# IF PROGRAMMING IS A GAME, WHAT KIND OF A GAME IS IT?

PROGRAMMING is a philosophical experience. You search for the underlying patterns in the things you program. Writing a program takes patience and concentration. It used to be that you heard the art of programming compared to chess. But is writing a program really like chess?

In chess you have a set of rules—these can be compared to the instruction set of the computer. Also, you combine these rules in combination with each other— the move of the pieces. But there are differences. You play chess against another chess player and you have only one goal—to win the game. When you play the game of hobby computers, you have no opponent other than the computer. With a computer there are an almost infinite number of goals—Jay Mallin, on page 6 of this issue of QUESTDATA says computers are "anything machines." He is right, and since computers have dropped in price, we find ourselves in the midst of a computer revolution. Computers are taking over more and more tasks.

Since computers can do lots of different things, hobbiests have chosen to have them do the fun things which they enjoy programming and watching their machines perform. This explains why Star Trek and other games are so popular—such games are fun to play. One of the big differences between the big corporation computers and your microcomputer is that you control your microcomputer. You do not have to solve some dull business application by Tuesday. Sure, you can use the computer to solve some business problem if you wish, but you can also use the computer to explore the areas in life which you wish to explore. If you like music or chess you can use your computer to gain new insight into these activities. If you wish to write a really interesting music program, you must explore and understand pitch and tempo.

Let's say the idea of building a really good chess program fascinates a fellow named Melvin. He explores why the Bobby Fisher P-K4 is a good opening for his computer. Melvin devours chess books night and day. Friends wonder why they haven't seen him for months. And then the day of the computer fair arrives and Melvin's 16K sure thing loses to a 2K homebrew in 5 moves. Sigh, it was all for fun. Also, by understanding more about chess Melvin's own game has improved— he no longer loses in three moves.
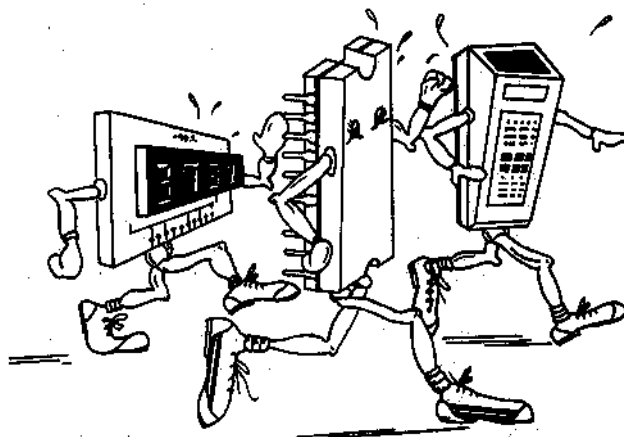
ELECTRONIC TECHNOLOGY IS ADVANCING RAPIDLY

Let's compare programming and golf. In golf you are really playing against your own score since you have very little control over the score of the other person. This is like programming since you want to write programs which are compact and waste as few instructions as possible. Using lots of memory used to be considered sloppy and expensive in the days of core memory. Another part of the programming game is speed, trying to make your program run as fast as possible—so that math problem pops onto your display within the blink of an eye. Both golf and programming take planning—you have to decide which club to use for "an approach shot." In programming the question is—"which instructions are the best choice."

As in other games, you will find your own style and personality coming out in the program. You will find yourself saying, "this must be Melvin's code, since he loves tangled spaghetti code."

In chess, music, golf, programming, and other games, you get better with practice. You become accustomed to favorite "syntaxes" of certain computer languages and you gravitate to FORTH or PASCAL. You find yourself reaching and there is always that next programing mountain to be climbed. Your computer cries out for more data. It seems to say, "may your questing and learning never cease."

FLOWCHARTS: KEY TO PROGRAM SUCCESS

James C. Nicholson

In the first issues of QUESTDATA, you probably have looked at the coding examples and skipped over the flowcharts. And if you didn't skip the flowcharts, you gave them a cursory examination to go on with the written material. You are not a "freak," or "different," or unusual in any way as this seems to be a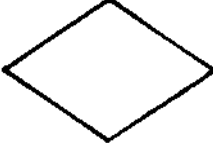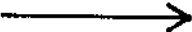 generalized pattern for new programmers. The comment, "Flowcharts are used only by the professionals," is not necessarily a true statement. Flowcharts are needed by EVERYONE. And they should be completed BEFORE you start your programming.

So what's the "big deal" about flowcharts? Very simply expressed, it is a pictorial view of your logic in solving problems. Symbols and quite rigid rules scare a lot of people away from the practice of flowcharting. Actually, they aid you in planning your logic just as a blueprint aids a carpenter in building a house. Breaking these "rigid" rules into a simple set of rules is not that difficult. Let's look at them:

SYMBOLS. There are literally dozens of symbols that can be used in flowcharting, but let's look at, and use, only 6.

SYMBOL	NAME	OPERATION
	terminal	start/stop processing
	operation	arithmetic operations, data movement, setting counters, internal work
	input/output	reading/writing data display status lights
	decision	comparison, questions
	connectors	tie points for branches
	arrows	direction of logic flow

RULES. The rules are actually very simple and can be stated as follows:

- Always start your flowcharts in the upper left corner of your paper.
- Move your logic flow down and to the right.
- Use connectors as often as possible to prevent confusing lines and eliminate line "crossovers."
- Use arrow points to show the direction of flow.
- Always have ONE entry point into any symbol. Multiple entries can be handled as shown below.
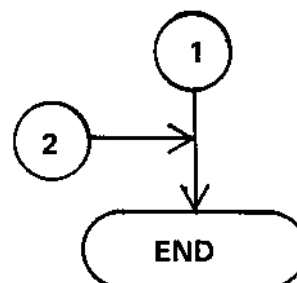
- Always have ONE exit from a symbol except for a decision symbol which may have either two or three exits depending on the comparison. This is the only time that this "one exit" rule can be broken.
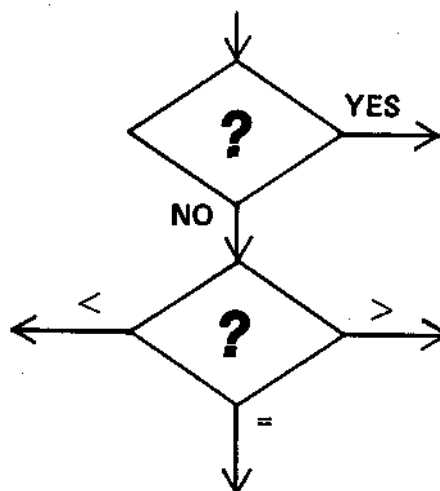
- Place only one distinct action within each symbol.

The "how-to-do-it" is again very simple. You can start with the following procedure in writing your program until you are somewhat proficient in using the symbols, at which time you may eliminate the use of arrows to show flow direction if you wish. The 7 steps in flowcharting are:

(1) Write a program statement to define your program.

(2) Define your input and output requirements, including final output design.

(3) Assign names (or registers in machine language) for your program variables.

(4) Write simple phrases showing the desired action of each step. Do not use symbols.

(5) Convert phrases to flowcharting symbols.

(6) Check out (debug) the logic of the flow-chart AS IF YOU WERE THE COMPUTER. Do not assume any action.

(7) When you are satisfied the logic is correct, code the program in the language you are using.

You will find your error rate in coding will drop immensely after you have started using flowcharting techniques before coding. True, you may have to change the flowchart after you have debugged the program and have it running, but the time taken to redo the flowchart will be much less than the time taken to debug a non-flowcharted program. After all, isn't it a good idea to have everything laid out in order before starting a job. Consider the time when you built your Super Elf.

