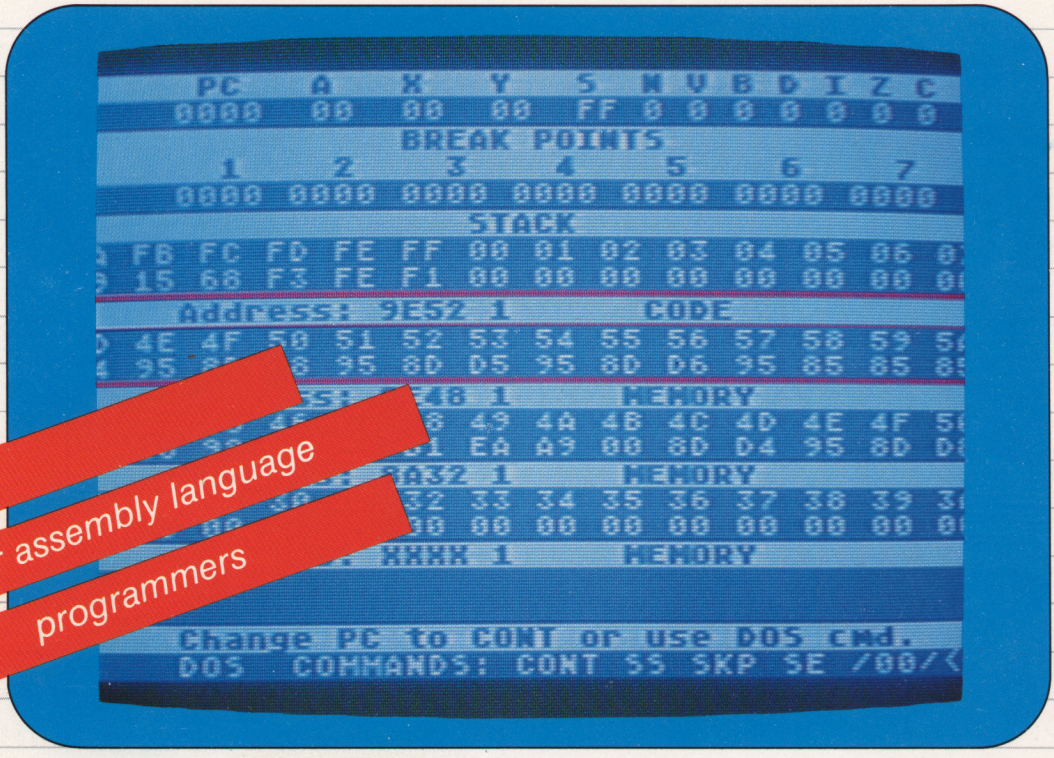
HEX-A-BUG

SYSTEMS/TELECOMMUNICATIONS

A hexadecimal-based, screen-oriented debugging tool for the ATARI Computer



For assembly language programmers

CONSUMER-WRITTEN PROGRAMS FOR

ATARI®

HOME COMPUTERS

APX

ATARI Program Exchange

Printed in U.S.A.

HEX-A-BUG

by

David Kano

Distributed By

The ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

To request an APX Product Catalog, write to the address above, or call toll-free:

800/538-1862 (outside California)
800/672-1850 (within California)

Or call our Sales number, 408/727-5603

Trademarks of Atari

Contents

Introduction

Overview

When speed, memory usage, and maximum use of the ATARI Computer's hardware capabilities are important in a program, using assembly language is worth the extra effort. But the speed and low-level, non-interpretive nature of assembly language programs makes finding bugs in these programs more difficult than debugging programs in higher level languages like BASIC. *Hex-A-Bug* can make assembly language programming on the ATARI Computer an attractive alternative to programming in high level languages.

You load this easy-to-learn and easy-to-use tool into memory along with your assembly language program, and you then use breakpoints to switch control from your program to *Hex-A-Bug*. Being able to determine the intermediate results of your program by studying memory locations and register values can be invaluable for locating errors. Your program's screen display remains intact and you can easily toggle between it and the *Hex-A-Bug* display.

This screen-oriented program uses very few commands. The main screen area consists of "strips" across the screen, each strip being one functional area. You move a flashing cursor from one functional area to another. In this way, you can directly change the contents of any register, breakpoint, address of a memory strip, or memory location. Horiztonal fine scrolling forwards and backwards from any location gives you quick and easy access to all information. In addition, you use simple commands to do such things as go to DOS, single step through your program, search for a string of values, and continue execution of your program. Users of the ATARI Program-Text Editor (which is part of the ATARI MACRO Assembler and is also available separately through APX) will find *Hex-A-Bug* particularly easy to use, since cursor movement, function keys, and main, error, and command windows are used for similar functions by these programs. The interaction is much like editing a piece of text, except you are directing *Hex-A-Bug* to do something specific with each edit. With *Hex-A-Bug*, you concentrate on your bugs, not on your debugging tool.

Required accessories

- 48K RAM
- ATARI 810 Disk Drive

Optional accessories

- ATARI MACRO Assembler or ATARI Assembler Editor Cartridge

Contacting the author

Users wishing to contact the author about *Hex-A-Bug* may write to him at:

RFD 4 Lincoln Rd.
Lincoln, MA 01773

Dedication

Hex-A-Bug is dedicated to my parents, Cyrus and Dorothy Kano, who bought me my first home computer.

Loading *Hex-A-Bug* into computer memory

1. Remove any program cartridge from the cartridge slot of your computer.
2. Have your computer turned OFF.
3. Turn on your disk drive.
4. When the BUSY light goes out, open the disk drive door and insert the *Hex-A-Bug* diskette with the label in the lower right-hand corner nearest to you. Close the door.
5. Turn on your computer and your TV set. The program will load into computer memory and start automatically.

Using this manual

This manual is intended for assembly language programmers with some knowledge of the internals of the 6502 microprocessor. Load *Hex-A-Bug* and experiment with its features as you read through the manual. Leave for last the section on theory of operation, but don't omit reading this section. Beginning assembly language programmers will find the glossary of terms at the end of this manual helpful.

Introduction

Before you're ready to use *Hex-A-Bug*, you need to write your assembly language program or subroutine. The first instruction in your program should be a BRK instruction (a $00 byte). Whenever your program executes a BRK instruction, *Hex-A-Bug* will be reentered. You need only program in this one BRK instruction, since you can use *Hex-A-Bug* to set up to seven additional breakpoints.

