Discs. Front port modules limit the number of ROM modules you may use. A great advantage of using any RAM module is to port-out blocks to keep access fast and CAT ALL uncluttered. Ported RAM is less susceptible (though not totally immune) to loss of data in a crash than main RAM. Remember, if you port-out your RAM modules, it will generally be in 32K segments; in this case a 4K RAM module may be a viable choice.

Third parties (and some hotshot users) sometimes wire modules inside of their 71s, to keep the front ports free for ROMs. Because space is limited, the first module will usually fit easily, though additional modules may require removing an internal RAM module. These mongerings will obviously void the warranty and can usually be avoided by using RAM Discs or port five modules.

EPROMs

Like ROM modules, Erasable Programmable Read-Only Memory (EPROMs) modules, available in 32K and 64K, contain permanent programs and data, and are virtually immune to erasure. However, EPROMs may be erased by a concentrated dose of ultraviolet light, then reprogrammed with an EPROM Burner. Programming requires expensive equipment, so most are done by programming services. When programs and data will not likely change, or the 71 will be used by "casual" (i.e. not 71 enthusiasts) users, not familiar with the gubbins inside the 71, EPROMs provide the most reliable storage. PROMs are similar, but are not erasable, primarily because they don't have the little window. Though suitable for one-time use only, they are less expensive than EPROMs and are better suited for commercially distributed software in volume.

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. Cards come pre-formatted with a single 650 byte file on each of its two tracks, and cannot be re-

formatted, but can be rewritten 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 over four 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, it's probably safe to assume it that 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

File handling on 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. COPY and INITIALIZE are the only commands many users will ever 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. The drives currently available are the Hewlett-Packard 9114B 3 1/2", HP82161A Cassette drive, CMT RAM Disc, and the IBM PC.

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 Vectra, HP-110 Portable and 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 an HP-IL/RS-232C interface. These drives are usually used for storage for 71 files, not for sharing with other machines.

3 1/2 Disc Drive

The HP 9114B Disc drive offers fast, reliable storage of programs and data. Even the most industrious pack-rat among us would tire before filling a single 9114 Disc; the average library of programs is unlikely to fill the over 600K capacity of a single Disc. More than likely it's under 100K, hardly enough to make the Disc drive break out in a sweat. 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. 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.

Early 9114A's had some design problems which have been corrected to large degree in the 9114B. The two main hurdles were that the statement PACKDIR didn't work correctly, and battery consumption was excessive. A new EPROM 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.

The standard 4-Ah, 6-volt 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, may 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 companies have developed battery eliminators for desktop use. A large external battery, say 10 Ah, can keep the 9114 alive for extended field use, though it could take three days to recharge when you get home.

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

Disc Media

HP was the first major company to use these sturdy and reliable 3 1/2" discs, and the industry has followed their lead. Double sided 3 1/2 inch Discs are becoming a standard, fast replacing 5 1/4" Discs on desktop machines, and are much easier to find than magnetic cards or cassettes.

The drive is designed to work with only double side certified Discs. 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. 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 warranted to be any good at all. Perhaps the second side failed a production test 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 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. Smoke particles lower the signal strength to 15% of normal amplitude; imagine what an inexpensive 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 when done.

Disc Drive Maintenance

Disc recording heads usually won't need cleaning until after 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 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/cardboard shipping protector) is partially ejected or there was no Disc in the drive. If you pull out the Disc before pushing the release button you will pull the heads out of alignment. 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 inexpensive insurance policy.

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 of continuous use), 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. A minimum of 500 accesses per file can be expected, though a tape file can usually be read reliably at least 1500 times. Re-wind tapes before removing them from the drive, and be sure to take up any slack (insert a pencil in the hub and turn) before inserting a tape into the drive, to minimize stress when the drive motor engages.

Cassette Drive Maintenance

Dirt is the main enemy of cassettes, making Cassettes unreadable long before their time is up. Keeping the drive head clean will greatly increase tape and drive life. 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 being unreadable. Keep the Cassette drive away from Monitors.

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, as is the case with most Nickel-Cadmium powered devices. 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 its way off of older units, leaving the drive useless. Usually the spring, spindle and washer can be slipped back on without necessitating 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.

RAM Discs

Living out on the loop and disguised as a Cassette drive, the latest new twist is the RAM Disc. The RAM Disc is not a new concept; basically, it's a covey of RAM chips and a 9-volt battery in a box slightly smaller than the 71. For the combat zone of life, away from battery chargers, a RAM Disc provides reliable off-line storage. Working completely in silicon, no motors or magnetic media, these are often the fastest HP-IL based storage. As of this writing, the HP Disc drive supports a

faster transfer rate, though the current RAM Disc does not have the start-up lag, making speed comparisons a toss up. Remember, they are of a finite size, so you will most likely also need another form of archival storage.

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. In February of 1988, HP announced improved ThinkJet print quality on regular bond paper. 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.

If you decide to use another brand of printer, be sure that it has the correct interface. 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.

The second consideration is connecting the printer. There are a number of HP-IL to RS-232 converters available, making it a case of wading through manuals and plugging the devices together. As of this writing, connecting the 71 to parallel printers requires an understanding of the processes at work and a parallel converter such as the HP82166A or PAC Screen Video interface. If you need to use a printer other than the ThinkJet, RS-232 will be somewhat easier to get started with than parallel. We'll cover RS-232 in greater detail in Chapter 16, "Communicating with Other Computers."

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, and will wrap text to the next line when it exceeds that width. The PAC Screen is an 80-column device which also supports graphics and can send those graphics to a parallel printer. A monitor can be very helpful for writing programs, or testing how a program will format data without having to print the data. When used with a monitor, the HP-71 becomes equal to or superior to a desk top computer.

Terminals

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 in Chapter 16, "Communicating with Other Computers."

Other HP-IL Devices

This is a partial list of HP-IL devices; not all are available at retail stores. There are other third-party devices not listed available.

Part Number	Description
HP 1630A/D/G	Logic Analyzer.
HP 2225B	ThinkJet Printer.
HP 2671A/G	Alphanumeric/Graphics Thermal Printer 8.5" paper.
HP 3421A	Data Acquisition/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 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 82166A	Interface Kit (discontinued).
HP 82166C	Interface Converter Kit.
HP 82168A	Acoustic Coupler. 300 baud, battery powered.
HP 82169A	HP-IB Interface.
HP 82402	Dual HP-IL Adapter (for HP-71 only).
HP 82905B	Impact Printer. Similar to Epson MX-80 (discontinued).
HP 82938A	Series 80 Interface.
HP 82973A	IBM PC Interface.
HP 9114A	Single 3 1/2" Disc Drive (discontinued).
HP 9114B	Single 3 1/2" Disc Drive.
HP 92198A	Mountain Computer 80 column Video Interface.
ADC71A	Interface Instruments Analog to digital converter.
CMT RAM Disc	128-512K RAM Disc; optional RS-232.
HP-IL A/D	Ocean Scientific A/D Interface.
Modem 300 Plus	Direct Connect modem&Bar Code Reader. Firmware.
PAC-Screen	80-col/graphics Video Intrfc with parallel printer port and mouse port by PAC Hardware GMBH (Germany).
RS-232/HP-IL	Battery powered RS-232. Firmware Specialists.
#111	HP-IL Repeater. Interloop.
#130	HP-IL Twinax Terminator. Interloop.
#200	HP-IL Step Motor Driver. Interloop.
#210	HP-IL IO Interface. Interloop.
SB10161A	Single Steinmetz&Brown 5 1/4" Disc Drive(discontinued)
SB10162A	Dual Steinmetz&Brown 5 1/4" Disc Drive (discontinued)

Communicating with Other Computers 16

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. HP-IB cables for example 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 principally used for printers. Much of the world accepts RS-232C, so we'll introduce a few applications.