Want some practice? Try some of these problems on for size:

● Draw a flowchart to make a peanut butter and jelly sandwich with selections of two types of jelly and three types of bread. Decide whether to cut it in half or serve it whole. (After you have completed the flow-chart, try it out in the kitchen. This is one time you can "eat your mistakes.")

● Draw a flowchart to set your alarm clock before retiring and test the time every minute until it is time to sound the alarm. Then turn on the Q-LED. (If you are using a relay mod-ification, sound an alarm). Try coding it for the Super Elf.

● Draw flowcharts from various coded pro-grams. This is a very helpful tool in under-standing the logic of the program and learn-ing the instruction set of the RCA 1802. It also will give you a more complete docu-mentation folder for the program applica-tion.

Well, that's it. Flowcharting is a blessing if you understand it and look at it as a joyful experience in computing. After all, anything that makes running our Super Elf easier should be fun. Try it. You'll get a lot out of it.

END
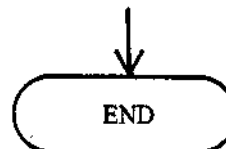
MEMORY RECALLER

By David K. Taylor

This program is a modification of the counter pro-gram which appeared in the March 1977 *Popular Electronics*. The program displays the stored memory bytes of the program you have entered so that you can check to see that all the digits are correct. THE MEMORY PROTECT SWITCH MUST BE ON WHEN YOU RUN THIS PROGRAM. If the mem-ory protect switch is not ON the contents of the memory will be changed. When the program is exe-cuted, the display will sequentially present the mem-ory contents at a rate determined by the byte in loca-tion 0D.

LOC.	CODE	MNEM.	COMMENTS
00	F8 00	LDI	INITIALIZE REG 1
02	B1	PHI R1	HIGH
03	F8 00	LDI	INITIALIZE REG 1
05	A1	PLO R1	LOW
06	E1	SEX R1	X=1
07	64	OUT 4	DISPLAY M(R(X)); R(X)+1
08	F0	LDX	LOC. R(X) INTO D REG.
09	FC 01	ADI	INCREMENT D REG.
0B	51	STR R1	STORE NEXT LOC.
0C	F8 30	LDI	LOAD DELAY
0E	B2	PHI R2	R2.1=DELAY
0F	22	DEC R2	R2-1
20	92	GHI R2	D=R2.1
21	3A 0F	BNZ	DELAY UP?
23	30 06	BR	YES THEN REPEAT

WHAT THE MACHINE IS THINKING

Let's take another look at the D-Register (the D stands for data). The **D-Register** is a kind of central communications center for the 1802 microprocessor. In fact, it is the Grand Central Station of the 1802. That is, information passes through the **D-Register** on its way to other registers and parts of memory. A lot of data bound for other points must make a temporary stop at the **D-Register**. It is like stopping at Chicago on your way to your destination of New York.

How would you load up the **D-Register** with data which is bound for another register as its destination? First we must load the **D-Register** with the hexadecimal number we wish to have end up in the General Purpose Register Matrix. If we want the hex number FF to be loaded we can use the LOAD IMMEDIATE (LDI or F8 in machine code). Looking at Users Manual Number 201 we recall: "The byte immediately following the current instruction byte replaces the byte in D." So that is what we will do, we scribble down F8 follwed by the number we wish to have in the **D-Register**—in this case FF.

There are a number of books out that compare programming to cooking: *The Programming Cookbook This* and *The Programming Cookbook That*, for example. Anyhow, we will let the egg and mayonaise set for awhile (errr, the loaded **D-Register**). Turning our attention to the asparagus (ahhh, General Purpose Registers), we prepare another part of the program recipe.

GENERAL PURPOSE REGISTERS

The COSMAC 1802 has 16 General Purpose Scratchpad Registers and they are each 16 bits long. We can guess that these General Purpose Registers are important or RCA wouldn't have made so many of them. . . and we would be right. In actual area, the General Purpose Registers can be seen to take up a good portion of the 1802. Turning to the back of your Super Elf manual or 1802 specifications sheet, you can see that these registers look something like a Navaho weaving and are located in the upper left hand corner of the microprocessor. Even though they represent a large part of the 1802's architectural investment, they are worth it for the programming tricks they will enable us to perform. Since there are 16 of these General Purpose Registers, they can be numbered 0 thru F. Each one of these 0 thru F registers has an upper and lower portion. For example, the high part of Register 9 has 8 bits and the low part has 8 bits, for a total of 16 bits in all.

LOADING A GENERAL PURPOSE REGISTER

So pick a register, any register. Let's say you picked Register 9 as the lucky register to be loaded. OK, so how do we do this? PUT LOW and PUT HIGH sound like they will do this job. Here is the official RCA description of PUT LOW: "The byte contained in the **D-Register** replaces the low-order byte of the register specified by N. The contents of D are not changed." This sounds like what we want. Since A is the first byte of this instruction and carries the meaning (PUT LOW), and since N (the second byte) is to stand for the register we want to put the D-Register data into: We quickly write down A9. Putting the **D-Register** high is a similar process—B9. Putting all parts of the programming puzzle together, we have:

```
LDI    F8
       FF
PLO    A9
PHI    B9
```

The instructions A9 and B9 say that the D-Register is not destroyed when we PUT LOW or PUT HIGH, so we know that Register 9 looks like:

HI	LO
FF	FF

WHAT DO YOU DO WITH A LOADED GENERAL PURPOSE REGISTER

Now that we have Register 9 loaded, we might as well do something with it. Let's see, we could use it as a pointer:

Anyhow, a pointer register points the way for other data to follow. In this case, we are pointing to FF FF, the very last location in memory. If we wanted to store something in this location we would load it into the **D-Register** (F8—followed by the data we wish to load in FF FF; say, hex 35) and with the STR (59 in machine language) instruction—project the data into the memory location. This is kind of like putting things in the transporter room of the Starship Enterprise (of Star Trek fame), and beaming them down to a strange planet. Well, in a way it is. Anyhow, we are not going to use the STR instruction today. If you want to use the instruction go ahead but QUEST-DATA cannot assume any responsibility for your use of his instruction. Remember the black box with the mysterious hand trick? Well, in that old toy automation, a hand would reach out of a box and throw the switch that turned itself off and then would sneak back inside the strange box. A very similar case to what we have here. See, this pointer hand just escape from this box and is now going back into the box.

Where were we. If we are not going to use Register 9 to point, why don't we just decrement it down to

zero and use it for a very long delay. To decrement it we can use the DEC or machine language 29 instruction. The left hand byte (the 2) stands for the decrement, and the right hand byte (the 9) tells which register we wish to decrement.

So after issuing this instruction we have FF FE in Register 9.

If we wish to test this number to see if it is zero, we have to dump it back into the D-Register bucket. Since the lower byte reaches 00 more quickly than the higher byte, let's test the higher part of Register 9 (because we want a long delay). Looking around for an instruction to use we quickly seize GHI (9N in machine language). This instruction takes the high part of the register designated by N and puts it into the D-Register. Thus, 99 in machine language will do exactly what we want. We now interrogate the D-Register, and ask it if it has reached zero yet. The machine language conditional 3A, as you recall from previous QUESTDATA's, does this task for us. Thus, we can continue to branch until the D-Register reaches zero (BNZ or 3A), and when the D-Register equals zero, the flow of instructions continues.

Reviewing what we have scribbled down thus far in machine language, we see:

```
       LDI      F8
                FF
       PLO (9.0) A9
       PHI (9.1) B9
  ┌──► DEC (9.0) 29
  │    GHI (9.1) 99
  │    BNZ      3A
  └──────────► X   (LET X=BRANCH LOC.)
       CODE CONTINUES . . .
```

If we put 7B at the start of the program and 7A at the end, we will have: Q-LED turned on, a long delay, Q-LED turned off. Try it. STARTING AT MEMORY LOCATION 00: 7B, F8, FF, A9, B9, 29, 99, 3A, 05, 7A, 00. Save your memory contents after entering this program since we will be using them in the next project.