Next, you load *Hex-A-Bug* into memory as described earlier. After loading *Hex-A-Bug*, use the DOS command to go to the Disk Operating System's menu. Then use the L (Load binary file) command to load your program. After your program loads into memory and runs, the BRK instruction will execute, and *Hex-A-Bug* will be reentered. You can set any additional breakpoints you need and use the CONT, SKP, or SS commands to restart your program.

If you're testing a subroutine, you may need to change the program counter (PC) to the start of the subroutine, and set up any data that it uses before continuing. In this way, you can test a subroutine that will be called from BASIC. If you're working on a large program, you can assemble new subroutines separately, and debug them before including them in your main program. This modular programming technique saves time in two ways: (1) reassembly for each test/fix cycle takes less time, and (2) proven subroutines may be used in many programs.

If your new program stops running correctly, you may need to try again, setting more breakpoints in the suspected routines until you find the bugs. Simple errors can often be "patched" right from *Hex-A-Bug* so that you can continue the debugging process. The more bugs you find in each session, the fewer test/fix cycles you'll need to finish your program.

Hex-A-Bug's display screen

Hex-A-Bug is "screen oriented". The main display consists of "strips" across the screen. Each strip is one functional area, containing two or more lines. Below this area are the message and command lines. Your current location on the display screen is indicated by a flashing marker (cursor).

The display screen looks like this:

STRIP NAME	SAMPLE SCREEN DISPLAY	NOTES
Register strip	PC A X Y S N V B D I Z C	(INVERSE)
	3304 0F 10 00 FB 0 0 0 0 0 0 0	Fields
Breakpoint strip	BREAKPOINTS	(INVERSE)
	1 2 3 4 5 6 7	(INVERSE)
	3304 0000 0000 0000 0000 0000 0000	Fields
Stack strip	STACK	(INVERSE)
	F7 F8 F9 FA FB FC FD FE FF 00 01 02 03	
	AA AB EE FF 1F 5F 23 31 1B C1 11 12 22	Fields
Code strip	Address: 3304 1 CODE	(INVERSE)
	00 01 02 03 04 05 06 07 08 09 0A 0B 0C	
	11 22 33 44 55 66 77 88 99 01 02 03 04	Fields
Memory strip	Address: XXXX 1 MEMORY	(INVERSE)
Memory strip	Address: XXXX 1 MEMORY	(INVERSE)
	Hex-A-Bug	
	Copyright 1982 David Kano	
Memory strip	Address: XXXX 1 MEMORY	Header line
		Label line
		Data line
Message line	Change PC to CONT or use DOS cmd.	(INVERSE)
Command line	DOS COMMANDS: CONT SS SKP SE /00/	

Figure 1 *Hex-A-Bug*'s Display

Each line of the screen that will accept input has two or more "fields," or display/input areas containing values. For example, the register strip has a field for each register and the breakpoint strip has a field for each breakpoint. Above each field is a label indicating the kind of field, or the memory address that field is displaying.

Current location in the display area. On color televisions or monitors, one of the strips is enclosed in a red border. On black and white screens, this border is wider than the lines above and below the other strips. This boxed area is the "current strip". Your cursor is in this strip when it's in the main screen area. Press the OPTION key to toggle between the command line (which is where the cursor displays when you first enter *Hex-A-Bug*) and the strips. Press the control-arrow keys, explained below, to move from one strip to another.

Moving around the display

You can move your cursor quickly from place to place on the screen with a few simple keystrokes. *Hex-A-Bug*'s cursor moves only to locations (fields) accepting input. After moving to a location, the next character you type replaces the one currently under your cursor. If you press a key that has no meaning, an error message displays in the message line and your cursor doesn't move.

Hex-A-Bug uses the following keys to move around the strips. To use "control" keys, which are indicated by "CTRL/", hold the CTRL key while pressing the other key indicated. The basic combinations are as follows:

CTRL/up arrow → moves up one input line
CTRL/down arrow → moves down one input line
CTRL/right arrow → moves right within a line
CTRL/left arrow → moves left within a line
TAB → moves to the next field in the line
DELETE/BACK S → move to the previous character in the strip (and set value to zero, except in a data line)

Press and hold any of these keys for repeated movement. The cursor wraps when you press a CTRL/arrow key with the cursor on the last line or character in a particular direction (except that the right and left arrow keys, TAB, and DELETE/BACK S keys scroll in data lines instead of wrapping). Once you master the use of these cursor control keys, you need to learn only a few commands to start using *Hex-A-Bug*.

Input areas

The stack, code, and memory strips contain only **one** input field in their data lines. To change a value in the data lines, move within the strip until the value you want to modify is under the cursor.

Memory strips

Use the memory strips to view and modify memory. The display has three general-purpose memory strips. The stack and code strips are special-purpose memory strips. The differences between them are described later.

Each memory strip is made up of three lines. Figure 2 below shows a sample memory strip.

```
(1)     Address:   4004 1          MEMORY
(2)     00 01 02 03 04 05 06 07 08 09 0A 0B 0C
(3)     00 01 34 AF AA BB CC DD EE FF 11 22 33
```

Figure 2 Sample Memory Strip

Line (1), the header line, has two fields. Upon loading *Hex-A-Bug*, the first field contains "XXXX". You replace these X's with an address in hexadecimal form for the strip, such as 4004. The second field is the "direction switch", indicating the direction in which the cursor moves. The switch is set to 1 in the sample. You'll learn about its use later.

Line (2), the label line, contains as a field label the two least significant hexadecimal characters of the address. (The two most significant digits are in the address field of line (1).) Line (3), the data line, displays the data in the memory locations labeled in line 2. The data field directly beneath the address in the header displays the value in that address; for example, the label field directly beneath the address 4004 is 04 and the value is AA. The other fields display the values in the next four lower and eight higher addresses.

When the *Hex-A-Bug* screen first appears, the label and data lines in the second memory strip display the name of the program, author, and the copyright. This information disappears the first time you use each of this strip.

Displaying values in hexadecimal or ASCII form

The address header and field labels always display in hexadecimal form. However, you can view and modify the memory values in either hexadecimal or in ASCII form. Press CTRL/C to switch from hex to ASCII and vice versa in the current strip.

Viewing memory

To view an area of memory, enter its address into the header of one of the memory strips. When you move your cursor out of the header, the strip displays the contents of that address.

