



A MONTHLY MAGAZINE OF IDEAS
FOR THE MICROCOMPUTER EXPERIMENTER

Publisher's Introduction:

This issue marks the first edition of 1975 and the beginning of a new monthly format for the ECS articles. From now on the publication will be on a monthly basis with 12 issues per year. Minimum issue size will be 20 pages printed photo-offset as in the past. The editorial policy will continue to emphasize materials useful in the creation and programming of home brew computer systems. In this first issue of 1975 readers will find:

1. ECS-6 Serial I/O Interface Conclusion: The last issue of the 1974 series of articles described the theory of operation and subsystem design of the UAR/T oriented 4-channel I/O interface unit. This issue contains additional materials including logic diagrams, tables, notes on detailed logic, and notation of a test program useful in debugging the design.
2. Notes on Notations: Taking into account numerous inputs from subscribers together with further arguments and rationalizations, a decision to use octal notation and "Intelese" is described in this issue.
3. Memory Dump Program "ELDUMPO": The application of the serial I/O interface device with a teletype is illustrated in the listing of this program's code. In true bootstrap practice, ELDUMPO is used to dump ELDUMPO! (To say nothing of the other listings in this issue.)
4. Manual Bootstrap Program "STUFFER": "STUFFER" is a program used in conjunction with the ECS keyboard (see ECS-5) and display lamps to load data at arbitrary locations in memory. It can be loaded by hand in locations 100 to 163 of page 0 using the ECS-3 program's bootstrap hardware method. Then, this program can be used to load in octal further programs such as ELDUMPO.
5. Programming Notes: Using Restarts. Both ELDUMPO and STUFFER make use of restart instructions (RSTx) to access utility routines. The method is described in this section of the issue.
6. Notes of Interest to Readers: Miscellaneous comments and a couple of errata presentations.

Carl Z. Helmers, Jr.
January 25 1975
Publisher

ECS-6 SERIAL I/O INTERFACE CONCLUSION:

In the last issue the general design and theory of operation of the UAR/T oriented serial I/O interface was presented. The major technical topic of this issue is the detailed description of this hardware as it is implemented in the ECS prototype system. The drawings found on pages 4 and 5 show the details of the circuit. In the text below reference should be made to the drawings and to the general description of the system presented in the previous issue.

UAR/T and Bus Input:

Drawing #1 on page 4 contains the logic of the UAR/T chip and its interface to the system data bus. The address bus input lines A0 to A7 are wired from the I/O socket #1 to the UAR/T parallel inputs TD0 to TD7 (the notation used by the manufacturer's documentation is TD1 to TD8, but renumbering is done here for consistency with the rest of the system.) The address lines are written into the UAR/T as data to be sent out the serial port whenever the following conditions hold:

- a. The mode selected by the control word (IC-9-, dwg. 2) is "output."
- b. The CPU I/O instruction decode logic of the ECS-5 design (or its equivalent) generates an OUT02 clock which is inverted by -7c- and enabled through gate -8a- to the TDS (transmitter data strobe) line of the UAR/T.

The negative going clock pulse which reaches the UAR/T chip from -8a- automatically starts the stored program of the UAR/T chip which transfers data to the output buffer shift register and begins generation of the serial data format. The serial data generated by the UAR/T appears on pin 25, Transmitter Serial Output (TSO).

The latest received data of the UAR/T is present at all times on the RD0 to RD7 lines (mfr's designations RD1 to RD8 - see above.) The output is always enabled due to wiring the receiver data enable line of the UAR/T to ground (pin 4). The actual control of this data enable is provided in fact by the IN02 signal provided by the ECS-5 I/O decode logic - going to the 8T09 gates which interface the CPU bus. Note that the IN02 instruction is one of the combined input/output instructions of the 8008 - the corresponding output of the accumulator is sent to the UAR/T if the controlword indicates output rather than input. But when input is exercised, the OUT02 clock time is also used - this pulse is used to reset the RDA flag of the UAR/T after input, to acknowledge that the CPU has processed the data. The acknowledgement from a CPU I/O handling program must come within one character period of the "RDA" signal's transition to the logical "1" state if the "receiver overrun" error is to be avoided. Note that just as the I/O transfer of data out to the UAR/T is ignored when input is involved, the output of data to the UAR/T also reads the UAR/T information into the accumulator, but this information is in general meaningless.

Note that the bus interface gates invert the sense of the data being read out of the UAR/T. In order to guarantee that the sense of the data being input to the computer is the same as that written out (ie: "1" is logical 1, "0" is logical 0) a level of inversion is required between the UAR/T and the bus interface buffer. Also note that an improvement in the design would be to use the data out strobes (pin 4 and 16 for data and status, respectively) to implement a local 3-state bus sharing a single set of inverters/bus interface gates to the CPU world.

The status bit outputs of the COM 2502 UAR/T are read into the CPU with the input operation IN3 (sometimes noted IN03). As with all the input operations of the 8008, there is a paired output of accumulator contents - in this case defining the control word content from the accumulator. The IN3 negative clock is used to enable control information onto the bus during the input operation, and is generated by the ECS-5 hardware or its equivalent from basic Intel 8008 signals. The status bit outputs are always enabled due to wiring of the status word enable pin (pin 16) to ground. As noted on the previous page, this output of the UAR/T can be multiplexed to a single set of inverters/8T09's in an improvement over the design used in the prototype.

As is the case with the data word output of the UAR/T, a series of inverters is shown in drawing #1, one per status word line, placed in the design in order to make the program operating the UAR/T be programmable based on bit definitions identical to the definitions produced by the UAR/T. If programming based on a single level of inversion (complements of the definitions) is tolerable, the inverters may be omitted. An alternative which retains the proper bit definitions without inversion twice is to use a non-inverting bus buffer instead of the 8T09.

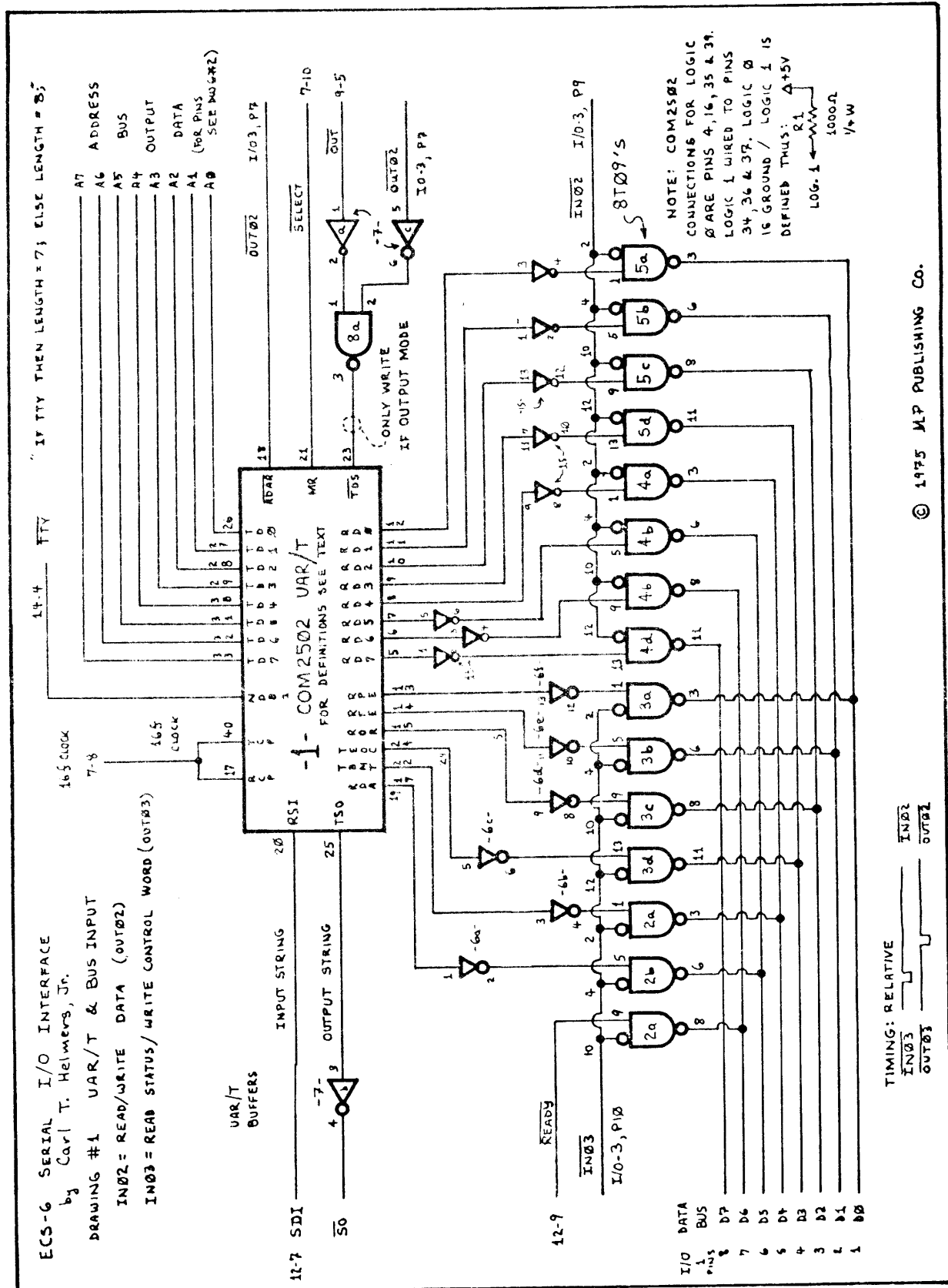
Control and Multiplexing Logic :

Drawing #2 on page -5- contains the remainder of the logic associated with this design. It includes the clock oscillator, control word register, input multiplexing and output selection logic.

The basic clock of the system is generated by a 555 circuit wired as an astable configuration acting as an oscillator. The clock rate is adjusted by R3 to the nominal frequencies required by the system. The logic of this design requires a clock setting of 56.32 Khz for a nominal 110 KBaud rate with the low frequency clock programming value set in the control word. The UAR/T logic requires a clock at 16 times the basic bit rate, thus with the divide by 16 mode of the clock rate counter set by a binary value of "1111" in the control word rate bits, a total division of 256 will be in effect. 110×256 is 28.16 Khz. The extra division by 2 is found in the flip flop used to turn the extremely short (ie: 50 ns or so) clock reset pulse into a square wave which is within tolerance limits of the UAR/T (eg: greater than 1 microsecond in length.)

The clock outputs are the Q and \bar{Q} pins of the 7473 section used to produce the square wave. One of these outputs (Q, pin 12) is used to drive the UAR/T clock pins for both transmitter and receiver sections (pins 1-17 and 1-40) The other output is routed to the I/O socket for use by the modems connected to the controller.

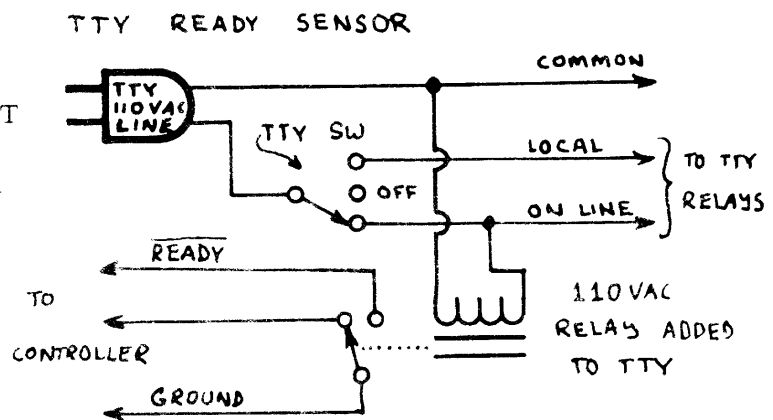
IC -9- is a 74100 used as a control word. This IC will store 8 bits of data when its clock lines are strobed with a positive logic pulse, derived in this case from an inversion of the negative logic OUT03 signal produced by the ECS-5 type controller or its equivalent. The four rate selection bits are wired to the 74193 counter used



to program the different data rates possible with this design. The two channel selection bits are wired to the multiplexor of the input data and ready signals, and to the output data and "select" selectors.

The In/Out bit, bit 1, is used to enable the output write function for the UAR/T with the OUT02 clock, and is also routed to the output plugs for use by the modems in setting up their operation. The select bit is multiplexed to one of the four channels of output via the 74155, IC -13-. The select outputs are in positive logic form. The teletype device, channel 0, has its select shown in the drawing as driving an inverter (-7f-) which in turn drives an LED indicator shown remote by the connector symbols. The purpose of this logic is to provide a visual indicator at the teletype telling the operator (ie: you) that the CPU is addressing that machine. This indicator is entirely optional and may be omitted if desired.

Ready Logic is provided by one section of the multiplexor 74153, -12-. This circuit is used to select the source of the "ready" signal which will be placed on the bus as a status bit (position 6) when the IN3 operation interrogates status. For the channel 0 case (teletype) the ready function may be driven by a relay connected in parallel with the "on line" side of the teletypewriter's front panel switch. The relay should be an SPDT contact variety with a 110VAC coil. The normally open contact is closed when the coil is energized by the switch, thus grounding the ready line input. For the tape recorder interface modems, the ready line is driven by a "turn on delay" one shot which is cued by the falling edge of the edge of the select signal to the device in question from the 74155 selector. The second section of the 74153 is used to multiplex the serial inputs of the device, from one of the four possible sources - TTY or tape channels 1 to 3.



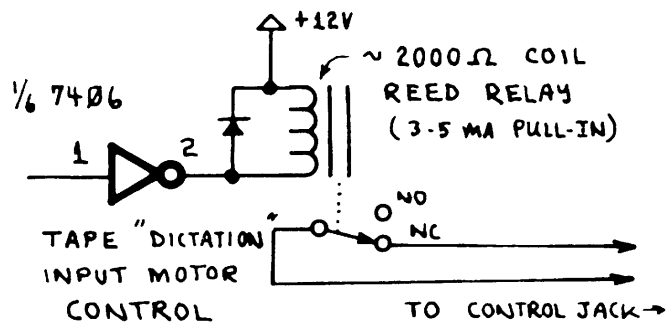
Serial Data Input is routed via the 74153 IC -12-. One section of this IC is used to select the source of the serial data input to the UAR/T. This input is taken from the teletype switch contacts for channel 0, and is the serial output of the tape recorder storage device's receiver section for the other 3 channels. The teletype data is generated by a carbon brush mechanical switch controlled by the mechanism of the keyboard button pressed. When using input from the teletype, the operation of the mechanical switch produces a contact closure for the current loop "mark" state (idle) and breaks the loop for the opposite ("space") state. This means that to make the proper sense to the UAR/T, there must be one level of inversion prior to the selector if the preferable "pull up" TTL input form is used.

Serial Data Output of the UAR/T emanates from pin 25 of IC -1-, and is first inverted by -7b- before being routed to the output data selector, 74155 -13-. Since the section of the 74155 used for the serial output data has one net level of logical inversion (unlike the other section of the same chip) the inverter is required if the signal sent to the modem or teletype is to be identical to the signal derived from the UAR/T.

The channel 0 serial data output is wired to the 7437 high power NAND gate section to generate a TTY current loop signal for driving the print mechanism. Since the "true" or "logical 1" state of the current loop is current flowing in the loop, this state must be generated by a logical zero output for the driver tied through the TTY electronics to the high level voltage. This single level of inversion provided by the driver suffices to create the proper signal - true data output of the multiplexing logic of the 74155 is the "mark" state which inverted generates a current loop "on" state when the UAR/T is idle.

The serial outputs of the other three channels are wired to I/O socket #2 along with the other signals necessary to drive the modems.

Select Output Logic is also provided by the 74155. As mentioned earlier, the select for channel 0 is wired to an indicator lamp. The source of the signal for all channels is the select bit of the control word. For the tape drive modems, the select signal for channels 1 to 3 is used to control the "motor on" state of the tape recorder. In the logic of the tape interface units, the rising edge of the select line for the channel in question should trigger a one shot "motor start" delay, as well as turn on the tape recorder's motor for the beginning of operations. The "motor start" delay one shot has sufficient delay involved to allow the motor to get up to speed and relatively stable operation. For cheap tape cassette devices this time may be as much as 5 to 10 seconds - if the motor and drive ever stabilize. For the more expensive forms of cassette recorders, a shorter delay may suffice. Given a cassette recorder, the characteristics of motor speed versus time from turn on should be examined to determine the minimum delay required for reliable operation. In the previously published ECS-2 design, one method of turning on the tape drive motor was detailed - a "tape drivebox" with a power supply and transistor switch to drive the motor via the "external power supply" jacks often found on battery operated cassette recorders. The diagram at the right shows an alternate and much simpler mechanism to control the motor via a "dictation" control input normally connected to a switch in the microphone. The relay used is a micro-reed design, in this case a "Grigsby-Barton #GB31C-G2150" removed from surplus equipment.



The relay used in the prototype of this circuit had a coil resistance of 2000 ohms (approximately) which gives a current of 6 milliamperes with a 12 volt drop when the open collector 7406 energizes the coil. The 7406 can drive up to 25-30 milliamperes with no difficulties, which means that using this particular IC as a driver, relays with resistances as low as 400 ohms could be used, provided a 12 volt drop gives sufficient current to pull the switch contacts. To see whether a given "unknown" relay will work in this application, its pull in current should be measured using a variable voltage power supply with a current meter. Hook an ohmmeter to the switch contacts of the relay and observe the current and voltage at which transitions in the switch contact state occur. If the current at which the contacts "pull in" does not exceed about 25-30 ma at supply voltages of up to 12 volts, then the coil can be wired into the circuit shown on page 7.

Wiring and testing the Serial I/O Interface:

The prototype of this design was built using wrapped wire construction techniques as described in M. P. Publishing Co. publications 73-1 and 74-5. As in any complex circuit, whatever your method of construction, use care in wiring and checking the wiring. The following steps are a suggested set of testing stages for this circuit.

1. Verify all wiring and check the circuit's power supply connections by applying power (with no IC's yet in sockets) and checking the proper pins as listed in the table on page 9.
2. Check out the oscillator and clock generation logic first. Plug in the entire complement of integrated circuits with the exception of the UAR/T chip for preliminary checkout. Check the oscillator output after applying power to the circuit. Adjust the frequency using an oscilloscope or a frequency meter. The frequency should be 56.32 Khz, which corresponds to a period of 17.76 μ s for those who use scopes for calibration.
3. Set up the following simple program in the CPU using the bootstrap mode of data entry:

000 010 INB next rate	011 020 select code
001 301 LAB rate to accum	012 111* IN3 write CW, read stat.
002 002 RLC move rate to	013 177* OUT30 display stat.
003 002 RLC to the	014 113* IN2 read UAR/T
004 002 RLC high order	015 175* OUT31 display data
005 002 RLC of accum.	016 006 LAI define the
006 044 NDI purge the	017 003 003 reset code
007 360 360 garbage bits	020 117 IN0 reset inter.
010 064 ORI or in the select	021 377 HALT

*I/O codes of ECS-5 altered for extraneous inverters

This program responds to interrupts by calculating the next rate code for the serial I/O controller and outputting it to the controller. Look at the frequency on the clock line of the UAR/T socket - and observe changes as an interrupt is raised on the keyboard. Note that the instructions marked with an "*" use codes consistent with ECS-5's drawing #1 - see the errata section of the last issue for comments regarding the inverters in that design's drawings and their effect on codes.

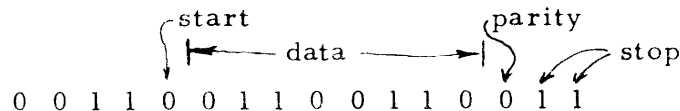
The following experiment can now be performed - with the UAR/T still out of the socket connect the clock pins of its socket to a .1mfd condenser to the input of a stereo amplifier channel. Listen to the clock generator output as the program is cycled and note aurally the different rates.

4. Now turn off the system power and plug in the UAR/T, taking into account the precautions listed below. Re-apply power to the system, and load the following simple program to test data transmission. Look at the UAR/T output at the pin of IC -13- which is selected by the channel code bits sent to the Control Word via IN3.

000	006	LAI	set CW pattern	005	175*	OUT31	display data
001	362	"110b, ch0, sel, out"		006	006	LAI	set int. enab.
002	006	LAI	define	007	003	003	enable
003	???	???	test data	010	117*	INO	code
004	113*	IN2	write/read UAR/T	011	377	HALT	wait next cycle

* See note in last example re instr. codes

5. Test the input operation of the UAR/T by applying a TTL square wave at 27.5 CPS to the input of channel 1. Using the above program, change word 001 to the octal code "367" (110 baud, ch. 1, select, input.) The data pattern of the 27.5 herz square wave will be interpreted by the UAR/T as four bit-periods per cycle of the wave form, as follows:



The teletype bit length was 7 - in this example, changing to channel 1 increases the data bit length to 8 bits. The UAR/T interpretation of the above square wave should be displayed in the data lamps by the OUT31 as "01100110"

CAUTIONS RE MOS I.C.'S

When you purchase an Intel CPU or a complex MOS device such as the UAR/T chip you should find it comes packed in a special block of conductive foam plastic shorting all pins with respect to high voltage static charges. In insertion and handling of the IC's, be sure to discharge body capacitance to ground. Do the same before approaching the wiring to make changes and alterations. In my own lab I have a rug - and in its typical low humidity winter state, I draw 1/4" sparks to ground after walking any distance! This note was suggested by Gordon French in phone conversation recently.

Also, observe the following precaution when handling and inserting the 40-pin IC parts such as the UAR/T: it is quite easy to mechanically stress the package to the point where it breaks in two - not so bad with a \$13.50 UAR/T but if you buy a \$360 CPU chip of the cadillac variety, it could be heartbreaking. Be sure to apply pressure evenly at all points and avoid letting one corner "get ahead" of the rest by too great a margin.

Tables & summaries of the ECS-6 Design:

Package Summary List for the Serial I/O Controller:

IC No.	Pins	Description	+5V	GND	-12v
1	40	COM2502 UAR/T - Std. Micr. Systems	1	3	2
2	14	8T09 Bus Interface - status	14	7	-
3	14	" " " - status	14	7	-
4	14	" " " - data	14	7	-
5	14	" " " - data	14	7	-
6	14	7404 inverters, misc.	14	7	-
7	14	7404 inverters, misc.	14	7	-
8	14	7437 NAND, high power	14	7	-
9	24	74100 Control word register	24	7	-
10	16	74193 Rate Counter	16	8	-
11	8	NE555 Oscillator	8, 4	1	-
12	16	74153 Input/Ready switches	16	8	-
13	16	74155 Output/Select switches	16	8	-
14	14	7473 JK Flipflops (div by 2)	4	11	-
15	14	7404 Inverters	14	7	-
16	14	7402 nor's	14	7	-

Miscellaneous parts:

R1, R2, R6 to R13 = 1000 ohm $\frac{1}{4}$ W	3 - 16 pin component sockets
R3 = 25K, trimmer potentiometer	3 - 16 pin I/O sockets
R4 = 200 ohms	1 - 40 pin socket
R5 = 120 ohms, 2 watt	1 - 24 pin socket
C1 = .005 mfd	1 - 8 pin socket
C2 = .005 mfd	LED = 10 ma LED indicator
Board, terminals, etc.	

Also required: a total of approximately 10mfd of electrolytic capacitance locally on the power supplies, to ground plus several ceramic (eg: .01) bypasses to ground from power supplies.

<u>I/O-1 List</u>	<u>I/O-2 List</u>	<u>I/O-3 List</u>
1 to 8 = bus 0 to 7	1 = TSO-1	1 = TTY-HI
9 to 16 = addr 0 to 7	2 = TSI-1	2 = TTY-LO current loop
	3 = SELECT-1	3 = TTY-RDY
	4 = RDY-1	4 = TTY-SELECT
<u>I/O-2 List</u>	5 = TSO-2	5 = +5V
14 = IN/OUT	6 = TSI-2	6 = TTY-TSI
15 = Master Reset	7 = SELECT-2	7 = OUT02
	8 = RDY-2	8 = OUT03
	9 = TSO-3	9 = IN02
	10 = TSI-3	10 = IN03
	11 = SELECT-3	14 = GND
	12 = RDY-3	15 = -12 v
	13 = 16-f CLOCK	16 = +5 v

NOTES ON NOTATION:

Some further inputs from readers and other sources, plus some thinking on the subject have led to a conclusion to use octal notation of programs in the ECS magazine for the 8008 computer and its 8-bit microcomputer successors/competitors. The basic arguments for and against hex have not changed - it still is a more compact notation which fits the word size exactly. However I have some new inputs accumulated on the pro-octal side, summarized here ...

1. A reader, Ward Christensen of Dolton Illinois, points out an argument in favor of octal based on character coding schemes. In both IBM's EBCDIC and ASCII, the letters and numbers occupy separate groups of number codes in the set of integers representable in "n" bits. Thus to convert a combined numeric/alpha field (as in HEX data entry) requires special case program logic whereas octal conversion, such as illustrated in STUFFER in this issue (bytes 120 to 136₈), can be done "in line" with no conditional execution by simply masking the low order bits of the character entered. The original hex input routine was not nearly so compact due to special case detection of the A through F case and subtracting off the appropriate bias.
2. Gordon French (more inputs from him in "Miscellaneous Notes" below) points out that hex coding can be justified for long word length machines with byte-multiple word widths because it is a more compact representation of the data than octal. However the short length of the 8008 word ("byte") means only one extra digit is required.
3. I got the Digital Equipment Corporation's rationalizations for using Octal at a recent meeting of the IEEE Computer Society in the Boston area. The meeting topic was the design and architecture of the PDP-11 computer, discussed by two individuals largely responsible for the machine's architecture. Strangely enough, the topic of notation of programs came up in their talk - with the following reasons being used to justify octal: a. Conservatism - its the way minicomputer programs have always been done. b. internal field structure - instructions on the PDP-11 (as with the 8008) have effectively been designed with a 3-bit internal field structure which is symbolically respected if octal is used, but ignored if hex were to be used.
4. I coded up several programs for my system using the hex notation for op codes (figuring the codes as I went along until receiving the latest copy of the 8008 manual from Intel - which lists codes in hex). I found that while hex is fine for reading IBM 360 machine code and dumps, adding and subtracting addresses occasionally to locate origins, etc. - it is not so convenient when hand assembly of code for the 8008 is concerned. This effectively provided the last straw in a reluctant decision to live with Intel's address notation and 8 bit octal data.

Accordingly, I rewrote a memory dump program I had originally written for hexadecimal oriented outputs and have listed all programs in this issue in absolute octal format. The program listings consist of three octal 3-digit columns. The first two columns contain the page (H) and byte (L) address locations, separated by a reverse

slash. The "equal" sign following the address fields separates the address from data at that address - either the octal form of some program data, or an 8008 operation code in octal. In the listings, comments have been typed to the right of the dumped information, and labels have been indicated on the left.

MEMORY DUMP PROGRAM "ELDUMPO":

"ELDUMPO" is an application program which will prove useful to anyone desiring an octal display of information on the teletype, or alternatively, on any other suitable output device if you substitute a different routine for the "TYPE" routine accessed via the RST3 instruction noted with mnemonic TYPE. The program begins execution with entry from location 0, or from the "IMP" interactive manipulator program. (To be listed and described in the next issue.) The absolute machine addresses used in this program as written and dumped are locations 011/000 to 011/235 which are part of a 1-kilobyte RAM design which will be described in the next issue along with IMP. The ELDUMPO program was entered into its memory locations using the "STUFFER" program described on page 16 and 17 below. The listings of ELDUMPO and STUFFER

were achieved using the ECS-5, ECS-6, ECS-4 and ECS-3 designs to drive a teletype interfaced as described on page 7 of this issue. Used teletypes (mine is a model 33) can be picked up at prices in the \$250 to \$500 range depending upon condition and model. At a recent auction sale, I saw teletypes with pin feed platens sold typically for \$350 (including card reader/punch attachments.) In lieu of a teletype, it would be quite reasonable to format the dump essentially as it is performed here, but stuff the data out onto a TV Typewriter or one of the several kit forms currently available - or onto an oscilloscope character generator. The major labels of locations within the program are listed and described below:

START: 011/000 - this is the program entry point. Come here to start off the program by turning off the interrupts while the dump is in operation. (Ignore the keyboard except when testing for end of job cue at the end of a line of dumping.)

ELDUMPO: 011/003 - this address is the main dump loop entry point, and is reached once for each line printed.

END: 011/110 - this address is the place execution transfers to when the data count is exhausted or the keyboard is found to have a non-null character at the end of a line of printing.

STRING: 011/126 - this is a character string data text area containing the end of job message as a length count (17₁₀) followed by bytes of 7-bit ASCII characters for the teletype.

TBYTEOCTAL: 011/150 - this address is a subroutine which prints 3 octal digits accessing the "OCTOUT" routine via an "RST4" instruction.

TSTRING: 011/166 - this address is a subroutine which is entered with H/L pointing to a character string such as STRING, and which types out the string.

SPACES: 011/177 - TYPE "e" spaces and return

TYPEIT: 011/207 - jump here from TYPE (RST3) to do the work of printing

The dump in octal of the ELDUMPO program is listed below with commentary

START:	011\000 = 006	LAI	
	011\001 = 002	interrupt disable code	
	011\002 = 117	OUT0 (IN0) - see ECS-6 p14	
ELDUMPO:	011\003 = 056	LHI	
	011\004 = 000	data RAM	} point to current count of data
	011\005 = 066	LLI	
	011\006 = 025	I(COUNT)	
	011\007 = 327	LCM	} decrement count to zero once per iteration
	011\010 = 021	DCC	
	011\011 = 372	LMC	
	011\012 = 150	JTZ END	} if count reaches zero, then all "n" locations have been dumped so go end it.
	011\013 = 110	L	
	011\014 = 011	H	
	011\015 = 300	NOP	} these three NOP's leave space to put in an extra line feed when TTY acting up!
	011\016 = 300	NOP	
	011\017 = 300	NOP	
	011\020 = 066	LLI	} point to memory address in RAM and load it into B/C
	011\021 = 006	I(MEMADDR)	
	011\022 = 317	LBM	
	011\023 = 060	INL	
	011\024 = 327	LCM	
	011\025 = 020	INC	} increment and save low order address
	011\026 = 372	LMC	
	011\027 = 110	JFZ GOTHOK	} if no low order overflow, then high order is OK as is...
	011\030 = 033	L	
	011\031 = 011	H	
	011\032 = 010	INB - increment high order if required...	
GOTHOK:	011\033 = 061	DCL	} point to high order byte and save it...
	011\034 = 371	LMB	
	011\035 = 006	LAI	} type out a carriage return
	011\036 = 015	"CR"	
	011\037 = 035	TYPE	
	011\040 = 006	LAI	} type out a line feed ...
	011\041 = 012	"LF"	
	011\042 = 035	TYPE	
	011\043 = 046	LEI	} set up number of spaces for call...
	011\044 = 012	10 ₁₀	
	011\045 = 106	CAL SPACES	} and blank out (literally)
	011\046 = 177	L	
	011\047 = 011	H	
	011\050 = 301	LAB - define input argument to print octal byte	
	011\051 = 106	CAL TBYTEOCTAL	} print high order address, (don't byte off more than you can choo however)
	011\052 = 150	L	
	011\053 = 011	H	
	011\054 = 006	LAI	} print separator between H/L
	011\055 = 134	"back slash"	
	011\056 = 035	TYPE	
	011\057 = 302	LAC fetch low order address for printing	

The listing of ELDUMPO continues below after an aside: RAM locations 000/006 and 000/007 are assumed to contain the current address in the dump, initialized to one less than the first address to be printed, upon entry to the program. RAM location 000/025 is assumed to contain a count up to 255₁₀ giving the number of bytes to print initially, and the number remaining thereafter.

	011\060 = 106	CAL TBYTEOCTAL	} go print low order address
	011\061 = 150	L	
	011\062 = 011	H	
	011\063 = 006	LAI	} print a blank, followed by...
	011\064 = 040	" "	
	011\065 = 035	TYPE	
	011\066 = 006	LAI	} print equal sign
	011\067 = 075	"="	
	011\070 = 035	TYPE	
	011\071 = 006	LAI	} print a blank
	011\072 = 040	" "	
	011\073 = 035	TYPE	
	011\074 = 351	LHB	} define data byte address and fetch it from memory
	011\075 = 362	LLC	
	011\076 = 307	LAM	
	011\077 = 106	CAL TBYTEOCTAL	} print data
	011\100 = 150	L	
	011\101 = 011	H	
	011\102 = 115	INI	} read the keyboard at end of line to test for a null code and
	011\103 = 074	CPI	
	011\104 = 377	"null"	
	011\105 = 150	JTZ ELDUMPO	continue if null...
	011\106 = 003	L	} otherwise fall thru to END
	011\107 = 011	H	
END:	011\110 = 056	LHI	
	011\111 = 011	h(String)	} go type string after defining the arguments as H/L
	011\112 = 066	LLI	
	011\113 = 126	l(String)	
	011\114 = 106	CAL TSTRING	} these instructions are put in to reset the keyboard scan state and return to the IMP program operation. For use without IMP, these can be replaced by a HALT or a return to a calling routine.
	011\115 = 166	L	
	011\116 = 011	H	
	011\117 = 056	LHI	}
	011\120 = 000	0	
	011\121 = 066	LLI	
	011\122 = 003	3	}
	011\123 = 076	LMI	
	011\124 = 002	2	
	011\125 = 025	KEYWAIT (RST2)	}
STRING:	011\126 = 021	17 ₁₀ length	
	011\127 = 015	"CR"	
	011\130 = 012	"LF"	} 7-bit ASCII for TTY end of data message
	011\131 = 012	"LF"	
	011\132 = 105	"E"	
	011\133 = 116	"N"	}
	011\134 = 104	"D"	
	011\135 = 040	"blank"	
	011\136 = 007	"bell"	
	011\137 = 000	"null"	

The listing of ELDUMPO continues, with another aside - The data definition of "STRING" is an example of a general form called the "character string." Suppose you want to edit a book, or a magazine for that matter - or a letter to a friend. One great way to do so is to use a string oriented program to store and maintain text as character strings. This basic form will recur in numerous ECS applications.

```

011\140 = 000 "null"
011\141 = 007 "bell"
011\142 = 000 "null"
011\143 = 007 "bell"
011\144 = 000 "bel
011\145 = 000 "null"
011\146 = 007 "bell"
011\147 = 007 "bell"
                                } a few bells and nullsles always
                                } help annunciate the end of
                                } a program's execution...

TBYTE- 011\150 = 340 LEA          save data in E work register
OCTAL: 011\151 = 002 RLC          }
      011\152 = 002 RLC          } shift high order two bits to low
      011\153 = 044 NDI          }
      011\154 = 003 "00000011"-} and mask for 0/1/2/3 digit
      011\155 = 045 OCTOUT      } go OCTOUT your fantasies
      011\156 = 304 LAE
      011\157 = 012 RRC          }
      011\160 = 012 RRC          } fetch saved data and shift middle
      011\161 = 012 RRC          } octal bits to low order
      011\162 = 045 OCTOUT      } and print them
      011\163 = 304 LAE          } - fetch saved data
      011\164 = 045 OCTOUT      } and print the low order data digit
      011\165 = 007 RET
TSTRING: 011\166 = 347 LEM      } here's the text string typing routine -
TSLOOP:  011\167 = 055 NEXTA    } get next address after saving length code.
      011\170 = 307 LAM          } fetch the next byte of string
      011\171 = 035 TYPE        } and go type it on TTY
      011\172 = 041 DCE          } decrement length count in E
      011\173 = 110 JFZ TSLOOP } if any count remains, continue
      011\174 = 167 L           } printing the string
      011\175 = 011 H
      011\176 = 007 RET          } return, if count exhausted...
SPACES:  011\177 = 006 LAI       } come here to print spaces
      011\200 = 040 " "
      011\201 = 035 TYPE        } print the space
      011\202 = 041 DCE          } decrement the space count
      011\203 = 110 JFZ          } if not zero have at it again
      011\204 = 177 L
      011\205 = 011 H
      011\206 = 007 RET
TYPEIT:  011\207 = 330 LDA        } come here to print a character
      011\210 = 006 LAI
      011\211 = 362 "110 baud, ch 0, select, output"
      011\212 = 111 IN3          } go write the TTY output control word
      011\213 = 303 LAD          } save status read
      011\214 = 113 IN2          } restore data and go write
      011\215 = 036 LDI          } make a wait loop to verify done-ness
      011\216 = 077 6310         } 63 times should suffice
TYPEWAIT: 011\217 = 006 LAI      } beginning of wait loop

```

The final segment of ELDUMPO code is printed here... in order to run the program, be sure to reference the section on restart instruction usage located later in this issue. The restart routines TYPE, KEYWAIT, OCTOUT, and NEXTA are all defined in that section and referenced at various locations in ELDUMPO.

011\220 = 362	110 baud, ch 0, select, output
011\221 = 111	IN3
011\222 = 044	NDI
011\223 = 030	mask for TBMT and TEOC bits
011\224 = 074	CPI
011\225 = 030	both bits on and it might be time to try again.
011\226 = 110	JFZ TYPEWAIT
011\227 = 217	L
011\230 = 011	H
011\231 = 031	DCD decrement loop count
011\232 = 110	JFZ TYPEWAIT
011\233 = 217	L
011\234 = 011	H
011\235 = 007	RET return after really done...

Note that a WAIT loop was inserted in this routine as a part of testing the UAR/T. An experiment you may wish to perform is to minimize the number of times through the extra wait loop iterations used to be "really sure" the UAR/T is done. Note also that throughout this code, the input and output instruction operations used are those required for ECS-5's decoder as printed in that article.

MANUAL BOOTSTRAP PROGRAM "STUFFER" :

The listing of "STUFFER" is found on page 17 of this issue. The basic idea is to make a program which essentially delivers the minimal subset of an editor program such as IMP needed to stuff data into locations in the memory of the CPU. This routine takes a total of 52₁₀ bytes of memory, and is amenable to loading via toggle switches - after which use of keyboard and octal coding will make for more efficient loading. The command keystrokes are as follows:

- "N" - ECS 5 code 316g is used to compute the next address and display the content at that address.
- "I" - ECS-5 code 311g is used to insert the last entry at the current, address, increment address and display data at the next location.
- All Else - treat the low order 3 bits as an octal digit shifted into the 8 bit entry register C

To initialize the program's H/L address to memory, put the CPU in single step mode, interrupt and go to location zero, define the H and L constants at locations 140 and 142, single step past location 101 of STUFFER, then go into execution.



STUFFER:

000\100 = 115*	IN2 - read keyboard data after an interrupt.
000\101 = 074	CPI } test for an "N" code on keyboard.
000\102 = 316	"N" }
000\103 = 150	JTZ NEXT - if the "N" is found, jump to the
000\104 = 150	routine which increments H/L and displays
000\105 = 000	the data at the next address...
000\106 = 074	CPI } test for an "I" code on keyboard.
000\107 = 311	"I" }
000\110 = 150	JTZ INSERT - if the "I" is found, jump to the
000\111 = 160	data insertion routine to define memory at
000\112 = 000	H/L from last entry...
000\113 = 074	CPI } test for null character code...
000\114 = 377	"null" }
000\115 = 150	JTZ INIT - to initialize H and L use single
000\116 = 137	step mode, start with a momentary keyboard
000\117 = 000	key stroke
OCTAL: 000\120 = 044	NDI Assume octal, and throw away the
000\121 = 007	"00000111" 5 high order bits with AND...
000\122 = 310	LBA - temporarily save digit in B...
000\123 = 302	LAC - fetch previous entry from C...
000\124 = 002	RLC }
000\125 = 002	RLC } make room for new digit, saving old high order
000\126 = 002	RLC } order information...
000\127 = 044	NDI } Delete previous bits hanging
000\130 = 370	"1111111000" } around in low order digit...
000\131 = 261	ORB - merge in new octal digit from B save...
000\132 = 320	LCA - save new entry in C for next time or use...
000\133 = 177*	OUT30 - entry displayed on the right...
000\134 = 306	LAL - fetch low order address...
000\135 = 175*	OUT31 - current L displayed on the left...
000\136 = 025	KEYWAITⓈ Wait for next key stroke...
INIT: 000\137 = 056	LHI } Come here to define initial value
000\140 = 000	??? } of the address registers H and L
000\141 = 066	LLI } for loading data. Define 140 and
000\142 = 200	??? } 142 manually via bootstrap mode of ECS3
LOOK: 000\143 = 307	LAM - fetch the currently addressed byte
000\144 = 177*	OUT30 - and display it in the right display
000\145 = 306	LAL - fetch the current low order address
000\146 = 175*	OUT31 - and display it in the left display
000\147 = 025	KEYWAITⓈ Wait for next key stroke...
NEXT: 000\150 = 060	INL - increment low order address...
000\151 = 110	JFZ LOOK - go look if not overflow...
000\152 = 143	
000\153 = 000	
000\154 = 050	INH - increment high order if required
000\155 = 104	JMP LOOK - and always go look thereafter
000\156 = 143	
000\157 = 000	
INSERT: 000\160 = 372	LMC - insert the data entry in M(H,L)
000\161 = 104	JMP NEXT - go calculate next address and
000\162 = 150	then display info with LOOK...
000\163 = 000	

END

NOTES:

* Output instruction codes are illustrated for the wiring of the prototype system - see note, p. 14 ECS-6.

Ⓢ KEYWAIT is mnemonic for RST2, used to access the keyboard interrupt wait routine. See page 20 .

PROGRAMMING NOTES: Using Restarts:

This is the first in a series of programming notes on the use of the Intel 8008 instruction set in the context of an ECS system or its equivalent...

The restart instructions of the 8008 are effectively one byte CAL instructions with an implied target address given by the operation code. The implied subroutine address of the instruction is one of the octal locations 000, 010, 020, 030, 040, 050, 060 or 070 in page 0 of memory address space, specified by the middle digit "?" in the operation code "0?5". The fact that only a single octal digit is available for this use immediately limits the application to a maximum of 8 critical (ie: much used) subroutines in a given software load. In a design such as that which was published in ECS-3 and ECS-5 during 1974, one of the restarts is attached to the I/O interrupt structure by using it as the "single instruction jam" which occurs when the CPU is to be interrupted. For the ECS series software, the interrupt structure is at present only used for keyboard interrupts which occur when a key is pressed on the typewriter keyboard of ECS-5. Alternatives to interrupting include use of a priority encoder to pick a restart routine in cases where fast vectoring is required. However, the fact that it is impossible to save the program state of an 8008 at interrupt time (without hardware augmentation that is) leads to the conclusion that the 8008 is best programmed as a "one process" machine at the hardware level - with software polling of interrupt status for most of the fairly slow peripherals likely to be used in a home brew computer context.