THE NEXT PROJECT

Pretend that at a certan hour the City of Los Angeles wants all its stoplights to change to blinking yellow. If you were given an 1802 microprocessor, how would you make a stoplight blink. In real life you would also have to program around a clocking device to start the blinking lights at a specific time of day. For this particular project you do not have to worry about time but are commissioned just to make a blinking program. One way to do this would be to test whether the Q-LED is OFF (BNQ), and if the Q-LED is not ON, then turn it ON. Given this information and being in command of the situation; you can now make the L.A.P.D. happy. For the answer, turn to page 13 but first give it a try on your own recognizance.
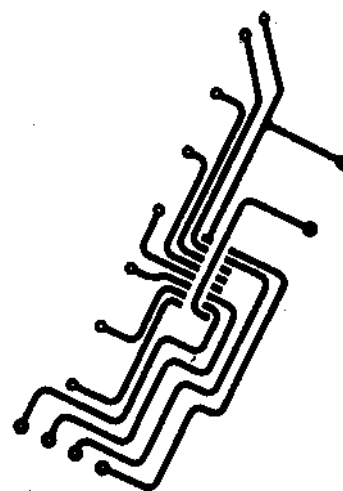
The above "picture" appeared in QUESTDATA Issue Number 1. After some quick research into the matter, QUESTDATA has determined that reader Peter Estelle is absolutely correct. . . .

"By the way I believe your crazed arachnid to be a lonely 4511 BCD-to-7 segment LED converter/driver. Here was my line of thought. Hmmm . . . Power and ground in the standard locations . . . Seven inputs or outputs on the same side of the chip . . . Four inputs or outputs on the other side . . . A converter? . . . BCD to seven segment? . . . V+ also goes out with the seven outputs . . . To the databooks I went to find the 4511. Am I right?"

—Peter Estelle

"Is it a drawing of that rare insect, 'Buggus Hardwarus?' or simply a Super Elf footprint?"

— Bob Richie

Reprint #1

YOUR ANYTHING MACHINE TURNS INTO AN ALARM CLOCK

By JAY MALLIN

A computer is basically an anything machine. With the right input/output devices, and within its speed limitations, it can imitate just about any other electronic device.

You can, for instance, program your Elf to play video games. At first this may not seem like much, since you can go out and buy a ready-made machine that just plugs in and does the same thing. However, that video game can never do anything else. Your Elf can also be a timer, tone generator, a calculator a . . . an anything machine. It's up to your imagination.

With the accompanying software and some supersimple hardware, you can turn your personal anything machine into a digital alarm clock. Designed for use on a Super Elf or Elf, the hours and minutes are displayed on the hex output display, the Q LED tells you whether or not the alarm is set, and a speaker attached to the Q output gives a quiet buzzing for one minute when the alarm goes off.

First put together the opto-isolator circuit, using wire wrap or other techniques. Layout is not critical, and just about any LED/phototransistor isolator will do. The input is attached in parallel with the computer across the 10 V power transformer—on the 10 V side, not the 120 V side. Attach the phototransistor side of the isolator to the $\overline{EF3}$ line and the computer ground, making sure the ground gets the emitter. Make sure the transformer won't be shorted by a loose wire to the computer, since the $\overline{EF3}$ line is connected directly to the microprocessor and a short could fry its brains out (nightmare of nightmares! By the way, an opto-isolator is a good way to keep that from happening, since it can provide the computer with thousands of volts of isolation from whatever circuitry is attached to its other half.)

Next, put the program into memory. Addresses 01 and 04 should be the minutes and hours, respectively, that you initially want to set your computer to; 1E and 23 are the minutes and hours you want the alarm to go off at if it is set.

When everything is set and checked, hit the RUN or GO button. The display should show the minutes that you set it to. Now depress the INPUT switch. The display should show the hours.

Depending on whether you have a keyboard or switches, hit the 0 key twice or set all your input switches to zero. The Q light will come on, showing that the alarm is set. To turn the alarm off, move any switch to the 1 position or hit any key other than 0, and the Q Led should go off. With the alarm set, the clock will buzz for one minute when the clock reaches the time you put into the memory. A regular Elf will need a simple transistor amplifier to hook an 8 ohm speaker to the Q line. The Super Elfs already have them.

If nothing has happened except for the run light coming on, the opto-isolator circuit probably is not working. To check this detach the leads going to the computer ground and $\overline{EF3}$, and substitute an ohmmeter. If it gives a reading near 0, or infinite with the transformer plugged in, then this is what's causing the problem. Try reducing the resistor and make sure the diode is connected correctly. Also make sure that the circuit is attached correctly to the computer and transformer. If the program seems to be behaving erratically in other ways, you probably made a mistake while putting the clock program into memory.

With everything working correctly, your Elf is keeping time as accurately as any clock that plugs into an AC outlet. Like them, it is based on the 60 Hz frequency in the power line. The opto-isolator's LED is flashing on and off at this frequency since it is powered by the AC from the transformer. To the computer, this looks like a switch attached to the $\overline{EF3}$ line being turned on and off 60 times a second. The program loops once each time the "switch" is "on," and every 3600 times advances the minutes by one.

Other parts of the program are used to make the minutes and hours advance correctly. The computer would like to count in hex, but it would not do to have it displaying times like 12:4A, so six is added to make it come out as 12:50.

Variations on this clock program are endless.

An interesting part of the program is the part which makes the buzzing noise when the alarm goes off. This is done by bytes 26 through 29, which "flip-flop" the Q latch. That means if the Q is on it is reset and if it is off it is set. Done once every loop, this gives a 30 Hz buzzing noise.

Variations on this clock program are endless. You can have the Q line turn things on and off as set by the clock. You can do away with the alarm, which lets you use Q for something like an AM/PM indicator and frees up the keyboard for other uses. You can change the program from byte 45 on to make a 24 hour clock instead of a 12 hour one. You could improve your programming skills this way since you would probably need to figure out how the current bytes work, and reading other peoples' programs is always a good way to pick up some new tricks. If you make the program longer, remember to change the stack location, set in byte 07.

If you want it to, your Elf could also keep track of days, months, years, and seconds. Finally, you can use the opto-isolator setup as a time base for timers and stop watches down to the 10th or 60th of a second.

You can have your Elf do just about anything. Just use your imagination and remember, it's your anything machine.

