REMark

Issue 5 • 1979

HAPPY NEW YEAR

:JB:

Official magazine for users of Heath computer equipment.

HUG MATERIALS AVAILABLE TO HUG MEMBERS

HERE IS A COMPLETE LIST OF MATERIALS AVAILABLE TO MEMBERS TO DATE.

HUG BINDER		885-4	$ 4.00
HUG TEE	S	885-1100	$ 4.50
SHIRTS	M	885-1101	$ 4.50
	L	885-1102	$ 4.50
SOFTWARE TAPE I		885-1009	$ 7.00
SOFTWARE VOLUME I		885-1008	$ 9.00
ADVENTURE (H8) (disk)		885-1010	$10.00
HDOS PROGRAMMING GUIDE		885-1018	$ 5.00
HDOS DEVICE DRIVER		885-1019	$10.00

NOTE: Always place your orders on the green order form and include payment plus shipping and handling.

on the stack

>CAT

"REMark" is a HUG membership magazine published quarterly. A subscription cannot be purchased separately without membership. The following membership rates apply.

	U.S. Domestic	Canada & Mexico	Internat'l
Initial	$14	$16	$24
Renewal	$11	$13	$18

Membership in England, France, Germany, Belgium, Holland, Sweden and Switzerland is acquired through the local distributor at the prevailing rate.

Send payment to: Heath User's Group, Hilltop Road, St. Joseph, MI 49085. Back issues that are available cost $2.50 postpaid to U.S. destinations. Request for magazines mailed to foreign countries should specify mailing method and add the appropriate cost.

HUG Manager and Editor ..... Jim Blake
Graphics ..... Ron Hungerford

-REMark

HT11/H27 TEST DRIVE:JB:

This magazine is 32 pages in length and the system manual for HT11 is almost two inches thick, so we aren't going to cover much here, but perhaps it will be interesting for some — so let's power up, insert the system diskette in the left drive, a scratch diskette in the right hand drive and close the doors. Type 'DX' and let HT11 do some housekeeping and check things over.

See opening HT11 dialogue on page 10.

That dot indicates that we are in the Keyboard Monitor Command mode. From here we can:

1) Reassign drives (give one a new name) although it would be rare to need to.
2) Run programs with .SAV extention. Programs with .SAV extention are HT11 executable programs such as BASIC, PIP, EDIT. User programs that have been properly prepared by the EDITOR, EXPANDER and ASSEMBLER etc..
3) Change or read the DATE;
4) Change or read the TIME of day if the LTC is turned on.

The LTC (line time clock) is enabled by removing a jumper wire on the power supply board. (very shortly there will be a mod kit available to allow you to switch the LTC from the front panel of the H11). Also, you can Examine memory locations, Deposit new values in memory, GET memory images programs, SAVE them, LOAD device handlers and many other things (15 in all) plus all the SET options.

From here we can run PIP, for instance, and if you already are operating HT11 and have been storing your programs on paper tape, here's how to retrieve those programs and put them on disk.

.R PIP

*FNAME.EXT=PR:/A This assumes you are retrieving ASCII files such as BASIC or SOURCE code. After the file has been read in 'close' the file by typing control 'D'.

Then, of course, any modifications necessary to the program can best be made using 'EDIT'. To exit PIP, type control 'C'.

There's our keyboard monitor prompt again. If you want to go back to PIP, type 'RE' (reenter PIP). Let's use the EDITOR.

.R EDIT

* That is the prompt for HT11 EDITOR. All commands are terminated by the escape key and a '$' is echoed. To load your program for editing do this.

*EBFNAME.EXT$R$$ This tells the editor you want to load your program, but leave it on disk as FNAME.BAK for 'backup'. Therefore, when you have finished you will have both your original program, plus the edited version. After you are satisfied with your new version then you can 'delete' FNAME.BAK.

The HT11 Editor is both line oriented and character oriented. And you may 'build' a mini-macro to be repeatedly executed. For instance, let's say you are modifying a paper tape program to write to the disk instead of the paper punch and you wanted to change all the 'PRINT #1:' statements to 'PRINT #2:'. (crude example.) Here is how you could do that. Write a macro.

*M/G#1:$=C#2:$V$/$$ This says..G get #1: C change to #2: V and verify each line as it is done. To execute the macro, type

*EM$$ The change would be made one time. To execute NN times, type NNEM$$ — If he comes back and says 'Macro search failed' you know that all the #1's had been changed to #2's, neat!?

Other editor commands are:

'A' Advance the pointer + or − N lines.
'J' move the pointer + or − N characters
'D' delete N characters
'K' kill N lines
'C' change characters
'X' exchange 'string'
'B$' move the pointer to the beginning of the text
/A' move the pointer to the end of text.
'L' List N lines
'G' get 'string'
'S' save N lines of text in an invisable buffer
'U' unsave those same lines and insert them at the current line pointer
'I' insert text

And many of these commands can be combined on one line. Here is a ridiculous example, with explanation.

```
*B$2GBLAKE$=J$D$$IF$+4A$+5J$L$V$$
```

This says: Move the line pointer to the beginning of the next (B$)
- — Get the second occurrence of the string 'BLAKE' (2G)
- — Back the pointer up to the beginning of the string 'BLAKE'(=J$)
- — Delete one character from the string 'BLAKE' (D$)
- — Back the pointer up one (−1J$)
- — Insert 'F' (IF$)
- — Advance the line pointer 4 lines (+4A$)
- — Advance the character pointer into the line 5 characters (+5J$)
- — List the line from that point only (L$)
- — and finally, verify the entire line (V$).

After editing is complete, you exit the edit like so.

*EX$$ And your new FNAME is written to disk — don't type control 'C' at any time while in the editor — all text is lost!

And finally, to originate a new text —

```
*EWNEW.EXT$$  (EDIT WRITE NEWFILE.EXT)
```

Back to the keyboard monitor — let's run XBASIC and play with string arithmetic and 'print using'.(It may be named CBASIC in a few early releases.)

The PRINT USING statement gives you much greater control over the placement and form of numbers and strings in your output. The following examples illustrate the greater control available to you with PRINT USING.

```
10  A = 22.5
20  B = 1.5
30  C = 37
40  D = 123.47
45  A$ = "$$###.##"
50  PRINT USING A$,A
60  PRINT USING A$,B
70  PRINT USING A$,C
80  PRINT USING A$,D
```

The output from this program is:

```
$  22.50
$   1.50
$  37.00
$ 123.47
```

These examples only deal with output to your console terminal. You can also use PRINT USING for output to files, but the examples in the remainder of this section will only use output to your console.

The standard form of a PRINT USING statement is:

PRINT USING string, list

That is, the words "PRINT USING" start the statement. The string that follows specifies where and how the data is to be printed. The list is the same as a normal BASIC PRINT list.

The characters shown in the table below are used in format specifications.

FORMAT SPECIFICATION CHARACTERS

#	Numeric field allocation.
.	Decimal point location.
,	Use commas every three digits.
**	Use leading asterisks.
$$	Use leading dollar sign.
—	Use trailing minus sign.
↑	Use scientific notation.
'	Start a string field.

For our purposes now, we will say that any occurrence of one of these characters is part of a specification. Any other character is not part of a specification and will be printed as is. This is a simplification of the actual situation, but it helps introduce the ideas of PRINT USING.

Note that the single quote (') is used as a specification character. It cannot be used to delimit a constant string. You must use double quotes to delimit constant format strings on PRINT USING statements.

The following example demonstrates how to print commas between every three digits in an output number. Note the comma within the string of number signs. It is only necessary to use one comma, but at least one number sign must precede it.

```
PRINT USING "#,#########", 123456789
    123,457,000
```

PRINT USING WITH STRINGS

PRINT USING can be used on strings as well as on numbers. A field for a string is always started with a single quote (') character. The field can be extended with an unbroken string of L'S, R's, C's, or E's. The letter specifies how the string is to be printed in the field.

The following program illustrates some of these features.

```
10 A$="HT11"
20 PRINT USING "LEFT ADJ +'LLLLLLLLLLLL+",A$
30 PRINT USING "CENTER +'CCCCCCCCCCCC+",A$
40 PRINT USING "RIGHT ADJ +'RRRRRRRRRRR+",A$
RUNNH
      LEFT ADJ    + HT11              +
      CENTER      +          HT11     +
      RIGHT ADJ   +                HT11 +
```

The program below demonstrated what happens when the string is larger than the field. BASIC will simply print what it can, and discard the rest of the string. The "E" fields are "extendable", so the entire string is to be printed.

```
10 A$="BIG, FAT STRING."
20 PRINT USING " 'LLLLLL", A$
30 PRINT USING " 'CCCCCC", A$
40 PRINT USING " 'RRRRRR", A$
50 PRINT USING " 'EEEEEE", A$
RUNNH
 BIG, FA
 BIG, FA
 BIG, FA
 BIG, FAT STRING.
```

STRING ARITHMETIC

BASIC is identical to the regular HT11 BASIC, except it includes a PRINT USING statement and STRING ARITHMETIC. The details of these features will be described shortly. Note that BASIC does not need the EIS/FIS extended arithmetic chip, and will not take advantage of it if your H11 has one of these chips.

The normal arithmetic that is used on an H11 is only capable of about six and a half digits of accuracy. In addition, BASIC will only print six digits on output. BASIC extends this accuracy by using string arithmetic.

The following example illustrates most of the features of string arithmetic. Note, however, that string arithmetic and ordinary arithmetic cannot be mixed. You have to be certain to use quotes when you want BASIC to use string arithmetic.

```
A$="123456789.987654321"
B$='-987654321.123456789'
PRINT A$ + B$
PRINT A$ - B$
PRINT -A$ * B$
PRINT B$/-B$
```

BASIC also has some other neat features not found with the paper tape version.

CHAIN — While running one program, you can 'chain' to another and begin execution at any line within that program. This conserves memory.

OVERLAY — Unlike the chain command, you may 'overlay' or merge programs with variables in tact.

VIRTUAL ARRAY FILES — This gives you random access to disk files.

BASIC

To exit BASIC, type control 'C' twice — you may reenter BASIC without losing your program.

ASSEMBLY TOYS

For assembly language programmers, you will find 46 'programmed requests' or MACROS at your disposal which prevents having to 're-invent' the wheel so often — Also, there's a librarian to keep track of things for you and your own user written MACROS.

Some of the MACROS allow you to read and write to the disk or the outside world — Get time and date — Print text — Get character — Write character chain — Define registers — Set memory limits — And fetch handlers to name a few.

Speaking of MACROS — There is a routine on page 29 that will give you some experience with the system and one that is also useful. When you get it all done it will give you system status when you type RUN CONFIG — have fun.

Just received this note which may interest you — :JB:

Dear Jim:

Due to rapid expansion of our computer line, we have several openings which some of your readers might be interested in.

COMPUTER TECHNICIANS

Associates degree or equivalent military training.

COMPUTER DESIGN ENGINEERS

BS Computer Engineering or equivalent, with minimum of three years experience in designing computer systems hardware.

SOFTWARE DOCUMENTATION WRITER

BS Degree in Liberal Arts or Computer Engineering Service, strong hobby computer interest a must.

COMPUTER MANAGEMENT ENGINEER

Three to five years experience in operating systems. Must have BSCS or BSEE degree and/or strong technical and computer background.

All these positions, Jim, are located in Southwest Michigan and I'm sure you've heard that liberal salaries and outstanding benefits are all part of the "way" here at Heath.

If you could, Jim, ask any interested readers to contact me either with a resume or by phone for considerations or further information. My address is:

J. K. Bartley
Professional & Technical Recruiter
Heath Company
Benton Harbor, Michigan 49022

Telephone: (616) 982-3673

Thanks Jim, for passing this information to your readers and keep up the super work on REMARK.

J. K. Bartley

THE BASIC IDEA

By: Sam Cox

Here are a couple of short programs that should prove useful to members using extended BASIC.

The first is a BASIC TIMER. Using the H8 real-time clock, the program allows the user to determine the execution time (in milliseconds) of single BASIC statements or multiple-line routines.

The second is a DIRECTORY program. Inspired by the PUT/GET article in REMark issue 3, DIRECTORY implements a VERIFY " " command via the POKE function. The program will do an 'auto-verify' on all or part of a cassette. By jotting down the tape counter settings as individual files are located, the user will have a directory of the entire tape. I have found it convenient to DUMP the DIRECTORY program as the first file on each side of a new cassette.

BASIC TIMER

Purpose: To determine the execution time (in milliseconds) of single statements and multiple-line routines.

Procedure:

1. Enter statements to be timed in the available space between lines 202 and 297.
2. Enter any preconditions (eg. DIM statements) in the space between lines 101 and 197.
3. RUN

Adjustments and Modifications:

The statements to be timed are executed 100 times. If the total execution time for 100 passes is greater than 131 seconds, then reduce the number of repetitions.

With lines 202 to 297 left blank, the time for 100 passes should be zero, plus or minus 2ms. (NOTE: 2ms represents 1 count of TICCNT). If your system doesn't return 0ms, 2ms, or 131070ms, then adjust the initialization of TICCNT in line 25 until it does. LISTING:

```
 10  REM — BASIC TIMER
 15  CNTL 4,0
 20  PRINT "TESTED STATEMENTS";: LIST 202,297
 25  POKE 8220,255: POKE 8219,65
 30  GOSUB 100
 35  CNTRL4,1: GOSUB 200: CNTRL 4,0
 40  L=PEEK(8219)
 45  H=PEEK(8220)
 50  PRINT "TIME - 100 PASSES:";2*(256*H+L);"MS";
 55  CNTRL 4,1: END
 60  :
100 REM — PRECONDITIONS FOR TIMED STATEMENTS
198  RETURN
199  :
200  REM — TIMED STATEMENTS / LINES 202,297
201  FOR Z = 1 TO 100
298  NEXT
299  RETURN
```

DIRECTORY

Purpose: Locate and identify tape files.
'Auto-verify' an entire cassette.

LISTING:

```
 10  REM — DIRECTORY PROGRAM
 15  PRINT: PRINT "TAPE DIRECTORY"
 20  PRINT " INSERT AND REWIND CASSETTE"
 25  PRINT " SET RECORDER FOR PLAYBACK"
 30  INPUT " HOW MANY FILES ? ";F
 35  : IF F< 1 or INT(F)<>F GOTO 30
 40  PRINT " 'ESCAPE' KEY LETS YOU ABORT"
 45  PRINT " AFTER THE CURRENT FILE!"
 50  PRINT "HIT ' RETURN' OR 'ESCAPE' ";
 55  : PAUSE: Z=PIN(250)
 60  : IF Z=155  GOTO 135
 65  : IF Z<>141 THEN PRINT CHR$(7);: GOTO 55
 70  REM — BEGIN DIRECTORY
 75  PRINT
 80  : FOR I=1 to F
 85  : IF PIN(250)=155 GOTO 135
 90  : GOSUB 110: STOP
 95  : NEXT I
100  : PRINT: PRINT "DIRECTORY COMPLETE";: END
105  :
110 REM — 'VERIFY' / 'CONTINUE' COMMANDS
115  POKE 8302,86:POKE 8303,69:POKE 8304,34:POKE 8305,34:POKE 8306,13
120  POKE 8307,67:POKE 8308,79:POKE 8309,78:POKE 8310,13:POKE 8301,9
125  RETURN
130 :
135  REM — ABORT MESSAGE
140  PRINT: PRINT "DIRECTORY ABORTED!";
145  STOP
```

EOF

H8 FRONT-PANEL TIMING EFFECTS

By: *D. E. Hamilton*
1952 Baird Rd.
Penfield, NY 14526

H8 front-panel operation is supported by timing and indicator circuits that reward detailed attention.

