M. P. Publishing Co. Box 378 Belmont, Mass. 02178 Vol. 1 No. 5 May '75

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 Oscilliscope 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 Oscilliscope Interface, by James Hogenson. Turn to page 2 for the details which turn your scope into a LIFE matrix, a checkerboard, 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 DGDOI. 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 DGDOI. The remaining circuitry is the control circuitry which decodes the 8-bit input word and allows for completely programmed operation.

PROGRAMMING

Table 1

Op Code			
Binary	*Octal*	*Mnemonic*	*Explanation*
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	¼ 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 mubtivibrator. 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 DGDOI 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 DGDOI 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 DGDOI 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 DGDOI 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 DECLOOP 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 DECLOOP.

XSECLOOP counts off 4 sections per row and jumps back to DECLOOP 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 DECLOOP between each group of rows.

YSECLOOP 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
            00/201 = 201   (TSF)
            00/202 = 121   OUT 10
            00/203 = 006   LAI
            00/204 = 000   (STX)
            00/205 = 121   OUT 10
            00/206 = 006   LAI
            00/207 = 100   (STY)
            00/210 = 121   OUT 10
CLEAR       00/211 = 016   LBI
REGISTERS   00/212 = 000
            00/213 = 321   LCB
            00/214 = 331   LDB
            00/215 = 351   LHB
            00/216 = 361   LLB
            00/217 = 046   LEI
PARITY REG  00/220 = 000
DECLOOP     00/221 = 040   INE
            00/222 = 304   LAE
            00/223 = 044   NDI
            00/224 = 001
            00/225 = 150   JTZ
            00/226 = 246
            00/227 = 000
            00/230 = 066   LLI
            00/231 = 332
DOTLOOP     00/232 = 301   LAB
            00/233 = 024   SUI
            00/234 = 020
            00/235 = 150   JTZ
            00/236 = 253
            00/237 = 000
            00/240 = 010   INB
            00/241 = 307   LAM
            00/242 = 121   OUT 10
            00/243 = 104   JMP
            00/244 = 232
            00/245 = 000
DECLOOPJMP  00/246 = 066   LLI
            00/247 = 333
            00/250 = 104   JMP
            00/251 = 232
            00/252 = 000
XSECLOOP    00/253 = 016   LBI
            00/254 = 000
```

```
            00/255 = 302   LAC
            00/256 = 024   SUI
            00/257 = 003
            00/260 = 150   JTZ
            00/261 = 267
            00/262 = 000
            00/263 = 020   INC
            00/264 = 104   JMP
            00/265 = 221
            00/266 = 000
ROWLOOP     00/267 = 026   LCI
            00/270 = 000
            00/271 = 303   LAD
            00/272 = 044   NDI
            00/273 = 037
            00/274 = 024   SUI
            00/275 = 017
            00/276 = 150   JTZ
