HP-71 LEX FILES:
DBLEX1
DBLEX2
IN FUTURE DBLEX3

TITAN
DEBUGGER
For the HP-71

User's Guide

ICC Release
February 3, 1986

Hewlett-Packard Company

CONTENTS

1. INTRODUCTION

The Titan Debugger offers programmers the ability to simulate assembly language on the HP-71.

In tandem with the FORTH/Assembler ROM or with PC-based Saturn Assemblers, the debugger provides an accurate, affordable and portable tool for assembly language development. Its many options allow the user to customize its operation, even midway through a debugging session. Its modularity is designed to allow the addition of more features through assembly language enhancements.

This user's guide provides a description of the file setup, CPU emulation, output and register manipulation. For details on HP-71 file structure, keywords and particulary Independent RAMs (IRAMs), see the HP-71 Owner's Manual.

The debugger will be made available to the general public through the HP-71 User's Library. Normal procedures for obtaining software are to specify the type of medium, and to purchase the program directly from the Library.

** Note for ICC release: The version of the debugger published February 3, 1986 is not complete. Many features described in this manual are not implemented. In the following sections notes such as this will point out which features are missing from the ICC release.

P 1-2 BLANK

2. OVERVIEW

2.1 Copied into Memory

The Titan Debugger is a RAM-based program of less than 12K bytes which resides in the HP-71's memory along with the target assembly language. It is designed in three modules, each a LEX file, which are meant to be copied into and run from Independent RAM (IRAM). The debugger is compatible with the HP-71's operating system, and may reside in memory without affecting the operating characteristics of the computer.

Invoking the debugger places the HP-71's operating system in a state of suspension, where individual CPU registers and memory locations can be examined and edited. Assembly language routines can be emulated, with interrupt, single-step and break point capability. The HP-71's keyboard is used to direct the debugger, and output is sent to both the LCD and the HPIL display device.

The first two LEX files, named DBGLEX1 and DBGLEX2, are the core of the debugger, including keyboard routines, opcode emulator and input/output. The third LEX file, named DBGLEX3, contains several options and enhancements. DBGLEX1 and DBGLEX2 are required to run the debugger, whereas DBGLEX3 may be left out of the HP-71 to increase the amount of memory available to the user's application.

** Note for ICC release: The options module, DBGLEX3, has not been implemented yet. Several options and enhancements described in this manual are not yet available.

2.2 System Vulnerability

Running the debugger out of IRAM permits it to use relatively fixed RAM locations for its buffer requirements. However, this makes it vulnerable to low-level configuration commands which de-address chips, so by necessity there are a few things the debugger cannot emulate. It is expected that fewer than 2% of debugger applications will encounter this limitation. In addition, since the debugger's emulator runs at about 1/75 real speed, critical timing loops in assembly language will yield poor results. Aside from these cautions, the debugger can handle almost any assembly language application.

The debugger is particularly vulnerable to the precise thing it is used to uncover -- bugs in assembly language code. Because it resides in RAM, writing data to any address within its file boundaries may render the debugger inoperable. Thus, encountering a bug which wipes out or moves a large chunk of memory will likely trash the debugger on its way to a Memory Lost.

2.3 How to use this document

The Titan Debugger is a keystroke-driven processor overlaid upon the HP-71's operating system. To use the debugger to its full potential, you must learn the stack structure (containing the emulated CPU registers), and the keyboard interface (for controlling the debugger/emulator).

Two factors complicate the keyboard interface: 1) the debugger shares the keyboard with the HP-71 operating system, and 2) the debugger offers many features, so many of the keys are used to select functions. You will find that the functions have been selected to match the typing aids on the HP-71, so a keyboard overlay, which might interfere with a target application, is not a necessity. Figure 1 is a representation of the HP-71 keyboard, with the debugger's keys highlighted. The keys in bold outline are the only keys used in the debugger, and the typing aids shown are the ones used as mnemonics for corresponding debugger functions. This figure may be useful for reference when working in the debugger.

The first four chapters of this manual should be studied by those using the debugger for the first time. You need to read Chapter 3 to learn how to enter and exit the debugger system. And you need to become familiar with the debugger stack structure, explained in Chapter 4, before attempting to use the emulator.

Chapters 4 through 7 contain all needed information on the debugger stacks and registers. They give overviews of the different debugger modes, usage of keys, and editing capabilities. Chapter 7 describes each stack and register in detail, and should be used for reference during debugger sessions. Chapters 8 and 9 give directions on using the emulator and interfacing to the BASIC operating system. These six chapters together give enough information to run a productive debugger session.

The remaining chapters of this manual are devoted to operating details and technical information. They describe advanced interfaces to assembly language routines, and techniques to use when emulating in the HP-71 operating system.

2.4 Notation for Keys

Throughout this document, keystrokes are indicated as follows:

- unshifted keys: [K] , [5] , [RUN] , etc.
- shifted keys: f[DELETE] (the f-shifted [M] key),
  g["] (the g-shifted [2] key),
  f[EDIT] (the f-shifted [Z] key), etc.

Titan Debugger
Keyboard

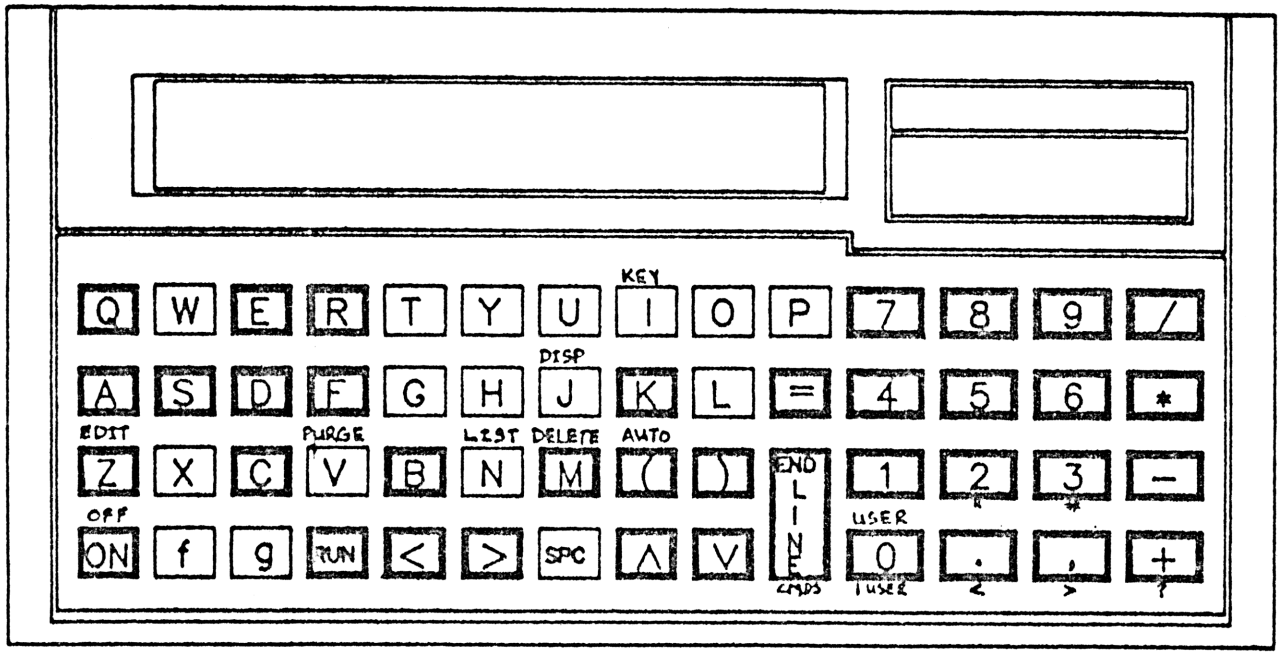


Figure 1

For clarity, the keystrokes for characters "less-than" ("<") and "greater-than" (">") are shown as g[<] and g[>], respectively. The four arrow keys are shown as [←], [→], [↑] and [↓]. The four corresponding g-shifted keys are g[←], g[→], g[↑] and g[↓].

2.5 Other References

No assembly language on the HP-71 should be attempted without thoroughly understanding the operating system. The HP-71 Software IDS, Volumes I through III, have been published to help programmers understand and work with the computer. The IDS part numbers are HP 00071-90068, 00071-90069 and 00071-90070.

Chapter 16, IDS Volume III, is indispensible when working with the debugger. It describes the Saturn CPU instruction set and some of the CPU and bus architecture.

The HP-71 Hardware IDS (part number HP 00071-90071) may also prove valuable, as it goes into more detail on the CPU and bus. Chapter 3 of this document also shows the CPU instruction set.

You need to be familiar with the source document for your assembler, for debugging often involves frequent code changes and reassemblies. For most users, this will be the HP-71 FORTH/Assembler ROM Owner's Manual, part number HP 82441-90001.

2.6 Equipment Needed

No extra equipment or peripherals are needed to run the Titan Debugger in the HP-71. You need to have enough RAM to read in the debugger (from 8K to 12K bytes, depending on your configuration), preferrably set up as IRAM. If this does not leave enough memory for your assembly language application, you may need to purchase more plug-in memory modules.

Debugging is easiest when you use an HPIL display device (which can also be a printer) for viewing registers and emulator results. This requires the HP-71 HPIL module and a display interface or printer. Not all debugging sessions can run with the external display device, however; see Chapter 11 for details.

For the safest use of an external display, you should use the HP-71 Dual HPIL module. This will allow you to use one loop for the debugger to display registers, while the other loop can be used by the HP-71 operating system in emulation. See Chapter 11 for details.

** Note for ICC release: The interface for the Dual HPIL module has not been implemented. The current version of the debugger uses the DISPLAY IS device and the LCD for output.

You may also use a KEYBOARD IS device in some debugging situations. This requires an HPIL module, a terminal keyboard, and the LEX file "KEYBOARD", available from the HP-71 User's Library, or found in the FORTH/Assembler module. See Chapter 11 for details.

** Note for ICC release: The KEYBOARD IS option has not been installed. Only the HP-71's keyboard may be used for input.

Using peripherals with the debugger requires a certain knowledge of HPIL. You should become familiar with operation and control of the loop by consulting the HP-71 HPIL manual.

2.7 Specific to HP-71

The Titan Debugger is HP-71 specific; that is, it will not run on other Saturn CPU systems. Several mainframe utility routines and RAM pointers are used in the debugger, and they are expected to reside at the addresses found in all versions of the HP-71.

The CPU instruction set matches that found in versions 1BBBB and 2CCCC of the HP-71 operating system. If later versions of the Saturn CPU are used, new additional opcodes will not be emulated properly. (However, because the emulator is RAM-based, you can change or add new instructions by changing the code. Guidelines for this are given in Chapter 12.)

P 2-6 BLANK

3. ENTERING AND EXITING THE DEBUGGER

3.1 File Setup

The Titan Debugger LEX files are provided on mass media, such as tape or disk. The LEX files should be copied into IRAMs in preparation for running the debugger. Two of the files, DBGLEX1 and DBGLEX2, are required -- DBGLEX3 is needed only to access several enhancements and options.

Copying DBGLEX3 into memory provides the following capabilities:

- Disassembler, to display opcode mnemonics
- Bias, to apply an offset to displayed addresses

This is the recommended procedure for loading the debugger LEX files:

1. Use the FREE PORT statement to create IRAMs. Unless you have a plug-in RAM module larger than 4K, you will need to free one port for each file. If you have larger RAMs, you can fit two or all three files into a single IRAM. (The HP-71 comes with four 4K RAMs internally, so you do not need to buy more plug-ins unless your assembly language routines need more memory.)

2. Copy DBGLEX1 and DBGLEX2, and, if desired, DBGLEX3 into the IRAMs. The file DBGLEX1 *must* reside at an address ending with "008"; copying it into an IRAM as the first (or only) file guarantees this to be the case. The other two files may reside at any address. All three files may be put in memory in any order.

For example,

```
FREE PORT(0)                          ! Create one IRAM for
FREE PORT(.01)                        ! each debugger LEX file.
FREE PORT(.03)
COPY DBGLEX1:TAPE TO :PORT(.03)
COPY DBGLEX2:TAPE TO :PORT(.01)
COPY DBGLEX3:TAPE TO :PORT(0)
```

Now the debugger is ready to run. Whether the target assembly language is a mainframe routine, a LEX file, a BIN file or a FORTH primitive created by the FORTH/Assembler ROM, you can invoke the debugger and start emulating the code.

3.2 Invoking the Debugger

When first invoked, the debugger needs to be set to the address of the target assembly language. For mainframe routines, consult the HP-71 IDS for the entry point address; for LEX or BIN file use the ADDR$ function to get this. Say you have a BIN file "BINTEST" which you have assembled with the FORTH/Assembler ROM. Before entering the debugger, execute the following:

```
ADDR$("BINTEST")
```

to get the address of the code. Let's assume the function returns the address (a string value) "348E6". Although this is the address of the file header, it is in the general area of the code you will be debugging. You need to remember this value for later entry.

The BASIC statement **DEBUG** is used to first enter the debugger. Type DEBUG and press [ENDLINE]; this will display the debugger's PC (CPU Program Counter) register. It is the PC which points to the opcode for emulation, so setting this register is crucial to further debugging. A discussion of register usage and editing follows in Chapter 6.