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 as 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, 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 available in the FORTH/Assembler ROM, from the Users Library (now Solve and Integrate of Corvallis) as LEX file #03194-71-5, and with the HPILLINK program for the IBM PC. It adds the keyword KEYBOARD IS which operates like PRINTER IS, and ESCAPE, which traps escape sequences sent out by the external keyboard, and turns them into keystrokes the 71 can understand. With this LEX file, any device which you can connect to the 71 and does not operate as a controller, can be a supplementary keyboard, without affecting how the 71's own keyboard operates. If you don't own the FORTH/Assembler ROM, the KEYBOARD LEX file must be in a higher numbered port than the HP-IL module; the LEX file doesn't operate correctly in main RAM. This means that you must purchase a RAM module and FREE PORT it to hold the LEX file.

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 device can 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 the KEYBOARD LEX file. The first uses a NEC PC-8201A as the keyboard, both use the RS-232 at the default 9600 Baud rate. The second example uses an inexpensive Zenith terminal. 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. This wouldn't happen normally because the 71 empties the key buffer during 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 offer us exposure to Microsoft BASIC. These machines have been superseded by more advanced (and expensive) models with larger displays and better software, but we'll concentrate on the less expensive models.

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 the escape sequences and turns them back into 71 keystrokes. If we had used TELCOM, the cursor keys would send their keycodes, not the keymap the 71 wants.

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] works like `I/R`. BS is the same as `f-BACK`. [DEL] works like `-CHAR`, [|] (the vertical bar key) works the same as `g-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] the entire contents of that buffer will be sent to the 71.

The other keyboard characters work as on the 71, however, remember that any key which does not have a counterpart on the 71 will be ignored. The `f-LC` key on the 71 inverts the way the 71 interprets case, so that if you are set to lower case on the 71, 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. This program assigns arbitrary escape codes and some thoroughly non-standard key assignments; you may change them if you'd like to traditional escape codes. The Zenith Terminal program uses standard escape sequences.

```
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$(150)        ' | (vertical bar) cmd
6030 PRINT#1,K$; : GOTO 6010
7000 IF K= 13 THEN K$=E$+"&"              ' rtn    Waste time by falling
7010 IF K= 29 THEN K$=E$+"/"              ' left     through the rest of
7020 IF K= 30 THEN K$=E$+"2":GOTO 7500    ' up       the conditionals.
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)        ' Shift-left
7080 IF K=  6 THEN K$=E$+CHR$(160)        ' Shift-right, ctl-right
7090 IF K= 26 THEN K$=E$+"T"              ' ctl-down
7100 IF K= 20 THEN K$=E$+CHR$(162)        ' Shift-up
7110 IF K= 23 THEN K$=E$+"S"              ' ctl-up
7120 IF K=  2 THEN K$=E$+CHR$(163)        ' Shift-down
7130 IF K=  8 THEN K$=E$+"g"              ' back arrow
7140 IF K= 27 THEN K$=E$+"!"              ' esc
7200 GOTO 6030
7500 PRINT#1,K$; : FOR K=1 TO 450 : NEXT K : GOTO 6010 ' delay loop
```

A Terminal as a Keyboard

The KEYBOARD LEX file is as easy to use with a terminal as with computer, though inexpensive 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 g-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 TELCOM program 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.

Since the NEC and Radio Shack do not add a line feed (CHR$(10)) after the carriage return 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. The program on the 71 will have to be stopped manually when the NEC has completed its transfer, or enter "!END" then press [RTN] then press CTL J (for line feed) while still in TELCOM. These machines can be made to add a line feed while in TELCOM by a simple POKE. Be sure to restore the location to zero for normal use. From BASIC on the NEC, RS or Olivetti machine, enter one of the following:

NEC PC-8201A: POKE 62469,1
RS Model 100: POKE 63066,1
Olivetti M-10: POKE 63069,1

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 based file, 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, the size will automatically expand as needed; Disc based files must have their size specified accurately (or too large).

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$(KEYWAIT$))
45 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$(KEYWAIT$))
75 IF NOT X THEN 20
80 KEYBOARD IS * @ CLEAR :RS232 @ IF X THEN 110
90 INPUT "size:";L @ CREATE TEXT F$,L  ! create the output file
100 DISP "start transfer"
110 DISP "working" @ ASSIGN #1 TO F$ @ ON ERROR GOTO 1000
115 IF 1=F THEN 130
120 READ #1;Q$ @ OUTPUT :RS232 ;Q$ @ GOTO 120  ! loop until EOF error
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$,T) @ ON ERROR GOTO 9640
9605 DISP CHR$(27)&">"; @ CAT F$ @ T$=DISP$[1,32]
9610 IF NUM(T$[12])=32 THEN T$=T$[3]
9620 T=POS("TESDDABILEKEBAFO",T$[12,13])
9625 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$) : M=M+1 : IF NOT EOF(1) THEN 60
70 CLS : CLOSE : PRINT "Len:";X;",Lines:";M
80 PRINT "min file size:";(M*3)+X
90 INPUT "press [RTN]";Q$
100 OPEN F$ FOR INPUT AS #1
110 OPEN "COM:" FOR OUTPUT AS #2
160 Q$=INPUT$(1,1) : PRINT#2,Q$; : IF NOT EOF(1) THEN 160
200 PRINT#2,"!END" : CLOSE : MAXFILES=1
```

Display Devices

The HP 82163A (32 columns, 16 rows), HP 92198A (80 columns, 24 rows), and PAC Screen, are the only video interfaces available as of this writing. HP terminals are preferred over other terminals because of compatibility. If other computers or other brands of terminals 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 troublesome escape sequences are listed below.

HP Code	Operation
% col row	Cursor to Address (MS: Y col row)
Q	Insert Cursor
R	Replace Cursor
>	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, the terminal is echoing the transmitted data. Set FULL DUPLEX on the terminal to eliminate the double vision. If nothing at all is displayed, set HALF DUPLEX. If you are getting garbage on the screen, go back and make sure the parity, stop bits and other protocol requirements match.

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 display a line only 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.
```

Communicating with RS-232C

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," "serial port," or "Asynchronous Communications Adapter."

The Black Box

In order to "speak RS-232" you'll need an HP-IL/RS-232C interface (in addition to the 82401A HP-IL adapter). The HP-82164A cannot run on batteries or a car adapter, if this is important to you consider one of the third-party adapters. 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 low-level HP-IL commands.

An 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 physically connect them, so we will first discuss acquiring a cable. "RS" stands for "Revised Standard" and as such, it's

(usually) fairly easy to connect things together using it, though even under the best of circumstances it will be much more work than connecting a 9114B Disc drive. Many companies (HP included) have used the "standard" 25 pin connector (called DB-25) for other purposes, or other connectors 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 do Apple and newer IBM computers. This 9-pin connector is often used for monitor connectors and has even been used for power cords (imagine the zap!), though it is becoming a de facto standard for serial connectors. Usually, on the computer end, a DB-9 or DB-25 connector will be male, and monitor or parallel connector will be female. For these reasons it's often necessary to have custom cables made, and be sure what you're plugging into is really a serial port.

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'll 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 your local electronics parts 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 their ubiquitous electronics jock employee 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. A new generation of connectors has come around which no longer require soldering. Merely strip the ends of the wires, clamp the pins on with a pair of pliers, and insert the pins into a pre-drilled connector.

When you look at the front end of a male connector (the kind that's 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; 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 through 8 and 20. Look inside of most devices and you'll see that the other pins don't do anything, often the pins have been left off of a male connector. Because of individual companies *improved* standards, it's possible that some of these extra pins will have strange voltages going through them, so it's safest to not connect anything to them. The simplest circuit will use only 2 and 3 (that's what the salesman was telling you to switch). Pin 2 generally transmits data while pin 3 receives, and this is the problem: Many companies reverse the use of 2 and 3.

A chart on the back of the interface is usually all that you will have to refer to when making a cable. Your printer or modem manual will probably have some cryptic chart to show if all is normal. If it appears that everything is going to be pretty much standard, wire the cable with pins 2 through 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 reread 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 connect to another computer, leave it that way. For some devices you will have to open the interface and turn the plug around. A separate cable for each device will save you constantly opening up the interface to change that plug, or cable to change a wire.