            00/277 = 305
            00/300 = 000
            00/301 = 030   IND
            00/302 = 104   JMP
            00/303 = 221
            00/304 = 000
YSECLOOP    00/305 = 303   LAD
            00/306 = 044   NDI
            00/307 = 340
            00/310 = 330   LDA
            00/311 = 024   SUI
            00/312 = 140
            00/313 = 150   JTZ
            00/314 = 326
            00/315 = 000
            00/316 = 303   LAD
            00/317 = 004   ADI
            00/320 = 040
            00/321 = 330   LDA
            00/322 = 040   INE
            00/323 = 104   JMP
            00/324 = 221
            00/325 = 000
END         00/326 = 006   LAI
            00/327 = 206   (TSN)
            00/330 = 121   OUT 10
            00/331 = 377   HLT
            00/332 = 204   (ZNI)
            00/333 = 205   (ZFI)
```

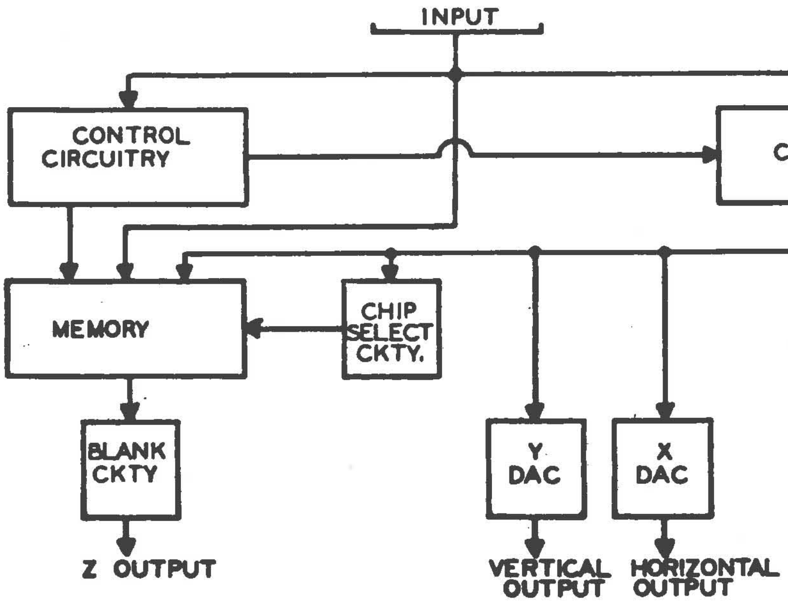


FIGURE 1.
DGDOI BLOCK DIAGRAM

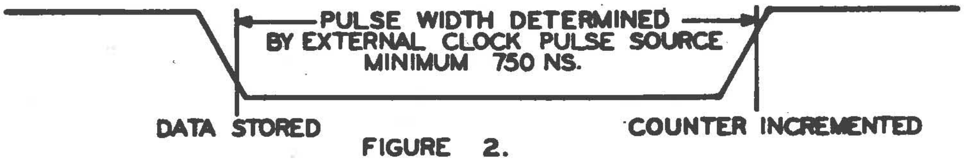

PULSE WIDTH DETERMINED BY EXTERNAL CLOCK PULSE SOURCE MINIMUM 750 NS.
DATA STORED
COUNTER INCREMENTED

FIGURE 2.

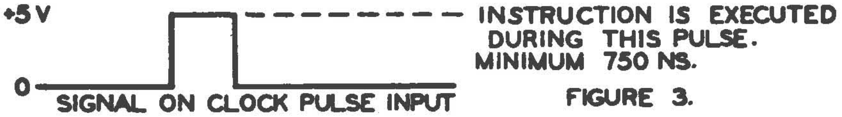

+5 V
0
INSTRUCTION IS EXECUTED DURING THIS PULSE. MINIMUM 750 NS.
SIGNAL ON CLOCK PULSE INPUT

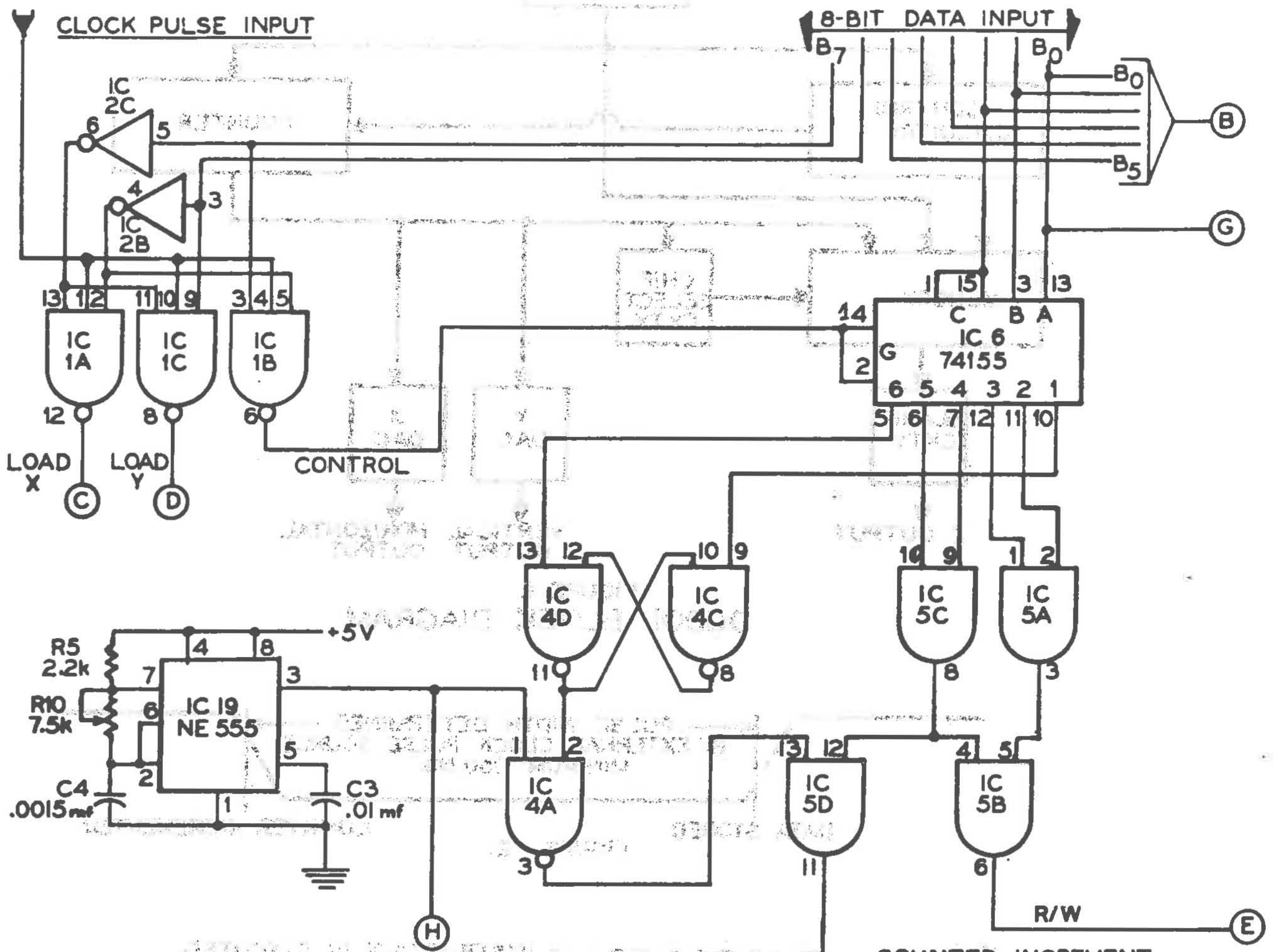
FIGURE 3.

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

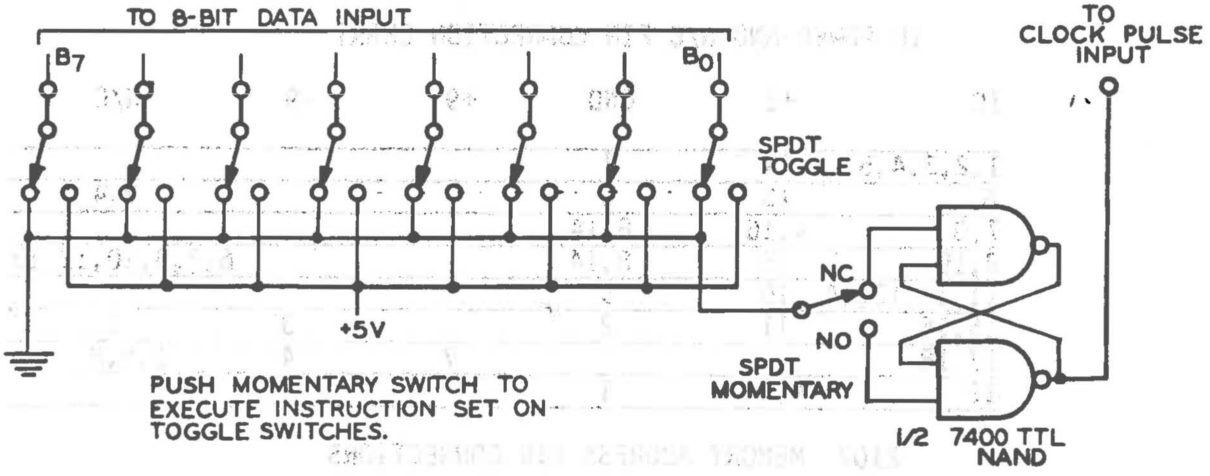
A-0 -- pin 8 : A-1 -- pin 4 : A-2 -- pin 5 : A-3 -- pin 6
A-4 -- pin 7 : A-5 -- pin 2 : A-6 -- pin 1 : A-7 -- pin 16
A-8 -- pin 15: A-9 -- pin 14



CIRCLED LETTERS INDICATE CONNECTIONS WITHIN CIRCUIT.