A key circuit assures proper lighting of LED display digits, preventing overheating and burnout. This protective-blanking circuit can be outside tolerance without immediate visible symptoms, however. Fortunately, there are easy ways to check the circuit. Correction is by simple resistor substitution.

Other, harmless timing phenomena show up in the individual front-panel indicator LEDs. Exploring how these indicators reflect internal processing conditions is highly informative. When understood, the indicators can provide clues to hardware/software malfunctions and special program situations. One indicator, RUN, is mediated by a timing circuit that also can be manually adjusted.

The subtle interdependence of front panel, 8080A processor, and PAM-8 monitor also poses some requirements on H8 programming. Appendices cover program usage of the display LEDs and discourage potentially-dangerous HASL-8 practices.

INTRODUCTION

Several timing signals important to operation of your H8 computer originate on the front-panel control circuit board:

- 1000 Hz square waves provide H8's distincitve "beep".
- 500 Hz square waves trigger the regular, two-millisecond-apart monitor interrupts that PAM-8 requires for front-panel coordination and software time-keeping purposes.

These regular frequencies are divisions of the 2.048 MHz, crystal- controlled 02 bus signal originated on the CPU circuit board. The H8 will not function properly in the absence of these pre cisely-timed signals-especially 02 and its derivative 500 Hz "clock". Less evident under typical usage of an H8 are two timing signals for front-panel lighting control: protective blanking and RUN indication. The H8 can appear to operate normally even when the important protective-blanking circuit is outside of tolerance. RUN indication is less critical, although one can learn some interesting things about H8 behavior by experimenting with RUN- influencing programs.

PROTECTIVE BLANKING

Front-panel LED digits are switched by PAM-8 on every 2 ms timer click. Refresh software presents only one of the nine digits each click, taking nine clicks to cycle through a complete display. That way, a digit segment is actually illuminated less than 2 ms out of every 18. The resulting 55 Hz flickering of each digit is not discernable with normal human vision. (Change seen in display values is actually the result of sampling actual data at 32-click intervals—about 15 times a second.) LED displays require just such intermittent operation when maximum brightness is desired. Illumination at high power for longer periods damages sensitive LED elements by preventing adequate heat dissipation. Polling through the digits (usually termed "multiplexing") is also electronically simpler.

Notice that PAM-8's display-refresh cycle serves two purposes: snapshotting of desired data and assuring that each digit is actually off (blank) most of the time, keeping the display "cool".

HANG-UP PROTECTION

Unfortunately, refreshing is disrupted in many ways. In particular, the front panel display can "hang" if interrupts become permanently suppressed or if the 2 ms timer interrupt is disarmed.

Because any LED segment illuminated at the time of a hangup is damagable by the sustained current, front-panel circuits automatically blank any digit left on at the time of disruption. This is accomplished by turning off any currently-selected digit prior to the next scheduled click of the 2 ms timer. (The timer continues to click whether or not the computer is paying attention.) If all is well, PAM-8 is about to present a different digit on the new click anyhow. If something is amiss, the current digit will blank out and stay out, effectively blanking the entire front-panel display. The protective blanking circuit is, in essence, a man switch!

An interesting wrinkle in the front-panel design is routing of blanking to the MON indicator. Although MON (LED113) indication is initiated by software, it will be blanked along with any current display digit, requiring refresh to sustain the indicator. Having MON extinguish on a hangup is certainly appropriate, since software control is decidedly lost.

DUTY-CYCLE CONTROL

Although hangups are rare, automatic blanking also contributes another protective operation: display duty-cycle control.

If PAM-8 presents a digit immediately after a timer click, with blanking immediately before the next one, the maximum illumination time is 2 ms out of every 18, or 11.1%. That is easily too much! Transistor circuits for feeding lighted digit segments operate near the peak of acceptable LED current drive. It is imperative to preserving display LED operating life that such current be present in very brief pulses.

The protective blanking circuit controls duty cycle (percentage of "on" time) by confining the time before blanking to less than 2 ms. The design value is 1.5 ms or less, reducing the duty cycle to a maximum of 8.3% with the PAM-8 refresh procedure.

Another way to look at this is by saying that the protective blanker must operate at least .5 ms before the next 2 ms timer click. The sooner that blanking takes place, the lower the power stress (and the longer the life) of the LEDs. Brightness diminishes too, but not enough to seriously impair visibility.

Presence of generous blanking pulses can be determined visually once you know what to look for. Because the MON (LED113) indicator is also blanked, its brightness is governed by the protective circuit. A generous blanking pulse is present when MON lights but is slightly dimmer than either ION (LED114) or PWR (LED111). Comparison with the RUN indicator is inappropriate, for reasons presented later.

I've adjusted my blanking circuit so that the circumference of the MON indicator is visibly but just slightly darker than that of ION and PWR. The overall display is adequate although noticeably fainter than on H8's with too-short or completely- absent blanking pulses.

BLANKING CIRCUIT TROUBLESHOOTING AND ADJUSTMENT

The pulse diagram in Figure 1 illustrates the 2 ms timing signal, TICK, and its synchronized blanking signal, BLNK. BLNK, the complement of BLNK, is actually used since circuits to be triggered respond directly to complemented (low) logic levels.

TICK is developed on the front panel as part of the free-running 2 ms timer circuit. It is generated at IC109 pin 2 (IC109-2) and routed to the blanking circuit at IC107-10. BLNK is derived at IC107-5. A correctly-adjusted BLNK will be a flipped-over version of the BLNK signal shown. (BLNK is itself available at unused output pin IC107-12.) To verify generation of blanking, make the easiest tests first:

- Visually check the MON indicator after initially turning on H8 power
- If there's any doubt, check IC107-5 with a logic probe. There should be ample evidence of pulsing even though the BLNK signal is predominately high.
- Probe IC107-10 to confirm presence of TICK pulses by equally- bright high and low level indications. IC107-5 output can be adjusted by comparison with IC107-10 probing.
- For precise verification and adjustment, use a dual-trace oscill oscope triggered on rise of TICK. This is the only safe way to confirm pulse duration signals on IC107 pins 6 and 7.
- If IC107-5 is constantly low or high, check the pulse duration circuitry. Also check the traces originating at IC107 for possible bridges, especially at socket connections.

Pulse duration of BLNK is controlled by the connection of R103 and C101 to pins 6 and 7. C101 is a good quality, .15 mfd capacitor that should not be replaced unless it is inoperative. R103 is nominally a 40 kohm, 1% resistor. Blanking time is increased (BLNK decreased) by decreasing this resistance to as little as 10 k. The easiest way to check pulse duration is by temporarily placing selected resistors in parallel with R103 until de finite pulsing is produced at IC107-5. Permanently connect the resistor value that results in generous blanking without excessively dim display lighting.

If TICK, pulse duration, and power connections check out to no avail, replace the 74123 at IC107 (Heath part 443-90; Radio Shack package 276-1817).

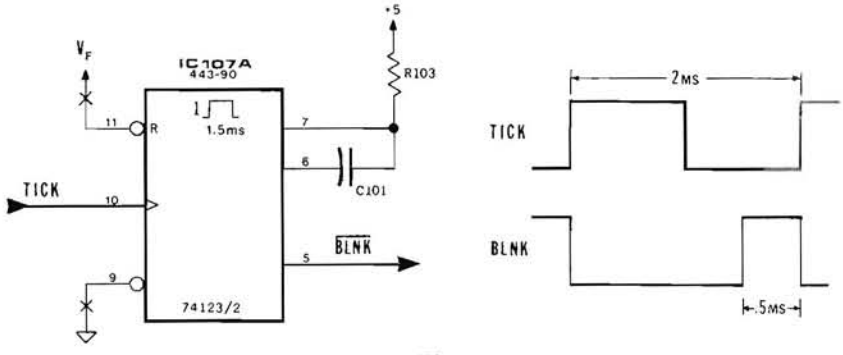


Figure 1

Vectored to Page 24

INTRODUCING THE ETA-3400 MICROPROCESSOR TRAINER ACCESSORY

Microprocessor Trainer to Personal Computer in one easy Step

Convert your ET-3400 Microprocessor Trainer into a full blown personal computer with this neat new product.

By: Lou Frenzel
Director, Educational Products

What do you do with an ET-3400 Microprocessor Trainer after you've completed the Heath Microprocessor Course? That's the question we kept getting from those student/customers who purchased the Heath Microprocessor Course. The ET-3400 6800 microprocessor-based trainer provides a means of implementing a wide variety of interfacing and programming experiments that teach microprocessor operation and application. But once the course is over, does this $190 investment become obsolete? The answer, of course, is NO. While you can always use the trainer as a miniature breadboard and development system, now you can convert it into a complete digital computer. The trainer is a minimum computer as it stands, but it is missing mass storage, higher level programming languages and an IO port to a terminal. Now all this is available in a low cost new accessory from Heath.

Introducing the ETA-3400

The ETA-3400 Accessory allows the trainer to be converted into a full blown microcomputer. Specifically, it gives you an additional 4K of random access memory, a new monitor in ROM, a tiny BASIC interpreter in ROM, an audio cassette interface, and a serial interface port to a video terminal.

Audio Cassette Interface

You get two levels of usefulness with this accessory. At the first level, the unit simply provides more random access memory and an audio cassette interface. With the trainer, you enter programs in machine code via the hexadecimal keyboard and display. This is not too difficult for short learning programs, but when larger more useful programs are developed, manually loading them over and over again is time consuming, tedious and error prone. The audio cassette interface helps to minimize this problem. Once a program is entered, it can be stored on audio cassette tapes. The built in interface allows the program to be stored and played back on any standard audio cassette player/recorder. We recommend the GE unit EC-3801. Then the next time you want the program, you load it automatically from the cassette unit.

Additional RAM

The Trainer Accessory also provides additional random access memory. The trainer comes with 256 bytes of RAM and an additional 256 bytes is added in the course. While this amount of memory is useful for short educational programs, it is not sufficient for longer more elaborate application programs. The Trainer Accessory allows you to expand the trainer memory to 4K (4096) bytes. 1K bytes of memory is supplied with the Trainer Accessory. An additional 3K bytes of memory is available at slight additional cost in the ETA-3400-1 chip set.

Terminal Interface

The second level of usefulness of the ETA-3400 is achieved when you attach the trainer and its accessory to an external terminal such as the H9 or H36. A serial IO port provided in the trainer accessory handles all of the interfacing. Actually almost any terminal can be used. Either video or hard copy types are suitable. The data transfer baud rate is fully programmable and either 20 ma or EIA interface signals can be used.

New Software

When using the accessory with a terminal, new software is required. This is furnished with the Trainer Accessory in two forms. First, there is a new monitor/debugger program provided in ROM. It has the same basic functional capabilities as the ROM monitor in the trainer itself. However, these functions are implemented through the terminal keyboard and CRT display rather than on the trainer keyboard and hex LED display. The monitor program allows the user to enter, debug and run machine language programs. Memory locations or registers can be examined or changed, programs can be executed, and break points can be initiated for program debugging.

The BASIC Language

BASIC is the most widely used higher level programming language in personal computing. It is powerful and easy to learn. With this language you can quickly write many useful applications programs.

A tiny BASIC interpreter in ROM is supplied with each ETA-3400. This is an enhanced version of the popular tiny BASIC programming language originally developed by Tom Pittman for the 6800. It is widely known throughout the personal computing field. This allows you to program the trainer and its accessory using BASIC. Provision for storing and reloading BASIC programs is also provided for through the audio cassette interface.

Availability, price and specifications are in the winter catalog.

EOF

```
.R PIP
*SY:/L
26-JAN-79
MONITR.SYS    43 24-NOV-78
TT    .SYS     2 24-NOV-78
PP    .SYS     2 24-NOV-78
PR    .SYS     2 24-NOV-78
LP    .SYS     2 24-NOV-78
BOOTUP.SYS    14 24-NOV-78
ODT   .OBJ     9 24-NOV-78
SYSMAC.SML    20 24-NOV-78
PIP   .SAV    12 24-NOV-78
EDIT  .SAV    14 24-NOV-78
LINK  .SAV    21 24-NOV-78
ASEMBL.SAV    26 24-NOV-78
CREF  .SAV     5 24-NOV-78
EXPAND.SAV    12 24-NOV-78
SRCCOM.SAV    11 24-NOV-78
DUMP  .SAV     5 24-NOV-78
LIBR  .SAV    15 24-NOV-78
PATCH .SAV     5 24-NOV-78
BASIC .SAV    36 24-NOV-78
BASIC .FIS    35 24-NOV-78
CBASIC.SAV    44 24-NOV-78
SAMPLE.MAC     2 24-NOV-78
DATE  .DAT     1 26-JAN-79
```

EOF

H-27/HT11 TEST DRIVE DEMO

```
 173000G
$DX
HT-11    H01A-3

WELCOME TO HT-11.          BOOTUP 209.00.00

HT-11 ALLOWS YOU TO "ERASE" A TYPING ERROR BY PRESSING THE DELETE OR RUBOUT
KEY.  IF YOU HAVE A CRT TERMINAL, HT-11 WILL BACKSPACE AND REMOVE THE
CHARACTER.  HOWEVER, IT NEEDS TO BE TOLD THAT YOUR TERMINAL CAN RESPOND TO
A BACKSPACE.

PRESS THE RETURN KEY IF YOU WANT   SET TTY SCOPE

TYPE ANYTHING AND PRESS RETURN FOR HARD-COPY.



HT-11 RECORDS THE CURRENT DATE WHENEVER IT CAN TO HELP YOU KEEP TRACK OF
WHEN YOU DID THINGS.  HOWEVER, YOU HAVE TO ENTER THE DATE EACH TIME YOU
BOOT-UP.  THIS PROGRAM WILL KEEP A COPY OF THE DATE, SO YOU WILL ONLY HAVE
TO VERIFY THAT THE DATE IS CORRECT, OR CHANGE WHAT HAS CHANGED.

WHAT IS THE DAY OF THE MONTH?

ENTER A NUMBER BETWEEN 1 AND 31.  BE SURE TO FOLLOW IT WITH RETURN.

?26


WHAT IS THE MONTH?

HT-11 ONLY SAVES THE FIRST THREE LETTERS OF THE MONTH.
IT EXPECTS EXACTLY THREE LETTERS.

?JAN


WHAT IS THE YEAR?

HT-11 ONLY EXPECTS THE LAST TWO DIGITS, AND
THEY SHOULD BE BETWEEN 78 AND 99.

?79


THE NEXT TIME YOU RUN THIS PROGRAM, THE DATE WILL BE READ BACK FROM
YOUR DISK AND DISPLAYED FOR YOU.  IT WILL BE SHOWN IN THE FORM:

          DD-MMM-YY

IF IT IS CORRECT, SIMPLY PRESS RETURN.  OTHERWISE, TYPE AS MUCH OF THE DATE AS
NEEDS CHANGING.  FOR PRACTICE, AND TO CHECK THE DATE YOU JUST ENTERED,
WE WILL RUN THROUGH THIS PROCEDURE NOW.


THE PREVIOUS DATE WAS   26-JAN-79

CHANGE?


IN ORDER TO PROTECT YOUR HT-11 SYSTEM DISK AS MUCH AS POSSIBLE, WE RECOMMEND
THAT YOU PUT YOUR FILES ON THE OTHER DISK.  (NOT THE ONE YOU BOOTED
FROM.)  WE WILL NOW LEAD YOU THROUGH THE PROCEDURE OF GETTING THIS OTHER DISK
READY TO RECEIVE YOUR FILES.

PRESS "RETURN" TO PROCEED.


THE LEFT HAND DISK DRIVE IS NAMED "DX0:" AND THE RIGHT HAND ONE "DX1:".
THESE ARE THE HARDWARE NAMES FOR THE DISKS.  THERE ARE ALSO SOFTWARE NAMES
FOR THE DRIVES, AND THESE WILL BE DISCUSSED MORE LATER IN THE TEXT.
THE DRIVE YOU BOOTED ON (DX0:) IS CALLED "SY:" AND IS THE SYSTEM DISK.

THE SCRATCH DISK IN HT-11 IS CALLED "DK:" AND SHOULD BE ASSIGNED TO DX1:, YOUR
SECOND DRIVE.  HOWEVER, HT-11 DOES NOT DO THIS AUTOMATICALLY FOR YOU.
THIS PROGRAM WILL MAKE THE ASSIGNMENT; ALL YOU HAVE TO DO IS TO PRESS "RETURN".

.ASSIGN DX1=DK

$DX
HT-11    H01A-3

WELCOME BACK TO HT-11.          BOOTUP 209.00.00

.SET USR NOSWAP

THE PREVIOUS DATE WAS   26-JAN-79

CHANGE?

.ASSIGN DX1=DK
```

THE 'INTELLIGENT' DISASSEMBLER

By: *Jim Warner*
10121 Hyway 55
Plymouth MA 55441

OK, so maybe we don't need another 8080 disassembler. It's true that the second issue of REMark contained a disassembler program which could serve us very well. But frankly, I've been intrigued with the concept of a disassembler almost since the day I first keyed in and successfully ran the Heath initial test routine.