Setting up the Interface

Once hooked up, it's time to configure the interface to the 71 and the device on the other end of the interface. Since RS-232 is versatile, it can interpret data in many ways. The HP interface does not save its configuration when you shut it off; you'll need a setup program (or key assignment) to run each time you turn it on. Some RS-232 Interfaces, such as CMT battery powered units, do remember configuration between sessions. Since the 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 HP 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.

```
10 REMOTE :RS232  ! A printer set-up program
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 with the REMOTE statement. Line 20 sends various commands as found on pages 37-40 of the HP RS-232 Manual. Look up the mnemonic for the commands, then enter them in the string, each separated by a semicolon, as many commands as are needed. In the example we used "P0" to set even parity and "SBA" to set the interface to 2400 baud. Line 30 restores everything to LOCAL mode then assigns the printer.

So, we've used REMOTE mode to tell the RS-232 how to communicate with the printer, and returned the loop to LOCAL mode, now the RS-232 becomes pretty much invisible and sends whatever we tell it to the printer.

The interface's default (how it works when first turned on) configuration will allow you to connect an 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. 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 its 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.

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 in the interface manual to send it in REMOTE mode. If you just can't get it to work consistently then go back and check the cable.

HP-IL to PC Interface Card

Many people use a separate computer for program development, then transfer the files to the 71 to be transformed into BASIC or Assembled. HP has seen this need and filled it with the HP 82973A HP-IL/IBM PC Adapter. This card plugs into a short slot on the PC and comes with software to connect it with the HP Portable Plus. The Link card *does not* come with software to connect the 71; HPILLINK is HP's extra-cost program to connect the PC and 71 to use the PC's display and keyboard. We'll discuss this program exclusively, though third-party packages are available. With this card and software, the 71 becomes an ideal data collection device, dumping the information to the PC when you get back to your desk.

It can be difficult moving between computers. Commands differ, and it's rare to find a program which will easily translate between the divergent BASICs. The work style requires a different mind set; the PC doesn't like to have more than one program in memory at a time, and bogs down each time you start a program. A prime number program takes 15-30 seconds to load, then runs in a little over a minute in BASIC (it runs in just a few seconds in compiled BASIC). On the 71, it starts immediately, but takes 5 minutes to run. Text files on the PC may have incredibly long lines (the end of the line often marks a paragraph), or embedded codes using high-bit set characters, while the 71 likes straight Text in lines under 96 characters. These differences surface in all manner of things: You may sit down to the keyboard and not remember if you should KILL, PURGE, DEL or ERASE a file. These caveats in mind, if you are prepared to play arbitrator in a marriage of machines, a PC and a 71 can be a powerful pair.

HPILLINK is compatible with most display escape sequences and handles word-wrap well, though it does not support <esc>%, so programs which do formatted display (screen editors or graphics) won't run correctly.

Once the 71 is connected to the PC and you have HPILLINK running, there are many ways to exchange information. The easiest is to copy files to or from the PC with the F1 function key; these files retain the original 71 format so that you can save LEX or BAS files on the PC. HPILLINK is a very polite program in PC terms. It'll run on an inexpensive "clone" MS-DOS computer and with RAM-resident programs on the PC such as Borland SideKick and Lotus Metro and the Microsoft Windows operating shell (copyrights of their respective manufacturers). These programs have cut&paste to help you move Text a line or screen-full at a time.

The PC Disk (we'll use the PC-world spelling of "Disk") is a great place to store 71 files of any kind, and can replace HP-IL storage devices. Nevertheless, Text is the choice if you want to easily read the files with PC programs. Text files to send to the 71 should be limited to 96 character line length and have the file name extension ".TEX". With this new freedom in editing and file size comes an interesting extension to program development: Self-documented program code. Writing BASIC for the 71 in Text files on the PC is a joy because you can add remarks as needed, without concern for memory use. To make the program more readable, we'll define a remark as any line which does not begin with a line number; you can use an exclamation mark for clarity, but it isn't essential. When the file is loaded into the 71, this program drops any un-numbered lines, then checks that line numbers are sequential (nasty crash otherwise).

```
10 CALL NOREMS @ SUB NOREMS @ DELAY 0,INF
20 INPUT 'source file: ' S$ @ IF FILESZR(S$)<1 THEN 20
30 DISP 'trimming file...'
40 DIM X$[256] @ ASSIGN #1 TO S$
50 FOR X=FILESZR(S$)-1 TO 0 STEP -1
60 READ #1,X; X$ @ IF NOT POS(X$[1,1],'0123456789') THEN DELETE #1,X
70 NEXT X
90 DISP 'done trimming'
200 DISP 'verifying line numbers...'
210 L=10000 @ FOR X=FILESZR(S$)-1 TO 0 STEP -1
220 READ #1,X; X$ @ L2=VAL(X$)
230 IF L2>=L THEN DISP 'pgm line:';L2;'text line:';X;'incorrect'
240 L=L2 @ NEXT X
250 ASSIGN #1 TO * @ DISP 'done verifying'
260 FOR X=1 TO 8 @ BEEP X*200 @ NEXT X
```

Once the source file with remarks becomes too large to fit in available RAM in the 71 (and it will when you get comfortable with all of those remarks), it's more practical to drop the remarks before copying the file to the 71, keeping the source file intact on the PC for later revisions. Assembler source files would be interpreted somewhat differently: Delete lines which begin with "*". NOBLANKS was written for BASIC compilers, if you are using interpreted BASIC, the program will have to be modified to ask the user for the file name, and of course, it will require line numbers instead of label references; the compiled program is preferred because of much greater speed. This program does not verify line numbers. Syntax is quite simple: Furnish a source file and a destination file name. You cannot copy a file onto itself. From the DOS prompt enter:

```
NOBLANKS source.fil output.tex
```

```
' NOBLANKS.BAS for Microsoft QuickBASIC and Borland TurboBASIC Compilers.
PRINT " NOBLANKS.EXE deletes lines which do not begin with a number."
PRINT " The output file, if it exists, will be overwritten."
PRINT " CAUTION:  THERE IS NO ERROR TRAPPING."
x$ = COMMAND$                  ' Replace with INPUT X$ for BASIC interpreters.
DEFINT L, X                                    ' Integer loop counters.
DEFDBL T                                       ' Double precision timer.
LineCount = 0  :  t = TIMER : ON ERROR GOTO ErrTrap
' Enable keyboard interrupts. Stop pgm with Ctrl-C, Ctrl-Z, or Ctrl-Break
KEY 15, CHR$(&H4) + CHR$(&H2E)                 ' Ctrl-C
KEY 16, CHR$(&H4) + CHR$(&H2C)                 ' Ctrl-Z
KEY 17, CHR$(&H4) + CHR$(&H46)                 ' Ctrl-Break
KEY(15) ON : KEY(16) ON : KEY(17) ON           ' Enable key trapping.
ON KEY(15) GOSUB KeyTrap : ON KEY(16) GOSUB KeyTrap :ON KEY(17) GOSUB KeyTrap
GOSUB wtrim : x = INSTR(x$, " ") : IF x = 0 THEN GOTO Scold
source$ = LEFT$(x$, x - 1)                     ' Get the source file name.
x$ = MID$(x$, x + 1) : GOSUB wtrim             ' Get the second file name.
dest$ = x$                                     ' dest$ has the file name.
```