With one of the restarts thus taken up by the keyboard interrupt, there are seven instructions RST1 to RST7 which can be used for "something else." What is that "something else." Basically an analysis of your programming of a problem will often show a set of instructions which are used over and over again - a criterion which of itself defines a potential subroutine. Of the set of all possible subroutines a program might use, certain of these subroutines will be executed most often in the static sense - they occur repeatedly throughout the code and occupy a lot of memory space with 3 byte CAL instructions. These frequently coded (but not necessarily frequently executed however) invocations are likely candidates for use of the RST call mechanism in place of the CAL instruction. In making a routine accessible by RST, the amount of memory occupied by the linkages to the routines in question will be decreased, but as is always the case, there is a price in execution time. Instead of taking one 11-state CAL instruction, the time required now includes RST - for a total of 16 CPU states, or 64 microseconds.

The basic use of the RST instruction for a subroutine invocation (where the subroutine is longer than 8 bytes) is illustrated by the following:

In place of CAL XX, use RSTn (where n is an available restart)

At location 000/0n0, code a JMP XX instruction to cause transfer of control to the routine as if CAL had invoked it.

No other changes are required in the subroutine in question, since its execution does not care how it got there

As can be seen in this use of the RST instruction, you will be trading an RST followed by a JMP for a direct CAL - to achieve the same functional effect in a program's operation. Adding up the overhead, two CAL's require 6 bytes, and the total memory required for the same two CAL's implemented via RST is two RST's plus the one JMP at the RST target location. Thus for two or more CAL's to a routine, a net savings tending asymptotically to 2 bytes per CAL will be realized. (Using this mechanism in the degenerate case of a single CAL to a routine will incur a one byte memory overhead penalty!)

In order to successfully use the RST operations it is imperative to structure the first 100₈ bytes of memory address space (which I assume will be RAM) to take advantage of the method. The software supplied in the current and future articles of ECS assumes such a structuring is being used, as described below. The text which follows presents the definitions of presently used RST routines which have been referenced in the listings of ELDUMPO and STUFFER given earlier. Note that most of the restart routines do not occupy a full 8 bytes (the maximum allowable without interfering with the next RST zone of memory.) Thus there is plenty of room for allocation of permanent or temporary RAM usage in the spare bytes left over following the RST routines proper and preceding the next RST location. Of these nominally "spare" locations, several are given permanent system-level allocations in the text below, in particular locations 3 to 7 and 15 to 17₈.

INTERRUPT RESTART:

The first restart zone of memory is that from addresses 000/000₈ to 000/007₈, which are accessed whenever an interrupt occurs in the ECS series designs or their equivalents. The "restart" routine for this case is the simplest - a branch to the prime entry point of the currently executed program. For instance, to run ELDUMPO in this issue, the address of ELDUMPO's START location should be patched in as the target of a JMP instruction's operation, at locations 000/000 to 000/002. The patching is done manually in the bootstrap mode of an ECS style CPU. Manually patching in the address of STUFFER instead will change the keyboard interrupt response to reference that program instead. The design of the IMP program which will be listed and explained in the next issue of this magazine will assume that it is the "primary" program of the system and will be the target of this branch. It will proceed from there to identify the source of the interrupt and return to the appropriate routine with the character it reads. (An element of the return from ELDUMPO to IMP is included in the current listing of ELDUMPO at locations 011/117 to 011/125.) The remainder of the RST0 zone of the 8008 address space is allocated to usages for system parameters as follows:

- 000/003 - IMPSTATE - this is an integer value which contains the current operating state code of the IMP program.
- 000/004 - IMPENTRY - this 8-bit byte contains the last entry interpreted by IMP from keystrokes representing octal digits.
- 000/005 - unassigned
- 000/006 - MEMADDRH - this is the high order of a system level memory pointer used by IMP as well as ELDUMPO

000/007 - MEMADDRL - this is the low order portion of the memory address pointer .

BYTE EXCHANGE RESTART: XCHG

The second restart zone is reserved for prime use as a routine to exchange the two 4-bit halves of a byte of data. The purpose for this routine (which is not accessed by the software listed in this issue) is to provide a simple means of manipulating BCD digits when writing routines for BCD arithmetic. The code of XCHG is as follows:

XCHG:	000/010	002	RLC
	000/011	002	RLC
	000/012	002	RLC
	000/013	002	RLC
	000/014	007	RET

The location 015 in this restart zone is reserved for a JMP instruction op code (104₈) followed by two variable bytes set whenever an indirect form of branching is required. This location (015, symbolically "GPJMP") is used by IMP for example to branch to an appropriate routine in response to keyboard commands stored in a table.

KEYBOARD WAIT RESTART: KEYWAIT

The third restart zone of address space extends from 000/020 to 000/027₈ and is accessed by the RST2 instruction code. The definition of this restart is assumed by both the IMP and ELDUMPO programs to be a routine which sets up the keyboard interrupt hardware then halts pending an interrupt. The routine occupies four of the 8 available bytes in the RST2 zone - the balance from 000/024 to 000/027 are available for use as temporary RAM locations at present, as for example ELDUMPO's use of location 25 to hold the number of lines remaining to be printed.

KEYWAIT:	000/020	006	LAI	load the
	000/021	003	003	interrupt enable code
	000/022	117	IN0	write - resets interrupts
	000/023	377	HALT	- wait for interrupt

PRINT A CHARACTER: TYPE

The fourth restart, RST3, has the purpose of implementing a single character TYPE function via RST mechanisms - where the character to be typed is assumed to be in the A register prior to entry. Its implementation as a RST routine is via the JMP mechanism - the invocation causes a jump to a location within the ELDUMPO routine which performs the actual typing:

TYPE:	000/030	104	JMP	TYPEIT
	000/031	207	L	
	000/032	011	H	

The remainder of this restart zone, addresses 033 to 037, are unallocated to software use at present, and might be used for temporary RAM storage or other purposes which do not conflict with the RST functions.

OCTAL OUTPUT ROUTINE: OCTOUT

The fifth restart, RST4, is used at present only by the ELDUMPO program, and might in fact be redefined for a more important application at some future time. It consists of the code needed to form a single octal digit in 7-bit ASCII code for the teletype, followed by a TYPE instruction (RST3) to print the octal digit in question.

OCTOUT:	000/040	044	NDI
	000/041	007	mask off low order - scrap high
	000/042	064	ORI
	000/043	060	or in the first numeric code
	000/044	035	TYPE and go type result
	000/045	007	RET

As in the previous case, the remainder of this zone is unused at present and might be employed by an application requiring temporary storage in RAM.

NEXT ADDRESS ROUTINE : NEXTA

The sixth restart, RST5, is the final one presented in this set of definitions. It is a routine to perform a double precision incrementation of the address stored in the H and L registers. It is currently used, for example, in the string typing routine of ELDUMPO found at locations 166 to 176 in page 011.

NEXTA:	000/050	060	INL	Increment L
	000/051	013	RFZ	Return if no overflow
	000/052	050	INH	Increment H
	000/053	007	RET	Return always.

The remaining portion of this zone is left undefined at present, for future allocation to permanent or temporary use.

NOTES OF INTEREST TO READERS....

Concerning Circuit Boards:

For the time being I am removing the circuit board products previously announced from this market place. The ECS-2 board is functional but represents an overly complex approach to an audio frequency tape recorder modem and I will shortly be replacing my own versions with simpler designs. For examples of a simpler modem see the current issue of Radio Electronics (February 1975) page 53 for use of the EXAR modem chips. The memory board which I previously announced works fine - in fact it was used to store the program ELDUMPO in this issue - but I have added some options which make the original board obsolete. The details of the 1K memory design will still appear in the next issue as announced.

In view of the fact that I am no longer providing the ECS-2 board, I will agree to refund purchase price to the handful of subscribers who have purchased this item upon receipt of a request for the refund.

Concerning Errata & Program Patches:

Since the previous issue, ECS-6, some further errata in previously published designs have come to my attention. First, two items received from Herman Demons-
toy of Painted Post, N. Y.:

- The output pins of the 2501 memories shown in drawing #5 of ECS-3 are incorrectly identified. Pin 14 (indicated as the D output) should be the \bar{D} output - and vice versa. To fix the drawing, write "14" wherever you see "13" on a 2501 output, and write "13" wherever you see "14" printed next to a 2501 output. The functional impact of this error is a logical inversion of the data stored in memory and read back out.
- The sense of the control lines numbered 134 and 136 on drawing #6 of ECS-3 is incorrect. The correct wiring can be obtained by either adding an inversion with a 7404 section or equivalent, or in the case of line 136, by eliminating the inverter shown in drawign #8 of ECS-3.

Mr. Demons-
toy receives a subscription extension of his subscription by one issue for his identification of these errors and detection of an error in the MEMZAP program which had been previously noted by my brother Peter.

The MEMZAP program listing has an error in it, page 65 of ECS-3. Word 6 of the program should read "371" and not "307" as printed. This error was first identified by Peter Helmers.

The ADD8 subroutine of the extended precision addition routine has several errors. Peter Helmers relays the following routine which works, created by his associate Loren Woody at the University of Rochester:

```

ADD8:      000/100   046   LEI      Set E to 0 (new carry)
           000/101   000
           000/102   361   LLB      Get AVAR
           000/103   307   LAM
           000/104   362   LLC      Point to BVAR
           000/105   207   ADM      Add BVAR
           000/106   100   JFC      ADDCARRY
           000/107   112
           000/110   000
           000/111   040   INE      Set E to 1
ADDCARRY:  000/112   203   ADD      Add old carry
           000/113   100   JFC      SETCARRY
           000/114   117
           000/115   000
           000/116   040   INE      Set E to 1
SETCARRY:  000/117   334   LDE      Save Carry
           000/120   370   LMA      Save Result in BVAR
           000/121   007   RET      and return...

```

Concerning Where To Get Parts (ie: 8008's)

Peter Helmers has just recently completed his version of an 8008 system (at least the initial stages.) As part of his shoestring approach, he did a survey of the various vendors advertising in the Radio Electronics, '73, and Popular Electronics. I will not repeat the vendor addresses here, since all of them advertise regularly in the above magazines. What follows is Peter's summary:

- a) Godbout Electronics was the fastest to reply. They also seemed the most open - especially considering their offer to talk via phone and an explicitly stated guarantee.
- b) Electronic Discount Sales - second best source - reminds me of an operation like yours is in publishing... Had as good a price as Godbout. Did offer guarantee in post card reply.
- c) RGS Electronics - "stuff". Gave an impressive reply, but are obviously trying to sell their kit rather than chip itself since they are way over the "market price" (eg: \$50) of the 8008. My only dislike I guess is their price since on re-reading their reply I would not hesitate to purchase from them.
- d) M&R Enterprises - I wouldn't purchase from them. I am not sure that I believe their story about "savings to the customer" since quantity prices of the 8008 are \$60 leaving them no profit. Also, considering that the Micro System International unit is offered (surplus) from Electronic Discount Sales, I wouldn't be surprised if these two companies bought from the same sources. Also, this company is the only one that did not mention any sort of guarantee.

Peter ended up buying his CPU for \$50 from Godbout and shipped immediately to me in late December. I ran it in my system in place of my regular CPU for about one week and could detect no differences executing a typical set of programs. His latest report is that the CPU is up in and running in his version of the 8008 type system, and operating at a clock rate of about 717Khz with no sweat (my \$120 CPU purchased from Cramer new in 1974 (March) craps out at 500 Khz - sigh!)

Concerning The 8080, ALTAIR and Better Systems.

Since the last issue was mailed, I read of the Altair computer in Popular Electronics. It is a welcome addition to the home microcomputer market place, since the fact that the entrepreneurs at MITS are willing to speculate on market acceptance of such an advanced (and expensive) product is an indication of the growth of the field of avocational computing. First, a note about the PE article - it was fairly obviously prepared by an individual with the following characteristics: little knowledge of computers, a package of materials handed to him with correct data on the device and its capabilities, and boundless enthusiasm. The net vector sum of all these inputs is a set of fairly outrageous statements. From what I have seen of the 8080, and a comparison with products such as the Motorola 6800, I tend to prefer the latter due to its much better documented and designed instruction architecture from a programming and systems standpoint.

On the same theme, a long letter from Gordon French arrived on my desk on the 10th of January or thereabouts (incidentally, composed and printed using an 8008 based text editor running to a teletype.) Several points are worth noting for readers: First, Mr. French lives in Menlo Park California, which is relatively close to the Intel facilities. The following excerpt from his letter concerns a visit he made to the Intel people:

"... I spent 2 hours talking about the Altair 880 with Intel engineers in the Intel Lobby. Gist of many subjects discussed is the following. Intel does not now nor will they ever, surplus out of spec parts to the market. Intel does not desire to cater to the Amateur Computer User to an extent that would mean product design intended for the ACU. They welcome the MITS effort, because it gives them a single source for a large volume sale (with no hassles). They say that the big problem is in instructing the engineer user on how to program the machine (no wonder, since they push hex as the source code!) Most of the people they train have had high level language schooling and find the assembly language tedious, difficult, or utterly impossible. They said there is definitely a market for tutorial texts on assembly language techniques. As for the 8008 or Altair 880 users - they advise the serious user to purchase their Intellec 8080 (\$3840) otherwise they are not interested. The feeling I came away with was that their whole marketing philosophy (understandably) is that they will go after the 100000 piece order. As for future products that they think might get into amateur machines (when I asked about future RAM costs and new easier to use RAM) they say that they sell all the product that they can produce and that this is going to keep the price of RAM up until that situation changes. They also say that they will continue to produce products that are specifically high volume productions. Draw your own conclusions."

With the current going price of the Intel 8008 at \$50, he draws some fairly obvious conclusions regarding amateur computing systems - it will remain extremely economical for some time to orient a system around the 8008 - with the newer 8080 or similar technology processors remaining fairly expensive for some time. Ultimately, the 8080 or other CPU products such as the Motorola 6800 will be coming down in price as production expands - at which point the 8008 will be relegated to the same place in amateur computing as the one tube triode transmitter occupies in amateur radio... a cheap and fairly low power introductory "rig".

Regarding RAM prices, the latest issues of Electronic News and other trade publications are running advertisements indicating a 1K static (2102 or 2602) price of \$4.95 in 1000 quantities. The current small quantity price according to Peter Helmers who just acquired 2K bytes worth is \$7 - new from a regular distributor. Conclusion: if you see a surplus house advertising these devices above the new price, it is suggested you talk 'em down to a reasonable level if possible. The basic systems prices are coming down - the market can only expand as more and more individuals can afford the technology. The parts in question are made by Advanced Micro Devices, whose distributors are Hamilton/Avnet, Cramer and Schweber.

ECS

A MONTHLY MAGAZINE OF IDEAS
FOR THE MICROCOMPUTER EXPERIMENTER

Publisher's Introduction:

This issue of ECS is the second for 1975. It is somewhat different from previous offerings in this series of publications in that it is the first issue to be almost exclusively devoted to software - two fairly large programs for an 8008 computer architecture are listed with commentary. The roster for this issue is...

1. The Interactive Manipulator Program (IMP-1): How can you make your task of loading and changing memory content easier? One way is to use an interactive editing algorithm such as IMP-1. In this section you will find the functional description, annotated listing and examples of the usage of IMP in conjunction with keyboard input, binary (or octal) display outputs - and if you have a character output device such as TTY or TV-Typewriter, - optional links to ELDUMPO (see last issue) are included.

2. Memory Module ECS-7 Hardware Description: As noted, the main theme of this issue is software - but software generally requires memory, so the 1024 byte memory page design is included with this issue. The writeup includes the logic diagram, tables, and notes on expansion to more 1024-byte banks and a very useful feature called "hardware write protect."

3. Memory Test Program (BITCHASER): What distinguishes good bits from bad bits? Hmm! Maybe the good ones are white and the bad ones...??? Not likely! But BITCHASER knows - in the form of a write/read verify of all the words in a selected segment of memory. You can put BITCHASER to work seeking out and counting bad bits - pursuing them relentlessly through the ins and outs of memory address space within a specified set of limits.

4. Programming Notes: Symbol Tables: How can you use the concept of a symbol table - in elementary form - to aid in the writing and debugging of programs in absolute binary? A hint was provided in last year's ECS-5 article. This issue illustrates with IMP and BITCHASER, as explained in this section of the magazine.

The next issue is scheduled for mailing on March 10 1975. The technical content will consist primarily of a new tape interface design along lines suggested in a Radio Electronics article using the XR-210 Modem chip. The article is to include the technical description of the hardware plus software extensions of IMP for the purposes of dumping and restoring data to/from the tape interface.

Carl S. Helmers, Jr.
Publisher February 13 1975

THE INTERACTIVE MANIPULATOR PROGRAM IMP - 1

Functional Description of IMP-1:

IMP is designed to be utilized from a keyboard such as the interface design of ECS-5 previously published, or any suitable typewriter keyboard with appropriate coding changes for the keystrokes. The purpose of the program is to manipulate and examine the content of memory as well as to invoke - and return from - various system utility routines and applications programs. These goals are accomplished using a set of internal RAM data areas sandwiched in among the restart routines described last issue, and a set of definitions for the keyboard buttons used by the program. The basic user data areas of concern are:

IMPENTRY (location 000/004). This byte contains the last data byte defined in octal notation by the keystrokes "0" to "7" (as well as the low order 3 bits of all unused keyboard codes.)

MEMADDR (locations 000/006 and 000/007). These two bytes contain the H (location 6) and L(location 7) portions of a complete memory address. They always maintain the current pointer to any memory location in the computer's memory address space, and are defined using the "H" and "L" keyboard commands.

Memory (arbitrary locations.) The entire memory address space (all $16,384_{10}$ bytes) is potentially accessible to IMP through MEMADDR. Please note however, that while you can address any location with MEMADDR this does not necessarily make the operation meaningful! If you do not have an ECS-7 memory page (or other design hardware) at a given location, writing sends data to the "bit bucket" and reading will result in a null code of 377_8 .

Displays. Left and right 8-bit binary displays are used with IMP for purposes of examining data 16-bits at a time. Although the original program development was done using binary lamps for 16 bits, an easier-to-use display can be made by decoding 6 octal digits with BCD to 7-segment integrated circuits driving LED display digits.

The current set of IMP commands used for manipulation of data is listed beginning on this page. At the end of the program listing/writeup several examples of the use of the commands are included.

IMP COMMAND LIST

"D" - link to ELDUMPO to print data on TTY or send character format octal data to an alternate display device. Use the last MEMADDR to define the starting address (minus one) and use the content of IMPENTRY as the number of bytes to dump.

"E" - examine the content of the two bytes at MEMADDR and MEMADDR + 1.

- "H" - set the H portion of MEMADDR from the last content of IMPENTRY,
- "I" - insert the last content of IMPENTRY in memory at MEMADDR and then increment MEMADDR and display the two bytes at the new MEMADDR and MEMADDR+1.
- "J" - replace the byte at MEMADDR with the last content of IMPENTRY - but do not increment MEMADDR or display the results.
- "K" - clear the value of IMPENTRY to 000_8 .
- "L" - set the L portion of MEMADDR from the last content of IMPENTRY.
- "M" - examine the current content of MEMADDR in the display.
- "N" - increment MEMADDR and display the two bytes at the new MEMADDR and the new MEMADDR + 1.
- "Shift X" - requires two keys to be depressed for safety - cause IMP to transfer execution to the location in MEMADDR after changing IMPSTATE to inhibit all keyboard decoding until return to IMP is desired.

Further commands will be added to this list in the future as IMP is extended in scope to cover such functions as tape interface manipulation, invocation of applications programs and compilers, etc. The basic design of IMP is a simple one - its command interpreter uses single key strokes as the fundamental "token" or particle of its semantics. By looking at the code as listed and explained in this issue, readers will be able to extend the above list for their own purposes by adding to the command table (see below) and supplying appropriate routines.

COMMAND TABLE: IMP is a "table driven" program. This means that the list of commands (keystroke codes) is contained in a table, along with a pointer to the appropriate software routine...

IMPCMDs: 000\354 = 304	}	"D" and L address of "DUMPER"
000\355 = 240		
000\356 = 305	}	"E" and L address of "EXAMINE"
000\357 = 156		
000\360 = 313	}	"K" and L address of "CLEARENTRY"
000\361 = 221		
000\362 = 314	}	"L" and L address of "SETL"
000\363 = 076		
000\364 = 311	}	"I" and L address of "INNEXT"
000\365 = 152		
000\366 = 312	}	"J" and L address of "INSERT"
000\367 = 150		
000\370 = 316	}	"N" and L address of "NEXT"
000\371 = 153		
000\372 = 230	}	"Shift X" and L address of "GOBLO"
000\373 = 200		
000\374 = 310	}	"H" and L address of "SETH"
000\375 = 106		
000\376 = 315	}	"M" and L address of "DISPM"
000\377 = 112		

NOTE: The character codes in this table are taken from ECS-5, page 13.

The actual listing of IMP begins at page address 013_g byte address 000_g with the entry point and the beginning of command decoding...

IMPSTRT:	013\000 = 006	}	How do you deal with noisy layouts? By a software failsafe to turn off interrupt hardware!	
	013\001 = 002			
	013\002 = 117			
	013\003 = 300	}	These "NOP" instructions allow room for a future call to high priority interrupt handlers.	
	013\004 = 300			
	013\005 = 300			
	013\006 = 006	LAI	}	define address of IMPSTATE in H/L using SYM table.
	013\007 = 002	s(IMPSTATE)		
	013\010 = 075	SYM		
	013\011 = 317	LBM	}	fetch IMPSTATE to B
	013\012 = 115	INL		
	013\013 = 011	DCB	}	if IMPSTATE was 1, B now zero so return to application program.
	013\014 = 150	JTZ EXEC		
	013\015 = 230	L		
	013\016 = 013	H	}	if IMPSTATE was 2, B now zero so return to IMP operation.
	013\017 = 011	DCB		
	013\020 = 150	JTZ IMPGO		
	013\021 = 026	L	}	allow for expansion patch to additional checks of IMPSTATE.
	013\022 = 013	H		
	013\023 = 300	NOP		
	013\024 = 300	NOP	}	
	013\025 = 025	KEYWAIT		

When IMP has figured out that it would be a neat thing to do to decode what the key-stroke meant, execution flows to IMPGO to begin a loop through the table.

IMPGO:	013\026 = 310	LBA	}	Save the character input...
	013\027 = 056	LHI		
	013\030 = 000	h(IMPCMDS)	}	Note the lack of use of SYM mechanisms - this is the only place the command table is ever used....
	013\031 = 066	LLI		
	013\032 = 354	l(IMPCMDS)		
IMPDECO:	013\033 = 277	CPM	}	compare and go branch to function if match is found...
	013\034 = 150	JTZ GOTFUNC		
	013\035 = 120	L	}	point to next entry in table
	013\036 = 013	H		
	013\037 = 060	INL	}	move to accumulator to test last time through one plus last table address...
	013\040 = 060	INL		
	013\041 = 306	LAL	}	restore character input... and recycle if more in table...
	013\042 = 074	CPI		
	013\043 = 000	000 _g	}	otherwise no match so fall thru and pretend input is an octal bit pattern in low order...
	013\044 = 301	LAB		
	013\045 = 110	JFZ		
	013\046 = 033	L	}	then go to sleep till woken up again by user...
	013\047 = 013	H		
	013\050 = 106	CAL OCTINTRP	}	
	013\051 = 054	L		
	013\052 = 013	H		
	013\053 = 025	KEYWAIT		

Note how all keystrokes which do not match the table are treated as octal digits by calling OCTINTRP to stuff the low order 3 bits into IMPENTRY...

The octal interpreter routine OCTINTRP is a simple-minded affair which reaches out like an "octalpus" and grabs every keystroke that isn't tied down to a well defined meaning...

OCTINTRP: 013\054 = 044	NDI	} discard high order keystroke data then save the data
013\055 = 007	octal mask	
013\056 = 310	LBA	
013\057 = 006	LAI	} use SYM mechanism to address the old IMPENTRY value... then fetch that value...
013\060 = 004	s(IMPENTRY)	
013\061 = 075	SYM	
013\062 = 307	LAM	
013\063 = 002	RLC	} shift left 3 drops 3 bits into a logical bit bucket - preparing for the AND which erases the bits with a mask for the new high order positions.
013\064 = 002	RLC	
013\065 = 002	RLC	
013\066 = 044	NDI	
013\067 = 370	h.o. mask	
013\070 = 261	ORB	} not a planet in the sky but a logical "OR" of B followed by saving.
013\071 = 370	LMA	
013\072 = 177	OUT30	ECS-5 blooper for OUT30 device. clear accumulator
013\073 = 250	XRA	
013\074 = 175	OUT31	ECS-5 blooper for OUT31 device code.
013\075 = 007	RET	return after displaying entry...



OCTALPUS

Two particular keystrokes which escape the octalpus are the H and L commands, which are serviced by the routines SETL and SETH. These two routines share a common set of code beginning at DISPM with the M command used to simply display the content of MEMADDR.

SETL: 013\076 = 006	LAI	} point to MEMADDR with H/L via SYM mechanism
013\077 = 006	s(MEMADDR)	
013\100 = 075	SYM	
013\101 = 060	INL	increment to point to low order
013\102 = 371	LMB	} got here with IMPENTRY value in B via "SYSSETUP" rtn.
013\103 = 104	JMP DISPM	
013\104 = 112	L	
013\105 = 013	H	
SETH: 013\106 = 006	LAI	} point to MEMADDR here too...
013\107 = 006	s(MEMADDR)	
013\110 = 075	SYM	
013\111 = 371	LMB	load the H value...
DISPM: 013\112 = 006	LAI	} looks redundant, but takes care of all cases - define H/L to display current MEMADDR value.
013\113 = 006	s(MEMADDR)	
013\114 = 075	SYM	
013\115 = 104	JMP EXAMINE	
013\116 = 156	L	
013\117 = 013	H	

Note the continued use of the SYM restart (described later in this issue) to define address pointers from the symbol table. This stretch of code references the address of MEMADDR from three places independently - demonstrating SYM thrice.

When the little IMP has gotten around to figuring out which function key was picked, the next task is to call the appropriate routine. This is accomplished by setting up an indirect jump through location 000/015 using the address found in the command table at the next address after the command code matched by the IMPDECO scanner.

GOTFUNC:	013\120 = 060	INL	} point to next entry in table after key code
	013\121 = 347	LEM	
	013\122 = 036	LDI	*all branches are assumed to be in page 013 ₈
	013\123 = 013		page 013 may have to branch elsewhere if full...
	013\124 = 106	CAL SETJMP	} go define GPJMP address for indirect jump to desired routine.
	013\125 = 212	L	
	013\126 = 013	H	
	013\127 = 106	CAL SYSSETUP	} go define system parameters prior to the indirect jump
	013\130 = 135	L	
	013\131 = 013	H	
	013\132 = 104	JMP GPJMP	} indirect jump to selected routine. squeezed in following XCHG restart.
	013\133 = 015	L	
	013\134 = 000	H	in page 0

The next stretch of code consists of the SYSSETUP subroutine followed by the function routines for memory insertion and examination. The EXAMINE routine is reached as a result of the E, H, L and M commands as well as the more obvious fall thru from the N or I command routines.

SYSSETUP:	013\135 = 066	LLI	} did not use SYM here !
	013\136 = 007	I(MEMADDR + 1)	
	013\137 = 347	LEM	define L parameter
	013\140 = 061	DCL	
	013\141 = 337	LDM	define H parameter
	013\142 = 061	DCL	} point to IMPENTRY
	013\143 = 061	DCL	
	013\144 = 317	LBM	define last IMPENTRY
	013\145 = 364	LLE	} point to memory at MEMADDR
	013\146 = 353	LHD	
	013\147 = 007	RET	end of setups

INSERT:	013\150 = 371	LMB	} insert entry and go to sleep!
	013\151 = 025	KEYWAIT	
INNEX:	013\152 = 371	LMB	insert entry with NO-DOZ
NEXT:	013\153 = 106	CAL INCMA	} fall thru to increment address and store back into MEMADDR
	013\154 = 164	L	
	013\155 = 013	H	
EXAMINE:	013\156 = 307	LAM	EXAMINE is indiscriminate!
	013\157 = 175	OUT31	it will display any data
	013\160 = 055	NEXTA	
	013\161 = 307	LAM	} get a second byte and out it too (sic)
	013\162 = 177	OUT30	
	013\163 = 025	KEYWAIT	and go to sleep after displaying

Now in the context of the IMP program, the simple H/L incrementation provided by the NEXTA restart function will not suffice - the new address obtained by incrementation should be saved in MEMADDR. INCMA calls NEXTA then saves the H/L address in MEMADDR and returns with H/L pointing to the computed address...

INCMA:	013\164 = 055	NEXTA	→ RST compute of next H/L
	013\165 = 346	LEL }	→ Save the address
	013\166 = 335	LDH }	
	013\167 = 006	LAI	
	013\170 = 006	s(MEMADDR)	Computed address to MEMADDR
	013\171 = 075	SYM	would be nice - keeps it around
	013\172 = 373	LMD	Save high order
	013\173 = 060	INL	point to next address
	013\174 = 374	LME	Save low order
	013\175 = 353	LHD }	Redfine the
	013\176 = 364	LLE }	pointer in H/L same as
	013\177 = 007	RET }	MEMADDR & return

When it is desired to bomb out by attempting to execute an unproven new routine, hold your breath, set the new routine's address in MEMADDR with H/L commands, press "shift" and "X" simultaneously and watch your program go blow up...

GOBLO:	013\200 = 106	CAL SETJMP	Come here to go ?
	013\201 = 212	L	First define ? address
	013\202 = 013	H	via subroutine...
	013\203 = 066	LLI	Define IMPSTATE
	013\204 = 003	l(IMPSTATE)	
	013\205 = 076	LMI	Reset IMPSTATE to l for ?
	013\206 = 001	l	
	013\207 = 104	JMP GPJMP	And go to ? defined by MEMADDR
	013\210 = 015	015	via indirect
	013\211 = 000	000	at location 000/015

Actually, the damage of faulty programming can be minimized somewhat when you first attempt to run a program. The mechanism is the "write protect" option on the ECS-7 RAM module design in this issue - simply put the switch in its "protect" position and then execute the routine with knowledge that it can't destroy the software carefully loaded into the RAM module via IMP or STUFFER. However you get to the program, one useful thing is to set up jumps. The routine SETJMP creates the indirect jump address in GPJMPL using the content of D and E for H and L respectively...

SETJMP:	013\212 = 006	LAI	
	013\213 = 010	s(GPJMP)	point to general purpose jump
	013\214 = 075	SYM	via the SYM mechanism
	013\215 = 374	LME	E argument to L of jump address
	013\216 = 060	INL	
	013\217 = 373	LMD	D argument to H of jump address
	013\220 = 007	RET	

Garbage in - garbage out is pure computerworld cliché. However what do you do if you get garbage in to IMPENTRY? Why of course get the garbage out by pressing the "K" key command to activate...

CLEARENTRY:

013\221 = 006	LAI	}	point to IMPENTRY via SYM
013\222 = 004	s(IMPENTRY)		
013\223 = 075	SYM		
013\224 = 375	LMH	}	H known to be 0 so use it to zap entry and go examine...
013\225 = 104	JMP EXAMINE		
013\226 = 156	L		
013\227 = 013	H		

The following is a routine used normally to intercept interrupts from an application program reached from location 013/014 if IMPSTATE is "I". It is designed for a normal transfer to the start of the application program via GPJMP as set by the original "Shift X" execution initiation or a subsequent setting of the application program. As shown, however, it needs a patch at location 013/232 to supply a JFZ and at 013/231 to insert an appropriate interrupt-producing character key code. You could use the "J" command to change it after loading and setting the proper address!

EXEC:	013\230 = 074	CPI	}	On resumption of application program check for escape mechanism Ignore escape until JFZ is used !! - change
	013\231 = 300	"Shft & Ctrl"		
	013\232 = 104	JMP GPJMP		
	013\233 = 015	L	}	013/232 to JFZ if needed.
	013\234 = 000	H		
	013\235 = 076	LMI		
	013\236 = 002	2	}	Reset IMPSTATE on escape... to normal interpreter mode...
	013\237 = 025	KEYWAIT		
				Then wait for user action.

The final routine inserted in page 013 for the preliminary release of IMP as IMP-1 is DUMPER - a short routine to define the data count for ELDUMPO (see last issue, Volume 1 No 1) then branch to the entry point of ELDUMPO. This mechanism was used to activate ELDUMPO for the listings of code found in this issue - the address (minus one) was defined, in MEMADDR, and the data count was left in IMPENTRY. Then the "D" key is pressed thus starting off this sequence...

DUMPER:	013\240 = 066	LLI	}	Define address of ELDUMPO data count word ... via the old fashioned mechanism without SYM - used once here.
	013\241 = 025	l(COUNT)		
	013\242 = 056	LHI		
	013\243 = 000	h(COUNT)	}	Increment copy of ENTRY for ELDUMPO and save it... Then go jump to ELDUMPO with return via the code at locations 011/117 - 011/125 of the last issue....
	013\244 = 010	INB		
	013\245 = 371	LMB		
	013\246 = 104	JMP	}	
	013\247 = 000	L		
	013\250 = 011	H		

In order to illustrate the usage of the IMP program, several worked examples are provided below. The program should be loaded using the STUFFER program found in the last issue, after which the new restart routine SYM described on page must be loaded in locations 70 to 101₈ of page 0 (overlying two bytes of STUFFER locations 100 and 101₈). The branch address in locations 000/001 and 000/002 should be setup to point to IMP (013/000) and the value 002₈ should be loaded in location 3, IMPSTATE, to initialize IMP in its editor mode. It is strongly suggested that when you first try out the IMP program as loaded, you put the memory modules of ECS-7 design in the "write protect" mode - this will prevent inadvertent errors in loading from destroying the information in memory loaded by STUFFER.

Beginning checking out IMP by demonstrating the octal data entry. Press any digit code on the keyboard - "7" - will do. Note that the rightmost 3 lamps of the right hand binary display will go on with "7" - or if you have octal readouts - a 7 will appear in the low order. Press another digit - IMP will shift the previous content left 3 bits or one octal digit, putting the new digit in the low order. The display is filled by pressing a third. In this mode of operation (pressing only octal digit keys on the keyboard) IMP displays only the current IMPENTRY value in the right display and keeps the left display cleared to zeros.

Now, suppose you want to define a full 14-bit address within memory address space. The key sequence is as follows for the address 012/372 ("Intelese" notation.)

0 1 2 H 3 7 2 L

transfer IMPENTRY to L of MEMADDR
after "2", IMPENTRY is complete in display as 372₈
transfer IMPENTRY to H of MEMADDR
after the "2" here, IMPENTRY is complete in display as 012₈

Following the last "L" key stroke, the current memory address of MEMADDR will be displayed in the display, with the H portion at the left, the L portion at right.

Having just defined the address of some byte, suppose you are in the process of loading the BITCHASER program illustrated in this issue. You want to place the code 103₈ in that location. To simply load the addressed location, takes 4 key strokes:

1 0 3 J

transfer IMPENTRY to addressed location with J
complete definition of IMPENTRY for this location

If you want to verify the transfer, the current location can be examined by typing "E" at this point.

Now, suppose you want to define the next three locations following this current location 012/372 from the BITCHASER program code. The following series of keystrokes will point to 012/373 and load consecutive locations...

N 1 1 7 I 1 2 5 I 1 1 6 I

define and insert at 012/375
define and insert at 012/374
define and insert at 012/373

The "N" keystroke of the example at the bottom of page 9 is required to increment the MEMADDR value to point to the next location, 012/373. The series of operations can be continued indefinitely - 3 octal digits followed by "I" - to load as many locations as desired. If, along the way, you lose your place in the program, you can display the current memory location by transferring MEMADDR to the display with the "M" command of the IMP program. Similarly, if you find you made a mistake in entering data for a given word before pressing "I", the entry can be cleared to 0 with "K" or you can simply re-enter 3 more digits.

After completely loading an application program with IMP, you will of course want to execute the program. The program can be invoked from IMP - with automatic return - provided the following conventions are used:

1. To invoke the program, enter its starting address into H/L via the appropriate commands. Then press the "Shift" and "X" keys simultaneously to cause IMP to change state and go to the program.
2. When the invoked program is finished, return to IMP by loading location 3, IMPSTATE with the value "2", then issuing the KEYWAIT restart. An example of this return is illustrated at locations 117 011/117 to 011/125 of the ELDUMPO program published last issue. ELDUMPO was constructed without the SYM mechanism - and this return could be performed with one less byte of explicit code by using SYM to reference IMPSTATE via the symbol table.

If the application program must wait for keyboard interrupt input, issuing KEYWAIT will cause it to halt until a keystroke occurs - after which control will transfer to the location last loaded into GPJMP's address.

MEMORY MODULE ECS-7 HARDWARE DESCRIPTION

The center pages of this month's issue (pages 12 and 13) contain the logic diagram of the ECS-7 memory module design. This module is basically a static 1024-byte "bank" of memory locations implemented with the 2602 or 2102 type of memory chip. (These two numbers are pin-compatible - and subject to various differences in the access time of alternative versions - are also electrically compatible.) The array is interfaced to the bus with the standard ECS series design technology: 8T09 bus drivers for memory outputs, and inverting inputs via 7404 sections to keep the sense of data consistent in this case. Note that there are alternatives to the bus interfaces used throughout this series of designs. One commonly used alternative is to make an "open collector" bus using 7401 (low fanout) or 7438 (high fanout, or drive capability.) Similarly, a non-inverting (but lower power) tristate interface commonly available is the 74125 circuit.

ADDRESS LINES:

One interface socket of the design is used for the 16 address lines used for cycle decode and address selection. The low order 10 bits of addressing are wired directly

to the 10 address input pins (A0 to A9) of the memory IC's. The diagram for clarity does not illustrate a direct connection - see the note provided. The "IN" lines of the 8 memory IC's in the bank are wired to the outputs of corresponding 7404 inverter sections of IC-11- and IC -12-. The "OUT" lines are wired to the data inputs of the 8T09 bus buffer gates. Since this system employs an interface for each bank, the chip select lines are shown wired permanently to ground. If desired, it is possible to create a local "bus extension" for the memory outputs using their tri-state capability and the chip-select inputs to enable one bank of memory at a time with a common 8T09 interface to the CPU bus. To do this, the appropriate bank selection output of the 74154 bank selector would be used to control which set of 8 2102's (2602's) is enabled at a given time.

BANK SELECTION LOGIC:

In its self-contained form as a single 1K by 8 RAM design, the circuit illustrated has all the parts needed to interface to the computer independently. However, if it is desired, the bank selection logic is designed to accomodate sharing of the decode provided by a single 74154. Here is how bank selection word works: the high order address bits of A10 to A13 provide an address of 1 of 16 1024-byte segments of the total 16,384 memory locations of an 8008 processor. The 74154 is always monitoring the address lines and selecting one of 16 banks in the memory address space, whether or not you provide the actual hardware (during I/O the PCW input to one 74154 gate prevents decode, and during interrupt the master enable input to the other 74154 gate also disables decode.) The output of the 74154 appropriate for the RAM bank address is selected by the choice of wiring from the select line to the appropriate bank select pin of the 74154. When the bank is selected, one of the uses of the select signal is to enable write pulses to pass through gate -14- and inverter section -12f- to the memory chips, provided hardware write protect is off. The other use of the select signal of a given memory bank is to enable the CPU-Input signal to control the output interface gate for the bank. Without sharing the bus output buffers of the memory circuit, it is thus possible to share the 74154 logic between several banks simply by omitting a repeat of the 74154 for the additional banks and wiring the select line of the additional banks to the appropriate pin of the 74154 in the first bank.

