14.

I hope that you've had good luck creating the program to take an ASCII message and translate it into a series of subroutine calls, calls that would, once the final beeping routines are created, transmit Morse Code.

Just to review, you'll remember that the structure of the program was set up to read an ASCII message, character by character. It would then locate an encoded version of the Morse Code from an in-memory table, and use that information to produce a pattern of dits and dahs. The program you've created up to this point should look something like the one I have for you next.

> Program #23, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

Let me take you through my program. I know that without real person-to-person interaction the things I've done and the things you've done won't match. You might feel like the work you have just finished is somewhat in vain. Not true. This is really the first program I've left you alone to structure, and it's invaluable that you contrast the two.

Last session I told you that some characters which didn't exist in the traditional Morse Code set might be ideal for using as end-of-message markers, to tell your program that the message was complete, and should either be ended or repeated. I chose to use the greater-than sign for end of message, and the less than sign for a continuous repeat of the message. Recall that the message itself would be stored beginning at **$3000**. As for the program, I suggested you put that at **$2E00**, leaving plenty of room for the program

```
Keep in mind that as this
program is developed,
translating codes and producing
sound is the object. Whether it
is Morse Code or any code isn't
critical, and whether it's a
beep or an entire musical tune
isn't the point. You are
finding out how to manipulate
sound and make translations.
Also, remember that the final
program is going to make
adjustments for that Morse Code
punctuation which is six beeps
long; these are exceptions which
don't affect the heart of the
program and its inherent
simplicity.

* Name the timing considerations
needed for Morse Code.

The length of the "dit", the
length of the "dah", the length
of silences, and the frequency
of the beep.

* The letter Y is
dah-dit-dah-dah. What is its
length in binary?

Its length is four, binary 100.

* The letter Y is
dah-dit-dah-dah. What is its
binary beep pattern?

Its beep pattern is 1011.
```

* What is the complete encoded value for Y, pattern dah-dit-dah-dah?

Beep pattern 1011 plus one unused bit (0) plus a length of 100. The result is 10110100.

* What is 10110100 in hexadecimal?

10110100 is hexadecimal $B4.

* If the A accumulator contains $B4, and you execute ANDA #$07, what is the result?

The result is $04.

* If the B accumulator contains $B4, and you execute ROLB four times, what is the condition of the carry flag after each ROLB?

C=1, C=0, C=1, C=1.

* How long is a "dah" with respect to a "dit"?

Three times as long.

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes it current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* What is the proper term for turning off an interrupt?

Masking the interrupt.

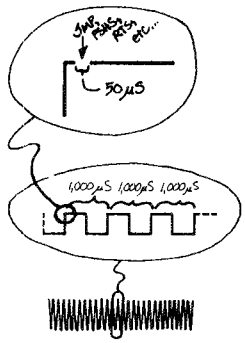
* What condition code bits determine whether interrupts are masked or enabled?

Bits 4 and 6.

and the Morse Code lookup table. I put the table at $2F00.

The program itself turns out to be surprisingly simple. There are three options for producing dits and dahs shown in my listing. The first option best represents the actual, expected circumstances — a dah is three times the length of a dit. Therefore, two separate subroutines, one dit and one dah, are created for this purpose. The other solutions might not be immediately obvious.

To understand the second option, consider that the real world we're dealing with here is lots slower than the computer world. In this case, the computer operations involved in determining the difference between dits and dahs, and the time required to call the dit subroutine, are minuscule. In fact, these operations are nearly inaudible in the course of a real-world beep. A jump to subroutine (JSR) and a return form subroutine (RTS) take only 13 clock cycles, and the two likely PUSH/PULL combinations used to save information before performing the beep subroutine itself add 28 more cycles. The total clock cycles, 41, demand under 50 microseconds for completion. Recall that I suggested a beep frequency of 1,000 Hz. That frequency means that each pulse that makes up this beep frequency is 1,000 microseconds — so these additional jumps and returns add only about 5 percent to the time it takes to create one single pulse of the beep. So you can see that in a case where the computer is much faster than a real-world event, alternative approaches such as this can simplify the actual machine code you must write.

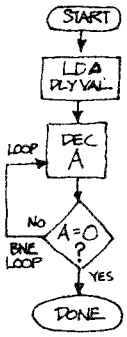


The third method is a favorite of programmers because it allows a subroutine to be an all-purpose building block. In this method, a value is given to the accumulator, a value which indicates a dit or a dah. The beeping routine then uses this value to calculate the overall loop length of dit or dah. In a different and more precise way, this subroutine performs s similarly efficient function to the previous one.

More on all of these when the actual beep-creation routines are assembled.

First, I'd like to turn to the problem of creating the beep itself. What is a beep? A beep is a tone, or a pitch — a rapid, consistent and regular fluctuation of air molecules. This isn't a lesson on acoustics, so I'll make it short. A rapid, consistent and regular compression and decompression of the air is perceived as a tone or pitch. A loudspeaker which is pushed forward and back rapidly, consistently and regularly will compress and decompress the air in a similar way. Electrical impulses which alternate between two voltage levels can create the necessary speaker motion. A computer program can provide those impulses.

So that reveals the structure of the computer program, and also gets us — the long way 'round — to the question of time

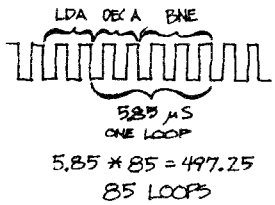


delays. Alternating between a one and a zero is a simple task, something you've done already. The task of the beeping program is to alternate between one and zero at a predictable rate — in this case, 1,000 Hz, or 1,000 alternations per second.

A simple delay loop in machine language might look like this (and you can follow along with me in the book):

```
        LDA     DLYVAL
LOOP    DECA
        BNE     LOOP
```

The A accumulator contains the delay value. The BNE instruction loops back until A equals zero. This simple delay allows a maximum loop of 256 iterations — the largest 8-bit number the A accumulator can hold. According to the MC6809E data book, LDA immediate requires 2 clock cycles to complete, DECA takes 2 clock cycles, and BNE needs 3 cycles. The goal of the delay is 500 microseconds total. LDA only happens once, so that leaves about 497 microseconds to go. The DECA/BNE combination of 5 clock cycles is 5.85 microseconds on the Color Computer, meaning a total of 85 loops (497.25 microseconds) does the trick. The value for label DELAY, then, is 85 decimal. We won't fix this in concrete yet, though, because certain bits and pieces of the program that might add extra delay to the process haven't been written yet. But it's working delay information for now.



You might have picked up on my delay of 500 microseconds. If the beep is 1,000 Hz, then a complete pulse is 1000 microseconds. A complete pulse. That means one pulse up for air compression, and one pulse down for air decompression . . . a total of 1,000 microseconds, 500 microseconds for the up pulse, 500 microseconds for the down pulse.