Executing **DEBUG** causes the debugger to restore its registers to the values they had when the debugger was last exited, or to zeroes if this is the first entry.

An alternate form of the DEBUG statement is **DEBUG ***, which causes the debugger to take over and begin emulation immediately. It sets up its registers for an exit through the mainframe entry point "NXTSTM", and runs from there. If you use this in a multi-statement command (or in a program line) to exercise your target code, you can make the debugger start emulating your code automatically. Using the example above, if you enter:

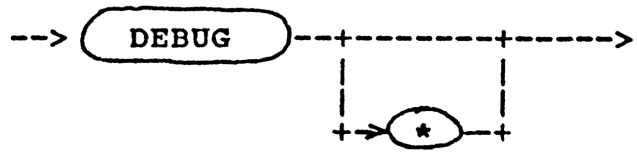
```
DEBUG * @ CALL BINTEST
```

the debugger will take over and begin emulation of you binary file.

Executing **DEBUG *** will cause the previous values of the emulated registers to be lost. The registers take on values necessary for an exit through "NXTSTM".

The **DEBUG** keyword is programmable; but the debugger, except for a few specific actions, is not. Executing **DEBUG** in a program will suspend the HP-71 operating system and throw keyboard control to the debugger, for manual direction. Executing **DEBUG *** acts similarly, but starts emulation automatically. When you exit the debugger, you may continue through your target assembly language, or you can drop into the HP-71's warm start code, where the actions of an INIT 1 are taken.

Syntax:



3.3 Errors in File Setup

The debugger will not prevent you from copying it into any location in memory, but it will generate an error or warning if you have not copied it into "protected" RAM (i.e., IRAM). You can run the debugger if any of the DBGLEX files reside in MAIN memory, but the HP-71 will report warnings, since you should be aware that MAIN memory is much more volatile than IRAMs. In any case, the debugger will error if DBGLEX1 does not reside at an "xx008" address.

Module name	In IRAM	In MAIN	ROM, PROM	Miss-ing	Condition
DBGLEX1	X *1				OK.
DBGLEX1	X *2				WRN:Verify Address
DBGLEX1		X *3			WRN:DGBLEX1 in MAIN
DBGLEX1			X		ERR:Verify Address
DBGLEX1				X	(Err: stmt not found)
DBGLEX2	X				OK.
DBGLEX2		X			WRN:DBGLEX2 in MAIN
DBGLEX2			X		OK.
DBGLEX2				X	ERR:Modules Missing
DBGLEX3	X				OK.
DBGLEX3		X			WRN:DBGLEX3 in MAIN
DBGLEX3			X		OK.
DBGLEX3				X	OK.

*1 : First file in IRAM.
*2 : Not first file in IRAM, but at address "xx008".
*3 : At address "xx008" in MAIN.

The above errors and warnings reported by the DBGLEX1 file show the prefix DEBUG to identify the source; for example

```
DEBUG ERR:Modules Missing
```

Running the debugger with any of the files in MAIN RAM is extremely risky. If you keep at it long enough, a Memory Lost is unavoidable, since the file chain in MAIN memory moves frequently.

Several copies of the DBGLEX files can reside in memory simultaneously. The debugger will choose which modules are in the most stable memory locations (in IRAM, or in the case of DBGLEX2 and DBGLEX3, in IRAM/ROM/PROM), and use these in execution.

You can rename the files as you wish without affecting the debugger's operation. DBGLEX1 contains all the register information needed to restore a debugging session (except for RAM pointers which might have been changed while in the BASIC environment). Hence, you can keep copies of it under different names for different sessions. If you rename the DBGLEX files, the above warnings will report their actual names. However, this manual refers to them as DBGLEX1, DBGLEX2 and DBGLEX3.

** Note for ICC release: The warnings currently report the fixed names DBGLEX1 and DBGLEX2 only.

3.4 Exiting the Debugger

** Note for ICC release: The keyscanning routines of the debugger have not been implemented yet. This section describes the intended keyboard interface for returning to BASIC from the debugger.

** Presently, the keystroke f[Q] exits the debugger, returning to BASIC to execute the next BASIC statement. The [Q][Q], [J][J] and [ON][/] RECOVER keystrokes have not been implemented.

There are four methods of exiting the debugger, with different effects. Only the first is described in detail here, so the first time user can feel comfortable with entering and exiting the debugger.

1. Pressing [Q][Q] (the [Q] key twice, within two seconds) drops the debugger into the HP-71 main loop, where the full BASIC operating system is available. Debugger registers are preserved so that a debugging session may be resumed at later reentry.

2. The RECOVER 2 sequence is used to deactivate the debugger and return complete control of the computer to the HP-71 operating system. This key sequence is described in Chapter 9.

3. Pressing [J][J] (the [J] key twice, within two seconds) while the PC register is in the display performs a hard jump to the PC. This key sequence is described in Chapter 9.

4. The RECOVER 3 sequence performs a Memory Lost, clearing all of memory and deactivating the debugger. This key sequence is described in Chapter 9.

You will find that the [Q][Q] sequence is very useful for interrupting debugging sessions to perform BASIC functions -- edit files, print CATalogs, check and set TIME, RUN programs, change output devices, etc. If you want to resume your debugging session where you left off, execute the DEBUG keyword. This restores all emulated CPU registers (but not necessarily RAM locations) to their values before you performed [Q][Q].

When the BASIC environment is reentered with [Q][Q], the debugger retains control of the keyboard, although this is transparent to the user. This condition (operating in BASIC with the debugger in control of the keyboard) is indicated by the ((*)) annunciator being lit. (Chapter 12 describes how to select a different annunciator if your assembly language application conflicts with this.) Other than the annunciator being on, the computer operates as a normal HP-71.

For more details on leaving the debugger, on different ways to reenter, and on the other methods of exiting, see Chapter 9.

4. EMULATED CPU REGISTERS

This chapter gives general details on the structure and purpose of the debugger stacks and registers. Information for editing and manipulating registers is in the following chapter.

4.1 Debugger Stacks

The debugger's registers are arranged in stacks. There are six stacks of emulated CPU registers, and four control stacks. Figure 2 shows the relative positions of stacks and registers.

Each of the ten stacks can be accessed with a "direct access" digit key, as indicated in Figure 2. In addition, the six CPU stacks make up a revolving set, accessible by scrolling left and right (with wrap-around) with the arrow keys. Within each stack, the registers are viewed by rolling the stack with the up- and down-arrows.

For example, when the debugger is first entered, the User's stack is empty, and the PC register is displayed:

```
     PC displayed:                          PC:00000: 2034EE100060
```

Pressing [>] moves to the A-D stack, at the top level:

```
     [>] to the A-D stack:                  A: 000000000000000 0
     [↓] rolls the stack to B:              B: 000000000000000 0
     [↓] rolls the stack to C:              C: 000000000000000 0
     [↓] rolls the stack to D:              D: 000000000000000 0
     [↓] wrap-around to A:                  A: 000000000000000 0
     [>] to the R0-R4 stack:                R0: 0000000000000000
     [>] to the RSTK stack:                 L0: 00000
     [>] to the D0-IN stack:                D0:00000: 2034EE100060
     [↑] wrap-around to OUT/IN:             OUT:000 IN:0000(0) I:E
```

Pressing [>] now wraps around the set of CPU stacks back to the User's stack; however, if the User's stack is empty, it is skipped, and the PC stack is brought up:

```
     [>] to the PC stack:                   PC:00000: 2034EE100060
```

The control stacks can only be selected by the direct access keys. Pressing [7] brings up the Window stack:

```
     [7] to select Window stack:            00000.2034EE100060F481
     [8] to select Brk Pt stack:             0 BPs. RTN Level:NONE
     [9] to select Options stack:           OPTIONS: podskhlar
```

The direct access keys can also be used to select a CPU stack:

Titan Debugger
Stacks and Registers

CPU Stacks						Control Stacks			
*									
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
User	PC	A-D	R0-R4	RSTK	D0-IN	Home	Window	BrkPts	Options
< >	PC	A	R0	L0	D0	Home	W 1	#BPs	OPTIONS
< >	PCcy	B	R1	L1	D1		W 2	BP 1	TRC
< >	XC/AC	C	R2	L2	ST		W 3	BP 2	BIAS
< >		D	R3	L3	HS		W 4	BP 4	AUTO
< >			R4	L4	OUT/IN		W 5	BP 5	LIST
< >				L5			W 6	BP 6	XQT
< >				L6			W 6	BP 6	
< >				L7			W 7	BP 7	
							W 8	BP 8	
							W 9	BP 9	
							W10	BP10	
<----- [<] -- scroll -- [>] ----->							W11	BP11	
through with arrow keys							W12	BP12	
							W13	BP13	
							W14	BP14	
							W15	BP15	

* direct access keys

PC: CPU Program Counter
PCcy: PC, Carry, Pointer and Mode
XC: Last executed PC
AC: Action Completed
ST: CPU software Status bits
HS: CPU Hardware Status bits
OUT/IN: OUT and IN registers.
A-D: CPU arithmetic registers
R0-R4: CPU scratch registers
L0-L7: Levels 0-7 of CPU RSTK
Home: Last register displayed
W1-W15: RAM/memory windows
#BPs: Number of Break Points
BP1-BP15: Break Point registers
OPTIONS: Control Options
TRC: Trace register
BIAS: Address Bias register
AUTO: Auto disassemble
LIST: List stacks selection
XQT: Single opcode execute

Figure 2

```
[1] to select PC stack:          PC:00000: 2034EE100060
[2] to select A-D stack:         A: 0000000000000000 0
[3] to select R0-R4 stack:       R0: 0000000000000000
[4] to select RSTK stack:        L0: 00000
[5] to select D0-IN stack:       D0:00000: 2034EE100060
```

The "Home" register is the last register displayed before running the emulator. It is useful for tracking the contents of a register between emulation steps. Upon first entering the debugger, the Home register is empty, and displays "< >" to indicate this. Similarly, when the User's stack is empty, selecting it with the direct access key [0] displays "< >".

```
[6] to select Home register:     < >
[0] to select User's stack:      < >
```

4.2 Getting to Know the Stacks

4.2.1 User's Stack

The User's stack is an expandable set of registers, from zero to eight, selectable by the user. This stack has two special purposes:

1. It makes for convenient access to the most fequently viewed registers.
2. It is the default display stack for several operations, such as hitting a break point during emulation.

The AC annunciator is turned on when the User's stack is in the display. This will help you keep track of where you are in the stack structure. (Chapter 12 describes how to select a different annunciator if your assembly language application conflicts with this.)

** Note for ICC release: The AC annunciator is not used in the ICC release. This will be implemented later.

4.2.2 PC Stack

The PC stack is the principal stack for the debugger. It directs the emulator to the address for emulation, provides a one-step history of emulation, and shows important CPU control registers.

4.2.2.1 PC register. This shows the current PC (CPU Program Counter) for emulation, and 12 nibbles of memory at that address.

4.2.2.2 PCcy register. This shows the current PC, the Carry, Pointer and arithmetic Mode. The Carry is shown as "1" or "0". The Pointer is shown as a hex value, "0" to "F". Mode is shown as either "H"

(Hex) or "D" (Dec).

4.2.2.3 XC/AC register. This shows the address of the last executed PC, and the Action Completed -- the reason for interrupting the emulator.

4.2.2.3.1 AC codes. The Action Completed codes displayed in the XC/AC register are as follows:

-ATTN-	The emulator was stopped by pressing the [ATTN] key. This is also the AC code when first entering the debugger.
S STEP	The emulator was single-stepped through an opcode, either by pressing the [S] key, or the [M] key.
M STEP	The emulator was macro-stepped through a subroutine with the [M] key, breaking on the return.
BRK PT	The emulator was stopped when a break point was encountered during running.
SHUTDN	The emulator encountered a "SHUTDN" instruction (807 opcode), along with option "S" being set.
X RSTK	The emulator's RSTK stack was exceeded; that is, a return level flagged for a break was pushed off the end of the RSTK. A return level can be flagged for a break through macro-stepping (which breaks when the subroutine returns), or with the RTN Level indicator in the Break Point stack.
RTN BK	The emulator was stopped when a return level break occurred. A return level break is set with the RTN Level indicator in the Break Point stack.
IN PROMPT	An "A=IN" or "C=IN" instruction was encountered (opcodes 802 or 803), along with the "?Prompt" setting in the IN register.
INITLZ	The debugger was initialized, or reset, with the "INITIALIZE" command in the g[CMDS] menu.
UNCNFG	The emulator encountered an "UNCNFG" instruction (opcode 804) which would unconfigure the device containing DBGLEX1 and/or DBGLEX2. The instruction cannot be emulated.
RESET	The emulator encountered a "RESET" instruction (opcode 80A). The instruction cannot be emulated.

SHUTDN/PC=0 The emulator encountered a "SHUTDN" instruction (opcode 807) with the emulated OUT register having bit 0 cleared. Further emulation will result in a "Memory Lost".

4.2.3 A-D Stack

The four registers in the A-D stack emulate the arithmetic registers in the CPU: A, B, C and D.