Scrolling to other memory locations

By horizontally scrolling the strip, you can view higher or lower memory locations. To start scrolling, move your cursor into the data line and press and hold the CTRL/right arrow or CTRL/left arrow keys. If you hold down the key for about three seconds, the scrolling speed doubles. When you stop scrolling, your cursor is positioned in the same location on the screen, but a new memory location displays in that position, as indicated in the label line.

Each memory strip is a window into the whole address space of your ATARI Computer. The section on commands you use in the command line describes how to step through tables and search for data strings using memory strips.

Modifying memory

Once you use a strip to view an area of memory, you can modify the contents of that memory by moving the value you want to change under the cursor, and typing in the new value. *Hex-A-Bug* checks to be sure you're not trying to modify ROM (Read Only Memory). If you are, it displays the message SORRY, ADDRESS IS IN ROM and waits for your next action. It won't let you modify ROM values.

Direction switch

We normally type from left to right. When modifying memory in the data line, this would change memory from low to high addresses. This order isn't always convenient, however. Push down stacks, for example, fill from high to low memory. All 16-bit addresses used by the 6502 microprocessor are stored low order byte first and then high. So that you can change memory in either direction easily, *Hex-A-Bug* has a direction switch. This is the function of the second field in each header line. If you type a "0" in this field, your cursor moves *left* one field after you change a data value. If you type any other hexadecimal value (i.e., 1-F) in the direction field, your cursor moves *right* one location when you change a data value. The direction value also affects the TAB key, which moves your cursor to the next logical field, not always the one to the right.

The easiest way to get used to this feature is to try it. Have a memory strip point into an unused memory area and modify some locations. Change directions and see how the TAB and the DELETE/BACK S keys act each way. Change to ASCII mode (by pressing CTRL/C) and see how easy it is to modify and read character data with *Hex-A-Bug*'s smooth scrolling feature. This feature makes *Hex-A-Bug* a powerful tool for changing and building tables, modifying code, and experimenting with different data when testing your programs or subroutines.

Because of the ease with which you can modify memory, be careful when moving your cursor from strip to strip.

Stack strip

The stack strip is a special-purpose memory strip used to view and modify the contents of your program's stack. When your program hits a breakpoint, *Hex-A-Bug* moves the data in the stack to a page of memory in *Hex-A-Bug*. The stack strip is for viewing and modifying this saved stack data. On reentry to your program, the saved data is moved back to the real stack. Saving the stack has a few advantages.

The size of the stack in the 6502 microprocessor is an inherent disadvantage. Other microprocessors have stacks that can use the entire memory of the computer. The 6502's S register is only 8 bits wide, so the stack is only 256 bytes (one page) long. If your program causes a stack overflow (by pushing more than 256 bytes onto the stack) it will overwrite the data that was first pushed to the stack. After the stack is saved, *Hex-A-Bug* resets the S register to the start ($FF) for its own use. Thus, even if your program uses the entire stack, *Hex-A-Bug* will not cause a stack overflow.

Saving the stack also allows you to add items to the stack without disturbing *Hex-A-Bug*'s stack entries. This is useful for testing subroutines that expect data to be passed on the stack. For example, subroutines called from BASIC (with the USR function) get both optional arguments and the number of arguments on the stack. When data is pushed (added) to the stack, the stack register is decreased. This adds the data from high to lower memory locations. The stack strip has its direction switch set to 0, so you can easily "push" data onto your program's stack in the correct direction. To do this, add the items starting at the byte pointed to by the S register. The stack strip is automatically initialized to this point on entry to *Hex-A-Bug*, or whenever you change the S register using the register strip. After entering the data, the byte that the stack strip is pointing to (the one your cursor ended up in) is the new stack register value. Be sure to *enter this value in the S field* in the register strip to complete the process.

The stack strip works like the 6502's stack in that it wraps from the end of the stack save area ($00) to its start ($FF). In other words, the page of memory the stack strip displays is always the same, and the label line refers to the byte number in that page. This is why the stack strip doesn't have an address field in its header.

Code strip

The code strip is another special-purpose memory strip used to view and modify your program's code. When a breakpoint is hit, *Hex-A-Bug* sets up the code strip to display the memory at that breakpoint. It works as if you typed the new value of the PC into the code strip's address field after each breakpoint.

Disassembling code

The code strip can display data in hex or ASCII, just like the other strips, but it can also disassemble code. By pressing CTRL/W you can open a "code window" to display eleven lines of instructions at a time. This window uses all the lines of the screen from the code strip's header to the message line. This display mode is for viewing instructions only; you can't position the cursor in the code window to change values.

Each line in the code window has four fields. An example is the following:

```
FF9F    20600E    JSR $0E60
```

Figure 3 Sample Code Window Line

From left to right these fields are:

(1) The address of the instruction

(2) The machine code at that address

(3) The instruction's assembly language mnemonic

(4) The instruction's operand

If the instruction is a branch, an additional field displays the address that the program branches to if the condition is true. For example:

```
FFC9    D020    BNE    $20 ↨$FFEB
```

Figure 4 Sample Code Window Line with a Branch Instruction

If the data at the address is not a valid instruction, ??? displays in the mnemonic field.

Scrolling in the code window

To open the code window, position your cursor either in one of the strips above the code strip, or in the code strip's header line, or in the command line, but **not** in the code strip's data line. Press CTRL/W to open or close the code window. Once you open the window, you can scroll it up by pressing CTRL/N (for "Next"). After you've scrolled the window up, use CTRL/P (for "Previous") to scroll it down. Since it's impossible to disassemble backwards in memory, CTRL/P scrolls the window down only until it reaches its original address.

The address of the code strip is used as an implied argument to the SKP command. The SKP command is described in the next section.

Register strip

The register strip displays your program's values of the 6502 microprocessor's internal registers. Each of the P (processor status) register's bits displays individually for your convenience. To change any register, move the cursor to the field for that register and type in the new value. Note that any non-zero value entered in a P register bit causes that bit to be set to a logic 1 state. To start or restart execution of your program at a new location, change the PC (program counter). If you're restarting the program from the beginning, you may want to reset the S (stack) register to the start ($FF).

If you change the address in the PC register, *Hex-A-Bug* does two things after you move your cursor out of the register strip:

(1) It checks to see if the data at the new address is a valid 6502 instruction. If it isn't, the message DATA AT ADDRESS NOT INSTRUCTION displays and your cursor is positioned at the start of the PC field so you can fix it. *Hex-A-Bug* can't tell if the data at the location is actually an operand when it happens to be an instruction, too, which means it can't prevent you from setting the PC to a data address if the contents of that address could also represent a valid 6502 instruction.