I remember that event very clearly. It took place at 4:00 a.m. after working 'round the clock' completing the front panel board. What a thrill to realize for the first time that here was my very own computer — and it was actually doing something! My excitement increased as each new Heath distribution tape was loaded and run. Now my computer was not only doing something, but it was doing something useful. Of course, that realization brought a flood of questions, mostly of the "wow, how does it do that?" variety.

Not being one to leave such questions unanswered, I set about to create a disassembler, in assembly language no less. Believe me, there's no better way to learn a new computer language than to tackle such a project. After several weeks, I had my first version up and "sometimes" running. It was a modest beginning. This disassembler was only capable of absolute disassembly. There were no provisions for symbolic labels and I frequently encountered the situation where data was being misinterpreted as instructions. Wouldn't it be nice, I thought, if the disassembler could make that distinction. I mean if I was able to apply standards of reasonableness to what I watched being disassembled, why couldn't he? That way he'd know that at a particular memory location was an ASCII "1", not a load stack pointer immediate instruction.

The 8080 instruction set contains a mere 245 separate instructions. Thus, of the 256 different bit configurations possible in an 8-bit byte, only 11 combinations constitute a non-instruction. And even those non-instructions could be valid as the second or third byte of a multi-byte instruction. So, we have a situation where virtually any random memory location would generate what looked like a valid 8080 instruction. Yet we know that a good portion of a computer program is really data — tables and literals.

At last, here was a project worthy of my attention — a clearly impossible task! I would create an "intelligent" disassembler, one which was capable of distinguishing between object program logic (instructions) and object program data. It would assign default labels to program and data segments plus provide for user designated labels as well. It's output would be acceptable input to the text editor (TED-8) for potential modification as well as the assembler (HASL-8) for re-assembly.

After investing many weeks in pursuit of this glorious hunk of software, it began to appear as though the task was indeed impossible. Version after longer, more patched version crashed and proved very difficult to debug. The need for frequent revisions along with the subsequent lengthly assemblies clearly suggested flaws in my original design. And these flaws prevented me from ever actually testing my key algorithms — those dealing with symbol table manipulation and access. My dream was slipping away! What I desperately needed was a means to make program modifications quickly and painlessly. I also needed a powerful run-time package which could monitor or alter program activity on the fly. Hey, wait a minute. What I needed was the Extended Benton Harbor BASIC interpreter!

Though I remain the type of person who likes the informal title of "assembly language programmer", my BASIC interpreter will never again be thought of in terms of last resort. Rather, it will be thought of as perhaps the best means of prototyping any complex task. I, therefore, offer this BASIC program as a permanent reminder to me, and a suggestion to other readers, of the power and flexibility of this sometimes snubbed high level language. And, I might add as an example of the latest Extended BASIC with data base capabilities.

So, before I get back to HASL-8 with this BASIC prototype to use as a model in developing my faster, more efficient assembly language version, let me share with you its inner workings. I think you'll find that this is not just another disassembler.

GENERAL BACKGROUND

The "intelligent" disassembler is designed to run under Extended Benton Harbor BASIC version 10.02.01. It consists of program text and a data base. The program text occupies approximately 6500 bytes of memory, but can be reduced to less than 5000 by deleting the REMark statements, all of which are dispensible without additional modification. The data base will occupy additional memory, but its size is variable — more on that later.

After an FLOAD, the CONTINUE command will result in an identification heading followed by a "START?" prompt. This invites the disassembly start address. All disassembler addresses are in offset octal format and require a full 6 digits, including leading zeros. The same is true of the "END?" prompt which follows a valid start designation.

After the disassembly range has been established, the "MODE?" prompt is printed. There are two distinct modes of operation under this disassembler ("SYMBOLIC" and "ABSOLUTE") and these are the only valid responses.

ABSOLUTE MODE

When this mode is specified, the next prompt is "FLAG?". Any single character input at this point will be used to alert you when disassembly has or might become suspect — when instructions are encountered which may signal a data area instead. For a description of such instruction types (in this case types #3-6), see Table 1.

The final prompt will be an invitation to configure your terminal and type Return. In the absolute mode the disassembler formats the print line so as to take advantage of Heath's H9 Short Form provision. In this way a larger area of memory can be displayed at any given time.

Each disassembled line is preceeded by the octal representation of the low order byte of the current memory location. In some cases this prefix will also display your designated "flag." When page boundries change (the high order byte of the current memory location)an additional line will be generated to so indicate.

At any point during the disassembly range, a CTL-B will result in a termination of the current process and the "P=" prompt will be printed. This is an invitation to alter the disassembler's current memory pointer (again in offset octal format) so as to advance or back up the disassembly process. This can be a convenient way to investigage a subroutine following a "CALL" instruction. When you wish to resume the abandoned memory location, another CTL-B followed by the new address will accomplish it.

SYMBOLIC MODE

When this mode is specified, the next prompt following the mode designation is "VALID LO '/' HI?". We'll discuss the reason for this later. For now suffice it to say that this requires two offset octal addresses separated by a slash (/). Depending on the length of your disassembly, the next prompt could be some time is arriving. However, here's what you will see at the terminal, at varying times of course.

> PASS ONE IN PROGRESS . . .
>
> SYMBOL TABLE EVALUATION IN PROGRESS . . .
>
> SYMBOL TABLE LABEL ASSIGNMENT IN PROGRESS . . .
>
> FINAL PASS BEGINNING. CONFIGURE TERMINAL. TYPE CR:"

Since we will talk about each phase in detail, at this point let me just briefly cover the symbolic CTL-B options.

By typing CTL-B just prior to the final pass, the disassembler will print the symbol table, including addresses and labels, for you to review. A second CTL-B while such a review is in progress will terminate the printout and return you to the configure terminal prompt. This process can be repeated unendingly if you desire.

DATA BASE (DRIVING TABLES)

The intelligent disassembler data base consists of several items — minus the symbol table. It is up to each user to dimension the symbol table to whatever size memory will support. This is accomplished by using the "DIM S$(xxx)" in the command mode. Care must be taken to leave room for the growth of the symbol table entries also. Thereafter, assuming the program is saved at this point (FDUMP), the data base would consist of T$(255), T(4), N$, S$(xxx).

The string variable N$ is simply an error note. Its value can easily be changed in the command mode. To find out how it's used, you might try a "SYMBOLIC" disassembly with a start of 000256 and an end of 000264. For this experiment you may specify any "VALID LO '/' HI" addresses.

The small numeric array T is used only during symbol table evaluation. Its elements are zero at this point.

The large string array T$ represents the primary disassembler table. Unlike the other members of the data base, it is crucial to both disassembly modes.

This table enables the disassembler to produce octal values, determine instruction lengths and types and print the mnemonic operation codes. It consists of variable length entries which contain the following information:

> LEFT$(T$(x),3) = Octal equivalent of x
>
> MID$(T$(x),4,1) = 8080 instruction length
>
> MID$(T$(x),5,1) = Disassembler instruction type
>
> MID$(T$(x),6) = 8080 operation code

Thus the entry for T$(205) would appear literally as "**31531CALL**" while the entry for T$(1) would look like "**00132LXI B**."

This should be fairly self-explanatory and its need vary apparent. For a more thorough review of the primary table, you might wish to issue the following immediate command after you FLOAD:

> FOR I=0 TO 255: PRINT LEFT$(T$(I),3) " " MID$(T$(I),4,1)
>
> " " MID$(T$(I),5,1) " " MID$(T$(I),6): NEXT I

Hit the Short Form buttom then Erase before you key Return, use CTL-S & Q to govern scrolling and review the primary table 'til your hearts content.

THEORY OF SYMBOLIC OPERATION

We know that the CPU will execute instructions sequentially within a given block of memory until told to do otherwise. One way to accomplish that "otherwise" is through the use of a branch type instruction. Examples of assembly language branches would be "CALL (address)" and "JZ (address)". The equivalent BASIC statements would read "GOSUB (line #)" and "IF Z=0 GOTO (line #)".

There is one other type of instruction which affects the sequential nature of program execution. This is the instruction through which processing cannot pass. JMP, RET and PCHL fall into this category and constitute the only such instructions in the 8080 instruction set. As far as the CPU is concerned, the only way processing could proceed beyond these points is if the address was reached through a branch instruction found elsewhere in the object program.

A disassembler which could assign symbolic labels to the addresses generated by these two types of instructions would have taken a big step toward "intelligent" disassembly.

PASS ONE (SYMBOL TABLE CREATION)

As we sequentially disassemble an area of memory, let's ask our disassembler to construct a table. In its inception it will be primarily an address table, one whose entries consist of addresses. However, it will eventually become our symbol table and, since that name sounds so much more impressive (and will later be more accurate), you'll hear me refer to it more often as such.

This table will contain variable length entries of the format:

LEFT$(S$(x),1) = Evaluation flag

MID$(S$(x),2,6) = Address

MID$(S$(x),8) = Type references/Lable

We'll insert the address of all branch objects (instruction type #1) into this table. While we're at it, let's also insert the address of all load objects (instruction type #2) into our table. Although such instructions do not affect the sequential nature of program execution, they may give a hint of a potential data pocket. Finally, we will take the address following those instructions through which processing cannot pass (current location + instruction length), these are instruction type #4, and place it in our table as well. These deferred impact instructions provide an even stronger hint of potential data pockets within the disassembled program.

Incidentally, the job of building this symbol table will be made easier if the entries are always kept in ascending order with no duplication. Such is the case with my BASIC disassembler. For the technique used, I direct your attention to lines 55000-55900 of the BASIC listing. You'll notice that element zero, S$(0), is used to hold the total number of entries currently in the table. It will be incremented whenever a new address is added, whether that address is inserted within the table or appended to the end of the table (depending on its value). Existing entries are simply updated with the new reference type. In this way we can maintain a history of how many references were made to a particular entry and, more importantly, in what ways that address was used in the disassembled program.

All right, we've completed the first pass. A block of memory has been sequentially disassembled and assumed to have consisted entirely of instructions — no data. In the process we created a table of addresses. Have we done enough? Well, we've done enough if all we want is the ability to assign symbolic labels to the disassembled listing. Unfortunately, our disassembler is still "dumb" and would merrily treat data as instructions. We must now ask him to analyze those table entries and, where appropriate, re-disassemble certain object program segments. In fact, we'll find that some areas of a program must be re-disassembled (disassembled for a third time) if this guy is to acquire real intelligence.

SYMBOL TABLE EVALUATION

In the evaluation phase, we can ignore all addresses in our symbol table which are less than the disassembly start address or greater than the disassembly end address. Such entries will be considered "external" and we won't try to classify them as pointing to data or instructions. Next, we can ignore all branch object addresses. We'll assume that the author of whatever it is we are disassembling wouldn't have directed the CPU to execute a table or literal pool (in other words data). Now you can see why it's important that each entry in the symbol table also provide an indication of how that address was used in the disassembled program.

Having ignored what is probably the majority of entries in our symbol table, what do we have left? We are left to consider only those addresses which were used in a load instruction (type #2) and those addresses generated by the disassembler because processing could not pass through the previous instruction (types #4& #5). Remember also that these remaini entries will fall within the range of our disassembly, having ignored external references.

At this point, any further disassembly will be controlled by the addresses contained in the symbol table entries we have just identified. For the purpose of re-disassembly we will develop a sub-range which begins at the address contained in one symbol table entry and extends to the address contained in the next symbol table entry. We have already disassembled such a "suspect" program segment once. What we need now is a means to weigh what we'll ask our disassembler to take a second look at — re-disassemble. This is where the full compliment of instruction "types" become important. We will also provide two additional tools to use during this phase.

Our disassembler knows the start and end points of the requested disassembly. But what if that range represented only a portion of the complete program? A branch instruction with an object address outside this range would appear less valid than one whose object fell within this range. In the later, we would be inclined to consider that an instruction whereas in the former, we might see it as data. In an effort to enhance the accuracy of such evaluations, we will give the disassembler a "valid" range, which may or may not correspond to the disassembly range (the "VALID LO '/' HI?" prompt). Now the disassembler can apply this "valid" range to his evaluation of three-byte instructions (types #1, 2 = 5).

The final tool we will provide is the means to consider any suspect program segment in its entirety. We want to avoid classifying any program area as data or instructions on the basis of a single instruction. Any such decision should be cumulative. However, since each instruction must be considered individually we will offer an evaluation variable which can be dynamically altered depending upon the instruction type and the instruction's content. This evaluation variable will be subjected to factors which combine the effect of instruction type and content.

The factors, or weights, I've chosen can be found on line 30100 of the BASIC listing. They are actually applied at lines **5700-57900**. These factors are somewhat arbitrary and you readers certainly have my permission to alter them as you see fit — we call that "tuning". However, they seem to do the job for me. Here's how they work.

If a branch instruction (type #1) is encountered in this re-disassembly and, if the object address falls within the "valid" range, the disassembler will add a positive bias to the evaluation variable. If the object address is outside the "valid" range, a somewhat weaker negative bias is applied to the evaluation variable. The same approach is used with load instructions (type # 2) but the effect on the evaluation variable is not as great. If a suspect instruction (type #6) is found, a strong negative bias is applied to the evaluation variable. Finally, if a non-instruction (remember, we're still disassembling sequentially at a location dependent upon the previous instruction's length) is encountered, type #3, the strongest negative bias is applied to the evaluation variable.