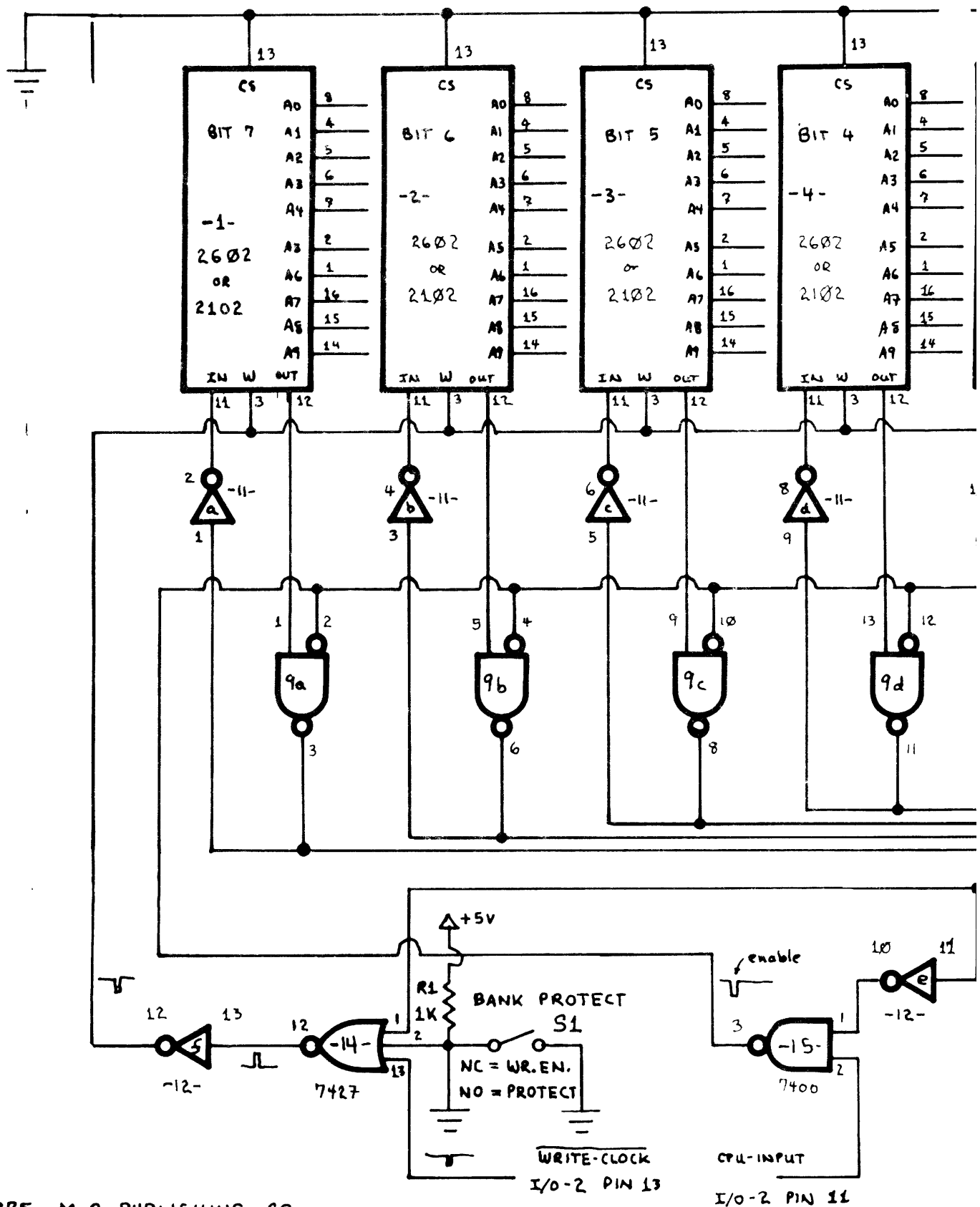
WRITE PROTECT HARDWARE:

The use of a hardware write protection concept in computers has been around for some time. In some computers, it is implemented as a software-controlled bit in the actual hardware of memory, protecting various segments of memory from access and/or modification by programs operating in other segments. In such a context, memory protect features are used to provide a means of minimizing interference between multiple users. Another handy use occurs in the microprocessor context - two very useful goals can be accomplished for your system:

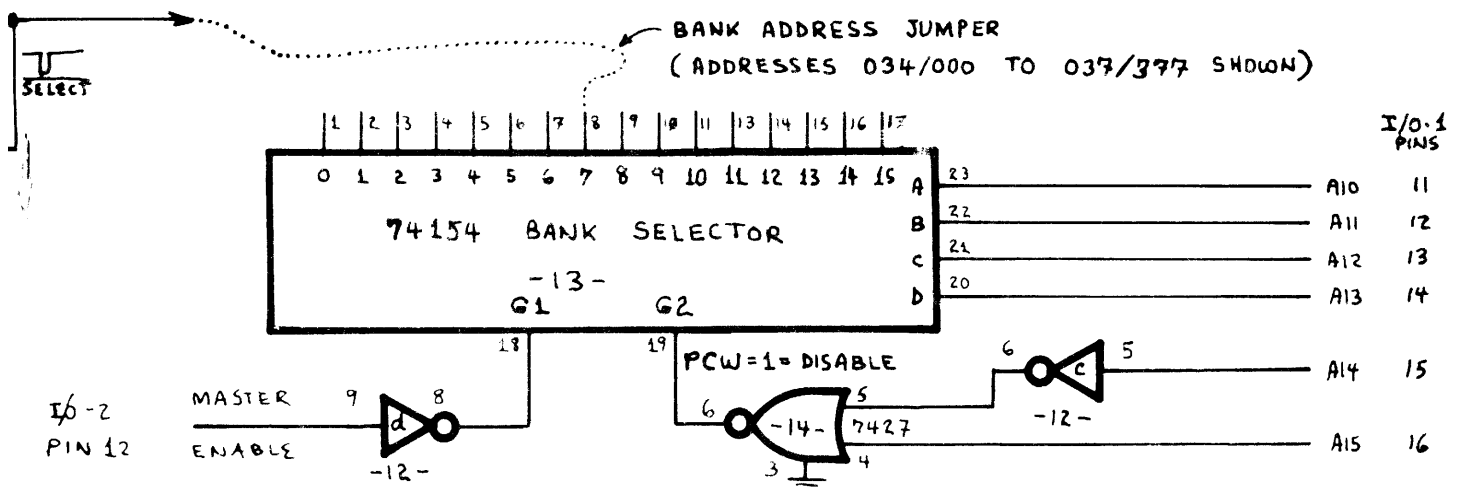
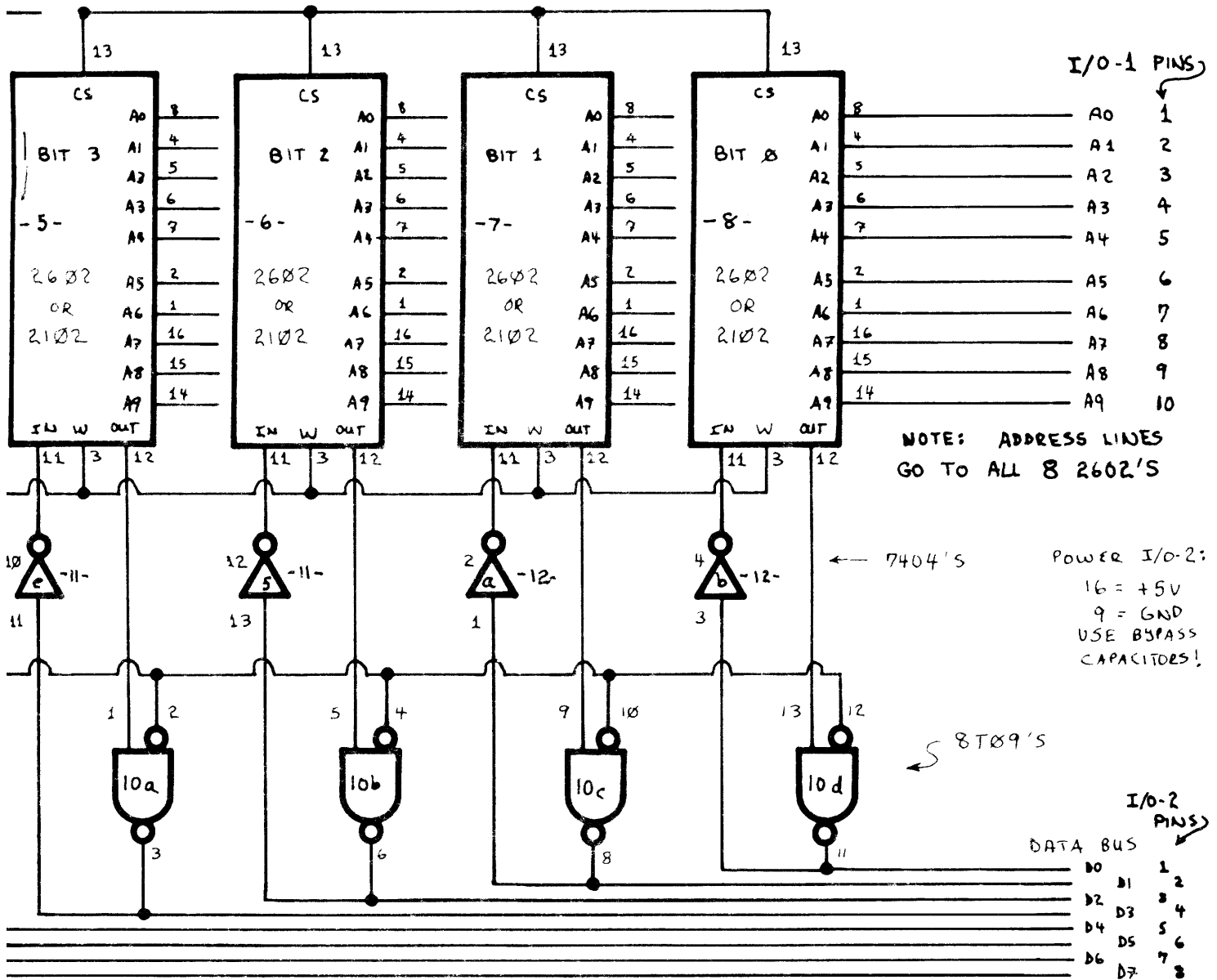
1. You can turn a RAM module into a "pseudo-ROM" by flicking a switch.
2. With the RAM module in protected mode, and with a separate power supply for memory, you can safely disconnect the memory from the computer main frame and maintain your software while changing various aspects of CPU hardware and/or peripherals.

EXPERIMENTER'S COMPUTER SYSTEM

ECS-7 : 1024 BYTE MEMORY



BANK & DECODE by Carl T. Helmers, Jr.



The advantage of the "pseudo-ROM" usage of the protection switch is that you can try out programs initially in a mode which prevents alteration of the program itself - just as if it were an ROM program - yet the ability is retained to switch off the write protection feature and load or alter the program with IMP-1 or its equivalent. The idea for this feature was obtained from the documentation of the Motorola M6800

"EXORciser" program development system's memory module. The idea of independent power supplies for volatile memories I have seen in several sources, such as the power fail logic of the TI minicomputers, HP21MX minicomputer, to mention one or two. (A note of interest - the only reason such a supply is needed is the volatility of semi-conductors. Core memory designs can be made non-volatile, and you will often find a mini with core memory coming out of the factory with some bootstrap software pre-loaded via that technique.) In the logic of ECS-7 as shown, the memory is protected whenever the switch input to IC -14- pin 2 is logical "1" (switch S1 is open.) When the switch is closed, the input to that pin is logical "0", thus enabling the gate whenever select enables it.

MEMORY PRESERVATION PROCEDURES:

Whenever it is desired to maintain programs in the RAM module via a separate power supply, the following procedure is suggested: when powering down the CPU for work:

1. Put the RAM bank in "protect" mode (S1 is open.) Halt the CPU!
 2. Unplug the data bus connector, I/O-2 of the ECS-7 design.
 3. Unplug the address bus connector.
 4. Power down the CPU or the rest of the system for maintenance or other hardware work.
 5. When finished, power up the CPU, put it in a HALT state.
 6. Connect the address bus to the RAM module.
 7. Connect the data bus connector to the RAM module.
 8. If modification of the memory is required, it can now be safely taken out of "protect" mode and used as a normal RAM module.
- "Safely" in this case means with respect to hardware bombing of data.

This procedure was used to great advantage when preparing the hardware and software of the previous issue - IMP was maintained in memory while the CPU was powered down for modifications and tests of the ECS-6 hardware.

ECS-7 PACKAGE SUMMARY

<u>Socket</u>	<u>Pins</u>	<u>Part/Description</u>	<u>5 volts</u>	<u>Ground</u>
1 } to } 8 }	16	2602 or 2102 1024-bit RAM	10	0
9, 10	14	8T09 Tri State Bus Interfaces	14	7
11, 12	14	7404 Inverters	14	7
13	24	74154 Bank Decode	24	12
14	14	7427 3-input NOR	14	7
15	14	7400 (1 section used)	14	7
16	16	I/O-1, address A0 to A15	-	-
17	16	I/O-2, data bus, power, misc.	16	9

MEMORY TEST PROGRAM (BITCHASER):

Once you have constructed the basic RAM module of ECS-7, you can test the memory in a random and un-systematic manner by using STUFFER to load IMP, then using IMP to write in and read the content of various locations (hopefully outside of IMP itself!) The purpose of BITCHASER is to provide a systematic method of testing all the memory locations within a specified address range - in this initial version by reading and writing a fixed pattern set prior to starting the program. The program is set up to periodically look at the ECS-5 keyboard and respond to any key by typing a summary message on the teletype:

```
ERRORS=0000033130
COUNT=0000033130
```

Note: message obtained by looking at non-existent (always bad) memory!

or a suitable substitute for the teletype such as a CRT terminal or TV Typewriter - in which case the Type routine of ELDUMPO would have to be modified. The program takes advantage of the restart routines used with ELDUMPO and IMP. BITCHASER also employs the SYM restart mechanism for table lookups, as is described later in this issue. BITCHASER is shown loaded in page 012₈ of memory, and all address constants for jumps reflect this location. The program begins execution by a short loop to clear out the ECOUNT and TCOUNT data (error count and total cycle count respectively) which are located in RAM page 0 at locations 200 to 207...

BITCHASER:

	012\000 = 006	LAI	
	012\001 = 012	s(ECOUNT)	} point to ECOUNT/TCOUNT
	012\002 = 075	SYM	
	012\003 = 250	XRA	address via SYM lookup
	012\004 = 016	LBI	} clear accumulator
	012\005 = 010	010	
			load loop count
MILOOP:	012\006 = 370	LMA	clear a word from accum.
	012\007 = 060	INL	point to next address
	012\010 = 011	DCB	} decrement count - this loop could be made more efficient - see if you can figure out how!
	012\011 = 110	JTZ MILOOP:	
	012\012 = 006	L	
	012\013 = 012	H	

Following the initialization of counts, the actual work of BITCHASE begins with the start of the major memory test loop at BIGMLOOP...

BIGMLOOP:	012\014 = 006	LAI	} software failsafe to turn off interrupts repeatedly when haywire prototype is victimized by TTL noise immunity problems.
	012\015 = 002	2	
	012\016 = 117	IN0	
	012\017 = 006	LAI	} use SYM to point to STRTADDR of tested region.
	012\020 = 032	s(STRTADDR)	
	012\021 = 075	SYM	} fetch H of STRTADDR then point to L fetch L of STRTADDR
	012\022 = 317	LBM	
	012\023 = 060	INL	
	012\024 = 327	LCM	} point to current CURRENTADR
	012\025 = 006	LAI	
	012\026 = 034	s(CURRENTADR)	
	012\027 = 075	SYM	

012\030 = 371	LMB	}	define H of CURRENTADR
012\031 = 060	INL		then point to L
012\032 = 372	LMC		and define L of CURRENTADR
012\033 = 115	INI	}	read keyboard
012\034 = 074	CPI		once per cycle
012\035 = 377	null	}	and test for not null
012\036 = 112	CFZ REPORT		calling the report typer if so
012\037 = 165	L		
012\040 = 012	H		

Now, if one had a high order language (such as PL/1, FORTRAN, etc.) for the 8008, the code shown above at locations 012/017 to 012/040 is what the compiler would generate for a statement of the following form (ala PL/1):

STRTADDR = CURRENTADR;

The reason for such languages for computers of course is to economize programmer time in generating programs - as you can see by comparison to the dump form.

The program continues with an inner loop - LITLOOP... - to test and increment the current addresses with a test for end of range conditions.

LITLOOP: 012\041 = 006	LAI	}	get address of test pattern via SYM mechanism and get the pattern to "b" reg
012\042 = 040	s(PATTERN)		
012\043 = 075	SYM		
012\044 = 317	LBM		
012\045 = 006	LAI	}	then point to current address value also via SYM
012\046 = 034	s(CURRENTADR)		
012\047 = 075	SYM		
012\050 = 327	LCM	}	point to current address in H/L
012\051 = 060	INL		
012\052 = 367	LLM		
012\053 = 352	LHC		
012\054 = 371	LMB		test write to memory
012\055 = 301	LAB		
012\056 = 277	CPM		followed by compare to check it
012\057 = 112	CFZ POSTERR	}	record the error for POSTERRity
012\060 = 127	L		
012\061 = 012	H		
012\062 = 106	CAL TALLY	}	and keep track of number of cycles for comparison to error count...
012\063 = 134	L		
012\064 = 012	H		

The inner loop continues on the next page, with a short section of code which is the equivalent (at addresses 012/065 to 012/102) to what a "high order language" for computers would specify as:

CURRENTADR = CURRENTADR + 1;

Again, note the amount of code which can be implied by a short and succinct functional notation - in this case the concept "add one to current address" denoted above is implemented at a low level by the detail of 14₁₀ 8008 machine instructions...

	012\065 = 006	LAI	}	set up current address pointer
	012\066 = 034	s(CURRENTADR)		
	012\067 = 075	SYM	}	fetch H of current address
	012\070 = 317	LBM		
	012\071 = 060	INL		
	012\072 = 327	LCM	}	fetch L of current address
	012\073 = 020	INC		
	012\074 = 110	JFZ NOHO	}	and test overflow...
	012\075 = 100	L		
	012\076 = 012	H	}	increment possibly
	012\077 = 010	INB		
NOHO:	012\100 = 372	LMC	}	save new low order address
	012\101 = 061	DCL		
	012\102 = 371	LMB		
				then save new high order address

After incrementing the current address of a location under test, the next task for BITCHASER's inner loop is to check for end of address range...

	012\103 = 006	LAI	}	point to end address value
	012\104 = 036	s(ENDADDR)		
	012\105 = 075	SYM	}	via SYM for comparison
	012\106 = 307	LAM		
	012\107 = 271	CPB	}	fetch H of end address and compare to current address
	012\110 = 110	JFZ LITTLOOP		
	012\111 = 041	L	}	keep going if not equal in H
	012\112 = 012	H		
	012\113 = 060	INL	}	if H portions equal, check L get L part of end address value and compare to L of current
	012\114 = 307	LAM		
	012\115 = 272	CPC	}	keep going if not equal
	012\116 = 110	JFZ LITTLOOP		
	012\117 = 041	L	}	
	012\120 = 012	H		
	012\121 = 300	NOP	}	These NOP's are inserted to allow for a future change - a CAL instruction to invoke a routine to change the test pattern...
	012\122 = 300	NOP		
	012\123 = 300	NOP		
	012\124 = 104	JMP CHECKEND	}	the end of execution check could have been put in line without this jump...
	012\125 = 332	L		
	012\126 = 012	H		

The above code completes the main routine of BITCHASER (with the exception of the short "CHECKEND" routine at 332 to 345 in page 012.) Now the next object of attention is the set of subroutines called from this main routine. The code starts with the multiple-entry-point POSTERR/TALLY routine. The "entry point" of a subroutine is a place where it can potentially begin. This routine has entry points to define the SYM pointer of the data to be incremented as a 32-bit number, called as TALLY and POSTERR - then with the symbol defined, common code is used to do the work.

POSTERR:	012\127 = 006	LAI	}	point to error count 32 bit number
	012\130 = 012	s(ECOUNT)		
	012\131 = 104	JMP I4B	}	then jump around alternate entry
	012\132 = 136	L		
	012\133 = 012	H		

TALLY:	012\134 = 006	LAI	} point to total count 32-bit number
	012\135 = 014	s(TCOUNT)	
I4B:	012\136 = 075	SYM	- and here the common 32 bit increment code starts
	012\137 = 060	INL	} in order to start from low order with a pointer to high order, must change addr.
	012\140 = 060	INL	
	012\141 = 060	INL	
	012\142 = 317	LBM	} fetch low order byte
	012\143 = 010	INB	
	012\144 = 371	LMB	} increment it
	012\145 = 013	RFZ	
	012\146 = 061	DCL	} and of course, save it... and return if no overflow...
	012\147 = 317	LBM	
	012\150 = 010	DCB	} ok - overflow, so point to next higher byte, fetch it
	012\151 = 371	LMB	
	012\152 = 013	RFZ	} and decrement it, and save it too, and also return if no overflow...
	012\153 = 061	DCL	
	012\154 = 317	LBM	} this count is getting large! go to next higher order byte, fetch it, decrement it, and save it too...
	012\155 = 010	DCB	
	012\156 = 371	LMB	
	012\157 = 013	RFZ	} and return if no overflow...
	012\160 = 061	DCL	
	012\161 = 317	LBM	} last resort - the highorder byte is fetched, is incremented, is saved,
	012\162 = 010	INB	
	012\163 = 371	LMB	
	012\164 = 007	RET	and you're out of luck if you over flow 4.29 billion!!!!

The next subroutine listed is a REPORT generator which prints the two counts shown on page 15 as 10-digit octal numbers. The routine has a branch in the middle of it to a patch due to a faulty memory location - ultimately caused by purchase the author made from a fly-by-night distributor called "Electronic Component Sales" perpetrated by a character named "Pete Kay" last September.

REPORT:	012\165 = 006	LAI	} point to the address of a character string message text via SYM
	012\166 = 022	s(STRING1)	
	012\167 = 075	SYM	
	012\170 = 104	JMP FLYBYNITE	- when you buy memories from a
	012\171 = 365	L	flybynight distributor who flies, you some-
	012\172 = 013	H	times have to branch around bad locations.

FLYBYNITE:	013\365 = 106	CAL TSTRING	} call the character string type routine found in ELDUMPO of last issue...
	013\366 = 166	L	
	013\367 = 011	H	
	013\370 = 006	LAI	} establish address of error count by defining symbol and then calling routine to print it as 10 octal digits (ignore 2 high order bits...)
	013\371 = 012	s(ECOUNT)	
	013\372 = 106	CAL TOCT10	
	013\373 = 214	L	
	013\374 = 012	H	
	013\375 = 104	JMP FLYBACK	
	013\376 = 200	L	
	013\377 = 012	H	


FLYBACK:	012\200 = 006	LAI	} point to second message string as address in H/L
	012\201 = 024	s(STRING2)	
	012\202 = 075	SYM	
	012\203 = 106	CAL TSTRING	} call the character string type routine found in ELDUMPO
	012\204 = 166	L	
	012\205 = 011	H	
	012\206 = 006	LAI	} point to symbol of total count(sic) and call the 10-digit octal printer
	012\207 = 014	s(TCOUNT)	
	012\210 = 106	CAL TOCT10	
	012\211 = 214	L	
	012\212 = 012	H	
	012\213 = 007	RET	finally, return from report....

The next subroutine is called "TOCT10" and is responsible for the output of a 10-digit octal integer representation of the low order 30 bits of the 32 bit count passed as a symbol in the accumulator. The first thing this routine does is to lookup the argument symbol and copy its data (all four bytes) to a working copy used for shifting the information 3 bits at a time to generate octal quanta.

TOCT10:	012\214 = 075	SYM - look up argument symbol left in A	
	012\215 = 317	LBM	} copy argument into registers first,
	012\216 = 060	INL	
	012\217 = 327	LCM	
	012\220 = 060	INL	
	012\221 = 337	LDM	
	012\222 = 060	INL	
	012\223 = 347	LEM	
	012\224 = 006	LAI	} point to work output area
	012\225 = 026	s(WKOUT)	
	012\226 = 075	SYM	
	012\227 = 371	LMB	} then copy argument to the work area...
	012\230 = 060	INL	
	012\231 = 372	LMC	
	012\232 = 060	INL	
	012\233 = 373	LMD	
	012\234 = 060	INL	
	012\235 = 374	LME	
	012\236 = 006	LAI	} point to loop index "I" used for octal digit location purposes, then definition of initial 2-bit discard.
	012\237 = 016	s(I)	
	012\240 = 075	SYM	
	012\241 = 076	LMI	} point to loop index "J" used to count bits, and load its initial value for 10 octal digits of shifting.
	012\242 = 002	2	
	012\243 = 006	LAI	
	012\244 = 020	s(J)	
	012\245 = 075	SYM	
	012\246 = 076	LMI	
	012\247 = 036	30 ₁₀	

At the start, WKOUT a has the following for the two counts typed on page 15.

	0	0	0	0	0	3	3	1	3	0	octal	
	00	000	000	000	000	000	011	011	001	011	000	binary

high order bits discarded.  BIT BUCKET

FIRST BYTE FOURTH BYTE

After initialization, TOCT10 enters the loop on the next page, shifting left (see above) three bits at a time, printing octal digits from high order to low order left to right. The two high order bits are discarded without printing due to the initialization.

The print loop shifts WKOUT left one bit at a time, and every third bit will look at the current high order of WKOUT and print an octal digit...

```

TOCT10L:  012\250 = 006  LAI
          012\251 = 026  s(WKOUT) } point to work register again...
          012\252 = 106  CAL SHL4B } shift it left four bytes
          012\253 = 312  L      } with the subroutine...
          012\254 = 012  H
          012\255 = 006  LAI
          012\256 = 016  s(I) } point to I for print test
          012\257 = 075  SYM
          012\260 = 317  LBM } fetch I
          012\261 = 011  DCB } and decrement I
          012\262 = 371  LMB } and save it again...
          012\263 = 110  JFZ TEND - if zero, is untrue, go test end
          012\264 = 300  L
          012\265 = 012  H
          012\266 = 076  LIMI } set I to 3 for next minor cycle
          012\267 = 003  3
          012\270 = 006  LAI
          012\271 = 026  s(WKOUT) } point to work register again...
          012\272 = 075  SYM
          012\273 = 307  LAM } in order to fit fetch high order after shifts
          012\274 = 002  RLC
          012\275 = 002  RLC } ... and rotate high order bits to low
          012\276 = 002  RLC } order position...
          012\277 = 045  OCTOUT then go OCTOUT as was done in
                               ELDUMPO...

```

When it is time to end, this is indicated by exhaustion of the count stored in the variable "J" (do not confuse with the keystroke designation in IMP).

```

TEND:     012\300 = 006  LAI
          012\301 = 020  s(J) } point to variable "J" (not the command code)
          012\302 = 075  SYM
          012\303 = 317  LBM } fetch J,
          012\304 = 011  DCB } decrement J
          012\305 = 371  LMB } and store J value back in J...
          012\306 = 110  JFZ TOCT10L
          012\307 = 250  L      keep going with print loop till done
          012\310 = 012  H
          012\311 = 007  RET and of course back to caller when done...

```

Then the 32-bit multiple precision shift routine, left shifting 4 bytes one position...

```

SHL4B:    012\312 = 075  SYM - go look up the argument of shift
          012\313 = 060  INL
          012\314 = 060  INL } got to point to low order before shifts...
          012\315 = 060  INL
          012\316 = 250  XRA clear accumulator and flags
          012\317 = 026  LCI
          012\320 = 004  4 } define loop count

SHL4BL:   012\321 = 307  LAM fetch current byte,
          012\322 = 022  RAL rotate old carry in, bit 7 to carry
          012\323 = 370  LMA and save the shifted bytes...
          012\324 = 061  DCL decrement the index...
          012\325 = 021  DCC decrement the loop count...
          012\326 = 110  JFZ SHL4BL:
          012\327 = 321  L and continue till count is exhausted...
          012\330 = 012  H
          012\331 = 007  RET then return to caller...

```


This shift routine (p 20) assumes that the argument is a 4-byte string pointed to via a symbol passed in the accumulator, looked up immediately on entry. Taking into account the symbol table lookup time and the sequence of instructions executed by this routine, at a 500Khz clock rate, it takes 262 cycles x 4 us = 1.048 milliseconds per single bit shift. For individuals with delusions of grandeur, note that to accomplish what an IBM 360 does in one "SLL" instruction - an "n" bit shift - the would-be emulator will require 1.048 n milliseconds! (Only about 3 orders of magnitude slower - depending on your choice of comparison model.)

The actual code of BITCHASER completes with the CHECKEND routine, added as an afterthought to cause the program to return to IMP with an "E" key on the keyboard.

CHECKEND:	012\332 = 115	INI	→ read display (modified ECS-5 code)
	012\333 = 074	CPI	} → check for end of memory test...
	012\334 = 305	"E"	
	012\335 = 110	JFZ	BIGMLOOP
	012\336 = 014	L	and continue until done...
	012\337 = 012	H	
	012\340 = 006	LAI	} → point to IMPSTATE
	012\341 = 002	s(IMPSTATE)	
	012\342 = 075	SYM	
	012\343 = 076	LMI	} → set IMPSTATE to 2
	012\344 = 002	2	
	012\345 = 025	KEYWAIT	and wait for IMP actions...

The remainder of page 012 is filled up with the data definitions of the two text strings printed by BITCHASER (see illustration on page 15 of the results.)

STRING1:	012\346 = 015	"length"	STRING2:	012\364 = 013	"length"
	012\347 = 007	"bell"		012\365 = 015	"cr"
	012\350 = 012	"lf"		012\366 = 007	"bell"
	012\351 = 007	"bell"		012\367 = 012	"lf"
	012\352 = 012	"lf"		012\370 = 040	" "
	012\353 = 015	"cr"		012\371 = 040	" "
	012\354 = 040	" "		012\372 = 103	"C"
	012\355 = 105	"E"		012\373 = 117	"O"
	012\356 = 122	"R"		012\374 = 125	"U"
	012\357 = 122	"R"		012\375 = 116	"N"
	012\360 = 117	"O"		012\376 = 124	"T"
	012\361 = 122	"R"		012\377 = 075	"="
	012\362 = 123	"S"			
	012\363 = 075	"="			

The RAM locations 200 to 225₈ in page 0 are used to store the data values of BITCHASER, pointed to by symbols stored at locations 312 to 341 in the symbol table of the RAM page 0. These work areas are as follows:

200 - 203	ECOUNT	215 - 216	CURRENTADR
204 - 207	TCOUNT	217 - 220	ENDADDR
210	I	221	PATTERN
211	J	222 - 225	WKOUT
213 - 214	STRTADR		

STRTADR and ENDADDR should be loaded with IMP prior to starting BITCHASER, in order to define the limits of the test.

PROGRAMMING NOTES:Symbol Tables:

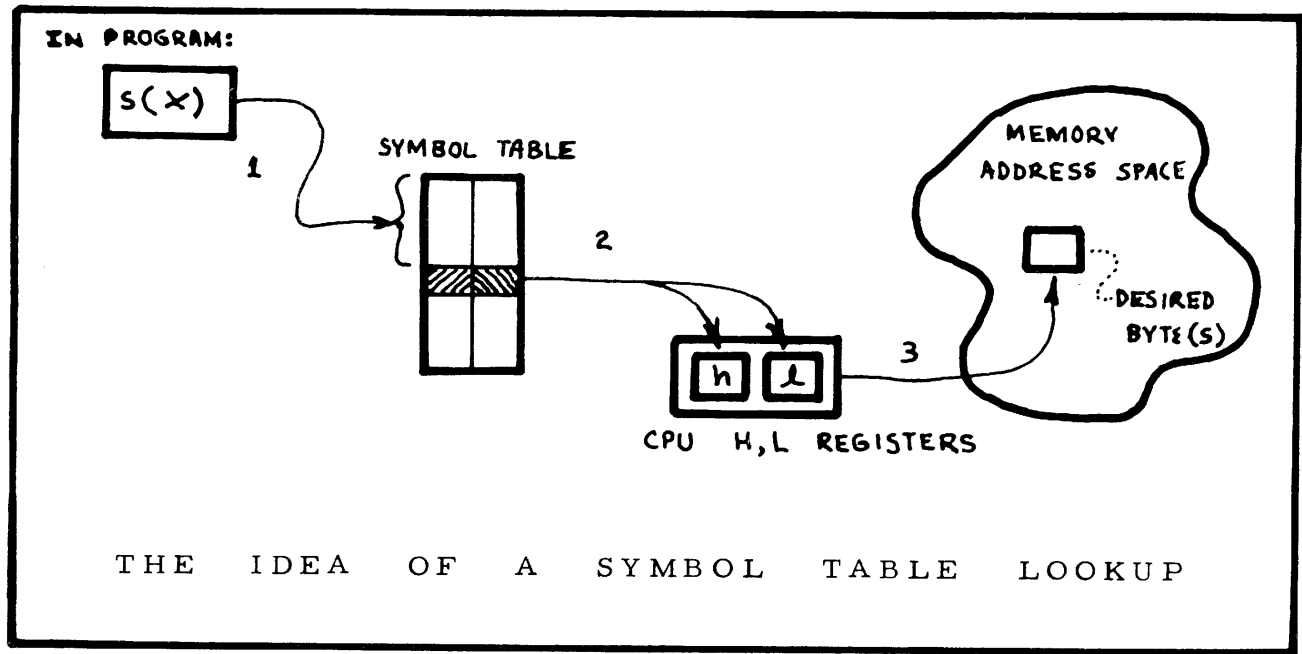
This is the second in a series of programming notes on the use of the Intel 8008 instruction set in the context of an ECS system or its equivalent...

The programs IMP and BITCHASER which are listed and explained in this issue of ECS make use of a rudimentary form of symbol table mechanism implemented via an RST7 instruction (octal 075, noted mnemonically as SYM). The purpose of the symbol table - used at run time - is to make up for a lack of an assembler or high order language compiler's "address resolution" functions. It achieves this purpose by concentrating detailed address determinations as much as possible in a single run time mechanism. "Address resolution" in this context means the definition of the content of the memory address pointer registers H and L of an 8008 CPU. Because the symbol table mechanism uses a run time lookup to compute addresses of data, its speed of access to the data will be lower than directly defined references. For extensively used variables, there will be an improvement in memory utilization efficiency approaching one byte per usage when compared against direct definition of H and L with the LHI and LLI instructions. Thus the usual speed versus memory tradeoff in this case becomes the 57 cycles (.228 ms) versus 16 cycles (.064 ms) of SYM compared to direct definition - with the average savings of one byte in four for the SYM usage applied to a large number of frequently used variables.

But the considerations are not quite as simple as the comparison of speed and memory utilization requirements. The real advantage of the symbol table approach comes in when you consider the problem of compiling and changing code for a program in absolute machine language using paper and pencil. (If you have a compiler or assembler with hardware to support it, the symbol table concept is still used - but the lookups are usually done once at compile time to generate the fastest possible run time code.) As noted in ECS-5, if it is desired to relocate the memory allocation of a widely used variable - say MEMADDR of IMP for example - you (or a suitable utility program) would have to adjust every instance where the address in question was defined and used. For the 8008 instruction set, this is further complicated by the fact that you have to consider the definition of two independent registers, H and L, required for addressing. (A better computer design such as the Motorola M6800 can use a single instruction 16-bit immediate operation for this purpose in loading index addresses.) For an extensive hand-compiled application program of 1000 bytes or more in the typical home-brew microprocessor system, such adjustments and relocations could be quite time consuming.

If the addressing is concentrated in one known place - the symbol table - then you only have to change the pointers in the symbol table in order to automatically change all references to the data made throughout the program. The mechanism gives you a form of "leverage" in control of your program design which can be quite powerfully used as the designs evolve. In the example of IMP, if I wanted to change the MEMADDR location from address 000/006 to some other place, it would only be necessary for me to change the symbol "6" entry of the symbol table at locations 306 and 307 (see below.) To achieve this power, however, the SYM mechanism has to be used 100% for all variables potentially subject to such relocation.

The diagram below depicts the basic idea of the symbol table as used in the IMP and BITCHASER software of this issue - and as will be used for the most part in subsequent ECS software designs. for the 8008.



In use in a program design, all symbolic references to data are made in three steps corresponding to the three numbered arrows of the diagram:

1. Define the symbol as a value in the accumulator, eg. with an LAI instruction - as for example at locations 221 and 222 of page 13 in IMP.

2. Call the symbol table lookup function with an RST7 instruction, noted mnemonically as SYM in the listings of ECS software. This invokes the 10₁₀ byte SYM routine:

000/ 070	056	LHI	
000/ 071	000	h(SYMBOLS)	} → define symbol table page
000/ 072	004	ADI	
000/ 073	300	l(SYMBOLS)	} → add starting address to the symbol giving table address
000/ 074	360	LLA	
000/ 075	307	LAM	→ which is moved to L pointer
000/ 076	060	INL	→ get H part of symbol address
000/ 077	367	LLM	→ point to L part of address
000/ 100	350	LHA	→ redefine L as symbol's content
000/ 101	007	RET	→ and move H part to H
			finally return with H/L pointing.

3. On return from the SYM function, use the H/L pointers of the CPU to address the data which is to be manipulated by the program you are writing.

In creating symbols, remember that every even numbered address offset is a potentially legal symbol - but that if the start of the symbol table is in the middle of a page of memory space as in this case, there will be a maximum size to the table less than a potential 128 table entries in a full page table. The notation "s(x)" is used to represent the value of the symbol associated with mnemonic "x".

The symbol table required by the IMP and BITCHASER software in this issue is printed below, and is loaded in locations 300 to 341 of page 0. The particular location of the symbol table is arbitrary subject to the following constraint: since the SYM routine uses an 8-bit addition to compute addresses, and makes no data validity checks, the symbol table must be so located as to avoid crossing a page boundary in the 8008 memory address space. This means that the maximum number of symbols possible with a given symbol table is one page full or 128 symbols. (Two bytes are required for each symbol definition.) By altering the constants at locations 071 and 073 in page 0 (the SYM routine) the origin of the symbol table can be placed at any point in memory - and such alteration if done carefully might be done under program control.

```

SYMBOLS: 000\300 = 000 } Just for kicks, the symbol table can point to itself
          000\301 = 300 } as well as anywhere else...
          000\302 = 000 } Symbol "002" is IMPSTATE
          000\303 = 003 }
          000\304 = 000 } Symbol "004" is IMPENTRY
          000\305 = 004 }
          000\306 = 000 } Symbol "006" is MEMADDR
          000\307 = 006 }
          000\310 = 000 } Symbol "010" is GPJMPAL
          000\311 = 016 }

```

The following are additional symbol definitions used by BITCHASER...

```

000\312 = 000 } Symbol "012" is ECOUNT
000\313 = 200 }
000\314 = 000 } Symbol "014" is TCOUNT
000\315 = 204 }
000\316 = 000 } Symbol "016" is I
000\317 = 210 }
000\320 = 000 } Symbol "020" is J
000\321 = 211 }
000\322 = 012 } Symbol "022" points to STRING1
000\323 = 346 }
000\324 = 012 } Symbol "024" points to STRING2
000\325 = 364 }
000\326 = 000 } Symbol "026" points to WKOUT
000\327 = 222 }
000\330 = 000 } This symbol unused at present...
000\331 = 212 }
000\332 = 000 } Symbol "032" points to STRTADDR
000\333 = 213 }
000\334 = 000 } Symbol "034" points to CURRENTADR
000\335 = 215 }
000\336 = 000 } Symbol "036" points to ENDADDR
000\337 = 217 }
000\340 = 000 } Symbol "040" points to PATTERN
000\341 = 221 }

```

ECS

THE MONTHLY MAGAZINE OF IDEAS
FOR THE MICROCOMPUTER EXPERIMENTER

Publisher's Introduction:

This March 1975 issue of ECS provides a new modem design to replace the ECS-2 design published in 1974. This modem, given the hardware designation "ECS-8" as the next in a series of plans, will provide the typical Experimenter's Computer System with the logical equivalent of a paper tape input/output facility - but implemented on re-usable magnetic tape media (eg: cassettes) with data rates up to 1210 baud. The March issue is exclusively devoted to this hardware design and its software implications, including...

1. BiDirectional FSK Modem Design ECS-8 - Hardware Description: information including system components, notes on the design theory of operation, interconnect summary, tuning procedures and 'the question of "standards"'.
(Turn to page 2.)

2. Retuning the ECS-6 UAR/T Clock Rates describes a logical error in January's issue and a new set of frequencies calculated based upon the requirement that the highest data rate selectable should be 1210 baud.
(Turn to page 10)

3. Logical Testing of the CPU/UART/Modem/Tape System is a section concerning the listing and use of two short test programs useful in the initial verification of the tape interface by writing and reading an integer sequential test pattern.
(Turn to page 11.)

4. Errata: Two short notes. (Turn to page 17)

5. IMP Extensions for Tape Interface Control: What does it take to perform the utility operations of data dumps to tape, reading from tape, and comparison of tape data to core? This section begins the description of IMP (Interactive Manipulator Program) extensions with the new command codes, modifications of old program code and the major portion of new routines. The information is not complete, and will be continued in the April issue.
(Turn to page 17).

The complete description of the ECS-8 Modem design required more space in this issue than originally intended. As a result several items have been deferred until the April 1975 issue of ECS: the conclusion of the IMP tape utility extensions, further notes on programming techniques for small microcomputer systems, a new column entitled "Navigation in the Vicinity of α -Aquila" concerning the Intel 8080 instruction architecture in an Experimenter's Computer System programming context, etc. I hate to pull a "perils of a Pauline" ending on the IMP extensions but there is a definite economic limitation on issue size. I am presently looking into methods of compactifying program notational formats - probably along lines of a more symbolic notation supported by uncommented absolute binary listings.

Carl S. Helmer, Jr.
Publisher

March 12 1975.

BIDIRECTIONAL FSK MODEM DESIGN ECS-8 - Hardware Description

The hardware portion of this article concerns a new tape interface modem design to replace the earlier ECS-2 design. The result of applying Occam's razor to a complicated design is a design of simpler concept, not "multiplying redundancies beyond logical necessity" to paraphrase the philosopher. The new ECS-8 design is printed as the detailed circuit diagram in this issue's centerfold, and is described in the text.

A modem, by definition, is a "modulator-demodulator." In the "modulate" mode of operation, the device accepts time-varying serial logic level data from the serial I/O interface (eg: ECS-6 as described in January's issue) and converts it into the "modulated" - in this case FSK - output signal which is sent to the audio memory device for recording. In the "demodulate" mode of operation, the device accepts the modulated FSK signal on an audio recording as read by the audio device, and converts the FSK back into a time varying stream of logic level data for interpretation by the serial interface device. The net result is a facility to store digital data on magnetic tape, potentially to transfer that data to other individuals' systems, and to recover such data at a later time.

The ECS-8 design accomplishes the audio mass storage function in conjunction with a suitable cassette tape recorder. During the course of development of this device in prototype form, three different cassette recorders were tried. The following is a summary of the results of this trial, giving the suitability of the recorders in question:

1. Realistic CTR-104: This Radio Shack product when tested with a continuous "mark" tone exhibits quite audible "wow and flutter" variations in frequency. When recording and reading data at 1210 baud, this \$35 recorder will occasionally exhibit an input parity error but gives good data in general.
2. Panasonic : This recorder costs approximately \$40, and the extra \$5 over the Radio Shack product gives a more than proportional increase in the quality of workmanship. Using the test programs in this issue, it was found capable of recording and reproduction at speeds up to 1210 baud with no observed parity error flashes with the INTEREAD program found in this issue.
3. Superscope C-104: This \$99 recorder is one which will be useful in the home computer context for several reasons: it has a tape position counter which can be used to index block locations on tape for large blocks of data, it has a pitch variation control of 20% which can be used to compensate for differences in tape speed when exchanging tapes with other individuals, and it has some nice "cue" and "review" controls which position the tape with heads active, potentially allowing a fast "block count" tape position search with manual intervention, computer control of the motor. This one also reproduced data at 1210 baud with no observed errors using the test programs in this issue.