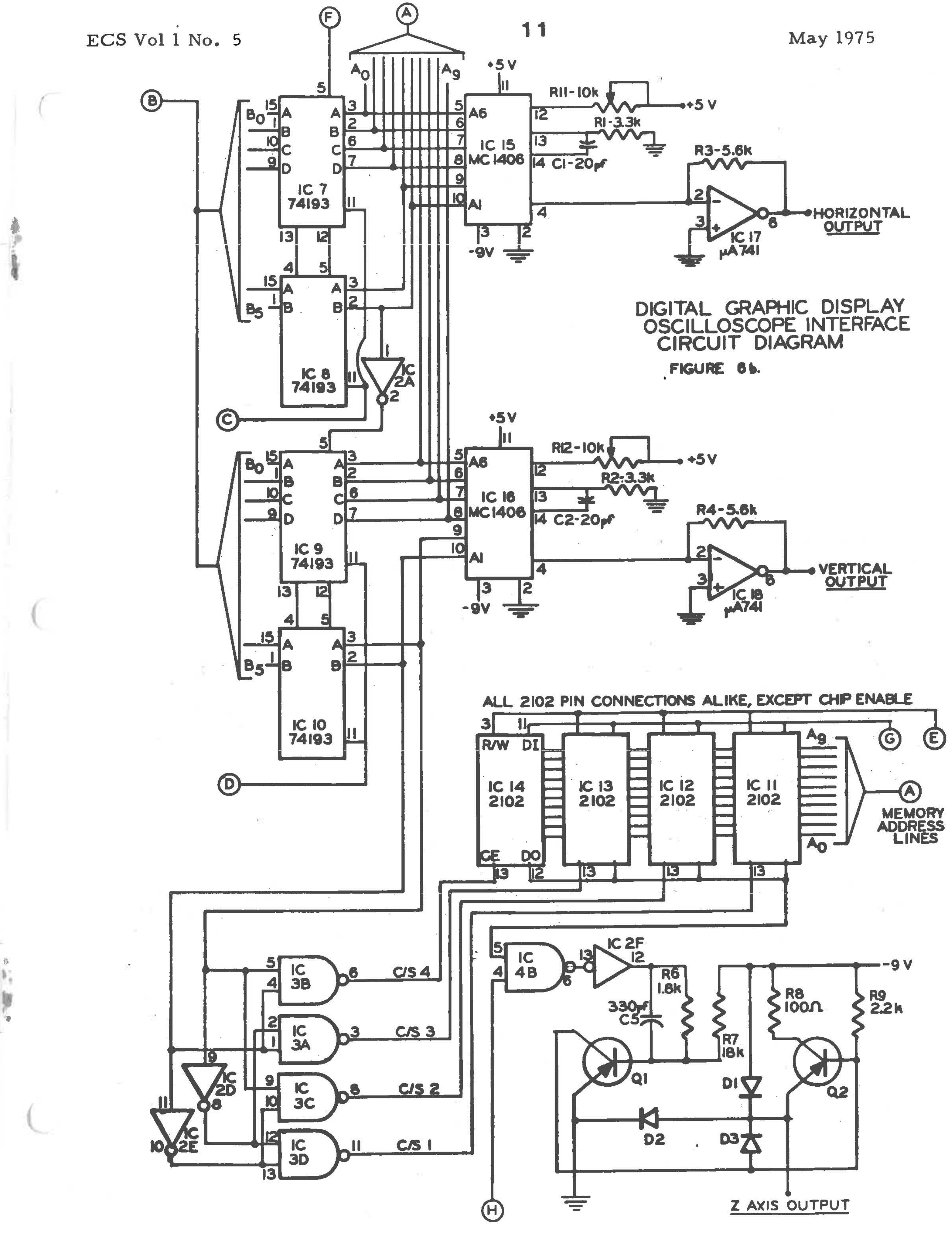
DIGITAL GRAPHIC DISPLAY OSCILLOSCOPE INTERFACE CIRCUIT DIAGRAM

FIGURE 6a.