Now how about the length of the beep? Eventually that's going to vary, too, but for the moment, let's make a dit one-fifth of a second, and a dah three-fifths of a second. Since the beep is 1,000 Hz, or 1,000 pulses per second, then one-fifth of a second is just 200 pulses. So the program begins to look like this (again, follow me in the book):

```
        LDB     200     * LENGTH
OUTLP   LDA     85      * FREQUENCY
INLP1   DECA            * DONE YET?
        BNE     INLP1   * WAIT
        LDA     85      * FREQUENCY
INLP2   DECA            * DONE YET?
        BNE     INLP2   * WAIT
        DECB            * BEEP END?
        BNE     OUTLP   * MORE
```

There's still no actual beeping going on here because I haven't described how to do it. For this you need to recall the detailed memory map presented several lessons

* What command specifically masks out (turns off) both interrupts?

ORCC #$50 (binary 01010000).

* What command specifically enables (turns on) both interrupts?

ANDCC #$AF (binary 10101111).

* What is the clock speed of the Color Computer?

.89 MHz (894,886 clock pulses per second).

* What does Hz mean?

Hz means Hertz, or cycles (pulses) per second.

* What does MHz mean?

MHz means megaHertz, or million cycles per second.

* Why does sound output require timing?

Because sound is made up of specific frequencies, and frequencies are inherently time-based.

* How long is the shortest BASIC beep (using SOUND X,1)?

Approximately 1/14 of a second.

* If a loop contains two load A immediates, two store A extendeds, and one branch to make a complete loop, how many clock cycles are required for this loop?

2 times 2 cycles, plus 2 times 5 cycles, plus 3 cycles ... a total of 17 cycles.

* At 894,886 clock cycles per second, how many loops is this?

894,886 divided by 17, or 52,640 cycles.

* At 894,996 clock cycles per second, how many clock cycles are available to produce a 1,000-Hz beep?

894,996 divided by 1,000, or about 895 clock cycles.

* If a beep frequency is 1,000 Hz, how long is each "pulse" of sound.

Each pulse is 1/1000th of a second (1 millisecond or 1,000 microseconds).

* How long is one Color Computer clock cycle?

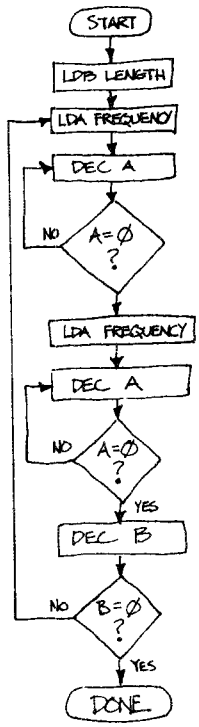
Approximately 1.11746 microseconds.

* How many clock cycles pass by in 1,000 microseconds?

Approximately 895.

earlier. You can turn back to that later; for the moment I'll tell you that "single bit sound," that is, sound produced by pulsing on and off one bit in a memory location, is found at address **$FF22**. Alternations of one and zero made at bit one of location **$FF22** will be heard on the television speaker or the cassette output.

So once again you discover that the subroutine becomes surprisingly simple in its final form. It's coming up next. After you've got the program loaded, take some time to look at it, then come back to the tape.



Program #24, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
2E00                00100          ORG     $2E00
                    00110 *
2E00 1A   50        00120 BEGIN    ORCC    #$50     * TURN OFF BOTH INTERRUPTS
                    00130 *
2E02 86   32        00140          LDA     #$32     * 0011 0010 SETS DD REGISTER
2E04 B7   FF23      00150          STA     $FF23    * ACCESS DATA DIR. REGISTER
2E07 86   FA        00160          LDA     #$FA     * 1111 1010 SETS S.B. OUTPUT
2E09 B7   FF22      00170          STA     $FF22    * TURN ON SINGLE BIT SOUND
2E0C 86   36        00180          LDA     #$36     * 0011 0110 SETS PD I/O
2E0E B7   FF23      00190          STA     $FF23    * RESTORE I/O CONFIGURATION
                    00200 *
2E11 C6   64        00210 BEEP     LDB     #100     * GET OUTSIDE BEEPWAVE VALUE
2E13 86   28        00220 OUTLP    LDA     #40      * GET INSIDE BEEPWAVE LENGTH
2E15 4A             00230 INLP1    DECA             * DECREMENT INSIDE DELAY
2E16 26   FD        00240          BNE     INLP1    * AND WAIT TO TOTAL LOOPS
2E18 86   02        00250          LDA     #$02     * GET HIGH PART OF BEEPWAVE
2E1A BA   FF22      00260          ORA     $FF22    * OR WITH PORT OUTPUT STATUS
2E1D B7   FF22      00270          STA     $FF22    * AND OUTPUT THE RESULT
2E20 86   28        00280          LDA     #40      * GET ANOTHER DELAY VALUE
2E22 4A             00290 INLP2    DECA             * AND COUNT DOWN THRU DELAY
2E23 26   FD        00300          BNE     INLP2    * WAIT THROUGH TOTAL LOOPS
2E25 86   FD        00310          LDA     #$FD     * SET LOW PART OF BEEPWAVE
2E27 B4   FF22      00320          ANDA    $FF22    * AND WITH CURRENT STATUS
2E2A B7   FF22      00330          STA     $FF22    * AND OUTPUT LOW BEEPWAVE
2E2D 5A             00340          DECB             * DECREMENT NUMBER OF WAVES
2E2E 26   E3        00350          BNE     OUTLP    * AND GO BACK TILL ALL DONE
2E30 39             00360          RTS              * AND BACK TO PROGRAM
          2E00      00370          END     BEGIN
00000 TOTAL ERRORS
BEEP    2E11
BEGIN   2E00
INLP1   2E15
INLP2   2E22
OUTLP   2E13
```

* What is a beep?

A tone; a pitch; a rapid, consistent and regular fluctuation of air molecules.

* What electrical device produces a beep?

A loudspeaker.

As it stands now, this program should be set up to provide a single beep. First, assemble this program in memory, at the origin shown. Type A/IM/AO, and hit ENTER. That's A/IM/AO. Now quit the editor/assembler. Type Q and hit enter.

You'll be in BASIC. To produce that beep, you have to remember the origin of the program. Type three lines:

```
10 EXEC&H2E00
20 FORX=1TO20:NEXT
30 GOTO10
```

Turn up the television volume, and RUN this program. You should hear a continuous series of beeps. As you listen, though, you won't hear that old familiar gargle in the sound. The interrupts have been turned off; the sound is pure. The routine works. What's left is to combine the beep routine, the Morse Code table lookup routine, and all the rest into a complete, usable program.

We'll be going on to complete the full program next, so it would be a good time to take a break if you wish to review. After that, load the program.

Program #25, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

* What causes the loudspeaker to produce a beep?

Electrical impulses which alternate voltage levels.

* Can a computer program provide electrical impulses?

Yes.

* How are the electrical impulses represented?

By binary ones and zeros.

* Two LDA immediates, two STA extendeds, and BNE make a 16-cycle loop; it might LDA #0 and #1 alternately and STA to an address to make the speaker produce sound. If 1,000 microseconds is 895 clock cycles, but the loop is only 16 cycles, what else is needed?

A delay.