```
PRINT "Deleting lines which do not begin with a number..."
OPEN source$ FOR INPUT AS #1 : OPEN dest$ FOR OUTPUT AS #2
WHILE NOT EOF(1)                                    ' Loop until out of data.
  LINE INPUT #1, x$                                 ' Read a line from the file.
  IF LEN(x$) THEN                                   ' If the line has data, then
    IF INSTR("1234567890", LEFT$(x$, 1)) THEN       ' Do a line number check.
      PRINT LEFT$(x$, 5) : PRINT #2, x$ : LineCount = LineCount + 1
    END IF
  END IF
WEND                                                ' We're out of data.
CLOSE : PRINT "The un-numbered lines have been deleted."
done:
  PRINT "Total lines:"; LineCount
  PRINT "Elapsed time:"; TIMER - t : END            ' End of program.
KeyTrap:                                            ' Keyboard interrupt handler.
  KEY(15) STOP : KEY(16) STOP : KEY(17) STOP        ' Disable key interrupts.
  PRINT : PRINT "Do you want to end program early?  Y/N"
YNLoop:                          ' Loop and wait for a good keystroke (Y or N).
  YN% = INSTR("YyNn", INPUT$(one%)) : IF YN% < one% THEN GOTO YNLoop
  IF YN% > two% THEN KEY(15) ON : KEY(16) ON : KEY(17) ON : RETURN
RETURN done                                         ' Give partial stats, end pgm.
wtrim:                                              ' Drop leading spaces in X$.
  IF LEN(x$) = 0 THEN GOTO Scold                    ' Not needed by some BASICs.
  WHILE ASC(x$) = 32                                ' Is first char a space?
    x$ = MID$(x$, 2)                                '   then delete it.
  WEND
  IF LEN(x$) THEN RETURN                            ' If no string then error-out.
ErrTrap:
  ON ERROR GOTO 0                                  'Disable err trap (not needed)
  CLOSE : PRINT : BEEP : PRINT "A run-time error has occurred" : PRINT
Scold:
  PRINT " ************************************************************"
  PRINT " *   Specify two valid file names, separated by a space.    *"
  PRINT " *                                                          *"
  PRINT " *   enter:    NOBLANKS SOURCE.FIL OUTPUT.FIL               *"
  PRINT " ************************************************************"
END
```

System RAM

Display Driver

2E100	ANNAD1	1	
2E101	ANN15	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 wrd #168F
2F43C	VECTOR	5	Interrupt vector
2F441	ATNDIS	1	ATTN key disable
2F442	ATNFLG	1	ATTN key hit?
2F443	KEYPTR	1	Key buffer pointer
2F444	KEYBUF	15*2	Key buffer
2F471	WINDST	2	Display window start
2F473	WINDLN	2	Display window length
2F475	DSPSTA	6	Display status
2F47B	ESCSTA	1	Escape status
2F47C	FIRSTC	2	Buffer pos of 1st char
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 program end
2F56C	CURREN	5	Current file end
2F571	IOBFST	5	System buffers
2F576	IOBFEN	5	System buffer end
2F576	CLCBFR	5	Calc Mode pointers
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	Program counter
2F67E	CNTADR	5	Cont address
2F683	ERRSUB	5	ON ERROR address
2F688	ERRADR	5	ONERROR stmt addr
2F68D	ONINTR	5	ON INTR stmt addr
2F692	DATPTR	5	DATA stmt pointer
2F697	TMRAD1	5	TMR#1 stmt address
2F69C	TMRAD2	5	TMR#2 stmt address
2F6A1	TMRAD3	5	TMR#3 stmt address
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 buffer start
2F6CB	AUTINC	4	Increment for AUTO
2F6CF	LEXPTR	5	Temporary RESPTR
2F6D4	CMDPTR	5	CMMD stack pointer
2F6D4	INADDR	5	Stmt len parse/decom
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
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
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	Temporary D0
2F896	STMTD1	5	Temporary 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	Temporary D0
2F8C0	FUNCD1	5	Temporary D1
2F8C5	TRFMBF	60	TRANSFORM scratch
2F901	SCRST0	4*16	Scratch stack (Mant)
2F941	SCREX0	5	Scratch stack (exp)

Display/Print

2F946	SCROLT	2	Display scroll rate
2F948	DELAYT	2	Display delay rate
2F94A	NEEDSC	1	Scroll needed
2F94D	DPOS	2	Current DISP column
2F94F	DWIDTH	2	Display 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	#CMD Stack Entries
2F977	CSPEED	5	Clock Speed (Hz/16)

HP-IL

2F97C	ERRLCH	1	
2F97D	TERCHR	2	ENTER term char
2F97F	HPSCRH	7	Reserved
2F986	RESERV	48*2	Reserved

Memory Map

00000

Operating system ROM.

20000

Memory mapped I/O.

2C000

Reserved for card reader.

2E100

Display RAM.

2E400

System RAM.

2F9E6

Reserved RAM.

CONFST

Configuration buffer.

MAINST

Main file chain.

MAINEN

System buffers

IOBFEN

CLCBFR

Command stack..

RFNBFR

Calc mode buffers.

OUTBS

Output buffers.

AVMEMS

Free RAM available for temporary buffers during function execution.

AVMEME

MTHSTK

Math stack.

FORSTK

For/next stack.

GSBSTK

GOSUB stack.

ACTIVE

Active variables.

CALSTK

Environments saved during CALLs.

RAMEND

Plug-in ROMs, independent RAM.

FFC00

Reserved for configuration.

FFFFF

Assembler Instruction Set

?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 if srvc req bit=0.
?ST#0	n	5	Test if status bit n#0.
?ST#1	n	5	Test if status bit n#1.
?ST=0	n	5	Test if status bit n=0.
?ST=1	n	5	Test if status bit n=1.
?XM=0		5	Test ext 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=AIB	fs	4	A OR B into A.
A=AIC	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 bye one.
A=A+A	fs	2,3	Sum of A+A into A.
A=A+B	fs	2,3	Sum of A+B into A.
A=A+C	fs	2,3	Sum of A+C into A.
A=A-1	fs	2,3	Decrement A by one.
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 use D0
A=DAT1	fsd	3,4	Load A from mem use 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.
AD0EX		3	Exchange A(A) and D0
AD0XS		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 circular.
ASR	fs	2,3	Shift A right 1 nib.
ASRB		3	Shift A right 1 bit.
ASRC		3	Shift A right 1 nib circular
B=-B	fs	2,3	Two's compl of B to B.
B=B-1	fs	2,3	One's compl of B to B.
B=0	fs	2,3	Set B=0.
B=A	fs	2,3	Copy A to B.
B=BIA	fs	4	B OR A into B.
B=BIC	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 by one.
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 by one.
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 circular.
BSR	fs	2,3	Shift B right 1 nib.

Inst.	Field	Nibs	
BSRB		3	Shift B right 1 bit.
BSRC		3	Shift B right 1 nib circular
BUSCC		3	Enter bus command "C".
C+P+1		3	Increment C plus P pointr.
C=C−C	fs	2,3	Two's compl of C to C.
C=C−1	fs	2,3	One's compl of C to 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=CIA	fs	4	C OR A into C.
C=CIB	fs	4	C OR B into C.
C=CID	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.
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 by one.
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 use D0
C=DAT1	fsd	3,4	Load C from mem use D1
C=ID		3	Request chip ID to C(A).
C=IN		3	Load C(0−3) w/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.
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 circular.
CSR	fs	2,3	Shift C right 1 nib.

Inst.	Field	Nibs	
CSRB		3	Shift C right 1 bit.
CSRC		3	Shift C right 1 nib circ.
CSTEX		2	Exchg 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 a hex const.
D1=HEX	hhhh	6	Load D1 with a hex const.
D1=HEX	hhhhh	7	Load D1 with a hex const.
D=−D	fs	2,3	Two's compl of D to D.
D=−D−1	fs	2,3	One's compl of D to D.
D=0	fs	2,3	Set D=0.
D=C	fs	2,3	Copy C to D.
D=C−D	fs	2,3	C minus D into D.
D=DIC	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 by one.
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 by one.
D=D−C	fs	2,3	D minus C to D.
DAT0=A	fsd	3,4	Copy A to mem using D0.
DAT0=C	fsd	3,4	Copy C to mem using D0.
DAT1=A	fsd	3,4	Copy A to mem using D1.
DAT1=C	fsd	3,4	Copy C to mem 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 circular.
DSR	fs	2,3	Shift D right 1 nib.
DSRB		3	Shift D right 1 bit.
DSRC		3	Shift D right 1 nib circular
GOC		3	Goto label if carry <=127
GOLONG		6	Go long to a label. Range: <=32767.
GONC		3	Go to label if no carry. Range: <=127.
GOSBVL		7	Gosub very long to label.
GOSUB		4	Gosub to label. Range: <=2047.

Inst.	Field	Nibs	
GOSUBL		6	Gosub long to a label. Range: <=32767.
GOTO		4	Goto a label. Range: <=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 w/const 0<m<=6.
LCASC		3+n	Load C with ASCII use P.
LCHEX		3+n	Load C with hex using P.
MP=0		3	Clear the mod. 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.
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 misg bit, rtn
RTNYES		2	Return if test is true.
SB=0		3	Clear the sticky bit.
SETDEC		2	Set cpu to dec mode.
SETHEX		2	Set the cpu to hex mode.
SHUTDN		3	Shut down bus and cpu.
SR=0		3	Clear service request bit.
SREQ?		3	Poll for service request.
ST=0	n	3	Clear program status bit n
ST=1	n	3	Set prgrm status bit n.
ST=C		2	Copy C(X) to status reg.
UNCNFG		3	Unconfigure.
XM=0		3	Clear external module missing bit.

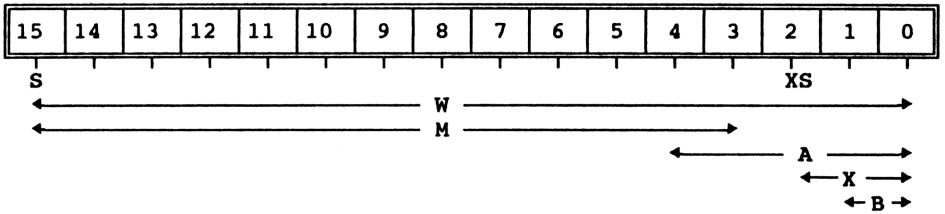
e	expression
l	label
b	byte

Inst.	Field	Nibs
	"	quoted string
	n	nib or nibs
BIN	"	Assemble a BIN file.
BSS	e	Evaluate, enter n nibs of '0'.
CHAIN	"	Subheader for BIN files.
CHAR	n	Type of BASIC Keyword.
CON(i)	e	Evaluate expr, enter (i) nibs.
EJECT		Formfeed in the listing.
END		Mark end of file (optional).
ENDTXT		Mark end of keyword table.
ENTRY	l	Begin def. of BASIC keyword.
EQU	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	"	Def immed. FORTH primitive.

Minimum LEX File Requirements