Since the P register (Pointer) greatly affects operations on the arithmetic registers, the value of P is indicated by separation within the digits of the registers. For instance, a P value of 3 causes the registers to show nibble 3 separated from the rest:

```
A: 6D031F9003E4 C 125          indicates P=3.
A: 6D03 1 F9003E4C125          indicates P=B  (=11dec).
A: 6 D031F9003E4C125           indicates P=F  (=15dec).
A: 6D031F9003E4C12 5           indicates P=0.
```

4.2.4 R0-R4 Stack

The five registers in the R0-R4 stack represent the emulated CPU scratch registers: R0, R1, R2, R3 and R4.

Note that the low five nibbles of R4 are unusable in assembly language, reserved for use in the interrupt routine. Since the debugger does not perform emulated interrupts, these nibbles do not change "randomly" as they do in real-time execution. The low five nibbles of the R4 register cannot be depended upon to represent the true contents of the real CPU.

4.2.5 RSTK Stack

The eight registers in the RSTK stack represent the emulated return levels: L0 through L7. The L0 register is the "top" return stack level -- the next one to be popped off for a "RTN" or "C=RSTK" instruction.

4.2.6 D0-IN Stack

The five registers in the D0-IN stack keep track of data pointers, status bits and the OUT and IN registers.

4.2.6.1 D0 register. This shows the address value in D0 and 12 nibbles from memory at that address.

4.2.6.2 D1 register. Similar to D0, but with the D1 address.

4.2.6.3 ST register. This shows the 16 CPU software status bits, in groups of four for readability. S15 is the left-most digit; S0 is the right-most digit.

4.2.6.4 HS register. This shows the 4 CPU hardware bits:

```
MP:0  SR:0  SB:0  XM:0
```

Where

- MP is the Module Pulled bit
- SR is the Service Request bit
- SB is the Sicky Bit
- XM is the External Module Missing bit

4.2.6.5 OUT/IN register. This shows three emulated read-only or write-only CPU registers. In the emulator, as in the real CPU, the OUT and Interrupt registers are write-only. However, the debugger keeps track of what values were last written to these registers. Similarly, the IN register is read-only, but the debugger allows you to specify what value to read for an "A=IN" or "C=IN" instruction.

For example, from the following display:

```
OUT:00F IN:0800(4) I:E
```

- The OUT register currently contains the value "00F", as last written to by an "OUT=C" or "OUT=CS" instruction (or as edited by you).
- By editing the IN register, you have specified that an "A=IN" or "C=IN" instruction will read the value "0800" into the low four nibbles of A or C, given that bit 2 (the 4's bit) of the OUT register is set. For detailed explanation of this, along with information on other ways to specify the IN register, see Chapter 7.
- The CPU Interrupt status is Enabled. This register is not editable, but is set by the emulator to "E" by an "INTON" (opcode 808n, where n is any nibble other than F), and to "D" by an "INTOFF" (opcode 808F).

The Interrupt register is shown for information only. Since the debugger does not emulate interrupts, there is no real effect caused by its setting.

4.2.7 Home register

The Home register is that register which was displayed at the last time the emulator was run. The Home register is reset upon pressing [S], [M] or [RUN].

The direct access key [6] recalls the Home register. This is useful

for tracking a register between steps of the emulator, when you might be looking at several different stacks.

4.2.8 Window Stack

The debugger provides 15 memory windows, W1 through W15. When first entering the debugger, they are all set to address 00000. Each window register shows 16 nibbles of memory for the address specified. There are three types of memory window available:

Direct window	Displays 16 nibbles from window address.
Indirect window	Reads address at window and displays 16 nibbles from there.
Relative window	Reads a 5-nibble offset from the window, uses it to compute the memory address, and displays 16 nibbles from there.

Chapter 7 tells you how to select the different types of windows. The default type is "Direct", indicated with a "." symbol in the display.

4.2.9 Break Point Stack

The Break Point stack always contains at least one register -- the #BPs register. Up to 15 break points can be added to this stack.

```
0 BPs. RTN Level:NONE
```

The #BPs register indicates how many break point addresses are in the stack. The RTN Level indicator tells at which RSTK level the emulator will break: 0-7, ALL or NONE.

Registers BP1 through BP15 (the break point addresses) can be entered and deleted by the user. A typical break point register looks like this:

```
BP 3~00209  (D1  )Halt
```

The address "00209" tells the emulator to break if the PC executes an instruction at this address (whether the opcode begins at this address, or if it spans this address). Also, the emulator will break if a data operation (reading or writing to memory with DAT0 or DAT1) occurs at this address (again, if the read/write begins or spans this address).

The register name in parentheses is the Show register -- this register is to be displayed if the break point is encountered. The term "Halt" is the Break Point Action. You can choose between three actions: "Halt", "Cont" or "Jump".

** Note for ICC release: In the current version of the debugger, only the "Halt" action works. You can select "Cont" or "Jump", but they will only halt.

If the break point stack is not full (less than 15 break points entered), the last level appears as:

```
BPnn~_____  (__  )Halt
```

indicating that break point #nn is next to enter. When deleting, entering or editing break points, they are sorted in increasing order, which greatly speeds up the emulator break point checking.

Details on adding, deleting or editing break points, and on the Show and Action registers, are included in Chapter 7.

4.2.10 Options Stack

This stack contains six registers to control the options and flow of the debugger.

4.2.10.1 OPTIONS Register.

** Note for ICC release: The OPTIONS register has not been finalized, nor are the options implemented. This section describes how the options are intended to work.

The OPTIONS register shows nine options available to the user. Each letter stands for a different option, a capital letter indicating the option is set, and a lower case letter indicating cleared.

```
OPTIONS: podskbrce
```

4.2.10.1.1 **P option** causes the PC register to be displayed along with the Show register for a Break Point or Trace.

4.2.10.1.2 **O option** causes an "OUT=C" or "OUT=CS" (opcodes 800 and 801) to be suppressed during emulation. That is, the true OUT register will not be written to.

4.2.10.1.3 **D option** causes the PC register to display the disassembled (decompiled) opcode at its address, rather than 12 nibbles. The disassembler is available only if DBGLEX3 is in memory.

4.2.10.1.4 **S option** causes the emulator to break at a "SHUTDN" instruction (opcode 807). With "S" clear, emulation will continue through a "SHUTDN". However, irregardless of the "S" setting, encountering a "SHUTDN" with bit 0 of the OUT register set to zero will always cause an emulator break, with an AC code of "SHUTDN/PC=0".

4.2.10.1.5 **K option** causes the emulator to beep when emulation is stopped for any reason other than AC codes of "-ATTN-" and "S STEP". (Note that "RESET", "UNCNFG" and "SHUTDN/PC=0" will always beep, regardless of the "K" option.)

4.2.10.1.6 **B option** causes the address bias (in the BIAS register) to be applied to the address registers. For a list of these registers, see the description for the BIAS register, below.

The bias option can only be active if DBGLEX3 is in memory.

4.2.10.1.7 **R option** causes the debugger to handle service requests when it is idle in view mode.

4.2.10.1.8 **C option** causes the debugger to allow the mainframe configuration routines to execute properly if encountered by the emulator. If this option is not set, the configuration routines will cause a break at the "RESET" intruction at address 10233. See Chapter 11 for details.

4.2.10.1.9 **E option** causes the debugger to output its display to the HPIL display device only when you press [ENDLINE]. If this option is not set, every display will be sent to the display device as you view registers. Regardless of the E option setting, Break Point Show and Trace Show registers are always sent to the display device. If the HPIL display device is not active, this option has no effect.

4.2.10.2 **TRC Register.** The trace register allows you to specify a register for continuous display during emulation, as well as the frequency of display.

```
TRC #0001A  (D1  )Cont
```

The appearance of the TRC register is similar to that of a Break Point register. However, the first 5 digits are a count (in hex -- the example specifies 1Ahex=26dec) of the number of emulated opcodes between displays. The Trace Show and Trace Action fields are identical to those of Break Point Show and Action described above.

4.2.10.3 **BIAS Register.** The BIAS option can only be active if DBGLEX3 is present in memory.

The BIAS register allows you to adjust displayed addresses by a 5-nibble hex value. This is handy for following code on a printout where the assembler address does not match the machine address, as is most often the case. The bias is only applied to address registers if the "B" option is set in the OPTIONS register.

The registers for which the bias is applied are the following. The bias is only applied at time of display, not within the register itself.

- PC and PCcy registers (PC address)
- XC register
- L0 through L7
- D0 and D1
- W1 through W15
- BP1 through BP15
- AUTO DSASSMBL register

The BIAS register shows three hex fields. For example,

```
BIAS-6A000,6A000,6C03C
```

specifies that any displayed address between the values 6A000 and 6C03C, inclusive, would first be decreased by 6A000 before displaying. Thus, a machine address of 6B012 would be displayed as 01012.

4.2.10.4 **AUTO DSASSMBL Register.** The AUTO DSASSMBL option can only be active if DBGLEX3 is in memory.

The AUTO DSASSMBL register specifies an address which is used for automatic opcode disassemble (decompile).

For example, if the AUTO DSASSMBL register is set as follows:

```
AUTO DSASSMBL @ 6B012
```

pressing f[AUTO] causes the debugger to display disassembled opcodes continuously until you press [ATTN]. Opcodes are disassembled sequentially in memory, not by execution flow. The address in the register is updated as each opcode is displayed.

4.2.10.5 **LIST STACK Register.** The LIST STACK register specifies which stacks to list, in their entirety, when f[LIST] is pressed. The stacks are numbered 0 through 9, corresponding to the direct access keys. For example,

```
LIST STACK: 01...5.7..
```

indicates that stacks 0 (User's), 1 (PC), 5 (D0-IN) and 7 (Window) will be displayed when f[LIST] is pressed.

4.2.10.6 **XQT Register.** The XQT register allows you to execute single opcodes through the emulator. Opcodes are entered as hex digits, and entering an opcode causes the emulator to be stepped once, as if the opcode were found at the current PC.

5. OPERATING IN VIEW MODE

The debugger operates essentially in two modes: view mode and edit mode. In view mode, the cursor is off, the arrow keys scroll through the stacks, and the digit keys are used as direct access keys to select stacks.

In edit mode, the cursor is on, the right- and left-arrow keys control the cursor, and the digit keys are used to edit register contents. In most cases, the up- and down-arrow keys are inactive.

Pressing [Z] (the key with the "EDIT" typing aid) switches to edit mode, and allows you to edit the displayed register. The next chapter, "Editing Emulated Registers", explains that mode in detail.

In view mode, all keys are direct-execute. That is, they are executed immediately, with no need to press [ENDLINE].

5.1 View Mode Keys

5.1.1 Cursor Keys

The four cursor, or arrow, keys, [←], [→], [↑], and [↓], are used to scroll through the set of six CPU stacks. As each stack is brought up, the top level is displayed. Figure 2 illustrates the stack structure, and Chapter 4 provides descriptions.

5.1.2 Digit Keys

The nine digit keys, [0] through [9], are used as direct access keys to bring up the corresponding debugger stacks. Figure 2 shows the direct access key for each stack.

5.1.3 [ATTN]

Pressing the [ATTN] key in view mode clears the key buffer and rebuilds the current register in the display. As in the HP-71 operating system, the debugger allows type-ahead, storing 15 keystrokes in its key buffer. Pressing [ATTN] clears any type-ahead keys, avoiding action on them.

5.1.4 f[OFF]

Pressing f[OFF] causes the HP-71 to turn off, with the debugger still active. It also clears the debugger's key buffer, rebuilds the LCD row drivers and resets display contrast to the default value of "9".

The f[OFF] keystroke is most useful for restoring the display in the case that emulated routines have written to the display control area. The debugger can be turned back on with the [ATTN] key.

It is not recommended that you turn off your computer for any length of time when you are in the debugger, as it does not process wakeups as comprehensively as the mainframe does. To turn the computer off for any length of time, you should drop into the BASIC environment using one of the four methods described in Chapter 9.

5.1.5 [Q][Q]

Pressing [Q][Q] (the [Q] key twice, within two seconds) exits the debugger, back to the BASIC main loop, in effect executing an INIT 1. See Chapter 9 for details.

** Note for ICC release: The [Q][Q] keystroke has not been implemented. The f[Q] keystroke is used to exit the debugger.

5.1.6 [J][J]

Pressing [J][J] (the [J] key twice, within two seconds) causes a hard jump to the address in the PC register. This keystroke can only be executed if the PC or PCcy register is in the display. See Chapter 9 for details.

** Note for ICC release: The [J][J] keystroke has not been implemented. The f[Q] keystroke is used to exit the debugger.

5.1.7 [Z]

Pressing [Z] (the key with the "EDIT" typing aid) enters edit mode. The next chapter, "Editing Emulated Registers", explains that mode.

5.1.8 f[EDIT]

Pressing f[EDIT] allows you to edit *memory contents* in an address register. See the next chapter for details.

5.1.9 [RUN]

Pressing [RUN] runs the emulator, starting at the address in the PC register. Emulation continues until a break is encountered, corresponding to one of the AC codes described in the section "XC/AC register" in the last chapter. You can always break the emulator with the [ATTN] key; other breaks are caused by break points or special opcodes. Chapter 8 provides more details on running the editor.