```
3E00                00100          ORG     $3E00
                    00110 *
3E00 1A   50        00120 BEGIN    ORCC    #$50     * TURN OFF BOTH INTERRUPTS
                    00130 *
3E02 86   32        00140          LDA     #$32     * 0011 0010 SETS DD REGISTER
3E04 B7   FF23      00150          STA     $FF23    * ACCESS DATA DIR. REGISTER
3E07 86   FA        00160          LDA     #$FA     * 1111 1010 SETS S.B. OUTPUT
3E09 B7   FF22      00170          STA     $FF22    * TURN ON SINGLE BIT SOUND
3E0C 86   36        00180          LDA     #$36     * 0011 0110 SETS PD I/O
3E0E B7   FF23      00190          STA     $FF23    * RESTORE I/O CONFIGURATION
                    00200 *
3E11 8E   3E99      00210 START    LDX     #MORSE   * POINT TO MORSE CODE TABLE
3E14 108E 3ED4      00220          LDY     #TEXT    * POINT TO MESSAGE IN MEMORY
3E18 E6   A0        00230 AGAIN    LDB     ,Y+      * GET VALUE FROM THE TEXT
3E1A C1   3C        00240          CMPB    #$3C     * CHECK IF LESS-THAN SIGN
3E1C 27   F3        00250          BEQ     START    * IF "<", THEN REPEAT MESSAGE
3E1E C1   3E        00260          CMPB    #$3E     * CHECK IF GREATER-THAN SIGN
3E20 27   43        00270          BEQ     OUT      * IF ")", RETURN TO BASIC
3E22 C0   20        00280          SUBB    #$20     * STRIP OFFSET (TABLE $20-$5B)
3E24 A6   85        00290          LDA     B,X      * ENCODED MORSE FROM TABLE
3E26 27   2C        00300          BEQ     SPACE    * IF 00, THEN SEND A SILENCE
3E28 34   02        00310          PSHS    A        * SAVE ENCODED MORSE VALUE
3E2A C1   10        00320          CMPB    #$10     * CHECK IF PUNCTUATION
3E2C 23   08        00330          BLS     EXCEPT   * IF SO, GO TO EXCEPTION
3E2E C1   1A        00340          CMPB    #$1A     * CHECK IF A COLON
3E30 27   04        00350          BEQ     EXCEPT   * IF SO, GO TO EXCEPTION
3E32 C1   1F        00360          CMPB    #$1F     * CHECK IF A QUESTION MARK
3E34 26   05        00370          BNE     NORMAL   * IF NOT, PROCEED NORMALLY
3E36 84   03        00380 EXCEPT   ANDA    #$03     * EXCEPTION USES RIGHT 2 BITS
3E38 48             00390          ASLA             * LEFT SHIFT CREATES NUMBER 6
3E39 20   02        00400          BRA     GETVAL   * NOW READY FOR THE ACTION
3E3B 84   07        00410 NORMAL   ANDA    #$07     * NORMAL USES RIGHT 3 BITS
3E3D 35   04        00420 GETVAL   PULS    B        * RESTORE ENCODED MORSE VALUE
3E3F 59             00430 NEWBIT   ROLB             * ROTATE BIT TO CARRY FLAG
3E40 25   04        00440          BCS     JUMP1    * IF SET, THEN JUMP TO DAH
3E42 8D   1D        00450          BSR     DIT      * OTHERWISE SEND A DIT
3E44 20   02        00460          BRA     NEXT     * AND GO TO THE NEXT BIT
3E46 8D   15        00470 JUMP1    BSR     DAH      * THEN SEND THE DAH
3E48 4A             00480 NEXT     DECA             * SEE IF DONE WITH ALL BEEPS
3E49 26   F4        00490          BNE     NEWBIT   * IF NOT, THEN GET ANOTHER
3E4B C6   04        00500          LDB     #$04     * OTHERWISE GET TIMING VALUE
3E4D 8D   3B        00510 LETRLP   BSR     QUIET    * AND CALL INTER-LETTER PAUSE
3E4F 5A             00520          DECB             * DECREMENT PAUSE COUNTER
3E50 26   FB        00530          BNE     LETRLP   * AND LOOP BACK TILL DONE
```

Program #25