```
        LEX     'FILENAME'
        ID      #nn
        MSG     0
        POLL    0
??????  EQU     #nnnnn
        ENTRY   label       keyword entry
        CHAR    #n          F= all uses
        KEY     'KEYWORD'   keyword name
        TOKEN   b           1-255
label                       start of code
```

FIELDS IN WORKING REGISTERS



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 System 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 a string header to a string on the math 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 string header. LEVELS: 2. USES: A(A), C(W), D1.

ARGPRP 0E8EF Pop & normalize a REAL. As ARGPR+ but user modes are 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 clr.
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 error if arry, cmplx, NaN.
LEVELS: 2. USES: A, B(X), D(A), P, SB, XM, S8−11.

ASCII 0079B Bit pattern tables. Each character has 10 nibs. 2 nibs per display column. Least significant bit of byte (nib pair) is top row. Read using ASCII + 10 * (chr#).
ENTRY: DON'T! This is a table only, not an entry point.

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 a 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 the display.
ENTRY: D1 points to chars terminated with an FF byte.

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 = current time, R0=timer value represents 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 nibs left circular.

CSLC14 1B424	**CSLC11** 1B41B	**CSLC8** 1B42C	**CSLC5** 1B435
CSLC13 1B421	**CSLC10** 1B418	**CSLC7** 1B42F	**CSLC4** 1B438
CSLC12 1B41E	**CSLC9** 1B415	**CSLC6** 1B432	**CSLC3** 1B43B

CSRC15 1B441 C shift nibs right circular.

CSRC14 1B43E	**CSRC11** 1B435	**CSRC8** 1B42C	**CSRC5** 1B41B
CSRC13 1B43B	**CSRC10** 1B432	**CSRC7** 1B415	**CSRC4** 1B41E
CSRC12 1B438	**CSRC9** 1B42F	**CSRC6** 1B418	**CSRC3** 1B421

D1C=R3 03047 Restore C(A),D1 from R3. This is the 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 decimal integer to hex integer.
ENTRY for DCHX=C: C(W)= decimal integer.
ENTRY for DECHEX: A(W)= decimal integer.
EXIT: P=0, HEXMODE, A(A)=hex int. Carry set 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 that a byte is in range ASCII "0"–"9".
RANGE 1B07C Verify that 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 boundary, C(3–2)=upper boundary.
EXIT: P=0, Carry clear if the byte was in the range. LEVELS: 0. USES: A(B), C(A), P.

DSPBUF 09723 Send a buffer of characters 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 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 character to display.
ENTRY: P=0, C(B)=character to follow the escape character.
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 srch), 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 **FNRTN2** 0F219
FNRTN3 0F235 **FNRTN4** 0F238
Return to BASIC from a function. Push result 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 exponent is >14 (but a finite) 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 buf exists will adapt 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 set then D1= past header, P=0,D0=past hdr at ID#.
LEVELS: 2. USES: A, B, C, D, D0, 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, D0, D1.

I/OFND 118BA Find I/O buffer, set high bit on buf ID# so it will be deallocated at next config.
IOFND0 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 0EC6E Integer divide in hex or decimal mode. Zeros nibs 5–15 of A&C then goes to IDIV.
IDIV 0EC7B Hex or decimal full word integer divide.
CAUTION: if denominator=0 then this routine will loop indefinitely.
ENTRY: hex or decimal mode, A=dividend, C=divisor.
EXIT: mode not changed, P=15, quotient 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.
ENTRY: P=0.
EXIT: Pops a key, places it on the stack, returns to BASIC. Does not return, be sure D0,D1 are accurate.
LEVELS: 3. USES: A, B, C, D(A), R0, R1, R2, S0–S2, D0, D1.

KEYCOD 1FD22 Keycode map. Maps the keycode to the definition. Example: ENDLINE is #38 which maps to 0D hex (ASCII 13).
ENTRY: DON'T! THIS IS NOT AN ENTRY POINT!

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, extending to P.
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 temp. buffer.
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 check, 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 0ECBB Multiply hex*hex or hex*dec.
ENTRY: If hex*hex then SETHEX, args in A,C. If hex*dec then SETDEC, hex arg in C, dec arg in A.
EXIT: P unchanged, Mode not changed, Carry clear. Result returned in A,B,C. If hex*hex then result is hex. If hex*dec then result is decimal.
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 the 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 line from Text file to output buffer. File must have FIB# (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. Error out if item on stack doesn't begin with proper str hdr.
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: hex, P=15, A(W)=hours(BCD int), B(W),C(W)=minutes(BCD), D(W)=sec(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 of 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 won't recognize a key that was down when it was entered (won't repeat while a key remains held down).
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 stack into a string back on the stack using current disp FIX.
ENTRY: D1=top of math stack. S0 set if return when done, else jumps 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 pos within strs 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=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. Exits through YMDH01.
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.

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 (DVZ).