(2) The code strip (or window) is set to view the new address.

If you change the S register and then move your cursor out of the register strip, *Hex-A-Bug* sets the stack strip to view the part of the stack save area indicated by the register.

Note that pressing the START key to execute a command invokes the above "implied" commands just as if you moved your cursor out of the register strip. For example, you might change the PC and press START to execute a CONT command in the command line without moving your cursor out of the register strip.

Breakpoint strip

The breakpoint strip has seven fields, one for each breakpoint. Entering an address into one of the fields causes execution to switch from your program to *Hex-A-Bug* at that point. When you enter a new breakpoint, four things happen automatically:

(1) If the address is 0000, then the breakpoint is cleared. Note that this means you can't set a breakpoint on the first byte in page zero.

(2) The location is checked to be sure that it holds a valid 6502 instruction (the same as described for the PC in the register strip). If it doesn't, the error message DATA AT ADDRESS NOT INSTRUCTION displays.

(3) A check is made to be sure that the location is in RAM (read/write memory). If it isn't, the error message SORRY, ADDRESS IS IN ROM displays. Because *Hex-A-Bug* temporarily replaces the instruction with a BRK instruction, all breakpoints must be in RAM.

(4) If the address passes the above tests, the instruction at the address is saved so that it can be restored to its original value.

If the address fails one of the tests in actions 2 and 3, then an error message displays, a beep sounds, and the cursor is positioned at the start of the offending address so that you can fix it.

If you want to "patch" your code by changing an instruction, first clear any breakpoint at that instruction by typing 0000 in its breakpoint field. After you change the instruction, you may reset the breakpoint if you wish. This will ensure that *Hex-A-Bug* restores the correct instruction to the location.

For more detailed information on the breakpoint system used, see the section on theory of operation.

Message line

Hex-A-Bug uses this line to display messages. Most error messages display for several seconds. However, some messages stay in the message line until another message replaces it or until you clear the message line by pressing the SHIFT/CLEAR keys.

Command line

Cursor control

Press the OPTION key to move your cursor to the command line. To return to the current strip in the main screen, press OPTION again.

The cursor acts somewhat differently in the command line. In the main screen area, the character you type replaces the one under your cursor. In the command line, *Hex-A-Bug* inserts a new character at the cursor. The character is put at the location of your cursor as before, but the characters under and to the right of your cursor move right one character to make room for the new one. The CTRL/left arrow and CTRL/right arrow keys work as before (without wrapping), but the CTRL/up arrow and CTRL/down arrow keys move your cursor to the start of the command line. DELETE BACK S deletes the character to the left of the cursor. CTRL/DELETE BACK S deletes the character under the cursor. SHIFT/DELETE BACK S clears the entire command line and places your cursor at its start.

General syntax

All the commands executed from the command line have a common syntax. The command and its arguments must use uppercase letters. In fact, in only one case might you want to use a lowercase letter in *Hex-A-Bug*: when modifying memory in ASCII mode.

On entry to *Hex-A-Bug*, the operating system (OS) variable SHFLOK (at $02BE, which controls the selection of CAPS/LOWR lock) is saved and set to CAPS lock. On exit, the original value of SHFLOK is restored. All commands must start in the far left column of the command line and have a space after the command.

Command argument

If a command has an argument, the argument must start one position after the required space. Because *Hex-A-Bug* ignores the rest of the command line after the completed argument (if any), you can leave other commands, arguments, or characters you want in the command line for future use or reference. Here is an example of the DOS command with A as its argument:

DOS A <all other characters are ignored>

Press the START key to execute a command. You can invoke the first command in the command line by pressing START regardless of where your cursor is located.

Descriptions of the *Hex-A-Bug*'s commands follow.

DOS – go to the Disk Operating System's menu

The DOS command works like the DOS command in BASIC. It causes *Hex-A-Bug* to turn control over to the Disk Operating System so that you can use the DOS utilities. For example, you would use the DOS menu option L (Load binary file) to load an assembly language program to test it with *Hex-A-Bug*. If you want to return to *Hex-A-Bug* from DOS, use DOS menu option M (Run at address). The address for *Hex-A-Bug* is $BEF0. This is different from the entry point placed in the BRK instruction vector. See "Theory of operation" for more details.

The DOS command has one optional argument. If you type an A (for Abort) after the required space, *Hex-A-Bug* releases the memory it uses (by changing RAMTOP and RAMSIZ) before going to DOS.

Note. The DOS command uses the operating system vector DOSVEC ($000A). If your program changes this vector, the DOS command won't work properly. After you debug your program, you can add the code to "steal" DOSVEC so that your program will reexecute when you press the SYSTEM RESET key.

CONT – continue execution of your program

This is the main command for returning to your program from *Hex-A-Bug*. It does the following:

(1) It restores the values of the CPU's registers.

(2) It restores the stack from the save area.

(3) It restores the OS locations that *Hex-A-Bug* uses. This includes switching the screen back to your program's display.

(4) It starts execution at the location indicated by the PC register.

(5) It sets all seven breakpoints.

All the commands that return to your program do the first four steps. CONT has two optional arguments. Option Z (i.e., CONT Z) omits setting the breakpoints (step 5). This option is convenient when you want to continue running your program without any breakpoints. It saves you from typing 0000 into all the breakpoint fields.

Option A (Abort), like the Z option, doesn't set any breakpoints, but it also releases the memory used by *Hex-A-Bug* like the A argument to the DOS command.

SS – single step through your program

The SS command does the first three steps of the CONT command, and then it executes the single instruction at the PC's address. SS determines the address of the instruction after the current one and sets a temporary breakpoint at that location. Note that this means you can't single step through ROM. However, Atari's excellent documentation on the use of the OS routines in ROM should keep you from wanting to single step through ROM. The SKP command (described next) is a convenient way to skip over a call to a ROM routine and stop at the next instruction to check the result.

SKP – set a temporary breakpoint at the code strip's address

The SKP command does the same thing as the CONT command, plus it sets a temporary breakpoint at the location of the code strip's address. The disassembling code window makes it easy to find the location desired.