```
3E52 20   C4       00540          BRA     AGAIN   * THEN GO BACK FOR MORE TEXT
                   00550 *
3E54 C6   08       00560 SPACE    LDB     #$08    * 8 SILENCES FOR A SPACE
3E56 8D   32       00570 SPCLP    BSR     QUIET   * AND GO SEND THE SILENCE
3E58 5A            00580          DECB            * DECREMENT THE SILENCE COUNT
3E59 26   FB       00590          BNE     SPCLP   * AND LOOP BACK TILL DONE
3E5B 20   BB       00600          BRA     AGAIN   * AND GO GET NEXT TEXT
                   00610 *
3E5D 8D   07       00620 DAH      BSR     BEEP    * PERFORM FIRST 1/3 BEEP
3E5F 8D   05       00630          BSR     BEEP    * PERFORM SECOND 1/3 BEEP
3E61 8D   03       00640 DIT      BSR     BEEP    * PERFORM DIT OR LAST 1/3
3E63 8D   25       00650          BSR     QUIET   * AND PUT IN A SILENCE
3E65 39            00660 OUT      RTS             * BACK TO PROGRAM (OR BASIC)
                   00670 *
3E66 34   06       00680 BEEP     PSHS    A,B     * SAVE COUNT AND MORSE CODE
3E68 C6   64       00690          LDB     #100    * GET OUTSIDE BEEPWAVE VALUE
3E6A 86   28       00700 OUTLP    LDA     #40     * GET INSIDE BEEPWAVE LENGTH
3E6C 4A            00710 INLP1    DECA            * DECREMENT INSIDE DELAY
3E6D 26   FD       00720          BNE     INLP1   * AND WAIT TO TOTAL LOOPS
3E6F 86   02       00730          LDA     #$02    * GET HIGH PART OF BEEPWAVE
3E71 BA   FF22     00740          ORA     $FF22   * OR WITH PORT OUTPUT STATUS
3E74 B7   FF22     00750          STA     $FF22   * AND OUTPUT THE RESULT
3E77 86   28       00760          LDA     #40     * GET ANOTHER DELAY VALUE
3E79 4A            00770 INLP2    DECA            * AND COUNT DOWN THRU DELAY
3E7A 26   FD       00780          BNE     INLP2   * WAIT THROUGH TOTAL LOOPS
3E7C 86   FD       00790          LDA     #$FD    * SET LOW PART OF BEEPWAVE
3E7E B4   FF22     00800          ANDA    $FF22   * AND WITH CURRENT STATUS
3E81 B7   FF22     00810          STA     $FF22   * AND OUTPUT LOW BEEPWAVE
3E84 5A            00820          DECB            * DECREMENT NUMBER OF WAVES
3E85 26   E3       00830          BNE     OUTLP   * AND GO BACK TILL ALL DONE
3E87 35   06       00840          PULS    A,B     * RESTORE COUNTER AND MORSE
3E89 39            00850          RTS             * AND BACK TO PROGRAM
                   00860 *
3E8A 34   06       00870 QUIET    PSHS    A,B     * SAVE COUNTER AND MORSE
3E8C C6   64       00880          LDB     #100    * GET OUTSIDE DELAY VALUE
3E8E 86   64       00890 QLP1     LDA     #100    * GET INSIDE DELAY VALUE
3E90 4A            00900 QLP2     DECA            * COUNT DOWN THRU INNER LOOP
3E91 26   FD       00910          BNE     QLP2    * AND WAIT FOR THE COUNT
3E93 5A            00920          DECB            * COUNT DOWN THRU OUTER LOOP
3E94 26   F8       00930          BNE     QLP1    * AND WAIT FOR THE COUNT
3E96 35   06       00940          PULS    A,B     * RESTORE COUNTER AND MORSE
3E98 39            00950          RTS             * BACK TO MAIN PROGRAM
                   00960 *
3E99      00       00970 MORSE    FCB     $00     * SPACE
3E9A      00       00980          FCB     $00     * ! = SPACE
3E9B      4B       00990          FCB     $4B     * " (.-..-.) (010010 11) **
3E9C      00       01000          FCB     $00     * # = SPACE
3E9D      00       01010          FCB     $00     * $ = SPACE
3E9E      00       01020          FCB     $00     * % = SPACE
3E9F      00       01030          FCB     $00     * & = SPACE
3EA0      7B       01040          FCB     $7B     * ' (.----.) (011110 11) **
3EA1      B7       01050          FCB     $B7     * ( (-.--.-) (101101 11) **
3EA2      B7       01060          FCB     $B7     * ) (-.--.-) (101101 11) **
3EA3      00       01070          FCB     $00     * * = SPACE
3EA4      00       01080          FCB     $00     * + = SPACE
3EA5      CF       01090          FCB     $CF     * , (--..--) (110011 11) **
3EA6      87       01100          FCB     $87     * - (-....-) (100001 11) **
3EA7      57       01110          FCB     $57     * . (.-.-.-) (010101 11) **
3EA8      93       01120          FCB     $93     * / (-..-.) (100100 11) **
3EA9      FD       01130          FCB     $FD     * 0 (-----) (11111 101)
3EAA      7D       01140          FCB     $7D     * 1 (.----) (01111 101)
3EAB      3D       01150          FCB     $3D     * 2 (..---) (00111 101)
3EAC      1D       01160          FCB     $1D     * 3 (...--) (00011 101)
3EAD      0D       01170          FCB     $0D     * 4 (....-) (00001 101)
3EAE      05       01180          FCB     $05     * 5 (.....) (00000 101)
3EAF      85       01190          FCB     $85     * 6 (-....) (10000 101)
3EB0      C5       01200          FCB     $C5     * 7 (--...) (11000 101)
3EB1      E5       01210          FCB     $E5     * 8 (---..) (11100 101)
3EB2      F5       01220          FCB     $F5     * 9 (----.) (11110 101)
3EB3      E3       01230          FCB     $E3     * : (---...) (111000 11) **
3EB4      00       01240          FCB     $00     * ; = SPACE
3EB5      00       01250          FCB     $00     * < = SPACE
3EB6      00       01260          FCB     $00     * = = SPACE
3EB7      00       01270          FCB     $00     * > = SPACE
3EB8      33       01280          FCB     $33     * ? (..--..) (001100 11) **
3EB9      00       01290          FCB     $00     * @ = SPACE
3EBA      41       01300          FCB     $41     * A (.-) (01 000 001)
3EBB      84       01310          FCB     $84     * B (-...) (1000 0 100)
3EBC      A4       01320          FCB     $A4     * C (-.-.) (1010 0 100)
3EBD      83       01330          FCB     $83     * D (-..) (100 00 011)
3EBE      01       01340          FCB     $01     * E (.) (0 0000 001)
3EBF      24       01350          FCB     $24     * F (..-.) (0010 0 100)
3EC0      C3       01360          FCB     $C3     * G (--.) (110 00 011)
```