−8 invalid operation (IVL).
2F6DB −9 USER mode set.
−10 option angle radians mode.

Round off setting

	near	zero	pos	neg
−11	inf	0	0	1	1
−12	neg	0	1	0	1

Display format

2F6DC

	std	fix	sci	eng
−13	0	0	1	1
−14	0	1	0	1

−15 lowercase mode (LC ON)
−16 option base 1

Display digits

	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 HPIL ENTER terminate by EOT
−24 RESTOREIO not reassign devices
2F6DF −25 beep loud
−26 don't show BASIC prompt ">"
−27 alternate error message language
−28 allocatable − test only!
2F6E0 −29 allocatable − test only!
−30 allocatable − test only!
−31 allocatable − test only!
−32 allocatable − test only!
2F6E1 −33 allocatable − test only!
−34 allocatable − test only!
−35 allocatable − test only!
−36 allocatable − test only!
2F6E2 −37 allocatable − test only!
−38 allocatable − test only!
−39 allocatable − test only!
−40 allocatable − test only!
2F6E3 −41 allocatable − 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 is active.
−48 control key was hit.
2F6E5 −49 DSLEEP from power down.
−50 req set TRNOF in MAINLP.
−51 turnoff at MAINLP.
−52 VIEW key pressed.
2F6E6 −53 reserved for HP use.
−54 reserved for HP use.
−55 reserved for HP use.
−56 reserved for HP use.
2F6E7 −57 "AC" annunciator lit
−58 USER suspended (see flag −9).
−59 key repeated.
−60 "((*))" annunciator lit.
2F6E8 −61 "BAT" annunciator lit.
−62 "PRGM" annunciator lit.
−63 "SUSP" annunciator lit.
−64 "CALC" annunciator lit.

Display Escape Codes

Esc	Note	Description
A	1	Cursor up one line. If at top of screen will stay; Link moves to bottom of screen.
B	1	Cursor down. Move cursor down one line. Link will wrap to top of screen.
C		Move cursor one position right. Some devices wrap to next line if the cursor is at end of line.
D		Move cursor one position left. Some devices will wrap to previous line if at beginning of line.
E		Clear display. Reset display, send the cursor home (upper left) and turn it on.
H		Cursor cursor to upper left corner of the display.
J		Clear display from cursor.
L		Insert line above cursor, move cursor far left.
M		Delete line with cursor on it, move cursor far left.
N	2	Change to insert mode with text wrap to next line.
O	4	Delete character at cursor position. Will wrap text from next line if necessary.
P		Delete character at cursor position without wrap.
Q	4	Change to insert cursor (underline or arrow). 71 will go to insert mode.
R	4	Change to overstrike cursor display. 71 goes into overstrike mode.
S	1,3,4	Roll screen up one line. Will roll a line from screen buffer if it exists. Does not alter what is displayed on the built−in LCD display, just the device.
T	1,3,4	Roll screen down one line.
<		Turn off the cursor. Does not change from insert to overstrike mode.
>		Turn the cursor on.
%	3,4	Move cursor to display address. Specify col then row.

Notes:
1 Not supported by HP−71.
2 Not supported by HP−82163A.
3 Not supported by HP−82477A HP−IL Link prog.
4 Not supported by some terminals.

Dec	Hex	Oct	Bin	CHR$	Dec	Hex	Oct	Bin	CHR$
0	0	0	00000000	NUL	128	80	200	10000000	
1	1	1	00000001	SOH	129	81	201	10000001	
2	2	2	00000010	STX	130	82	202	10000010	
3	3	3	00000011	ETX	131	83	203	10000011	
4	4	4	00000100	EOT	132	84	204	10000100	
5	5	5	00000101	ENQ	133	85	205	10000101	
6	6	6	00000110	ACK	134	86	206	10000110	
7	8	7	00000111	BEL	135	87	207	10000111	
8	8	10	00001000	BS	136	88	210	10001000	
9	9	11	00001001	HT	137	89	211	10001001	
10	A	12	00001010	LF	138	8A	212	10001010	
11	B	13	00001011	VT	139	8B	213	10001011	
12	C	14	00001100	FF	140	8C	214	10001100	
13	D	15	00001101	CR	141	8D	215	10001101	
14	E	16	00001110	SO	142	8E	216	10001110	
15	F	17	00001111	SI	143	8F	217	10001111	
16	10	20	00010000	DLE	144	90	220	10010000	
17	11	21	00010001	DC1	145	91	221	10010001	
18	12	22	00010010	DC2	146	92	222	10010010	
19	13	23	00010011	DC3	147	93	223	10010011	
20	14	24	00010100	DC4	148	94	224	10010100	
21	15	25	00010101	NAK	149	95	225	10010101	
22	16	26	00010110	SYN	150	96	226	10010110	
23	17	27	00010111	ETB	151	97	227	10010111	
24	18	30	00011000	CAN	152	98	230	10011000	
25	19	31	00011001	EM	153	99	231	10011001	
26	1A	32	00011010	SUB	154	9A	232	10011010	
27	1B	33	00011011	ESC	155	9B	233	10011011	
28	1C	34	00011100	FS	156	9C	234	10011100	
29	1D	35	00011101	GS	157	9D	235	10011101	
30	1E	36	00011110	RS	158	9E	236	10011110	
31	1F	37	00011111	US	159	9F	237	10011111	
32	20	40	00100000	SPACE	160	A0	240	10100000	
33	21	41	00100001	!	161	A1	241	10100001	!
34	22	42	00100010	"	162	A2	242	10100010	"
35	23	43	00100011	#	163	A3	243	10100011	#
36	24	44	00100100	$	164	A4	244	10100100	$
37	25	45	00100101	%	165	A5	245	10100101	%
38	26	46	00100110	&	166	A6	245	10100110	&
39	27	47	00100111	'	167	A7	247	10100111	'
40	28	50	00101000	(	168	A8	250	10101000	(
41	29	51	00101001	)	169	A9	251	10101001	)
42	2A	52	00101010	*	170	AA	252	10101010	*

Dec	Hex	Oct	Bin	CHR$	Dec	Hex	Oct	Bin	CHR$
43	2B	53	00101011	+	171	AB	253	10101011	+
44	2C	54	00101100	,	172	AC	254	10101100	,
45	2D	55	00101101	-	173	AD	255	10101101	-
46	2E	56	00101110	.	174	AE	256	10101110	.
47	2F	57	00101111	/	175	AF	257	10101111	/
48	30	60	00110000	0	176	B0	260	10110000	0
49	31	61	00110001	1	177	B1	261	10110001	1
50	32	62	00110010	2	178	B2	262	10110010	2
51	33	63	00110011	3	179	B3	263	10110011	3
52	34	64	00110100	4	180	B4	264	10110100	4
53	35	65	00110101	5	181	B5	265	10110101	5
54	36	66	00110110	6	182	B6	266	10110110	6
55	37	67	00110111	7	183	B7	267	10110111	7
56	38	70	00111000	8	184	B8	270	10111000	8
57	39	71	00111001	9	185	B9	271	10111001	9
58	3A	72	00111010	:	186	BA	272	10111010	:
59	3B	73	00111011	;	187	BB	273	10111011	;
60	3C	74	00111100	<	188	BC	274	10111100	<
61	3D	75	00111101	=	189	BD	275	10111101	=
62	3E	76	00111110	>	190	BE	276	10111110	>