LOC.	CODE		MNEM.	ACTION
00	F8	__	LDI	SET MINUTES
02	AB		PLO	INTO RB.0
03	F8	__	LDI	SET HOURS
05	BB		PHI	INTO RB.1
06	F8	55	LDI	STACK LOCATION
08	A2		PLO	INTO R2
09	F8	00	LDI	
0B	B2		PHI	
0C	E2		SEX	X IS R2
0D	30	2E	BR	GO TO 2E
0F	36	0F	B3	RETURN IF EF3=1
11	37	15	B4	IF INPUT DOWN BRANCH
13	8B		GLO	MINUTES INTO D
14	38		NBR	SKIP NEXT
15	9B		GHI	HOURS INTO D
16	52		STR	STORE D IN MR2
17	64		OUT4	OUTPUT MR2
18	22		DEC	RESET R2
19	6C		INP4	INPUT
1A	3A	29	BNZ	D NOT 0 BRANCH
1B	8B		GLO	MINUTES INTO D
1D	FF	__	SMI	SUBTRACT ALARM MINUTES
1F	3A	27	BNZ	IF D NOT 0 BRANCH
21	9B		GHI	HOURS INTO D
22	FF	__	SMI	SUBTRACT ALARM HOURS
24	3A	27	BNZ	IF D NOT 0 BRANCH
26	CD		LSQ	FLIP FLOPs Q FOR
27	7B		SEQ	BUZZING NOISE
28	38		NBR	
29	7A		REQ	
2A	9A		GHI	RA.1 INTO D
2B	3A	32	BNZ	IF D NOT 0 BRANCH
2D	1B		INC	INCREMENT MINUTES
2E	F8	0F	LDI	RESET RA TO 0F0F
30	AA		PLO	(USED TO DIVIDE
31	BA		PHI	60 Hz. by 3600)
32	2A		DEC	RA-1
33	3B	33	BN3	RETURN IF EF3=0
35	8B		GLO	MINUTES INTO D
36	FF	5A	SMI	IF MINUTES = 5A
38	32	44	BZ	THEN BRANCH
3A	FA	0F	ANI	IF MINUTES DOES NOT
3C	3A	0F	BNZ	NEED A CARRY,RETURN
3E	8B		GLO	IF IT DOES, ADD 6
3F	FC	06	ADI	AND RETURN
41	AB		PLO	
42	30	0F	BR	
44	AB		PLO	RESET MINUTES
45	9B		GHI	IF HOURS
46	FF	09	SMI	IS EQUAL TO 9,
48	C6		LSNZ	THEN CARRY
49	F8	06	LDI	
4B	FF	09	SMI	IF HOURS
4D	C6		LSNZ	IS EQUAL TO 12,
4E	F8	EE	LDI	THEN RESET IT
50	FC	13	ADI	
52	BB		PHI	
53	30	0F	BR	RETURN

To EF3 Line

To Computer Ground

Opto-Isolator

1KΩ

To Power Transformer
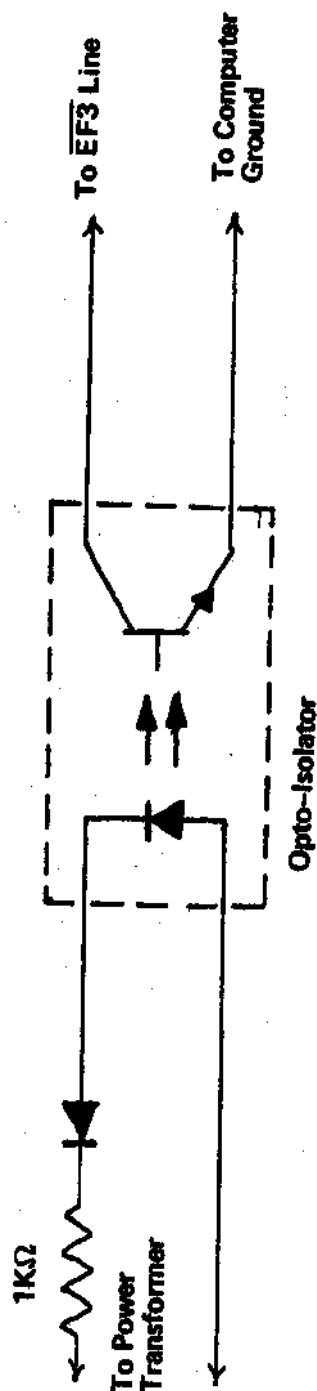
REGISTERS

R0 is program counter

R2 is X counter

RA is used to divide 60 Hz at EF3 by 3600 for minutes

RB.0 contains the minutes byte of the current time

RB.1 contains the hours

START

RB.0←MINUTES
RB.1←HOURS
R2←0055 X←2

EF3=1
?
— NO

YES

INPUT SWITCH DOWN ?
— YES

NO

OUTPUT MINUTES

OUTPUT HOURS

INPUT KEYBOARD

INPUT≠00
?
— YES → Q←0

NO

RB≠ALARM TIME ?
— YES → Q←1

NO

Q←∿Q
FLIP-FLOP
Q

RA.1≠0
?
— YES

NO

R8+1
RA←0F0F

RA-1

EF3=0
?
— NO

YES

RB.0=5A
(1 HOUR)
?
— NO

YES

RB.0=XA
?
— NO

YES

RB+6
(CARRY)