```
3EC1    04      01370          FCB     $04     * H (....) (0000 0 100)
3EC2    02      01380          FCB     $02     * I (..) (00 000 010)
3EC3    74      01390          FCB     $74     * J (.---) (0111 0 100)
3EC4    A3      01400          FCB     $A3     * K (-.-) (101 00 011)
3EC5    44      01410          FCB     $44     * L (.-..) (0100 0 100)
3EC6    C2      01420          FCB     $C2     * M (--) (11 000 010)
3EC7    82      01430          FCB     $82     * N (-.) (10 000 010)
3EC8    E3      01440          FCB     $E3     * O (---) (111 00 011)
3EC9    64      01450          FCB     $64     * P (.--.) (0110 0 100)
3ECA    D4      01460          FCB     $D4     * Q (--.-) (1101 0 100)
3ECB    43      01470          FCB     $43     * R (.-.) (010 00 011)
3ECC    03      01480          FCB     $03     * S (...) (000 00 011)
3ECD    81      01490          FCB     $81     * T (-) (1 0000 001)
3ECE    23      01500          FCB     $23     * U (..-) (001 00 011)
3ECF    14      01510          FCB     $14     * V (...-) (0001 0 100)
3ED0    63      01520          FCB     $63     * W (.--) (011 00 011)
3ED1    94      01530          FCB     $94     * X (-..-) (1001 0 100)
3ED2    B4      01540          FCB     $B4     * Y (-.--) (1011 0 100)
3ED3    C4      01550          FCB     $C4     * Z (--..) (1100 0 100)
                01560 * NOTE:
                01570 * THE ITEMS MARKED WITH A DOUBLE ASTERISK (**) ARE
                01580 * PROCESSED BY THE EXCEPTION BEEPING ROUTINE.
                01590 *
3ED4    59      01600 TEXT     FCC     /YOU ARE LISTENING TO THE /
3EED    4D      01610          FCC     /MICRO LANGUAGE LAB, PRESENTED
3F0B    42      01620          FCC     /BY GREEN MOUNTAIN MICRO.)/
                01630 *
        3E00    01640          END     BEGIN
00000 TOTAL ERRORS
AGAIN   3E18
BEEP    3E66
BEGIN   3E00
DAH     3E5D
DIT     3E61
EXCEPT  3E36
GETVAL  3E3D
INLP1   3E6C
INLP2   3E79
JUMP1   3E46
LETRLP  3E4D
MORSE   3E99
NEWBIT  3E3F
NEXT    3E48
NORMAL  3E3B
OUT     3E65
OUTLP   3E6A
QLP1    3E8E
QLP2    3E90
QUIET   3E8A
SPACE   3E54
SPCLP   3E56
START   3E11
TEXT    3ED4
```

* If the A accumulator contains a delay value, what is the simplest possible delay procedure?

Decrement A accumulator, and branch on not equal back to decrement A accumulator. When A reaches zero, the loop ends.

* How long is this delay loop (excluding the original load A accumulator)?

2 cycles (DECA) plus 3 cycles (BNE) is 5 cycles, that is 5 times 1.11746 or 5.5873 microseconds. (The less precise value of 5 times 1.15 gives 5.75 microseconds).

By this point the program should be no surprise. Display it using the P command. You'll notice little new; the program has been structured precisely along the lines of the original description. There are just two additions: a comparison is made to find out if the ASCII value in the message represents a greater-than sign or a less-than sign. On greater than, the transmission is completed, the program is ended and returned to BASIC; on less than, the message is repeated.

The other addition to the program is the comparison for **$20**, the ASCII value for a space. There is no space in Morse Code, but for purposes of clarity in transmission, a little extra time is traditionally inserted between words. If a space is found in the ASCII text, then, the QUIET routine

* How many loops would be required for a delay of 500 microseconds?

500 microseconds divided by 5.5873, or approximately 89 (not including the remainder of the program; 85 is used as a test value in the example).

* Why is a delay of 500 microseconds used instead of 1,000 microseconds?

Because the sound alternates between 1 and 0, half the time on and half the time off. 500 microseconds delay is needed for each half.

* What are the Color Computer port addresses?

$FF00, $FF02, $FF20 and $FF22.

* At port $FF22, what is the purpose of bit one?

It is used for sound output.

* What action causes sound?

Alternations of one and zero made at bit one of port $FF22.

* Where is the sound heard?

On the television speaker or through the cassette output.

* What is the term for "setting up" a computer device?

Configuring.

* What PIA address configures port $FF22?

Address $FF23.

* What does PIA mean?

Peripheral Interface Adapter.

is called to insert a slight pause in the transmission.

Interestingly, this program represents quite closely the actual human process from which is was derived. A person reads a message, recalls the code, and transmits it with a series of accurate, trained muscular movements. Programming is not always such a parallel to real life; enjoy it this time.

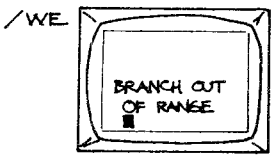
I'd like to take the remaining time to review editor/assembler assembly commands. I haven't presented these before, but you have undoubtedly come across them as you worked with EDTASM+ while completing this program.

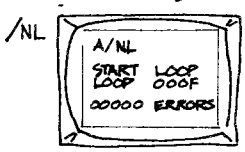
EDTASM's command to assemble your mnemonics into machine language is "A". The "A" command has a number of options called "switches". If you enter simply the letter "A", the assembler will display a scrolling assembled listing, will provide a table of all labels you've used in the source code, note any errors you've generated, and — assuming you have a tape recorder connected and ready — will prepare a cassette containing the final binary object code. The object code tape will be called "NONAME", and will load under BASIC's CLOADM command.

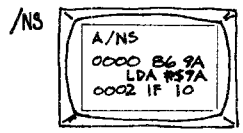
There are many other options that this one. The format of the "A" command is A, space, filename, switch. For example, to assemble and save the resulting machine-language program to tape under the name "DISPLAY", you would type A DISPLAY <ENTER>.

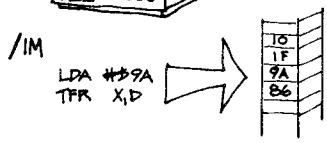
The switches are two-letter command options separated by slashes. These are:

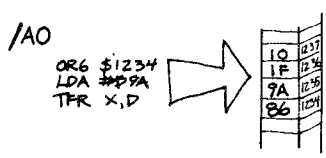
/WE	Wait for errors. The display stops if you have made an error in an opcode, an operand, a range, a typo, etc. The assembler will display a descriptive error message.
/NO	No object code tape is created. I use this switch until I have eliminated all errors picked up by the assembler.
/NL	No listing is displayed. Especially during correction of minor errors, or when you only want to see a list of labels, you can turn off the long listing.
/NS	No symbol table is displayed. The symbol table is the proper name for a list of all the labels used in the program, together with the addresses at which they appear.
/LP	Line printer command. Everything that is displayed on screen is also sent to the Color Computer's serial printer.
/IM	In-memory assembly. This is an excellent debugging tool. The program is not only assembled into binary code, it is also placed directly in memory, ready to run.
/AO	Absolute origin. If you do an in-memory assembly, you can let the machine assemble the program at its predetermined location,









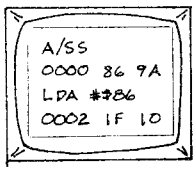


or at the memory location you specified in your ORG statement. Absolute origin uses your ORG.

/MO — Manual origin. This permits you to move the source code, tables, and so forth, that EDTASM needs to work with, so that your own program doesn't conflict with it.

/SS — This is the short screen option. Finding your way through an assembling program with the Color Computer's 32-character screen can be messy. The short screen places the assembled hex address, opcodes and operand on a line by themselves, with the mnemonics out of the way on the next line.

/ss



For details on all these commands, of course, read pages 13 through 16 of your EDTASM+ manual. For the moment, I want you to try the In-Memory assembly option, at your own origin, for the Morse Code program and for as many other programs as you would like to try up to this point. Next time, a new topic.

* Name the timing considerations necessary to produce a beep.

The length of the beep and the frequency of the beep.

* The following questions refer to EDTASM+ assembly (A) commands.

* What does A/NO produce?

An assembled listing and symbol table.

* What does A produce?

An assembled listing, symbol table, and object code sent to the cassette.

* What does A/NL/NS/NO produce?

Only a report of errors at the end of the assembly. No listing appears on the screen.

* What does A/NL/NO produce?

A listing of the symbol table.

* What is the symbol table?

A listing of all labels used in a program, together with the addresses at which they appear.

* What does A/NO/LP produce?

A complete assembled listing send to the printer.

* What does A/IM/AO produce?

A screen listing of the assembled program and symbol table, as well as an object code placed in memory at the origin specified in the source code.

* What does A/SS produce?

A screen listing in "short screen" format, where lines are broken up for easier reading; a symbol table and object code to cassette are also produced.

* What does A/WE produce?

A listing, symbol table, and object code; it also stops at any line in which an error occurs.

* What does A mean?

Assemble.

* What is assembly?

The translation of source (mnemonic) code into binary (object) code.

15.

Hands-on programming takes a back seat for this lesson as the topic once again returns to addressing modes. For this session, you'll want to have your MC6809E data booklet out again. Turn to pages 15 through 17. On pages 15 through 17, where you've previously learned about inherent, immediate, extended, direct, relative, and indexed addressing, you'll also find information about additional applications of those modes.

These remaining addressing modes are called indirect addressing. "Indirect" is an excellent description of this concept, because the operand is not the data (as in immediate addressing), nor is the operand the address of the data (as in extended addressing). No, in the case of indirect addressing, the operand is an address which points to an address where the data can be found. Once again. The operand is an address. That address points to another address. In turn, that address points to the data.

This is easier to understand through example than description. In fact, I've already introduced indirect addressing, but not by name. Recall how I described the power-up of the 6809 microprocessor. When the power is turned on, I said, the processor immediately identifies addresses **$FFFE** and **$FFFF**. It concatenates the 8-bit data found at these addresses, producing a 16-bit number. That 16-bit number becomes the address of the first instruction the processor is to follow. That's indirect addressing.

I know this method, properly called "indexed indirect", sounds like a clumsy and roundabout way of getting information. It's not clumsy, but it is roundabout, and that roundaboutness is its precise advantage. Let's say you've got a super-high-speed action game in the writing, and you need to make moves based on keyboard input. We'll talk about keyboard input itself later, but imagine for the moment that the numbers 0 to 9 are crucial in your game. Say each number causes an entirely different game action, such as shooting balls or using flippers in some sort of arcade pinball. You could, of course, check the value of each number, and if it fits, jump off to a routine. It might

One of the most important differences among the dozen or so popular microprocessors is their respective architectures. The 6809's architecture is created to facilitate finding data, and its myriad addressing modes are key to finding data. Indexed addressing is part of your programming library already; indirect addressing is coming up, together with information on handling and manipulating high-resolution graphics.

* Where does the 6809's instruction decoder get its instructions?

From memory.

* What memory locations does the processor use when the power is turned on?

It uses $FFFE and $FFFF when the power is turned on.

* What does the processor get from memory location $FFFE?

The most significant byte of a 16-bit number.

* What does the processor get from memory location $FFFF?

The least significant byte of a 16-bit number.

* What does the processor do with the two bytes from $FFFE and $FFFF?

It concatenates them.

* What is the result of concatenating the bytes from $FFFE and $FFFF?

A 16-bit number.

* What does the 16-bit number represent?

The address of the first instruction the processor will execute.

* The processor obtains the two bytes at $FFFE and $FFFF, concatenates them, and uses the 16-bit result as an address. What addressing mode is this?

Indirect addressing.

* What addressing mode is LDX #$1234?

Immediate addressing.

* What addressing mode is LDX $1234?

Extended addressing.

* What addressing mode is LDX ($1234)?

Indirect addressing.

* In the immediate addressing mode as represented by LDX #$1234, where is the data?

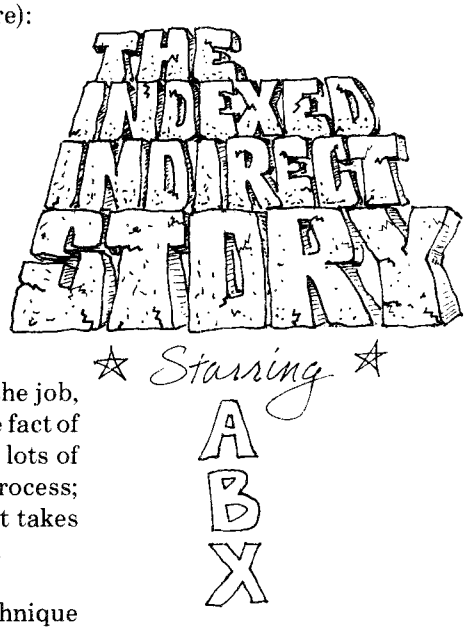
Immediately following the opcode, that is, the data is $1234.

* In the extended addressing mode as represented by LDX $1234, where is the data?

At address $1234 and $1235.

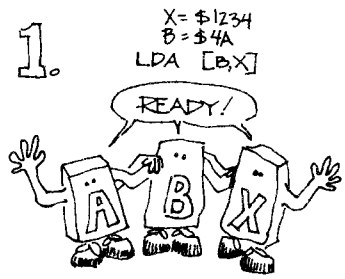
look something like this (follow me in the book here):