The test results here are a heuristic first look at the suitability of various recorders with actual data. Later formal testing using programs to evaluate the units and other factors such as tape brand and quality will be reported in subsequent issues. If you want to select a recorder for use in your own system, this first inspection would seem to indicate

that the choice of a recorder is fairly broad. There is one consideration which will have to be checked out if you want to take advantage of software which M. P. Publishing Co. will be supplying in recorded form. That consideration is the manufacturer's tolerances on tape speed. For the typical Panasonic, Superscope or Sony cassette recorders in the \$50 to \$100 range with AC adapters, this will probably be close enough to the nominal 1.875 IPS to get compatibility with other recorders. I have my doubts about that aspect when the Radio Shack or other inexpensive recorders are considered.

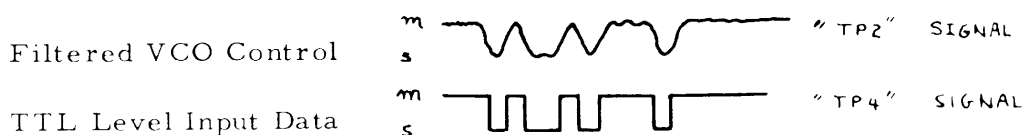
THE FSK RECORDING SYSTEM:

In the diagram of the ECS-8 design, the central element is a Phase Lock Loop, the XR-210 circuit made by EXAR Integrated Systems, Box 4455, Irvine Ca. 92664. This chip is widely available and will cost from \$5 to \$6 in plastic packages, depending upon your source of supply. I have seen at least one advertisement in Popular Electronics classifieds for this chip, and there is the Radio Electronics information cited in a previous issue of ECS. The IC serves the following functions, as programmed by the IN/OUT line of the serial interface device (ECS-6 or equivalent.)

OUTPUT: For output operation, only the VCO section of the PLL is used. The control logic of the design in this mode programs a "mark" frequency when the data line is "1" and programs a "space" frequency when the data line is "0". Thus the time sequence of information on the TSO bit line of the serial interface will be directly mapped into a time sequence of frequencies in the VCO. The VCO output is tapped and run through a 741 buffer amplifier to the tape recorder which is assumed to be in the "record" mode for output.

INPUT: For input operation, the PLL is used to decode the information coming in from the audio information source, turning the FSK modulations into a time sequence of information on the serial data line TSI at the interface. The control logic of this design programs the VCO to the " f_0 " frequency so that the loop will idle in lieu of a signal half way between the mark and space frequencies.

The phase lock loop itself is an example of the feedback principle in action. When an input signal is received, the signal is compared against the VCO signal frequency. The output of the phase detector is an error signal with a sign appropriate to cause the integrated control signal (the voltage into the VCO) to move thus causing the VCO to move in the necessary direction to make the two signals equal in frequency. The PLL thus "locks" onto the signal frequency, causing the VCO to track it. In FSK applications, the control voltage exhibits two "steady" states - and transitions between these states. The comparator section of the XR-210 loop circuit is used to translate the rough VCO voltage into a logic level signal which can be interpreted by the serial interface port. When you have built your first Modem, use of a dual trace scope with chopped input will illustrate a pair of signals like this:



The amplitudes are not to scale, and this diagram is typical of a 1210 baud signal with the filter components of this issue's circuit drawing.

In addition to the XR-210 Phase Lock Loop circuit, several auxiliary elements are found in the ECS-8 design to build a modem system.

An output buffer amplifier provided by the 741 operational amplifier IC -4- is used with capacitive feedback to integrate the VCO square wave, amplify it to several volts, and to isolate the VCO terminal of the XR-210 from the tape recorder connection. A voltage divider in the form of potentiometer R20 is used to set a suitable input level for the tape recorder being interfaced. Note that the input to the tape recorder is shorted to ground when data is to be read from tape. This prevents an unwanted coupling between the tape input and tape output which was observed to occur with all three of the tape recorders mentioned on page 2. A similar feature switches the output of the tape.

An input clipping amplifier is provided by the 741 operational amplifier IC -5- to provide isolation of the PLL input from variations in tape recorder output amplitude. The dual diode feedback around the operational amplifier restricts the amplitude range to essentially the diode forward voltage drop (positive and negative) thus clipping the signal to approximately twice this drop peak to peak. This output of the clipping amplifier is applied to potentiometer R23 which sets the actual PLL input level. During output operations, switch S4a grounds the input to the PLL to prevent coupling of the tape recorder signal via the tape drive electronics.

The motor start delay oneshot is used to give the computer program a "time out" at the beginning of tape read or write operations. When the "SELECT" line goes low indicating the start of an I/O to the modem, this cues the oneshot through the differentiator provided by C13 and R22. The $\overline{\text{RDY}}$ output to the interface logic then goes low for a time period - set by potentiometer R21. At the end of the time period, nominally 2 seconds, the tape drive motor is assumed to have "settled down" to a steady state condition after the initial startup transients, thus the data transfer is not liable to errors caused by transport variations.

The motor control relay is used to turn on the tape recorder's motor under computer control whenever the $\overline{\text{SELECT}}$ interface line is in the low state. Due to the inverting driver of the 7426 section, the selected condition is the "off" state of current in the relay coil - hence the "NC" contacts of the relay (terminal connections 3 and 4) should be used to make / break the "remote" input to the tape recorder.

Control Logic is provided by the two 7426 open collector NAND gate sections, a TEST/CPU mode switch S1, and two TEST control switches S2 and S3. When the mode of S1 is CPU, the control logic is connected to the computer's serial interface for control by a suitable program. When the mode of S1 is TEST, the control functions of TEST DATA (S2) and TEST IN/ $\overline{\text{OUT}}$ (S3) govern the control of VCO frequency settings.

The purpose of including the test switches is to provide a means of initially tuning the device and/or of re-tuning it to a different set of standards at a later time.

FREQUENCY SELECTION CONTROL:

The truth table of the control inputs (whether SI is in CPU or TEST mode) is noted in the centerfold diagram of the ECS-8 design. The A and B columns of the table indicate the logic level on the lines at the points marked "A" and "B" in the diagram. As is usual for such tables, the "X" indicates a "don't care" input. The output of the logic is listed in the third column as " f_{VCO} " - the XR-210 VCO frequency which will result for the given combination of bits.

The XR-210 has two inputs for frequency setting. One is the "keying" input of pin 10 which is normally used to generate FSK in an output-only application according to the EXAR application notes on this device. The second is the "fine tune" input which is supposed to be used to set the center frequency (free running frequency) of the loop in receiving situations. In this design, the same XR-210 is used for both input and output by programming both of these inputs digitally, so that a total of three frequencies is obtained - mark, space, and free running frequency " f_0 ". Two potentiometers R10 and R8 are used to set the "mark" and "space" frequencies using a procedure described below. Optionally, a third potentiometer can be used in this section for the fine tuning of the free running frequency - in place of the fixed 100K resistor R9, illustrated by the "dotted" arrow in the drawing.

OPEN COLLECTOR LOGIC:

Note that all of the "NAND" logic in this circuit design is provided by 7426 high voltage open collector NAND driver gates. For the control logic applications, this means that "pull up" resistors must be provided to the 5 volt logic supply level. The pull up resistors for this use are R4 and R5. For the relay drive application, the "pull up" is provided by the relay coil acting as a load instead of the resistor used in logic applications. The high voltage gate was chosen so that a large (ie: 12 volt) voltage could be applied to the relay coil to guarantee operation with a current greater than the 3 ma required for it to change state. The relay is used as described on page 7 of the January ECS issue. The remaining section of the 7426 circuit can be used as noted in the drawing to drive a relay with DPDT contacts if it is desired to automate the function of switch S4. Open collector logic is also used for the XR-210's output stage, so you will note R3 is used to define the output logic level voltage for the PLL.

NOTES ON CONSTRUCTION OF THIS CIRCUIT:

The ECS-8 design prototype was built with wire wrap construction techniques as documented in M. P. Publishing Co. publications 73-1 and 74-5. With only 5 integrated circuits, a very small board might be used, or a very roomy 4" by 6" board could be used as was used in the prototype. Other interconnection techniques can be used if desired, however for convenient and permanent one-of-a-kind construction wire wrap is really the "only way to go".

A PC board version of this design is in the process of layout as this article goes to press. An announcement of price and availability is expected to be included in the next issue of ECS. The board will be labelled with the component designations in the ECS-8

centerfold of this issue, and very little additional documentation is expected to be required beyond that supplied in this issue of ECS. With whatever technique you employ - wire wrap, point to point solder, PC - it is highly recommended that you use sockets for all integrated circuits. This prevents heating of the IC's if soldering is employed, and provides a convenient means of removing and replacing the chips if you should make a damaging mistake. Three 8-pin "minidip" sockets are required; one fourteen pin DIP and one sixteen pin DIP socket are required.

INTERCONNECTIONS:

The RDY, SEL, TSI, TSO and IN/OUT lines of the modem should be routed to the corresponding lines of one of the serial interface unit ports (eg: ECS-6 I/O-2 lines for one of the channels of tape interface.) If an alternate UAR/T control interface design is used, these lines will have to be run to the equivalent definitions in the controller. To summarize, the lines are:

RDY - this line goes to the RDY input of the channel chosen for the modem in an ECS-6 type multi-channel serial interface.

SEL - this line goes to the SElect output of the channel chosen for the modem.

TSI - this is the serial input line from the demodulator to the TSI line of the serial interface controller for the channel in question.

TSO - this is the serial output line from the appropriate serial interface channel to the modem modulator.

IN/OUT - this line is logic "1" for input, logic "0" for output, and is used to program the frequency control logic of the XR-210.

In addition, the connections for ground, positive 12 volts, positive 5 volts, and negative 12 volts must be made.

The interconnections to the tape recorder are made via the three jacks J1, J2 and J3 (the latter is not drawn explicitly in the diagram.) The jacks can be omitted if you do not mind "pigtails" wired to the modem board with appropriate plugs for the tape recorder. The following connections must be made: a phono-plug to miniature phone plug patch cord is required to go from J1 to the tape recorder's audio output jack - typically marked "Aux Speaker" or "8-ohm Earphone;" A phono-plug to miniature phone plug patch cord is required to go from J2 to the tape recorder's audio input jack, typically marked "Auxiliary Input" or "Microphone"; A phono-plug to sub - miniature phone plug patch cord is required to go from J3 (relay contacts NC and COM) to the motor control input of the tape recorder, typically marked "Remote" or "Dictation."

The modem may be physically mounted along with the rest of the system in a common card rack or "breadboard" layout, or it might be reasonable to put the modem in a separate box associated with the tape recorder.

TUNING PROCEDURES: USING THE TEST CONTROLS

Having made the interconnections, verified proper wiring and power voltages, and inserted the integrated circuits, the tuning of the modem frequencies is the last step prior to testing the unit under computer control. In order to identify points in the circuit for purposes of tuning and understanding the circuit, a new feature has been added to the ECS-8 circuit diagram - notation of several test points as "TPn" where "n" is replaced by an appropriate arbitrary number starting at unity. The basic test point for use in tuning the circuit is test point #1 (TP1) - the amplified VCO signal. The basic test instrumentation can be as simple as an oscilloscope or frequency meter - or both can be used. With the components shown in the circuit diagram, turning on the modem power, independent of any switch settings of S1 to S4, will produce a waveform looking approximately as follows:



What the test switches do is set up data conditions which affect the period of this waveform logically, and enable the corresponding frequency settings to be obtained by trimming resistors.

Trimming the Free Running Frequency f_0 :

Set the TEST/CPU mode switch S1 to the TEST mode and set the test IN/OUT switch S3 to the IN position (S3 open so that line B is logic "1"). This will program the phase lock loop's VCO to the free running frequency logic inputs - and the TP1 signal will be f_0 assuming no interloping frequencies are coming in the J1 connection. The free running frequency can be trimmed by two methods in this mode:

1. By trimming the capacitor C0 by adding extra low value capacitance lumps in parallel with the main C0 with its nominal .03 mf value.
2. By trimming the resistance of R9. However, to keep a reasonable control range for the other adjustments, R9 should not be made much lower than the 100K ohms shown in the diagram.

In the prototype, with a 5% tolerance 100K fixed resistor for R9 and a 10% tolerance C0 of .03mfd without trimming, the oscillator was found to be at 5.555 KHz when power was first applied. The final value of 5.50 KHz (see "Standards" section below.) was achieved by trimming with small silver mica capacitors on a "cut and try" basis. The circuit diagram shows two such "phantom capacitors" as dotted lines in parallel to the main C0. In the PC board version now being prepared, space is left for two such trimming capacitors.

Trimming the "Mark" and "Space" frequencies.

Once the f_0 frequency setting has been trimmed, the following procedure may be used to set the "mark" and "space" frequencies of the FSK modulation. First, set the

the test IN/OUT switch S3 to the "OUT" position (S3 closed so that the B signal line is now logic "0" in the test mode.) This logically programs the VCO control lines to either the "mark" or "space" frequencies depending upon the state of the A signal line. The two FSK frequencies are set according to the "Standards" section below using the following iterative procedure to converge on the final settings. An iterative procedure is required in order to overcome the interaction between the two controls R8 and R10 .

1. Set the test data to "space" - the logic "0" level which occurs on line A when S2 is closed. Adjust R8 until the desired "space" (lower than f_0) frequency has been obtained.
2. Set the test data to "mark" - the logic "1" level which occurs on line A when S2 is open in the test mode with S3 closed. Then set the observed frequency at TP1 to the nominal "mark" frequency.
3. Repeat steps 1 and 2 in sequence until both settings are within the nominal 1% tolerance discussed below in the "standards" section.

Note that this procedure of adjusting the mark and space frequencies should have little if any effect on the f_0 setting. But, if you want to check and "be sure" you might look at f_0 again after these adjustments have been completed.

THE QUESTION OF "STANDARDS:"

Several individuals and representatives of groups of amateur computer enthusiasts have written concerning the subject of standards for data interchange between multiple systems, enabling the distribution of coded software rather than listings which must only be re-entered by hand. With the definition of an audio tape interface scheme comes the question of a standard for data interchange via that method. There are several comments which can be made regarding such standards:

1. Within broad limits, the physical parameters of the recording or interchange method are essentially arbitrary. Thus for example in tape recordings, it is fairly arbitrary whether one uses a series of octave-related tape speeds starting at 2 IPS, 1.875 IPS or even 1.75 IPS. The idea of the standard is to arbitrarily pick one such value of the range and stick to it in a given context of application.
2. Given the same general method of reproduction or interchange, the most useful standard is that which gains the largest market acceptance. Thus all the sour grapes in the world will not change the fact that in certain areas of the computer markets that which IBM designs de-facto becomes industry standard. IBM's arbitrary choice of design and interchange standards is as good as anyone else's choice given the same physical concepts of recording or interchange so its wide market acceptance makes such a standard attractive to other instances.

So, what are the physical parameters affecting the FSK recording method, the general ranges of interest, and the market factors shaping a choice of recording parameters? Answers to these questions - at whatever level of detail required - are implicit in any

selection of a set of standards. In the list here, you will find a summary of the physical parameters and value I have chosen as a "first cut" at the problem. Some notes concerning the choices follow the list.

ECS-8: FSK RECORDING PARAMETERS....

1. Center Frequency: $f_o = 5.50 \text{ Khz}$ 1%
2. Mark Freq: $f_{\text{mark}} = 107.5\% f_o = 5.93 \text{ Khz}$ 1%
3. Space Freq: $f_{\text{space}} = 92.5\% f_o = 5.09 \text{ Khz}$ 1%
4. Data Rate of UAR/T: 1210 baud 1%
5. Asynchronous format parameters:

Stop Bits:	2
Data Bits:	8
Parity:	odd

The basic specification of the FSK signal is its center frequency and deviation. The above set of parameters reflects a choice of 5.50 Khz center and deviation of 7.5% in either direction to produce the two data frequencies. The choice of these particular numbers reflect the following general considerations:

1. The frequency should be kept as high as possible relative to the data rate of the interchange, to provide a large number of cycles (between 4 and 5 in this case) at the space frequency for the PLL to lock on.
2. The frequency of transmission should not be higher than about 6Khz when the typical 10Khz band limit of the usual inexpensive recorder is considered - this guarantees that the wide band signal of the FSK will be recorded with sufficient accuracy to recover the data later. The information theory prediction that at least the second harmonic information would be required was verified in the prototype by attempting interchange at approximately 8 Khz f_o with other parameters identical. Result: errors in subsequent read operations. (At 6 Khz, there is still sufficient reproduction at the harmonic 12 Khz to ensure accuracy, but the drop off with increasing frequency puts a 16Khz signal outside the range of reproduction.)
3. The deviation of 7.5% (relative to center frequency) was chosen to make the basic frequency shifts large compared to possible erroneous shifts such as tape recorder "wow" and "flutter" or steady state differences in tape speed. With the prototype circuit, deviations as large as 12% were found possible, but were at the limits of control ranges and less stable than the 7.5% figure. Smaller deviations were also tried . The final 7.5% choice is a good balance between the small deviation consideration and the limits of this circuit.
4. The baud rate and format considerations are taken from the ECS-6 design - subject to the considerations stated on the next page of this issue.

With these physical parameter considerations for an FSK modem taken care of, what are the market considerations - considerations of more than one user? A standard is only a standard when it is useful to the individuals employing it. For your own in-house use, you could potentially use any set of parameters within the capability of the basic design. My purpose in publishing this list of parameters is one and only one: to provide a definition of the FSK parameters which I will use in recording programs for distribution to subscribers, whether generated by myself or by other individuals now in the process of creating articles for this publication. If a design such as the ECS-8 modem is used, there is room for a fairly broad variation in these parameters to allow retuning for other sets which may or may not be used by other sources of software. I make no claim to special knowledge or universal acceptance of this particular set of parameters - and the flexibility of the basic modem design allows later re-specification should there be widespread dissatisfaction among subscribers with the particular choices in this set.

A final note on the standards subject: this discussion has only concerned the physical (low level) details of recording standards. There is another whole "can of worms" involved in the programmed format of data which is conveyed by tapes using this method. To keep the size of this issue within the bounds of sensibility I am deferring discussion on that topic for now.

RETUNING THE ECS- 6 UAR/T CLOCK RATES

The following frequency settings are achieved as a result of retuning the ECS-6 oscillator to 38.720 KHz (25.83 μ s for those who set frequencies via oscilloscopes) and taking into account a logical error in the writeup of the ECS-6 design as published. The logical error in question was the assumption that a 16 division ratio is possible with the 4-bit 74193 counter used to establish clock frequencies, when in fact the maximum is division by 15 and two of the 4-bit codes are identical. The retuning is done so that the highest bit rate will be approximately 1200 baud (1210 baud is .83% off the typical commercial rate of 1200 baud) and the 110 baud rate will be retained at one point in the series for use with the teletype. The complete list of frequencies and codes is thus:

<u>Code</u>	<u>Iden.</u>	<u>Baud Rate</u>	<u>Code</u>	<u>Ident.</u>	<u>Baud Rate</u>
0000	0	1210 (tape)	1000	8	151.25
0001	1	1210 (tape)	1001	9	134.44
0010	2	605	1010	10	121.00
0011	3	403.33	1011	11	110.00 (TTY)
0100	4	302.5	1100	12	100.83
0101	5	242.	1101	13	93.08
0110	6	201.66	1110	14	86.43
0111	7	172.86	1111	15	80.67

With this retuning, the control word for the channel 0 teletype output becomes octal 262 instead of 362, and word 011/220, word 011/211 of the previously published ELDUMPO routine must be changed to reflect the new TTY rate code.

LOGICAL TESTING OF THE CPU/UART/MODEM/TAPE SYSTEM:

Once the modem has been checked out at the level of tuning described on pages 7 to 9 of this issue, the next step is to check out the ability of the system to record data generated by a program and later read that data. Two self-contained programs are provided in this issue for the purpose of testing the interface by a very simple method: An integer number sequence displayed in the binary lamps has a very characteristic visual pattern when the rate of generation is lower than the eye's characteristic "flicker" limit. By writing then reading the sequence of binary numbers 000_8 to 377_8 repetitively, this sequence will be put on a test tape for corroboration visually in the display when reading. The other 8-bit display can be used to flash any parity errors and to continuously monitor the difference between one word and the next when reading data. The first program of interest is the data generation routine INTEGEN:

Note: Starting with this issue, I will be mnemonically referencing the 8008 I/O commands of the system I actually wired by their proper symbols. No changes are made in the actual codes printed in previous issues of the magazine - which differ from the published and corrected ECS-5 codes by a level of inversion in the 3-bit selection of device within an 8008 I/O channel. It is not a major point, since an individual system of hardware can potentially use any one of the 8008 I/O codes (with the proper characteristics) for a given function.

INTEGEN:	004\000 = 006	LAI	First turn off interrupts as usual
	004\001 = 002	00 000 0010	
	004\002 = 117	IN7	I/O Interrupt control code
INTGLOOP:	004\003 = 006	LAI	
	004\004 = 026	0001 01 10	} 1210 baud, ch. 1, select, output
	004\005 = 111	IN4	→ Tape unit control word code (formerly called
	004\006 = 310	LBA	"IN3" due to ECS-5 error)
	004\007 = 175	OUT36	Write status to left display.
	004\010 = 301	LAB	Recover status
	004\011 = 044	NDI	
	004\012 = 030	00011000	} Mask with TEOC/TBMT positions
	004\013 = 074	CPI	
	004\014 = 030	00011000	} And test for valid TEOC & TBMT
	004\015 = 110	JFZ INTGLOOP	→ Keep looping around until
	004\016 = 003	L	the UAR/T is ready for more.
	004\017 = 004	H	
	004\020 = 302	LAC	→ then give the Uar/T some more stuff.
	004\021 = 113	IN5	
	004\022 = 302	LAC	→ and display the same stuff on right lights
	004\023 = 177	OUT37	
	004\024 = 020	INC	Increment the data for next output word
	004\025 = 104	JMP INTGLOOP	And reiterate the whole cycle ad
	004\026 = 003	L	infinitem. . . you stop this program
	004\027 = 004	H	manually with the single step
			control.

This programlet can be entered into memory at the absolute addresses shown by using the IMP program previously published. Then the "Shift X" operation with appropriate address setup can be used to enter execution at location 004/000.

Once the INTEGEN program has been entered and execution initiated from IMP, a first check of the system can be done aurally by connecting a high fidelity amplifier and speaker to test point TPl. The characteristic FSK signal should be heard, which in this case (going gung-ho at 1210 baud) sounds somewhat like a multi-engine propeller driven aircraft during takeoff - especially when the volume is turned up through a good set of speakers! To make the test tape, the following manual procedure is suggested:

1. Temporarily suspend program execution by flipping the CPU panel controls to the single step mode. After this is done, the steady state "mark" tone of 5.93 Khz should be heard in the speaker if you use the setup suggested above.
2. Put the recorder into its recording mode and start it up. Leave the remote control input temporarily empty so that the controls are active independent of computer motor control operation.
3. After 10 to 20 seconds of mark tone recording, turn the CPU back to the run mode so that the actual data will be recorded - an integer sequence of numbers generated by INTEGEN at the maximum data rate of the system, 100 CPS (Characters per second.) When the program is running, observe the integer pattern in the display.
4. After a coffee break or suitable 5 to 15 minute period of time, come back, turn off the recorder, put the CPU in single step, use the bootstrap mode to change location 3 (IMPSTATE) back to 002g, interrupt the CPU and re-enter IMP. You now have a test tape with an integer sequence of numbers on it at 1210 baud.

With this process of making the tape completed, rewind the cassette (or reel if you use reel-to-reel) and enter the INTEREAD program code as found on the next page. The INTEREAD program is designed to set up for read operations at 1210 baud, and read any characters detected by the UAR/T with display on the binary lamps. The program also does a rudimentary error check as follows:

- The difference between one character and the next is continuously calculated and displayed as the lefthand bit of the OUT37 display lamps. If this lamp ever flickers, it indicates that an invalid sequence of integers was read - it should be solidly "on" during input operations.
- The three receiver status bits - OVERRUN, FRAMING ERROR and PARITY ERROR - are displayed in the righthand section of the OUT37 display lamps. If bits 2, 1 or 0 of this lamp array ever flash, then one of the error conditions was detected. In practice, except when the phase lock loop is free running, these lamps were usually always "off" indicating a lack of errors. At rates higher than 1210 baud, all three recorders tested would occasionally produce read parity errors. At the 1210 baud rate, the Radio Shack recorder would occasionally (once in several minutes) flash a parity error.

Once entered, set MEMADDR of IMP to 005/000 and start INTEREAD with the IMP "shift X" operation.

INTEREAD:	005\000 = 006	LAI	} → Turn off interrupts code
	005\001 = 002	00 000 010	
	005\002 = 117	IN7	→ is sent out to interrupt control port
INTRLOOP:	005\003 = 006	LAI	} → 1200 baud, channel 1, select, input
	005\004 = 027	0001 01 11	
	005\005 = 111	IN4	→ is set up in ECS-6 control
	005\006 = 310	LBA	→ save status just read
	005\007 = 044	NDI	→ mask off RDA bit
	005\010 = 040	00 100 000	} → and test RDA for data available
	005\011 = 074	CPI	
	005\012 = 040	00 100 000	
	005\013 = 110	JMP INTRLOOP	loop around if not available
	005\014 = 003	L	
	005\015 = 005	H	
	005\016 = 113	IN5	- read code for ECS-6 channel
	005\017 = 320	LCA	- save data in C-register
	005\020 = 175	OUT36	- write data just read in the left binary display
	005\021 = 302	LAC	- restore saved data
	005\022 = 223	SUD	- subtract previous data left in D-register
	005\023 = 012	RRC	- rotate difference into high order
	005\024 = 340	LEA	- and temporarily save it in E
	005\025 = 301	LAB	- restore status from B
	005\026 = 044	NDI	} → and mask off the error indicators
	005\027 = 007	00 000 111	
	005\030 = 264	ORE	- and merge the result with high order difference
	005\031 = 177	OUT37	- and display in the right hand display lamps
	005\032 = 332	LDC	- and create the new "old" data value.
	005\033 = 104	JMP INTRLOOP	and back to gobble up some more bits
	005\034 = 003	L	from the tape...
	005\035 = 005	H	

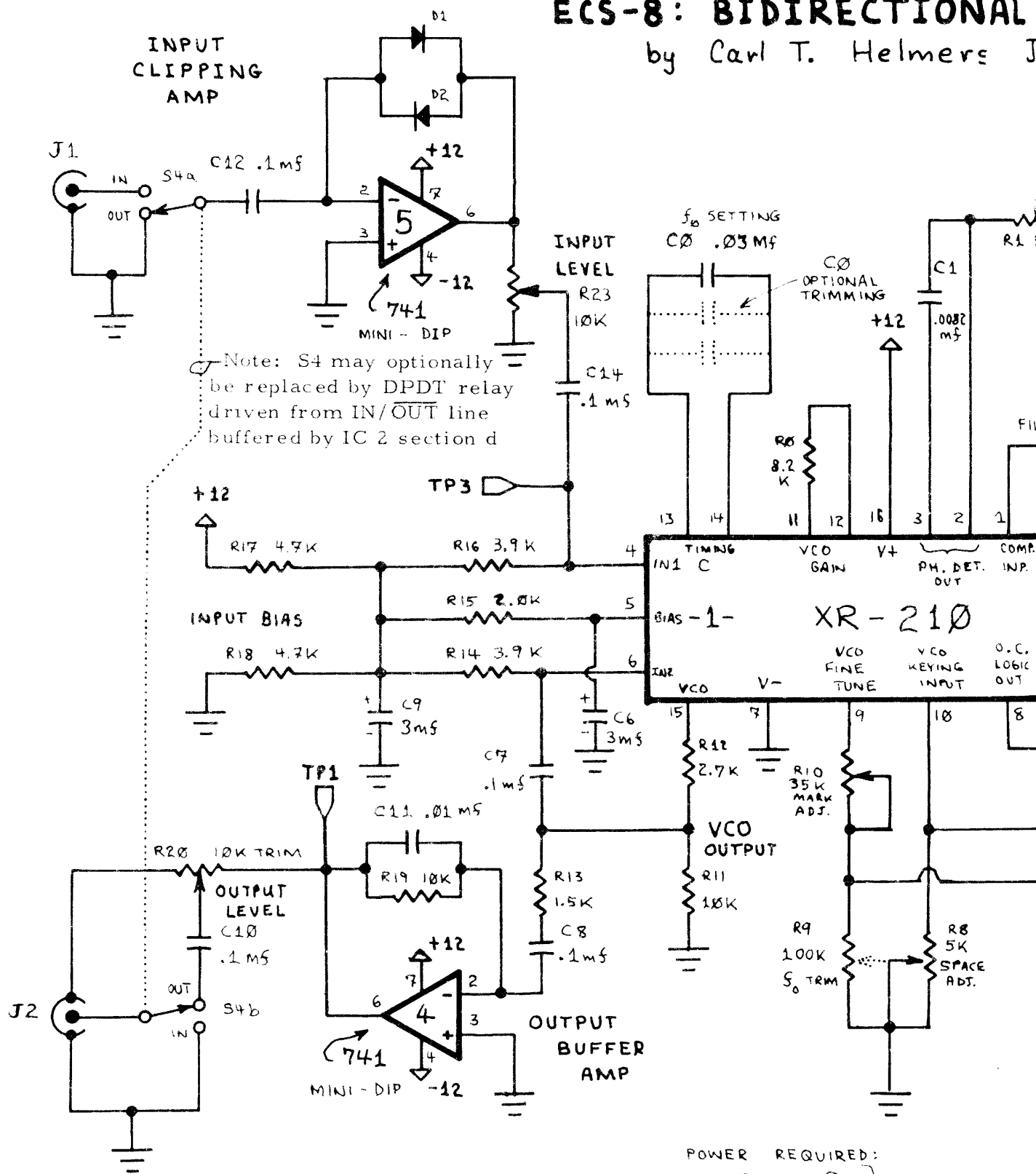
When INTEREAD has been initiated in operation, with a blank tape noise signal, display outputs should "run wild". The reason is that when the PLL oscillator is free-running without locking to either the mark or space frequencies, the control voltage is at the center of its range, "hunting" around for the proper lock. If you examine test point 2 at this point, you should find a "random" waveform with an amplitude of several 100's of millivolts with the recorder playing back a blank tape.

When the first "mark" tone appears on the tape, the loop should quickly lock solidly onto a fixed level at TP2. Then, when the data begins to appear, you will be able to set up the chopped dual trace scope display illustrated on page 3 - if you have or can borrow the use of a dual trace display. With an oscilloscope as a tool, you can adjust the input level to get the cleanest waveform at TP2 - or, using only your CPU and program INTEREAD as a tool, you can adjust the level while watching the error lamps - with too little level, errors occur - and the same goes if you over drive with a combination of high tape recorder amplitude and high input level setting.

COMPUTERS IN SCIENCE FICTION? Imaginative applications of technology are often anticipated years ahead of realization by fiction writers - thus Jules Verne's well known anticipations of TV and fast powerful submarines. Good and well known science fiction writers like Robert Heinlein and Poul Anderson have often come out with neat computer applications. I am interested in readers' contributions to a bibliography in this area including short descriptions of the computer-related theme of the story being referenced.

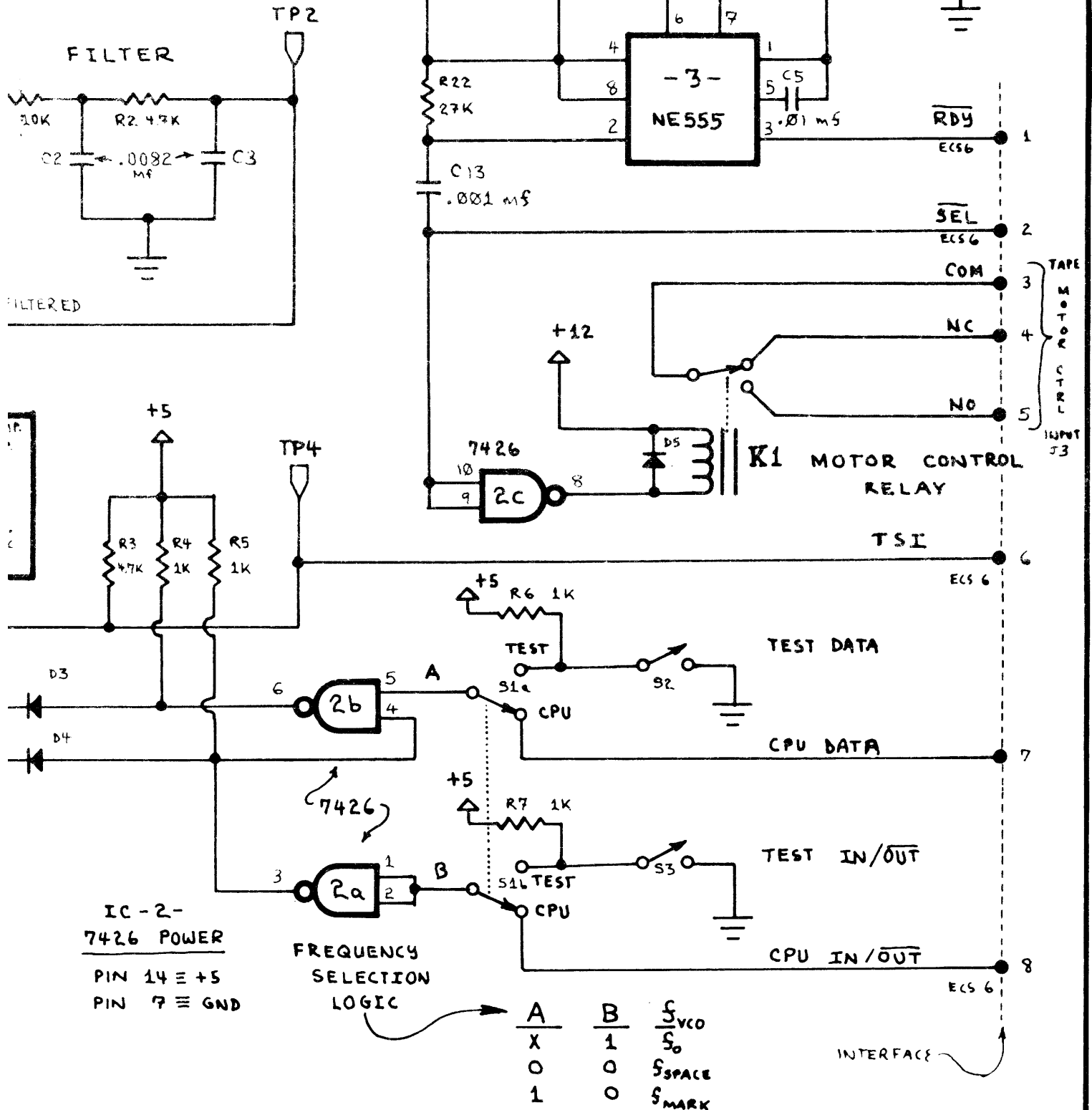
ECS-8: BIDIRECTIONAL

by Carl T. Helmers J



FSK MODEM

Jr



ECS-8 MODEM DESIGN PARTS LIST ...

C0 = .03 mfd base plus optional
trimming as required.

C1,C2, C3 = .0082 mfd (10% tol.)

C4 = 30 mfd 10 volt electrolytic

C5,C11 = .01 mfd (20% tol.)

C6,C9 = 3 mfd, 25 volt electrolytic

C7,C8,C10,C12,C14 = .1 mfd 50 volt

C13 = .001 mfd 50 volt

D1,D2,D3,D4,D5 = silicon switching
diode, 1n914 or equivalent

J1,J2,J3 = RCA style phono jacks
for interconnect to tape recorder
via patch cords (optional.)

P1 = DIP plug interface to ECS-6
I/O sockets from modem terminals

Integrated Circuit Listing:

IC -1- XR-210 Phase Lock Loop
Modem Circuit

IC -2- 7426 Open Collector High
Voltage NAND Driver

IC -3- NE555 Ready Delay Timer

IC -4- 741 "minidip" op amp

IC -5- 741 "minidip" op amp

K1 = SPDT Miniature Reed Relay Tape
motor control. The prototype uses
a surplus Grigsby-Barton #GB31C-G2150
But any relay which will operate with
a 6 to 10 volt potential and less than
16 ma can be used with the 7426 driver.

R0 = 8.2K VCO gain

R1,R11,R19 = 10K

R2,R3,R17,R18 = 4.7K

R4,R5,R6,R7 = 1.0K

R8 = 5K space adjust pot

R9 = 100K

R10 = 35K mark adjust pot

R12 = 2.7K

R13 = 1.5K

R14, R16 = 3.9K

R15 = 2.0K

R20 = 10 K output level pot

R21 = 500K ready delay pot

R22 = 27K

R23 = 10K input level pot

S1 = DPDT Test/CPU mode switch

S2 = SPST Test Data Switch

S3 = SPST Test Switch

S4 = DPDT Tape Signal Routing

The four test points indicated in the diagram may be implemented with teflon feed thru insulated standoffs, or appropriate test prod jacks.

ERRATA CORRECTIONS FOR PREVIOUS ISSUES:

The following errata have been detected in the referenced issues, and are noted here for the record:

January 1975, page 4: The "NDB1" pin of the COM2502 UAR/T is pin 38. The TTY line's source is 16-4 not 14-4.

February 1975, page 13: The reference to "PCW" as the output of IC 14 pin 6 should have read "PCC". This error is also found in the text of the BANK SELECTION LOGIC description on page 11 .

IMP EXTENSIONS FOR TAPE INTERFACE CONTROL:

The IMP program is extended , as documented in this section, to handle an added capability - the dumping of absolute binary data onto the tape interface for long term storage, followed by later recovery of that data. A comparison function is also incorporated to allow the data written on tape to be checked against core data so that one will be certain of the veracity of the tape copies. The new functions added to the IMP program are the following:

"T" - this function is used to set IMPSTATE to a value of 3 so that a second letter can be decoded as a two letter tape control sub-command. The two letter tape control command sub-command combinations are listed later on page 18 .

"Shift W" - this command requires two keys to be depressed for safety, and is used to invoke the data write utility. Pressing this key assumes that the program parameters of data count and a starting MEMADDR value been set up using the H, L, TL and TH control commands of IMP. It also assumes that the tape recorder has been set up in the "record" mode to receive data from the appropriate modem. The channel/rate selections are also assumed, as defined by the TR and TU commands to be described.

"Shift C" - this command requires two keys to be depressed for safety, and is used to invoke the data comparison utility routine. It assumes the same program setup as the corresponding "Shift W" write operation which produced the block, and assumes that the physical setup of the tape recorder is for a read operation.

"Shift R" - this command also requires two keys to prevent accidental activation. It is used to invoke the data read utility, which is identical to the data comparison utility with the exception that data is stored in appropriate memory addresses rather than compared against the addresses.

In addition to these direct extensions to the IMP command facility, the subcommands of the "T" operation include the functions described on the next page.

The tape control subcommands are used to define the content of several RAM data areas, display the content of these data areas, and to perform tape utility control actions.

"TB" - this command/subcommand combination is used to display the current content of the 16-bit tape block length count.

"TD" - this command/subcommand combination is used to display the current content of the tape control word used for determining tape unit and rate.

"TF" - this command/subcommand combination is used to display the current formatting error count - errors in the three UAR/T status bits detected during read and comparison operations.

"TH" - this command/subcommand is used to transfer the current IMPENTRY value to the H portion of the block length count.

"TI" - this command/subcommand is used to initialize the tape control parameters of block count, error counts (format and comparison data), and control word. All the control data is zeroed out.

"TL" this command/subcommand is used to transfer the current IMPENTRY value to the L portion of the 16 bit block length count.

"TR" - this command/subcommand is used to define the UAR/T rate portion of the control word from the low order content of IMPENTRY.

"TS" - this command/subcommand is the tape leader spacing command, and is used to turn on the tape motor for a period of time (ten seconds) sufficient to move the cassette position past the leader after a rewind. It invokes a routine which uses timing loops to count approximately 2.5 million 8008 CPU states in terms of the structure of the counting subroutines and data used to call them.

"TU" - This command/subcommand is used to define the control word unit as the current two low order bits of IMPENTRY.

"TX" - this command/subcommand is used after a comparison operation to display the count of words read which differed from internal memory data (the count is valid assuming a previous "TI" initialized the count data areas.

The decoding of these subcommands is done in a manner which is identical to that used for the main set of IMP commands - the software extensions for the tape facility include a generalization of the "IMPDECO" routine given in last month's issue to allow symbolic specification of the command table address and the command branch target taken when a match page address taken when a match is found.

One other change has been made in the previously published IMP program - the symbol table has been moved out of page 0 (bootstrap page in ECS systems) to reside at address 012/260 in the protected memory of the system software. This minimizes restart activity to the initialization of the first 1028 bytes of page 0 after the CPU has been powered down for additional interfacing or other activities.

The first items to present in the course of redefining the software load to include extensions for tape interface functions are the changes in the IMP decoding algorithm. Basically, the IMPDECO algorithm is made to begin with a SYM function call to define the command table address. In current software, there are now two command tables - for the IMPSTATE value of 2, the same codes are defined as in the previous case, but the table is extended by four entries for the four new regular IMP commands. The second command table is used for the IMPSTATE value of 3 (ie: following the "T" command) to decode the second character of the two character tape control commands. In this second command decode application, a different page for the command branch is also required, thus the IMP "GOTFUNC" routine must also be modified to provide this new generality.

Because the decode routine has been generalized, it is now necessary to use a setup procedure to define the parameters for IMPDECO. In this patched version of the IMP software, a branch is made to "IMPSETUP" with a return to "IMPRESUME" when the state 2 decode is used. IMPSETUP is used to define the high order portion of the GOTFUNC branch address in DECOGO, and to setup the symbol for the state 2 command table, IMPCMDS. IMPRESUME is the normal entry point to the decoding routine in this new version. Note that a jump to IMPSETUP could also have been made from the jump at location 020. Note also that the NOP's and KEYWAIT at location 013/023 have been replaced by a jump to location 010/000, the place where the setup for an IMPSTATE value of 3 is executed, defining the "T" subcommand table instead of the normal IMP table.