5.1.10 [S]

Pressing [S] single-steps the emulator, at the address in the PC register. Chapter 8 provides more details on running the editor.

5.1.11 [M]

Pressing [M] macro-steps the emulator, starting at the address in the PC register. A macro-step is identical to a single-step, except when a GOSUB (or GOSUBL or GOSBVL) instruction is executed. Then the emulator runs through the subroutine, breaking at the return (or any other break point if it occurs within the subroutine). Chapter 8 provides more details on running the editor.

5.1.12 Memory Windows

Most debugger registers will display a memory window or opcode mnemonic from the address (or address field) they contain. These registers are:

- PC and PCcy
- XC
- The arithmetic (A-D) and scratch (R0-R4) registers, which frequently hold an address in the low five nibbles.
- L0 through L7
- D0 and D1
- W1 through W15
- BP1 through BP15
- AUTO DSASSMBL

5.1.12.1 **Momentary Nibble Display.** Several keys are used to display memory windows from the address registers. The window is displayed until the key is let up.

5.1.12.1.1 [.] Pressing [.] momentarily displays a direct window -- 16 to 18 nibbles read directly from the register address.

5.1.12.1.2 g[<] Pressing g[<] momentarily displays an indirect window -- the 5 nibbles at the register address become the address of the window.

5.1.12.1.3 g["] Pressing g["] momentarily displays a relative window -- the 5 nibbles at the register address are used as an offset to compute the address of the window.

5.1.12.2 **Momentary Disassemble Windows.** Several keys are also used to display disassembled (decompiled) opcodes from these registers. The mnemonic is displayed until the key is let up.

The disassembler is only available if DBGLEX3 is in memory.

** Note for ICC release: The disassembler has not been installed. These keystrokes will display "Not Implemented".

5.1.12.2.1 [,] Pressing [,] momentarily displays a direct disassemble -- the opcode mnemonic read directly from the register address.

5.1.12.2.2 g[>] Pressing g[>] momentarily displays an indirect disassemble -- the 5 nibbles at the register address become the address of the mnemonic.

5.1.12.2.3 g[#] Pressing g[#] momentarily displays an indirect disassemble -- the 5 nibbles at the register address are used as an offset to compute the address of the mnemonic.

5.1.13 Address increment/decrement

Several debugger registers are "address monitors" -- that is, they point to areas of memory which might require frequent changes, or they themselves might need to be adjusted often to control the flow of a debugging session. These registers are:

- PC and PCcy registers (PC address)
- D0 and D1
- W1 through W15
- BP1 through BP15
- AUTO DSASSMBL register

Several keys can be used to increment or decrement the address in these registers, without switching the debugger to edit mode.

[+] Increments address by 1.
[*] Increments address by 10hex (16dec).
[-] Decrements address by 1.
[/] Decrements address by 10hex (16dec).

5.1.14 [ENDLINE]

In view mode, pressing [ENDLINE] sends the display contents to the HPIL display device, if it is active. If the "E" option is set, this is the only time registers are sent to the display device (except for a Break Point Show or Trace Show display).

** Note for ICC release: The [ENDLINE] keystroke has not been installed. All displays are sent to the HPIL display device, and this capability cannot be disabled.

Pressing g[CMDS] gains access to a menu of special commands

5.1.15 g[CMDS]

** Note for ICC release: The g[CMDS] keystroke has not been installed. This section describes the planned implementation of the debugger commands. The KEYBOARD IS option has not been installed, and use of the DISPLAY IS device is fixed.

Pressing g[CMDS] gains access to a menu of special commands which control the debugger. There is a menu of five commands which can be scrolled up and down with the arrow keys. When the desired command is in the display, pressing [ENDLINE] will execute it. The commands menu can be aborted by pressing the [ATTN] key.

Debugger commands menu:

```
INITIALIZE
Ext DISPLAY on
Ext DISPLAY off
Ext KEYBOARD on
Ext KEYBOARD off
```

5.1.15.1 INITIALIZE. Executing this command resets the entire debugger, clearing all registers and buffer areas. The debugger will prompt

```
PC:_____
```

to indicate all registers have been reset.

To protect the debugger from emulating itself into oblivion, the INITIALIZE action sets one break point at address 00209, the address where the cold start routine clears the VECTOR address. Since the debugger needs to be able to intercept the interrupt vector, emulating this instruction would be self-destructive. You can delete this break point if you like, but leaving it in the stack will provide you with some protection in case the debugger is run from address 00000. See Chapter 11 for details.

5.1.15.2 Ext DISPLAY on. Executing this command allows you to use an external DISPLAY IS device with the debugger. You must have executed the DISPLAY IS command in BASIC before entering the debugger, as you can only activate and deactivate the display -- not designate it -- within the debugger. There are special considerations when using DISPLAY IS with the debugger, as some code cannot be emulated while DISPLAY IS is active. See Chapter 11 for details.

5.1.15.3 Ext DISPLAY off. Executing this command causes the debugger's output to go only to the LCD. In many situations, an external display device cannot be used, because the mainframe display code or HPIL code would interfere with the emulation.

5.1.15.4 Ext KEYBOARD on. Executing this command allows you to use an external KEYBOARD IS device with the debugger. You must have executed the KEYBOARD IS command in BASIC before entering the debugger, as you can only activate and deactivate the keyboard -- not designate it -- within the debugger. There are special considerations when using KEYBOARD IS with the debugger, as some code cannot be emulated while KEYBOARD IS is active. See Chapter 11 for details.

5.1.15.5 Ext KEYBOARD off. Executing this command causes the HP-71 keyboard to be the only input device. In most situations, an external keyboard cannot be used, because the polls and system buffers would interfere with emulation.

5.1.16 f[USER]

To add a register to the User's stack, press f[USER] when the desired stack is in the display. A beep will be sounded to signal that it has been pushed onto the User's stack. This does not make the User's stack the current stack; to do that, press the direct access key [0].

If the register is already in the User's stack, f[USER] will be ignored. A maximum of eight registers may reside in the User's stack. If a ninth register is pushed, the last register in the stack is lost from the stack.

A break point can be selected for the User's stack. As break points are reordered or deleted, the register number will change accordingly. If a break point register in the User's stack is deleted, the corresponding level in the User's stack will be deleted (as if g[lUSER] were pressed -- see below).

** Note for ICC release: The current version of the debugger does not allow break points to be put in the User's stack.

5.1.17 g[lUSER]

To remove a register from the User's stack, select the User's stack and bring up the desired register in the display. Pressing g[lUSER] removes it from the stack.

When all eight levels of the User's stack have been deleted, selecting the stack will show an empty register: "< >".

5.1.18 f[DISP]

Pressing f[DISP] momentarily displays the HP-71's emulated display buffer, which otherwise cannot be seen since the debugger overwrites the LCD.

** Note for ICC release: The current version of the debugger uses the mainframe display routines, so the f[DISP] keystroke will

rebuild the current debugger register. The emulated display buffer cannot be recovered.

5.1.19 f[DELETE]

Pressing f[DELETE] resets the register to the default contents. In almost all cases, this means to set all digits to zeroes. Special cases are as follows:

5.1.19.1 **Break Point register.** Deleting a break point register removes it from the Break Point stack, reducing the number of break points by one.

5.1.19.2 **Mode.** In the PCcy register, Mode is reset to show "M:H" (Hex).

5.1.19.3 **RTN Level indicator.** Resets the RTN Level indicator in the #BPs register to "NONE".

5.1.19.4 **OPTIONS register.** Resets all options to "off" (all lower case).

5.1.19.5 **LIST register.** Resets all stack numbers to "." (all stacks deselected).

5.1.19.6 **BP Show and Trace Show.** Resets the Show field to "__" (null), which will cause the User's stack (or, if empty, the $\overline{PC}$ register) to be displayed at a break.

5.1.19.7 **BP Action and Trace Action.** Resets the Action field to "Halt".

5.1.20 f[PURGE]

Pressing f[PURGE] resets the entire current stack. In this way, you can "delete" all break points, or set registers A through D to zeroes, for example. This keystroke is ignored in the User's stack.

5.1.21 f[LIST]

Pressing f[LIST] sends to the display all stacks selected in the LIST STACK register. The current register is redisplayed after this operation.

5.1.22 f[AUTO]

Pressing f[AUTO] invokes the disassembler at the address specified in the AUTO DSASSMBL register. Opcode mnemonics are sent to the display continuously until [ATTN] is pressed. Opcodes are disassembled sequentially in memory, not by execution flow. The address in the AUTO DSASSMBL register is updated as each mnemonic is displayed.

The disassembler is available only if DBGLEX3 is in memory.

** Note for ICC release: The disassembler has not been installed.
The f[AUTO] keystroke displays "Not Implemented".

6. EDITING EMULATED REGISTERS

As explained in the last chapter, "Operating in View Mode", you can change the contents of registers without the need to edit them -- such as DELETE, PURGE and increment/decrement. But these methods are limited; for full control over the contents of debugger registers, you must enter edit mode.

Pressing [Z] (the key with the "EDIT" typing aid) switches to edit mode, and allows you to edit the displayed register. For address monitor registers (see section 6.1.2 below) pressing f[EDIT] enters edit mode and allows editing of the *memory contents* at the address. When in edit mode, the cursor blinks in the display, indicating that the digit or character at that position is ready to be changed.

Every register except XC/AC is editable, although certain registers will accept only appropriate digits, such as 0 and 1 for status bits.

When in edit mode, the debugger will respond only to edit keys. That is, stacks cannot be scrolled, nor can the emulator be run. The normal method for leaving edit mode is to press [ENDLINE], which enters the edited register contents and returns to view mode.

The following keystrokes will exit edit mode:

[ENDLINE] Enters the edited contents.

f[DELETE] Resets the register to the default contents. See the "f[DELETE]" section in the previous chapter. f[DELETE] may be executed in edit or view mode.

f[PURGE] Resets the entire current stack. See the "f[PURGE]" section in the previous chapter. f[PURGE] may be executed in edit or view mode.

[ATTN] Pressing [ATTN] aborts the edit, returning the register to its previous contents. It also clears the debugger's key buffer.

f[OFF] Same action as [ATTN], plus turns off the computer. See the section in the last chapter which describes the f[OFF] keystroke.

6.1 Edit Mode Keys

This section describes the general use of keys in edit mode. For specific details of each register, see Chapter 7.

Some specialized editing keys are described only in Chapter 7, in the sections dealing with the appropriate registers. In particular, consult that chapter for details on:

- IN register editing
- RTN Level indicator
- Break Point Show register and Action
- Trace Show register and Action
- OPTIONS editing
- BIAS sign
- LIST STACK editing

6.1.1 [Z]

The [Z] key (the key with the "EDIT" typing aid) is used to enter edit mode for changing the contents of a debugger register.

6.1.2 f[EDIT]

Several debugger registers are "address monitors" -- that is, they point to areas of memory which might require frequent changes, or they themselves might need to be adjusted often to control the flow of a debugging session. These registers are:

- PC and PCcy registers (PC address)
- D0 and D1
- W1 through W15
- BP1 through BP15
- AUTO DSASSMBL register

The f[EDIT] key is used to edit *memory* at the address shown in an address monitor register. In the Window stack, even though a register might be an indirect or relative window, f[EDIT] always works on the memory *at the address shown in the display*.

f[EDIT] may be pressed in view mode or in edit mode. Repeatedly pressing f[EDIT] allows you to pull up indirect addresses for memory editing. For example, at address 00005 of the mainframe you will find the nibbles: E1000. If the PC is set to address 00005, pressing f[EDIT] will allow you to edit memory at that address (although, in this case, 00005 is in ROM, so the contents cannot be changed). At address 0001E are the nibbles 47190; at address 09147 are the nibbles 0003A.

```
PC in display:                      PC:00005: E100060F4812
f[EDIT] (cursor on 'E'):            00005:E100060F4812C136
f[EDIT] (cursor on '4'):            0001E:471901150719O2AF
f[EDIT] (cursor on '0'):            09174:003A41CFBE2AF8EF
   and so on.
```

The debugger provides no protection for editing memory. Not only can you write to private files, system pointers and even the debugger itself, but there is no check made on the type of device. Using f[EDIT] for memory in ROM will do no harm (and do nothing), but using it in EEPROM may wipe out the device's memory.

6.1.3 Digit keys [0]-[9], [A]-[F]

In edit mode, the digit keys [0]-[9] and [A]-[F] enter hex digits to change the contents of the register or memory, as applicable. In some registers, such as ST, only 0 or 1 are allowed.

6.1.4 Cursor keys

The two cursor keys, [<] and [>], are used to move the cursor to an editable digit or character. The two shifted cursor keys, g[<] and g[>], move the cursor to far-left and far-right, respectively.

6.1.5 [ATTN]

See the description of [ATTN] in the previous chapter. [ATTN] returns the debugger to view mode.

6.1.6 f[OFF]

See the description of f[OFF] in the previous chapter. f[OFF] returns the debugger to view mode.

6.1.7 f[DELETE]

See the description of f[DELETE] in the previous chapter. f[DELETE] clears the current register.

6.1.8 f[PURGE]

See the description of f[PURGE] in the previous chapter. f[PURGE] clears the entire current stack.