RB.0←00

RB.1=12
?
— NO

YES

RB.1←00

RB.1=09
?
— NO

YES

RB.1+06

RB.1+1
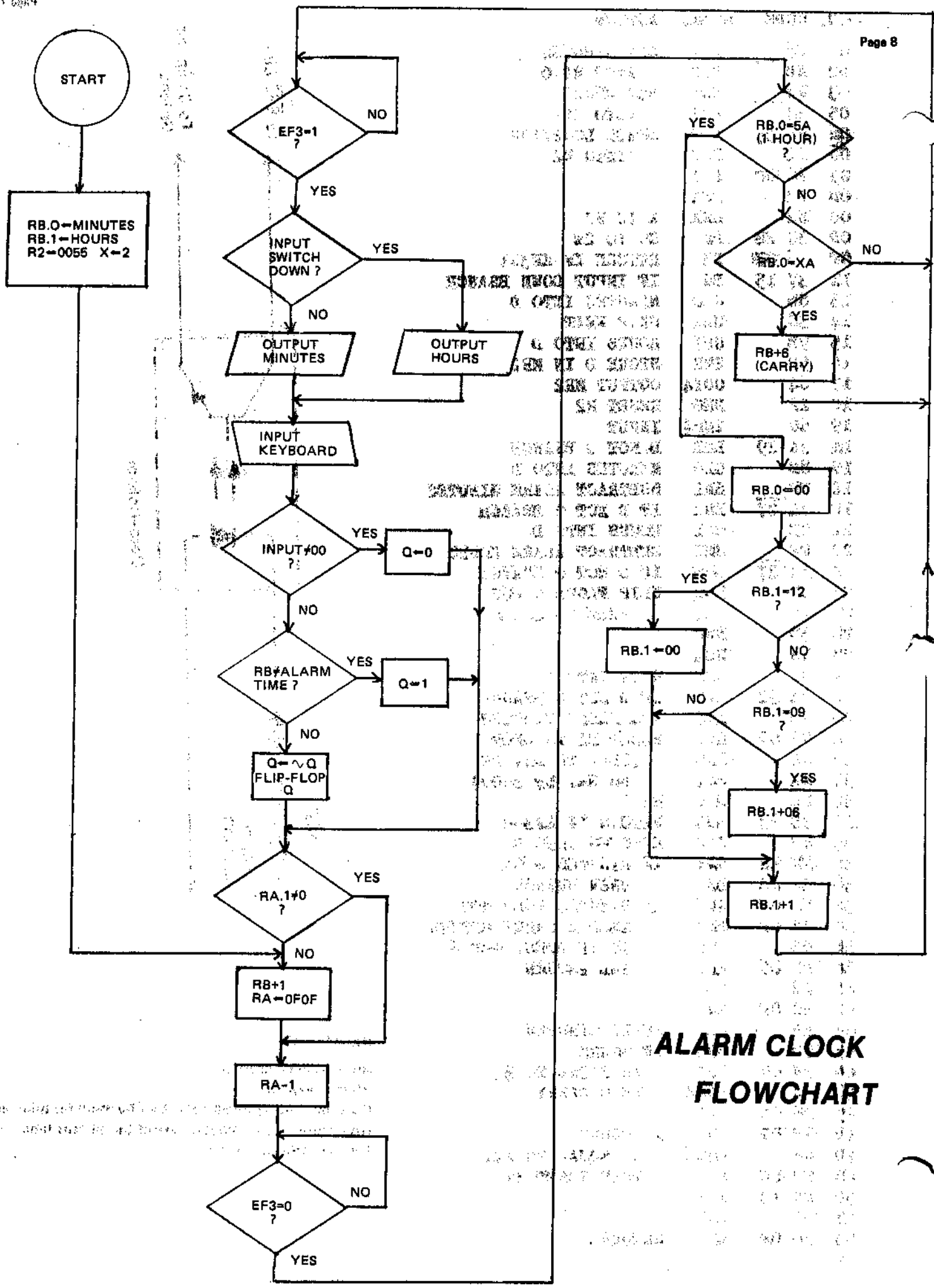
ALARM CLOCK
FLOWCHART

Returning to the original purpose of the interrupt subroutine, if the routine was merely to take care of the odd-cycle timing correction and just provide the display starting address, obviously the intent of displaying just page 0 would not be satisfied. So the approach taken is to display each line four times, thereby cutting the memory display to one page. This, then, requires the resetting of the data display pointer between every group of four DMA bursts. However, when this approach is taken, the automatic way that the refresh would take care of itself is now given up, and the subroutine has to continually reset the display pointer throughout the refresh period. This, therefore, adds to the overhead of the microprocessor timing cycles needed to refresh the display about 50% of the total time, up from less than a third otherwise. In addition, the termination of the refresh period must now be tested for the subroutine periodically in order to know when to get out of it. The status line EF1 is provided for that purpose. It goes logically high when the refresh period is up.

INTERRUPT SUBROUTINE ANALYSIS

The first location of the subroutine is location 000F. Paradoxically, the first two instructions are not to enter the subroutine, but to leave it. The microprocessor does not start executing instructions at location 000F when an interrupt occurs but rather at location 0011, which, if you recall, was loaded into REGISTER 1 during initialization for that purpose. The reason why the first part of the subroutine contains the exiting instructions is so that REGISTER 1, which is the P REGISTER pointer while in the subroutine, is left pointing to the entry again when the exit is made. This is a typical way to provide the entry address of a subroutine after it has been accessed instead of having to reload the address. This makes the subroutine "re-entrant."

To make following the flow of the subroutine easier, it is best to analyze the entering and exiting in that order first. Since we have taken care of the overhead instructions, we can now proceed to the heart of the subroutine.

The entry and exit overhead instructions are for the saving of data values in the stack which is necessary to get the main program running again from where it left off when the interrupt occured. At location 0011, the "entry" to the subroutine, is a DEC instruction to point to the "next" preceding position in the stack. Note that REGISTER 2 is decremented here. The interrupt re-adjusts X to refer to REGISTER 2 as part of its automatic function. The main program also uses REGISTER 2 as the stack pointer so the DEC instruction insures that whatever value is in the stack does not get stored over if the same position were used. The DEC is also needed because the RET instruction in the subroutine exit advances the stack, and without the DEC, the wrong stack position would be handed back to the main program.

The next instruction in location 0012 is the SAV instruction. This causes the current main program values of X and P, which in this program are always X=2 and P=3, to be obtained and stored in the stack. At interrupt, these values were stored by the microprocessor. The next instruction in location 0013 is another DEC to get the next preceding location of the stack for saving the current value of the D function REGISTER to restore it to the main program at exit. This is accomplished by the STR in location 0014.

The exiting instructions work in reverse to the above instructions in locations 0011 and 0014. In location 000F, the LDXA instruction loads from the stack, which is still sitting in the same position as it was left after the DEC instruction in location 0013, and the previously stored D value is put back into D again. The LDXA also advances the stack to point to the position now where the DEC instruction in location 0011 left it. Now the RET instruction in location 0010 can retrieve the prior X and P values from the stack, restore the main program X and P REGISTERS, and advance the stack back to where the main program was using it. Thus, this portion of the program is re-enabled and further interrupts will start the main program running again.

INTERRUPT SUBROUTINE

The only purpose of the interrupt subroutine is to provide the starting address of the data to be copied and put on the TV screen by the video generator. In order to explain how this is done, a thorough analysis of what happens when an interrupt occurs and the subsequent video display is completed is needed. Thus, an explanation of the video generator function and video synchronization will be given.

VIDEO GENERATOR

The video generator of the Super Elf works very efficiently with the 1802 and memory. It can be thought of as an output device whose sole purpose is to drive a TV monitor and to present data from memory on the screen of the TV. It is capable of displaying up to four pages of memory or as little as eight bytes of memory if so desired. The way the data is obtainied by the video generator for the display is by means of what is called DIRECT MEMORY ACCESS, or DMA. DMA means that the video generator obtains the data directly from memory without the aid of the microprocessor itself. Therefore, a program doesn't need to provide the data by looping through and executing output instructions but just by providing the location of the data. If no further action is taken by the program, the rest of the operation of the video generator is automatic and 1024 contiguous bytes of data would be fetched automatically and routed to the TV screen. An important thing to realize is that not only is the operation of the video generator automatic, but it is necessary as well. The video display consists of 128 lines on the screen of the TV and 8 bytes of data make a line. Thus, the video generator requires a stream of 1024 bytes every time the screen is written onto (or "refreshed"). Complete refresh occurs about 61 times a second in the Super Elf. The video generator presents an interrupt to the microprocessor which causes a holdup on whatever it is doing. Then, 1024 bytes of memory data is obtained via DMA and written onto the TV screen. At the time of the interrupt, exactly 29 machine cycles later, which is approximately 130 microseconds, DMA starts.

The DMA actually occurs in bursts. At the moment of DMA, REGISTER 0 is automatically used for the location of data in memory. Each burst obtains just 8 bytes, lasts 8 machine cycles, (approximately 36 microseconds), after which REGISTER 0 will have been incremented by 8. There is a time lag of 6 machine cycles until the next DMA burst which is exactly like the prior one. This process continues for a total of 128 DMA bursts and the TV screen has been written upon from top to bottom. The first burst of DMA provides the top line of the screen and the data is written from left to right for a total of 64 positions, each bit, of the eight bytes, representing one of the 64 positions. This process then repeats itself. If the bit of a byte of data is a logical "1" then the corresponding position on the screen is a bright rectangle, if the bit is a "0" then its position is a dark rectangle.

If it were desired to display four consecutive pages of memory, it can be guessed that the interrupt sequence would be simple. Upon entering the interrupt subroutine, all that is needed is an address in REGISTER 0 and a return to the main routine. However, the odd timing of the 29 machine cycles before the first burst of DMA must be dealt with, and also the microprocessor instruction and DMA timing must be synchronized.

VIDEO SYNCHRONIZATION

To properly use the video generator, the microprocessor timing must be synchronized with the video generator timing to avoid "jitter" or tearing of the image on the TV screen. Most instructions take two machine cycles for execution. The clock that drives the cycling of the microprocessor also drives the video generator. The synchronization of both microprocessor and video generator is on even cycles. DMA bursts only occur when a current instruction finishes execution, at that time the DMA proceeds, and the next instruction waits until the DMA burst is over. As mentioned, DMA lasts 8

cycles and the time between bursts is 6 cycles. If no three cycle instructions (such as a long branch) occur in the program anywhere, one doesn't have to worry about the synchronization since, in this case, it would occur naturally. However, an important fact which hasn't been yet mentioned enters the situation—the interrupt operation itself is only one cycle in duration and one cycle must be "made up" to keep in even synchronization. This is the reason for the odd number of cycles until the first DMA burst after the interrupt.

The timing of 29 cycles from the interrupt until DMA and 6 cycles between DMA is due to the requirements for the TV circuits and is outside of the scope of this article.

This completes the explanation of the overhead functions of the subroutine. Now we can get into its operation. As mentioned, 29 machine cycles elapse before the first burst of DMA. We shall see how this timing is accounted for, how the display pointer is handled, and how to make the final exit from the routine. The instructions from locations 0011 through 0014 account for 8 cycles. The NOP instructions in locations 0015, 0016 and 0017 add 9 more cycles for a total so far of 17 cycles. In addition the NOP's provide the odd cycle re-synchronization required.