Once we complete this re-disassembly, if we find that the evaluation variable yields a positive result we can flag that symbol table entry as pointing to program logic. In reality, all symbol table entries are so flagged upon entry — innocent until proven guilty, as it were. If our evaluation variable remained zero, it means we don't have enough information to make a definite determination. We'll flag that entry as "unknown" and later, through convention, treat the program segment as instructions. However, if the evaluation variable is negative (a likely result if a non-instruction was encountered, a probable result if any "bad" branch or load instructions were found), we'll flag that symbol table entry as data. We should also ask our disassembler to find his second wind because there's more work to do with any such program segment (remember re-re-disassembly?).

There are a couple of reasons for taking what amounts to a third look at any "confirmed" data area. First, a single byte of data may have been interpreted as a multi-byte instruction during pass one. In such a case, we may run the risk of losing the beginning of an otherwise valid instruction through the interpretation of data bytes 2-3 as an address or value. This very situation was clearly illustrated in the earlier REMark disassembler at address 000033.

The second reason for re-examining this data area lies in the way our symbol table was constructed. The next entry in the table (the entry containing the ending address of this sub-range) got there for a reason. If that reason was because of JMP instruction (type #5) and, if that instruction now generates a "valid" object address, then we owe our symbol table one more entry — one corresponding to the beginning of that instruction. Such an approach allows us to properly evaluate and label the JMP instruction we find at address 000050 following the data at address 000043. Our only risk in "forcing" entries in this way is a few extraneous labels during the final pass, with no harm done. I prefer that to too few labels.

With the evaluation phase we find the last maintainence being applied to our symbol table. We won't be adding any more entries. However, this entire process will be repeated with each non-branch table entry throughout the range of disassembly.

SYMBOL TABLE LABEL ASSIGNMENT

The time has come (finally?) to transform our lowly address table into a true symbol table. The entries are now flagged as program logic, unknown and data. These flags will control the disassembly process (how memory is looked at) during the final pass. But what about us, the user, and the effect symbolic label content has on readability? Wouldn't it be nice if the labels we develop also reflected the distinction of program logic, etc.? Well, why not?

Let's call program logic entries "RTN" (for routine), un known segments "UNK" and data areas "DAT". We'll call all external labels "EXT" and we will append to each label an in dication of where it resides within the table. Thus, our first few entries might appear as "EXT.1", "RTN.2", "UNK.3" and "DAT.4". In this way, using the CTL-B and CTL-C provisions, we could return to command mode during symbol table review and issue the following immediate command:

```
S$(2)=LEFT$(S$(2),7)+"ANYVAL"
```

Now we have replaced "RTN.2" with "ANYVAL" while retaining the evaluation flag and address. Thereafter, the dis assembler would associate "ANYVAL" with value in the symbol table whenever it was used during the final pass. Thus we have a simple, effective means for user designated labels.

This technique can be used over a period of time, as program segment functions are identified by you, to make an extremely readable disassembly listing. If at any time you wished to terminate a final pass listing, and resume at the start of the final pass later, the following immediate mode commands would accomplish it: —

```
UNLOCK

1 CNTRL 0,65000: GOTO 34000

FDUMP "any name"
```

THE FINAL PASS

With labels assigned we can now begin our final pass, producing a symbolic assembly listing for our disassembly. Actually, this pass is not unlike the very first pass except we will be printing what we find as we PEEK( ) at various memory locations. Additionally, we'll use labels instead of octal values for three- byte instruction operands and we'll sometimes preface a line with a label — when that label's address matches our current memory location. Of course, if at that point the symbol table indicates we've reached a data area. We will force the "DB" (define byte) interpretation on what we find.

There you have it, symbolic (intelligent) disassembly. Simple, wasn't it?

PROGRAM VARIABLES

A	utility address, generally = I'
A$	utility address in offset octal, generally = I'
C$	mode identifier control byte/CTL-B enable
E	disassembly end
E$	disassembly end, offset octal
F'$	absolute flag/CTL-B enable
I,I1	utility array reference indicies
I2,I3	symbolic re-disassembly symbol table reference indicies
I$	utility string
L	instruction length, from T$(I)
L$	symbolic line prefix label
L1$	symbolic instruction operand label
L9$	symbolic end statement start label
N$	symbolic intregrity violation message
O$	instruction mnemonic op code, from T$(I)
P	current memory pointer
P$	utility offset octal address, generally user input
Q$	utility octal value, from T$(I)
Q1$	symbolic two-byte instruction operand
Q2$	multi-byte instr's byte 2 octal value, from T$(I)
Q3$	multi- Q3$ multi-byte instr's byte 3 octal value, from T$(I)
S	disassembly start
S$	disassembly start, offset octal
S$(I)	symbol table
T,T$	instruction type, from T$(I)
T1	symbolic evaluation variable
T(I)	symbolic symbol table type?references decode array
T$(I)	primary disassembler driving table
WO-W2	symbolic evaluation weights
S1,S1$ E1,E1$	symbolic sub-range re-disassembly start/end addresses

Jim's program will be available in the next release of HUG Software which is expected to be available late next month.
:JB:

EOF

!ERROR SYNTAX

Issue #4 Page 8 — Space Wars Patch

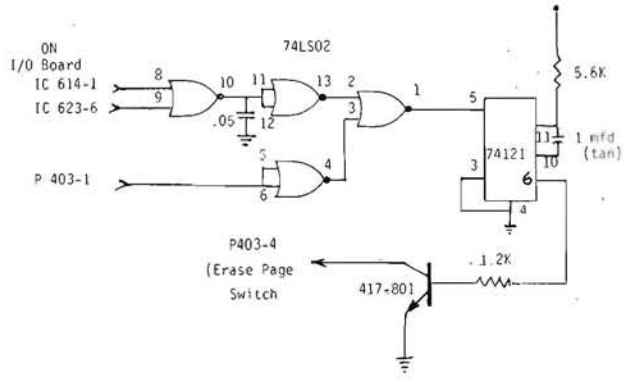
Line 3270 should be: T8=CIN(0):IF T8<0 GOTO 3260
Line 3285 should be: A$=LEFT$(A$,LEN(A$)-1)

Issue #4 Page 15 — Under Factorial Function

DEF FN F(N)<>GARBAGE
DEF FN (N)=INT(SQR(P2*N)*EXP(N*LOG(N)−1)+1/(12*N)))

Issue #4 Page 27 — Wrong Schematic!

Here's the right one — (Note: the schematic in issue #4 is supposed to make lower case when the switch is open, but hasn't been tested here — lower case would not be displayed on the H9 CRT, of course, but would be useful in the Editor.)



Issue #4 Page 16:

Line 150 should read IF L>L1 THEN . . .

EOF

```
**********************************************************************************

                    HELLO THERE ! I AM THE

        HHH          HHH            11           4444
        HHH          HHH           111          44444
        HHH          HHH          1111         444444
        HHHHHHHHHHHHHHH            111        444 444
        HHHHHHHHHHHHHHH            111       444  444
        HHH          HHH           111      444444444
        HHH          HHH           111     4444444444
        HHH          HHH           111           444
        HHH          HHH           111           444

 WITH FEATURES LIKE SOFTWARE SELECTABLE LINE WIDTH AND PAGE SIZE FOR

                132 CHARACTERS PER LINE
                     96 CHARACTERS PER LINE
                          80 CHARACTERS PER LINE

                WITH BOTH UPPER AND lower case

 666                THE H 14 PRINTER IS IDEAL FOR  PERSONAL AND SMALL
6      LINES        BUSINESS COMPUTER SYSTEMS. WHERE HIGH PERFORMANCE AND
6666   PER          LOW COST IS REQUIRED. RELIABLITY IS ASSURED
6   6  VERTICAL     THROUGH TOTAL HEATH DESIGN
 666   INCH         AND A PRINT HEAD BY PRACTICAL AUTOMATION.


 888                THIS PRODUCT AND SERVICE SUPPORT  IS AVAILABLE
8   8  LINES        AT ANY OF THE 53 HEATHKIT ELECTRONIC CENTERS IN
 888   PER          THE U.S., DIRECTLY FROM THE HEATH COMPANY
8   8  VERTICAL     IN BENTON HARBOR MICHIGAN, AS WELL AS HEATH MARKETING
 888   INCH.        OUTLETS AROUND THE WORLD.

 AND I CAN BE YOURS IF YOU SUBMIT THE BEST PROGRAM
 TO CONTEST # 5 !! MY OUTPUT IS STANDARD EIA/20 MA
 WITH HARDWARE HANDSHAKING ABOVE 300 BAUD. SUBMIT
 YOUR PROGRAM TO HUG BEFORE APRIL 9, 1979. GOOD LUCK!!

**********************************************************************************
```

--EDIT

PAUSE SWITCH MOD FOR ECP-3801

by: Jim Meyers

If you have electrical expertise, and are not afraid of violating your recorder's warranty, perform this simple mod to eliminate your having to "pull the plug" everytime YOU want to operate the transport.

Locate the area where the red, pink and violet wires are soldered to the board inside the recorder. Remove the red and pink wires from the board and solder them across the remote jack. Jumper the pads together where the red and pink wires were. Cut and tie back the violet wire. Record these changes in your manual so you can "restore" your recorder to normal in the future.

When you have connected the recorder to the H8 again you will find that by sliding the switch to PLAY you can operate the transport without disconnecting the remote plug. Slide the switch back to PAUSE to give the H8 control.

REVERSING MEDIA ON SINGLE HEAD FLEXIBLE DISK DRIVE

Flexible Disk Drives (Floppies) offer the end user low cost random access to data records. Prior to the introduction of the floppy, the only other alternatives were sequential tape cassettes, low cost one-half inch tape, or single cartridge hard media disk drives. The floppy disk has been an ideal peripheral for low cost CPU's (Mini's and microprocessors) and low cost systems. It is a natural solution for storage in the lower end of the computer market.

There has been a tendency by some end users, to economize by attempting to use the media on both sides in a single head disk drive. We must not lose sight of the fact that the value of the data stored on diskettes exceeds the cost of the media by a wide margin. Loss of data on either read or write means time delays, reconstruction of lost data, and customer dissatisfaction with the system, drive and/or media manufacturer. All of this can be avoided in advance if the end user is made aware of the whys and why nots.

HEAD SHOE AND PAD OPERATION

The relationship of the head to the media is such that when the jacket is properly inserted, and all interlocks are satisfied, the head is loaded on to the media on the recording side, and a felt loading pad is applied to the non-recorded side. In normal operation, a gradual build-up of oxide will accumulate on the pressure pad. There might even be some wear on the non-recorded side due to a scouring action of the oxide impregnated pad.

If the media is reversed, the scouring action will now occur on the prime recorded side, and the previously scoured side is now presented for recording. The recorded data is now subjected to an abrasive wearing by the contaminated load pad. Since this data is not being read, there is not any means of detecting the amount of wear or the loss of data. While a catastrophic failure might not occur, it is possible that some drop-out or other read error might go undetected. Worse yet, is the possibility that the error condition might be intermittent, which makes the entire operating system suspect. Another adverse effect of reversing the media, is caused by reversing the direction of rotation of the media against the pressure pad. This reversal of direction is apt to "break off" any build-up of oxide particles. This presents a potential loose contaminent situation.

The net effect of this reversing (or flipping) action over a period of time is to reduce performance and increase the probability of drop outs and errors.

DISKETTE TENSIONING

On most Floppy Disc Drives, when the diskette is properly inserted and operation has begun, pressure is applied to the jacket on both sides so that proper tension is created on the flexible media prior to the recording head. This also provides a wiping action of the liner material against the flexible media. When the jacket is reversed (or flipped), the direction of rotation is reversed, breaking loose any extraneous particles built-up by prior wiping. Thus, reversing the media increases the probability of extraneous contamination and again increases the possibility of errors.

TWO HEAD DRIVES

The above problem areas do not occur on two head drives that are designed for two sided applications. On a two head drive, the pressure pad has been replaced by a second head mounted in a ceramic shoe. The operation now consists of a head-media-head relationship. The soft pressure pad with possible oxide build-up has been eliminated.

The diskette tensioning apparatus is the same on one and two head drives. Since media spin direction is not reversed by flipping, the oxide break-off problem does not occur.

SUMMARY

The foregoing summarizes the reasoning why Dysan and major OEM suppliers of diskette drives do not recommend two sided media for one head drive operation. Dysan feels that the potential operating problems would make an unwarranted reflection on our reputation by using media in an unsuitable fashion. When IBM introduced the 3740 diskette, they intentionally interlocked reversal possibilities by off setting the index hole from the centerline. IBM does not make a reversable diskette.

Dysan does test and supply two sided media for operation in two head (two sides) disc drives.

This info provided by Dysan. :JB: EOF

CASSETTE INTERFACE BECOMES "AT:" TWO PORTS FOR THE PRICE OF ONE

So now you have your disc system for your H8, 32K of memory, and an H8-5 serial card. Most likely your probably screaming — "I 've also got a parallel board and I want to add a hard copy device, but there ain't no $$%&! room on the motherboard for another serial board. "Well, never fear, the following article will describe how to add another serial port to your H8-5 interface. You're probably saying, "I left my black rod with the rusty star on one end back in the pirate's maze, so it can't be magic". You're right, it's not magic. It's going to cost you the cassette interface capability, but so what, you have all your cassette programs transferred to disc and you don't use the darn thing anyway.

Ok, enough ratchet jawin'. You will need a good set of handtools including a sharp exacto-kni fe and a 9/64" drill bit and hand drill. You will also need about a half dozen new components which you can purchase from Heath or your nearby parts supply house. The following is a list of parts you will need to complete this project:

1. 2 — 14 pin IC sockets (Heath P/N 434-298)
2. 1 — 75188 IC (Heath P/N 443-794)
3. 1 — 75189 IC (Heath P/N 443-795)
4. 1 — +12V three terminal regulator (Heath P/N 442-663)
5. 1 — −12V three terminal regulator (Heath P/N 443-664)
6. 4 — 2.2 mfd tantalum capacitors (Heath P/N 25-221)

The following is a pseudo step-by-step procedure for making the conversion on your H8-5 card. I would suggest reading the entire requence over carefully before proceeding. Remember, your cassette interface will be totally destroyed once the conversion is made.

1. The first thing you need to do is remove all the IC's associated with the cassette interface. These IC's are the following, IC104, IC105, IC106, IC107 and IC111.

2. Next remove the following components and jumpers: R124, R125, Q106, Q107 and the two L-H jumpers at IC107.

3. Carefully remove the two sockets at IC locations 107 and 111. These two locations will later contain the two RS-232 level translator IC's.

4. Now is the time to pull out your hand drill (electric or otherwise) and 9/64" drill bit. Locate two locations on the board where the two regulators will comfortably fit. These two regulators can be mounted off the board on two 6-32×3/16" spacers. The two most logical locations that I found are as follows: By pin 1 of IC113, and about 1" right of pin 9 of IC118. Make absolutely sure that when you drill the holes, that you do not cut any foil. Also make sure that the hole is centered such that when a 6-32 nut is installed, it will not come in contact with any foil when tightened down.

5. The following foil cuts must be made in order to isolate the IC pads for reconnection.

   A. Isolate pin 17 of IC123 from any other foil.
   B. Isolate pin 22 of IC123 from any other foil.
   C. Isolate pin 25 of IC123 from any other foil.
   D. Isolate pin 3 of IC123 from any other foil.
   E. Isolate all of the pads to the two IC's just removed (IC107 and IC111).

6. Make the following connections on either side of the board (which ever is convenient), using a plastic or teflon covered small gauge stranded wire.