6.1.9 Menu Editing Keys

Several registers allow speed editing by means of a stack menu. Using this, you can set the address in your register to the value in any other debugger register.

The menu edit registers are the following, with examples of their displays:

- PC register: PC:00000=__
- D0 and D1 registers: D0:00000=__
- W1 through W15: W 1/00000=__ Type/D.
- BP1 through BP15: BP 1~00000=__ (..)..
- AUTO DSASSMBL: AUTO@00000=__

In these registers, menu editing is indicated by the "=__" symbol that follows the address. To use menu editing, you must first enter edit mode with [Z], then press [=]. This moves the cursor over the "=" character, and gives you a window on the debugger register stacks.

Then the arrow keys, along with the direct access keys, can be used to select any register in the stacks.

Say you have just stepped the emulator through a few instructions, and found that you need to set the D1 register to the value in the C register (A field). Here's how to use menu editing to do this:

```
[5] to select D0-IN stack:          D0:2F43C: 000000000303
[↓] to D1 register:                 D1:2F599: 000A3000A300
[Z] for edit mode, cursor on '2':   D1:2F599=__
```

The null menu entry signals that the address is not being copied from any debugger register; the address is the last value found in the edit buffer.

```
[=] for menu edit, cursor on '=':   D1:2F599=__
```

Now the digit keys [0] through [9] act as direct access keys for the debugger stacks, and the arrow keys are used to scroll through the stack registers.

```
[2] to select A-D stack in menu:    D1:04009=A
```

The address in the A register (A field) has been copied into the D1 edit buffer.

```
[↓] to B register in menu:          D1:00000=B
```

The A field of B has been copied into the D1 edit buffer.

```
[↓] to C register in menu:          D1:3A558=C
```

The A field of C has been copied into the D1 edit buffer.

```
[ENDLINE] to enter new D1 value
    and return to view mode:        D1:3A558: 15AF215AF208
```

During menu editing the User's stack is ignored. Pressing the direct access key [0] instead shows the null menu entry. (Any register in the User's stack can be found in its normal place in the debugger stacks.)

When editing a break point, the Show and Action fields, because of the limited display, both appear as ".." to indicate their presence:

```
    [Z] to edit:                    BP 1~00000=__   (..)..
```

They are restored when you return to view mode, or when you edit either of the fields.

All registers can be selected in the edit menu, but some do not have valid addresses associated with them. The following registers, if called up in the edit menu, will show the address 00000:

- ST register
- HS register
- OUT/IN register
- #BPs register
- OPTIONS register
- TRC register
- BIAS register
- LIST STACK register
- XQT register

7. REGISTER DETAILS

This chapter explains each register in detail, with information on special editing actions. Refer to the two previous chapters, "Operating in View Mode", and "Editing Debugger Registers" for descriptions of terms. Those chapters also describe general procedures for viewing and editing.

7.1 PC Stack

7.1.1 PC Register

- [Z] to edit PC address
- Menu editing
- f[EDIT] to edit memory contents
- Address increment/decrement
- Memory window keys

The PC specifies the address for emulation. Both the PC and PCcy registers display the same PC address. The PC address is updated continuously during emulation, between each step.

The PC register displays 12 nibbles from memory or the disassembled opcode at the PC address, according to the "D" option setting. However, the disassembler is only available if DBGLEX3 is in memory.

7.1.2 PCcy register

- [Z] to edit PC, Carry, Pointer and arithmetic Mode
- f[EDIT] to edit memory contents
- PC address increment and decrement
- Memory window keys

Both the PC and PCcy registers display the same PC address. The PCcy register does not use menu editing. The Carry register accepts only digits 0 or 1. The Pointer register is displayed as a single hex digit, 0 through F. The Mode is displayed as, and accepts in editing, only "D" for decimal, and "H" for hex.

7.1.3 XC/AC register

- Not editable
- Memory window keys

The XC address is copied from the PC before the PC is updated during emulation. The AC codes are described in Chapter 4.

7.2 A-D Stack

The four registers in this stack are identical in their operation.

- [Z] to edit contents
- Memory window keys, for A field address

The value of the Pointer is reflected in these registers by separation of the digit to which it points. For example, if P=3, then digit 3 (4th from right) will be separated by single blanks from the others.

If you want to edit the C register, you can also use the XQT register to perform an "LCHEX" instruction.

7.3 R0-R4 Stack

The five registers are identical in their operation.

- [Z] to edit contents
- Memory window keys, for A field address

7.4 RSTK Stack

The eight registers are identical in their operation.

- [Z] to edit contents
- Memory window keys

You can push and pop the debugger's RSTK by executing opcodes "06" ("RSTK=C") and "07" ("C=RSTK") in the XQT register. Executing "GOSUB" and "RTN"-type instructions with XQT will also affect the stack, but these change the PC.

You can set a one-time break at a return level, or specify a break at all returns, with the RTN Level indicator in the #BPs register.

7.5 D0-IN Stack

7.5.1 D0 and D1 registers

These two registers are identical in their operation.

- [Z] to edit address
- Menu editing
- f[EDIT] to edit memory contents
- Address increment/decrement
- Memory window keys

7.5.2 ST register

- [Z] to edit status bits

The status bits are separated into groups of four, for readability. They are ordered the same as in standard CPU representation: S15 is the left-most digit, S0 is the right-most digit. Only digits 0 and 1 are accepted during editing.

7.5.3 HS register

- [Z] to edit status bits

The hardware status bits are identified by name, and their relative positions are the same as in standard CPU representation.

- MP: Module Pulled bit
- SR: Service Request bit
- SB: Sticky Bit
- XM: External Module missing bit

Only digits 0 and 1 are accepted during editing.

7.5.4 OUT/IN register

- [Z] to edit contents

Although the OUT register is not truly readable, the debugger maintains a record of the last value from "OUT=C" or "OUT=CS". The IN register is a psuedo-register, in that it is used only to supply a value for an "A=IN" or "C=IN" instruction. The Interrupt register is for information only, and reflects the last "INTON" ("E" for "Enabled") or "INTOFF" ("D" for "Disabled") instruction. It is not editable, and has no real effect on the emulator.

There are four ways to specify the contents of the IN register. They are selected by pressing certain keys when the cursor is on the first digit of the IN register (the fourth digit in the display).

** Note for ICC release: In the current version of the debugger, the only option for the IN register setting is "Digits". The "KEY#", "KEYBD" and "?Prompt" settings are not installed.

7.5.4.1 Digits. This is the default method. When first entering the debugger, the OUT/IN register reads:

```
OUT:000 IN:0000(0) I:E
```

which specifies that "C=IN" will read all zeroes. Select this method by pressing any digit key ([0]-[9] or [A]-[F]) when the cursor is on the first digit of the IN register.

The digit in parentheses indicates which OUT-register bits to mask for. For instance,

```
OUT:000 IN:0800(4) I:E
```

Bit 2 (the 4's bit) of the OUT register corresponds to the key row [A] through [*] (second row from top). Thus, if the emulated OUT register has this bit set, a "C=IN" instruction will read "0800" -- the [5] key -- in this example. If bit 2 of the OUT register is not set, zeros will be read.

Other examples:

IN register specifies:	Will read only:
OUT:00F IN:0400(3) I:E	[0] or [1] keys.
OUT:00F IN:0012(8) I:E	[W] or [T] keys.
OUT:00F IN:1C00(E) I:E	digit keys [1] through [9].

This method of specifying the IN register allows the most flexibility, for you are not limited to one key, and you have available all input lines.

7.5.4.2 **KEY#**. You can specify one key for the input register. This will cause an "A=IN" or "C=IN" instruction to read the appropriate input value only when the right keyboard row is energized in the output register.

To select this method of specifying the IN register, press f[KEY] when the cursor is in the first IN position. The display will prompt:

```
OUT:000 IN:KEY#__  I:E
```

and you can fill in the key number. Key numbers are found in the HP-71 Owner's Manual, page 123. They are to be entered in decimal form, from 00 to 56. Key number 00 is a special code for the debugger which means "read 0000". Key numbers greater than 56, or containing hex digits, are rejected; an error beep is issued and the debugger waits for proper input.

Note that key numbers are a subset of the digit method of specifying the IN register. For example, the following two displays are equivalent, and specify the [5] key:

```
OUT:000 IN:KEY#26  I:E
OUT:000 IN:0800(4) I:E
```

7.5.4.3 **KEYBD**. You can have the emulator read the actual HP-71 keyboard when it encounters an "A=IN" or "C=IN" instruction when running. This will cause the low four bits of the emulated OUT register to be copied to the real OUT register (regardless of the "O" option setting), as in an "OUT=CS". Then the true IN register is read from the keyboard into A or C.

Select this method of specifying the IN register by pressing [K] when the cursor is in the first IN position. The display shows:

```
OUT:000 IN:KEYBD   I:E
```

When single-stepping (or macro-stepping) an "A=IN" or "C=IN" instruction, this option will cause a prompt, for certainly the [S] (or [M]) key will still be down. The next item explains an IN prompt.

7.5.4.4 IN PROMPT. You can have the emulator prompt for an IN register value when an "A=IN" or "C=IN" instruction is encountered. Select this option by pressing g[?] when the cursor is in the first IN position. The display will show:

```
     OUT:000 IN:?Prompt I:E
```

When you run the emulator ([RUN], [S] or [M]), this will cause an IN PROMPT break. The IN PROMPT register will be displayed, already in edit mode:

```
     OUT:001  <A=IN>:____ ?
 or, OUT:001  <C=IN>:____ ?
```

This display is unlike any other in the debugger, because it indicates the emulator has paused for input -- as shown by the question mark. This is not a time to edit other registers, including the OUT register. After entering a value for the A or C register, pressing [ENDLINE] will resume emulation.

The OUT value shown is the current value of the emulated OUT register. This would be important for you to determine what value to supply for the A or C register.

During an IN PROMPT, only four keys are active, besides the digit and cursor keys:

[ENDLINE]	Enters the desired IN register value. If not all four spaces have been filled, the debugger prompts again.
[ATTN]	Aborts the edit and returns to view mode, with OUT/IN the current register. You can view and edit other stacks, if desired. Resuming the emulator (with [RUN], [S] or [M]) will pick up at the "A=IN" or "C=IN" instruction, and reprompt.
f[OFF]	As [ATTN], plus it turns off the debugger.
f[DELETE]	Resets the IN register to blanks, remains in edit mode.

Filling in the digits for an IN PROMPT does not assign a value to the IN register; it merely provides a one-time value for this "A=IN" or "C=IN" instruction.

7.6 Window Stack

All fifteen registers are identical in their operation.

- [Z] to edit window address
- Menu editing
- f[EDIT] to edit memory contents
- Address increment/decrement
- Memory window keys
- [/] to edit window-type

Because of the limited display, the register number is not shown. To obtain the register number, press a window key (such as [.]), or go into edit mode with [Z]. For example,

```
A window register in the display:  00005.E100060F4812C136
[.] for momentary direct window:   W 7. E100060F4812C1361
[Z] to edit mode:                  W 7/00005=__   Type/D.
```

There are three types of windows:

1. Direct window -- shown with '.'
2. Indirect window -- shown with '<'
3. Relative window -- shown with '"'

To select the window type, enter edit mode with [Z]. Press [/] to move the cursor to the "Type/" position, on the "/" character. Now the up- and down-arrow keys are used to roll through the window-type menu:

```
Window register in display:        00005.E100060F4812C136
[Z] to edit mode:                  W 7/00005=__   Type/D.
[/] to edit type, cursor on '/':   W 7/00005=__   Type/D.
[↑] to roll window-type menu:      W 7/00005=__   Type/I<
[↑] to roll window-type menu:      W 7/00005=__   Type/R"
[ENDLINE] to enter:                00005"115071902AF41507
```

The relative window is indicated by the quote character, '"', and shows that address 00023 (=00005+0001E) contains the nibbles 115071902AF41507.

Even though a window register might be an indirect or relative window, f[EDIT] always works on the memory at the *address shown in the display*. Pressing f[EDIT] repeatedly will pull up indirect addresses for memory editing.

7.7 Break Point Stack

7.7.1 #BPs register

- [Z] to edit RTN Level indicator

The #BPs register shows how many break point addresses have been entered in the stack, from 0 to 15. The RTN Level indicator is editable, and provides a way to emulate the remainder of a subroutine, and to monitor RSTK usage.

The allowable selections for RTN Level are 0 through 7, "ALL" and "NONE". The default is "NONE". To edit the indicator, press [Z] to enter edit mode. For 0 through 7, press the appropriate digit key. For "ALL", press [A]; for "NONE", press [N].

If a RTN Level break is set for 0 through 7, the level indicator will track the level number as the RSTK is pushed and popped. Any action which pops that level off the stack (any type of "RTN", or a "C=RSTK" instruction) will cause a "RTN BK". If the level is pushed off the end of the stack (any type of "GOSUB", or a "RSTK=C" instruction), an "X RSTK" break will occur. This is useful to insure that routines do not use too many return levels.

If the RTN Level indicator is set for "ALL", any type of "RTN" will cause a "RTN BK". However, a "C=RSTK" will not cause a break, nor will any type of RSTK push cause an "X RSTK" break.