In locations 0018 to 001D, the refresh pointer gets loaded with the beginning location of the data to be displayed. The LDI instructions in 0018 and 001B load zero to D and the PHI and PLO load the zero values into REGISTER 0. These instructions add some synchronization (8 cycles). The value of zero is still left in D and the next instruction redundantly gets zero to D from the low end of REGISTER 0. The redundancy is for the first time through this instruction only because this location (001E) is looped through a total of 32 times. The other 31 times the value that the GLO instruction gets into D is actually the address of the next line of data for the display. The reason for getting the address in this manner is to have it available in D to keep on resetting it to REGISTER 0 so the current line of data gets pointed to four times. As you will recall, at each burst of DMA, REGISTER 0 gets incremented by 8, thus resetting it in this manner overrides the automatic incrementing and the goal of displaying each string of data four times is achieved.

The next instruction in location 001F is the SEX instruction to REGISTER 2. REGISTER 2 is already the X pointer REGISTER so this instruction is redundant. It is used for the same reason as the timing NOP's: it provides 2 machine cycles as required from the interrupt. The instructions in location 001E and 001F are doing double duty as far as timing is concerned, they also are part of the 6 cycle timing required between DMA bursts for the other loop-throughs until the end of the video refresh period.

Referring to Figure 3, note the DMA notation right after location 001F. This is when the first burst of DMA occurs in the first time through the loop. As previously stated, the microprocessor actually stops executing the subroutine at this point and becomes busy with DMA for 8 Machine cycles. Next, the instruction in location 0020 is executed. Again, a SEX instruction is executed, but for timing only. Location 0021 then executes a decrement REGISTER 0 instruction. At first glance this may seem peculiar, but it is needed for more than timing (2 cycles). It is actually needed only for the last loop-through where REGISTER 0 will have been incremented to point to the beginning of the next page of memory.

Decrementing by one restores the pointer to the current page and the next instruction in location 0022 corrects the data location. The prior GLO in location 0015 provides the the current data address in D and the PLO in location 0022 takes the D value and puts it into REGISTER 0 again. The instructions in location 0023, 0024, and 0025 repeat the activity of the locations 0020, 0021, and 0022 just as the instructions in location 0026, 0027 and 0028 do. When we reach the DMA after location 0028, it is the fourth DMA and a string of eight memory data bytes has been displayed four times. Now in location 0029, a test of the refresh status

occurs. The instruction is a short branch if the status line 1 is zero, which it will be in the middle of the refresh period. This instruction takes 2 cycles. The branch is taken to the GLO instruction at location 001E which provides the address of the next bunch of data into D and adds 4 cycles, making, with the BN1 isnstuction at location 0029, six cycles to the next DMA burst. The process, as you can see, will proceed over and over until the test for status line 1 equal-to-zero fails and the short branch instruction at location 002B is executed. Thus, the refresh is over, the short branch is taken to location 000F where the LDXA instruction resides and the exit is made. This concludes the subroutine analysis.

The next VIDEO GRAPHICS SOFTWARE section in QUESTDATA will discuss the main program and some improvements/changes that you can make.

NEW GRAPHICS PROGRAM BOOKLET IS PUBLISHED BY PAUL C. MOEWS

There are a lot of interesting things you can do with graphics with your basic 256 byte Super Elf or Elf II. This booklet shows you how to make your Elf into a kaleidoscope pattern generator and tickertape-like display system. You can also have a horse race on your video screen. The final two programs are a TV stop watch and a TV clock; the clock shows hours, minutes, and seconds and is completely setable. The booklet contains a discussion of graphic interrupt routines along with the eight programs.

This exciting new booklet can be yours for only $3.00 and 50 cents extra for postage and handling. Please make checks payable to Quest Electornics.

BUG SQUASHER

The Close Encounters Theme in QUESTDATA Number 2 does not repeat itself. Mr. James Nicholson's original listing sent to us was correct and did not include the erroneous 30 AC at location CA. Substitution of the data 00 at location CA will correct this bug. Mr. Nicholson has our sincere apology for this blunder. While we are on the subject of music programs, we should mention that the February Popular Electronics 1978 music program will not work as written under Super Elf MONITOR control. What happens is that the registers used by the program and the MONITOR's PC register conflict... hence, all you will hear is one note when you attempt to run the program under MONITOR control. The program uses R3 for a data register; this conflicts with the RCA recommended use of R3 as the program counter. Changing the program so that it uses some other unused register for data storage will solve the problem. For example, Change locations A5, A6, A7 to BF, 2F, 9F. All Quest monitors set R3 as the PC when executing programs. If you 30 66 in locations 00 and 01, and press reset, run; the music program will run since the MONITOR-program register conflict is avoided altogether. Either solution to the problem will work.

Reader Sam Schmidt, an Ontario Canada COSMAC fan, writes: "I have found a small bug in the Video Graphics Software Part I. In locations 40 and on is the data for the words 'Super Elf.' When I ran it with my Super Elf I go an open U and an F for the E. In locations 78 to 7F the data reads; F0, 88, 80 F0, 90, F0, 0F, 08, is should be F0, F8, 80, F0, 90, 0F, 0F, 08. I hope I have been a help." You have been a help, Sam. Quesdata thanks you.
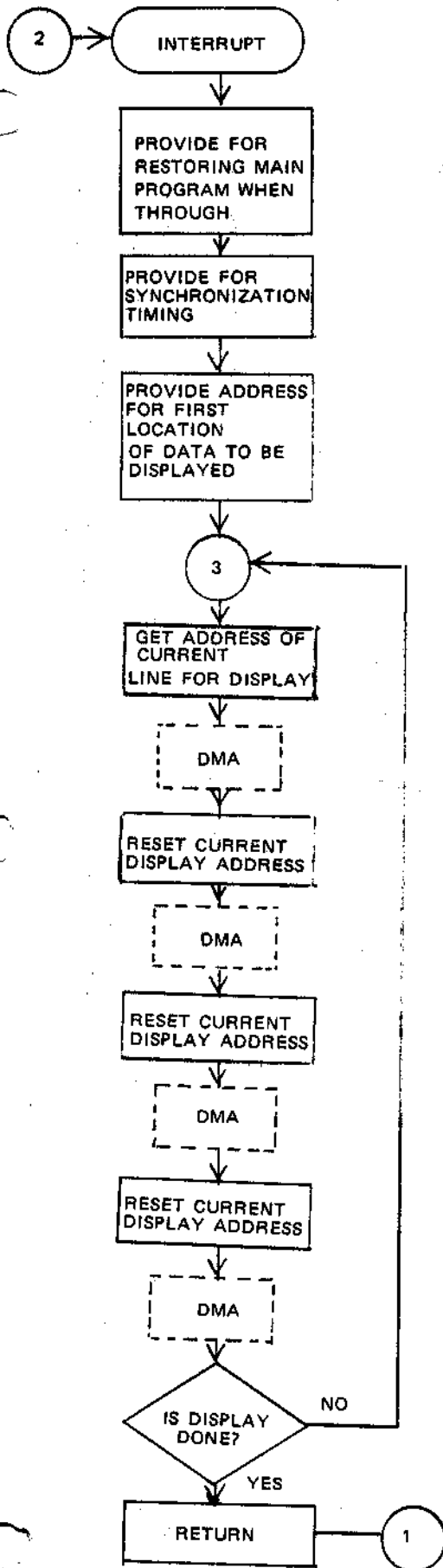
INTERRUPT SUBROUTINE