```
        CMPA    $30     * 2 bytes
        BNE     NEXT1   * 2 bytes
        JMP     FLIP    * 3 bytes
NEXT1   CMPA    $31
        BNE     NEXT2
        JMP     FLAP
NEXT2   CMPA    $32
        BNE     NEXT3
        JMP     FLOP
```

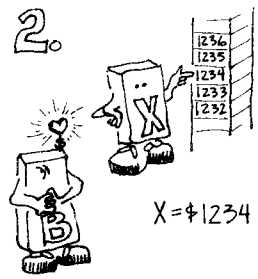


. . . and so on. This kind of programming would do the job, and naturally it would be quite fast due to the simple fact of its being written in machine language. But it grabs lots of memory, and, if timing is critical, it is an uneven process; that is, getting to the last possible choice in the list takes more machine time than getting to the first choice.

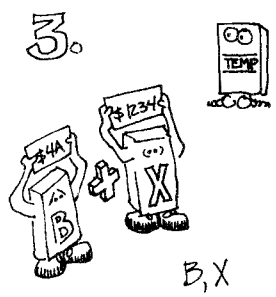
There's an entirely different and very powerful technique available with indexed indirect. Consider this. You can load an index register X or Y with the zeroeth element of a table of subroutine addresses, subtract an ASCII offset from your accumulator, double A, and simply jump to the address indexed indirectly by X plus A. Look in the book:



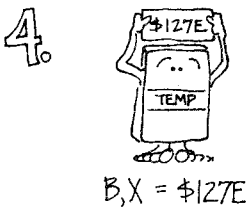