This command is convenient in a number of cases. You can use SKP to skip over a loop when you're single stepping, thus saving the time of single stepping dozens of times through a loop that you know works. It's also useful when single stepping through a high level routine that calls many proven subroutines. In this case you don't need to single step through each routine in turn, but you may want to check the results of each routine. Using the code window, you can scroll past the calls you want to skip and execute SKP.

Scrolling strip commands

Use the following commands in conjunction with one of the memory strips (including the stack and code strips). These commands use the current strip (the strip bordered in red). If you execute one of these commands when the current strip is not a memory strip, the error message SCROLLING STRIP ONLY displays. If the current strip hasn't been set to a location, the error message STRIP NOT IN USE displays.

Each of the following commands changes the memory location a strip is viewing. Each has a complementary command that displays the new location immediately instead of scrolling to it. You can invoke these "fast" versions of the commands by adding the letter F to the desired command. You can't use the fast versions in the stack strip, but you rarely need to scroll this strip more than a few bytes at a time.

To stop the scrolling versions of these commands, press the BREAK key. A description of each command follows.

SE and SEF – search for data strings

Use the SE command to search for a string of bytes starting at the address of the current strip. SE works in either direction. The general syntax is:

SE [delimiter][hex #][space][hex #][space]...[delimiter][optional <]

The comments in brackets [ ] describe one character or number. The first character in the argument is a delimiter. You can use any character for a delimiter, as long as you use the same character to end the data string. Following the first delimiter is the first byte of the data to search for. Data bytes less than $10 must have a leading zero. One space is required between each byte. You can search for any length string that will fit in the command line. The completed string is as it will appear in the data line of the strip (assuming the strip is in hexadecimal mode). The preset search direction is from low to higher memory. You can search in the other direction by appending a < after the second delimiter. For example,

```
SE /01 23 45 67 89 AB/<
```

This example searches for the string "01 23 45 67 89 AB", from high memory to low memory.

The fast version of this command searches all of memory within four or five seconds and uses the same syntax. If the string isn't found, the message SEARCH FAILED displays.

CR and CRF – scroll to higher memory in steps

Use the CR command, which stands for "Cursor Right", to scroll the current strip to higher memory locations. The two-character (8-bit) hexadecimal number of locations to scroll is the one argument to this command. A leading 0 is not needed for hex values less than $10. The general syntax is:

CR [hex #]

For example,

CR 1B

This command is useful for stepping through tables. Use the length of each entry in the table as the argument to CR to look at each entry in turn. The fast version of this command is useful for looking at a location that is accessed using indexed addressing. Open a window to the base address, and use CRF with the offset value to display the desired location immediately.

With its powerful set of indexed addressing modes, the 6502 lends itself to a programming style using lots of tables of data. This is especially true with programs that run in real time and require lots of decisions and calculations. When a time-consuming calculation can be replaced by a table lookup, the speed and overall performance of the program increase. *Hex-A-Bug* is especially good for testing this kind of program. The memory strips are great for experimenting with new values in tables and rerunning the program to see the result. You can then enter the best values into your source code during your next editing session.

CL and CLF – scroll to lower memory in steps

The CL command, which stands for "Cursor Left", is the same as CR, except that the scrolling direction of the strip is from high to lower memory locations.

SMOOTH – turn on and off smooth scrolling

Use the SMOOTH command to turn on or off the smooth scrolling feature in the memory strips. *Hex-A-Bug* starts out with it on. To turn it off, use the command with no argument. To turn it back on, use the command with a 1 as the argument. The SMOOTH command doesn't change the location viewed by a strip, and you can use it regardless of the current strip's function. Examples are as follows:

SMOOTH (turns off smooth scrolling)

SMOOTH 1 (turns on smooth scrolling)

A message in the message line indicates the state of the feature (ON or OFF).

The hardware register used to control smooth horizontal scrolling (HSCROL) is a "write only register". In other words, even though you store values to this location, just like any other memory location, it isn't really a memory location. If you view this location, the value $FF always displays. This makes it impossible for *Hex-A-Bug* to save and restore your program's value for this register.

To demonstrate this, view the location HSCROL (at $D404) using *Hex-A-Bug*. The last value stored in HSCROL when the current strip isn't scrolling is $00, but $FF displays. Now try to change the location to $0F. The error message SORRY, ADDRESS IS IN ROM displays to warn you you're trying to change ROM, and the strip scrolls sixteen color clocks (four whole characters) to the right. *Hex-A-Bug* assumes that a location that doesn't change after a store is in ROM, but you actually did write a $0F into the hardware register, changing the display accordingly.

If your program uses smooth scrolling, you may want to turn off the SMOOTH feature in *Hex-A-Bug* so that the value your program last stored into HSCROL isn't disturbed. You can still use the memory strips, but they may be a little off center. The display will be impossible to read while scrolling at the faster speed, so you can use this feature to show your friends the value of the ATARI Computer's smooth horizontal scrolling.

Change PC to CONT or use DOS cmd.

This is a reminder that the address in the PC was set to 0000 when you entered Hex-A-Bug using the entry point $BEF0. Therefore, you must change it to the start of your code to start debugging, or you can use the DOS command to exit to the DOS menu.

CIO ERROR NUMBER

The displayed error was returned after a call to the Central Input Output routine (in the Operating System). *Hex-A-Bug* uses the CIO to set keyboard characters by opening the K: device.

The IOCB (Input Output Control Block) number 7 may have been in use. Look up the error number (displayed in decimal form) in your DOS (or other) manual to determine the problem.

A CIO error is usually fatal (that is, it locks up your computer). Be sure your program isn't using IOCB #7, and that it isn't disturbing the control block itself.

DATA AT ADDRESS NOT INSTRUCTION

You're trying to change the PC to, or set a breakpoint on, a location that doesn't hold a valid 6502 instruction. Check the address and try again.

DELIMITER ERROR

The syntax used in an SE or SEF command was wrong. Use the same delimiting character to end the string as you used at the beginning of the string.

NO FAST COMMANDS IN STACK

You're trying to use the fast version of a scrolling strip command in the stack strip. Use the smooth scrolling version of the command instead.

NO SUCH COMMAND

You're trying to execute a command not included in *Hex-A-Bug*. Remember that all commands require a space between the command and any arguments. Check the quick reference sheet for a list of commands.

PLEASE ENTER HEX NUMBER: 0 - F

You typed a character that isn't within the required range of 0 - F. Your cursor didn't move; type a valid hexadecimal digit.