7.7.2 Break Point registers

All fifteen registers are identical in their operation.

- [Z] to edit Break Point address
- Menu editing
- f[EDIT] to edit memory contents
- Address increment/decrement
- Memory window keys
- [(] to edit Break Point Show field
- [)] to edit Break Point Action field

The address in a break point register is used to break the emulator if the PC executes an opcode at that address (whether the opcode begins at that address or spans it), or if a data operation (read/write with DAT0 or DAT1) occurs at that address (whether beginning at or spanning it).

Break point registers are sorted in increasing order as you add, edit or delete them. As you edit a break point, you may notice its register number change as it takes a new place in the stack. Ordering the break points greatly enhances the speed of the emulator.

Setting the "K" option will cause a beep when a break point is encountered, as well as any other break except "-ATTN-" or "S STEP".

The break point registers allow you to specify a Show register (a register to display) and an Action to take when the break point is encountered.

7.7.2.1 **Editing the Show field.** Press [(] when in edit mode. For example, to set the Show field, do the following:

```
Desired break point in display:     BP 1~3A41C  (__  )Halt
[Z] to enter edit mode:             BP 1~3A41C=__   (..)..
```

When editing a break point, the Show and Action fields, because of the limited display, both appear as ".." to indicate their presence.

```
[(] to edit Show, cursor on '(':    BP 1~3A41C  (__  )Halt
```

The Show and Action fields are restored. Now, while editing the Show register, you have a window on the debugger stacks. The arrow keys and the direct access keys are active to scroll through the stacks and bring up the name of any register.

```
[5] select D0-IN stack for Show:    BP 1~3A41C  (D0  )Halt
[↓] roll down to D1 register:       BP 1~3A41C  (D1  )Halt
[↓] roll down to ST register:       BP 1~3A41C  (ST  )Halt
[ENDLINE] to enter:                 BP 1~3A41C  (ST  )Halt
```

Now when running the emulator, if this break point is encountered, the ST register will be brought up to the display immediately.

During Show editing the User's stack is ignored. Pressing the direct access key [0] instead shows the null menu entry. (Any register in the User's stack can be found in its normal place in the debugger stacks.) A null Show entry indicates that, when encountering this break point, the debugger will display the top level of the User's stack by default. If the User's stack is empty, the PC register will be displayed.

During Show editing, the Home register is not accessible by its contents -- it is referred to by its formal name, "Home". Selecting "Home" as the Show register means that the debugger will display the Home register (the last displayed before running the emulator) at the break point.

A break point can be selected for the Show register. As break points are reordered or deleted, the Show register number will change accordingly. If a break point register selected for Show is deleted, the Show register reverts to the null entry.

** Note for ICC release: In the current version of the debugger, break points cannot be selected for the Show register.

Two registers perform special actions when selected for Show: the LIST STACK and XQT registers. Instead of *displaying* these registers, the debugger actually executes them. Thus, if you select LIST for the Show register, all selected stacks in LIST STACK will be displayed at the break point. If you select XQT, the opcode in the XQT register will be executed. This latter option can be useful for "programming" the debugger. If you find, say, that a "P=0" instruction is missing from your code, you can put the "20" opcode in the XQT register, set "XQT" and "Cont" for the BP Show and Action, respectively, and the missing instruction will be executed at the break point. If more than one instruction is missing, you can execute a "GOSBVL" to some unused area of RAM where you have filled in the opcodes.

** Note for ICC release: In the current version of the debugger, selecting LIST and XQT will display these registers only, not execute them.

If you want the PC register displayed along with the Show register, set the "P" option.

7.7.2.2 **Editing the Action field.** To set the Action field, press [)] when in edit mode. There are three Break Point Actions available:

1. Halt -- display Show register and stop emulator.
2. Cont -- display Show register and continue.
3. Jump -- display Show register and perform hard jump to PC.

For example,

```
Desired break point in display:     BP 1~3A41C  (ST  )Halt
[Z] to enter edit mode:             BP 1~3A41C=__   (..)..
```

When editing a break point, the Show and Action fields, because of the limited display, both appear as ".." to indicate their presence.

```
[)] to edit Action, cursor on ')': BP 1~3A41C  (ST  )Halt
```

The Show and Action fields are restored. Now, while editing the Action field, you have a window on the Action-type stack; the up- and down-arrows are used to roll through this stack and make a selection:

```
[↑] to roll Action-type menu:       BP 1~3A41C  (ST  )Jump
[↑] to roll Action-type menu:       BP 1~3A41C  (ST  )Cont
[ENDLINE] to enter:                 BP 1~3A41C  (ST  )Cont
```

Now when running the emulator, if this break point is encountered, the ST register will be brought up to the display immediately, and the

emulator will continue. This provides a means of tracing a register through an instruction at a particular address.

The "Jump" option returns to the HP-71 BASIC operating system, restoring all real CPU registers from their emulated counterparts. This would be useful when a critical timing loop needs to be run at real speed, for instance. For information regarding hard jumps to the PC and reentering the debugger, see Chapter 10.

7.8 Options Stack

7.8.1 OPTIONS Register

- [Z] to edit options

The nine options are described in the "OPTIONS register" section of Chapter 4. To set or clear options, press [Z] to enter edit mode. Now the up- and down-arrow keys are used to change the option letters to upper or lower case, respectively. The arrow keys operate on the letter at which the cursor is blinking. An upper case letter indicates the option is set, while lower case indicates it is cleared.

7.8.2 TRC Register

- [Z] to edit Trace count
- [(] to edit Trace Show field
- [)] to edit Trace Action field

This register allows you to specify how often to display a register. The five digit Trace count is a hex value, and is the number of emulated opcodes to execute between displays. A value of 00000 (default) means no trace is in effect (actually 16^12, essentially an infinite count).

The Trace Show and Trace Action fields operate identically to those for break points. See the above section, "Break Point registers", for details.

If you want the PC register displayed along with the Show register, set the "P" option.

7.8.3 BIAS Register

- [Z] to edit BIAS fields
- [+] and [-] keys for BIAS sign

The first 5 nibbles displayed in this register are the amount of bias (signed) to apply to an address. The second and third 5 nibble fields are the lower limit and upper limit (inclusive) in which to apply the bias. The bias is applied only if the "B" option is set in the

OPTIONS register, and only to the following registers:

- PC and PCcy registers (PC address)
- XC register
- L0 through L7
- D0 and D1
- W1 through W15
- BP1 through BP15
- AUTO DSASSMBL register

The bias is only applied at time of display, not within the register itself.

The bias field is signed; the first position accepts only "+" or "-" during editing. All other positions accept any hex digit.

The bias option can only be active if DBGLEX3 is in memory.

** Note for ICC release: The current version of the debugger does not have the bias option installed, although you can edit the BIAS register.

7.8.4 AUTO DSASSMBL Register

- [Z] to edit AUTO address
- Menu editing
- f[EDIT] to edit memory contents
- Address increment/decrement
- Memory window keys

The AUTO DSASSMBL register specifies where the debugger is to start displaying opcode mnemonics when f[AUTO] is pressed. Using that keystroke in view mode starts a continuous display of mnemonics, until the [ATTN] key is pressed. Opcodes are disassembled sequentially in memory, not by execution flow. The address in the AUTO DSASSMBL register is updated as each mnemonic is displayed, so it can be resumed after being interrupted with the [ATTN] key.

The debugger, of course, has no way of determining if nibbles in memory are opcodes, BASIC tokens or tables, or if the disassembly is being started at a correct opcode boundary.

The disassembler is available only if DBGLEX3 is in memory.

7.8.5 LIST STACK Register

- [Z] to edit stack numbers

The LIST STACK register specifies which stacks are to be listed, in their entirety, when f[LIST] is executed in view mode. The list will also be displayed when LIST is the Show register for a break point or

trace. (LIST and XQT are treated specially for show registers -- see the section above on break point registers.)

Stack numbers correspond to the direct access keys shown in Figure 2, "Titan Debugger Stack Structure". To select a stack for LIST, press [Z] to enter edit mode. Now the up- and down-arrow keys are used to select and deselect stack numbers, respectively. Each position has either a digit, from 0 to 9 (indicating the corresponding stack is selected for LIST), or a dot, "." (indicating the stack is deselected for LIST).

The User's stack is perhaps most useful with the LIST STACK register. You can choose a few most frequently viewed registers, place them in the User's stack and have them all listed when you press f[LIST].

7.8.6 XQT Register

- [Z] to edit XQT opcode

The XQT register provides a means to execute a single opcode in the emulator. To execute an opcode, press [Z] to enter edit mode. Type in the desired opcode; the debugger interprets each nibble to determine the appropriate length of your opcode, and will expand or contract the available nibbles to match it. When the opcode is entered press [ENDLINE], which does two things:

1. executes the opcode in the emulator, and
2. sets the contents of the XQT register.

Once the contents of the XQT register are set, you can reexecute an opcode merely by pressing [Z], then [ENDLINE]. If you view other registers and come back to the XQT register, it will display the last opcode which was entered and executed (or the default "00", if the debugger was just initialized).

If while editing the XQT register, you decide to abort the edit and avoid execution, press [ATTN]. This will return the XQT register to its previous contents, with no opcode execution.

The XQT register is also executed when it is selected as the Show register for Break Point or Trace. Both it and the LIST STACK register are treated specially when selected for Show.

7.8.6.1 Effect on PC. Using the XQT register is like *inserting* an instruction at the PC address -- the instruction is executed, but unless the opcode involves a jump or return, the PC is not updated. Here is the effect on the PC for the different classes of opcodes:

7.8.6.1.1 Non-PC related instructions. The vast majority of opcodes are in this class, such as

- Arithmetic, such as "B=C", "A=A+A", "GR2EX", "LCHEX", etc.
- Pointer and data pointer, such as "P=2", "D1=C", "A=DAT0", "D1=D1+5", etc.
- Control, such as "SETHEX", "OUT=C", "INTOFF", "C=RSTK", etc.

These instructions are executed, and the PC is not changed.

7.8.6.1.2 **Relative jump instructions.** These are

- GOC (with carry set); GONC (with carry clear)
- A test instruction, jumping when true
- GOTO and GOLONG

For this class of opcode, XQT causes the PC to be adjusted by the relative offset of the jump. Thus, if the PC reads "3A041", and you edit and enter in the XQT register:

```
XQT: 6A01              a "GOTO +10B relative"
```

the PC will be incremented by +10B to 3A14C.

For a GOC/GONC which does not branch, or for a test which fails, the PC is not changed.

7.8.6.1.3 **Relative subroutine instruction.** These are

- GOSUB
- GOSUBL

For this type of instruction, the PC is pushed into L0 of the RSTK as the return address. Then the PC is adjusted by the relative offset, as for a GOTO.

7.8.6.1.4 **GOVLNG and GOSBVL.** For a GOVLNG, the PC is simply set to the address of the jump. For a GOSBVL, the PC is pushed into L0 of the RSTK as the return address, and then the PC is set to the address of the jump.

7.8.6.1.5 **Return instructions.** All types of returns (RTN, RTNC, RTNCC, RTNYES, etc.) are handled by popping level L0 off the RSTK and setting the PC to this value.

For a RTNC, RTNNC or RTNYES which does not return, the PC is not changed.

7.8.6.2 **Similar to S STEP.** Any action which could be expected when running the emulator may happen with the XQT register. For example, executing a "RESET" instruction (80A opcode) will cause a beep and display the XC/AC register. The only exception is for an IN PROMPT, caused by executing an "A=IN" or "C=IN" (802 or 803) instruction, with the IN register set to "?Prompt". This will cause the XC/AC register

to be displayed, rather than the usual IN PROMPT.

The XQT register allows all instructions except an "LC(14)" or "LC(15)" (opcodes 3E and 3F), since the XQT takes only up to 16 digits. If an "LC(14)" or "LC(15)" is desired, you must edit the C register instead.

The AC code for XQT is "S STEP".

8. USING THE EMULATOR

Once you have mastered the debugger's stacks and registers, the emulator will prove to be quite easy to use. It emulates assembly language by manipulating RAM buffers as the real CPU manipulates its registers. The real power of the debugger is in its viewing and editing capabilities when the emulator is suspended.

The emulator runs at about 1/75th real speed. Routines which take about five seconds to execute in the BASIC environment will take more than six minutes to emulate, assuming no break points are encountered. One of the most noticeable effects will be to make the display building process visible. Those applications which use the display routines may find this a good way to study their display procedures.

8.1 Emulating at the PC

Make sure the PC is set to the correct address for emulation. Once you have set up the emulated registers as needed (either by editing them, or through the RECOVER sequences described later), the emulator can be used from view mode by three methods:

1. Single-step: [S]
2. Macro-step: [M]
3. Run: [RUN]

8.1.1 Return to Home register

In each method, when the emulator breaks, the debugger will display the Home register -- the register displayed before [S], [M] or [RUN] was pressed -- unless the break is for one of the following reasons, as indicated by AC code:

8.1.1.1 **BRK PT.** If the emulator is running, encountering a break point will cause it to display the Show register you specified.