013\023 = 104	JMP TSETUP	IMPSETUP:	
013\024 = 000	L	013\120 = 310	LBA
013\025 = 010	H	013\121 = 006	LAI
IMPGO:		013\122 = 030	s(DECOGO)
013\026 = 104	JMP IMPSETUP	013\123 = 075	SYM
013\027 = 120	L	013\124 = 076	LMI
013\030 = 013	H	013\125 = 013	h(IMP)
		013\126 = 006	LAI
IMPRESUME:		013\127 = 032	s(IMPCMDS)
013\031 = 075	SYM lookup addr	013\130 = 104	JMP IMPRESUME
013\032 = 301	LAB restore chr.	013\131 = 031	L
		013\132 = 013	H

Note that the IMPSETUP routine is located in a region of memory address space which had formerly been occupied by the "GOTFUNC" routine (see last issue.) The GOTFUNC routine has been moved to location 013/251 and modified to define the high order target address from the data stored in the variable DECOGO, rather than the default page 013 in the original version. The address located at 013/035 must accordingly be changed to 251 so that the new location of GOTFUNC will be reached from IMPDECO. The new version of GOTFUNC is listed on the next page at the top.

Also at the top of page 20, right hand side, is a listing of the jump instructions which are located in page 013 so that the new tape commands can be reached outside of page 013. When the normal IMP decode occurs, it references a page 013 address - one of these jumps if one of the new commands is detected.

GOTFUNC:

013\251 = 060 INL
 013\252 = 347 LEM
 013\253 = 006 LAI point
 013\254 = 030 s(DECOGO) to DECOGO
 013\255 = 075 SYM
 013\256 = 337 LDM ((was LDI)
 013\257 = 106 CAL SETJMP
 013\260 = 212 L
 013\261 = 013 H
 013\262 = 106 CAL SYSSETUP
 013\263 = 135 L
 013\264 = 013 H
 013\265 = 104 JMP GPJMP
 013\266 = 015 L
 013\267 = 000 H

Here are the four jumps used to reach outside page 013 when normal IMP decode finds read, write, compare or "T" command characters...

WRITEJ: 013\310 = 104 JMP WRITE
 013\311 = 015 L
 013\312 = 010 H
 READJ: 013\313 = 104 JMP READ
 013\314 = 123 L
 013\315 = 010 H
 COMPJ: 013\316 = 104 JMP COMPARE
 013\317 = 116 L
 013\320 = 010 H
 TJ: 013\321 = 104 JMP TSETUP
 013\322 = 325 L
 013\323 = 012 H

And here is the new command table at location 013/344, including the four new command codes as well as all the old commands...

IMPCMDs:	013\344 = 227	"Shift W"	} → write command (to tape)
	013\345 = 310	1(WRITEJ)	
	013\346 = 203	"Shift C"	} → compare tape to memory
	013\347 = 316	1(COMPJ)	
	013\350 = 222	"Shift R"	} → read tape into memory
	013\351 = 313	1(READJ)	
	013\352 = 324	"T"	} → initiate tape control state
	013\353 = 321	1(TJ)	
	013\354 = 304	"D"	} → this section of table is identical to the previously printed version but in page 013 instead.
	013\355 = 240		
	013\356 = 305	"E"	
	013\357 = 156		
	013\360 = 313	"K"	
	013\361 = 221		
	013\362 = 314	"L"	
	013\363 = 076		
	013\364 = 311	"I"	
	013\365 = 152		
	013\366 = 312	"J"	
	013\367 = 150		
	013\370 = 316	"N"	
	013\371 = 153		
	013\372 = 230	"Shift X"	
	013\373 = 200		
	013\374 = 310	"H"	
	013\375 = 106		
	013\376 = 315	"M"	
	013\377 = 112		

There is one final modification of the old code which must be noted: the symbol table has been moved from page 0 to page 012, location 260, for the same reason that the regular IMP command table was moved. Thus word 000/071 of the SYM has to be changed to 012_g and word 000/073 of SYM must be changed to 260_g to complete modifications.

With the preliminaries completed, the next item of interest is the beginning of the list of routines required to implement the new IMP functions. The first routine in address sequence is the TSETUP routine used to fool IMPDECO into decoding via the "T" subcommand table TAPECMDS for the first character following a "T". This occurs when IMPSTATE is 3 following the "T" command...

TSETUP:	010\000 = 310	LBA	save the character just read	
	010\001 = 006	LAI		
	010\002 = 002	s(IMPSTATE)		} change state for the next input character by referencing and redefining IMPSTATE for normal code interpret after state 3 special ...
	010\003 = 075	SYM		
	010\004 = 076	LMI		
	010\005 = 002	2		
	010\006 = 006	LAI		
	010\007 = 012	s(TAPECMDS)		} point to tape command table
	010\010 = 075	SYM		
	010\011 = 301	LAB		} recover tape subcommand character and jump to the new generalized IMPDECO as if normal entry but with alternate command table
	010\012 = 104	JMP IMPDECO		
	010\013 = 033	L		
	010\014 = 013	H		

If you look at this routine carefully - and observe its coordination with IMPDECO - see if you can find a way to eliminate 2 bytes and thus compactify the software... such an improvement is possible.

When the IMPDECO routine was entered in the normal "2" IMPSTATE, a write command "Shift W" might have been decoded. If so, the WRITE thing to do is to branch to WRITEJ in page 013, and thence to this little wroutine...

WRITE:	010\015 = 006	LAI		} first thing in writing - to tape, not necessarily for publication - is to reference the command word set up by TR/TU commands,
	010\016 = 014	s(TAPECTRL)		
	010\017 = 075	SYM		
	010\020 = 307	LAM		
	010\021 = 044	NDI		} then extract the rate/unit bits,
	010\022 = 374	11 111 100		
	010\023 = 064	ORI		} and superimpose output select...
	010\024 = 002	00 000 010		
	010\025 = 370	LMA		} the new command code is good for this I/O to control logic & later..
	010\026 = 111	IN4		
	010\027 = 106	CAL WAITOUT		} go wait for the proper TBMT/TEOC and RDY flags to indicate that the motor start is done...
	010\030 = 147	L		
	010\031 = 012	H		
	010\032 = 106	CAL OUTCOUNT		} go wait "x" milliseconds more and then write out a 16-bit data count
	010\033 = 200	L		
	010\034 = 012	H		
	010\035 = 006	LAI		} define a temporary copy of the block data count by referencing its location then copying data prepared by OUTCOUNT from CPU registers B and C to storage reserved for TCOUNT
	010\036 = 016	s(TCOUNT)		
	010\037 = 075	SYM		
	010\040 = 371	LMB		
	010\041 = 060	INL		
	010\042 = 372	LMC		
	010\043 = 016	LBI		} then define a 10 centisecond wait
	010\044 = 012	1010		
	010\045 = 106	CAL WAITCS		} interval, then wait it out before commencing the main data block.
	010\046 = 116	L		
	010\047 = 012	H		

The writing of data onto the tape immediately brings to mind the question of the data format. The basic data format implied for this program can be stated explicitly: a block of data consists of (on output) a leader of (nominal) 2 seconds while the motor gets up to speed, followed by an additional delay for output of "x" seconds to allow "slop" in tape positioning on input, followed by two characters containing the block count, then a delay of .1 seconds then the actual data bytes, followed by another "x" second delay and a repeat of the block count for verification. The block is closed out mechanically by turning off the motor after a final delay of 2 seconds. As a tentative value of "x" for this software, I have used .1 seconds - although I it is not yet obvious that this is the best value. I chose a delay between the count words and the actual data block for the following reason: when listening by ear to the data, a characteristic rhythm pattern is heard at the start of the block - a single blip of data followed by the actual block. This gives an indication - roughly - that the data is likely to be in the right format. A further reason is to allow easy detection of the block start and end when scanning the tape fast using a tape recorder with "cue" and "review" controls such as the Superscope model mentioned.

The code of page 21 has gotten the output operation down to a point where the start of a block has completed and the program is now ready to output the main sequence of data of interest...

OUTLOOP:	010\050 = 106	CAL WAITOUT	} do not procede any further until the flags have been cleared by UAR/T and it is ready for output...
	010\051 = 147	L	
	010\052 = 012	H	
	010\053 = 106	CAL ATMEMA	} to find out what the current MEMADDR is by loading it in H/L
	010\054 = 000	L	
	010\055 = 012	H	
	010\056 = 307	LAM	} go fetch the current byte so that it can be sent to tape
	010\057 = 113	IN5 (true code)	
	010\060 = 106	CAL INCMA	} not an ancient andean indian, but the routine in-crements the value of MEMADDR
	010\061 = 164	L	
	010\062 = 013	H	
	010\063 = 307	LAM	} fetch the next byte and display it ...
	010\064 = 175	OUT36	
	010\065 = 060	INL	} and increment address LO and fetch next 1 byte
	010\066 = 307	LAM	
	010\067 = 177	OUT37	} and display it too...
	010\070 = 006	LAI	} point to data count
	010\071 = 016	s(TCOUNT)	
	010\072 = 106	CAL D2B	} then decrement the count temporary TCOUNT
	010\073 = 132	L	
	010\074 = 012	H	
	010\075 = 140	JTC ENDOUT	} zresult of zcountdown determines what will happen...
	010\076 = 103	L	
	010\077 = 010	H	
	010\100 = 104	JMP OUTLOOP	} if the carry was false, underflow has not occurred, so the block is not done - reiterate it!
	010\101 = 050	L	
	010\102 = 010	H	

Now in every instance except the last, the jump true carry at 010/075 will fail, causing routine to loop back for the next byte of the block being dumped. Finally, the carry will set to 1 by D2B, causing execution to flow to ENDOUT, listed at the top of page 23.

ENDOUT:	010\103 = 106	CAL OUTCOUNT	} at end of block, write the count again for confirmation...
	010\104 = 200	L	
	010\105 = 012	H	
	010\106 = 016	LBI	
	010\107 = 310	200 ₁₀	} then set up for a 200 centisecond (2 sec)
	010\110 = 106	CAL WAITCS	
	010\111 = 116	L	} block trailer interval...
	010\112 = 012	H	
	010\113 = 250	XRA	→ clear accumulator
	010\114 = 111	IN4	→ and output null code to tape units.
	010\115 = 025	KEYWAIT	→ having completed the output, back to IMP command interpreter...

The following routine is accessed by the tape control command "TS" and is used to space the tape a fixed interval after rewinding and setting up for forward motion.

LEADER:	010\272 = 250	XRA	} clear the accumulator so that a momentary null code can be output to the tape controller...
	010\273 = 111	IN4	
	010\274 = 006	LAI	} then point to the tape control word via SYM mechanisms...
	010\275 = 014	s(TAPECTRL)	
	010\276 = 075	SYM	
	010\277 = 307	LAM	→ load the accumulator with the selection
	010\300 = 064	ORI	→ then force "output select" onto whatever rate/channel had been selected...
	010\301 = 002	00 000 010	
	010\302 = 111	IN4	→ then output the command code, turning on the motor...
	010\303 = 056	LHI	} set up for 10 second leader delay (adjust this constant to suit your tastes...)
	010\304 = 012	10 ₁₀	

The H register is thus used as a temporary count for the number of seconds of delay in the leader, serving to cycle the following leader delay loop...

LDELAY:	010\305 = 016	LBI	} the inner loop of leader delay is a 1 second delay programmed by the WAITCS routine...
	010\306 = 144	100 ₁₀	
	010\307 = 106	CAL WAITCS	called to delay a total of
	010\310 = 116	L	100 centiseconds as programmed
	010\311 = 012	H	by the value in B on entry
	010\312 = 051	DCH	→ after the inner loop delay, decrement the
	010\313 = 110	JFZ LDELAY	the seconds counter (H register)
	010\314 = 305	L	} and branch back if needed...
	010\315 = 010	H	
	010\316 = 250	XRA	→ clear the accumulator...
	010\317 = 111	IN4	→ then output the turn off (null) code...
	010\320 = 025	KEYWAIT	→ and back to keyboard interpreter...

The next routine to be listed is a service routine which is activate by "TH" and is used by IMP to set the H portion of the tape block count working storage...

COUNTH:	010\321 = 006	LAI	} point to block count via SYM
	010\322 = 022	s(COUNT)	
	010\323 = 075	SYM	
	010\324 = 371	LMB	→ then define H portion from last entry, left in
	010\325 = 104	JMP EXAMINE	the B register by GOTFUNC...
	010\326 = 156	L	} then go to EXAMINE in IMP proper in order to output the count...
	010\327 = 013	H	

A similar routine is used to perform the same function for the L portion of the block count when invoked by the "TL" command to transfer IMPENTRY to the count's low order.

```
COUNTL:    010\330 = 006  LAI
            010\331 = 022  s(COUNT) } → point to block count (length)
            010\332 = 075  SYM
            010\333 = 060  INL → increment to look at low order...
            010\334 = 371  LMB → and load the low order from entry
            010\335 = 061  DCL → and look again at start of COUNT
            010\336 = 104  JMP EXAMINE
            010\337 = 156  L
            010\340 = 013  H } → and jump to EXAMINE it
                               via the IMP routine...
```

The next section of code consists of two utility functions for display of tape control data, "DSPLYCTRL" invoked by the "TD" command and "DSPLYBLK" invoked by the "TB" command...

```
DSPLYCTRL: 010\341 = 006  LAI
            010\342 = 014  s(TAPECTRL) } → first point to tape control
            010\343 = 075  SYM } → word via SYM mechanism...
            010\344 = 307  LAM → then fetch TAPECTRL...
            010\345 = 175  OUT36 → and output to display
            010\346 = 250  XRA
            010\347 = 177  OUT37 } → clear the other display to all zeros...
            010\350 = 025  KEYWAIT → and back to IMP as usual...
```

The following routine displays the block count in the two display lamp sets...

```
DSPLYBLK:  010\351 = 006  LAI
            010\352 = 022  s(COUNT)
```

This label identifies shared code to execute(sic) SYM and go to EXAMINE of IMP...

```
GOEXAM:    010\353 = 075  SYM
            010\354 = 104  JMP EXAMINE
            010\355 = 156  L
            010\356 = 013  H
```

The next set of code is a utility routine "ATMEMA" which is called at several places in the tape control extensions (and later software to be published soon) in order to place the current content of MEMADDR into the L and H registers - pointing AT MEMA...

```
ATMEMA:    012\000 = 006  LAI
            012\001 = 006  s(MEMADDR) } → point to MEMADDR
            012\002 = 075  SYM
            012\003 = 307  LAM → put H part of MEMADDR into A temporarily
            012\004 = 060  INL to point to the L part of MEMADDR
            012\005 = 367  LLM → which enables the L result to be defined...
            012\006 = 350  LHA → after which the H result can be loaded from A
            012\007 = 007  RET → and it is now safe to return to caller.
```

The tape utility commands "TR" and "TU" are used to set the "rate" and "channel" sections of the ECS-6 tape interface control word respectively. The next page lists the service routines for these commands. "RATE" is reached when the alternate command table decodes an "R" following the "T". "CHANNEL" is reached similarly when the character "U" follows a "T". In either case, the current IMPENTRY low order data is used to define the corresponding field of TAPECTRL.

RATE:

012\010 = 006	LAI	} → point to the TAPECTRL word
012\011 = 014	s(TAPECTRL)	
012\012 = 075	SYM	
012\013 = 307	LAM	→ and fetch the old control value first....
012\014 = 044	NDI	} → save all except old rate by
012\015 = 017	0000 1111	
012\016 = 370	LMA	→ and temporarily save back in TAPECTRL
012\017 = 301	LAB	} → so that IMPENTRY copy can be fetched to A
012\020 = 015	XCHG	
012\021 = 044	NDI	and exchanged to high order (see ECS
012\022 = 360	1111 0000	VI#1 p. 20)
		→ clear extraneous IMPENTRY stuff...

This label identifies common code of RATE and CHANNEL used for recombinations...

```

NEWCTRL:    012\023 = 267   ORM → move saved portion of TAPECTRL into aligned
            012\024 = 370   LMA      set of new bits and save it again...
            012\025 = 104   JMP  DSPLYCTRL
            012\026 = 341   L
            012\027 = 010   H

```

The channel routine is analogous to RATE, but zaps new stuff into different bits...

CHANNEL:	012\030 = 006	LAI	} → point to TAPECTRL word
	012\031 = 014	s(TAPECTRL)	
	012\032 = 075	SYM	
	012\033 = 307	LAM	and fetch the old content...
	012\034 = 044	NDI	as with RATE, save all bits <u>except</u> channel bits
	012\035 = 363	1111 00 1 1	by "anding" with mask...
	012\036 = 370	LMA	then stuff it back into memory temporarily
	012\037 = 301	LAB	and turn attention to IMPENTRY copy
	012\040 = 044	NDI	} → which must be first
	012\041 = 003	0000 0011	
	012\042 = 002	RAL	} → then shifted left into the right position
	012\043 = 002	RAL	
	012\044 = 104	JMP	} → after which the same combining maneuver is used as for the RATE case...
	012\045 = 023	L	
	012\046 = 012	H	

The software control structure used to define the tape block format references the following routine several times. WAITCS accepts a parameter in the B register which specifies the number of nominal 10 millisecond wait intervals (centiseconds) required...

WAITCS: 012\116 = 026 LCI } load an inner loop count which
 012\117 = 147 103₁₀ } approximates a 10 millisecond delay...

The timing loop here is not quite 10 milliseconds with the above constant!

[illegible]

Another WAIT function required for coordination is the WAITOUT routine. Here the object is to centralize the instructions required for testing the status bits when output is being done to one of the ECS-6 controller's channels. Note that this routine is general, only requires that TAPECTRL be initialized prior to entry. The analogous routine in the previously published ELDUMPO program is at locations 011/217 to 011/235 and could be potentially consolidated by a CAL WAITOUT if ELDUMPO is re-written to use the SYM mechanism. A characteristic of software written without automated assembly and compilation aids is the price in time paid to modify routines - thus the point is academic at the present time.

```

WAITOUT:  012\147 = 006  LAI
          012\150 = 014  s(TAPECTRL) } point to control word
          012\151 = 075  SYM
          012\152 = 307  LAM and fetch it to A...
          012\153 = 111  IN4 and peruse the status bitz...
          012\154 = 044  NDI
          012\155 = 130  01 011 000 } and isolate RDY, TBMT and TEOC
          012\156 = 074  CPI → and test for all in proper state...
          012\157 = 130  01 011 0000 of readiness...
          012\160 = 110  JFZ WAITOUT } and loop around ad infinitum if
          012\161 = 147  L not ready to return...
          012\162 = 012  H
          012\163 = 017  RET (middle digit is a mistake, but the RET instruc-
                           ↑
                           "don't care"
                           tion spec sez "don't care" - so why bother to
                           change it at this point? )

```

The next routine to be listed in this issue is the "OUTCOUNT" routine used to dump the 16-bit block data count onto the tape after waiting "x" centiseconds, where "x" is here compiled as 10. This effectively allows a 1/10 second error in the positioning of a tape block relative to the end of the last previous block - since reading operations will wait 2.0 seconds from motor startup and writing will wait 2.1 seconds. In order to avoid missing data, the read "listening" must begin prior to the commencing of actual data bytes.

```

OUTCOUNT: 012\200 = 016  LBI } set up the "x" second wait with
          012\201 = 012  1010 "x" equal to .100 second (10 centiseconds)
          012\202 = 106  CAL WAITCS }
          012\203 = 116  L } with the setup, go waitonit
          012\204 = 012  H
          012\205 = 006  LAI
          012\206 = 022  s(COUNT) } setup for I/O by pointing to COUNT
          012\207 = 075  SYM
          012\210 = 307  LAM → fetch high order of count to A
          012\211 = 310  LBA → save it for later use in B
          012\212 = 113  IN5 → and then send it out as the first byte of data...
          012\213 = 060  INL → point to low order
          012\214 = 307  LAM and fetch it to A
          012\215 = 320  LCA but save it in C
          012\216 = 113  IN5 before zapping A with the output side
          012\217 = 007  RET of IN5 and returning...

```

Note that this routine also has a hidden extra function in its definition of the content of B and C as the high and low order block count for later use.

The next segment of the IMP extensions is the routine accessed by the "TI" command of the extended program...

```

INITIAL:      012\220 = 006  LAI
               012\221 = 014  s(TAPECTRL) } → point to first data byte...
               012\222 = 075  SYM
               012\223 = 016  LBI } data count for initialization by crude method of
               012\224 = 012  10  } zapping 10 bytes in a row...
INILOOP:      012\225 = 076  LMI } by immediate movement of
               012\226 = 000  0  } zero to the memory location...
               012\227 = 060  INL  increment the memory address pointer
               012\230 = 011  DCB      and decrement the count...
               012\231 = 110  JFZ INILOOP } back for more until done
               012\232 = 225  L
               012\233 = 012  H
               012\234 = 006  LAI } three brownie points and a pat on the back if
               012\235 = 000  0  } the reader can figure out a better way to
               012\236 = 111  IN4 } clear A for the I/O control word reset...
               012\237 = 025  KEYWAIT

```

This initialization takes advantage of the fact that all the tape specific data is located in addresses 200 to 211_h and can thus be zapped as a block... without separate symbolic references.

Then comes a bunch of miscellaneous jumps from page 012 to page 010 for the new subcommands... due to IMPDECO's single-page orientation...

```

JLEADER:      012\240 = 104   JMP LEADER - this jump is used to get out of page
               012\241 = 272   L           012g after "TS" command is decoded by the
               012\242 = 010   H           IMPDECO routine as modified...
JDSPYBLK:     012\243 = 104   JMP DSPYBLK - same here for "TB" command...
               012\244 = 351   ↗ H ↘ - oops - if 8008 were decent would be correct
               012\245 = 010   ↖ L ↙
JDSPYCTRL:    012\246 = 104   JMP DSPYCTRL - same here for "TD"...
               012\247 = 341   L
               012\250 = 010   H
JCOUNTL:      012\251 = 104   JMP COUNTL - same comment for "TL"...
               012\252 = 330   L
               012\253 = 010   H
JCOUNTH:      012\254 = 104   JMP COUNTH - same for "TH"
               012\255 = 321   L
               012\256 = 010   H

```

The following is inserted out of sequence for editorial reasons... it fits.

```
TSETUP:      012\325 = 006   LAI
              012\326 = 002   s(IMPSTATE)
              012\327 = 075   SYM
              012\330 = 076   LMI
              012\331 = 003   3
              012\332 = 006   LAI
              012\333 = 030   s(DECOGO)
              012\334 = 075   SYM
              012\335 = 076   LMI
              012\336 = 012   h(TAPECMDS)
              012\337 = 025   KEYWAIT
```

In order to setup IMP for a second character to follow the "T" command, the IMPSTATE value must be set to 3 to force the alternate decoding of the next character in the stream.

Must also point to the word which holds the "GOTFUNC" high order address and load that word with the (non symbolic) H address of the tape subcommand table...

Then return - as always - to the IMP keyboard wait routine...

This issue concludes with the new symbol table for IMP and the command table of the IMP tape extensions...

SYMBOLS:	012\260 = 012	- "00" is symbol table self-pointer
	012\261 = 260	
	012\262 = 000	- "02" is IMPSTATE
	012\263 = 003	
	012\264 = 000	- "04" is IMPENTRY
	012\265 = 004	
	012\266 = 000	- "06" is MEMADDR
	012\267 = 006	
	012\270 = 000	- "10" is GPJMPMA
	012\271 = 016	
	012\272 = 012	- "12" is TCMDS
	012\273 = 354	
	012\274 = 000	- "14" is TCTRL
	012\275 = 200	
	012\276 = 000	- "16" is TCOUNT
	012\277 = 201	
	012\300 = 000	- "20" is INOPS
	012\301 = 203	
	012\302 = 000	- "22" is COUNT
	012\303 = 204	
	012\304 = 000	- "24" is BADDATA
	012\305 = 206	
	012\306 = 000	- "26" is BADFORM
	012\307 = 210	
	012\310 = 000	- "30" is DECOGO
	012\311 = 212	
	012\312 = 013	- "32" is IMPCMDS
	012\313 = 344	
	012\314 = 012	- "34" is TAPECMDS
	012\315 = 354	
TAPECMDS:	012\354 = 330	"X" -Input errors in data display...
	012\355 = 054	1(EDATAD)
	012\356 = 306	"F"
	012\357 = 047	1(EFORMATD) - Input errors in format display...
	012\360 = 311	"I"
	012\361 = 220	1(INITIAL) - Tape data initialization routine...
	012\362 = 323	"S"
	012\363 = 240	1(JLEADER) - Tape leader routine
	012\364 = 302	"B" - Tape block size display...
	012\365 = 243	1(JDSPLYBLK)
	012\366 = 304	"D" - Tape control word display...
	012\367 = 246	1(JDSPLYCTRL)
	012\370 = 314	"L" - Low order block length setter...
	012\371 = 251	1(JCOUNTL)
	012\372 = 310	"H" - High order block length setter...
	012\373 = 254	1(JCOUNTH)
	012\374 = 322	"R" - Rate setter (not monopoly bureaucrat)
	012\375 = 010	1(RATE)
	012\376 = 325	"U" - Channel setter....
	012\377 = 030	1(CHANNEL)

As noted in the introduction, the tape control software is only partially listed in this issue due to space considerations. The remainder will become a major portion of the April issue of ECS... CTH

ECS - The Monthly Magazine of Ideas for the MICROCOMPUTER EXPERIMENTER

News & Notes to accompany Volume 1, No. 3 - March 1975. Some midnight madness written on completion of the present issue...

THE DEMISE OF MICROSYSTEMS INTERNATIONAL: Current issues of electronic trade publications report the demise of Intel's 8008 and 8080 second source, Microsystems International. This Canadian firm is withdrawing from all IC business due to a lack of profits - a necessary input to any durable enterprise.

RISE OF A NEW CPU? General Instrumentation and Honeywell have come up with a new "CP-1600" 16-bit single chip computer reportedly 5 times faster than another recent 16 bit announcement by National. The EE Times note had a price reported as \$250 for just one, with no information on when the part would be available.

WANT TO SEE WHAT TEXAS INSTRUMENTS has to say about microprocessors? April 15 to 18, nationwide, TI is sponsoring 4 half-hour TV lectures on the subject early in the morning. I can't print the entire schedule of stations, but interested readers might lookup a local TI or distributor number in the Yellowpages and inquire - if you don't already have the information from trade publications.

REGARDING FLOPPY DISKS: Don Whitehead (980 New Haven Avenue, Milford, Connecticut) will be running the floppy disk pooled purchase previously announced. Write him for complete details.

A summary is as follows: Drives will be the new Memorex model (original mechanical design with late user-oriented electronics). Price for the drive will be \$575 assuming 11 orders total by the appropriate deadline, \$700 if less than 11 units are purchased. A \$150 deposit will be required pending the 11-unit order deadline - or if you can not wait, the single unit price can be used to get the fastest possible turn-around for the order. The price will include shipping to continental USA. A manufacturer's documentation package of 4 books is \$12 extra, and a recommended package is the manufacturer's support kit including 10 disk cartridges, the documentation package, a test cartridge, and cleaning kit for a price in the \$150-170 range, above the drive cost alone. As previously announced, if the drive deal goes through, M. P. Publishing Co. will provide an interface article. One final point - once 11 orders are reached, the offering will be extended indefinitely - but it requires serious individuals to act very soon to assure the first order needed to begin the "OEM" pricing operation.

SOFTWARE FOR SALE: With the availability of the ECS-8 PC card (layout and price to be in April's issue) tapes of ECS 8008 software will be made available beginning with the IMP program. Price for a BASF C15 Cassette & Mailer with IMP recorded redundantly is \$7.50. Later versions incorporating improvements in the program will be available to previous purchasers on a cassette-recycling basis for \$2.50. First class mail is part of the price - with extra postage required for airmail or overseas purchasers. Tapes will be recorded in binary image format using the ECS-8 type of modem, from the working software in the ECS 8008 prototype system.

WANT TO BLOW YOUR OWN HORN? As a new feature, subscribers' descriptions of their own Experimenter's Computer Systems (not necessarily the M. P. designs, Intel CPU's or other fixed restraints on hardware) are solicited. Write it up in a few pages, covering the system design, unique features, problems you have encountered, etc. Oh yes, while it won't make you rich, there is a royalty of 10% on of sales prorated by the fraction of space devoted to the article in each issue, payable in an initial lump based on current circulation with residuals thereafter...

CTH March 13 1975

ECS

THE MONTHLY MAGAZINE OF IDEAS
FOR THE MICROCOMPUTER EXPERIMENTER

Publisher's Introduction:

Here you have the April 1975 issue of ECS, complete and unexpurgated. The main theme of this issue is the introduction of the "SIRIUS-MP" language as a notational form for expressing programs. The idea of SIRIUS-MP is to slightly generalize the low level code approach to program notation so that it will be fairly expedient for subscribers to hand "cross compile" programs on whatever variation of the "home brew computer" concept they have implemented. The variations on this theme include...

1. The SIRIUS-MP Language... This article, beginning on page 2, is a first statement in these pages of some of the concepts involved in the language. It also provides information useful in understanding the several SIRIUS examples found in this issue.
2. BOOTER: An "Emergency" Bootstrap Loader... It is common knowledge what to "do when the lights go out." But what do you do after the lights go out when your computer and volatile software were on the same power source as the lights? Turn to page 11 for a description of an emergency bootstrap loader concocted one weekend to combat electron deficiency anemia.
3. IMP Extensions For Tape Interface Control (Continued...) In the last issue, I did not quite fit all I intended to print within the confines of 28 pages. The remaining segments of the tape interface are presented in a SIRIUS fashion along with the equivalent 8008 code, beginning on page 14.
4. Comments on the ECS-8 Design: Turn to page 19 for a short note on one aspect of the ECS-8 design which I should have pointed out in the March article, and was the source of a complaint from my brother Peter Helmers.
5. Notes on NAVIGATION IN THE VICINITY OF ~~Q~~-AQUILA ... #1. So, you went out and got yourself an Altair computer? Now what? Turn to page 20 for the first in a continuing series of articles on the use and abuse of the Intel 8080 instruction set in an ECS context - with occasional intermingled information on hardware interfaces to be supplied from time to time (but not this time however.)
6. Erratum: Turn to page 24 for a short note about an ECS-7 diagram error.
7. A Note Concerning The Motorola 6800 MPU: Also on page 24 is a short note concerning the use of the M6800 in an ECS context, now possible to contemplate on a practical basis in the near future.

This issue is going to press April 21 1975. The next issue is fairly well defined as of this date, and will include: an article by subscriber James Hogenson concerning the design of a unique oscilloscope graphics interface featuring a 4096 point (64x64 grid) matrix of spot locations; a continuation of the software discussions begun in this issue; and possibly a review of one or two tools which will be of interest to readers.

Carl T. Helmers, Jr.
Carl T. Helmers, Jr.

Publisher April 20 1975

The SIRIUS-MP Language...

an approach to machine independent low level code.

This issue begins a subject which will continue in the pages of ECS for some time to come: the subject of expressing programs in a fairly well defined low level "language" which is in principle independent of any particular microprocessor or other small computer you might have. This will facilitate your use of published programs written for an 8080 if you own an IMP-16, or programs written for 8008 if you own an M6800, etc. - provided the programs in question are expressed in the SIRIUS way.

The name I have chosen for this language is "SIRIUS-MP". The SIRIUS is a combination of an April pun and the following input: if Altair is the brightest star (visual magnitude) in the constellation Aquila, then let me modestly name this mode of program expression after the brightest star in the sky, the star α -Canis Major or SIRIUS. So, if you are SIRIUS about Altair (or other computers available inexpensively both now and in the near future) you will find this series of articles illuminating. So much for the advertisement now to turn to some information content....

WHAT IS A COMPUTER LANGUAGE?

The answer to this question (as is always the case with complicated subjects) can range from the superficial to the formal mathematical intricacies of compiler-writing and language design. Since this publication is not a technical journal on software engineering, it must necessarily leave out a lot of the detailed information on the subject, to concentrate on the application of the concept. (Upon sufficient interest - one inquiry - I'll spend an evening sometime and compile a bibliography on the subject of compilers and computer languages.) With this disclaimer I'll proceed to the subject of computer languages in the context of a home brew microcomputer system.

Starting from first principles, what is a "language" (eg: English, German, Pidgin, integral calculus, set theory) in general? I'll confine the subject arbitrarily to the concept of "written languages" and put forth the following formulation:

A LANGUAGE IS A HUMAN INVENTION FOR THE PURPOSE OF
EXPRESSING THOUGHTS.

This definition is filled with implications: language is an invented technology (probably the first) of humans (or other critters.) language is utilized in communicating thoughts between individuals. Language is appropriate to thinking beings. Now what could this possibly have to do with your urge to program and use a microcomputer ?

A fair amount of course! The specific application of the language concept to the problem of programming a computer is the concept of a "programming language." The specific part of this application is the limiting of computer languages to certain classes of thoughts...

A COMPUTER LANGUAGE IS A HUMAN INVENTION FOR THE PURPOSE OF EXPRESSING COMPUTER PROGRAMS.

Just as there are numerous variations on the "natural language" concept (Eg: ENGLISH), the diversity of human thought has lead to a wide range of computer languages from the most general to the specific and application oriented. In each such language, the author(s) have selected a set of elements needed to solve the particular problem and combined these in a (more or less) self consistent manner and come up with a solution to the problem of expressing programs of a particular class.

The creation of a programming language for the particular case of a microprocessor system in the "homebrew" (ie: limited hardware) environment is the object of this series of articles in ECS. When you design and or build a hardware system, your first problem is solved - a computer that "works". To get beyond this first phase the problem becomes developing the programs enabling your system to do interesting things. A language can be used for ≥ 3 purposes in the process of programming your computer:

- a. An appropriate language enables you to abstractly specify a program in a first iteration of design without worrying "too much" about details. Get the control flow figured out first, then worry about low level subroutines!
- b. An appropriate language will enable you to hand compile programs expressed in that language for use on your own computer, even if the program was developed and debugged on another computer. You know the "algorithm" works even though you have not yet translated it to your own use.
- c. A language appropriate for the home microprocessor will be of sufficient simplicity to allow hand compilation or compilation by a very simple compiler.

These considerations - the definition of a "home brew computer" context - are a major input into the design of the SIRIUS-MP method of program expression.

SETTING THE PROJECT IN CONTEXT: HOW WILL SIRIUS-MP COMPARE TO EXISTING LANGUAGES?

The approach taken in the choice of elements for the SIRIUS-MP language is that of a "pseudo assembly language." An assembler is the simplest of all software development aids to write, so this choice tends to satisfy criterion "c" above. But what about "a" and "b"? This is where the "pseudo" part enters the description: it is a language one step removed from the detailed instruction level in many of its operations. SIRIUS is an assembly-type language for a class of similar machine architectures - with operations found in general on such machines forming its "primitives." The subject of address resolution is left intentionally non-specific and symbolic so that variations in the way

data is accessed can be left to the hand or machine-aided process of generating code for your own system. Many of the statements written in this form will generate only a single instruction on the "object" machine - but others will require a series of several instructions to specify required actions on a given machine. It is my intention to include within this "pseudo assembly language" concept several programming constructs borrowed from high order languages in current usage - but stripped of the complex syntax of a true high level language and specified in the simplified form of the SIRIUS-MP syntax, such as it is. This adaptation of a language to a specific purpose and class of users is a widespread practice in the compiler/language design business. Several examples come to mind of specific languages for specific usage contexts:

XPL - this language is the compiler-writer's language to a great extent. It is a specific and limited subset of PL/1 by McKeeman, Wortman and Horning which is documented in a book entitled "A Compiler Generator." The adaptation here is to concentrate on those features necessary for the writing of compilers and exclude all else. (Intel PL/M is very close to XPL)

HAL/S - this language was developed for guidance, navigation and control applications of NASA by Intermetrics Inc., the author's employer of several years. HAL/S is specialized to include the vector and matrix data forms used in spacecraft navigation - and to provide highly visible "self-documented" code which was not possible in the assembly language style approach used in the Apollo program.

SNOBOL - here is a language which is primarily oriented to "string handling" programs - a very broad range of applications, in some sense including the writing of compilers as well.

ALGOL - this language is the antecedent of many currently used languages, whose original intent was a specialization in generality - the ways in which algorithms could be best specified, in the abstract form.

These languages are all examples of much more extensive and complex methods of program expression from a compiler writer's standpoint - although from the user's standpoint they are orders of magnitude easier to program with than doing the equivalent in a low level "pseudo assembly language" or formal assembly language for a specific machine. It is the problem of generating code by hand or with minimal program aids which limits the possibilities of SIRIUS program specifications to the low level approach.

WHAT ARE THE COMPONENTS OF A COMPUTER LANGUAGE?

For those readers with a software or computer-science background, this discussion is in the nature of a review. For readers with little programming background this will present new information.

When you build a computer from a kit or from scratch, your problem is to put together a set of hardware components according to a certain system design (usually inherent in the microcomputer chip design) such that all the components play together as a working system. At a level of abstraction far removed from - yet still within the context of - the detailed hardware, a language for computers is also a construction of component parts which must "play together" according to a particular design if the language is to be

useful as a means of expressing programs. At the most abstract level of discussion, a language consists of two major component parts designed to provide an interface between a human being's thoughts and the requirements of computing automata. These are:

SYNTAX: - this component of the language is the set of rules concerning the correct formulation of basic "statements" or "expressions" in the language in question.

SEMANTICS: - this component of the language is the set of rules governing the intelligible combinations of syntax elements - the combinations which produce a well defined and translatable meaning which can be used in turn to generate machine code for some "object" or "target" machine of a compiler.

The syntax and semantics of a programming language can be chosen with a somewhat ill-defined border: one of the major trade-offs to be done in designing a language and associated compiler is deciding how much of the work is to be performed by the syntactical analysis and how much is to be left to semantic interpretation. At one extreme there is the complex syntax of a high order language in which much of the semantic intent of a statement is inherent in the syntax used; at the other extreme there is the case of the simple "assembly language" style of syntax in which very little function is inherent in the syntax - which merely distinguishes labels, operators and operands.

SIRIUS-MP is at the "assembly language" end of the trade - its syntax is kept simple, so that a minimal compiler (or hand compilation) will be used to translate it to machine codes, and the semantic interpretations are largely look-ups based on the specific content of the statements coded in a program, with very little variation on certain basic forms for operands and operators.

SPECIFICATION OF SIRIUS-MP:

The specification of a language can be a very formal and very dry process. A language specification is ultimately required in order to clearly convey the meaning of statements coded in the language, the legal variations on such statements, etc. etc. A certain level of consistency in specification is required, for instance, if I want to write a compiler for a given language. At the present time, however, my reasons for formulating SIRIUS are much less demanding than the formal specification of a language: I am interested in creating a method of describing programs which will be heavily commented and used principally for publication in ECS (and possibly other publications.) Thus the specification is left in a fairly "soft" form for the time being within a general framework described in this issue. The time for a formal specification will be the day I sit down and write an appropriate compiler - or a reader decides to do so through impatience and the desire to write one for publication (with the usual royalty of course.)

In lieu of a really formal specification of the SIRIUS-MP language, the next few pages contain an informal description of several notational devices employed in the examples of SIRIUS-MP programs in this issue, and comments on why the forms are used. The areas covered are: STATEMENTS, ADDRESSING & REFERENCE, DATA REPRESENTATIONS, and OPERATIONS. Omitted in the present discussion are several languages forms to be described at a later time, including certain "structured programming" concepts and details of argument/parameter linkage conventions for subroutine calls in SIRIUS-MP.

STATEMENTS :

The basic notational unit of a program which is written in SIRIUS-MP is the "statement." The statement concept embraces the others mentioned on page 5, as can be illustrated by the following prototype format:

LABEL:

TARGET OP SOURCE * COMMENTS ;

As in most decent assemblers, the intent is to make the statement "free form" and thus requiring no fixed column or line boundaries. Hence the following devices are used as a part of the syntax:

The end of a statement is indicated by a ";" (semicolon) as in a host of PL/I-like languages.*

A label, if present, is distinguished from the first (TARGET) operand or the operation mnemonic (OP) by a ":" (colon). With this choice of trailer, labels must not duplicate any operation codes (OP) which can have similar endings.

An asterisk (*) is shown as a separator between the main part of the statement and the comments field at the right.

For examples of the use of this format, see the several program listings included with this issue below. The fields in this prototype statement are as follows:

LABEL - this field (and its ":" separator) is optional and is used to define a symbolic program label. A label is ultimately required to define all symbols used in a program with the exception of certain implicitly defined symbols such as CPU registers and flags.

TARGET - this field (optional) specifies a symbolic reference or absolute address for the memory location(s) or I/O devices which will receive data as a result of an operation. Certain operations will not require a target field for proper notation.

OP - this field is required in order to specify an "operation" to be performed at some time. Certain operations will correspond to executable code in the translation. Others will be used to reserve storage and indicate aspects of the program generation process.

SOURCE - this field is required to specify a minimum of one operand for each operation. Its format will vary depending upon the type of operation intended - variations will include various forms of symbolic reference as well as compound forms used to control functions such as "FOR" loop constructs or "IF" statements.

COMMENTS - here the field intent and use is fairly obvious - to explain what is going on it is useful to make notations.

* Note: The alternate form of statement boundary indication to the ";" is to start a new statement on a new line. The examples in this issue all omit the ";" specified above - a detail to be corrected in future issues.

ADDRESSING AND REFERENCE:

For those individuals who have experience with high level languages (eg: FORTRAN, COBOL, PL/1, ALGOL, BASIC etc.) the common experience is to blithely go ahead and program an application with the various "variables" declared within a program by implicit or explicit means. This approach is appropriate for a high order language in most instances because the problem of addressing and referencing data in the computer has been solved in a fairly general and quite reliable manner by the compiler writers. When the time comes to drop down one level of abstraction to the assembly level, the problem of addressing has to be again considered in a more explicit manner since many more details of machine architecture are inherent in such programming. In deciding what forms of addressing and data reference to include in SIRIUS-MP, the low level approach is augmented by several methods of more abstract reference. The following are some key referencing concepts:

ABSOLUTE ADDRESS: The concept here is of a fixed location in the memory address space of the computer or a given I/O instruction channel designation. In a system built around a Motorola 6800 for example, most I/O operations will be carried out with reference to absolute addresses for the I/O interface memory locations - at least in simple programs this will be the case. In the INTEL or National IMP-16 architectures explicit choices of I/O channel require designation of numbers, often in an absolute form.