   A. Connect pin 17 to pin 4 on IC123.
   B. Connect pin 25 to pin 9 on IC123.
   C. Move the Y-O jumper to Y-4 (alternate terminal port
   D. Place a jumper at interrupt select 'R×R' (IC128 pin 3)
   E. Place a jumper from 'C' to 'I3'.
   F. Install a jumper to turn interrupts off at IC128C,D

7. Install two 14 pin IC sockets at locations 107 and 111. Make sure the pin 6 and 7 pads are not shorted together at either location.

8. Now mount the two voltage regulators on the component side of the board at the two previously drilled locations using 6-32 hardware. Use a 3/16" spacer between the regulator and board to elevate the regulator off the board. It is not important at which location either regulator is mounted.

9. Place a jumper from the 'tape TX Speed' to the proper baud rate hole for your printer. The 'Tape TX Speed' is now the trans mit and receive baud rate for the alternate terminal port.

10. Up to this point, I have described in detail the modifications necessary to your H8-5 interface board. From this point on, wire the rest of your board as per the schematic shown, using plastic or teflon covered small gauge stranded wire.

See Schematic on Page 20

11. Since all that is necessary for RS-232 communication is three wires, it is very convenient to use three pins on P102 for bringing out the proper signals to the back panel connector. The pins I chose were pins 1, 2 and 6 for a signal ground. Pins 1 and 2 of P102 have been isolated from any other circuitry by the removal of Q106 and Q107.

Jim "Buzzweed" Buszkiewicz

Buzzweed has another neat idea on Page 21

EOF

H10 MODIFICATIONS — IMPROVING ADJUSTMENT ACCURACIES AND LENGTHENING THE CABLE

By: Carroll Hennick

Although often maligned, the H10 is well designed, rugged and represents a true bargain, considering its quite reasonable price. My only problems had been with getting stable alignment of punch pitch (50 bytes/5 inches, exactly) and proper timing adjustment of the reader sprocket wheel. These problems have been solved.

The layout of the room where my H11 system is does not provide for placing the H10 and H11 cabinets side-by-side, as is demanded by the short cable provided with the H10 kit. (Adding the disk unit and printer soon forthcoming from Heath aggravates this problem further.) In order to place them six feet apart it was necessary to use a ten-foot cable. The circuits which drive the lines in this cable are not designed to operate with the extra load presented by this long cable, and so a few modifications are required. They are quite simple, inexpensive and require no cutting or removal of wires or circuit board lands.

The techniques for improving alignment, and for permitting a long connecting cable are given below.

The reader alignment problem had a strange history. The kit assembly manual procedure for alignment was followed initially. It merely says: Adjust the sprocket wheel so that the tape holes are lined up with the reader assembly. So I centered the holes over the light channels. This worked flawlessly for reading ABS LDR, ED-11, FOCAL, etc for several months, but evidenced considerable difficulty with reading my FOCAL programs which I had punched. Examining the waveforms of READER READY and the outgoing data lines on P3 revealed that the data transition time was marginally close to the end of the 16.5 ms period. A readjustment (whose technique is given below) which decreased the average data transition time to 9-11 ms after the beginning of the 16.5 ms pulse cured this completely.

The parts required for the long cable modification should cost less than ten dollars, and are commonly available. (They may all be available from Heath, although I have not searched them out.)

IMPROVING THE READ ADJUSTMENT

I found three mechanical modifications necessary or desirable to enable the H10 to successfully read fanfold tape:

* Increase the tension of the Reader Guide Springs, by shortening their length about 25%. Else the tape creases occasionally cause a timing jitter of data sensed, garbling a byte.

* Where the tape passes through the front panel, if necessary, file the top of the slot higher. Else the tape creases occasionally catch there, crumpling the tape beyond use.

* Cement a board of metal or plastic to the Reader Trough bottom, so that it protrudes three inches. Else the supply tape falls off the trough.

The following method of adjusting the reader sprocket wheel seems necessary in cases where the data is read with wide variances in timing between bytes. (See the discussion of FOCAL above.)

* Modify the instruction given in the H10 Operation Manual, page 18, from "Line up the tape holes with the holes in the reader assembly" to the following:

"Line up the rear part of the tape holes with the side of the Reader Circuit Board which is nearer the sprocket wheel."

* It is better to err on the side of having the tape too far forward. If an oscilloscope is available, adjust for data transfer time (on P3 pins) about 10 ms after the rise of READER READY (P2-3).

IMPROVING THE PUNCH ADJUSTMENT

Quite contrary to the advice given in the H10 Operation Manual, page 16, the punch ramp must press quite firmly against the sprocket wheel to ensure a stable pitch adjustment. Else, no matter how carefully the sprocket wheel is adjusted, minor tensions on the punch ramp (such as a torn tape) can grossly alter the pitch (nominally 10 bytes/inch). Since most tape readers (including the H10) are quite sensitive to pitch, this is very important. I have installed a hefty spring between the chassis and the mid-point of the punch ramp, which applies nearly one pound of pressure against the wheel. Rather than causing problems , it has removed what was a most vexing problem. Also, it is necessary to adjust the sprocket wheel set screw very tightly, using a screwdriver held by vise-grip pliers.

IMPROVING THE BEHAVIOR OF PUNCHED FANFOLD TAPE

When first punched, fanfold paper tape creases are stiff, and do not stack properly in the fanfold tray. Pinch the creases as they emerge from the H10 cabinet.

LONG CABLE MODIFICATION

Parts

1	25-conductor cable (like the 347-65 provided with the H10 kit)
2	2-conductor cable of the same type (These three cables should be as long as needed, up to about 15ft)
10	100 pF mica capacitors
1	330 ohm 1/2 watt resistor
1	470 ohm 1/4 watt resistor
2	connector shell (Heath 432-704) and pins (Shells and pins are left over from the H11-2 kit.)

Procedure

1. Remove the H10 top cover and right side panel.
2. Add 8 capacitors which connect between the incoming data lines and ground. These are best attached to the lugs of SW4 on the component side of the circuit board (SW4-25, 22, 13, 10, 7, 28, 4, 1). Use the illustration given in Pictorial 1-12 of the Assembly Manual Illustration Booklet, and the Circuit Diagram. There are convenient ground points in the vicinity

H10 REHABILATATION CONT'D

of Q8 from which to run a short insulated wire to the capacitors.

3. Add a capacitor on the non-component side of the board from P2-11 to ground, and from AM (IC7B-4) to ground. (Ground is available at IC8-7.)

This completes the modifications for the PUNCH circuits. The 10 shunt capacitors decrease the succeptability of these data and control lines to crosstalk pickup.

4. Swap H10 IC16 (7474) with H11-2 IC30 (74LS74). This gives the Parallel I/O Board greater drive power for the READER START line.
5. Add a 470 ohm, 1/4 watt resistor on the component side of the board from P2-2 to P2-4. This increases the pull-up current on the READER START line.
6. Add a 330 ohm 1/2 watt resistor on the non-component side of the board from P2-3 to C7's +9V terminal. Slide insulating tubing on the full length of the leads. This increases the pull-up current on the READER READY line.

This completes the modifications for the READ circuits.

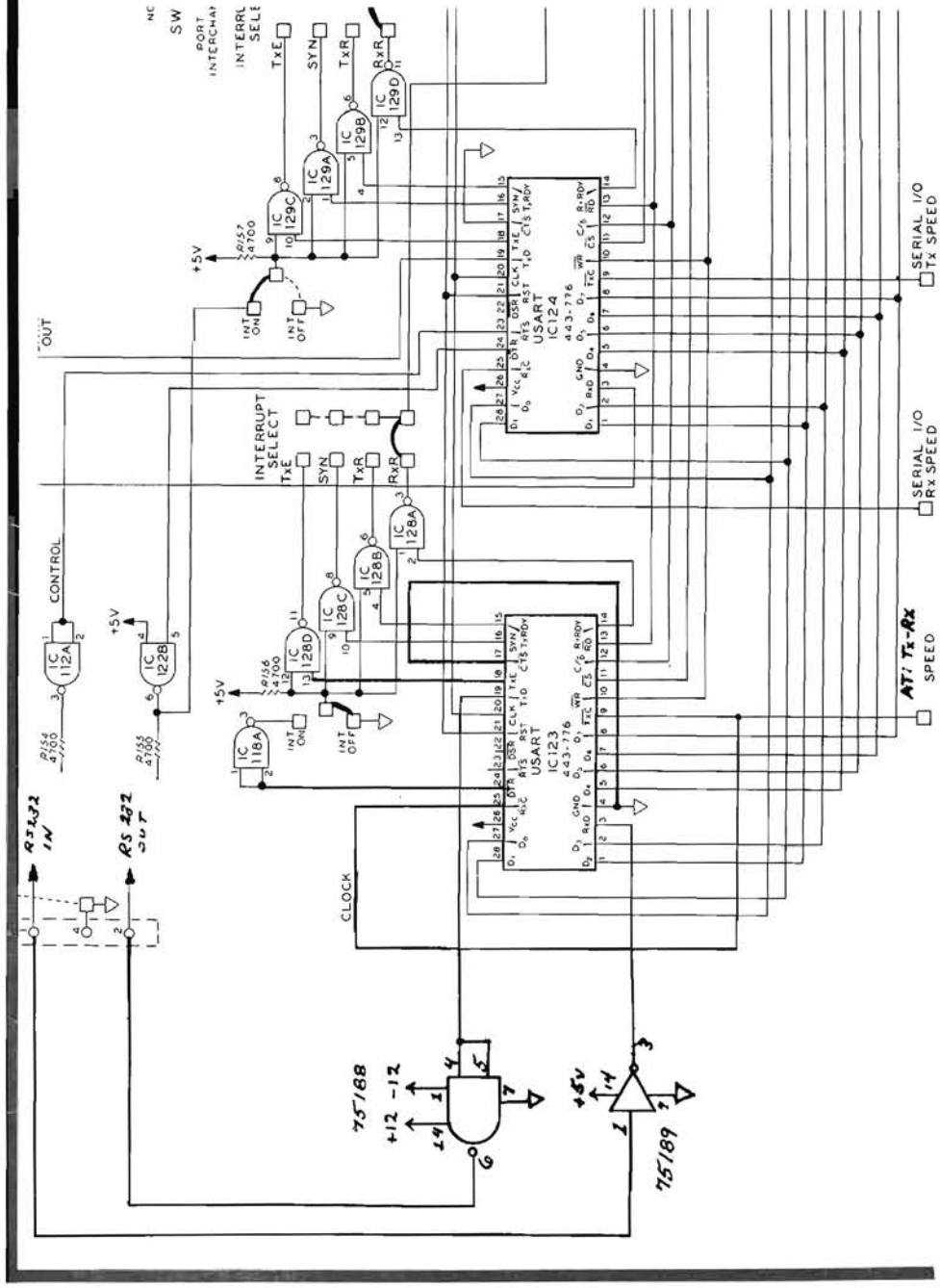
7. Cut the 25-conductor and the two 2-conductor cables to the desired length. (Approximately 15 ft max.)
8. Prepare the cables according to the instructions given in the H10 Assembly Manual, pages 77-79, with these differences:

— Use wire colors as available, marking the colors used in the Manual. — The four control lines are the small cable wires, as shown in the box:

— In the large cable, connect six wires (instead of two) to pin 24 (ground). Use a short bare wire from the pin to attach the wires to.

9. Tape the three cables together at intervals of two feet or more. It is not desirable to keep the small cables near the large one or to each other very much.
10. Reassemble the H10. *EOF*

TWO PORTS FOR THE PRICE OF ONE



	H10 pin	H11 pin		
Cable A	10	21	READER START	SEND DATA L
	20	9	PUNCH START	TAKE DATA L
Cable B	9	20	READER READY	DATA SENT L
	21	10	PUNCH READY	DATA TAKEN L

H8-2&4/5--OR HOW TO USE YOUR H8 AND H9 VIA H8-2.

By: Jim Buszkiewicz

I can just hear it now — "Wow!!, What in the world took you so long. This is just what I've been looking for. You must have been lost in the pirate's maze too long to think that I would butcher my cassette interface just to get an extra serial port." Well, for you die hard cassette jocks, here it is.

Basically the H8-2 is a parallel-to-serial-to-parallel converter. Notice I used the word 'serial'. If you eliminate the last serial to parallel UART, you are basically left with an identical serial port as found on the H8-5 interface card. By making a few foil cuts, adding a couple of jumper wires, and adding four resistors, you can convert one of the three parallel ports into a serial port, which will communicate with the H9 via TTL levels. The schematic shows the modifications needed to convert channel '0' into a serial port. The conversion, however, could be done to any of the three channels. TTL levels were chosen as a means of communication because in order to use RS-232, more components would need to be added to the parallel interface board and there isn't any more room on the board for additional components, other than a few resistors or such.

Many of you have asked about interfacing the H8-2 with the H9 PARALLEL BOARD.

. . . Can't be done, sorry. :JB:

Since the H8-2 card normally uses the 'phase 2 not' for it's baud rate clock, dividing it down to the proper frequency would again require additional components. What I did to obtain a baud clock for the serial port, is to steal the baud rate clock pulses from the H9. All that is needed is one additional wire, run between the H9 and H8-2. This also allows for total baud rate control from the H9 terminal which means you can list a program at 9600 baud, and if you see something you like, just hit the baud rate switch on the keyboard, and presto!! You're crawling at a readable 110 baud!!! OK, enough sales pitch, let's start converting.

1. The first step I took, was to convert the I/O card in the H9 to TTL input-output levels. This is very simple to do, and fully explained in the H9 operation manual on page 14 and 15 (TTL Serial Input/Output). I then proceeded to check the operation of the card by shorting pins 2 and 10 of connector P-603 to see if the H9 could 'talk' to itself. This test is very similar to the one performed in the second column on page 135 of the assembly manual. The input and output pins on the backpanel connector are now pins 3 and 2 respectively.

The following is a step by step procedure in making the necessary modifications to your H8-2 card. Follow each instruction very carefully. Some of the foil cuts and soldered jumper wires are in very close quarters, so take your time, the results are worth it!!

1. Carefully remove the UART (IC104), and place it in some of the conductive foam that you received with your H8 system. This UART can be used as a backup in case the one in your H9 should ever go sour.

2. Temporarily remove the USART (IC103), and likewise place it in some of the conductive foam. This was done mainly as a visual aid.

3. Solder a jumper wire from E1 to E2 on the top side of the board if one is not already there.

4. Isolate pin 8 on the output connector (P101).

5. Solder a jumper wire from pin 3 of IC101 (carefully) to pin 3 on the USART (IC103). (All jumper wires are soldered to the foil side of the board.)

6. Solder a jumper wire from pin 2 of IC101 to the output connector P101 pin 8.

7. Isolate pins 9 and 25 of the USART (IC103) from any other circuitry. This can be done by cutting the foil, on the back side of the board, to pin 17 of the UART (IC104). NOTE: pins 9 and 25 of the USART (IC103) should remain shorted together.

8. Solder a jumper wire from pin 17 of the USART (IC103) to pin 4 of the same IC.

9. Solder a jumper wire from pin 19 of the USART (IC103) to pin 12 of IC101.

10. Solder a 1000 Ω 1/4 watt resistor from pin 11 of IC101 to pin 14 of the same IC.

11. Solder a 1000 Ω 1/4 watt resistor from pin 2 of IC101 to pin 14 of the same IC.

12. Isolate pin 22 of the USART (IC103) by cutting the foil on the top side of the board between that pin, and pin 1 on IC105. This foil cut must be done very carefully because of the close proximity of other foils.