MAIN PROGRAM

FIGURE 2

FIGURE 1

VIDEO GRAPHICS
FLOWCHARTS

Video Graphics Flowcharts — Figure 1 (Main Program) and Figure 2 (Interrupt Subroutine)

Figure 2 (Interrupt Subroutine):

- (2) → INTERRUPT
- PROVIDE FOR RESTORING MAIN PROGRAM WHEN THROUGH
- PROVIDE FOR SYNCHRONIZATION TIMING
- PROVIDE ADDRESS FOR FIRST LOCATION OF DATA TO BE DISPLAYED
- (3)
- GET ADDRESS OF CURRENT LINE FOR DISPLAY
- DMA
- RESET CURRENT DISPLAY ADDRESS
- DMA
- RESET CURRENT DISPLAY ADDRESS
- DMA
- RESET CURRENT DISPLAY ADDRESS
- DMA
- IS DISPLAY DONE? — NO → (3) / YES
- RETURN → (1)

Figure 1 (Main Program):

- PROVIDE BEGINNING ADDRESS VALUES FOR THE OPERATING REGISTERS
- RESET THE PROGRAM COUNTER AND BEGIN MAIN PROGRAM
- (4)
- CONDITION FOR I/O
- TURN ON VIDEO
- (5)
- IN DEPRESSED ? — NO → (5) / YES
- GET STARTING ADDRESS FOR DATA LOADING FROM HEX KEYBOARD — — — STORES DATA INTO STACK
- LOAD ADDRESS INTO DATA POINTER
- (6)
- IN DEPRESSED ? — NO / YES → (7)
- (7)
- IN DEPRESSED ? — NO / YES → TO NEXT COLUMN

CONTINUED FROM PREVIOUS COLUMN

- GET DATA FROM HEX KEYBOARD — — — STORE DATA IN STACK
- MORE DATA INTO CURRENT POINTER LOCATION
- INCREMENT DATA POINTER → (6)

FIGURE 3   VIDEO DISPLAY PROGRAM LISTING

LABEL	LOCATION	CODE	DATA/BR	MNEMONIC/REF	COMMENTS	FUNCTION
START	0000	90		GHI (0.1)	R1.1, R2.1=00	PROVIDE BEGINNING
	0001	B1		PHI (1.1)		ADDRESS VALUES
	0002	B2		PHI (2.1)		FOR THE REGISTER
	0003	B3		PHI (3.1)		AS REQUIRED FOR
	0004	B4		PHI (4.1)		VIDEO HARDWARE IN
	0005	F8	2D	LDI	R3.0=(MAIN)	THE SUPER ELF
	0007	A3		PLO (3.0)		
	0008	F8	3F	LDI	R2.0=(STACK)	
	000A	A2		PLO (2.0)		
	000B	F8	11	LDI	R1.0=(INTERRUPT)	
	000D	A1		PLO (1.0)		
① 000E	000E	D3		SEP 3	P=(GOTO MAIN)	NOTE: NEXT ADDRESS IS 002D
RETURN	000F	72		LDXA	restore D, R2+1	PROVIDE FOR RESTORATION OF THE MPU (i.e. THE D, X AND P REGISTERS) SO THAT THE MAIN ROUTINE WILL
	0010	70		RET	restore XP, R2+1	CONTINUE OPERATION WHERE IT LEFT OFF.
② INT.	0011	22		DEC (2)	R2-1	PROVIDES A METHOD TO SAVE THE D, X AND P REGISTERS TO RESTORE ORIGINAL VALUES LATER
	0012	78		SAV	save XP, @ M2	(SEE ABOVE)
	0013	22		DEC (2)	R2-1	
	0014	52		STR (2)	save D@ M2	
	0015	C4		NOP	no-op delay (9 cycles)	USE MPU CYCLES REQUIRED FOR KEEPING IN TIME SYNCHRONIZATION WITH VIDEO HARDWARE
	0016	C4		NOP		
	0017	C4		NOP		
	0018	F8	00	LDI	re-initialize ptr high	PROVIDE ADDRESS FOR FIRST LOCATION OF DATA TO BE DISPLAYED
	001A	B0		PHI (1.1)	R0=0000 (refresh ptr)	
	001B	F8	00	LDI	re-initialize ptr low	
③ REFRESH	001D	A0		PLO (0.0)	D=R0.0	
	001E	80		GLO (0.0)	8 DMA cycles (R0+8)	
	001F	E2		SEX (2)		MAKE AVAILABLE THE CURRENT ADDRESS OF THE NEXT GROUP OF DATA TO BE DISPLAYED
	0020	E2		SEX (2)		THIS IS ACTUALLY A NO-OP USED FOR THE SAME REASON AS THE NOPS IN LOCATION 0015, 0016 AND 0017
	0021	20		DEC (0)	R0-1	HERE OCCURS A BURST OF DMA WHICH PROVIDES THE DATA APPEARING ON THE TV SCREEN. AT EACH BURST, ONE LINE IS WRITTEN ON THE SCREEN. THIS BURST ACTUALLY STOPS THE MPU FOR 8 x 4.47 μ SEC.
	0022	A0		PLO (0.0)	8 DMA cycles (R0+8)	THIS SECTION RESETS THE CURRENT ADDRESS (R0+8) OF DATA TO BE DISPLAYED. THE REST
	0023	E2		SEX (2)	R0-1	OCCURS THREE TIMES SO THAT EACH GROUP OF DATA IS DISPLAYED FOUR TIMES.
	0024	20		DEC (0)		
	0025	A0		PLO (0.0)	R0-1	
	0026	E2		SEX (2)	R0.0=D	
	0027	20		DEC (0)	R0-1	
	0028	A0		PLO (0.0)		
	0029	3C	1E	BNI ③	go to refresh (EF1=0)	THE SEX COMMANDS ARE SIMPLY NOP'S TO CONTROL TIMING SYNCH. THE DEC'S CONTINUOUSLY ASSURE THAT THE DATA WILL ALWAYS BE OBTAINED FROM THE CURRENT PAGE OF MEMORY. THE PLO'S CONTINUOUSLY PROVIDE REGISTER 0 WITH THE CURRENT ADDRESS OF THE GROUP OF DATA BEING DISPLAYED FOUR TIMES. THE ADDRESS MADE AVAILABLE AT LOCATION 001E.
•						HERE A CHECK ON THE VIDEO HARDWARE STATUS LINE (EF1) IS MADE. IF STILL '0', THEN THE TV SCREEN DOES NOT HAVE THE FULL SET OF DATA DISPLAYED YET. THEREFORE, GO BACK TO LOCATION 001E AND REPEAT THE STEPS FROM LOCATION 001E THROUGH 0028. THE EXECUTION THROUGH THIS SECTION IS MADE 32 TIMES TOTALLY.
④	002B	30	0F	BR ①	go to return (EF1=1)	NOW THE VIDEO DISPLAY IS COMPLETE SO THE RETURN TO THE MAIN PROGRAM IS MADE.
⑤	002D	E2		SEX (2)	X=2	SET UP FOR INPUT/OUTPUT (I/O) AND OUTPUT COMMAND TO TURN ON VIDEO HARDWARE.
	002E	69		INP 1	turn TV on	
⑤	002F	3F	2F	BN4 ⑤	wait for IN pressed	THE PROGRAM STAYS HERE UNTIL THE I BUTTON IS DEPRESSED. NOTE: ONCE THE INP1 (69) COMMAND IS SENT, THE VIDEO CIRCUIT STARTS TO INTERRUPT HERE AND THE REST OF THE PROGRAM AT THE RATE OF 61 TIMES PER SECOND.
MAIN	0031	6C		INP4	set MX, D, R4.0=keypad	WHATEVER WAS KEYED INTO THE HEX KEYBOARD IS HERE READ-IN AND USED FOR LOCATING THE START OF WHERE DATA IS TO BE SUBSEQUENTLY ENTERED INTO MEMORY FROM THE HEX KEYBOARD
	0032	A4		PLO (4.0)		
⑥ 0033	0033	37	33	B4 ⑥	wait for IN released	THE PROGRAM WAITS FOR THE IN BUTTON TO BE RELEASED AFTER THE PRIOR DEPRESSION.
⑦	0035	3F	35	BN4 ⑦	wait for IN pressed	THE PROGRAM WAITS FOR THE IN BUTTON TO BE PRESSED AGAIN.
	0037	6C		INP4	Set MX, D=keypad	NOW THE DATA INPUT FROM THE KEYBOARD IS READ IN.
	0038	54		STR (4)	set M4=D, R4+1	THE DATA READ IN IS NOW TRANSFERRED INTO MEMORY LOCATED BY THE PREVIOUSLY INPUTTED ADDRESS WHICH IS CONTINUOUSLY ADVANCED BY ONE EVERY TIME THE I BUTTON IS DEPRESSED/RELEASED. THEREFORE THIS
	0039	14		INC (4)	R4+1	SECTION OF THE PROGRAM ALLOWS WRITING INTO MEMORY WHILE MEMORY IS BEING DISPLAYED ON THE TV SCREEN, CONTINUOUSLY AN ENTRY AT A TIME, STARTING FROM THE LOCATION FIRST ENTERED ABOVE. THE
⑥	003A	30	33	BR ⑥	go to M33	PROGRAM FROM NOW ON STAYS IN THIS SECTION FROM LOCATION 0033 TO 0038.
	003C	NOT USED				THESE LOCATIONS ARE RESERVED FOR THE STACK USED IN EVERY 6C COMMAND AND IN THE INTERRUPT
	003D					SUBROUTINE.
⑧ STACK	003E					