EXAMPLE: The Octal expression 020023 could represent an absolute address.

SYMBOLIC ADDRESS: The concept here is to reference the name of a data item in an instruction rather than its actual address. In principal all such names map into a fixed and unique address at execution, either through the operation of a compiler's address resolution or through a run time lookup mechanism such as the SYM routine used in the previously published ECS 8008 software. In SIRIUS notation, a symbol is defined by its appearance as a LABEL of a statement, or its existence as a pre-defined entity such as a register designation.

EXAMPLE: Given label ANYSYM, a reference in some other (eg: assignment) statement might be:

ANYSYM =: 0 (as the TARGET operand.)

INDEXED SYMBOLIC ADDRESS: The concept here is to reference the starting location of a block of memory by the first symbol involved, and to indicate an offset (from zero up) in bytes by a second symbol or literal in parentheses following the first. Thus:

ANYSYM(OFFSET) is a reference to the location ANYSYM plus the current value of OFFSET when the statement is executed.

or

ANYSYM(23) is a reference to address ANYSYM plus 23.

An alternate form of expression for this would be to show an addition (+) operator rather than use a FORTRAN or PL/1- like subscript reference with parenthesis.

SPECIAL SYMBOLIC ADDRESSES: Here the concept is the notation of certain symbols with a fixed meaning, which in an assembler would effectively become "reserved" symbols not subject to redefinition. The forms used in the listings in SIRIUS in this issue are the following :

W(ANYSYM) means "the whereabouts of ANYSYM" and is the notation used to indicate a reference to the absolute address of the symbol.

M(ANYSYM) means "memory reference to the location found in the value of ANYSYM." This is the basic "pointer" form used, and will assume that the value in ANYSYM is a full address (eg: 16 bits for most machines.)

T(ANYJMP) means "the address portion of a jump instruction at ANYJMP". This notation was introduced to allow the equivalent of a FORTRAN assigned GO TO to be used by altering a jump instruction.

A, B, C, D, E, H, L are symbols used freely to represent registers on the Intel 8008 and 8080 type of machine architectures. In translating this reference to a Motorola 6800 or National IMP-16, or other computer architecture, an appropriate software equivalent would be used if registers are not available.

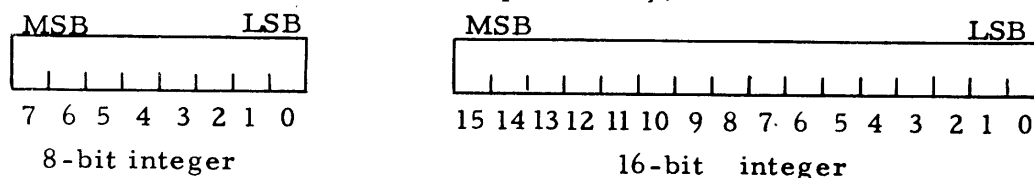
L(ADDRESS), H(ADDRESS) are used to reference the Low and High order portions of a full address (eg: 14 or 16 bits) on typical microcomputers when it is desired to examine only one byte. This is especially useful as a notation for the Intel architectures, but the same functional meaning goes on other machines.

The various forms of addressing and reference described can be used to specify the "operands" - SOURCE and TARGET - of a statement. The concept of a "SYMBOL" is the generalized idea of one of these forms of reference (excluding absolute references.) A "symbol table" for a program is a list of such symbols, usually including some additional information about the item. In a future article on the hand generation of code this concept will be explored in more detail.

DATA REPRESENTATIONS:

A "data representation" is a method of conceptually treating a group of data bits in the storage of a machine, and is usually fairly dependent upon hardware features of a given machine. The basic data representation of all the extant 8-bit microcomputers is the 8-bit binary integer (two's complement is the rule.) This is augmented in certain machines such as the 8080 and the 6800 by a limited set of 16-bit operations implemented to handle address calculations. For the 16-bit microcomputers and minicomputers, the word length as a rule sets the basic representation as a 16-bit integer, although smaller 8 bit quanta can usually be employed. This immediately suggests that the basic assumption to be built into SIRIUS-MP is that data ought to be operated upon in 8 and 16 bit

quanta. This will prove a useful decision for most processors likely to be in common use by readers of this publication (if there is enough interest, I'll make some comments at a future time on adaptation to 12-bit machines such as the DEC PDP-8 and its imitators.) The two representations are thus (pictorially)...

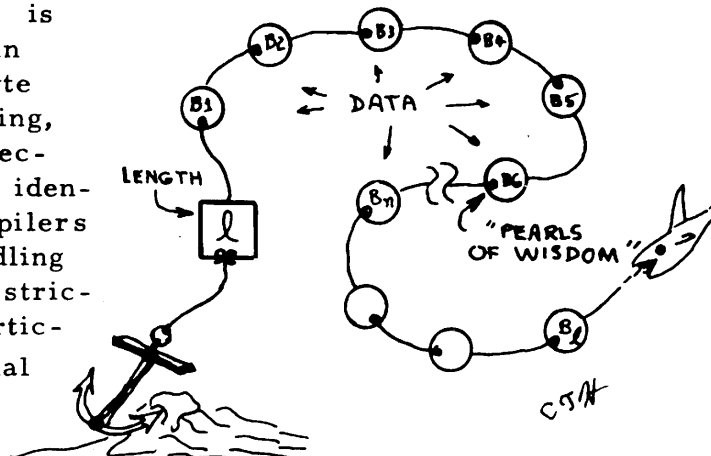


The fact that there are two possible ways to reference integers built into the hardware operations of the typical 8 and 16 bit microcomputer formats, (8008 excluded) leads to a desire to specify a notation for the length of data involved. I could choose among two basic alternatives in this area:

- Specify data type in some form of declaratory way. This would be analogous to an XPL statement such as "DECLARE X FIXED;" or a FORTRAN statement such as "INTEGER X".
- Specify data type(length) as a part of the choice of operands used. Here the information on length of operations is specified when the data is used - thus the program has a bit of extra redundancy in its notation (the extra characters needed to specify this type information) but the operations performed are much more visible at the local level.

The choice I made was for the second alternative, primarily to reduce the need for a symbol table to the barest minimum of information - consistent with the simplifications needed for a compact assembler or hand compilation. A secondary reason is the one stated in "b" - local type indications give a better documented program. In the integer operations used by programs in SIRIUS, a single colon (as in "AND:") is used to indicate where an 8-bit operation is involved, and a double colon (as in "AND::") is used to indicate the 16-bit form of an operation. A final comment on integers: where a signed integer representation is required in two's complement notation, the sign of the number is represented by the most significant bit (bit 7 of length 8 words, bit 15 of length 16 words.) This is the bit tested by the "S" flag on the various microcomputers.

Byte String Data: One additional data type will be required for programming the various microcomputers using SIRIUS-MP. This data type is the generalized concept of a "byte string." The representation is designed for manipulation of blocks of data in memory, in a form consisting of a length byte at the "anchor" (starting address) of the string, followed by from 0 to 255 data bytes at consecutive addresses. This is a format which is identical to that used in many byte oriented compilers (eg: XPL) and is a virtual necessity for handling character texts. Applications will not be restricted to character texts, however, for one particular use could include variable length decimal arithmetic using packed BCD byte strings.



Byte strings are most conveniently handled on computers which have byte addressability of memory locations - eg: the IBM 360/370 series as well as the smaller (8080, 8008, 6800) microcomputers. For 16 bit minicomputers and microcomputers, the concept is still useful, but requires explicit address calculations as a part of unpacking and manipulating two bytes per word. Operations on byte strings will use the notation of a number sign "#" to indicate the variable number of bytes involved.

OPERATIONS:

With the above introduction regarding data representations, it is now possible to consider the basic operations possible. The list here represents those used in the notation of the programs in this issue. In a later issue I'll expand the explanations of some of these operations and corresponding machine code for typical machines. There are also several operations which I have not used in the notation of the current set of programs, but which will be the subject of future notes in this area. The following is a list of the operations used with program notation in this issue, omitting the type indicators :

AND	GOTO	INPUT
Assignment(=)	HALT	IOEXCH
CALL	IF	KEYWAIT
CLEAR	IFNOT	OR
DECR	INCR	OUTPUT

The operations AND, OR, GOTO, HALT, INPUT and OUTPUT all have direct analogs in the CPU operations when 8-bit quantities are used with machines such as the 8008, 8080 or 6800. The examples' 8008 generated code versions illustrate one such representation. Some further notes will help illuminate the code generation process for the other operations.

For all operations which have direct analogs in the machine architecture, the code used for the machine level version must consist primarily of establishing the addressability of operands (source and target) and then execution of the operation. This process is illustrated in the several examples. For 8 bit machines with 16 bit operations, the code generated must be generalized to 16 bits - for the 8008 this is done in the illustrated programs by appropriate subroutines for increment, decrement and comparison, so code generation consists of writing down machine codes for a subroutine call and argument linkage.

Assignment always will map into a sequence of operations needed to move data from the source to the target. The 8008 generated code of these examples is an extension of the previously described symbol table mechanism for address lookup (see February 1975 ECS.) For 16 bit quanta this process can often be done using a CPU register pair for the 8 bit machines, but will invariably require a subroutine when byte strings are involved.

The IF statement form used in the examples is found in both a negative and positive sense. In either case the TARGET (lefthand) operand is the place where execution will go if the condition tests true. Two forms of the condition (SOURCE) operand are used:

- a. Flag Reference: Here the intent is to use a mnemonic key word, for example "ZERO" to reference one of the CPU flags of a typical micro after an instruction which might alter such flags.
- b. Tests: Here the intent is to specify two operands symbolically which are to be compared. I have grouped such references in parenthesis to simplify mechanical interpretation by a compiler, and have used the assignment symbol "=" with its length code with the usual duplicity to indicate the comparison test operation.

A disclaimer is appropriate at this point - I am not satisfied with the IF condition test format illustrated in these examples of several programs, and will be experimenting with some alternatives.

GENERATION OF CODE:

The semantic intent of the language forms used to represent the several programs in this issue can be deduced from the comments in the listings and the general descriptive information in the previous pages. One remaining problem is the generation of code. For the time being, I am limiting information on this (very large) subject to the examples illustrated below for an 8008 case and the notes accompanying the examples. I think there is sufficient information content to facilitate interpretation and generation of corresponding machine code for processors such as the 8080 (very close) or the 6800.

BOOTER: AN "EMERGENCY" BOOTSTRAP LOADER

The first example of a SIRIUS-MP program is a short and self-contained program called "BOOTER." All programs ultimately solve problems. This particular program solved a problem which I had one weekend, and served as an "acid test" of the utility of the ECS-8 tape interface. As soon as I had the interface software up and running (the dump portion presented in March ECS's pages) I began dumping the entire CPU software load to cassettes at regular intervals as a "failsafe" against Boston Edison's next power failure. The planning for that contingency - which by the way did happen in an ice storm in January to my consternation - paid off in a different way: I made the foolish mistake of turning off the power via a switch on my bench, now taped over solidly. Since I was working on SIRIUS-MP as a program writing tool, I took the opportunity to test out the expression it provides by writing the BOOTER source program appearing at the top of the next page. I won't claim perfection, however the original form of the program was essentially the same as the listing illustrated.

Loading is accomplished as follows: in the tape format described in the last issue, the first legitimate data is the length code (two bytes which I knew had "007" and "377" values for my tapes.) Since none of the tape spacing and preparation routines of the IMP program would be available in the blank computer memory being bootstrapped, the only way to synchronize tape data with the program was to listen continuously for the "007" character (state 1, LOOKFIRST tests for "007"), then check for a succeeding "377" byte (state 2, WELLMAYBE tests for "377"), then commence loading bytes starting at

The BOOTER program, listed in SIRIUS-MP...

```

1  BOOTER:      B      =:      1      * INITIAL STATE IS 1
2  X            =:      2000     * (INTELESE 004/000) START ADDR
3  36          OUTPUT  377      * TURN ON A DISPLAY
4  CLEAR      A
5  IOEXCH     4      * RESET THE IO UNIT
6  BLOOP:     A      =:      27     * "0001 01 1 1" UNIT CONTROL
7  A          IOEXCH  4      * CHECK STATUS OF TAPE
8  A          AND:    140     * MASK OFF RDY & RDA BITS
9  BLOOP      IFNOT   (A=:140) * LOOP BACK UNTIL READY
10 GETCHAR:   M(X)    INPUT  2      * READ THE DATA (NO EXCHANGE)
11 DECR:      B
12 LOOKFIRST  IF      ZERO     * HAVE STATE 1 DETECTED
13 DECR:      B
14 WELLMAYBE  IF      ZERO     * HAVE STATE 2 DETECTED
15 DECR:      B
16 FORSURE    IF      ZERO     * HAVE STATE 3 DETECTED
17 HALT      * (OOPS! SHOULDN'T GET HERE)
18 36          OUTPUT  M(X)     * WRITE TO DISPLAY
19 37          OUTPUT  L(X)     * LOW ORDER ADDR TO DISPLAY
20 INCR:      X
21 B          =:      3      * RESET STATE 3 INDICATION
22 GOTO      BLOOP    * BACK FOR MORE INDEFINITELY
23 B          =:      1      * DEFAULT STATE 1 CONTINUE
24 BLOOP      IFNOT   (A=:007) * LOOK FOR OCTAL "007"
25 B          =:      2      * IF FOUND, STATE SET TO 2
26 GOTO      BLOOP    * AND GO BACK TO FIND "377"
27 WELLMAYBE: B      =:      1      * DEFAULT BACK TO STATE 1
28 BLOOP      IFNOT   (A=:377) * LOOK FOR OCTAL "377"
29 B          =:      3      * MAIN LOAD LOOP IF FOUND NOW
30 GOTO      BLOOP

```

Variables

A : CPU register for I/O
 B : CPU register or mem.
 X : Address pointer (CPU)
 ZERO : CPU flag for zero result

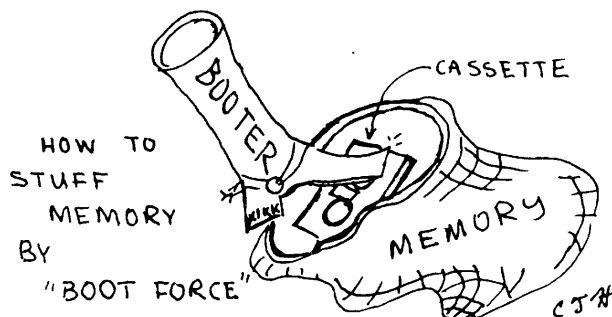
Notations

M(X) : memory at location in
 pointer variable X.

L(X) : low order 8 bytes of X

And the equivalent 8008 version of this algorithm....

Label	8008 Code Bytes	SIRIUS-MP Statement	Label	8008 Code Bytes	SIRIUS-MP Statement
BOOTER:	00 \110 = 016 LBI	s 1.	LOOKFIRST:	00 \166 = 016 LBI	s 23.
	00 \111 = 001 I			00 \167 = 001 I	
	00 \112 = 056 LHI	s 2.		00 \170 = 074 CPI	s 24.
	00 \113 = 004 h(LOAD POINT)			00 \171 = 007 7	
	00 \114 = 066 LLI			00 \172 = 110 JFZ BLOOP	
	00 \115 = 000 i(LOAD POINT)			00 \173 = 123 L	
	00 \116 = 006 LAI	s 3.		00 \174 = 000 H	
	00 \117 = 377 377			00 \175 = 016 LBI	s 25.
	00 \120 = 175 OUT36			00 \176 = 002 2	
	00 \121 = 250 XRA	s 4.		00 \177 = 104 JMP BLOOP	s 26.
BLOOP:	00 \122 = 111 IN4	s 5.		00 \200 = 123 L	
	00 \123 = 006 LAI	s 6.		00 \201 = 000 H	
	00 \124 = 027 "0001 01 11"		WELLMAYBE:		
	00 \125 = 111 IN4	s 7.		00 \202 = 016 LBI	s 27.
	00 \126 = 044 NDI	s 8.		00 \203 = 001 I	
	00 \127 = 140 "01 100 000"			00 \204 = 074 CPI	s 28.
	00 \130 = 074 CPI	s 9.		00 \205 = 377 377	
	00 \131 = 140 "01 100 000"			00 \206 = 110 JFZ BLOOP	
	00 \132 = 110 JFZ BLOOP			00 \207 = 123 L	
	00 \133 = 123 L			00 \210 = 000 H	
	00 \134 = 000 H			00 \211 = 016 LBI	s 29.
	00 \135 = 113 IN5 (Read Tape)	s 10.		00 \212 = 003 3	
	00 \136 = 370 LMA			00 \213 = 104 JMP	s 30.
	00 \137 = 011 DCB	s 11.		00 \214 = 123 L	
	00 \140 = 150 JNZ LOOKFIRST	s 12.		00 \215 = 000 H	
	00 \141 = 166 L				
	00 \142 = 000 H				
	00 \143 = 011 DCB	s 13.			
	00 \144 = 150 JNZ WELLMAYBE	s 14.			
	00 \145 = 202 L				
	00 \146 = 000 H				
	00 \147 = 011 DCB	s 15.			
	00 \150 = 150 JNZ FORSURE	s 16.			
	00 \151 = 154 L				
	00 \152 = 000 H				
	00 \153 = 377 HALT	s 17.			
FORSURE:					
	00 \154 = 307 LAM	s 18.			
	00 \155 = 175 OUT36				
	00 \156 = 306 LAL	s 19.			
	00 \157 = 177 OUT37				
	00 \160 = 055 NEXTA	s 20.			
	00 \161 = 016 LBI	s 21.			
	00 \162 = 003 3				
	00 \163 = 104 JMP BLOOP	s 22.			
	00 \164 = 123 L				
	00 \165 = 000 H				



the known load point (location 2000g = inteles 004/000) as initialized at the beginning of the program.

The program is a "state driven" algorithm which has 3 states of execution set by the content of the variable "B" (which maps into a register in the generated code for a microcomputer such as the 8008 code illustrated.) The sequence of states during execution of the main loop "BLOOP" during normal execution is as follows:

Start: 1 1 1 1 1 1 1 1 1 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3, End

Scan for "007" —————

Found it, look for "377" —————

Found it, transfer any further bytes to memory —————

The program is set up so that if a false synchronization pattern is detected ("007" followed by any byte other than "377") the "WELLMAYBE" branch of the loop concludes "maybe not" and goes back to scanning the input. The reason for scanning in this manner is to enable the program to be started via an interrupt, after which you can turn on the manual controls of the tape drive confident that the invalid data produced by the MODEM/UART combination during the leader and start up periods will not be falsely interpreted as good data - the specific 16-bit pattern of two bytes involved is not likely to occur due to random noise.

The 8008 code corresponding to the BOOTER program's SIRIUS-MP notation is shown at the bottom of page 12 with symbolic notations of labels, mnemonic op codes and reference numbers to the SIRIUS-MP statements in the listing at the top of the page. The specific hardware assumptions used for this code are documented in previous ECS issues and are not repeated in detail here. For this simple program, the "X" data quantity (a memory pointer) is translated as the content of the H and L register pointer of the 8008. One of the restart routines defined in January ECS is utilized by the generated code - "NEXTA" calculates the next address in H and L. On an 8080 this could be performed without a subroutine using the INX instruction with H and L selected. On a 6800 the corresponding function would be performed using its INX instruction, with the variable X assumed to signify the index register "X".

BOOTER uses output instructions directed at a binary display to illustrate the progress of the program. At initialization, the display left half (OUT36) is loaded with 8 "on" bits. (SIRIUS statement 3). Then, following the synchronization detection, the data transfer branch FORSURE displays the current byte at left (OUT36, statement 18) and the current low order address at right (OUT37 generated by statement 19).

The small loop from statements 6 to 9 is used to cause the program to wait until the flags of the UAR/T subsystem (see article ECS-6 and January 1975 ECS) indicate that a character has been received. The tape unit control code "027₈" defined at statement 6 is used to signify the data rate ("0001" for 1210 baud), channel ("01") and selection for input (the last two bits.)

If you use BOOTER to load IMP from one of the cassettes supplied by M. P. Publishing Co. (\$7.50 each post paid) you will have to additionally load by hand the content of the other restart instructions routines before changing the interrupt branch to point to the IM1 entry point at location 013/000 (Inteles.)

IMP EXTENSIONS FOR TAPE INTERFACE CONTROL (Continued...)

In the March issue of ECS, I started a presentation of extensions to the Interactive Manipulator Program for tape block write, compare and read operations. This article contains the remainder of the listings. With the exception of the three routines on this page, the additional 8008 code is given in its SIRIUS-MP form and in absolute octal with mnemonics decoded.

One aspect of the SIRIUS-MP language which I have not dealt with explicitly in this issue's discussion is that of argument/parameter linkage for subroutine calls. Because a machine-dependent argument/parameter linkage is used for the 8008 versions of the three routines on this page, I present them here in the same commented listing form used for previous issues of ECS. The routines are utility functions for the two-byte increment/decrement functions and comparison. The parameter linkages to these routines are formed by passing symbols (see Feb. '75 ECS) in registers for lookup.

D2B is the two byte decrement operation, which is entered with the symbol of the operand contained in the 8008's A-register. The operand is decremented by subtraction due to the properties of a zero underflow (the Zero flag detects this state one number too early at 0, not -1.) On return, the carry flag indicates a 16-bit underflow if any.

I2B is the corresponding two byte increment operation, which is also entered with the symbol of the operand in the 8008's A register. The 8008's increment instructions are used, since the zero state is a reliable overflow indicator. On return, the zero flag indicates a 16-bit overflow if any.

C2B is a two byte comparison operation, with a more complicated linkage. The two operands are passed as symbols in the B and C registers. The result is passed back as the content of the "E" register: 1 if not equal, 2 if equal. This can be tested by a decrement instruction followed by a jump on zero.

D2B:

012\132 = 075 SYM	Go pick up argument address
012\133 = 060 INL	Point ahead (assume not at page bound)
012\134 = 307 LAM	Fetch the low order byte.
012\135 = 024 SUI	Subtract 1 - decrement will not do!
012\136 = 001 1	Save result
012\137 = 370 LMA	Return on no borrow condition.
012\140 = 003 RFC	Point to high order byte
012\141 = 061 DCL	Fetch it
012\142 = 307 LAM	Also decrement with subtract
012\143 = 024 SUI	so that borrow (C) may be set...
012\144 = 001 1	Save result
012\145 = 370 LMA	With carry indicating net underflow.
012\146 = 007 RET	

I2B: Routine to increment two bytes - enter with symbol parameter in A

011\313 = 075 SYM	Look up the parameter address
011\314 = 060 INL	Point to,
011\315 = 317 LBM	load from memory,
011\316 = 010 INB	increment and
011\317 = 371 LMB	save the low order byte.
011\320 = 013 RFZ	Return direct if no overflow
011\321 = 061 DCL	Point to,
011\322 = 317 LBM	load from memory,
011\323 = 010 INB	increment,
011\324 = 371 LMB	and save the high order byte.
011\325 = 007 RET	Then return always.

C2B: Routine to compare bytes - in two's. Enter with symbol parameters in registers B and C.

010\234 = 046 LEI	Return default 1 (not equal.)
010\235 = 001 1	
010\236 = 301 LAB	Fetch first parameter address
010\237 = 075 SYM	and fetch the parameter.
010\240 = 337 LDM	Fetch second parameter address
010\241 = 302 LAC	and compare against
010\242 = 075 SYM	first parameter value...
010\243 = 303 LAD	Return (E=1) if unequal.
010\244 = 277 CPM	Point to next address of second parm.
010\245 = 013 RFZ	Fetch second parm second byte
010\246 = 055 NEXTA	
010\247 = 337 LDM	
010\250 = 301 LAB	Point to first parm again
010\251 = 075 SYM	look NEXTA him too!!!
010\252 = 055 NEXTA	
010\253 = 303 LAD	Compare first parm, second byte
010\254 = 277 CPM	And again return (E=1) if unequal.
010\255 = 013 RFZ	Otherwise both bytes of both
010\256 = 046 LEI	two sets are equal and can
010\257 = 002 2	return with equality result.
010\260 = 007 RET	

The notational power of a more abstract method of programming is illustrated by comparing the expression of the new IMP extension segments on page 16 with the corresponding "generated code" for the 8008 printed later. The routines listed in SIRIUS-MP form for the tape extension begin with the main portion of the program...

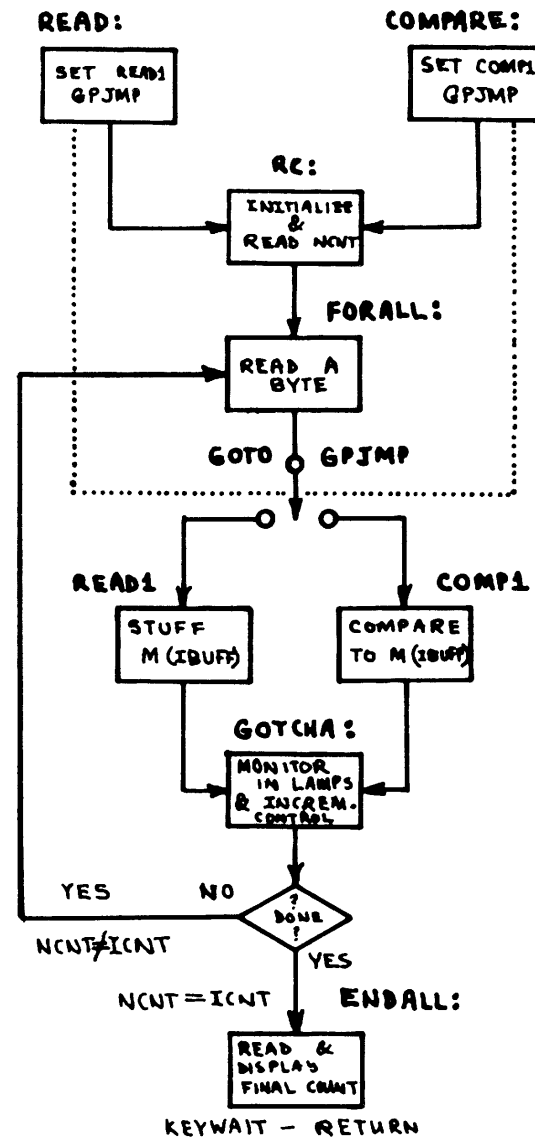
READ/COMPARE main routine is at the left hand side of page 16 held sideways. This 33-statement SIRIUS-MP program is invoked when the IMP command decoder detects a "shift R" for read or "shift C" for compare. The difference in the two routines is determined by the entry point - line 1 for READ, line 28 for COMPARE. The logic at the entry points sets up a jump address in the "GPJMP" indirect branch location (this overwrites the previous use of GPJMP to get to READ or COMPARE from IMP.) This switch (the choice of branch paths) is required so that the same general control flow can be used for both the READ and COMPARE operations - the difference being in what is done with the information read from tape. The switch point in the flow occurs at statement 14, and can be illustrated in flow chart terms by the diagram at the right.

The common portion of the program provides the overall structure of a read operation: initialize the UAR/T, read a dummy character at the first RDA time, read the two length code bytes written by the OUTCNT routine (see below) when the tape is prepared, then enter a loop which continues until the data count is exhausted.

When the READ1 branch of the flow is taken during a read operation, the current memory location pointed to by IBUFF receives the input character found in a variable called "B" (a CPU register for the 8008 version of the program.)

When the COMPI branch of the flow is taken during a compare operation, the current byte pointed to by IBUFF is compared to the input byte in the variable "B" - and an error count is incremented in the variable "BADATA" (16 bits worth) to keep a tally of the badnesses.

The data count is kept in the variable "ICNT" which starts out at -1 and is counted up until it equals the block count stored in "NCNT" after it is read from the tape. The test for end of transfer is found at statement 20, a SIRIUS "IFNOT" operation.



IMP program tape extensions expressed in a SIRIUS fashion...

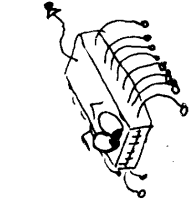
1	READ:	T(GPJMP)	==:	W(READ1)	* SET READ JUMP SWITCH	1	INPUT2:	A	TAPECTRL	* FETCH IO CONTROL WORD
2	RC:	TAPECTRL	OR:	"0000 00 1 1"	* FORCE INPUT SELECT	2	A	A	4	* EXCHANGE FOR STATUS
3			OR:	A		3	B	B		* SAVE STATUS IN B
4			IOEXCH	4	* RESET THE IO UNIT	4	A	A		"01 100 000" * MASK DESIRED BITS
5	INITIALIZE:		OUTPUT	TAPECTRL	* SET SELECTED CONTROL STUFF	5	INPUT2	B		(A="01 100 000") * WAIT TILL READY
6	IBUFF		==:	MEMADDR	* START INPUT AT MEMADDR	6	A	A		* RESTORE STATUS FROM B
7	ICNT		==:	-1	* INITIAL COUNT TO MATCH OUTPUT	7	A	A		"00 000 111" * MASK ERROR BITS
8	DUMMYIN:		CALL	INPUT2	* GO FETCH BYTE (WAIT LOOP)	8	INPUTIT	A		(A="00 000 111") * INVERTED NO ERRORS
9	HIGHLIGHT:		CALL	INPUT2	* GET HIGH ORDER LENGTH	9	INCR::	B		BADFORM * INCREMENT DATA FORMAT ERRORS
10	NCNT		==:	B	* SAVE B INPUT IN NCNT H.O.	10	INPUT	A	5	* READ THE LATEST CHARACTER
11	LOWLENGTH:		CALL	INPUT2	* GET LOW ORDER LENGTH	11	A	A		* PASS BACK VIA B REGISTER
12	NCNT(1)		==:	B	* STORE AT NCNT+1	12	RETURN			* BACK TO CALLER
13	FORALL:		CALL	INPUT2	* NORMAL DATA BYTE FETCH					
14			GOTO	GPJMP	* SELECT COMPARE OR READ VIA					
					* VARIABLE JUMP TARGET					
15	READ1:	M(IEUFF)	==:	B	* IF READ THEN STORE IT					
16	GOTCHA:	37	OUTPUT	B	* DISPLAY INPUT DATA					
17		36	OUTPUT	0	* CLEAR OTHER DISPLAY TO ZERO					
18			INCR::	0	* POINT TO NEXT INPUT ADDRESS					
19			INCR::	ICNT	* INCREMENT WORKING COUNT					
20	FORALL		IFNOT	(ICNT==NCNT)	* TEST END OF BLOCK					
21	ENDALL:									
22		36	CALL	INPUT2	* READ FINAL LENGTH BYTE					
23			OUTPUT	B	* AND DISPLAY					
24		37	CALL	INPUT2	* READ SECOND FINAL LENGTH BYTE					
25			OUTPUT	B	* AND DISPLAY IT TOO					
26		4	TAPECTRL	"1111 11 0 0"	* TURN OFF INPUT SELECT					
			OUTPUT	TAPECTRL	* TURN OFF THE DRIVE... PATCH					
					* IN A 2 SECOND WAIT HERE					
					* IF NEEDED - SEE TEXT...					
					* SLEEP PERCHANCE TO DREAM					
27	COMPARE:		KEYWAIT							
28	T(GPJMP)		==:	W(Comp1)	* SET COMPARE JUMP SWITCH					
29	BADDATA		==:	0	* ZERO OUT BAD DATA...COUNT					
30			GOTO	RC	* ENTER NORMAL FLOW					
31	COMPL:		IF	(M(IEUFF)=B)	* TEST TAPE AGAINST MEMORY					
32			INCR::	BADDATA	* MISSED SOME BITS!!!					
33			GOTO	GOTCHA	* BACK FOR MORE...					

Note: Reference numbers to SIRIUS statements are provided at the local level for each block of functional code illustrated here. They correlate to the 8008 examples of executable machine codes, within each block.

Notations: T(GPJMP) : address part of jump

W(READ1) : mem. address of READ1

NAME(n) : nth byte of NAME



8008 Generated Code for READ/COMPARE routines (p. 16, left)

Label	8008 Code Bytes	SIRIUS-MP Statement	Label	8008 Code Bytes	SIRIUS-MP Statement
READ:			READI:		
	004\000 = 006 LAI	s 1.	GOTCHA:	004\107 = 371 LMB	s 15.
	004\001 = 010 s(GPJMPAL)			004\110 = 301 LAB	s 16.
	004\002 = 075 SYM			004\111 = 177 OUT37	
	004\003 = 076 LMI			004\112 = 250 XRA	s 17.
	004\004 = 107 L(READI)			004\113 = 175 OUT36	
	004\005 = 060 INL			004\114 = 006 LAI	s 18.
	004\006 = 076 LMI			004\115 = 020 s(IBUFF)	
	004\007 = 004 H(READI)			004\116 = 106 CAL I2B	
RC:				004\117 = 313 L	
	004\010 = 006 LAI	s 2.		004\120 = 011 H	
	004\011 = 014 s(TAPECTRL)			004\121 = 006 LAI	s 19.
	004\012 = 075 SYM			004\122 = 016 s(ICNT)	
	004\013 = 307 LAM			004\123 = 106 CAL I2B	
	004\014 = 064 ORI			004\124 = 313 L	
	004\015 = 003 "00000011"			004\125 = 011 H	
	004\016 = 370 LMA	s 3.		004\126 = 016 LBI	s 20.
	004\017 = 250 XRA	s 4.		004\127 = 016 s(ICNT)	
INITIALIZE:				004\130 = 026 LCI	
	004\021 = 006 LAI	s 5.		004\131 = 022 s(NCNT)	
	004\022 = 014 s(TAPECTRL)			004\132 = 106 CAL C2B	
	004\023 = 075 SYM			004\133 = 234 L	
	004\024 = 307 LAM			004\134 = 010 H	
	004\025 = 111 IN4	s 6.		004\135 = 041 DCE	
	004\026 = 006 LAI			004\136 = 150 JTZ FORALL	
	004\027 = 006 s(MEMADDR)			004\137 = 074 L	
	004\030 = 075 SYM		ENDALL:	004\140 = 004 H	
	004\031 = 317 LBM			004\141 = 106 CALL INPUT2	s 21.
	004\032 = 060 INL			004\142 = 061 L	
	004\033 = 327 LCM			004\143 = 012 H	
	004\034 = 006 LAI			004\144 = 301 LAB	s 22.
	004\035 = 020 s(IBUFF)			004\145 = 175 OUT36	
	004\036 = 075 SYM			004\146 = 106 CALL INPUT2	s 23.
	004\037 = 371 LMB			004\147 = 061 L	
	004\040 = 060 INL			004\150 = 012 H	
	004\041 = 372 LMC			004\151 = 301 LAB	s 24.
	004\042 = 006 LAI	s 7.		004\152 = 177 OUT37	
	004\043 = 016 s(ICNT)			004\153 = 006 LAI	s 25.
	004\044 = 075 SYM			004\154 = 014 s(TAPECTRL)	
	004\045 = 006 LAI			004\155 = 075 SYM	
	004\046 = 377 "IIIIIIII"			004\156 = 307 LAM	
	004\047 = 370 LMA			004\157 = 044 NDI	
	004\050 = 060 INL			004\160 = 374 "II III 100"	
	004\051 = 370 LMA			004\161 = 370 LMA	
DUMMYIN:				004\162 = 111 IN4	s 26.
	004\052 = 106 CAL INPUT2	s 8.	COMPARE:	004\163 = 025 KEYWAIT	s 27.
	004\053 = 061 L			004\164 = 006 LAI	s 28.
	004\054 = 012 H			004\165 = 010 s(GPJMPAL)	
HIGHLNGTH:				004\166 = 075 SYM	
	004\055 = 106 CAL INPUT2	s 9.		004\167 = 076 LMI	
	004\056 = 061 L			004\170 = 206 L(COMPI)	
	004\057 = 012 H			004\171 = 060 INL	
	004\060 = 006 LAI	s 10.		004\172 = 076 LMI	
	004\061 = 022 s(NCNT)			004\173 = 004 H(COMPI)	
	004\062 = 075 SYM			004\174 = 006 LAI	s 29.
	004\063 = 371 LMB			004\175 = 024 s(BADDATA)	
LOWLNGTH:				004\176 = 075 SYM	
	004\064 = 106 CAL INPUT2	s 11.		004\177 = 250 XRA	
	004\065 = 061 L			004\200 = 370 LMA	
	004\066 = 012 H			004\201 = 060 INL	
	004\067 = 006 LAI	s 12.		004\202 = 370 LMA	
	004\070 = 022 s(NCNT)			004\203 = 104 JMP RC	s 30.
	004\071 = 075 SYM			004\204 = 010 L	
	004\072 = 055 NEXTA			004\205 = 004 H	
	004\073 = 371 LMB		COMPI:		
FORALL:				004\206 = 301 LAB	s 31.
	004\074 = 106 CALL INPUT2	s 13.		004\207 = 277 CPM	
	004\075 = 061 L			004\210 = 150 JTZ GOTCHA	
	004\076 = 012 H			004\211 = 110 L	
	004\077 = 006 LAI			004\212 = 004 H	
	004\100 = 020 s(IBUFF)	Globally optimized code moved ahead of the GPJMP		004\213 = 006 LAI	s 32.
	004\101 = 106 CALL MEMSYM			004\214 = 024 s(BADDATA)	
	004\102 = 002 L			004\215 = 106 CAL I2B	
	004\103 = 012 H			004\216 = 313 L	
	004\104 = 104 JMP GPJMP	s 14.		004\217 = 011 H	
	004\105 = 015 L			004\220 = 104 JMP GOTCHA	s 33.
	004\106 = 000 H			004\221 = 110 L	
				004\222 = 004 H	

8008 Generated Code for MISCELLANEOUS routines (p16, right)

Label	8008 Code Bytes	SIRIUS-MP Statement #	Label	8008 Code Bytes	SIRIUS - MP Statement #
INPUT2:			ONOFF:		
012N061 = 006 LAI		s 1.	011N264 = 006 LAI		s 1.
012N062 = 014 s(TAPECTRL)			011N265 = 014 s(TAPECTRL)		
012N063 = 075 SYM			011N266 = 075 SYM		
012N064 = 307 LAM			011N267 = 307 LAM		
012N065 = 111 IN4		s 2.	011N270 = 044 NDI		s 2.
012N066 = 310 LBA		s 3.	011N271 = 002 "00 000 010"		
012N067 = 044 NDI		s 4.	011N272 = 150 JTZ TON		s 3.
012N070 = 140 "01 100 000"			011N273 = 302 L		
012N071 = 074 CPI		s 5.	011N274 = 011 H		
012N072 = 140 "01 100 000"			TOFF:		
012N073 = 110 JTZ INPUT2			011N275 = 016 LBI		s 4.
012N074 = 061 L			011N276 = 000 0		
012N075 = 012 H			011N277 = 104 JMP EITHER		s 5.
012N076 = 301 LAB		s 6.	011N300 = 304 L		
012N077 = 044 NDI		s 7.	011N301 = 011 H		
012N100 = 007 "00 000 111"			TON:		
012N101 = 074 CPI		s 8.	011N302 = 016 LBI		s 6.
012N102 = 007 "00 000 111"			011N303 = 002 2		
012N103 = 150 JTZ INPUTIT			EITHER:		
012N104 = 113 L			011N304 = 307 LAM		s 7.
012N105 = 012 H			011N305 = 044 NDI		s 8.
012N106 = 006 LAI		s 9.	011N306 = 374 "11 111 100"		
012N107 = 026 s(BADFORM)			011N307 = 261 ORB "xx xxx xBo"		s 9.
012N110 = 106 CALL 12B			011N310 = 370 LMA		s 10.
012N111 = 365 L			011N311 = 111 IN4		s 11.
012N112 = 010 H			011N312 = 025 KEYWAIT		
INPUTIT:					
012N113 = 113 IN5		s 10.			
012N114 = 310 LBA		s 11.			
012N115 = 007 RETURN					
OUTCOUNT:					
012N200 = 104 JMP NEWOUTCNT	Here is a patch to get to the				
012N201 = 116 L	new version of OUTCOUNT.				
012N202 = 010 H					
NEWOUTCNT:					
010N116 = 016 LBI		s 1.			
010N117 = 017 1510					
010N120 = 106 CALL WAITCS		s 2.			
010N121 = 116 L					
010N122 = 012 H					
010N123 = 006 LAI		s 3.			
010N124 = 022 s(COUNT)					
010N125 = 075 SYM					
010N126 = 307 LAM					
010N127 = 310 LBA		s 4.			
010N130 = 113 IN5		s 5.			
010N131 = 106 CALL WAITOUT		s 6.			
010N132 = 147 L					
010N133 = 012 H					
010N134 = 006 LAI		s 7.			
010N135 = 022 s(COUNT)					
010N136 = 075 SYM		s			
010N137 = 060 INL					
010N140 = 307 LAM					
010N141 = 320 LCA		s 8.			
010N142 = 113 IN5		s 9.			
010N143 = 106 CALL WAITOUT		s 11.			
010N144 = 147 L					
010N145 = 012 H					
010N146 = 007 RETURN		s 12.			
Patches to Previous Code			Tape Extension VARIABLES (in order of appearance)		
TAPECMDS:			GPJMP, symbol 10		
012N352 = 317 "0"			TAPECTRL, symbol 14		
012N353 = 321 L(JONOFF)			A, CPU register		
012N272 = 012	"34" is TAPECMDS (new value)		MEMADDR, symbol 06, input to tape transfers.		
012N273 = 352			IBUFF, symbol 020		
JONOFF:			ICNT, symbol 016		
012N321 = 104 JMP ONOFF	IMP entry to the		NCNT, symbol 022		
012N322 = 264 L	ONOFF routine sand-		B, CPU register		
012N323 = 011 H	wiched in spare bytes.		BADDATA, symbol 24		
READJ:			BADFORM, symbol 26		
013N313 = 104 JMP READ	New IMP READ		COUNT, symbol 22		
013N314 = 000 L	entry address in		ZERO, CPU flag		
013N315 = 004 H	this jump.		Note: NCNT, COUNT are equivalent; ICNT and TCOUNT (see March ECS) are equivalent.		
COMPJ:					
013N316 = 104 JMP COMPARE	New IMP COMPARE				
013N317 = 164 L	routine entry address				
013N320 = 004 H	now in this jump.				