13. Solder a 4700 Ω 1/4 watt resistor between pin 22 and pin 26 of the USART (IC103).

14. Isolate pin 6 of the output connector P101 by cutting the foil between that pin and pin 8 of IC101.

15. Solder a jumper wire from pin 9 of IC101 to pin 6 of the output connector P101.

16. Solder a jumper wire from pin 8 of IC101 to pin 9 (9 and 25) of the USART (IC103).

17. Solder a 1000 Ω 1/4 watt resistor between pin 9 of IC101 and pin 14 of the same IC. NOTE: If pin 14 on that IC is getting too crowded, you can use any other +5 volt foil which is close by.

18. In order to work properly with the H9, the interrupt for the port you are using must be enabled. To do this, solder a jumper on the top side of the board from H1 to H2. Also solder a jumper on the top side of the board from ICHX (X=port number you are using) to I3.

Vectored from page 21

H8-2 MOD — MAKE IT TTL SERIAL :JB:

19. Now program the port you are using for the main console device port number by inserting jumpers to select port 372. Remember, the serial I/O port on your H8-5 card is also set to this port, so it will have to be changed, disabled, or the card temporarily removed (two or more ports cannot have the same address number).

20. Carefully replace the USART (IC103) back into it's socket.

This completes the necessary modifications to the H8-2 interface card. All that is needed is to make up a cable to go between this card and the H9. The following is a helpful wiring diagram in making up that cable.

H9 Backpanel Output Connector Description		P101 on H8-2 Board
Pin 2 TTL Serial Data Output	-Connects to-	TTL Serial Input Pin 8
Pin 3 TTL Serial Data Input	-Connects to-	TTL Serial Out Pin 7
Pin 7 TTL Baud Rate Out	-Connects to-	TTL Baud Clock in Pin 6
Pin 9 Ground	-Connects to-	Ground Pin 9

Because of the fact TTL levels are used for data communication, the length of cable used to connect the H9 and H8-2 should be kept to a minimum. The length of cable used that I used was 7 meters with no adverse affects.

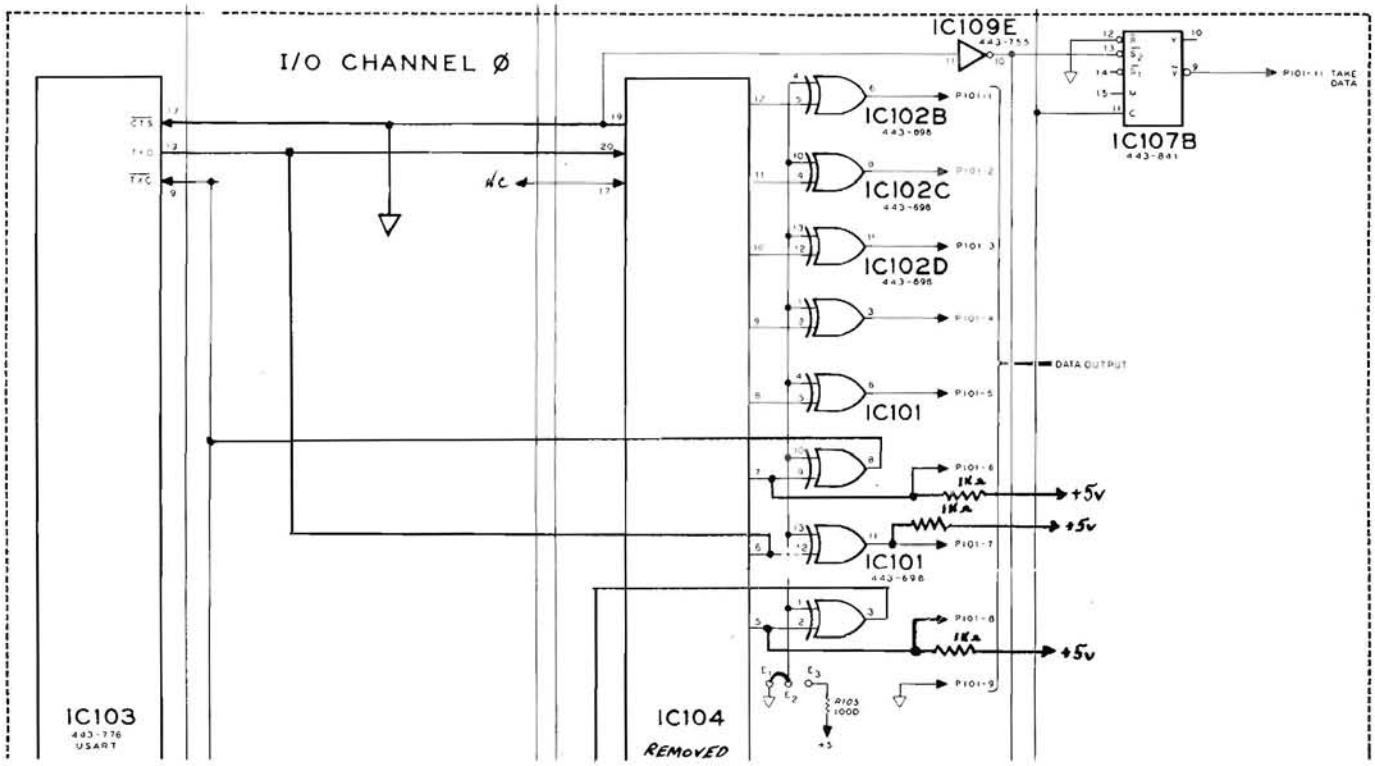
WORKSHOPS:

I am writing to ask your Journal to make an announcement of our seminar program in your January 1979 issue. Dr. Peter Rony, Dr. Paul Field, Dr. Chris Titus and I are directing these workshops.

A new and expanded series of Four 3 day hands on workshops on 8080/8085 Design, Microcomputer Interfacing, Software Design and Digital Electronics are being given by the authors of the popular Bugbooks. Participants have the option of retaining equipment used in these courses. Dates are March 19 to 28, 1979. For more information, contact Dr. Linda Leffel, CEC, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061 (703-961-5241).

This effort on your part to bring these programs to the attention of your readers is greatly appreciated by the Virginia Polytechnic Institute and State University Extension Division and the course directors.

David Larsen



EOF

!02 ERROR — FATAL SYSTEM ERROR

HDOS is very protective and we've all been reminded by various error messages. These unhappy events can be more tolerable by re-writing the messages as illustrated here by one of our members.

Call ERRORMSG.SYS into the EDITOR and modify to suit yourself — NEWOUT under any file name — DELETE ERRORMSG.SYS and finally — RENAME your new file to ERRORMSG.SYS — Have fun.

128 YOU MEAN YOU'VE ACTUALLY FOUND THE "CTRL" AND "C" KEYS?
129 CTL-B STRUCK — LOOKS LIKE YOU DO HAVE MORE THAN ONE FINGER!
130 DATA EXHAUSTED — YOU SURE YOU KNOW HOW TO PROGRAM??
131 ATTEMPTED DIVIDE BY ZERO — I'M A COMPUTER STUPID! — NOT GOD
132 YOU CALL THAT A NUMBER!!!
133 ILLEGAL USAGE — YOU SHOULD PLAY WITH TINKER TOYS NOT COMPUTERS
134 YOUR DATA IS LOCKED DUMMY
135 YOU MAY SAY THERE'S VARIABLE IN THE NEXT LINE, BUT I CAN'T FIND IT!!
136 IF YOU CAN'T COUNT THAT HIGH DUMMY, HOW DO YOU EXPECT ME TO?
137 HOW DO YOU EXPECT ME TO RETURN FROM A GOSUB THAT AIN'T THERE?
138 CAN YOU READ 256 CHARACTERS IN ONE BREATH? — I SURE CAN'T!!
139 THERE AIN'T NO SUCH LINE NUMBER, SLICK!!
140 BEING ABLE TO SPELL AND READ HELPS!!
141 TYPE CONFLICT (THERE IS A DIFFERENCE BETWEEN LETTERS AND NUMBERS DUMMY)
142 BUY SOME MEMORY CHEAPSKATE
143 SUBSCRIPT OUT OF RANGE — RECAPITULATE — SLICK!!
144 WRONG NUMBER OF SCRIPTS — HEATH CO. DOES SELL A BASIC PROGRAMMING COURSE
145 WHY DON'T DIMENSION THIS ARRAY FIRST BEFORE RUNNING THIS PROGRAM!
146 WHERE DID YOU LEARN TO SPELL?
147 WHAT?????
148 I'M GLAD THIS INFANTILE PROGRAM IS OVER
149 I'M GLAD THIS STUPID PROGRAM IS OVER
150 HOW MANY OF THE SAME FILES DO YOU WANT DUMMY??
151 YOU CAN'T CALL IT THAT STUPID!!
152 TOO MANY OR TOO FEW ARGUMENTS SPECIFIED (TRY AGAIN STUPID)
153 WOULD YOU WALK THROUGH A CLOSE DOOR? (FILE IS NOT OPEN DUMMY)
000 HEATH HDOS ISSUE $50.00.00 (SO WHAT!! JUST SO IT WORKS!!!)
001 I'VE NEVER READ ANYTHING SO BORING IN MY WHOLE LIFE!!
002 HOW MUCH DO YOU THINK THIS DISC WILL HOLD STUPID??
003 ENGAGE BRAIN BEFORE USING 'SYSCALL', SLICK!!
004 THIS ISN'T A STEREO STUPID, CHANNEL ALREADY IN USE!!
005 WHAT DO YOU THINK I AM — AN I.B.M. 370?
006 THIS DEVICE DOES HAVE A PROPER NAME — HOW 'BOUT IF'N I CALL YOU HAZEL?
007 IF YOU DON'T KNOW WHAT YOUR PROGRAM IS CALLED HOW DO YOU EXPECT ME TO??
008 BUY SOME MORE MEMORY FOR THAT DEVICE DRIVER — CHEAPSKATE!!
009 CHANNEL IS NOT OPEN (YOU'D PROBABLY WALK THROUGH A CLOSED DOOR!!)
010 WOULD YOU MIND REPEATING WHAT YOU WANTED ME TO DO.
011 FILE USAGE CONFLICTS (YOU SURE YOU KNOW HOW TO USE THIS THING?)
012 I CAN'T FIND ANYTHING BY THAT NAME, STUPID!!
013 LIKE YOU, I DO HAVE A PROPER NAME; USE IT, SLICK!!
014 DOES YOUR T.V. HAVE CHANNEL 147? I SURE DON'T!!!
015 THE VOLUME DIRECTORY IS FULL AND THAT AIN'T THE ONLY THING!!
016 YOU CAN'T DO THAT WITH THIS FILE, STUPID!!!
017 BUY SOME MORE MEMORY FOR THIS PROGRAM 'SCROOGE'!!
018 I CAN'T READ THIS???? THING — WHAT'D YOU USE, A 78 R.P.M. RECORD?
019 I'D HAVE TO USE A CHISEL TO WRITE ON THIS DISC!!
020 TAKE OFF THE WRITE PROTECT LABEL!
021 DO YOU NEED GLASSES? THERE'S A WRITE PROTECT LABEL ON THIS DISC.
024 WHY DO YOU WANT TO GET RID OF THIS FILE?? YOU'RE GONNA NEED IT, STUPID!
025 THIS ISN'T A STEREO, DUMMY, A FILE IS ALREADY OPEN!!
026 YOU'D HAVE BETTER LUCK WITH THE LIGHT SWITCH THAN THIS DEVICE!!
027 WHY DON'T YOU MOUNT THE DISC FIRST DUMMY!!
028 HEY! RASMUS, USE THE FULL FILE NAME.
029 I CAN'T WRITE ON THIS THING.

Vectored from page 8

BLANKING CIRCUIT OPERATION

Commonly known as TTL 74123, IC107 provides a pair of independent, retriggerable monostables: one-shots. The IC107A circuit "program" one monostable of the pair for blanking. (The remaining monostable serves in RUN indication.)

In Figure 1, IC107A circuit connections are diagrammed according to their specific function in development of blanking pulses. This form of logic diagramming, although encouraged worldwide, is still uncommon. To see how application of IC107 is clarified, compare with the same circuit on the H8 front-panel schematic.

Each time that a logical 1 is detected at the input of IC107A, a logical 1 is produced and held at the output for a specified length of time, even after the input ceases to be a logical 1. (In accordance with international convention, the 1 and pulse symbol in the block identify this characteristic.)

Input to the monostable is the TICK signal entering at pin 10. It is not the presence of a high voltage level that signals a logical 1 input. Rather, transitions from low to high levels are logical 1's. (This important edge-triggering qualification is signified by the open arrowhead penetrating the block.) That is, this monostable only sees logical 1 when the TICK input rises. Nevertheless, the output pulse is held high for a specified duration thereafter, independent of input pulse duration.

TICK is the sole normal input. Other connections shown on the input side of the diagram provide supporting functions.

Pin 9, connected to ground, contributes no logical function. (The cross on the ground path confirms that this connection is logically irrelevant even though electrically essential for correct operation, just like a power-supply connection.)

Actually, this connection effectively programs the 74123, which is more flexible than needed here, to accept positive-going inputs at pin 10.

Pin 11 is the reset (clear) input of the monostable. ("R" within the block identifies this special significance.) When ever this input is at a low level, any output pulse is dropped and no further pulse output is allowed until reset is removed. (That low instead of high signals reset is symbolized in the usual way by the circle at the input point.) Since VF is always at a high level somewhat below 5V, the reset capability is unused, as indicated. (Wiring in this manner has some value during transient conditions such as power-on when supply regulation is not yet stabilized.)

The only significant output from the circuit is at pin 5. There will be a high pulse every time that TICK rises (in the absence of a reset signal). The pulse drops at a prescribed time following the most recent rise of TICK. (This is the retriggerability property; it is not used in this application.)

I have now described everything useful to be gleaned about operation of the IC107A circuit except for answering the most important question: how long after an input before the output falls?

Good question. The desired pulse duration is printed in the logic block, but its achievement is determined by C101, R103, and the particular 74123 installed. Other mysteries of electronics, such as wiring capacitance, no doubt influence operation. In any case, everything at the logical level is specified. At the electronic level, it is presumed that R103 and C101 are selected to provide the necessary pulse duration. This is a matter of specific electronics, not digital logic. You need to consult a 74123 data sheet in order to select appropriate resistor-capacitor combinations. Chapter 4 of Don Lancaster's TTL Cookbook (Heath catalog number EDP-183) gives excellent analysis of 74123 operation and application along with that of other timing components.

If you're ever puzzled how one circuit diagram is chosen over another, as I frequently am, a certain inconsistency may be evident at this point. I'm trying, on the one hand, to show the essence of this troublesome circuit in all of its logical purity. At the same time, it's important to show all necessary electrical connections for the actual components employed, hence the presence of C101, R103, and the ground connection of pin 9. So why aren't crosses drawn on the lines to pins 6 and 7? All I can say is that in light of the importance of those connections, it just doesn't seem right.

That's about all that can be quickly said about blanking operation . Now we can have some fun with the application of IC107.

STATUS INDICATION

The four LED indicators on the left of the front-panel display area reflect operating status of the H8. Each LED allows visual monitoring of some distinct system condition.

Lighting of each indicator is by simple, fixed circuitry. However, conditions that cause lighting are not themselves always straight forward, as we will see.