PUSH MOMENTARY SWITCH TO EXECUTE INSTRUCTION SET ON TOGGLE SWITCHES.

MANUAL TEST CIRCUIT FIGURE 5.



DIGITAL GRAPHIC DISPLAY OSCILLOSCOPE INTERFACE CIRCUIT DIAGRAM

FIGURE 6b.

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 aquisition 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.

- CTH

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 you 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.

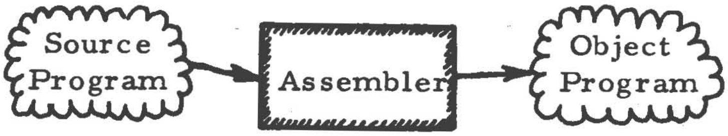
- it provides the microcomputer enthusiast with a method of software development (albeit cumbersome) to be used until his or her personal computer is integrated to the point needed for a real assembler.

- 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.

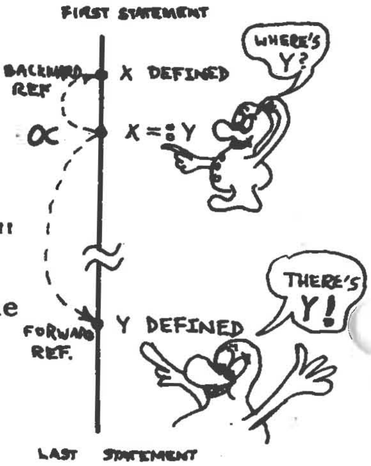
Step 1: Translate the symbolic notations into equivalent sequences of the machine's operations . Pay attention to any address calculations which may be required, but leave "open" the question of addresses of operands for which no address is yet assigned. The purpose of this step is primarily to allocate 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 α 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/1, 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:

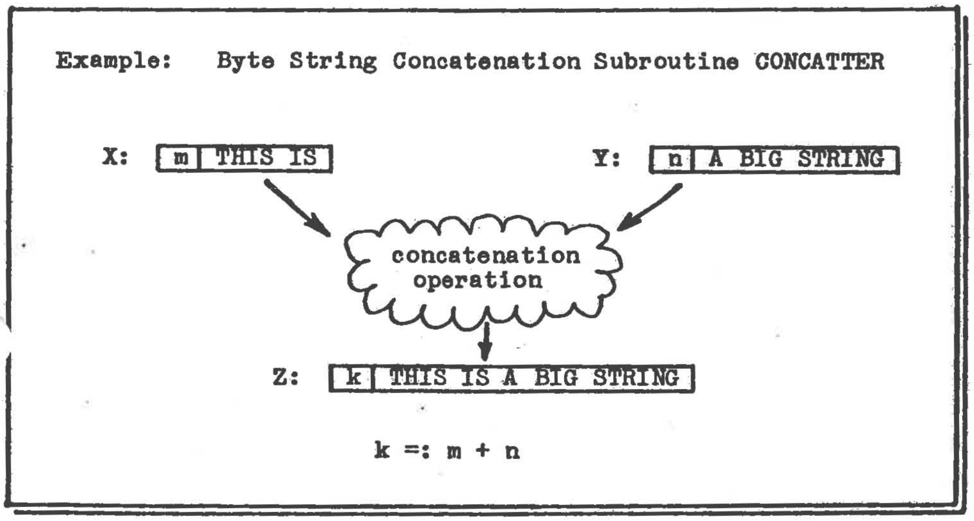
- it has a portion of a literal value which must be translated into its machine code equivalent.