```
        LDX     TABLE   * Addresses
        SUBA    $30     * Strip ASCII
        ROLA            * Double A
        JMP     A,X     * Indexed indirect

TABLE   FCB     $1234   * Subroutine #0
        FCB     $1366   * Subroutine #1
        FCB     $1A9C   * Subroutine #2
        FCB     $20EF   * etc.....
```



The A accumulator is rotated left (that is, doubled) because it takes two bytes to create an address. The indexing process needs to skip every two bytes. Observe that the original compare-branch-or-jump routine takes 70 bytes for ten choices. This indexed indirect routine has the advantage of being more regular and much faster, yet it takes only 29 bytes. For long or fast programs, the savings in time and memory can be significant, and for timed programs, the regularity can be meaningful. Let's put it to work.

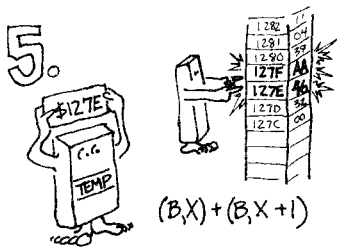


The program will be the Game of Life, a nifty set of rules by which theoretical populations of cells are born, live, and die. The rules are simple. First, this mythical population lives in a regular, two-dimensional grid. On this grid, which can be imagined simply as intersecting horizontal and vertical lines, any given cell position is surrounded by 8 other cell positions. Three "live" cells will give birth in any cell they surround; two or three surrounding cells will keep that cell alive. If the neighborhood population grows over three cells, or falls under two cells, the cell dies.



These simple rules can cause an incredible number of predictable population patterns to arise. Civilizations grow and shrink, rise and fall. Some stabilize in tiny colonies, or rise to great empires. On a video screen, these changes can transform a random population into an astoundngly regular ebb and flow. It becomes hypnotic.

You don't play the Game of Life. Once you have created an initial population, it plays itself until the populations have stabilized in life or death.



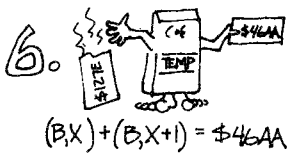
And, with such a simple set of rules, it becomes a perfect computer application. The set of rules by which any applicaton is completed is called an algorithm. Let me reiterate the algorithm for the Game of Life.

1. Where three cells surround an empty position, cell birth takes place.

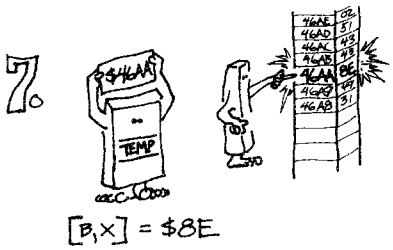
2. Where two or three cells surround a live cell, life goes on.

3. When the surrounding population drops below two, a cell dies.

4. When the surrounding population rises above three, a cell dies.

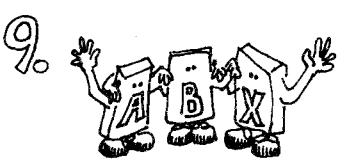


I'd first like you to see this in slow motion. I've got a BASIC program for you.



Problem #26, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 PMODE0:PCLEAR1:DIMA(65,33)
20 FORX=1024TO1535
30 POKEX,127+RND(17):NEXT
40 FORX=1TO62:FORY=1TO30
50 IFPOINT(X,Y)<>0THEN60ELSE140
60 A(X-1,Y-1)=A(X-1,Y-1)+1
70 A(X,Y-1)=A(X,Y-1)+1
80 A(X+1,Y-1)=A(X+1,Y-1)+1
90 A(X-1,Y)=A(X-1,Y)+1
100 A(X+1,Y)=A(X+1,Y)+1
110 A(X-1,Y+1)=A(X-1,Y+1)+1
120 A(X,Y+1)=A(X,Y+1)+1
130 A(X+1,Y+1)=A(X+1,Y+1)+1
140 NEXT:NEXT
150 FORX=0TO63:FORY=0TO31
160 IFA(X,Y)<2THENRESET(X,Y):GOTO190
170 IFA(X,Y)=3THENSET(X,Y,1):GOTO190
180 IFA(X,Y)>3THENRESET(X,Y)
190 NEXT:NEXT
200 FORX=0TO65:FORY=0TO33
210 A(X,Y)=0:NEXT:NEXT:GOTO40
```



The
~End~

* In the indirect addressing mode as represented by LDX ($1234), where is the data?

At the address determined by concatenating the bytes found at $1234 and $1235.

* If addresses $1234 and $1235 contain $A0 and $00 respectively, and if addresses $A000 and $A001 contain $7F and $F0 respectively, what does the X register contain after LDX #$1234?

X contains $1234.

* If addresses $1234 and $1235 contain $A0 and $00 respectively, and if addresses $A000 and $A001 contain $7F and $F0 respectively, what does the X register contain after LDX $1234?

X contains $A000.

* If addresses $1234 and $1235 contain $A0 and $00 respectively, and if addresses $A000 and $A001 contain $7F and $F0 respectively, what does the X register contain after LDX ($1234)?

X contains $7FF0.

* What specific addressing mode is LDX $1234?

Extended addressing.

* What specific addressing mode is LDX ($1234)?

Extended indirect addressing.

* What specific addressing mode is LDA ,Y?

Zero-offset indexed.

* What specific addressing mode is LDA (,Y)?

Zero-offset indexed indirect addressing.

Checking the neighbors

```
* There are basic rules to the
Game of Life.  What is the
general term for a set of
rules?

An algorithm.

* There are four parts to the
Game of Life algorithm, each
involving a cell position on a
grid.  How many potential
neighbors are there in a
regular, two-dimensional square
grid?

Eight neighbors.

* What do three neighbors
produce in a dormant cell?

A cell "birth".

* What do two or three neighbors
produce in a live cell?

No change.

* What do less than two
neighbors produce?

A cell "death".

* What do more than three
neighbors produce?

A cell "death".

* What is the general term for a
set of rules?

An algorithm.

* what does VDG mean?

Video Display Generator.

* The VDG performs what
functions?

The display of alphanumeric
characters, and the display of
several resolutions of color and
monochrome graphics.

* What does SAM mean?

Synchronous Address
Multiplexer.
```

RUN this program. As you watch, a random population is generated in low-resolution graphics. This is the starting population, the garden of Eden, if you will. Once the population has been established, the Game of Life begins. As you watch, I'll tell you that the Game of Life is now a traditional computer problem, originally invented and proposed by British mathematician John Conway. His proposal delighted computer people at the time, and continues to be fascinating as more detailed color screens and more capable computers are developed.