The INPUT2 subroutine is at the top right hand side of page 16 held sideways. This 12-statement SIRIUS-MP subprogram is invoked by a subroutine CALL whenever another program wants to "read" a byte from the tape unit according to the content of TAPECTRL. The reading method incorporated in the software of IMP to date is a "polling" technique in which a loop tests status bits of the I/O device (UAR/T "RDA" and a motor turn-on oneshot "ready" signal.) The loop consists of SIRIUS-MP statements 1 to 5 of INPUT2. The routine breaks out of the loop, reads the data and returns with the data byte in the variable "B" (a register in the 8008 generated code). The three UAR/T reception status bits (parity error/framing error/overflow error) are checked and an error count in BADFORM is incremented if no errors are detected.

The OUTCOUNT routine of the March issue of ECS was modified to improve performance in the course of rewriting the comparison software in SIRIUS for this issue. The problem with the original version was the fact that an explicit output wait is required for reliable reading of the data. Thus a patch is placed at location 012/200 to jump to the new version of the program, loaded in some spare memory address space at 010/116. The NEWOUTCNT has two changes: a) I increased the time delay before output to 1.5 seconds (SIRIUS statements 1 and 2); b) I have inserted calls to WAITOUT after each output of a byte (SIRIUS statements 5 and 9 of NEWOUTCNT.)

The ONOFF routine is a new routine added to support a new tape control command, "TO" entered from the keyboard device. The idea here is to have a way to turn on the motor for purposes of listening to data with the ear, for rewinds of long duration, or for recording non-digital comments with the cassette recorder's built-in microphone. The ONOFF routine itself is very simple, comprising a set of 12 SIRIUS statements which map into 23 8008 bytes in the sample generated code. The "TO" function complements the current state of the motor control bit in TAPECTRL and outputs the result to currently selected tape drive via the "IN4" instruction connected to the tape controller.

In setting up to run IMP with the new extensions, the patches to TAPECMDS, JONOFF, and READJ/COMPJ locations of IMP must be made as indicated in the detail listing of page 18. The TAPECMDS table is extended for the new "O" subcommand by starting it one byte earlier; the symbol table symbol "34" for TAPECMDS is adjusted to reflect this addition. The new execution jump JONOFF is added to get the program into the ONOFF routine, and the READJ/COMPJ jumps are changed to reflect altered placement of these routines from the original layout. One other change is required to the symbol table published previously: the address of symbol "20" should be changed to "220" in byte 012/301 of the 8008 code. This symbol has been changed from its original use and now becomes the memory pointer "IBUFF" with two bytes instead of the original 1 byte of reserved space.

COMMENTS ON THE ECS-8 DESIGN:

The output of the TSI (serial data to the computer interface) line is not suitable for an interrupt driven UAR/T software interface without use of some masking logic. The problem is this: the FSK input decode is done by the phase lock loop of the XR-210. When null inputs (eg: tape leader period, or any time without a mark signal) occur, the phase lock loop hunts around for a lock - thus causing the comparator to have its input switch back and forth with the result being a digital noise signal on the TSI line. If the UART is listening, it will decode erroneous characters in this mode. The software of this article ignores the problem by not listening unless good data is coming.

This article begins a regular series of information and commentary on the use of the Intel 8080 in an ECS context, with occasional specific reference to packaged systems such as the MITS Altair product. In addition to the MITS product, there is at least one other source of the 8080 chips and boards advertising in the pages of Radio Electronics/Popular Electronics. This first installment concerns some general comments on the 8080 instruction set and specific suggestions concerning 16-bit arithmetic operations (addition/subtraction) in applications other than address calculations.

AQ-1.1: Addressing Modes.

One of the most basic questions to be asked whenever you ponder the use of a new computer instruction architecture is "what are its addressing modes?" The answers all lie in the hardware designer's backyard whenever a specific existing machine such as the 8080 is considered. How do I get at the data in memory when I want to perform some operation in the machine? Are there different ways of reaching the same data item? And so on. The effects of addressing and data reference will color the whole process of generating programs for the architecture of the machine in question. For instance, if the machine is a "stack machine" (not a machine with a stack, but one designed for operations between stack elements) then the addressing can almost exclusively be implied by the way operations are done. On such a machine, the only bits needed for an instruction are the data bits which specify an operation. But in the real world of existing and implemented machines available to the ECS type of application, the coloring of coding is much more conventional - addressing is performed as part of the instruction or as part of an implied setup in a CPU register under program control. In the Intel 8080 (as in the 8008) the design of addressing modes is a fairly arbitrary pot-pouri of methods fraught with special cases not amenable to concise summary without losing information. In order to write programs these addressing modes must be known and understood so that the best of alternatives (if any) can be evaluated and used in a given programming situation. In the comments below, a few of the conventional addressing concepts in computer designs are isolated and illustrated with regard to the 8080.

AQ-1.2: Immediate Addressing.

Immediate data addressing exists in some form in most contemporary computers, with the usual definition being a constant bit pattern of one word length, following the operation code in a program. The 8080 includes this form of addressing with all the immediate operations which exist on its antecedent the 8008, plus some extensions which make the architecture more useful as a general purpose computing element. The primary extension of immediate addressing is to the inclusion of a long (16-bit) form of the concept in certain limited classes of move (load/store) operations with respect to CPU registers. The 8080 partitions 6 of the 7 CPU registers into three pairs "index registers" which may be loaded with 16-bit numbers using immediate addressing. The primary intention of such operations is the loading of an address, but programmers can and

do use operations for whatever purpose is required to solve a problem - so whenever one needs a 16-bit "literal" data item this form of double byte immediate operation can be used to load CPU registers.

One particular use of the two-word immediate form in its intended application is the initialization of the stack pointer as a part of setting up execution of a program. In large scale systems the equivalent of a stack pointer (ie: system defined addressing parameters) is usually determined by the "operating system" prior to the call which invokes a user-program. But in your use of a microcomputer of the 8080 (or Motorola 6800) design, with minimal software, you can make no assumptions about the initialization. To be used, the stack must exist in random access read/write memory so that the temporary linkage data associated with the CALL operation and its arguments can be stored. In order for this linkage to occur, the stack pointer (SP) must point to the RAM area. One way to initialize the stack pointer following the start of execution is contained in the following SIRIUS-MP notation and its 8080 translation:

SIRIUS:		8080:
SP	==: location	LXI SP, location

In both instances, the "location" is the 16-bit integer number which is the address of the stack area.

AQ-1.3: Absolute Addressing.

The design of a computer instruction set involves many trade-offs, the evaluation of options with inputs ranging from the preferences of programming individuals to the physical constraints of the LSI chip. In the best of all possible programming worlds, one would like to see a consistent set of addressing modes applicable in principle to any of the basic operations possible. In particular, a more extended use of an absolute (in-instruction stream) form would be desirable than has been implemented with the 8080. There are two basic operations available in the 8080 instruction set which reference memory from within the instruction stream. These are the load (LDA, LHLD) and store (STA, SHLD) operations in 8 and 16 bit variations. For program code which involves fixed data areas at locations allocated by hand or by an assembler/compiler, these operations will be used extensively to prepare data for the execution of actual "work" -since the actual work cannot reference memory directly. The use of load and store for this purpose is highly conventional in many minicomputers, although usually at least one of the algebraic/logic operation operands can be acquired by a direct or indirect memory reference in the instruction stream. (As a point of contrast, the Motorola 6800 microcomputer can perform most of its arithmetic/logical operations with one in-instruction address reference to memory.)

AQ-1.4: Pointer Addressing.

One area where the 8080 has some excellence is in the number of CPU registers it has and the fact that three different pairs can be used as "index registers" for fetching

data to an accumulator (all pairs) or referencing memory operands (H/L only) of the arithmetic operations. It is thus fairly easy to keep pointers around locally in the CPU without the need to transfer them to another location when making a reference based upon the index. The pointers are, however, only good for one operation in general - referencing data in load/store situations, and thus not as useful as they might otherwise have been. The memory reference modes of all the 8-bit arithmetic and logical instructions use one of these pointers, the H/L register pair, to address the one memory operand (the implied second operand is the accumulator register A.) All the procedures and tricks applicable to setting up H/L pointer addresses in the earlier 8008 microcomputer design apply as well to the equivalent H/L forms of the 8080.

One particular programming trick which will prove useful in manipulating blocks of data involves the use of one pointer pair - D/E - to point to one operand block and a second pointer pair - H/L to reference the second block. Suppose the problem is to "AND" all the bytes of one block with the bytes of another and to store the result in the second. The basic set of instructions used to set up the loop would be:

LXI D	address 1	
LXI H	address 2	set up addresses

With this setup, the heart of a loop to transfer the data with an AND condition as required by the problem statement would be:

MOV, A, M	Fetch first operand byte
XCHG	Establish second operand address, but save first operand address
ANA M	AND with second byte
MOV M, A	Save in second operand byte
INXH	Increment address
XCHG	Move back in exchange
INXH	Increment address

This code does not include the instructions needed to establish a loop - to transfer a block with this operation would require a loop count and loop count decrement followed by conditional test for continuation.

This same general scheme of switching the D/E with H/L registers can be used quite widely your program must step simultaneously through two regions of memory. The technique only works with D/E & H/L unless you want to take a calculated risk and exchange with the stack pointer instead of D/E.

AQ-1.5: 16-Bit Operations & 16-Bit Addition/Subtraction.

The 8080 has a specific and limited set of 16-bit operations which can be used to some advantage both for the intended purpose (address calculation and setup) and in more general problems. The 16-bit operations are ...

16-bit Load and Store between register pairs and memory or immediate
(Load only) data.

16-bit Addition intended for address calculation.

16-bit Increment/Decrement useful in loop counting & address changing.

For the more general usage of the 16-bit addition operation in programs requiring the extended precision addition / subtraction, the H/L register pair can be treated as if it were a 16-bit accumulator for the purposes of calculation with the actual results being stored ultimately in memory operands. The boxes below illustrate two calculations in 16 bit precision, under the following assumptions:

- Variable P is a two-byte operand at locations P and P + 1.
- Variable Q is a two-byte operand at locations Q and Q + 1.
- The content of A, H and L registers is irrelevant prior to and following the calculation.
- Absolute addressing will be used with the result stored back in P, as if P were a "software accumulator."

Note the differences in the size of the little routines involved - for the addition case, the setup and execution is fairly compact. For subtraction the need to form the two's complement negative of the Q operand complicates the picture...

The SIRIUS-MP statement:		P +:: Q * 16-BIT ADD
generates...		
LHLD	Q	Get first operand bytes to Q
XCHG		Move first op to D/E
LHLD	P	Get second operand (soft. accum.)
DADD		Add Q to P giving P
SHLD	P	Store result back into new P value

The SIRIUS-MP statement:		P -:: Q * 16-BIT SUBTRACT
generates...		
LDA	Q	Get first byte, negative operand.
CMA		Complement it
MOV	D, A	Move it to D of D/E pair.
LDA	Q+1	Get second byte, negative operand.
CMA		Complement it.
MOV	E, A	Move it to E of D/E pair.
INX	D	Increment complement giving -Q value
LHLD	P	Get software accumulator value
DADD		Value of P - Q now in H/L
SHLD	P	Save back in software accumulator.

After either of these operations, the carry flag can be tested to find out if an overflow occurred, thus in principal allowing extended precision of greater precision than 16 bits.

One particular 16-bit operation may prove of use in certain contexts. This is the 16-bit addition of the H/L register pair to itself by means of the DADH instruction. There are two instances where this variation of 16-bit addition stands out for potential utility:

- Suppose I want to address an extended array of data kept in 2, 4, 8 or 2^n byte quanta. The shift properties of this addition (it multiplies H/L by 2) can be used "n" times to modify an integer array index ala FORTRAN or PL/1 into a useful address offset.
- This left shift operation can form the basis of an integer multiply operation.

AQ-1.6 A Ceremonial "Nit":

It serves no good end to act the part of a contentious critic, but... at the risk of being in the position of a pot calling the kettle black I do protest MITS' use of the Anquish Languish (technical dialect) in the Altair 8800 manual I examined recently:

Implement: This verbalized noun is conventionally used in technical contexts such as "to implement a system." (Ie: to create the system.) A computer designer implements an LDA or STA instruction; the programmer codes said implemented instruction (ie: selects it) as part of his own process of implementing a software system. Programmers never use unimplemented instructions as a matter of course. (If you take Webster literally one might come out with the MITS definition of the term implement.)

Variance: A variance exists and is defined in the legalese terminology of "obtaining a variance (exception)" to some law by bootlicking and bribing the appropriate petty bureaucrats. It is also the square of the standard deviation in the terminology of statistics. A variance is not a variation on an instruction's operation, that is unless one wished to redefine conventional usage.

I have been collecting reports from several subscribers on the Altair product and with the exception of what appear to be relatively minor technical problems, most purchasers of the system indicate satisfaction with the product and service on it.

ERRATUM:

Charles S. Lovett receives a one issue subscription extension for being the first subscriber to report an error in the ECS-7 design article of February 1975 ECS. The line from pin 2 of IC -14- which is shown connected to ground should instead have been a .01 mfd capacitor to ground. (Switch S1 would have no effect if wired as drawn.)

A NOTE CONCERNING THE MOTOROLA 6800 MPU...

With this issue, I have started to make references to the M6800 MPU system, primarily because I expect it to be available to the Experimenter's Computer System market in the near future. I have been in fairly close contact with the local Motorola sales office in connection with some hardware/software design work I am currently doing, and I have indications that supplies of this product will soon be fairly widely distributed.

If you want to find out about the M6800 in detail, I wholeheartedly recommend purchase of the M6800 Microprocessor Applications Manual (approximately 700 pages 8.5 x 11 @ \$25.00) and the M6800 Microprocessor Programming Manual (approximately 250 pages @ \$10.00). The applications manual includes lots of useful information including interfaces (hardware and software) to floppy discs, cassette tape drives, teletype, Burroughs self-scan displays, adding machine tape printers, etc. etc. I have verbal assurances from the local Motorola sales office that these books will be sold to private individuals on request. If you are interested I suggest that you look up the telephone number of the nearest office and inquire. If you have any problems, let me know and I'll try to make formal arrangements to distribute copies. These documents will set the standard for some time to come, and would easily serve as the basis of a "software engineering" course in applications.

ECS

THE MONTHLY MAGAZINE OF IDEAS
FOR THE MICROCOMPUTER EXPERIMENTER

Publisher's Introduction:

For every process there is an initialization segment - a starting point in time, during which time the program for the process sets up data values and begins its operation. In a sense, this issue represents such an initialization - it is the first issue to contain a subscriber-written article, the Digital Graphic Display Oscilloscope Interface design and writeup prepared by James Hogenson. The graphics device was conceived by Jim as a neat idea to add to his own computer system which he was building for a high school science fair. He first mentioned it to me in a letter late last year. I suggested to him (or was it the other way around?) that it might be appropriate to turn it into an article for ECS. After a fair amount of time spent researching the various options - plus one lengthy phone conversation with me - Jim settled on the design shown in this issue. He constructed the prototype using wire wrap techniques, and interfaced it with his 8008 built using the RGS kit. The interface is very simple, and can be adapted to virtually any computer with a parallel 8-bit output and a clock pulse arriving to the interface during periods of stable data. The device is programmed using a simple two-bit op code field and six-bit data/control field within the 8-bit interface.

I have a PC board version of the design completed as of the date of publication of this issue (so I can get one myself) - with artwork by Andy Hay using Jim's layout. I expect to have the board debugged and ready to offer to customers with the June issue of ECS. The roster for this issue is equal in size to the base of that number system which all computer "nuts" know and love...

1. Digital Graphic Display Oscilloscope Interface, by James Hogenson. Turn to page 2 for the details which turn your scope into a LIFE matrix, a checker-board, a ping-pong game or whatever your imagination, a 64x64 bit-matrix and appropriate software can represent.

2. Concerning the Hand Assembly of Programs, by yours truly, in which the "assembly" of programs by hand is discussed at some length, along with several more comments on SIRIUS matters and an example in the form of CONCATTER - a routine to concatenate byte strings.

This issue is going to press May 12 1975. The limits of space precluded the next instalment of "Notes on Navigation in the Vicinity of α -Aquila." In the next issue, the 8080 machine architecture will again be visited in the form of further "notes." Also in the next issue, a SIRIUS-MP specified bootstrap sequence will be presented, along with the 8008 code for same. In this case, I mean a "real" planned-in-advance bootstrap load method with all the bells and whistles. Up and coming designs for the near future include an electronic music peripheral (not necessarily as good as Peter Helmers' "Metapiana") as well as an article with a small amount of hardware and a lot of software concerning the programming of interesting digital clocks.

Carl T. Helmers, Jr.
Carl T. Helmers, Jr.

Publisher May 11 1975

DIGITAL GRAPHIC DISPLAY OSCILLOSCOPE INTERFACE
designed and written by James Hogenson

INTRODUCTION

If you want your computer to cough up alpha-numeric information, chances are, you won't have too much problem finding a suitable output device. But if you want your computer to draw pictures, you may find yourself facing a dead end. You could use one of those fancy commercially available graphic CRT terminals, but the IBM you'd need to run the thing might not fit on your workbench. If you do have a spare IBM collecting dust on your closet shelf, fine, but if you're like the rest of us, you need something inexpensive, uncomplicated, and within the scope of the average 8008 or similar system. Thus we have the ECS Digital Graphic Display Oscilloscope Interface. For \$50 worth in semiconductors, your computer can have under its own completely programmed control a full raster on the screen of your oscilloscope.

The digital graphic display oscilloscope interface (DGDOI) is programmed and operated through an 8-bit TTL compatible input. The picture is produced by a pattern of dots. These dots are set in patterns according to the computer's instructions, resulting in a computer generated drawing. The entire pattern of dots is stored within the DGDOI's own internal memory. Once the pattern has been generated and loaded into the DGDOI, the computer no longer needs to retain any related data. This also means the pattern may be generated and loaded in small parts, one part at a time. During the scan cycle, the digital information is converted to analog waveforms and displayed on the oscilloscope.

PRINCIPLE OF OPERATION

The raster begins its scan in the upper left-hand corner, scanning left to right and down. The full raster contains 4096 dots; 64 rows of 64 dots each. The horizontal scan is produced by a stepping analog ramp wave. Each step of the ramp produces one dot. There are 64 steps in the wave. The vertical scan is similar. It is a stepping ramp wave consisting of 64 steps. However, there is only one step in the vertical wave for each complete horizontal wave. The result is 64 vertical steps with 64 horizontal steps per vertical step. This produces 64 rows of 64 dots.

The ramp waves originate at a 12-bit binary counter, the center of the entire circuit. The six lower (least significant) bits of the counter are connected to a digital-to-analog converter (DAC), which converts the digital binary input to a voltage level output. The output of the DAC is the horizontal ramp wave. The six upper (most significant) bits are connected to a second DAC. This DAC produces the vertical ramp wave. Incrementing the 12-bit counter at high frequencies results in a raster on the screen of the oscilloscope.

The control of the pattern of dots needed to represent a picture is dependent upon the intensity of each dot. From this point, we will assume a dot can be either on or off. An "on" dot will show up on the screen as a dot of light. An "off" dot will be a dim spot or blank on the screen.

When a particular dot is selected for programming, it is programmed as either on or off. The on-off control can be represented by a single bit. It is this bit which is stored in the internal memory of the DGD01. There is one bit in the memory for each of the possible 4096 dots on the screen. When selecting a dot for programming, you are actually addressing the memory location of that particular dot. You then set the dot for on or off. When displaying the image, the 12-bit counter which produces the raster addresses each dot in the memory as it is displayed on the screen. The on-off bit taken from the memory is converted to a Z-axis signal which controls the intensity of the dot. The Z-axis signal is fed into the Z-axis input on the scope.

Much of the circuitry is taken up in the 12-bit counter, the DAC's, and the memory. Figure 1 shows a block diagram of the DGD01. The remaining circuitry is the control circuitry which decodes the 8-bit input word and allows for completely programmed operation.

PROGRAMMING

Table 1

<u>Op Code</u>			
<u>Binary</u>	<u>Octal</u>	<u>Mnemonic</u>	<u>Explanation</u>
00DDDDDD	0DD	STX	Set X
01DDDDDD	1DD	STY	Set Y
10xxx000	2x0	CNO	Control - No Op
10xxx001	2x1	TSF	Control - Turn off scan
10xxx010	2x2	ZON	Control - Set Z on
10xxx011	2x3	ZOF	Control - Set Z off
10xxx100	2x4	ZNI	Control - Set Z on with increment
10xxx101	2x5	ZFI	Control - Set Z off with increment
10xxx110	2x6	TSN	Control - Turn on scan
10xxx111	2x7	CNO	Control - No Op
11xxxxxx	3xx	CNO	No Op

D = DATA X = NULL

The programming instruction format is shown in Table 1. Bits 7 and 6 of the input word are the high-order instruction code. We will assume that the addressing of dots is done on the basis of X and Y coordinates. The X coordinate is the 6 bits in the lower half or horizontal section of the 12-bit counter. The Y coordinate is the 6 upper bits or vertical half of the counter. In programming from an 8-bit input source, all 12 bits of the counter cannot be set at once. The counter is set one half or 6 bits at a time. It is for this reason we assume an X and Y coordinate for programming. When the instruction code (bits 7 & 6) is set at 00, the data in bits 0 through 5 of the input word is loaded into the lower half of the counter as the X coordinate.

When the instruction code is set at 01, the data in bits 0 through 5 is loaded into the upper half of the counter as the Y coordinate. In effect, the Y coordinate will select a row and the X coordinate will select a dot in that selected row. The coordinates loaded into the counter will address the memory and select the dot location we want to program.

After loading the coordinates of the dot for programming, we set the dot itself. Setting the instruction code at 10 directs the control circuitry to decode the three lower bits of the data word for further instruction. We will call the lower three bits the low order control code.

The first low order control is a No Op instruction. The eighth control and the fourth high order instruction are also No Op's.

The second control will turn off the scan. The seventh control will turn the scan on. When the scan is on, the counter is incremented at a high frequency and the programmed image is displayed on the scope. The scan must be turned off before a dot can be programmed.

The third control, set Z on, will program a dot to appear at the dot location presently loaded into the counter. The fourth control, set Z off, will program a blank to appear at the dot location presently loaded into the counter.

The fifth and sixth control instructions set Z in the same manner as controls three and four. However, after setting Z, these instructions will also increment the counter by one. This will allow the entire 4096 dots to be programmed using only a repeated "set Z" instruction. The counter will naturally follow the regular scan pattern of the raster. This is especially useful in clearing the contents of the DGDOI memory so that a new image can be programmed. It can also be used in making horizontal lines or other patterns in the image.

CIRCUIT OPERATION

Once the data word on the input is stable, only one clock pulse is needed to execute the instruction. The high order instruction is decoded by the 7410 triple three-input NAND gate and two inverters. The clock pulse is enabled by the NAND gate to the appropriate counter section, or to the strobe input of the low order control decoder. The clock pulse is enabled according to the instruction of bits 7 and 6.

The 12-bit counter consists of two 6-bit counting sections. Each section consists of two cascaded TTL 74193 presettable binary counters. Bits 0 through 5 of the data input are common to both sections of the counter. The set X instruction will pulse the load input of the lower or X section of the counter. The pulse on the load input will cause the data on bits 0 through 5 to be loaded into the counter section.

The Y instruction, similar to the X instruction, will pulse the load input of the upper or Y section of the counter.

The two sections are cascaded by connecting the upper data B output of the X counter section, pin 2, IC 8, through inverter 'a' of IC 2 to the count up input, pin 5, IC 9, of the Y counter section.

The low order control code is decoded by a 74155 decoder connected for 3 to 8 line decoding. Bits 0 through 2 are decoded by the 74155. The control code is enabled by the pulse coming from the 7410 high order instruction decoder. The low order control is enabled only when the high order code is set at 10 on bits 7 and 6.

Decoder lines 1 and 6 are connected to an R/S flip flop which provides the scan on/off control. The R/S flip flop enables a high frequency square wave to increment the 12-bit counter.

Control instructions 2 through 5 are 'set Z' instructions, therefore involving a data write operation. Decoder lines 2,3,4, and 5 are connected to a group of AND gates (IC 5a,b,c) functioning as a negative logic OR gate. The output of the gate is the Read/Write control line for the memory. When this line is in the low state, the data present on the data input line of the memory will be written into the memory location presently being addressed by the 12-bit counter.

The data input of the memory is connected directly to bit 0 of the 8-bit input word. A bit will be stored in the memory only when a 'set Z' instruction is executed. The Z-axis circuitry requires a high state pulse for a blank. As shown in the binary format, Table 1, bit zero will be a binary zero for 'set Z on' instructions and binary one for 'set Z off' instructions. The backward appearance of this binary format will be overlooked when programming in octal notation.

The high frequency square wave controlled by the R/S flip flop and decoder lines 4 and 5 are negative logic ORed. The resulting pulse increments the counter according to the control instruction.

The same clock pulse is used to write data into the memory and increment the counter in control instructions 4 and 5. The data is written into the memory on the leading edge of the pulse. The counter is incremented on the trailing edge. Figure 2 shows this waveform.

Output bits 0 through 9 of the 12-bit counter are connected to the address inputs of the memory. The memory uses four MM2102 1024 x 1 bit MOS RAM's (Random Access Memories). Bits 10 and 11 of the counter output are connected to the chip select circuitry which enables one chip at a time for addressing and data input/output operations. The chip select circuitry uses 2 inverters and a TTL 7400 Quad two-input NAND gate.

The data outputs of the RAM's are OR-tied and connected to an AND gate. The data output is synchronized with the high frequency clock for better blanking performance. The output of this gate is connected to the Z-axis blanking circuitry. This circuitry converts the TTL level signal to a scope compatible signal.

Bits 0 through 5 of the 12 bit counter are connected to the X coordinate DAC. Bits 6 through 11 of the counter are connected to the Y coordinate DAC. The DAC's are Motorola MC1406 IC's. They operate on voltages of +5 and -9. A current output is produced by the DAC's. The current output is converted to a voltage output and amplified by the 741 Op Amps. The output from the X coordinate circuitry is connected to the horizontal input of the scope. (The scope should be set for external horizontal sweep.) The output from the Y coordinate circuitry is connected to the vertical input of the scope.

CONSTRUCTION

A printed circuit board is being planned for this project, but for the time being, the method of construction is left for the reader to decide upon for himself.

Remember that the memory IC's are MOS devices and should be handled as such. Static electricity will not do them any good.

Remember to use bypass capacitors. A 100 mfd electrolytic and several .01 mfd disc capacitors are usually recommended. An acceptable "rule of thumb" is one disc capacitor for every two to three TTL chips and one electrolytic per p.c. board.

The parts list is shown on the next page. The schematic diagram is also included in one of the following pages.

PARTS LIST

C1, C2	20pf	disc capacitor	
C3	.01mf	disc capacitor	
C4	.0015mf	disc capacitor	
C5	330pf	disc capacitor	
Bypass	100mf	electrolytic capacitor	
Bypass	.01mf	disc capacitors	
D1-D3		silicon rectifier (1N914 or similar)	
IC 1	7410	TTL Triple 3-Input NAND Gate	
IC 2	7404	TTL Hex Inverter	
IC 3, IC 4	7400	TTL Quad 2-Input NAND Gate	
IC 5	7408	TTL Quad 2-Input AND Gate	
IC 6	74155	TTL Dual 2-to-4-line Decoder	
IC 7-IC 10	74193	TTL Presettable 4-bit Binary Counter	
IC 11-IC 14	2102	MOS 1024-bit Static RAM	
IC 15, IC 16	MC1406	Motorola 6-bit DAC	
IC 17, IC 18	741	Op Amp	
IC 19	NE555	Oscillator	
Q1, Q2	2N5139	Transistor	
R1, R2	3.3k ohm	resistor	
R3, R4	5.6k ohm	resistor	
R5, R9	2.2k ohm	resistor	all resistors
R6	1.8k ohm	resistor	1/4 watt, 10%
R7	18k ohm	resistor	
R8	100 ohm	resistor	
R10	7.5k ohm	miniature potentiometer	
R11, R12	10k ohm	miniature potentiometer	

SET-UP, TESTING, AND OPERATION

Supply voltages needed are +5 VDC at 400 mA, +15 and -15 VDC at 10 mA. The TTL and memory IC's operate on +5 VDC. The DAC's use +5 and -15 VDC. The Op Amps use +15 and -15 VDC. The DAC's and Op Amps will also operate with voltages of 9 or 12 instead of 15. This will allow you to use your existing computer's power supply for the DGDOI as well.

When you are satisfied that your DGDOI is ready for operation, do not immediately connect it to an I/O channel on your computer. For initial testing, use the test circuit shown in Figure 5 (Included in following pages). The only requirement is that the test rig be able to provide an 8-bit binary input word and a clock pulse. If a computer is used for initial testing, it is difficult to pinpoint a problem as being in the circuit. A problem can often be found in the software used with the DGDOI.

The clock pulse should be active in the high state as shown in Figure Three. If your computer operates with an active-low pulse, an inverter is needed for inverting the clock pulse.

When you are ready to test, turn on the power and load a 'turn on scan' instruction. The turn on scan instruction should produce a raster. If a distorted concentration of dots appears, adjust the DAC voltage reference pots.

The high frequency square wave is provided by a 555 timer IC connected as an astable multivibrator. Adjusting the frequency may be necessary to obtain a stable appearing raster. (Note: you don't need a fancy scope for this project. A cheap 250kHz scope was used with the proto-type.)

The next step is to check the blanking. You should get a mixture of on and off dots simply by turning on the power. The frequency of the scan and voltage supplied to the Z-axis circuitry both affect blanking performance. The Z-axis amplifier may be disconnected from the -15 volt supply and connected to up to -25 volts. The frequency may be adjusted with the 7.5k pot. It should be noted however, that raising either of these too high will have adverse effects. Keep in mind that the Z-axis is connected through a capacitor (in most cases) within the scope. Charging the capacitor with too much voltage at a given frequency will cause the blank to carry over into the next dot. Thus one blank pulse blanks out two dots. Avoid this situation.

Performance varies, depending upon each particular scope. The best way to find the best contrast and blanking performance is by experimenting. If you are unable to obtain any blanking, connect the Z-axis output to the vertical input of your scope. If no pulses are present, your trouble is back in the DGD01 circuit.

After you have obtained a satisfactory raster, execute each instruction manually to verify its operation. Clear the memory by setting the input at 205 (octal) and connecting a 10kHz square wave to the clock pulse input. (Remember: Scan must be turned off before programming any dots) Execute a set X, set Y, a number of set Z on with increment's, and turn on scan. Your programmed dots should now appear.

If all operations seem good, connect your computer. You may write programs to your hearts content, but just in case, there is a test pattern program included in this article. If your DGD01 doesn't operate correctly after connecting your computer, check all software first. This is usually the cause of most problems.

The data output of the DGD01 memory may be connected as a computer input, but this is optional. To read the status of a dot, you would load the coordinate of the selected dot, then read the single bit data output.

TEST PATTERN PROGRAM

The program listed on the following page(s) will program the DGD01 for a test pattern. The pattern will be a checkerboard pattern of 16 alternating light and dark squares.

The program counts off 4 sections of 16 dots per section. Each section is alternated to get a pattern of light-dark-light-dark or dark-light-dark-light. Rows are also counted off in groups of 16. Each row in the same group is set with the same pattern, but each group is set with an alternate pattern.

The set Z with increment instructions are used. The least significant bit of the E register is used in DECL00P to alternate between set Z on and set Z off.

The various loops in the program are briefly described in the following paragraphs.

DOTLOOP counts off each section of 16 dots and programs the section of dots according to DECL00P.

XSELOOP counts off 4 sections per row and jumps back to DECL00P to alternate the set Z instructions between sections.

ROWLOOP counts groups of 16 rows and increments the E register an extra time to reverse the order in DECLLOOP between each group of rows.

YSECLLOOP counts off 4 groups of 16 rows to halt computer when checkerboard has been loaded into DGDOI.

To invert the pattern on the screen, load E with 001 instead of 000 in location 00 220. This will have the effect of inverting the parity register. The result would produce a pattern of the opposite light and dark arrangement.

START	00/200 = 006	LAI	00/255 = 302	LAC
	00/201 = 201	(TSF)	00/256 = 024	SUI
	00/202 = 121	OUT 10	00/257 = 003	
	00/203 = 006	LAI	00/260 = 150	JTZ
	00/204 = 000	(STX)	00/261 = 267	
	00/205 = 121	OUT 10	00/262 = 000	
	00/206 = 006	LAI	00/263 = 020	INC
	00/207 = 100	(STY)	00/264 = 104	JMP
	00/210 = 121	OUT 10	00/265 = 221	
CLEAR	00/211 = 016	LBI	00/266 = 000	
REGISTERS	00/212 = 000		00/267 = 026	LCI
	00/213 = 321	LCB	00/270 = 000	
	00/214 = 331	LDB	00/271 = 303	LAD
	00/215 = 351	LHB	00/272 = 044	NDI
	00/216 = 361	LLB	00/273 = 037	
	00/217 = 046	LEI	00/274 = 024	SUI
PARITY REG	00/220 = 000		00/275 = 017	
DECLLOOP	00/221 = 040	INE	00/276 = 150	JTZ
	00/222 = 304	LAE	00/277 = 305	
	00/223 = 044	NDI	00/300 = 000	
	00/224 = 001		00/301 = 030	IND
	00/225 = 150	JTZ	00/302 = 104	JMP
	00/226 = 246		00/303 = 221	
	00/227 = 000		00/304 = 000	
	00/230 = 066	LLI	00/305 = 303	LAD
	00/231 = 332		00/306 = 044	NDI
DOTLOOP	00/232 = 301	LAB	00/307 = 340	
	00/233 = 024	SUI	00/310 = 330	LDA
	00/234 = 020		00/311 = 024	SUI
	00/235 = 150	JTZ	00/312 = 140	
	00/236 = 253		00/313 = 150	JTZ
	00/237 = 000		00/314 = 326	
	00/240 = 010	INB	00/315 = 000	
	00/241 = 307	LAM	00/316 = 303	LAD
	00/242 = 121	OUT 10	00/317 = 004	ADI
	00/243 = 104	JMP	00/320 = 040	
	00/244 = 232		00/321 = 330	LDA
	00/245 = 000		00/322 = 040	INE
DECLLOOPJMP	00/246 = 066	LLI	00/323 = 104	JMP
	00/247 = 333		00/324 = 221	
	00/250 = 104	JMP	00/325 = 000	
	00/251 = 232		00/326 = 006	LAI
	00/252 = 000		00/327 = 206	(TSN)
XSECLLOOP	00/253 = 016	LBI	00/330 = 121	OUT 10
	00/254 = 000		00/331 = 377	HLT
			00/332 = 204	(ZNI)
			00/333 = 205	(ZFI)

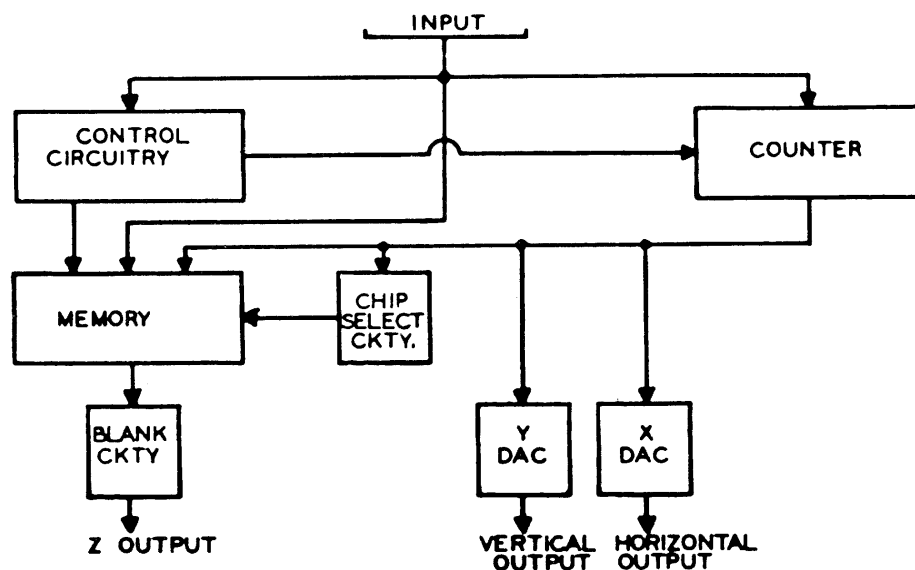


FIGURE 1.
DGDOI BLOCK DIAGRAM

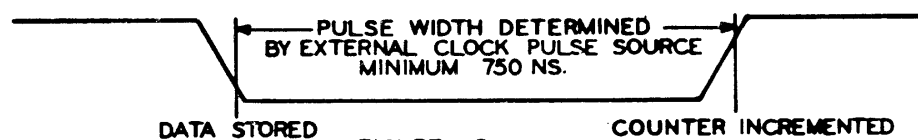


FIGURE 2.

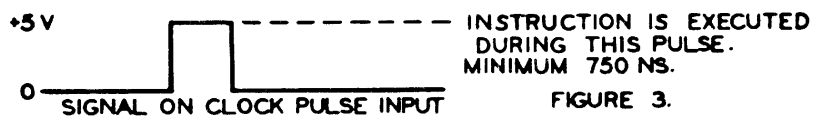


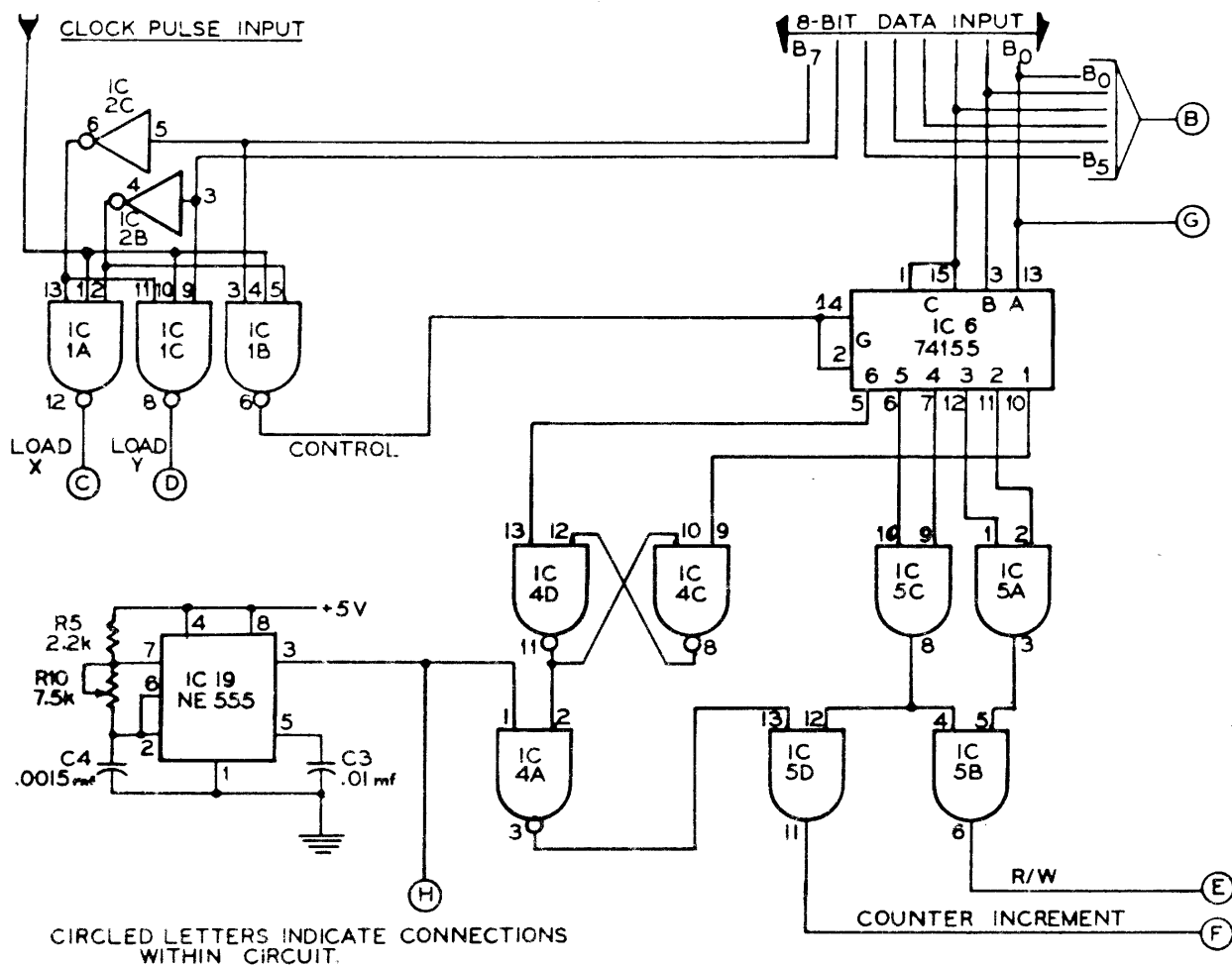
FIGURE 3.