SCROLLING STRIP ONLY

You're trying to use a scrolling strip command in a nonscrolling strip. Move to a scrolling strip.

SEARCH FAILED

The SEF command returns this message when the data string you were searching for wasn't found in memory.

SORRY, ADDRESS IS IN ROM

You're trying to change the value of a memory location or set a breakpoint in ROM (read only memory), which is impossible. When you're using the SS command, this message appears if the next instruction is a call or jump to a routine in ROM, since SS sets a temporary breakpoint on the next instruction after the current one. Use the SKP command to stop execution on return from the call to the subroutine in ROM.

STRIP NOT IN USE

You're trying to use a scrolling strip command in a strip that isn't displaying memory yet. First type the address into the header field to view the desired memory.

Theory of operation

Hex-A-Bug loads into memory automatically using the AUTORUN.SYS feature of Atari DOS. It occupies the top 12K of memory, ($9000-$C000) moving the OS pointers RAMSIZ and RAMTOP down to reserve this space. Any program that can be run on an ATARI Computer with 32K of RAM can be debugged with *Hex-A-Bug* on an ATARI Computer with 48K of RAM. If you have any programs that use the same trick to reserve memory, you can move RAMSIZ and RAMTOP down to make room for them from *Hex-A-Bug*, and then load them from DOS. Of course, they will have to be set to start below *Hex-A-Bug*. There is also some memory not used by *Hex-A-Bug* between $B8D1 and $BEEF that is reserved for future enhancements. Lots of application programs assume that RAMTOP and RAMSIZ are at a 4K boundary so *Hex-A-Bug* was set to start on one, leaving some room to spare.

You can run any programs that adhere to the Operating System rules regarding memory usage while *Hex-A-Bug* is in memory, including the ATARI MACRO Assembler and Program-Text Editor. This feature saves the time to reboot and load *Hex-A-Bug* during the debug-edit-fix cycle. The main disadvantage is that the editor will have less space for holding text. If you're working on a small program, this limitation may not affect you. If your new program accesses the disk, it's a good idea to move your new .OBJ file onto a "TEST" disk that *doesn't* have any valuable source files on it, just in case your program trashes the disk.

Once *Hex-A-Bug* is in memory, you can reenter it in two ways. You can use DOS menu option M (Run at address), using $BEF0 as the address. Or, you can run any code that has a BRK instruction ($00) in it.

Let's look at the differences between these two options. When your program hits a breakpoint, *Hex-A-Bug* does a number of things before it gives you control of its cursor:

(1) It saves all of the 6502's internal registers

(2) It saves the stack (page 1)

(3) It restores the original instructions at all breakpoints

(4) It saves all of the OS variables it uses

(5) It replaces the OS vertical blank routine with its own routine

(6) It refreshes its display so that all the values are updated

When you enter *Hex-A-Bug* using the "jump location" $BEF0:

(1). All the registers are cleared to $00 (except the S register, which is reset to $FF)

(2) It saves the stack

(3) It saves all of the OS variables it uses

(4) It replaces the OS vertical blank routine with its own routine

(5) The DOS command is placed in the command field

(6) A message in the message line reminds you that you must change the PC to continue (unless you want to run code at location $0000)

(7) It refreshes its display

Note that step 3 in the breakpoint entry is *not* done in the JUMP entry. This is important if you want to debug two different programs (or different versions of the same program) that load in the same memory, one after the other. If you aren't careful, *Hex-A-Bug* will restore the instructions from the first program into the second when it hits a breakpoint. The safest procedure in this case is to clear all breakpoints to $0000 before starting to debug the second program.

All of RAM is initialized to $00 on cold start, so running at any unused location will result in entering *Hex-A-Bug*. The easiest way to use *Hex-A-Bug* is to include a BRK instruction as the first instruction in your new programs, as described in the introduction to "Using *Hex-A-Bug*".

When your program hits a breakpoint, *Hex-A-Bug* saves any OS variables that *Hex-A-Bug* uses so that it can restore them when you return control to your program, or when you press the SELECT key to see your program's display. Here is a list of the variables and the locations in *Hex-A-Bug* they're saved in:

TABLE 1
OS Variables used by Hex-A-Bug

Variable name	Saved addr.	Actual addr.	Description of use
ATACHR	$95F0	$02FB	Character save used by OS keyboard routine
BRKKEY	$95F2	$0011	BREAK key flag.
CH	$95F6	$02FC	Character save byte used by keyboard interrupt routine.
INVFLG	$95F1	$02B6	Inverse flag; used by keyboard routine to show state of inverse lock (ATARI key)
IOCBAS	$95E3	$0020	Base of zero page IOCB. 12 bytes at this address are saved
SHFLOK	$95F3	$02BE	Controls CAPS/LOWR lock function
VBREAK	$95DF	$0206	Break vector, this is saved when Hex-A-Bug is loaded, and restored when the abort option is used with the DOS or CONT commands
VDSLST	$95DC	$0200	Display list interrupt RAM vector
VVBLKI	$95E1	$0222	Vertical blank RAM vector

To change any of the above locations with *Hex-A-Bug*, change the saved location, not the actual location.

Note that the system Vertical Blank routine is replaced by *Hex-A-Bug*'s own routine. This means that the system software timers won't be incremented while in *Hex-A-Bug*. This is important when you're using the VBLANK routine to keep track of real time events you want to debug. The hardware timers will continue to tick, however.

Hex-A-Bug moves all players and missiles off the screen by setting their horizontal positions to 0. These hardware registers are write only registers, like the previously described HSCROL register. When using these registers, a good programming practice is to use "shadows" of them. Shadows are memory locations that keep track of the value last stored in a hardware register. Shadows should be used by the vertical blank interrupt routine to set the value in the register itself. Then, since *Hex-A-Bug* uses a custom vertical blank routine, the players will remain off the screen while in *Hex-A-Bug*. As soon as you return to your program, or press the SELECT key, your program's VVBLKI (vertical blank RAM vector) will be restored, and the players will become visible again. You needn't write a custom vertical blank interupt routine; you can use the deferred vertical blank vector to run your VBLANK code after the OS routine. For more information, see the Technical Users Notes, available from Atari.

Another read-only register used by *Hex-A-Bug* is NMIEN, nonmaskable interrupt enable. Since *Hex-A-Bug* uses display list interrupts, they are enabled by setting bit 7 of this register. Therefore, display list interrupts will be enabled upon exiting *Hex-A-Bug*.