8.1.1.2 **IN PROMPT.** Encountering an "A=IN" or "C=IN" instruction, with the IN register set to "?Prompt", will cause a break with AC code "IN PROMPT".

8.1.1.3 **SHUTDN.** If the "S" option is set and a "SHUTDN" instruction (807 opcode) is encountered, the emulator will break and display the XC/AC register, AC code "SHUTDN". You can run the emulator through this instruction with [S], [M] or [RUN].

8.1.1.4 **SHUTDN/PC=0.** If a "SHUTDN" instruction is encountered, and the emulated OUT register has bit 0 set to zero, the emulator will beep and break, with the XC/AC register displayed, AC code "SHUTDN/PC=0". This break action will take place irregardless of the "S" option setting.

In this case, you can run the emulator through the "SHUTDN" instruction with [S], [M] or [RUN]. But since the PC has been set to 00000, this will execute the cold start routine, clearing memory and inevitably causing a Memory Lost.

8.1.1.5 **UNCNFG.** If an "UNCNFG" instruction is encountered, and the addressed device contains either of the files DBGLEX1 or DBGLEX2, the emulator will beep and break, with the XC/AC register displayed, AC code "UNCNFG". You cannot step or run the emulator through this instruction.

8.1.1.6 **RESET.** If a "RESET" instruction is encountered, the emulator will beep and break, with the XC/AC register displayed, AC code "RESET". You cannot step or run the emulator through this instruction.

8.1.2 Single-stepping

Press [S] to single-step the emulator. The PC is incremented to the next instruction.

The AC code after a single-step is "S STEP", unless one of the above special breaks occurs. Setting the "K" option will help to identify them, since a beep will be sounded at any break other than [ATTN] or [S STEP].

8.1.3 Macro-stepping

Press [M] to single-step the emulator for all but GOSUBs (including GOSUBL and GOSBVL). For a GOSUB, the emulator will continue running until the subroutine returns, or until another reason for break occurs within the subroutine. Setting the "K" option will cause a beep at any break other than [ATTN] or [S STEP]; this will alert you to a break when macro-stepping a subroutine.

For macro-step, the AC code is "S STEP", unless a GOSUB is encountered and the emulator runs through the subroutine. In this case, when the return causes a break, the AC code will be "M STEP".

Encountering a GOSUB causes the emulator to begin running, and it will display the PC at the GOSUB address as follows:

```
     PC:3A14C:  *RUN*
```

If trace is in effect (TRC count is nonzero), the Trace Show register will be displayed periodically.

If you macro-step into a subroutine, and a break occurs within the subroutine for any reason (including "-ATTN-"), the break flag on the RSTK level is lost. If you want to restore the break at the same return address, you must set the RTN Level indicator in the #BPs register, or set a break point in a BP register.

8.1.4 Running

Press [RUN] to run the emulator. This causes the emulator to execute instructions continuously, until [ATTN] is pressed, or until another reason for breaking. Setting the "K" option will alert you with a beep when the emulator stops running, except for an [ATTN] break.

The debugger will display the PC at the starting address as follows:

```
     PC:3A14C:  *RUN*
```

If trace is in effect (TRC count is nonzero), the Trace Show register will be displayed periodically.

8.2 AC Hierarchy

The Action Completed code, shown in the XC/AC register, reflects the most significant reason for break. Since any one instruction may have several break actions associated with it, the AC code is chosen according to this hierarchy:

AC CODE	AC Hierarchy
BRK PT (*1)	A . . . . .
X RSTK	. B . . . .
RTN BK	. B . . . .
IN PROMPT	. B . . . .
UNCNFG	. B . . . .
RESET	. B . . . .
SHUTDN/PC=0	. B . . . .
SHUTDN (*2)	. . C . . .
S STEP	. . . D . .
M STEP	. . . D . .
BRK PT (*3)	. . . . E .
-ATTN-	. . . . . F

Notes: (*1) Break point at first nibble of instruction.
(*2) SHUTDN with "S" option set.
(*3) Break point at other than 1st nibble of instruction.

For example, if while emulating a "RTN" instruction, it had the following break actions associated with it:

1. A RTN Level indicator break;
2. A break point set on the first nibble;
3. A macro-step break flag ([M] had run through subroutine);
4. The [ATTN] key had been hit;

then the emulator would break and report "BRK PT" for the AC code.

Note that for AC codes at the same level of hierarchy, because each is

caused by a different type of instruction or action, only one of the codes can be in effect for any given instruction.

9. BACK AND FORTH FROM BASIC

** Note for ICC release: The keyboard routines for the debugger have not been installed. The debugger shares the HP-71's key buffer, removing keys from the buffer for action. This chapter describes the planned implementation of the debugger key routines. *This entire chapter does not apply to the current version of the debugger.* To return to BASIC, press f[Q]. To reenter the debugger, execute DEBUG.

The Titan Debugger uses its own key detection routines and key buffer. The main implication of having the debugger active is that keys are stored in its own key buffer, rather than the mainframe's. From its key buffer, the debugger removes and acts upon each key for direction.

Having its own key routines means the debugger takes over keyboard interrupts. This allows you to set up execution in the BASIC environment, then interrupt the operating system for processing to be returned into the debugger emulator. This process is described below.

9.1 Feeding the Key Buffers

There are several actions you can take to force keys to be put into the mainframe's key buffer, for the operating system (whether emulated or real-time) to process. Any time this condition is in effect (the debugger sending keys to the HP-71's key buffer), the ((*)) annunciator is lit in the display. (If this annunciator conflicts with your application, it can be changed. See Chapter 12 for details.)

One time when you want to feed keys to the HP-71 operating system is when you drop into BASIC from the debugger. This is the topic of the next section.

The other time you may want to send keys to the HP-71 operating system is when you run the emulator, and want emulated routines to detect keys. When in the debugger proper, pressing g[ON] causes the ((*)) annunciator to toggle. When it is *on*, keys are sent to the HP-71 key buffer; when it is *off*, keys are sent to the debugger's key buffer. This can be done when in view mode or when the emulator is running, but not when in edit mode. (If you have redefined the g[ON] keystroke, you will not be able to force it into the HP-71 key buffer with this method. However, you can still use the key definition when you run in the BASIC environment.)

9.2 Dropping into BASIC

When in the debugger proper, you can drop into the HP-71's BASIC operating system by four methods. Two of the methods leave the debugger's keyboard active, so you can resume emulating. The other two methods deactivate the debugger, returning the computer to the control of the operating system.

The HP-71 operating system does not, of course, detect the debugger break points. Only the emulator can do this, so you must reenter the debugger for this capability.

9.2.1 Four Methods

9.2.1.1 [Q][Q] to main loop. Pressing [Q][Q] (the [Q] key twice within two seconds) drops processing into the BASIC main loop, essentially performing an "INIT: 1". From here, you can enter BASIC commands, run programs, change HPIL devices, etc. The ((*)) annunciator remains on to signal that any keys you press will be put into the HP-71's key buffer for the mainframe to process.

You can remain in this state indefinitely, if desired, although any change to the DBGLEX files may cause a Memory Lost. For this reason, you should not drop into BASIC if your DBGLEX files are in MAIN RAM.

9.2.1.2 [J][J] for hard jump. Pressing [J][J] (the [J] key twice within two seconds) performs a hard jump to the PC address. All restorable registers in the CPU are restored from their emulated counterparts, and execution picks up at the PC address. (For a list of all restorable registers, see Chapter 10.)

[J][J] can only be executed when the PC or PCcy register is in the display.

As with [Q][Q], the ((*)) annunciator is lit to indicate that keys are being processed by the mainframe, although the debugger is active. You can remain in this state indefinitely, if desired, with the risks mentioned above.

9.2.1.3 RECOVER methods. The other two methods of returning to BASIC are described below, under "RECOVER: 2" and "RECOVER: 3".

9.3 The RECOVER sequence

The familiar INIT sequence in the HP-71 mainframe is replaced by the RECOVER prompt. When the debugger is active, pressing [ON][/] (both keys simultaneously) causes the prompt "RECOVER: 1". You can select four levels of recovery by pressing one of the digit keys [0] through [3], followed by [ENDLINE].

You must be very careful when operating in the BASIC environment with the debugger active, that files DBGLEX1 and DBGLEX2 are not moved or disturbed. If they are, the interrupt address -- not to mention the buffer area and utility routines -- will be invalid, and, at best, the "RECOVER: 0" sequence may cause a Memory Lost. At worst, any keyboard interrupt will tank the computer.

9.3.1 RECOVER: 0

When in the debugger proper, "RECOVER: 0" is not allowed. The computer will beep and reprompt for the RECOVER level.

When in BASIC with the debugger active -- you get there by pressing [Q][Q] or [J][J] from the debugger proper -- the "RECOVER: 0" sequence interrupts the operating system and copies all recoverable CPU registers into their emulated counterparts. At this point, the debugger displays the PC register and waits for manual direction. (For a list of recoverable registers, see Chapter 10.)

The "RECOVER: 0" sequence is a powerful method of directing the debugger. With precise timing, you can interrupt the real-time execution of a target routine and set up all emulator registers for debugging. For example, say you have written a new BASIC keyword, LEAP and need to study its execution. After assembling the LEX file and putting it in memory, perform the following steps:

1. In BASIC, get the address of your LEX file, say "LEAPLEX":
   ```
   ADDR$("LEAPLEX")
   ```
   You must use this address to compute the location of the execution routine you wish to debug. Say the ADDR$ function returns "3A041", and you know that the target routine starts at 10B nibbles beyond the file header. Then the desired debug address is 3A041+10B=3A14C. You need to remember this address.
2. Execute DEBUG to enter the debugger. Select the Break Point stack and enter the address 3A14C in the first available break point register. Now press [Q][Q] to drop into the HP-71 main loop.
3. Type LEAP to execute the keyword, but don't press [ENDLINE] yet.
4. Now press [ENDLINE], and quickly press [ON][/] to interrupt the operating system and get the RECOVER prompt. Press "0" to select "RECOVER: 0", then [ENDLINE].
5. The debugger is now ready to use, with all registers, including the PC, at the point of the [ON][/] interrupt. Press [RUN] to run the emulator, and soon the break point at 3A14C will be encountered. (If the interval between [ENDLINE] and [ON][/] was too long, you might have already passed the 3A14C address. You should examine the PC before you press [RUN] to make sure you haven't passed the break point. You will soon get the feel of how long an interval is needed to get close to the target address without passing it.)

9.3.2 RECOVER: 1

When in the debugger proper, the "RECOVER: 1" sequence returns you to a stable point, displaying the PC register.

When you are in BASIC with the debugger active -- after [Q][Q] or [J][J] -- the "RECOVER: 1" sequence performs an "INIT: 1", returning you to the stable point of the BASIC main loop.

In either case, a "RECOVER: 1" is not guaranteed to be a safe action. If memory is being altered or moved at the time of the [ON][/] interrupt, the system may not be able to recover from the disruption, and cause a Memory Lost. "RECOVER: 1" (and, for that matter, "INIT: 1") should not be performed lightly.

9.3.3 RECOVER: 2

This deactivates the debugger, returning complete control to the BASIC operating system, including reestablishing keyboard control and the INIT prompt. The effect on the BASIC system is identical to performing an "INIT: 1". To reenter the debugger, you must execute the DEBUG keyword.

When you are done with a debugging session, you should routinely perform a "RECOVER: 2" to deactivate the debugger. This will prevent problems in the case (though remote, unless the files are in MAIN) that the DBGLEX files are moved or changed.

9.3.4 RECOVER 3

This performs an "INIT: 3" -- a Memory Lost, deactivating the debugger in the process.

9.4 Reentering the Debugger

If you are executing in the BASIC environment -- including not having yet invoked the debugger -- you can always execute the BASIC statement DEBUG. This will reestablish all emulated CPU registers to the contents held at the time you last left the debugger. (Note that RAM contents might have been changed in BASIC.)

If you have dropped into BASIC with [Q][Q] or [J][J], the "RECOVER: 0" sequence will be most useful for reentering the debugger. As described above, this sets up the debugger for emulation at the point of the [ON][/] interruption.

There is a third way to reenter the debugger -- through an assembly language jump to the debugger reenter address. This it described in the next chapter.

10. HARD JUMP AND ASSEMBLY LANGUAGE REENTER

** Note for ICC release: The features described in this chapter have not been installed. The [J][J] keystroke is ignored, and the "REENTR" entry point has not been implemented. *This entire chapter does not apply to the current version of the debugger.*

10.1 Hard Jump to PC

Pressing [J][J] when the PC is in the display, or executing the "Jump" Action for Break Point or Trace will cause the debugger to perform a hard jump to the PC. This causes real-time execution of the code at the PC address. You can reenter the debugger with the "RECOVER: 0" sequence, with the DEBUG keyword from BASIC, or with an assembly language reenter (see below).

Before jumping to the PC address, the following restorable registers are copied from the emulator to their CPU counterparts:

```
     Registers restored:
          A   B   C   D
          R0  R1  R2  R3  R4
          First 7 levels of RSTK  (L0 - L6)
          ST
          HS bits: SB and XM only
          OUT
          Interrupt
          D0  D1
          Carry, Pointer, Mode
          PC

     Registers not restored:
          Level 8 of RSTK  (L7)   (set to zeroes)
          HS bits: SR and MP      (cleared)
 .fi
```

10.2 Assembly Language Reenter

Your assembly language routine can jump directly back to the debugger, without using the DEBUG keyword or the manual "RECOVER: 0" sequence. The entry point "REENTR" in the debugger is for this purpose.