One indication, RUN, is derived with that half of IC107 "left over" from implementation of protective blanking. The RUN LED is an entirely neutral output intended to confirm that the 8080A micro-processor is actually processing instructions. Beside offering reassurance, this indicator is also an extremely crude thermometer of the kind of running your H8 is doing. Since RUN has little worthwhile use beyond making you feel comfortable, it can be adjusted to your liking. To decide what RUN means and how its behavior is influenced, let's first review the significance of all front-panel indicator LEDs.

GENERAL INDICATOR FUNCTIONS

Starting at the bottom of the indicator stack,

- PWR lights whenever 5 V regulator output is present on the front-panel control circuit board. This is a reliable indication that the H8 main frame has power.
- RUN triggers each time that the 8080A processor starts a new instruction. The indicator dims or extinguishes whenever in struction initiation slows or

halts, respectively. Because instructions usually start every 2 to 9 microseconds, the indicator appears constantly lighted under normal conditions.

- MON is lighted and refreshed under software control to signify that PAM-8 is in control of the keypad and numeric display. PAM-8 extinguishes MON regeneration prior to executing "user" program in response to a GO (4) keypad command.

- ION lights whenever the 8080A is conditioned to accept interrupt requests. ION is extinguished whenever such requests will be ignored. Typical interrupt requests announce 2 ms timer clicks, depression of RTM/O, completion of a single-instruction execution , and availability of an input character from the console-device keyboard. ION dimming/extinguishing signals bursts of interrupt keyboard. ION dimming/extinguishing signals burts of interrupt processing or of other program sections that may not be interrupted.

PWR and MON are absolute indicators, being visibly on or off and having definite significance.

Although RUN and ION are dynamic indicators, being switched on and off as a function of processor activity, switching is at micro-second intervals. Variations are rarely discernable unless there is a prolonged pattern of operation having visible effect.

ABSOLUTE ION INDICATION

Despite the dynamic nature of RUN and ION, there are genuine though extraordinary conditions that extinguish these lamps. You can demonstrate the range of symptoms with a few minutes spent at the H8 front panel.

First, turn the H8 on and off a few times. Notice that appearance of ION, MON, and RUN is delayed beyond the lighting of PWR. That's because the internal reset that accompanies power rise is held for half a second or so. Holding RST/O depressed also forces a reset condition until the two keys are released. Only when the reset condition is removed does the 8080A commence normal sequencing.

Now, to see what it means to disable honoring of interrupt requests, introduce the following short program.

```
040.100  363

040.101  303  100  040

040.100  Pc
```

Commence execution with a GO keypad command. There are a number of interesting effects:

- The horn will sound a continuous tone.

- MON and all LED display digits are extinguished.

- ION is extinguished but RUN stays lit.

- RTM/O has no effect whatsoever.

Use RST/O to restore PAM-8 control. Set up the program again but this time initiate execution with the SI keypad command. Behavior should be identical except for the absence of an audible tone.

What have we demonstrated?

Instruction code 363 is the DI (Disable Interrupts) command. Once executed, this command causes all interrupt requests to be ignored until interruption is enabled. Since the following three bytes command a jump back to the instruction at 040.100, the processor is executing perpetual DI commands with no provision to re-enable interrupts. Consequently, ION goes out although RUN correctly indicates that the processor is still chugging away.

The other effects demonstrate the degree to which an H8 depends on interrupt requests for routine operation. MON and the display digits are blanked because refreshing is inhibited: PAM-8 fails to generate display because interrupts from the 2 ms timer are not being honored. Manual generation of that interrupt request by RTM/O is also fruitless, with PAM-8 therefore failing to see that its control is being requested at the keypad. Finally, the hooter sounds because it is still turned on from acknowledging the GO-key depression: PAM-8 is set to turn off the speaker on the next 2 ms timer-click interrupt, an event we've prevented.

Usage of the SI (Single Instruction) keypad command instead of GO demonstrates another dependency on interrupt processing. The ability to execute precisely one 8080A instruction and then return to the monitor involves a cleverly-derived interrupt request. Since the single instruction being attempted prevents the interrupt from being received, the processor goes its merry way. (The speaker happens to be turned off, in this case, as part of PAM-8's enabling of the "single-instruction interrupter.") There's a moral here for developers of machine-level programs: SI is not usable in debugging of program sequences that have interrupts disabled. SI mode of program execution is still valuable in teaching yourself the behavior of individual instructions though.

The main point is that ION and RUN indication are essentially independent. Even though general-purpose programs such as BASIC do not inhibit interrupts for very long at a time, you may encounter a dedicated application that manages to keep ION off. The operating instructions should emphasize that unusual behavior.

ABSOLUTE RUN INDICATION

To see what it means for the RUN light to go out, now try a variation of the previous program:

```
040.100  363

040.101  166

040.102  303  100  040
```

Continuous execution is initiated by either an SI or GO keypad command, as before, since interrupts are going to be ignored. This time, however, the RUN light should be out as well.

In this case, we've brought the 8080A processor to a screeching halt. After disabling interrupts, execution of further instructions is suppressed by instruction code 166 (HLT). The only event that causes a HLT instruction to "end" is acceptance of an interrupt or occurrence of reset. Because no new instruction is started until the HLT waiting condition is broken, RUN also blinks off.

It is important to recognize that HLT does not, strictly speaking, stop the processor. Rather, the processor waits for an allowed interrupt, any interrupt, before continuing. If the 2 ms timer interrupt is the culprit, PAM-8 will normally notice that the processor was apparently HLT-ed and simulate an automatic RTM.

Idiosyncrasies of the 8080A make PAM-8 have to guess that timer interrupt is on the heels of a HLT. Since there are other ways for 166 to appear in program code, HASL-8 programmers are periodically rewarded with mystery monitor returns. The problem can be aggravated when additional memory is installed in the H8! Fortunately, the programmer can over-rule PAM-8's guesswork using demonstrated in the next section and in Appendix B.

Meanwhile, redo the HLT-demonstrating program with the 363 at 040.100 changed to 000. You can now SI right through the sequence , watching Pc stage from 040.100 to 040.102 and back to 040.100. Execution by GO command comes back immediately with Pc = 040.102 . This is PAM-8's HLT detector in action, another example of the H8's dependence on interrupt processing. The discrepancy between Pc and actual location of the HLT, 040.101, is the source of PAM-8 guesswork. PAM-8 peeks in front of the captured Pc location and concludes that any 166 code is for a HLT that was being executed. All right this time, but not totally reliable. Try the program

```
040.100  041  104  040

040.103  076  167

040.105  326  001

040.107  302  105  040

040.112  064

040.113  303  100  040

040.100  Pc
```

with the GO key held down for a few minutes. What's happening?

DYNAMIC RUN INDICATION

Every time that the H8's processor starts a new instruction, the event is signalled by provision of a unique bus signal, M1.

Although HLT instructions have indefinite duration, other instructions operate in fixed numbers of processor cycles according to systematic rules. The least number of cycles for an instruction is 4. The greatest is 18.* Single cycles take just under .5 microseconds with 2.048 MHz clocking, so instructions are expected to take 2 to 9 μsec apiece.

Because of instruction-time variations, M1 pulses are not uniformly spaced. The upper trace in Figure 2 illustrates the predominant M1 pattern while PAM-8 is awaiting a keypad command. In this particular viewing, we're seeing a composite of M1 intervals, 7- and 10-cyle instructions being very popular. Other timings occur, but too infrequently to make a visible trave.

M1 is unsuitable for direct usage as a RUN indication. Its pulses are too brief and, as a bus signal, it should directly drive a minimal number of TTL inputs, permitting future usage by other bus occupants.

The lower trace of Figure 2 exhibits a corresponding RUN signal easily derived from M1 using IC107B circuitry. This trace was actually obtained at unused output IC107-13 after dropping R153 to about 10 kΩ. The "on" time of the RUN indicator (LED112) corresponds exactly to the "high" duration of he exhibited RUN signal.

Pulse duration of RUN signals is fundamentally a matter of taste. Using long pulses (5 μsec or more) keeps the RUN indicator solidly lighted. Shorter pulse times allow changes in the prevailing pattern of instructions to make pulsations of the RUN indicator. Because of the speeds involved, only dramatic situations will have a pronounced effect.

To demonstrate dynamic variation in RUN indication, work through the following exercises. You can then experiment with values of R153 between 5 k and 50 k ohms until likable results are obtained. Award yourself the compulsive-hacker title when you can tell BASIC is running from the RUN indicator alone.

First, notice the appearance of RUN when the H8 is "idling" in PAM-8 after power is applied. There should be no perceptible change if you execute the following do-nothing loop:

```
040.100  303  100  040

040.100  Pc
```

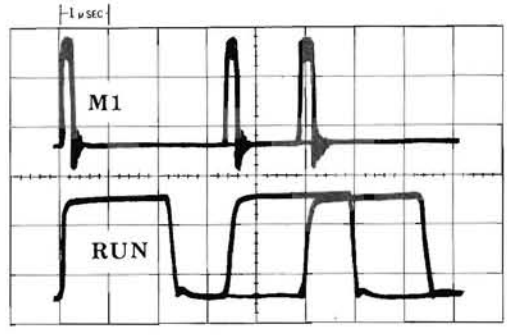


Figure 2

*Actually, not every cycle need be given over to progress on a current instruction, so the elapsed time can be greater than the minimum required by the actual instruction. Such instruction-time "stretching" is absent in typical cassette-oriented configurations.

Now compare with a program dominated by very fast instructions:

```
040.100  041  103  040

040.103  204

...      ...

040.117  204

040.120  351
```

Any brightening of RUN will be very subtle, even with RUN pulse durations as short as 2 μsec.

For the other extreme, try this loop of time-consuming instructions:

```
040.100  041  103  040

040.103  345

040.104  343

...      ...

040.147  343

040.150  311
```

This is the dimmest RUN that can be obtained with anything like normal instruction execution. Obviously, the RUN indicator is not a terrific speedometer.

Many computer systems, including some of the largest, provide visible indications when they're not working very hard. This is also possible with the H8, provided that programs do "hard" waiting instead of "busy" waiting.

By busy waiting is meant any procedure where the processor is kept active simply waiting to see if there is work to be done. An example is the waiting that PAM-8 does for keypad commands and that BASIC does for input characters. Hard waiting is the kind provided by HLT instructions, once PAM-8 is discouraged from doing anything about them. A typical "idle loop" demonstrating this technique is accomplished by the following instructions:

```
040.100  072  010  040

040.103  366  200

040.105  062  010  040

040.110  166

040.111  303  100  040

040.100  Pc
```

HASL-8 programmers hoping to introduce new system software for the H8 can avoid a lot of grief by considering the implications of the example routines:

- It's not nice to disable interrupts any more than absolutely necessary.
- PAM-8 HLT detection doesn't really work and is best avoided.
- HLT instructions written with the expectation of being detected are also very dangerous.

The first warning is because someone, someday, will want to run your program in conjunction with a concurrently-running application having critical interrupt requirements.

Secondly, re-assembly of your program will someday produce an unfortunate 166 to be incorrectly trapped as an actual HLT. You may have already produced such a program but not been caught by the necessary timing coincidence. The only sure way of avoiding such embarrassment is to disallow halt detection. Coding of the following form should exist in all serious HASL-8 programs:

```
.MFLAG   EQU   040010A

UO.HLT   EQU   200Q

...

LDA      .MFLAG

ORI      UO.HLT

STA      .MFLAG

...
```

The prohibition on HLT instructions as a programming technique follows pretty immediately once we've agreed not to rely on PAM-8 for HLT detection. There's another reason too. Even **with** HLT detection, the HLT can be missed! Remember that **any** accepted interrupt breaks the HLT. There's no reason to presume that PAM-8's timer interrupt will always be the winner.

EOF

Curing Single-Drive HASL's

By: *John Beetem*
Quillen 4-I
Escondido Village
Stanford, CA 94305

One problem with the HASL-8 assembler is that you need two tape drives to assemble a program with origin in low memory (i.e. below the HASL high-memory address.) This is because two files must be open: the source code file (input) and the object code file (output.) You cannot assemble directly into memory because the assembler will be written over. This article describes a simple modification to HASL-8 so that programs with origin in low memory can be assembled using **one** tape drive.

The idea is very simple: the object file is stored in memory above the high memory address instead of being dumped onto tape. Note that this is different from assembling the program directly into memory, because the record information described on pages 0-12 through 0-15 of the H8 Software Reference Manual is stored also. After assembly, the source tape is dismounted and the data stored in high memory is dumped verbatim onto an object tape.

This modification is necessary for single-drive systems to be able to assemble programs with origin=041.144, the first address after the console driver. It is also an improvement over assembling directly into memory and then dumping using PAM since the dump parameters (PC, start of dump, end of dump) are supplied automatically by HASL.

Using the modification described in the Listing

(you must have at least 12K of memory.)

(1) HASL-8 must be configured so that HIGH MEMORY = 18431.
(note: 18431 = 110.000 −1)

(2) HASL always outputs to tape using the $TDOUT (Tape Data Out) subroutine at location 040.133 in the console driver. This three-byte subroutine must be replaced with a jump to 110.004, i.e. patch 040.133 with 303,004,110.

(3) Load the program Listing 1 before assembling. This causes bytes intended to be sent to tape to be stored in high memory.

(4) Assemble your program as usual. When asked BINARY TAPE? answer Y.

(5) When assembly is complete, the data which would have been written onto tape is stored in memory starting at 110.100. Location 110.002 (COUNT) contains the number of bytes stored.

(6) To dump the data onto tape, set PC equal to 110.030 and press GO. COUNT+2 bytes are written so that no significant bytes are lost when the tape is turned off.

(7) Multiple copies can be written by pressing GO again.

(8) The tape stored can be loaded using PAM or BUG. The initial PC will be the address specified in the END statement of the source program.

(9) Since POINT and COUNT are changed during assembly, step (3) should be done before assembling another program, unless you want to concatenate object files.

```
* Program to write assembler object tape into memory, so that
* you don't need two tape drives.
* answer 'Y' to HASL-8's question: Binary Tape?
* Make the following patch to HASL-8:
* replace 040.133:  323,370,311  with   040.133: 303,004,110
* HASL should be configured so that high memory = 18431 [107.377A]
* The following program should be loaded at 110.000
*

110.000   100,110       POINT   DW      AREA
110.002   000,000       COUNT   DW      0
                        * simulate dumping of one byte
110.004   345           DMPSIM  PUSH    H               No registers changed
110.005   052,000,110           LHLD    POINT           save byte in AREA
110.010   167                   MOV     M,A
110.011   043                   INX     H
110.012   042,000,110           SHLD    POINT
110.015   052,002,110           LHLD    COUNT           count=count+1
110.020   043                   INX     H
110.021   042,002,110           SHLD    COUNT
110.024   341                   POP     H
110.025   311                   RET
110.026   000,000               DB      0,0
                        * Dump tape stored in memory onto object tape
                        * This program is to be run after the
                        * assembly is finished.  DE= # of bytes left
110.030   052,002,110   DMPAREA LHLD    COUNT           DE = count+2
110.033   353                   XCHG
110.034   023                   INX     D
110.035   023                   INX     D
110.036   315,152,040           CALL    SPRSCL          preset UARTs
110.041   041,100,110           LXI     H,AREA          HL points to AREA
110.044   076,021               MVI     A,21Q           Turn on tape
110.046   323,371               OUT     371Q
110.050   333,371       DA1     IN      371Q            wait for trans. ready
110.052   346,001               ANI     1
110.054   312,050,110           JZ      DA1
110.057   176                   MOV     A,M             get byte
110.060   043                   INX     H
110.061   323,370               OUT     370Q            output byte
110.063   033                   DCX     D               count=count-1
110.064   172                   MOV     A,D             count=0?
110.065   263                   ORA     E
110.066   302,050,110           JNZ     DA1
110.071   323,371               OUT     371             turn off tape
110.073   166                   HLT
110.074   303,030,110           JMP     DMPAREA         re-dump
110.077   000                   DB      0
                        * Dump area...
                        * This area must be big enough to save the
                        * entire object tape
110.100   000           AREA    DB      0
                                END
```

EOF

CONFIG.SAV — . . .“Take My Pulse, Doc.”