[Answer to L.A.P.D. stoplight project, page 5]

Here is one possible way to make a flashing light for the City of Los Angeles. Note that the delay is made into one operation. We can do this since we have studied its inner working on page 4 and 5. This is how a systems analyst might block out the delay operation. Using the tool of flowcharting, given by James C. Nicholson (given on pg. 2 and 3), you are prepared to explode this block into its component parts. The next WHAT THE MACHINE IS THINKING column will cover memory organization and pointer registers.

LOC.	CODE	MNEM.
00	78	SEQ
01	F8	LDI
02	FF	
03	A9	PLO
04	B9	PHI
05	29	DEC
06	99	GHI
07	3A	BNZ
08	05	
09	39	BNQ
0A	00	
0B	7A	REQ
0C	30	BR
0D	01	
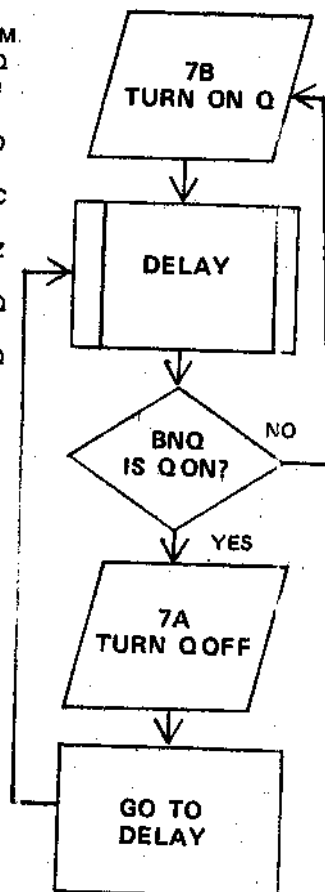
POCKET CARD LISTS ALL COSMAC CODE

HEX-ASCII TABLE

00	NUL	21	!	42	B	63	c	
01	SOH	22	"	43	C	64	d	
02	STX	23	#	44	D	65	e	
03	ETX	24	$	45	E	66	f	
04	EOT	25	%	46	F	67	g	
05	ENQ	26	&	47	G	68	h	
06	ACK	27	'	48	H	69	i	
07	BEL	28	(	49	I	6A	j	
08	BS	29	)	4A	J	6B	k	
09	HT	2A	*	4B	K	6C	l	
0A	LF	2B	+	4C	L	6D	m	
0B	VT	2C	,	4D	M	6E	n	
0C	FF	2D	-	4E	N	6F	o	
0D	CR	2E	.	4F	O	70	p	
0E	SO	2F	/	50	P	71	q	
0F	SI	30	0	51	Q	72	r	
10	DLE	31	1	52	R	73	s	
11	DC1	32	2	53	S	74	t	
12	DC2	33	3	54	T	75	u	
13	DC3	34	4	55	U	76	v	
14	DC4	35	5	56	V	77	w	
15	NAK	36	6	57	W	78	x	
16	SYN	37	7	58	X	79	y	
17	ETB	38	8	59	Y	7A	z	
18	CAN	39	9	5A	Z	7B	{	
19	EM	3A	:	5B	[	7C		
1A	SUB	3B	;	5C	\	7D	}	
1B	ESC	3C	<	5D	]		(ALT MODE)	
1C	FS	3D	=	5E	^	7E	(')	
1D	GS	3E	>	5F	_	7F	(←)	
1E	RS	3F	?	60	`		DEL (RUB OUT)	
1F	US	40	@	61	a			
20	SP	41	A	62	b			

1800 CMOS
### MICROPROCESSOR POCKET GUIDE

FEATURES

- Instruction Cycle Time 2.5 - 3.75 us at 6.4 MHz.
- 8 Bit Parallel Data Organization
- Memory Addressing to 65,536 Bytes
- On Chip Direct Memory Access
- 16 x 16 General Purpose Register Matrix
- Direct Memory to Peripheral Transfer on I/O Instructions
- TTL Compatible
- Single Phase Clock, Single Voltage Supply
- 91 Instructions

PIN CONFIGURATION
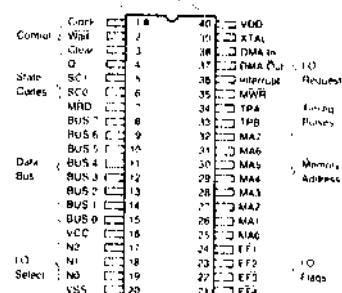
This shirt pocket data card gives all the COSMAC microcomputer codes, functional diagrams, and information you need to get your computer to obey your every command. The card gives you the convienience of having the mnemonics and operation codes at your fingertips. To have this card for your COSMAC, send 50 cents to cover postage and handling to QUEST Electronics. No longer will you have to thumb through bulky data books in search of the code to feed your hungry 1802.

HOW TO SUBMIT YOUR PROGRAMMING MASTERPIECE TO QUESTDATA

Your readership response to QUESTDATA has been fantastic. Keep the interesting applications and programs coming. When sending QUESTDATA your programming gem; document your masterpiece with as much information as you can think up. It is important for readers to see a flowchart so they will know how your program was done and be able to modify it, if they so wish. Tell us the story of how you thought up your program. Relate all of its capabilities with as many possible applications as you can. If at all possible type your text and double-space it on 8½ x 11 white paper. It is wise to place your name and address in the upper right corner of all sheets submitted.

QUESTDATA will also be publishing a letters column in the future. What kinds of things would you like to see your 1802 perform? Send us (in letter or article form) your ideas and daydreams that you have about the 1802. Certain areas that already appear to have high reader interest are: Music, games, video graphics, programming tricks and math subroutines. Modificataions and improvements to existing programs are welcome. So—keep the imaginative work and interesting ideas flowing.

COSMAC CLUB COSMAC CLUB COSMAC CLUB COSMAC CLUB COSMAC CLUB COSMAC CLUB COSMAC CLUB

3