To use the "REENTR" feature, you must have a strategically placed "GOSBVL REENTR" in your RAM-based assembly language application. The actual address of REENTR has to be computed by you, according to the address of DBGLEX1 in memory. You can specify the address in your assembly source file, or fill it in afterwards with a POKE or by memory editing in the debugger.

The "REENTR" label occurs XXX nibbles beyond the DBGLEX1 file header address. Compute its address by executing ADDR$("DBGLEX1"), and add XXX to the result (in hex). Where your "GOSBVL REENTR" occurs in your assembly file, fill in the address. For example, say you want to debug a keyword LEAP at the following routine starting at "LABEL3". Your assembly language code should look like this (the four instructions marked "*" are for example only):

```
          A=C    A        *   Opcodes
          CD1EX           *     as desired...
          GOSBVL #00000       Fill in address of REENTR !
LABEL3    B=B+C  A        *   More opcodes
          D0=A            *     as desired...
           ...
            ...
             ...
```

Follow this procedure, for the example:

1. Assemble and load the LEX file in memory.

2. Compute the address of "REENTR" as described above, and remember the address.

3. Now enter the debugger by typing DEBUG -- this sets up some addresses and pointers required for later reentry. Set a window register to the address of the "GOSBVL" in your file. Using f[EDIT] to edit memory, fill in the address at that opcode (remember to reverse the 5 nibbles).

4. Then press [Q][Q] to drop into BASIC. Now type the keyword LEAP, and press [ENDLINE]. The debugger will take over at the "GOSBVL REENTR", with the PC displayed (it is the address of "LABEL3"), waiting for direction.

When reentering the debugger, the emulator's CPU registers are recovered from the real CPU registers. The recoverable registers, which also apply to the "RECOVER: 0" sequence, are:

Registers recovered:
- A B C D
- R0 R1 R2 R3 R4
- First 7 levels of RSTK (L0 - L6) (see note below)
- ST
- HS bits: all (MP, SR, SB, XM)
- D0 D1
- Carry, Pointer, Mode
- PC (see note below)

Registers not recovered:
- Level 8 of RSTK (L7) (set to zeroes)

OUT register	(retains last debugger value)
Interrupt register	(retains last debugger value)

The following registers are untouched; they retain their values from when the debugger was last exited:

User's stack
Window stack
Break Point stack
Options stack

The RSTK levels recovered are actually levels L0 through L6 *at the moment before* [*ON*][/] *interrupt* (in the case of "RECOVER: 0"), or *at the moment before* "*GOSBVL REENTR*" (in the case of assembly language reenter). When the debugger is reentered with either of these two methods, the PC is set to the address of the return (from interrupt, of from REENTR), and level L7 is lost. This is why, in the example above, a "GOSBVL REENTR" is used to set the emulator's PC to LABEL3.

The AC code after reenter is "BRK PT".

11. DEBUGGING TECHNIQUES

11.1 Avoiding Cold Start

** Note for ICC release: The debugger's key routines have not been installed, so the cautions regarding VECTOR do not apply for the ICC release.

There are some special considerations when debugging the HP-71 mainframe. You have to be careful that you don't emulate through the cold start code (address 00000), since these routines clear all of memory. Many assembly language bugs show up as a Memory Lost, so you will probably find the debugger jumping here occasionally.

Since the debugger uses the interrupt vector, the VECTOR location in main RAM cannot be cleared! Doing so would send the debugger into oblivion, emulating without any way to detect keys. If this does happen, "INIT: 1" may bring back the computer, but "INIT: 3" might be the only way to recover. In any case, if the RAM clearing has already taken place, it may be to late to avoid a Memory Lost.

You may want to set a break point at address 00000 to prevent emulation of the cold start routine. Breaking at this address will prevent any RAM from being cleared, and you can almost always reset the debugger without causing a Memory Lost.

The debugger will provide you protection from the clearing of VECTOR if you execute the INITIALIZE command in the g[CMDS] menu. After this action resets all of debugger memory, it sets an automatic break point at address 00209, the location where VECTOR is cleared. If the debugger ever runs through the cold start section, it will halt at break point 00209 with D1=2F43C. You should not continue emulation; this has prevented the debugger from losing control of the keyboard, but you may need to reset all debugger registers and RAM pointers if you want to avoid a Memory Lost.

11.2 System Timers

Each of the three display chips contains a six-nibble countdown timer. When working with these system timers, special handling may be required during emulation. The CLKSPD routine (address 0E9F1) is a trap for the debugger. This routine computes the CPU speed in reference to a timer, waiting for a specific timer value to exit a counting loop. Since the debugger runs so slowly in relationship to the HP-71 operating system, it can run indefinitely in this counting loop as the timer speeds along. There are actually three locations in the CLKSPD routine where the timer is read and compared, so if you want to run the debugger through this routine, you must adjust the A and C registers for each comparison. Use the HP-71 IDS, Volume III, as a guide.

Other routines which read timers (such as WRITMR, address 15392) will not hang up the debugger indefinitely, since they do not enter a loop. However, the timer value read by the debugger will reflect the fact that it is running relatively slowly; if you are concerned about critical time intervals, you should halt the debugger at locations like this and edit a specific value into the A or C register.

11.3 Configuration

** Note for ICC release: The "C" option has not been installed. You cannot jump through or otherwise execute the configuration code.

Because the debugger resides in soft-configured RAM, it cannot allow emulation of the configuration routines. The following BASIC statements or actions cause the computer to reconfigure memory:

1. FREE PORT and CLAIM PORT.
2. OFF, BYE or the f[OFF] keystroke, then wakeup.
3. Copying a LEX file into memory, or purging a LEX file.
4. Reconfiguration forced by an assembly language application.
5. Cold start.

If you try to emulate any of the above, the debugger will encounter the "RESET" instruction at address 10233 in the configuration routine. If you have set option "C", the debugger will "jump through" the configuration code (hard jump) and recover with the configuration poll. If option "C" is not set, the emulator will break, beep and display the AC code "RESET".

The debugger may not be able to restore properly if the configuration of the computer undergoes a major change -- if too many ROMs are plugged in, for example. However, if the configuration of the computer underwent a major change, the debugger may not be able to restore its pointers at reentry, and cause a Memory Lost.

If your target assembly language tries to execute an "UNCNFG" instruction which would affect a DBGLEX file, or a "RESET" instruction at an address other than 10233, you have two options:

1. fill in the debugger registers as if it had emulated through the routine, or
2. perform a hard jump, [J][J], at that address and try to pick up emulation quickly with the "RECOVER: 0" sequence.

Most often, however, trying to emulate configuration routines signals that something in the emulation has gone wrong. You may be best off restarting the entire debugging session.

11.4 Conflicts with VECTOR

If your assembly language application uses the VECTOR interrupt RAM location, you will not be able to use the debugger to emulate it. You can still emulate those parts of the application which do not depend on the VECTOR. If you alter your code to not use the VECTOR location, you can use the debugger to simulate an interrupt and jump into the interrupt handling code. Only when you are done using the debugger would you reinstall the VECTOR interception.

11.5 Using a display device

** Note for ICC release: The ability to deactivate the display device during debugging, and the ability to use a second HPIL loop dedicated to the debugger have not been implemented.

11.6 Using a keyboard device

** Note for ICC release: The ability to use a keyboard device during debugging has not been implemented.

12. MODIFYING THE DEBUGGER

12.1 Code modification

Listings of the debugger will be made available through the User's Library. There are empty spaces built into the debugger modules which allow you to install features as you desire.

As new CPUs are developed, they may have additional opcode capabilities. If your HP-71 has such a CPU installed, you can add the necessary code to emulate the new opcodes, if your assembly language is specific to your computer. Even more remote is the possibiliy of adapting the debugger to another Saturn-based system. You would need to rewrite the display and keyboard routines, and several HP-71 mainframe utilities.

12.2 Annunciator Selection

** Note for ICC release: The annunciators are not used in the ICC version. The ability to select annunciators is not yet installed.

A. WORKING WITH THE ICC RELEASE

This chapter lists the missing features in, and the best methods for using, the ICC release of the debugger, dated February 3, 1986.

The VER$ response for the ICC release is "DEBUG:icc". If you are running this version of the debugger, you will find missing several features described in this manual.

A.1 Missing Features

- The third debugger module, DBGLEX3. This means the following features are not installed.
  - Bias. The bias cannot be applied, although you can edit the BIAS register.
  - Disassembler. The f[AUTO], [,], g[>] and g[#] keystrokes show "Not Implemented".
- The DEBUG * form of the DEBUG keyword.
- The debugger key routines. This means that the following features are not installed.
  - The RECOVER sequence; [ON][/] always prompts for INIT, and is handled by the mainframe. You can reenter the debugger only with the DEBUG keyword.
  - [Q][Q] and [J][J]. The only way to exit the debugger is with the f[Q] keystroke. This means that the debugger is not left "active" when you drop down to BASIC. You don't have to worry about the DBGLEX files moving in memory.
  - The debugger key buffer. Both the debugger and the HP-71 operating system share the same key buffer. While in view mode or edit mode, the debugger interprets keys; while the emulator is running, the HP-71 operating system interprets keys (in emulation), if you run it through the main loop.
  - The g[ON] toggle. Keys are always put into the HP-71's key buffer.
- The debugger's display routines. The debugger uses the mainframe display routines, sharing the display with the operating system. This means the following features are missing:
  - f[DISP] to display the emulated display buffer. It will redisplay the current debugger register.
  - Ability to emulate the mainframe display routines. You can "run" the emulator through them, but if you break the emulator, single-step or display the trace register, you will overwrite the emulated display buffer.

- The g[CMDS] menu. This means the following features are missing:
  - The KEYBOARD IS capability.
  - The ability to turn off the external display device.
  - The ability to use a second HPIL loop dedicated to the debugger display.
- Warnings (such as "WRN:DBGLEX1 in MAIN") use the fixed names DBGLEX2, DBGLEX2, and DBGLEX3, instead of actual file names.
- The annunciators, AC to signal being in the User's stack, and ((*)) to indicate that the debugger key routines are active.
- The OPTIONS register. No options are installed, but you can edit this register.
- The ability to put break points in the User's stack.
- The ability to select break points for BP Show or Trace Show.
- Selecting LIST or XQT for the BP Show or Trace Show register does not execute them. It merely displays them.
- Only the "Halt" setting for BP Action is installed. You can set the BP Action for "Cont" or "Jump", but they will only halt.
- Only the "Cont" setting for Trace Action is installed. You can set the BP Action for "Halt" or "Jump", but they will only continue.
- The REENTR entry point for assembly language reentry.
- The ability to select "KEY#", "KEYBD" or "?Prompt" for the IN register.

A.2 How to Run it

Since the [Q][Q] and "RECOVER: 0" features are not installed, picking up emulation in the middle of an assembly language routine is difficult. Not only would you have to enter the debugger and set all registers manuall, but you would have to set critical RAM locations, too.

A better way, although suffering from the debugger's speed factor, is to enter the debugger and emulate the operating system's main loop. When you run the emulator, you can enter keys or run a program to exercise your assembly language routines. For example, say you have a BINary file, "BINTEST", to test with the debugger. Perform the following steps:

1. After BINTEST is loaded in memory, execute the DEBUG keyword. This will bring up the PC register in the display (or the current register when you last exited).

2. Edit the PC, setting the address to 002FD, the entry point "MAINLP". No other registers need be changed, although you will probably want to set a break point in the BINTEST file.

3. Now press [RUN] to run the emulator. Type in "RUN BINTEST" [ENDLINE]; the keys will be processed by the BASIC main loop, and the BINTEST file will start running. The emulator will soon halt any break point you specified in the BINTEST file.

An easier way might be to enter a program file, such as

```
10 RUN BINTEST
```

Then, after you run the emulator at "MAINLP", just press the [RUN] key again to emulate your program. This involves fewer keystrokes, and will execute faster.

Running the emulator from "MAINLP", address 002FD, will allow you to emulate practically anything. You can try it with BEEP, DISP VER$, SIN(5), etc. But remember, since the debugger uses the mainframe key routines, that pressing [ATTN] to break the emulator will also cause the rest of the keys to be cleared from the key buffer.

A.3 Working without the Options

Without the "S" option ("stop at SHUTDN"), you have to set a break point at a SHUTDN instruction, if you want it to break. The main SHUTDN instruction for the operating system is at address 006EA, during light sleep (waiting for keys with the cursor blinking). But you don't need to break at a SHUTDN unless you have a specific reason for doing so.

To make the "K" option work ("beep at all breaks, except -ATTN- and S STEP), you can poke (or use memory edit in the debugger) into DBGLEX1. At 0184E(hex) nibbles Find ADDR$("DBGLEX1") and add 0184E(hex) to it. (The resulting address should be xx856, absolute.) The opcode at that address is 5D0. Change it to 4D0, and all emulator breaks (other than the two above) will cause a beep.

To read keyboard