- it has a reference to an address-related value, which for an 8-bit micro means either half of a 16(or 14 for 8008) bit address.
- it has a mnemonic operation code which must be looked up in a table of equivalent 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:

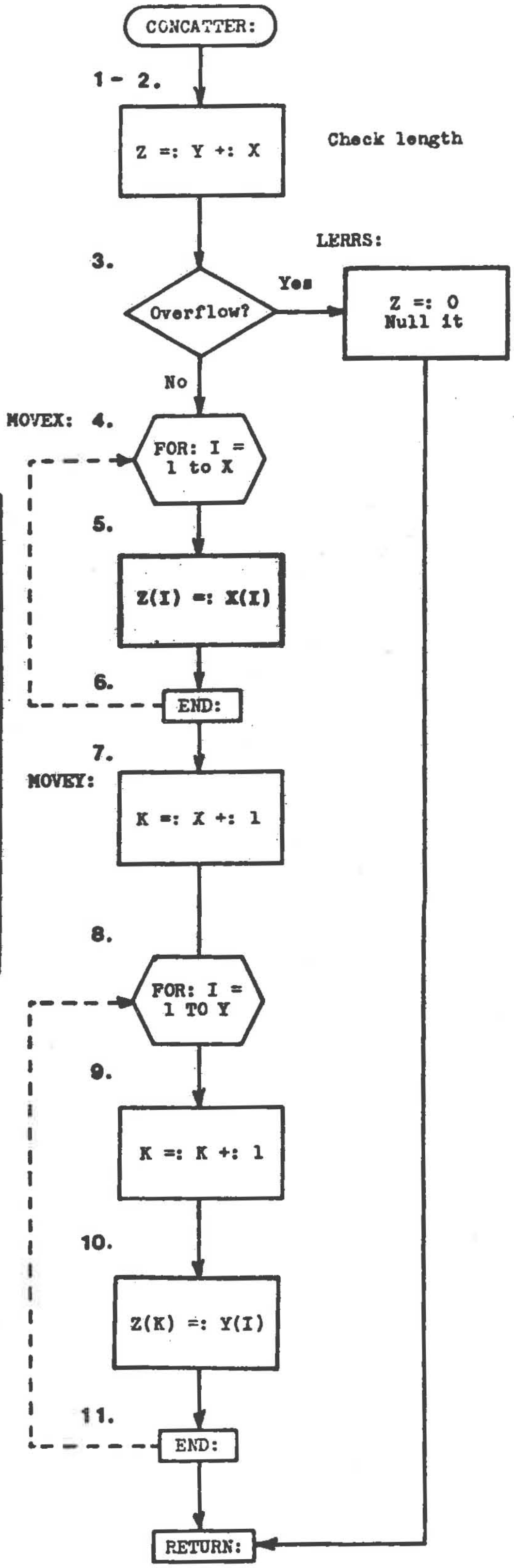
THE BYTE STRING CONCATENATION SUBROUTINE "CONCATTER."

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.