63	3F	77	00111111	?	191	BF	277	10111111	?
64	40	100	01000000	@	192	C0	300	11000000	@
65	41	101	01000001	A	193	C1	301	11000001	A
66	42	102	01000010	B	194	C2	302	11000010	B
67	43	103	01000011	C	195	C3	303	11000011	C
68	44	104	01000100	D	196	C4	304	11000100	D
69	45	105	01000101	E	197	C5	305	11000101	E
70	46	106	01000110	F	198	C6	306	11000110	F
71	47	107	01000111	G	199	C7	307	11000111	G
72	48	110	01001000	H	200	C8	310	11001000	H
73	49	111	01001001	I	201	C9	311	11001001	I
74	4A	112	01001010	J	202	CA	312	11001010	J
75	4B	113	01001011	K	203	CB	313	11001011	K
76	4C	114	01001100	L	204	CC	314	11001100	L
77	4D	115	01001101	M	205	CD	315	11001101	M
78	4E	116	01001110	N	206	CE	316	11001110	N
79	4F	117	01001111	O	207	CF	317	11001111	O
80	50	120	01010000	P	208	D0	320	11010000	P
81	51	121	01010001	Q	209	D1	321	11010001	Q
82	52	122	01010010	R	210	D2	322	11010010	R
83	53	123	01010011	S	211	D3	323	11010011	S
84	54	124	01010100	T	212	D4	324	11010100	T
85	55	125	01010101	U	213	D5	325	11010101	U

Dec	Hex	Oct	Bin	CHR$	Dec	Hex	Oct	Bin	CHR$
86	56	126	01010110	V	214	D6	326	11010110	V
87	57	127	01010111	W	215	D7	327	11010111	W
88	58	130	01011000	X	216	D8	330	11011000	X
89	59	131	01011001	Y	217	D9	331	11011001	Y
90	5A	132	01011010	Z	218	DA	332	11011010	Z
91	5B	133	01011011	[	219	DB	333	11011011	[
92	5C	134	01011100	\	220	DC	334	11011100	\
93	5D	135	01011101	]	221	DD	335	11011101	]
94	5E	136	01011110	^	222	DE	336	11011110	^
95	5F	137	01011111		223	DF	337	11011111	
96	60	140	01100000	`	224	E0	340	11100000	`
97	61	141	01100001	a	225	E1	341	11100001	a
98	62	142	01100010	b	226	E2	342	11100010	b
99	63	143	01100011	c	227	E3	343	11100011	c
100	64	144	01100100	d	228	E4	344	11100100	d
101	65	145	01100101	e	229	E5	345	11100101	e
102	66	146	01100110	f	230	E6	346	11100110	f
103	67	147	01100111	g	231	E7	347	11100111	g
104	68	150	01101000	h	232	E8	350	11101000	h
105	69	151	01101001	i	233	E9	351	11101001	i
106	6A	152	01101010	j	234	EA	352	11101010	j
107	6B	153	01101011	k	235	EB	353	11101011	k
108	6C	154	01101100	l	236	EC	354	11101100	l
109	6D	155	01101101	m	237	ED	355	11101101	m
110	6E	156	01101110	n	238	EE	356	11101110	n
111	6F	157	01101111	o	239	EF	357	11101111	o
112	70	160	01110000	p	240	F0	360	11110000	p
113	71	161	01110001	q	241	F1	361	11110001	q
114	72	162	01110010	r	242	F2	362	11110010	r
115	73	163	01110011	s	243	F3	363	11110011	s
116	74	164	01110100	t	244	F4	364	11110100	t
117	75	165	01110101	u	245	F5	365	11110101	u
118	76	166	01110110	v	246	F6	366	11110110	v
119	77	167	01110111	w	247	F7	367	11110111	w
120	78	170	01111000	x	248	F8	370	11111000	x
121	79	171	01111001	y	249	F9	371	11111001	y
122	7A	172	01111010	z	250	FA	372	11111010	z
123	7B	173	01111011	{	251	FB	373	11111011	{
124	7C	174	01111100	|	252	FC	374	11111100	|
125	7D	175	01111101	}	253	FD	375	11111101	}
126	7E	176	01111110	~	254	FE	376	11111101	~
127	7F	177	01111111		255	FF	377	11111111	

HP-71 Keyboard Map

f-	#57 IF	#58 THEN	#59 ELSE	#60 FOR	#61 TO	#62 NEXT	#63 DEF	#64 KEY	#65 ADD	#66 LR	#67 PREDV	#68 MEAN	#69 SDEV	#70 SQR
	#1 Q	#2 W	#3 E	#4 R	#5 T	#6 Y	#7 U	#8 I	#9 O	#10 P	#11 7	#12 8	#13 9	#14 /
g-	#113 q	#114 w	#115 e	#116 r	#117 t	#118 y	#119 u	#120 i	#121 o	#122 p	#123 '	#124 {	#125 }	#126 ^

f-	#71 CALL	#72 GOSUB	#73 RETURN	#74 GOTO	#75 INPUT	#76 PRINT	#77 DISP	#78 DIM	#79 BEEP	#80 FACT	#81 SIN	#82 COS	#83 TAN	#84 EXP
	#15 A	#16 S	#17 D	#18 F	#19 G	#20 H	#21 J	#22 K	#23 L	#24 =	#25 4	#26 5	#27 6	#28 *
g-	#127 a	#128 s	#129 d	#130 f	#131 g	#132 h	#133 j	#134 k	#135 l	#136 ;	#137 $	#138 %	139 &	#140 :

f-	#85 EDIT	#86 CAT	#87 NAME	#88 PURGE	#89 FETCH	#90 LIST	#91 DELETE	#92 AUTO	#93 COPY	#94 RES	#95 ASIN	#96 ACOS	#97 ATAN	#98 LOG
	#29 Z	#30 X	#31 C	#32 V	#33 B	#34 N	#35 M	#36 (| #37)	#38 ENDLINE	#39 1	#40 2	#41 3	#42 -
g-	#141 z	#142 x	#143 c	#144 v	#145 b	#146 n	#147 m	#148 [	#149]	#150 CMDS	#151 !	#152 "	#153 #	#154 @

f-	#99 OFF			#102 SST	#103 BACK	#104 ←CHAR	#105 I/R	#106 LC	#107 ←LINE		#109 USER	#110 VIEW	#111 CALC	#112 CONT
	#43 ON	f–	g–	#46 RUN	#47 ←	#48 →	#49 SPC	#50 ↑	#51 ↓		#53 0	#54 .	#55 ,	#56 +
g-	#155			#158 CTRL	#159 ⇤	#160 ⇥	#161 ERRM	#162 ⤒	#163 ⤓		#165 1USER	#166 <	#167 >	#168 ?

Index

Addr$ 57
Algebraic 22
Ansi 27
Array
  Descriptors (Assembler) 113
Arrays 30
  Math Rom 31
  Re–Dimensioning 31
  String 31
Ascii 27, 141
  Codes 27
Assembler 6
  Array Descriptors 113
  Bin 58
  Bugs 116, 117, 123
  Decompiling 110
  Entry 108, 111
  Entry Points 113
  Equ 108
  ID 109
  Instruction Set 150
  Introduction 101
  Key 109
  Math Stack 111, 120
  Minimum Requirements 152
  MSG Table 108
  Parameters 110
  Parsing 109, 110
  Poll 108
  Return Stack 118
  Scratch 120
  Source Files 101
  Token 109
Assign IO 11
Assign# 67, 69
Basic 55
  File 55
  Format 57
  Functions 50
  History 39
  HP 41
  Interpreted 50
  Labels 41
  Line Editor 47
  Line Numbers 57
  Programming 39, 47
  Statements 50
  Tokens 50
  Translating 93
Battery
  Cassette 134
  Disc 17
  Disc Drive 132, 133
  HP71 3, 37, 82, 102, 131
  Printers 135
  Ram Disc 134
Baud 144
Beep 21
Bin 58, 101
Binary 5
Bit 5
Borland Turbo Basic 93, 146
Bugs 20, 44, 59, 76, 77, 81, 114, 116, 117, 123, 132
Calc Mode 33
  Command Stack 35
  Def Fn 36
  Keyboard 34
  Long Formulas 34
  Precedence 33
  Res 34
  Variables 36
Call 7, 9, 11
  Cautions 10
Capricorn 5
Card Reader 7, 131
  Finding 90
Cassette 3, 16, 134
  Battery 134
Cat 11, 12, 55
Cat All 11
Char (Assembler) 108
Charset 79
Chr$ 26
Clock Speed 5, 6
Cmdstk 86
Command Line 128
Command Stack 35
  Setting Depth 86
Concatenation 23, 41
  Date$ 81
Conflicts
  Lex 59, 107
Cont 11
Control Codes 53, 76, 94, 135, 141, 160
Copy 12
CPU 5, 102
  Clock Speed 5
  P Pointer 104
  Registers 5, 102
    Carry 102
    Control 103
    Fields 103, 105