IC POWER AND N/C PIN CONNECTION CHART

IC	+5	GND	+9	-9	N/C
1,2,3,4,5	14	7			
6	16	8			9,4
7,9	4,16	8,14			
8,10	16	8,14			6,7,9,10,12,13
11,12,13,14	10	9			
15,16	11	2		3	1
17,18			7	4	1,5,8
19	4,8	1			

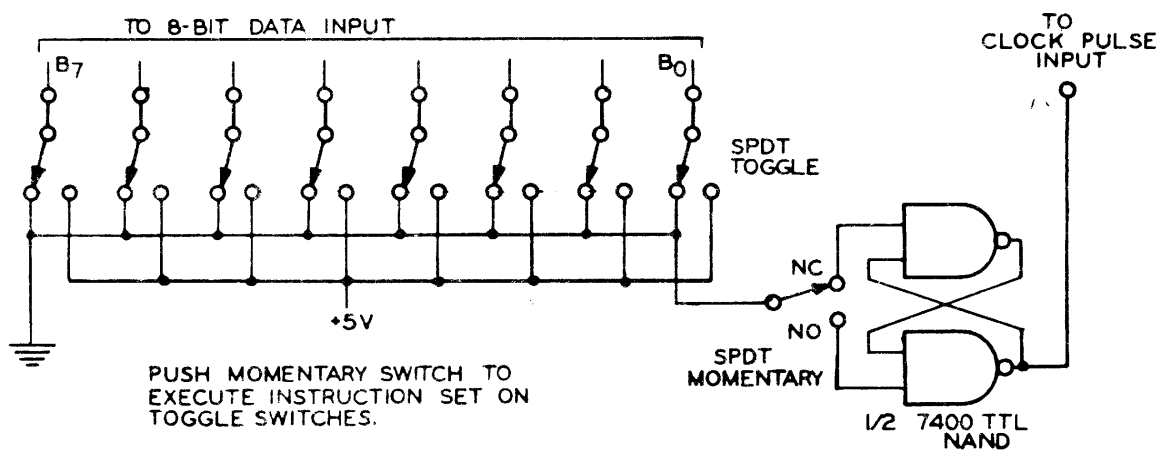
2102 MEMORY ADDRESS PIN CONNECTIONS

A-0 -- pin 8 : A-1 -- pin 4 : A-2 -- pin 5 : A-3 -- pin 6
 A-4 -- pin 7 : A-5 -- pin 2 : A-6 -- pin 1 : A-7 -- pin 16
 A-8 -- pin 15: A-9 -- pin 14

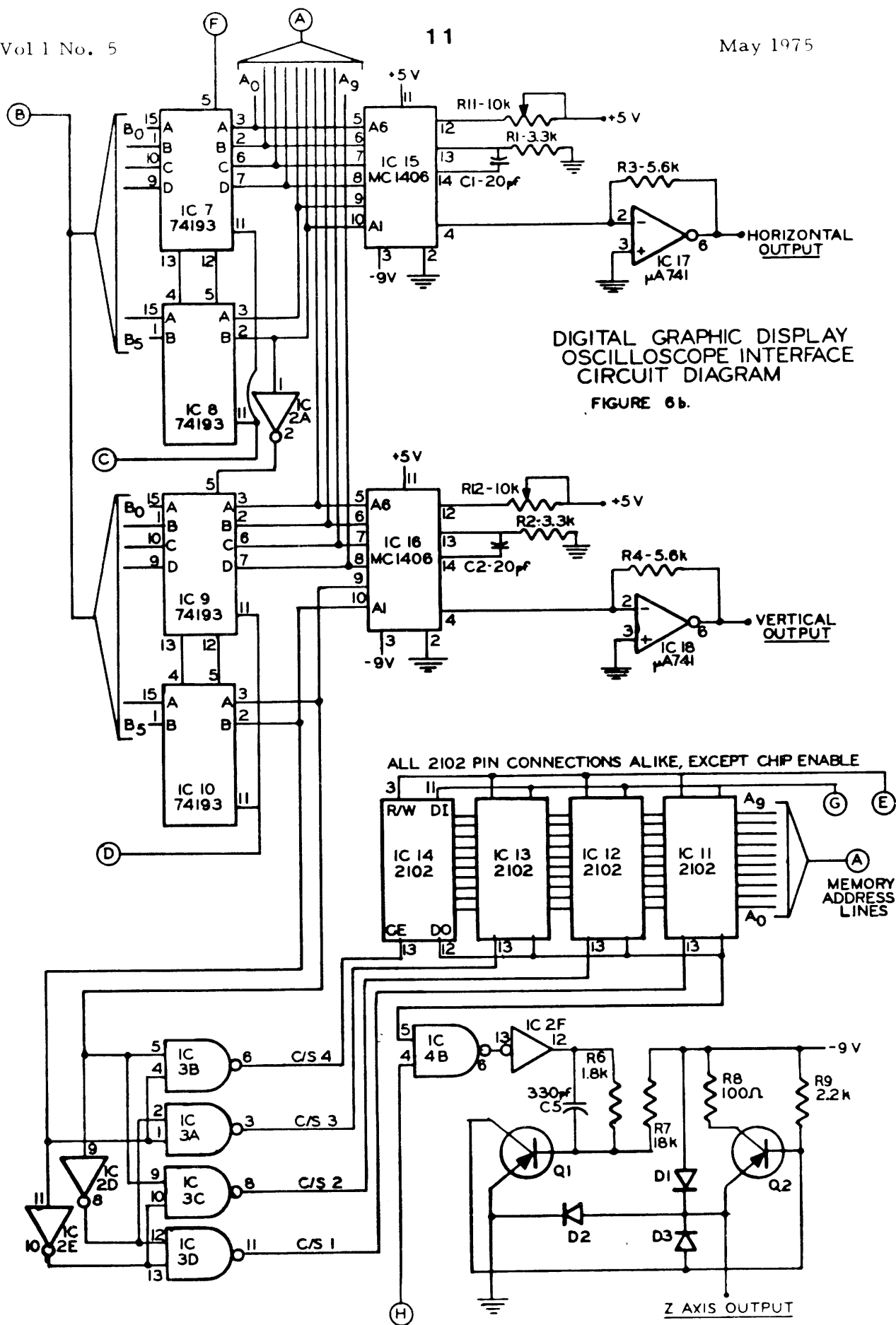


DIGITAL GRAPHIC DISPLAY
OSCILLOSCOPE INTERFACE
CIRCUIT DIAGRAM

FIGURE 8a.



MANUAL TEST CIRCUIT FIGURE 5.



CLEAR DGDOI PROGRAM

This program is used to clear the memory of the DGDOI. It simply sends out a 'set Z off with increment' instruction 4096 times. It uses the B and C registers to keep track of the 4096. The register contents are decremented once for each I/O instruction.

The program turns the scan off before clearing, but does not turn scan back on. The DGDOI will then remain ready for programming.

START	00/344 = 006	LAI	00/357 = 150	JTZ
	00/345 = 201	(TSF)	00/360 = 365	
	00/346 = 121	OUT 10	00/361 = 000	
	00/347 = 006	LAI	00/362 = 104	JMP
	00/350 = 205		00/363 = 355	
	00/351 = 016	LBI	00/364 = 000	
	00/352 = 377		00/365 = 021	DCC
	00/353 = 026	LCI	00/366 = 110	JFZ
	00/354 = 021		00/367 = 355	
	00/355 = 121	OUT 10	00/370 = 000	
	00/356 = 011	DCB	00/371 = 377	HLT

These two programs are just to get you started. Although uncertain of the medium, we expect to have further programs available in the future. Carl Helmers has plans for a 'Life' game and possibly a 'Space War' game using the DGDOI. The author of this article is planning a Tic-Tac-Toe game and a program which would use an octal keyboard for rapid construction of images. (It will be the closest we can reasonably come to an electronic pen.)

These programs, of course, will be in addition to your own. There are many applications of a DGDOI. Outside of games, it could be used to graph solution sets of mathematical problems. It could be used to graph results of data acquisition programs. It could plot results in a digitally controlled analog computer system. It could . . . well, who knows how many things it could be used for? The exciting point is that such applications are finally within the economical range of the 8008 system.

PRINTED CIRCUIT BOARD FOR THE "DGDOI" DESIGN:

As this issue of ECS goes to press, the first layout of a two-layer PC board with plated-thru holes has been completed. A first printing of the board will be executed prior to the next issue of ECS, at which time I expect to have details of pricing on the board.

SOME LAST MINUTE IMPROVEMENTS:

In cassette conversation with Jim Hogenson, the following items were pointed out regarding updates of the article as it stands: 1) by connecting the "0" output of IC 6 (6-9) to IC 9 "decrement input" (9-4) the "2x0" (octal) opcode becomes decrement Y. 2) by connecting the "7" output of IC 6 (6-4) to IC 7 "decrement" (7-4) the "2x7" (octal) op code becomes decrement X. 3) The DAC chips may exhibit non-linearities due to manufacturing variations - sometimes observable in particular cases.

CONCERNING THE HAND ASSEMBLY OF PROGRAMS ...

by Carl T. Helmers, Jr.

The purpose of computing is to solve problems. Problems are solved by analysis followed by generation of a method - an algorithm - for accomplishing the desired ends. The computing approach to problem solution consists of automating the steps of such methods by preparing a "program" for the computer to execute. This article concerns the process of preparing programs for execution on the assumption that you have previously generated a detailed symbolic specification of your problem's algorithm in the SIRIUS-MP language (or any other method of program specification for that matter.) The remaining task of program preparation is the translation of the symbolic form into a detailed set of machine codes (numbers).

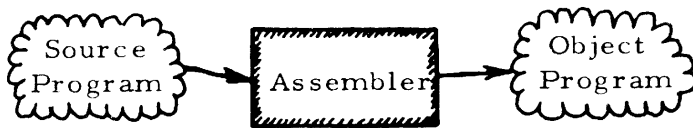
In April 1975 ECS, an introduction to the SIRIUS-MP language was presented as a means of expressing programs for inexpensive "home brew" computer systems. The present article continues this SIRIUS information by discussing the process of hand assembly of machine code from the symbolic representation. Hand assembly is a process which the serious student of computing should perform as an exercise at some point in time - whether or not the computer under study has an assembler available. The tutorial value of "walking through" the assembly process is well worth the effort - whether or not the hardware limits of your system make it mandatory.

The "hand assembly" process is in some respects a retrograde motion in computer science - a step "against the normal direction" of progress towards more and more automated programming aids and methods of expression. It is a process which is the translation of existing assembler algorithms (no particular assembler among a myriad of assemblers is singled out as a model here) back into the realm of a manually executed process - just as the first programmable machines had to be programmed before the invention of software development tools. As an adaptation of the "typical" assembler algorithm to manual operations, the manual assembly process to be described is useful in several areas...

- it illuminates the process of assembly as performed automatically, so that the reader will be less tempted to blame all manner of programming problems on the poor simple-minded assembler programs.
- it provides the microcomputer enthusiast with a method of software development (albeit cumbersome) to be used until his or her personal computer is integrated to the point needed for a real assembler.
- it highlights the problems of code generation from symbolic notation.
- it can serve as a model for the implementation of an assembler system by the reader for his own variation on the microcomputer concept.

AN ASSEMBLER SYSTEM

The concept of an assembler system is illustrated at its highest level by the functional diagram... a "black box" of processing which accepts some input and produces some output:



The input at the left of the diagram is the "source program" - a generalized and symbolic representation of your program. The output at the right (the principal output of

the assembler) is the "object program" equivalent of the source program - a set of binary (octal or hex) numbers which potentially can be loaded into appropriate memory locations and executed. (I am leaving out the concepts of linkage editors, relocatable loaders and other post-assembly tricks for the time being.)

What is this assembler "black box?" In an automated conventional assembler system the black box is computer program used to translate a text file (eg: ASCII characters as input from a teletype or other keyboard) of the source program into its equivalent binary object file representation. The term "file" here means a set of many (eg: "n") computer words containing some form of information - often used to signify such data sets as stored on magnetic tape or disc. The usual assembler program is implemented and runs on computer "X", producing an object program for computer "X" (self assembly) or for computer "Y" (cross assembly.) In the corresponding hand assembly conception the assembler "black box" is defined as you - the reader - performing a variation of the steps required to translate the symbolic representation into its machine code form.

THE SOURCE PROGRAM

The source program for the assembly is usually written in the appropriate "Basic Assembly Language" for the computer in question - each computer manufacturer comes up with its own version of the type of program involved, usually running on one of the manufacturer's own machines. For the microcomputer case, this is not usually possible, since the number of variables in individual CPU implementations using the same chip is immense. For the purposes of this publication and the generality of notation, the article assumes a source program written in the SIRIUS-MP formulation which is to a large extent independent of any particular chip design. If you were to substitute "Language X" for SIRIUS-MP in the ensuing pages, you can do so and apply the same process - although your translation function will technically be that of a compiler or interpreter if any language other than an assembly language is used. This article's methodology could in particular be applied to the translation of some of the immense number of published computer "games" in BASIC for instance, if you want to get such programs up and running - however tackling a high order language translation will tend to get you bogged down in detail and in routines you have to write to get anything done, so it is only recommended in the simplest of cases when performed by hand.

THE OBJECT PROGRAM

The output of the assembly process is an "object program" - a potentially executable set of codes for the computer. The form in which an object program is specified should be chosen according to the needs of the assembly process and the intended use of the results. In a "real" assembler (ie: a computer program running on some computer) two major classes of output come to mind:

1. Absolute Machine Code. Here the object module output consists of information needed to define the specific content of each memory location in the program, tied directly to a specific range of memory address space in the computer. In this variation of output, all the work is done at the time of assembly, and loading the program then becomes a task of copying this "memory image" (archaic term: core image) into the computer.
2. Relocatable Machine Code. Here the object module is built by the assembler program relative to an arbitrarily chosen starting address (often "0"), with the final resolution of addresses for symbolic references, jumps, etc. left to an appropriate "relocating" loader. The object module in this form is more complicated for in addition to the binary image of the program, information on the address references inside the program must be retained so that the loader can alter them during the load process.

In addition to the specific form of the modules, there is the question of linking multiple program segments - which can open up a whole "can of worms" best ignored at this stage. For the purpose of hand compilation, the "KISS" rule applies - "keep it simple, stupid." The assumption will be that linkages between modules are made by commonly addressed absolute address regions (for example, the first 256 bytes or base page of a Motorola 6800, the first 256 bytes of an 8008 designed according to my plans published earlier, or an arbitrary region if no particular location is suggested by the characteristics of hardware or software.)

In order to keep the process simple, the Hand Assembly method as described here is limited to the production of absolute machine codes (type 1 object modules as listed above.) The actual form will be a list of hardware addresses in memory address space and the corresponding machine code for that address. I have written the article under the assumption that the M.P. Publishing Co. Kluge-I Assembler coding sheets are used for the final output, but this is by no means to be interpreted as an absolute "requirement" of the method. They are available at 5¢ each plus postage, and were created primarily to satisfy my own purposes after I got tired of writing the same low order address sequences over and over and over again. An alternate source of paper for the process is used computer paper recycled from a handy local computer center, or if you are in position to make arrangements for time - you could whip off a quick FORTRAN or PL/1 (or ?) program to write the address sequences onto blank paper in a manner similar to the Kluge-I sheets but on a line printer instead.

The process of assembling and generating the code for a program has two major (conceptual) steps which must be performed, assuming that a suitable symbolic notation for the algorithm exists.

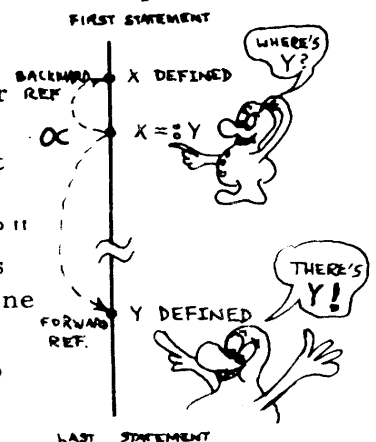
Step 1: Translate the symbolic notations into equivalent sequences of the machine's operations. Pay attention to any address calculations which may be required, but leave "open" the question of addresses of operands for which no address is yet assigned. The purpose of this step is primarily to allocate the memory address space requirements of the program by determining the number of bytes of code required for each elementary statement of the program which is translated.

Step 2: With all the required program and data locations allocated (typically in a sequence of consecutive memory locations starting at a chosen "origin" or first address) "fix up" all the unresolved references hanging around in the code prototypes created in step 1.

This set of steps is a universal one, and is performed by every code generation process - whether it is an assembler, a compiler's code generation phase, or even an interpretively executed programming language such as BASIC. The variations (and there are many) in particular approaches to compiler and assembler code generation strategies concern ways of implementing these conceptual processes of allocation and reference resolution (the "fix ups"). In a classical two-pass assembler and/or compiler, there is an explicit separation into these two steps - pass one is the allocation phase (also syntax checking), followed by pass two which fixes things up. If one restricts the types of references possible at any given point in the program source, it is possible to achieve a "one pass" compiler - the restriction being the rule that no "forward" references be made to portions of a program yet to be referenced, or that such forward references be made through a special mechanism in the generated code such as a run time symbol table lookup/calculation. In the hand assembly version of the process described here, a classic two-pass approach is taken, but the first pass is further broken down into two operations which might be conceptually considered "passes" through the data. The text continues following a short aside...

WHY ARE TWO PASSES NECESSARY IN THE UNRESTRICTED CASE AS A MINIMUM NUMBER OF SCANS THROUGH THE DATA?

The necessity of the second "fixup" pass becomes obvious when you consider the problem of forward references. (References to previously allocated symbols are no problem - I already have their addresses figured out.) The assembly process can only sequentially process the statements of the program, starting with the first. A "forward reference" to some symbol in the program is a symbolic reference made prior to the definition of the symbol in question - relative to the order of scanning the source. Pictorially, a forward reference is illustrated by the assembler (an "imp") finding the statement "X = : Y" closer to the beginning of the scan than the definition of the symbol Y. At \propto the little imp says "where's Y?" and files it as an open question. A bit later in the first pass he can say "aha - I know where Y is" but - he has already gone past the point where Y was referenced. Then on the second time around, the little imp can use this information to fix up the incomplete information in the statement with the forward reference. Either the minimum two passes through the data, or a logically equivalent "trick" is required to resolve the forward reference.



The hand assembly process is outlined in the paragraphs following immediately below. The process is broken down into three sequential steps which I have found to be components of a useful procedure: generate skeleton code, allocate addresses, then fill in the final code of the program replacing mnemonic notations and symbolic address references. Of these steps the first two correspond roughly to the allocation pass of a two pass assembler, and the last corresponds roughly to the reference resolution (fix up) pass. Following this descriptive summary of the process, a detailed example is presented for the case of a subroutine used to "concatenate" bytes strings of the form described on page 9 of April 1975 ECS.

SKELETON CODE GENERATION:

The first pass of the hand assembly process begins with a "skeleton code generation" operation. The purpose of this operation is to figure out the mnemonic operation codes required for the corresponding operations of the source program. If you program exclusively in the mnemonic assembly language appropriate to a given machine you have already performed this operation by writing your program on paper. If you use a "higher level" specification such as SIRIUS-MP (or FORTRAN, PL/I, BASIC, and any other language you might care to use) this step is required in order to turn the basic operations of the source program into sequences of operation appropriate for your computer's instruction set. For the SIRIUS-MP language, this corresponds to a table lookup (in your head) of an appropriate method of carrying out the functions of each statement, and in many cases will result in a fairly one-to-one correspondence of operations in the source program and in the machine code. If you automate this process, it becomes roughly equivalent to a "macro expansion" process tacked on the front end of many assemblers. I have found scrap computer listings to be most effective in this stage since it involves no address allocation, merely listing the symbolic equivalents of the program bytes on paper.

ADDRESS ALLOCATION:

The hand assembly process as conceived here is oriented to the generation of the absolute, executable machine code for specific locations in the computer's memory address space. This bypasses the question of generating relocatable code and keeps the process simple. Error possibilities increase with complexity, especially when a program is assembled by biological computing machinery with all its foibles. This address allocation stage consists of taking the skeleton code sequences for the program and assigning a memory address for each byte in turn. One way to do this is to record the byte addresses on the paper used to write the original skeleton sequences. Another method is to use the M. P. Publishing Co. Kluge-I Assembler coding sheets with pre-printed low order addresses in octal to provide the allocation function - if you write an operation code at some place on the sheet, it's address is "used up" and no longer available for allocation. The skeleton code generation and allocation process can be done simultaneously on the Kluge-I sheets provided you are fairly sure of the code being generated (or don't mind erasing a bit if you make a mistake.) The problem of the combined skeleton/allocation approach is that whenever you write down the use of a specific address, it commits the location to a specific utilization, which may

be "premature." I like to get a program done completely in the skeleton form prior to allocation of any addresses, so a review of its operation can be done. Then after the review, I proceed to do the allocation by copying to the Kluge-I sheets. (Even so, I make many mistakes and change things when I see a better way - one of the things which guarantees an incentive on writing an assembler for SIRIUS and at a later stage some form of compiler for a decent programming language.)

An Aside:

It may be possible for you to gain access to a minicomputer facility and/or large computer facility. (Particularly for the readers of ECS who are still in school and can wangle computer time.) One way to implement an assembler for a language such as SIRIUS-MP is to use an existing assembler with a macro facility - eg: the IBM 360 Assembler, or a DEC PDP-10 assembler or a host of others - and write a special set of macros to implement the primitive operations as expansions based on the skeletons of octal(hex) codes required for your target computer. Then all the symbol table lookup and management of the original assembler can be used as is. The troubles with this approach are several: most macro expansion operations of assemblers tend to be inefficient; it is a lot of work to write a complete set of generalized macros and debug them as well; and so on.

FILLING IN THE CODE:

Once the addresses have been allocated to the skeleton, the final step is to fill in the octal (or hex if you prefer) codes of each byte in the program by looking up the mnemonics of the operation codes as noted on the Kluge-I sheets prepared during the allocation stage. This step in the hand assembly corresponds to the "second pass" of the classic two-pass code generation process, but with the added provision that the mnemonic op codes which would be translated in the first pass of an ordinary assembler program are left until this last pass for translation. When the process reaches this stage, all address references are known (as allocated in the allocation step) so that all references can be made in the code resulting. Each byte of the allocated code has one of the following possibilities:

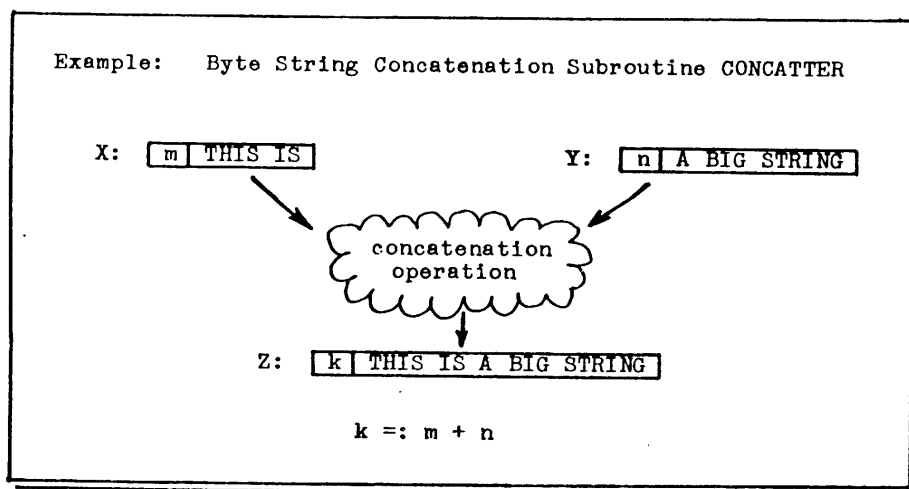
- it has a portion of a literal value which must be translated into its machine code equivalent.
- it has a reference to an address-related value, which for an 8-bit micro means either half of a 16(or 14 for 8008) bit address.
- it has a mnemonic operation code which must be looked up in a table of equivalent octal or hex operation codes.
- it represents a byte of data which is not to receive any initialization, which is simply reserved for use as a run time data storage area.

Whatever the intent, the result for each byte is 3 digits octal (or two digits hex) representing the machine coding for that piece of the program. In the "don't care" cases of reserved data areas (the last option listed above) no explicit action is required to generate the loaded codes of the program.

HAND ASSEMBLY BY EXAMPLE:

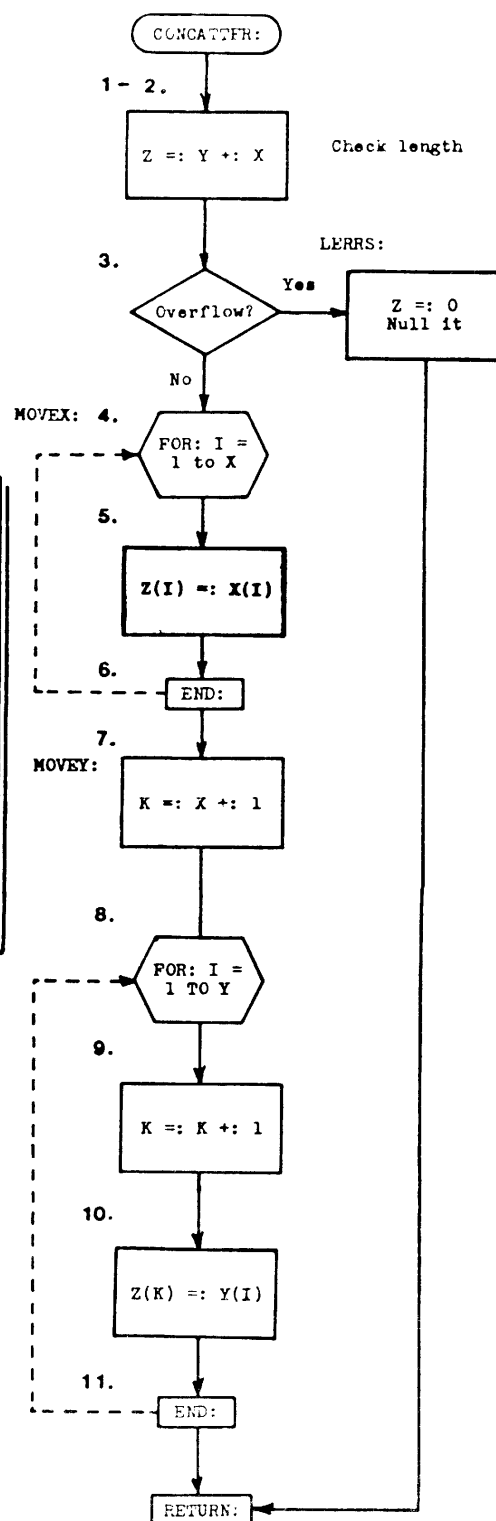
THE BYTE STRING CONCATENATION
SUBROUTINE "CONCATTER."

An example always helps to illustrate a new process or method. To illustrate a hand assembly operation, I have selected a simple little subroutine to perform a string operation called "concatenation". In words, the operation of concatenation is the building of a new string (for example "Z") composed of a left half input (for example "X") and a right half input (for example, "Y"). In symbols, the following diagram illustrates the operation....



If you are familiar with arithmetic and algebra, you of course know there exists a set of operations which are in some sense "fundamental", such as addition, subtraction, etc. Similarly, in boolean algebra, there is a set of fundamental operations - AND, OR, NOT. The same holds when byte string operations are considered as well: the manipulation of "text" is best done using a few fundamental operations, including concatenation, "substring" extraction (the opposite of concatenation), comparisons, etc. The concatenation operation is one of the most useful.

The concatenation operation is shown in its most abstract form by the flow chart running down the right margin of this page. This flow chart describes the steps of concatenation - test the result length for an error, move the left half to the result, then move the right half to the result. The numbers on the diagram correspond to the statement numbers of the equivalent SIRIUS-MP program listed on the next page of this article.



The flow chart illustrated on the previous page is an afterthought - the original written form of the SIRIUS-MP program shown in the box below was created without using a flow chart as a tool. This SIRIUS form of the CONCATTER is assumed as an input to the assembly process for the purpose of the example.

```

1  CONCATTER:      =:      Y      * FORM SUM OF LENGTHS      ;
2  Z              +:      X      * AND TEST FOR OVERFLOW    ;
3  LERRS          IF      CARRY  * OF 8-BIT MAX VALUE      ;
4  MOVEX:         FOR:    1,X     * TRANSFER LOOP CONTROLLED ;
5  Z(I)           =:      X(I)    * BY X LENGTH BYTE      ;
6  END:           * END OF LAST PREV. FOR      ;
7  MOVEY:         =:      X      * Z INDEX FOR Y TRANSFER  ;
8  I             FOR:    1,Y     * Y TRANSFER LOOP CONTRLD ;
9  INCY:          K      * BY Y LENGTH BYTE      ;
10 Z(K)           =:      Y(I)    * TRANSFERS EACH Y      ;
11 END:           * UNTIL DONE      ;
12 RETURN        * WITH Z CONTAINING RESULT ;
13 LERRS:         =:      0      * NULL STRING WITH ZFIRST ;
14 Z             RETURN  * BYTE LENGTH=0      ;

```

New SIRIUS-MP operations in CONCATTER:

+: --- Addition, with 8-bit length indicator, replaces the target operand (eg: Z of statement 2) with the sum of the old target's value and the source operand value.

FOR: --- Incremental "FOR" loop header. This sets up the start of a FOR loop with an assumed integer 8-bit index (":" length code), a starting value given by the first source operand subfield (see note #1 below), and an ending value given by the second source operand subfield. The target operand is optional - if omitted, the generated code will keep its internal count which is then not available to program segments within the loop. A third source operand subfield will be kept available (optional) separated by a comma and used for the increment value if other than one.

END: --- Incremental "FOR" loop trailer. All the statements from the FOR to the END are considered part of the loop. An implicit (ie: "structured") branch back to the last previous FOR occurs if the iteration count is not exceeded. As with the FOR statement, the END has a type modifier to indicate the loop index precision.

Note 1: In order to provide for complex operations such as the FOR loop operation, multiple "source" parameters are sometimes required. The idea of an operand subfield accomplishes the necessary inputs to the FOR loop operation. This concept will recur when the various byte manipulation operations are introduced in later discussions of byte strings.

Note 2: The FOR/END construct is a "natural" for code generation using the CPU stack temporary data concept as it exists on machines such as the PDP-11, M6800 or 8080. When the "FOR" is encountered, a loop return address is pushed onto the stack, followed by the initial count value and the final count value. Then when the "END" is encountered during execution the stack is referenced (offset from stack pointer) to increment the loop count and compare it to the final count. If the final count is not exceeded, execution jumps indirectly through the loop return address (also referenced off the stack pointer) back to the first executable statement of the body of the loop. If the branch back is not taken, the "END" cleans up the stack by adjusting the stack pointer to its original value prior to the FOR statement execution. The stack automatically can handle "nested" FOR loops to as many levels as there is temporary RAM memory to store the stacked data. More on this subject in a later issue...

As in the examples of SIRIUS programs published in April ECS, I have not included a generalized treatment of argument linkages in this example. The example of a subroutine uses specific RAM string areas - X, Y and Z - as its arguments, so that any program utilizing this version would have to first copy X and Y's values from some other place then call CONCATTER - and copy the Z result after getting back. With this formulation, X, Y and Z might be considered the software equivalent of the accumulators (ie: CPU registers) of some hypothetical 3-register "string machine." For large scale text processing applications, someone will sooner or later microcode a processor with the string operations.

Given the starting point of the previous page, the first hand assembly step is begun with the expansion of the SIRIUS code as a skeleton of the final code. I have illustrated a small portion of the skeleton listing of CONCATTER at the left in the following illustration:

SKELETON

```

#1 { LAI
    S(Y)
    SYM
    LBM
    #2 { LAI
        S(X)
        SYM
        LAM
        ADB
        #3 { JTC
            L
            H
            LBA
            #2 { LAI
                S(Z)
                SYM
                LMB

```

← temp for Z is B

← Setch y

LERRS Out of seg. jump
to avoid carry flag
save mechanism.

KLUGE-I ALLOCATION

#1	CONCATTER:	200	LAI
		201	S(Y)
		202	SYM
		203	LBM
#2		204	LAI
		205	S(X)
		206	SYM
		207	LAM
		210	ADB
#3		211	JTC LERRS
	?	212	L
	?	213	H
#2		214	LBA
		215	LAI
		216	S(Z)
		217	SYM
		220	LMB

The code illustrated here is for an 8008 processor (my own "ECS" system) and uses the software conventions (eg: SYM table lookup) described in earlier issues. The Kluge-I allocation of addresses for the Skeleton code is illustrated at the right. In the allocation step, numbers are used to reference SIRIUS statements of the source program, and the question marks (" ? ") serve to denote address references prior to definition. The LERRS example here is a "forward reference" to later code which resolves (after allocation of the whole routine) to be location 007/334.

The code generated for the remainder of CONCATTER (8008 mnemonics from the original Intel documentation) is printed on the next page. This listing contains the results of the third hand assembly pass (filling in code and allocated address references) along with mnemonics and statement number references back to the original SIRIUS-MP code.

The subroutine named "OFFSET" was coded to perform the index calculation of the type implied by the SIRIUS notation NAME(INDEX). It adds (16 bit calculation) the current 8-bit loop count maintained in B (CPU register) to the address found in the H/L pointer pair. For 8080 machines, this subroutine would not be necessary since there is the 16-bit address calculation possibility for the H/L pair.

The FOR/END group code is generated in a form using an index variable I which happens to be redundant in this example. The actual loop indices in this simplest case are maintained in the CPU B register (moving index) and CPU C register (end index).

CONCATTER: 8008 Code Equivalent

#1	007\200 = 006 LAI	#8	007\270 = 006 LAI
	007\201 = 040 S(Y)		007\271 = 040 S(Y)
	007\202 = 075 SYM		007\272 = 075 SYM
	007\203 = 317 LBM		007\273 = 327 LCM
#2	007\204 = 006 LAI	#8B	007\274 = 006 LAI
	007\205 = 036 S(X)		007\275 = 044 S(I)
	007\206 = 075 SYM		007\276 = 075 SYM
	007\207 = 307 LAM		007\277 = 371 LMB
	007\210 = 201 ADB	#9	007\300 = 040 INE
#3	007\211 = 140 JTC #13	#10	007\301 = 006 LAI
	007\212 = 334 L		007\302 = 040 S(Y)
	007\213 = 007 H		007\303 = 075 SYM
#2	007\214 = 310 LBA		007\304 = 106 CAL OFFSET
	007\215 = 006 LAI		007\305 = 367 L
	007\216 = 042 S(Z)		007\306 = 007 H
	007\217 = 075 SYM		007\307 = 337 LDM
#4	007\220 = 371 LMB		007\310 = 351 LHB
	007\221 = 016 LBI		007\311 = 314 LBE
	007\222 = 001 I		007\312 = 345 LEH
	007\223 = 006 LAI		007\313 = 006 LAI
	007\224 = 036 S(X)		007\314 = 042 S(Z)
	007\225 = 075 SYM		007\315 = 075 SYM
	007\226 = 327 LCM		007\316 = 106 CAL OFFSET
#4B	007\227 = 006 LAI		007\317 = 367 L
	007\230 = 044 S(I)		007\320 = 007 H
	007\231 = 075 SYM		007\321 = 373 LMD
	007\232 = 371 LMB		007\322 = 351 LHB
#5	007\233 = 006 LAI		007\323 = 314 LBE
	007\234 = 036 S(X)		007\324 = 345 LEH
	007\235 = 075 SYM	#11	007\325 = 301 LAB
	007\236 = 106 CAL OFFSET		007\326 = 272 CPC
	007\237 = 367 L	#12	007\327 = 053 RTZ
	007\240 = 007 H	#11	007\330 = 010 INB
	007\241 = 337 LDM		007\331 = 104 JMP #8B
	007\242 = 006 LAI		007\332 = 274 L
	007\243 = 042 S(Z)		007\333 = 007 H
	007\244 = 075 SYM	#13	007\334 = 006 LAI
	007\245 = 106 CAL OFFSET		007\335 = 042 S(Z)
	007\246 = 367 L		007\336 = 075 SYM
	007\247 = 007 H		007\337 = 076 LMI
#6	007\250 = 373 LMD		007\340 = 000 0
	007\251 = 301 LAB		007\341 = 007 RET
	007\252 = 272 CPC		
	007\253 = 150 JTZ #4E		
	007\254 = 262 L		
	007\255 = 007 H		
	007\256 = 010 INB		
	007\257 = 104 JMP #4B		
	007\260 = 227 L		
	007\261 = 007 H		
#4E/7	007\262 = 006 LAI		
	007\263 = 036 S(X)		
	007\264 = 075 SYM		
	007\265 = 347 LEM		
#8	007\266 = 016 LBI		
	007\267 = 001 I		

OFFSET:

007\367 = 306 LAL
007\370 = 201 ADB
007\371 = 360 LLA
007\372 = 003 RFC
007\373 = 305 LAH
007\374 = 004 ADI
007\375 = 001 I
007\376 = 350 LHA
007\377 = 007 RET

In cases where it is desired to call one or more levels of subroutines within a loop mechanization such as the two FOR loops of CONCATTER, it will be necessary to save the content of the B and C registers whenever a conflicting use is encountered.

In the FOR/END loop mechanization, note that there is a "generated" label for the branch back. The statement number of the for statement itself does not suffice since there is some "initialization" (set up B and C) prior to entrance into the first loop cycle. The assignment into the symbolic loop index "I" implied by the left operand (target) of the FOR statements is done at the beginning of each cycle and serves to mark the branch back points. The branch back points are noted in the 8008 code generation by the statement number followed by the letter "B".

In the FOR/END group shown, the test for end of execution is made after a cycle is completed and before the calculation of the next value of the index. In the first case, statements #4/#6 of CONCATTER, a statement number is required for the exit case - indicated as "#4E" or (in this example) #7 of the original statements. In the second FOR loop of the example, I moved the return statement (#12) ahead to follow the comparison, rather than placing a branch forward at that point. In so doing I was acting as an "optimizing" compiler of the SIRIUS language - using as input the global knowledge of the program in order to figure out a "special case" allowing the movement of code. A similar special case was recognized at statements #2/3 where the jump on condition of #3 is placed ahead of the data storage portion of #2 in order to avoid insertion of a mechanism to save the carry flag across the SYM lookup.

On the following page is one additional set of SIRIUS coding and equivalent 8008 generated code. The routine is a "DRIVER" to call the CONCATTER routine with test data in X and Y (printed separately as two lines), followed by printing of the results of CONCATTER as a single line. The SIRIUS code is extremely simple - virtually a series of calls. A routine called TSTRING is used to do the typing of byte strings, as found within the "ELDUMPO" program of January 1975 ECS. If you employ any form of hard copy or CRT output, an equivalent routine would of course be employed to transfer byte strings to the appropriate external unit. In the driver, the term "HL" is used to denote the H/L pointer pair of an 8008, which would be the H/L pair if you generate for an 8080, or the "X" register of a Motorola 6800. This use of the pointer for argument passage is a workable one but only a temporary "kluge" at present.

What good is concatenation you ask? The idea is illustrated by the diagram given previously. Its use is its justification. The primary application is in the process of "building" a character string, as often occurs when you want to format the output of a program. The CONCATTER routine only handles two strings, but by feeding the output of one concatenation into the next, strings of arbitrary length (to 255 with CONCATTER) can be built from numerous components. As an example, suppose that a conversion routine has provided a program with the strings "X" and "Y" as answers to a problem, and that the text "FIVE GLEEPS AT [??X??] WERE SIGHTED NEXT TO [??Y??] GLOOPS." is to be printed. Start with Z="FIVE GLEEPS AT "; concatenate [??X??] on the right giving a new Z; concatenate " WERE SIGHTED NEXT TO " on the right giving a new Z; concatenate [??Y??] on the right giving a new Z; then concatenate " GLOOPS." on the right giving a new Z which is printed.

THIS IS ← X value...
 A BIG STRING. ← Y value...
 THIS IS A BIG STRING. ← Z = X cat Y

Output of Driver Program

021
 VIM
 81

CONCATTER Test Driver (8008)

SIRIUS Code of Driver...

#1	007\000 = 106	CAL	1	DRIVER:	CALL	NEWLINE
	007\001 = 354	L	2		=::	W(X)
	007\002 = 007	H	3	HL	CALL	TSTRING
#2	007\003 = 006	LAI	4		CALL	NEWLINE
	007\004 = 036	S(X)	5		=::	W(Y)
	007\005 = 075	SYM	6	HL	CALL	TSTRING
#3	007\006 = 106	CAL	7		CALL	CONCATTER
	007\007 = 166	L	8		CALL	NEWLINE
	007\010 = 011	H	9	HL	=::	W(Z)
#4	007\011 = 106	CAL	10		CALL	TSTRING
	007\012 = 354	L	11		EXIT	
	007\013 = 007	H				
#5	007\014 = 006	LAI		NEWLINE:		
	007\015 = 040	S(Y)	1	HL	=::	W(NLTEXT)
	007\016 = 075	SYM	2		CALL	TSTRING
#6	007\017 = 106	CAL	3		RETURN	
	007\020 = 166	L				
	007\021 = 011	H		NLTEXT:		
#7	007\022 = 106	CAL			"006,000,012,000,015,000,007"	
	007\023 = 200	L				
	007\024 = 007	H				
#8	007\025 = 106	CAL				
	007\026 = 354	L				
	007\027 = 007	H				
#9	007\030 = 006	LAI				
	007\031 = 042,	S(Z)				
	007\032 = 075	SYM				
#10	007\033 = 106	CAL				
	007\034 = 166	L				
	007\035 = 011	H				
#11	007\036 = 006	LAI				
	007\037 = 002	S(IMPSTATE)				
	007\040 = 075	SYM				
	007\041 = 076	LMI				
	007\042 = 002	2				
	007\043 = 025	KEYWAIT				

NEWLINE:

#1	007\354 = 056	LHI
	007\355 = 007	h(NLTEXT)
	007\356 = 066	LLI
	007\357 = 342	l(NLTEXT)
#2	007\360 = 106	CAL
	007\361 = 166	L
	007\362 = 011	H
#3	007\363 = 007	RET

NLTEXT:

007\342 = 006	Length
007\343 = 000	NULL
007\344 = 012	LF
007\345 = 000	NULL
007\346 = 015	CR
007\347 = 000	NULL
007\350 = 007	BELL

New SIRIUS-MP Operations in DRIVER:

CALL - this translates to the simple sub-routine linkage of the target computer. (No SIRIUS argument linkage assumed.)

EXIT - this translates to the set of instructions needed to return to the "monitor" or "executive" of your software systems - if the ECS software is used, the return is to the "IMP" or its equivalent code on non-8008 computers.

The notation "<series of octal numbers>" preceded by a label is used to denote literal data to be loaded with program.

IMP Symbol Table Extensions for Use With CONCATTER (temporary).

012\316 = 006	} "36" is X
012\317 = 000	
012\320 = 006	} "40" is Y
012\321 = 011	
012\322 = 006	} "42" is Z
012\323 = 100	
012\324 = 000	} "44" is I
012\325 = 230	