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

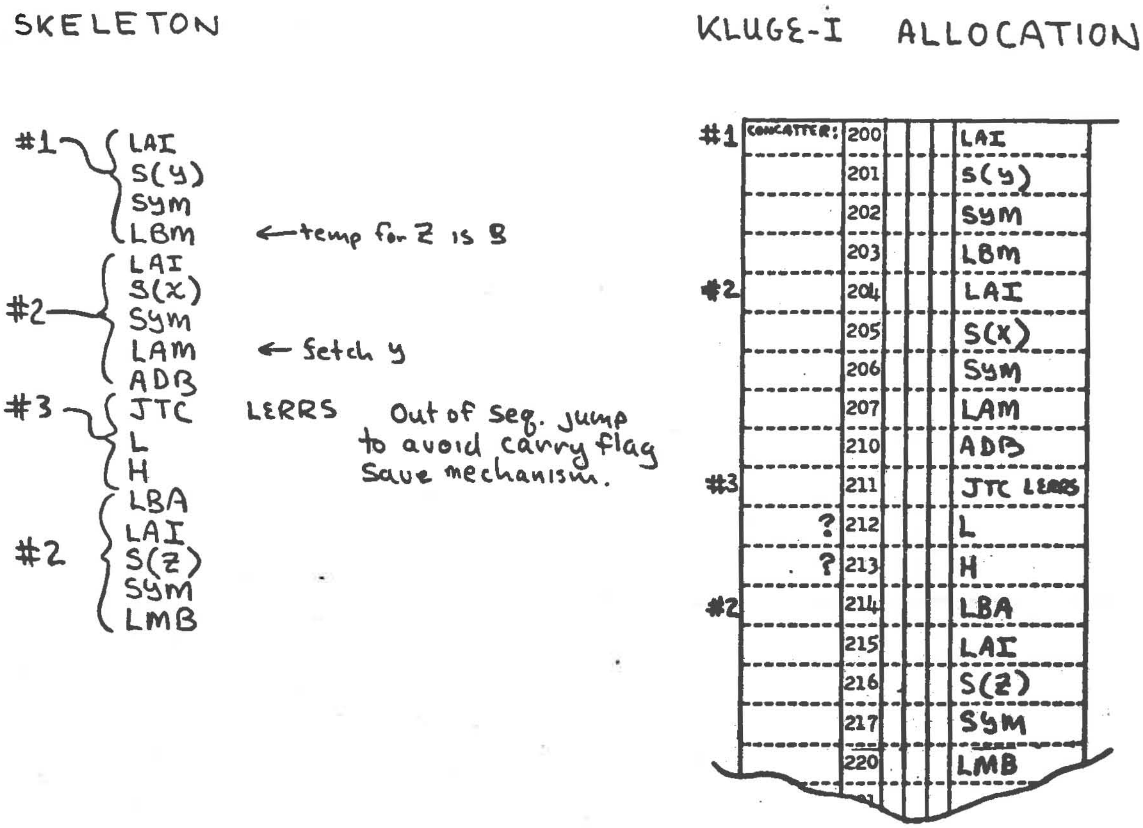
END: --- Incremental "FOR" loop trailer. All the statements from the FOR to the END are considered part of the loop. An implicit (ie: "structured") branch back to the last previous FOR occurs if the iteration count is not exceeded. As with the FOR statement, the END has a type modifier to indicate the 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      ←temp for Z is B
   { LAI
#2   S(x)
     SYM
     LAM      ← fetch y
     ADB
#3 { JTC      LERRS   Out of seq. jump
     L                to avoid carry flag
     H                save mechanism.