Every time you enter *Hex-A-Bug*, IOCB 7 is open to K: for its use to get keyboard input. When you leave *Hex-A-Bug* (that is, when you restart your program), the IOCB is closed. Your program must not use this control block.

Breakpoints are set (replaced with a BRK) only when you restart your program. This allows you to view the code being executed instead of seeing BRK instructions on all your breakpoint locations. The instructions are restored every time a breakpoint is hit.

Hex-A-Bug uses locations $80-$9A in page 0; therefore, your program can't use these memory locations. The first half of zero page $00 to $80 are used by the OS. The locations from $9A to $FF are available to your program.

Glossary of computer terms

6502: The microprocessor used as the ATARI Computer's central processing unit (CPU).

A: A register, or Accumulator. The CPU register used for most arithmetic instructions.

Abort: To stop execution of a program before its intended end.

Address: An identification describing a specific memory location. In the 6502, an address consists of four hexadecimal characters (16 bits).

Argument: A variable to which either a logical or a numerical value may be assigned.

ASCII: Acronym for American Standard Code for Information Interchange. A code for the representation of alphanumeric data, i.e., characters, adopted to facilitate the interchange of data among various types of data processing and data communications equipment.

Assemble: To translate assembly language into its corresponding machine code.

Assembly language: The low level programming language unique to each computer that lets a programmer use mnemonics instead of numeric instructions. The language closest to the machine language codes the computer can execute directly.

Base address: A specified address combined with a relative address to form the absolute address of a storage location.

Bit: A binary digit; the smallest unit of computer storage, and the basis of all digital computing. One bit can store a value of 1 or 0.

Branch instruction: A program instruction providing a means for choosing between alternative paths, based on the state of a P register bit. The program branches to a different location if the required condition is true; otherwise, execution continues with the next instruction.

Breakpoint: A point in a program where execution of the CPU is interrupted and control passed to the monitor or to a debugging utility.

Bug: A mistake in a program preventing the program from working as planned.

Byte: Eight adjacent binary digits operated on by the computer as a unit.

Code: Numbers in memory that can be executed by the computer. Machine code.

Color clock: The standard unit of horizontal distance on the television screen. A horizontal scan line has 228 color clocks, but only 160 are displayed in a normal width playfield.

CPU: Acronym for Central Processing Unit. The main logic circuit in a computer that interprets and executes machine code instructions.

Cursor: A symbol on the display screen indicating where the next character typed in will appear.

Debugger: A program designed to help find and correct errors (or bugs) in programs.

Delimiter: A character used to separate variables in a list or one string of characters from another.

Disassemble: To translate machine code to its corresponding assembly language instructions.

Display list: ANTIC's "program" defined by the user or provided automatically (through a GRAPHICS command) in BASIC. The display list specifies where the screen data may be found, what display modes to use to interpret screen data, and what special display options (if any) may be implemented. (ANTIC is the ATARI Computer's separate, programmable microprocessor dedicated to the television display.)

Hex: Abbreviation for hexadecimal.

Hexadecimal: A numeral system with base 16. Digits greater than 9 are represented by letters. The letters A-F stand for the decimal numbers 10-15 and hexadecimal 10 equals decimal 16. Hexadecimal numbers are usually prefixed with a $ in text (e.g., $10).

Hex-A-Bug: A debugging utility designed to work on the ATARI Computer.

High level language: A computer language more nearly like English and oriented toward the problem to be solved or the procedures to be used. The higher the language, the less it resembles the machine code executed directly by the CPU.

Horizontal fine scrolling: The process of sliding the screen window to the left or right over display memory in color clock or scan line increments to display more information than could be seen with a static screen.

Indexed Addressing: A system that modifies an address by the content in an "index register" prior to or during execution of a computer instruction to compute the final address of the desired data.

K: Two to the tenth power (1024) when referring to storage capacity. For example, 12K represents 12,288.

Loop: A series of instructions that are executed repetitively until specific conditions are met.

Mnemonic: A word or name for a machine language instruction that is easy to remember and identify. Assembly language is made up of these words or instructions.

Modular programming: The technique of designing a program as a number of logically self-contained units.

Offset: The difference between the value or condition desired and that actually attained.

Operand: The symbols or data following a program instruction indicating what registers, memory locations, or data values are to be used in executing the instruction.

OS: Acronym for Operating System. The program providing basic routines many programs can use to perform common tasks. The ATARI Computer's OS is in ROM.

P: Processor Status register. The 6502 register used to store the internal "flags" or individual bits indicating special conditions in the CPU. The 6502 flags are: N-sign, V-overflow, B-break, D-decimal, I-interrupt, Z-zero, and C-carry.

Page: A portion of memory starting and ending on even, 256-byte boundaries. For example, locations $0000 to $00FF are all in "page 0", since the most significant byte (the first two characters) of the address of all these locations is $00.

Patch: Computer slang for a temporary fix of a bug. A section of coding inserted into a program to correct a mistake.

PC: Program Counter. The register the CPU uses to keep track of the address of the instructions it is executing.

Pop: To retrieve data from the top of a program push down stack; the stack pointer is incremented to address the last word pushed on the stack and the contents of this location are moved to one of the accumulators or to another register.

Push: To put data into the top location of a program stack; the stack pointer is decremented to point to the next location, which becomes the top of the stack.

Push down stack: A set of memory locations or registers in a computer that implements a push down list (a list written from the bottom up, with each new entry placed on top and with the item on top the one processed first).

RAM: Acronym for Random Access Memory. The main memory chip used with the 6502 that can be written to as well as read, but whose contents are lost when the power is shut off.

Real time: A term describing online computer processing where the speed of the program is an integral part of the program's design and the output of processed data affects or controls the outcome of an ongoing activity. Arcade style games are one example.

Register: A high-speed device in the CPU or other circuit used to store data or intermittent results temporarily during processing.

ROM: Acronym for Read Only Memory. Non-erasable, permanently programmed memory used to store programs and data. ROM cannot be written to.

S: S register or Stack register. The CPU register pointing to a memory location where the next stack data will be stored. The stack is a "first in, last out" (FILO) memory area used to store data temporarily. The CPU uses it to keep track of return addresses during subroutine calls.

Shadowing: A process in which values are moved between hardware locations and RAM locations, thereby allowing the program to monitor the contents of write-only hardware registers or check the input from read-only hardware registers.