    Scratch 103
    Working 104
  Saturn 5
  Working Registers 102
Crashes 4, 117, 119
Create 66
Data
  File
    Format 61
    Files 57
      Using 59
Data Files 59
Date
  Date$ 20
Decompiling (Assembler) 110
Def Fn 82, 83
  Calc Mode 36
  File Sub–header 57
Def Key 12
Delay 15, 89
Delete# 61
Destroy 15
Dim 29, 45, 72, 73, 127
Disc 16, 132
  Battery 133
  Ram 134
Disp 8
  Implied 21, 29
Display
  Display Is 19
  Finding 87
  Format 89
Display Format 20
Display Is 19, 87
Edit 8, 15, 55, 66
EDLEX 61
End 16
End All 16
Eng 20
Entry Points 6, 113
  Listing 153
Entry (Assembler) 108, 111
Environments 7
  Suspended 9
Eprom 131
Equ (Assembler) 108
Error 16, 89
Escape 49, 53, 76, 94, 137, 138, 141, 145
Fetch 16
FIB 68
Fields (CPU) 105
File
  Bin 58
  Chain 57, 66
  Closing 69
  Creating 66
  Data 57, 59
  Fib 68
  Finding 55
  Forth 58
  Header 55, 56
  Key 64
  Lex 58
  Opening 67
  Pointer 68

Index

- Recalling Data 69
- Sdata 64
- Text 61, 69
- Types 55, 56
- Workfile 15

Fileszr 62
Fix 20
Flags 88
- Beeper 21
- Global 7
- Listing 45
- Peek$ 88
- System Flag Table 160

For 44
- Step 44

Forth 58, 107
Functions 25, 50
- Exiting (Assembler) 121

Global Environment 7
Gosub 42
- Computed 43

Goto 40
- Computed 43
- Implied 45
- Label 79

Hex 5
- Explained 5
- Table 5

HP71
- Battery Life 3
- Command Stack 35
- CPU 5, 102
- Crashes 4
- Environments 7
- Operating System 6
- System RAM Table 148

HP–75 3, 5, 63, 73, 81, 98
HP–IB 137
HP–IL
- Ascii Codes 27
- Assign IO 11
- Cassette 134
- Cat 11
- Copy 12
- Data Files 66
- Device List 135
- Device Names 100
- Disc 132
- Display 135
- Display Is 19
- HP75 100
- Initialize 16
- Label 17
- Mass Storage 3, 130, 132
- Off IO 18
- Plist 17
- Printer Is 19
- Printers 135
- Purge 15
- Pwidth 20
- Rename 18
- Reset 18
- Restore IO 18
- Volume Label 16

Htd 5, 14, 84
IDS 58, 108, 109, 113
If 45
Inf 20
Initialize 16
Input 39
Insert# 62
Instruction Set (Asm) 150
Integer 29
Key Files 64
- Format 64
- List 17

Key (Assembler) 109
Keyboard
- Alternate Character Set 80
- Calc Mode 34
- Cont 11
- External 145
- Put 36
- Re–defining 12
- Run 11
- Waiting For a Key 82

Keydown 82
Keywait$ 13, 82
Keyword 11, 50, 58, 61
Labels 41
- Goto 79
- Mass Storage 17

Lcd 10, 15
Leeway 121
Len 53
Lex 58, 107
- Array Descriptors 113
- Conflicts 59, 107
- Decompiling 110
- Entry 111
- Entry Points 113
- Equ 108, 109
- File Requirements 105
- ID 107
- Math Stack 111, 120
- Minimum Requirements 152
- Parameters 110
- Parsing 109, 110
- Poll 108
- Return Stack 118
- Scratch 120
- Token 109
- Using 58

Lexical Analyzer 110
LexID 109
Line Numbers 57
List 17
Loop 44
- Step 44

Maintenance
- Cassette 134
- Disc Drive 133
- Magnetic Cards 132

Mass Storage 132
Math
- BASIC 22
- Calc Mode 33
- Calc Mode Precedence 33
- Parentheses 25
- Precedence 24, 35

Math Rom 31, 58, 112
- Not 59
- Variables 32

Math Stack 111
- Numbers 120
- Strings 120

Max 50
Mem 12, 18, 55
Memory 6
- File Chain 66
- File Size 89
- How Much? 130
- Main 7
- Map (Simplified) 9
- Map (Complete) 149
- Modules 131
- Ports 7, 12

Menus 128
Microsoft 93, 138
- QuickBasic 93, 94, 146

Modular Programs 126
Name 18
Next 44
- Loop 44

Nib 5
Nibhex 110
Null 19
Num 26, 81
Nybble 5
Off IO 18
Operating System 6, 102
Option Base 30
P Pointer 104
Pack 17
Parameters
- Assembler 110
- Passing 9, 48

Parsing 22
Parsing (Assembler) 109, 110
Pause 10
Peek$ 84, 93, 117

Index

Planning 125
Plist 17
Poke 84, 93, 117, 140, 142
Poll (Assembler) 108
Pop 43
  Assembler 103
Ports 5, 7, 12
Pos 53
Precedence 24, 35
  Calc Mode 33
Print# 68
Printer
  Control Codes 53
  Non–HP 142, 144
  Plist 17
  Printer Is 19
  Pwidth 20
Printer Is 19
Printers 135
Private 55
Programs
  Acyducy 47
  Bior 70
  Bublsort 126
  Calcaid 36
  Decide 51
  Environ 10
  Frog 46
  Graphix 32
  Incat 48
  Mathquiz 38
  Noblanks 146
  Norems 146
  Prflags 45
  Revlex 105
  Sample Lex 114
  Sdtext 91
Purge 12, 15, 55, 66
Pwidth 20, 89
Ram 6
  Disc 134
  How Much? 130
  Main 7
  Modules 131
  Ports 7, 12
  System 85
  System RAM Table 148
Randomize
Read# 69
Registers 28
  Carry (CPU) 102
  Control (CPU) 103
  CPU 5, 102
  Fields Within (CPU) 103
  P Pointer (CPU) 104
  Res 25, 34, 88
  Scratch(CPU) 103
  Sdata 64, 90
  Working (CPU) 102, 104
Remarks 41, 43, 71, 94, 99, 102, 106, 125, 126, 145
Rename 18
Replace# 62
Res 25, 34, 88
  Complex 59
Reset HPIL 18
Restore 42, 68
Restore IO 18
Restore# 68
Return 42
Return Stack 118
Rom 6, 55
RS–232 3, 132, 135, 137
  Cable 142
Run 7, 11
Saturn 5, 102
Sci 20
Scratch
  Assembler 120
  File 15
  Lex ID 107, 109
  Registers (CPU) 5, 102, 103, 104
  Saving 127
  Variables 73, 81
Sdata Files 64
  Format 65, 92
  Using 64
Search# 62
Secure 19
Setdate 19
Settime 19
Short 29
Startup 19
Statements 25, 50
Std 20
Str$ 27
Strings
  Arrays 31
  Assembler 112
  Centering 81
  Dim 72
  Filling With Spaces 81
  Lowercase 81
  Math Stack 120
  Replacing Chars 81
  Reversing 81, 105
  Rotate Left 82
  Rotating Right 82
  Trimming Spaces 82
  Variable Names 29
Sub–Programs 9, 48
Subroutines 42
  Assembler 121
Syntax 21
  Basic 47
  Case 26
  Concatenation 23, 41
  Multi–Statement Lines 23
  Spaces 23
System Ram 85
  Table 148
Tape 16
Terminal 3, 27, 87, 98, 135, 137, 139, 141, 143
Text File
  Format 63
Text Files 61
  Keywords 61
  Reading 69
  Using 61
TEXTUTIL 61
Then 45
Time 19
  Time$ 20
Timers 77, 83, 88
  Bug 77
Titan 5
Token (Assembler) 109
Tokens 50
Unsecure 19
Val 27
Variables
  Arrays 30
  Calc Mode 36
  Calculator 28
  Complex 32
  Global 7
  Lists 127
  Math Rom 32
  Memory Use 31
  Names 32
  Re–Dimensioning 31
  Rules 32
  Scratch 73, 81
  Strings 52
  Symbolic Names 29
  Types 29
Volume Label 16
Width 20
WorkBook71 4, 114
Workfile 55