   { LBA
#2   LAI
     S(z)
     SYM
     LMB
```

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 "OFSET" 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
       007\201 = 040  S(Y)
       007\202 = 075  SYM
       007\203 = 317  LBM
#2     007\204 = 006  LAI
       007\205 = 036  S(X)
       007\206 = 075  SYM
       007\207 = 307  LAM
       007\210 = 201  ADB
#3     007\211 = 140  JTC #13
       007\212 = 334  L
       007\213 = 007  H
#2     007\214 = 310  LBA
       007\215 = 006  LAI
       007\216 = 042  S(Z)
       007\217 = 075  SYM
       007\220 = 371  LMB
#4     007\221 = 016  LBI
       007\222 = 001  1
       007\223 = 006  LAI
       007\224 = 036  S(X)
       007\225 = 075  SYM
       007\226 = 327  LCM
#4B    007\227 = 006  LAI
       007\230 = 044  S(I)
       007\231 = 075  SYM
       007\232 = 371  LMB
#5     007\233 = 006  LAI
       007\234 = 036  S(X)
       007\235 = 075  SYM
       007\236 = 106  CAL OFSET
       007\237 = 367  L
       007\240 = 007  H
       007\241 = 337  LDM
       007\242 = 006  LAI
       007\243 = 042  S(Z)
       007\244 = 075  SYM
       007\245 = 106  CAL OFSET
       007\246 = 367  L
       007\247 = 007  H
       007\250 = 373  LMD
#6     007\251 = 301  LAB
       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  1
```

```
#8     007\270 = 006  LAI
       007\271 = 040  S(Y)
       007\272 = 075  SYM
       007\273 = 327  LCM
#8B    007\274 = 006  LAI
       007\275 = 044  S(I)
       007\276 = 075  SYM
       007\277 = 371  LMB
#9     007\300 = 040  INE
#10    007\301 = 006  LAI
       007\302 = 040  S(Y)
       007\303 = 075  SYM
       007\304 = 106  CAL OFSET
       007\305 = 367  L
       007\306 = 007  H
       007\307 = 337  LDM
       007\310 = 351  LHB
       007\311 = 314  LBE
       007\312 = 345  LEH
       007\313 = 006  LAI
       007\314 = 042  S(Z)
       007\315 = 075  SYM
       007\316 = 106  CAL OFSET
       007\317 = 367  L
       007\320 = 007  H
       007\321 = 373  LMD
       007\322 = 351  LHB
       007\323 = 314  LBE
       007\324 = 345  LEH
#11    007\325 = 301  LAB
       007\326 = 272  CPC
#12    007\327 = 053  RTZ
#11    007\330 = 010  INB
       007\331 = 104  JMP #8B
       007\332 = 274  L
       007\333 = 007  H
#13    007\334 = 006  LAI
       007\335 = 042  S(Z)
       007\336 = 075  SYM
       007\337 = 076  LMI
       007\340 = 000  0
       007\341 = 007  RET
```

OFSET:

```
       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  1
       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...  }-- Output of Driver Program
THIS IS A BIG STRING. <-- Z = X cat Y }
```

CONCATTER Test Driver (8008)

```
#1    007\000 = 106 CAL
      007\001 = 354 L
      007\002 = 007 H
#2    007\003 = 006 LAI
      007\004 = 036 S(X)
      007\005 = 075 SYM
#3    007\006 = 106 CAL
      007\007 = 166 L
      007\010 = 011 H
#4    007\011 = 106 CAL
      007\012 = 354 L
      007\013 = 007 H
#5    007\014 = 006 LAI
      007\015 = 040 S(Y)
      007\016 = 075 SYM
#6    007\017 = 106 CAL
      007\020 = 166 L
      007\021 = 011 H
#7    007\022 = 106 CAL
      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
```

SIRIUS Code of Driver...

```
   DRIVER:
1                 CALL     NEWLINE
2         HL      =::      W(X)
3                 CALL     TSTRING
4                 CALL     NEWLINE
5         HL      =::      W(Y)
6                 CALL     TSTRING
7                 CALL     CONCATTER
8                 CALL     NEWLINE
9         HL      =::      W(Z)
10                CALL     TSTRING
11                EXIT

   NEWLINE:
1         HL      =::      W(NLTEXT)
2                 CALL     TSTRING
3                 RETURN

   NLTEXT:
          "006,000,012,000,015,000,007"
```

New SIRIUS-MP Operations in DRIVER:

CALL - this translates to the simple subroutine 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 }
012\317 = 000 }  "36" is X
012\320 = 006 }
012\321 = 011 }  "40" is Y
012\322 = 006 }
012\323 = 100 }  "42" is Z
012\324 = 000 }
012\325 = 230 }  "44" is I
```