Single step: To operate a computer by executing each computer instruction or part of an instruction in response to a manual operation.

Subroutine: A routine nested within another routine, within which initial execution never begins.

Syntax: The formal grammatical and structural rules of any assembly or higher level programming language.

Table: A collection of data often stored in consecutive storage locations or written as an array of rows and columns in which an intersection of a labeled row and column locates a specific piece of information. Data in tables is usually accessed using indexed addressing.

Toggle: To alternate between two states.

Variable: A quantity that can assume any of a given set of values.

Vector: A data structure permitting the location of any item by the use of a single index or subscript; often used to store an address of another memory location where a program or subroutine starts.

Vertical blank: The period during which the electron beam (as it draws the screen image) returns from the bottom of the screen to the top. This period is about 1400 microseconds.

Word: A group of bits, characters, or bytes considered an entity and capable of being stored in one storage location. In the 6502, a word is 2 bytes, or 16 bits.

Wrap: In *Hex-A-Bug*, to move the cursor positioned at the end of a row or column of the screen display to the initial position of the same row or column, depending on the direction of cursor movement.

X: X register. A 6502 register. One of two "index registers".

Y: Y register. A 6502 register. One of two "index registers".

Function keys

Key	Function
OPTION	Moves cursor from command line to current strip or vice versa
SELECT	Toggles between program screen and Hex-A-Bug screen
START	Executes first command in command line
CTRL/	Moves cursor up
CTRL/	Moves cursor down
CTRL/	Moves cursor left
CTRL/	Moves cursor right
TAB	Moves cursor to next field
CTRL/C	Toggles conversion in data lines between ASCII and hex
CTRL/W	Opens and closes disassembly window in code strip
CTRL/N	Scrolls disassembly window to next instruction
CTRL/P	Scrolls disassembly window to previous instruction

Commands

Name:	CONT (Continue program)
Syntax:	CONT (or) CONT Z (or) CONT A
Use:	Continue execution of your program starting at the address indicated by the PC field. Sets all break points.
Name:	DOS (Go to Disk Operating System)
Syntax:	DOS (or) DOS A
Use:	Jump through DOSVEC to go to the DOS menu. A aborts Hex-A-Bug.
Name:	SS (Single Step)
Syntax:	SS (no meaningful arguments)
Use:	Execute the single instruction at the address in the PC field. and reenter Hex-A-Bug.
Name:	SKP (Skip to address of code strip)
Syntax:	SKP (no meaningful arguments)
Use:	Same as CONT, but sets a temporary breakpoint at the Address of the code strip.
Name:	SE and SEF (Search)
Syntax:	SE /## ## ## ##/
Use:	Searches for the string of values using the current scrolling data line. The BREAK key aborts the search.
Name:	CR and CRF (Cursor right)
Syntax:	CR ## (where ## is a hex value)
Use:	Scrolls the current memory strip ## locations higher in memory.

Name: CL and CLF (Cursor left)

Syntax: CL ## (where ## is a hex value) Use:
Scrolls the current memory strip ## locations lower in memory.

Name: SMOOTH

Syntax: SMOOTH (off) or SMOOTH 1 (on)

Use: Turns on or off smooth scrolling feature in memory strips.

APX
ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

Review Form

We're interested in your experiences with APX programs and documentation, both favorable and unfavorable. Many of our authors are eager to improve their programs if they know what you want. And, of course, we want to know about any bugs that slipped by us, so that the author can fix them. We also want to know whether our instructions are meeting your needs. You are our best source for suggesting improvements! Please help us by taking a moment to fill in this review sheet. Fold the sheet in thirds and seal it so that the address on the bottom of the back becomes the envelope front. Thank you for helping us!

1. Name and APX number of program.

2. If you have problems using the program, please describe them here.

3. What do you especially like about this program?

4. What do you think the program's weaknesses are?

5. How can the catalog description be more accurate or comprehensive?

6. On a scale of 1 to 10, 1 being "poor" and 10 being "excellent", please rate the following aspects of this program:

   _____ Easy to use
   _____ User-oriented (e.g., menus, prompts, clear language)
   _____ Enjoyable
   _____ Self-instructive
   _____ Use (non-game programs)
   _____ Imaginative graphics and sound

7. Describe any technical errors you found in the user instructions (please give page numbers).

8. What did you especially like about the user instructions?

9. What revisions or additions would improve these instructions?

10. On a scale of 1 to 10, 1 representing "poor" and 10 representing "excellent", how would you rate the user instructions and why?

11. Other comments about the program or user instructions:

From

STAMP

APX ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

[seal here]

HEX-A-BUG
by David Kano

- **Set breakpoints in your assembly language programs to track down bugs**
- **Study memory locations and register values at intermediate stages to locate errors**
- **Switch back and forth easily between your program and *Hex-A-Bug***

New programs rarely work as planned on the first run. But finding the errors is difficult at speeds at which the computer usually runs. *Hex-A-Bug* is an easy-to-use tool for stopping your program so you can find and correct the bugs. You load *Hex-A-Bug* and your program into memory, and you use breakpoints to switch control from your program to *Hex-A-Bug*. Being able to determine the intermediate results of your program by studying memory locations and register values can be invaluable for locating errors. Your program's screen display remains intact, and you can easily toggle between it and the *Hex-A-Bug* display.

This screen-oriented program uses very few commands. The main screen area consists of "strips" across the screen, each strip being one functional area. You move a flashing cursor from one functional area to another. In this way, you can change the contents of any register, breakpoint, address of a memory strip, or memory location. Horizontal fine scrolling forwards and backwards from any location gives you quick and easy access to all information. In addition, you use simple commands to go to DOS, single step through your program, search for a string of values, and continue executing your program.

Requires:
Diskette
(APX-20199)
- ATARI 810™ Disk Drive
- 48K RAM

Optional:
- ATARI MACRO Assembler
- ATARI Assembler Editor Cartridge

About the author

DAVID KANO

When the surf's up, David Kano's thoughts turn from his computer to his other hobby. The author of HEX-A-BUG is an ardent windsurfer who has competed on the world championship level, traveling as far as Italy and Mexico. In inclement weather, David writes home computer programs, striving to make them easy for hobbyists to use. Before starting to free-lance from his home in Lincoln, Massachusetts, David was a programmer at a company that produces systems for the publishing industry.

Diskette: version 1
Edition B