H.G. Miller

```
.nlist ttm
.enabl ama,lc
.title confis
.sbttl Hampton G. Miller at the Heath Company 9-Dec-78
;+
;       Display various system parameters.
;
;       System has:
;       [4-28]K of memory
;       EIS/FIS Chip
;       Paper-tape Reader
;       Paper-tape Punch
;       Line-printer
;       Line-time-clock running
;       H27 Extended Mode / RX01 compatible Mode
;-

.mcall .readef,..v2...,.print,.exit
..v2..
.readef

.slobl confis

trap4=4
trap10=10
clkvec=100
PR=177550       ; paper tape reader
PP=177554       ; paper tape punch
LP=177510       ; line printer
RXCS=177170     ; H27 device registers
RXDB=177172

confis: .print  #sysmss                 ; 'System has'
        mov     @#TRAP4,-(sp)
        mov     #memtst,@#TRAP4         ; install our own vector
        mov     #sizmsg,R0              ; point at '4' message
        mov     #20000,R1               ; scan in 4K increments
10$:    tst     @R1                     ; memory there? (will trap if not)
        add     #20000,R1               ; point to next 4K bank
        cmp     (R0)+,(R0)+             ; step to next 4K ascii string
        br      10$                     ; sooner or later a trap will get us

memtst: mtps    (sp)+                   ; restore previous PS
        tst     (sp)+                   ; discard interrupt PC
        mov     (sp)+,@#TRAP4           ; restore system vector
        .print                          ; string already in R0
        .print  #kmss                   ; 'K of memory'

; test for EIS/FIS Chip
eischk: mov     #eismss,R0              ; 'EIS/FIS Chip'
        mov     #eistst,R1              ; FADD R0 instruction
        jsr     pc,test4                ; perform common test code

; test for paper-tape reader present
prchk:  mov     #ptrmss,R0              ; 'Paper-Tape Reader'
        mov     #ptrtst,R1              ; TST @#PR instruction
        jsr     pc,test4                ; call test independent routine

; test for paper-tape punch present
ppchk:  mov     #ptpmss,R0              ; 'Paper-Tape Punch'
        mov     #ptptst,R1              ; TST @#PP instruction
        jsr     pc,test4

; test for line-printer present
lpchk:  mov     #lptmss,R0              ; 'Line-Printer'
        mov     #lptst,R1
        jsr     pc,test4

; test for line time clock present (this is not quite so easy...)
clkchk: mov     @#clkvec,savclk         ; save clock vector
        mov     #setclk,@#clkvec        ; install our own
        mov     #40000,R0               ; allow plenty of time for clock tick
10$:    sob     R0,10$                  ; twiddle thumbs
        mov     @#clkvec,R0             ; get system vector
        mov     savclk,@#clkvec         ; restore regardless
        cmp     R0,savclk               ; did we catch clock tick?
        one     20$                     ; no, no clock
        .print  #clkmss                 ; yes, clock is running
20$:

; Test for RX01/H27 mode (this can be dangerous: H27 microcode may change
; without warning.)
H27chk: mov     #rxmss,R0               ; assume dumb
        CLR     @#RXDB                  ; select H27 NOP
        mov     #11,@#RXCS              ; say the magic word(Unused RX01 code)
10$:    tst     @#RXCS                  ; wait for response
        ben     10$
        bit     #40,@#RXCS              ; DONE set?
        bne     20$                     ; yes, he's playing dumb
        mov     #H27mss,R0              ; no, assume he's smart
        clr     @#RXDB                  ; finish out H27 NOP sequence
20$:    .print                          ; print string addressed by R0

        mov     sp,R0                   ; quiet exit (re-startable, too!)
        .exit

; Here to test for EIS/FIS chip
eistst: mov     sp,R1                   ; point to FADD args
        cmp     -(R1),-(R1)             ; move down into stack
        fadd    R1                      ; cause trap if no EIS chip
        rts     pc                      ; else return

; Here to test for paper tape reader
ptrtst: tst     @#PR                    ; cause trap if no paper tape reader
        rts     pc

; Here to test for paper tape punch
ptptst: tst     @#PP                    ; cause trap if no paper tape punch
        rts     pc
```

```
; Here to test for line printer
lpttst: tst     @#LP                    ; cause trap if no line printer
        rts     pc

; Here upon clock tick. Restore vector and propagate dispatch.
notclk: mov     savclk,@#clkvec
        mtps    @#clkvec+2              ; set PS
        jmp     @clkvec                 ; finish up tick accounting

;+
; Test routines TEST and TEST4.
;
;       Entry   (R0) = address of ascii string to be printed if test wins
;               (R1) = address of device dependent test routine
;               (R2) = address of test vector (TEST only. TEST4 sets R2)
;
;       Exit    none
;-
test4:  mov     #TRAP4,R2       ; select illegal memory reference trap
test:   mov     @R2,-(sp)       ; save vector
        mov     #10$,@R2        ; install our own
        jsr     pc,@R1          ; use callers routine
        .print                  ; string address is already in R0
        br      20$             ; skip interrupt dispatch recovery

10$:    tst     (sp)+           ; discard return PC
        mtps    (sp)+           ; restore previous status
        tst     (sp)+           ; discard interrupt return PC
20$:    mov     (sp)+,@R2       ; restore vector
        rts     pc              ; return to caller


.nlist bin
sysmss: .asciz  \System has:\
kmss:   .asciz  \K of Memory\
eismss: .asciz  \EIS/FIS Chip\
ptrmss: .asciz  \Paper-tape Reader\
ptpmss: .asciz  \Paper-tape Punch\
lptmss: .asciz  \Line-printer\
clkmss: .asciz  \Line-Time-Clock\
rxmss:  .asciz  \RX01 Compatible Mode\
H27mss: .asciz  \H27 Extended Mode\

; SIZTAB consists of 8 4-byte entries. Each entry can be printed as
;  an ASCII string.
siztab: .ascii  \4\<200><0><0>\8\<200><0><0>\12\<200><0>\16\<200><0>
        .ascii  \20\<200><0>\24\<200><0>\28\<200><0>\30\<200>
        .even

savclk: .blkw   1

        .end    confis
```

"H8 CRASH ARRESTER"

Intermittent failures seem to be the most frustrating experiences a new computerist can go through. I had been having problems with my H8 for several months — first, with a Selectric typewriter I had hooked up through a parallel port; then, the system began to crash, occasionally, then often, until I was ready to give up. I had taken the computer to the store (couldn't find anything wrong), and had made numerous calls to Benton Harbor, but nothing I tried helped. I even removed the motherboard and resoldered every connection and cleaned all the pins (there are a lot of pins on that mother!). I complained a lot at the local HUG meetings, too.

Then, one of our club members, Bob Craig, heard me, and said he had had the same mysterious problem. An inspiration prompted him to look at the voltage regulator plug-in connection on the CPU board. (That's sacred ground, right? Heath didn't want us to touch that board, right?) He reasoned that an oxidized connection could cause the CPU to lose its 5 volts, perhaps even just for microseconds, and the 8080 would get out of step with its program. So he cleaned up the connectors, both on the IC and the plug, bent the springs out a little to tighten the contact, and burnished them together. And he had experienced no crashes for several weeks.

Well, I had spent hundreds of hours trying to figure out what was wrong, so I'd give it one more try. It took about fifteen minutes to do, and I've been up and running ever since.

I keep thinking of that old joke about the guy who charged a dollar for hitting the machine with a hammer, and a hundred dollars for knowing where to hit it. What it demonstrates is that the value of the local HUG group can be incalculable, because where else can you accumulate so much practical experience with the exact problems you are likely to have yourself?

It's fun, and time-saving, to share software through HUG. But it's sharing solved problems that really makes the group worth while.

Donald Skiff
2448 Vera Ave
Cincinnati OH 45237

Thanks Don — We've heard of perhaps a dozen other H8 owners with the same mysterious problem and just about the time your letter came in we verified that it is, indeed 'tacky' contacts on the +5 V regulators on any of the boards. For now, if any one else is experiencing this problem, I suggest eliminating the sockets and soldering the wires directly to the leads of the regulator — :JB:

Just at press time, we acquired a new Text Editor for the H8/H17 system — BWEDIT is a character oriented editor. Multiple, one letter commands may be placed on one line without delimiters — A FNAME.BAK (backup) is automatically generated and a string of commands may be repeatedly executed by a single command. (MACRO)

BWEDIT.ABS and BWEDIT.DOC is availabel from HUG by ordering (on green order form) 885-1022, cost is $15.00.

:JB:

COMING SOON! A TEXT FORMATER

ERROR IN 'BIORHYTM' SOFTWARE TAPE — ISSUE 40.00.00

This problem came about when I ran into someone with a birth certificate in their pocket. I compared the computer results with two different perpetual calendars and found the computer program to be in error.

After entering your birth date, the program returns the 'day of the week' on which you were born. This data is correct except for the month of February of any Leap Year.

example: Input 'Birth Date' 2,24,1928 (NOT MINE!!)
Results show that day to be Saturday. Actually it was Friday.

I am attaching a program of my own which covers the day of the week for any date from September 14, 1752 to 2000 A.D..

Thank you for your kind consideration of this matter, I remain,

D. A. Provancher
441-L Church Ave
Chula Vista CA 92010

```
 5  PRINT TAB(15)"**DAY OF THE WEEK**-BY:D.A. PROVANCHER":PRINT
 6  PRINT TAB(15)"EXTENDED BENTON HARBOR BASIC*ISSUE #10.02.01":PRINT:PRINT
10  DIM D$(7),M$(12)
20  D$(0)="SATURDAY":D$(1)="SUNDAY":D$(2)="MONDAY":D$(3)="TUESDAY"
30  D$(4)="WEDNESDAY":D$(5)="THURSDAY":D$(6)="FRIDAY"
40  M$(1)="JANUARY":M$(2)="FEBRUARY":M$(3)="MARCH":M$(4)="APRIL":M$(5)="MAY"
50  M$(6)="JUNE":M$(7)="JULY":M$(8)="AUGUST":M$(9)="SEPTEMBER":M$(10)="OCTOBER"
60  M$(11)="NOVEMBER":M$(12)="DECEMBER"
70  PRINT TAB(5)"DAY OF THE WEEK FOR ANY DAY SINCE SEPTEMBER 14, 1752";
71  PRINT" TO 2000 A.D."
80  FOR X= 1 TO 64: PRINT TAB(5) "=";: NEXT X:PRINT :PRINT
90  INPUT "     MONTH,DAY,YEAR  (USE COMMAS) >"; M,D,Y
95  REM JAN AND FEB BEING THE 13TH & 14TH MONTHS OF THE PREVIOUS YEAR
100 IF M=1 THEN M=13 : Y=Y-1
110 IF M=2 THEN M=14 : Y=Y-1
115 REM FOR DAY OF THE WEEK BETWEEN 1900 AND 2000 A.D.
120 W1=D+INT((13/5) * (M+1))+INT(1.25 * Y):IF Y>=1900 GOTO 140
125 REM FOR DAY OF THE WEEK BETWEEN 1752 AND 1999 A.D.
130 W2=W1-INT(/100)+INT(Y/400): W=W2-((INT(W2/7))*7): GOTO 150
140 W=W1+6-((INT((W1+6)/7))*7)
145 REM REVERT JAN AND FEB BACK TO 1ST & 2ND MONTHS OF THE YEAR ENTERED.
150 IF M=13 THEN M=1 : Y=Y+1
160 IF M=14 THEN M=2 : Y=Y+1
170 PRINT :PRINT TAB(10) D$(W);".....";M$(M);"";D;"";Y: PRINT: GOTO 90
```

HUG MEMBERSHIP RENEWAL FORM

You can determine your expiration date by examining the last six digits of your ID number — example: 780202 indicates your membership began 02/02/78 and expires one year from then.

IS THE INFORMATION ON THE REVERSE SIDE CORRECT? IF NOT FILL IN BELOW

Name _____

Address _____

City-State _____

Zip _____

REMEMBER — ENCLOSE CHECK OR MONEY ORDER

CHECK THE APPROPIATE BOX AND RETURN TO HUG

NEW MEMBERSHIP?
FEE IS:

RENEWAL RATES			NEW MEMBERSHIP
US DOMESTIC	$11 ☐		$14 ☐
CANADA	$13 ☐	US FUNDS	$16 ☐
INTERNAT'L*	$18 ☐	US FUNDS	$24 ☐

* Membership in England, France, Germany, Belgium, Holland, Sweden and Switzerland is aquired through the local distributor at the prevailing rate.

THE BACK PAGE —

As soon as these words are written, work will resume on putting the finishing touches on Volume II of HUG Software. Let me tell you about it —

Volume II and Tape II is being prepared so that you may assemble them into a complete, neat package, including a HUG binder and cassette holder. All programs are written for the H8 cassette system — (Some may convert to disk, most would require some degree of modification).

Some of the programs included are:

An excellent Data Base Management program.
Two programs that allow you to 'Create' a **customized** program to suit your needs (mailing lists, for example) with no programming knowledge.
Miscellaneous I/O Routines.
Complete File Maintenance and Sort programs.
DC Circuit Analysis.
Metric Conversion (Good Educational Tool).
Calendars.
Utility Graphs.
Electronic Formulas.
BASIC Renumbering Programs and 'Merge'.
Mailing Lists.
More Disassemblers.
Tape and File Management.

And the next release will include:

Personal Accounts Payable.
Credit Card Management.
Inventory Control.
Home Budget Control.
Expense Account Maintenance.
And Many More Surprises.

We're beginning to accumulate some very nice software for the H8 disk system — these are being evaluated and assembled into a very nice package that will be available real soon — hopefully after March.

You will be receiving an updated catalog soon that details the software available and ordering information.

Here's another reminder about continuing your membership in HUG — details are on the inside back cover — some very exciting things are in store for us in '79 and we want to share them with you.

Contest #4 deadline — April 9. Same rules apply as in the past — see page 16 for details. By next issue we hope to hear from you H27 and ETA-3400 users.

H17 owners — A new release of HDOS should be arriving any day — if you didn't return your registration card, you will not receive it. Primarily, some irritating bugs were fixed in BASIC and an LP driver was added. Also, you will be able to list BASIC program statements **from** BASIC in the command mode.

What's H8-18 and H8-14?

H8-18 is the system cassette software for the H8. If you do not have the H8-4 serial interface card, you don't need it — same goes for H8-14 which is Extended Benton Harbor BASIC with 'file capability'. Same as H8-13 (version 10.02). These were modified to talk to the 4 port serial board and the line printer. This means you can change ports and baud rate from the keyboard.

:JB: (right) at AVA show in Dallas

Heath Users' Group
Hilltop Road
St. Joseph MI 49085

BULK RATE
U.S. Postage
PAID
Heath Users' Group

POSTMASTER: If undeliverable, please do not return.

885-2005