The populations you are watching develop slowly, since BASIC must make a large number of simple comparisons and calculations for which it is ill-suited. Doing such calculations by hand can take hours per generation. Yet the simple-mindedness of machine language finds this a fertile area.

The process of moving from generation to generation is made up of one overall task: check the "neighborhood" of cells, so to speak, and than maintain the status quo, give birth to cells, or kill cells. There are many ways of dealing with that task, however. You might check by neighborhood, or by cell, or look for the presence of any population in an area. Statistically, the Game of Life more often results in a lesser number of live cells — at least after the Garden of Eden has been created and the generational growth has begun.

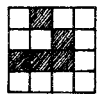
My old friend and teacher Phil approached the algorithm from this point of view. It complicated the programming slightly, but sped along the real time required to move from generation to generation. That's the approach I'm going to use for this example, so I'd like you to keep it in mind. As you progress with 6809 assembly language, you might like to give the Game of Life a try using other approaches.

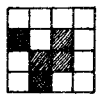
To begin with the Game of Life on the Color Computer, you have to know how to establish the degree of screen resolution you wish to use, and how that mode is manipulated. This is especially important when using the 6847 video display generator because each graphics mode has a different number of colors and a different manner of dealing with how the bits in a memory byte are reflected on the screen. In your notebook, find the MC6847 video display generator data sheet, and turn to page 19.

These modes should already be familiar to you from an earlier lesson, but you should review them now. Also, you'll want to look in the MC6883 data sheet to Appendix A on pages 20 through 22. Because of the way the SAM and the VDG are connected, these special modes are also available. Take time now to read the information on graphic display modes.

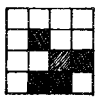
Turn to the MC6847 video display generator (VDG) data booklet and read the information on page 19. Also, open the MC6883 synchronous address multiplexer (SAM) data booklet and read Appendix A, pages 20 through 22. Return to the tape when you have completed the reading.

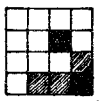
I'd like to select detailed, regular and square picture elements. The highest resolution mode offers individual pixels, but I'd also like color so that different generations and empty cells are shown in different colors — empty cells in black, perhaps, births in yellow and established cells in blue. The mode labeled CG3 offers a 2x2 pixel in four colors. I'll use it.


GARDEN OF EDEN


GENERATION 1

GENERATION 2


GENERATION 3


GENERATION 4

Now, rules in hand and video mode selected, I can structure the neighborhood counting process. To create this screen, 3072 bytes are required to produce 12,288 screen points, at a resolution of 128 by 96. In the screen memory, combinations of bit pairs determine the color, and four bit pairs fill a byte.

By now, you should be able to establish mode CG3 and select the screen memory using the upper memory address and the SAM registers. To accomplish this, remember to refer to both the MC6847 VDG data booklet and the memory map in the MC6883 SAM data booklet. I won't take time for that here; you'll be able to double-check your results against the program listing in the next lesson.

Among the other things to establish is the color set — that is, which set of four possible colors to display. The sets are green, yellow, blue and red for set #0; buff, cyan, magenta, and orange make up set #1. The choice for color set is specified in the Color Computer memory map as bit 3 of output port address **$FF22**.

Some arbitrary decisions must be made. I've selected addresses **$0000** through **$0BFF** for the video display; that address has to be presented to the SAM. Recall that the SAM contains write-only registers which are set or reset to produce the 7-bit upper portion of the display address. Review the MC6883 data sheet if you need to refresh that information.

And finally, interrupts must be turned off to speed the execution of the program. Again, the details of all these setup routines will be shown in the final program in the next lesson; you should attempt to do them in the meantime.

Let me give you a summary now of the pre-program setup:

1. Interrupts must be disabled.

```
* On the Color Computer, there
are two considerations necessary
to establish VDG modes and
colors. What are they?

Port $FF22 and the SAM video
registers.

* What bit of port $FF22 selects
the color set?

Bit 3.

* What are the SAM memory
addresses called?

Write-only registers.

* Name the addresses of the SAM
write-only registers that
control the video display
offset.

Addresses $FFC6 to $FFD3.

* What is the video display
offset address the address of?

The upper-left-most picture
element shown on the video
display screen.

* What is the term for "picture
element"

Pixel.

* In the most detailed mode,
what is the pixel size of the
video screen (pixels wide by
pixels high)?

256 pixels wide by 192 pixels
high.

* What is the pixel size in mode
CG3 (color graphics 3)?

2 pixels by 2 pixels.

* What is the size of the screen
in mode CG3 (width by height)?

128 wide by 96 high.
```

* How many different points are displayed on the screen in mode CG3?

12,288 points.

* How many bytes are required for the screen in mode CG3?

3,072 bytes.

* How many colors are available in mode CG3?

Four colors.

* What are the four colors of VDG color set #0?

Green, yellow, blue and red.

* What are the four colors of VDG color set #1?

Buff, cyan, magenta and orange.

* What bit of what port address selects the color set to be displayed?

Bit 3 of port $FF22.

* In the Game of Life algorithm, what causes cell birth in a dormant cell?

Three immediate neighbors.

* In the Game of Life algorithm, what causes cell death?

Either less than two or more than three immediate neighbors.

* In the Game of Life algorithm, what causes no change to a living cell?

Either two or three immediate neighbors.

* What is another name for a work area of memory?

A scratchpad.

2. One of two possible color sets must be selected. This program will use set #1 for greatest definition.

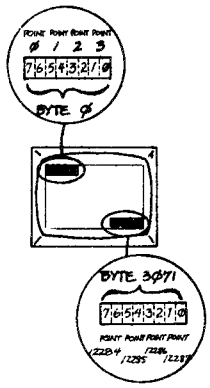
3. The display screen memory must be defined. Screen memory will run 3,072 bytes from **$0000** to **$0BFF**.

4. The color graphics modes must be established. Color Graphics 3 will be used, binary mode 100, to achieve a screen resolution of 12,288 points in four colors.

The final setup information is actually the Garden of Eden population itself. Because there's memory garbage and other information present upon powering up into BASIC or EDTASM+, you'll be able to use that residual material as the Garden of Eden. Creation of random numbers is a subject for later in this course; so until then, Life begins in the garbage pile of memory.

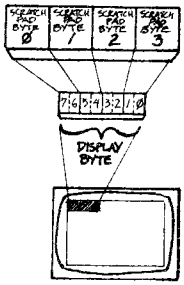
I'll be speaking often about scratchpad memory. Also called a work area, scratchpad memory acts as temporary storage for calculated information on the way to a final result. For example, long division on a microprocessor is a fairly complicated task, and there aren't enough registers inside the 6809 processor to complete it. All the temporary quotients, remainders, and so forth, are stored in a working arithmetic area. When you call for the answer to a complicated mathematical formula in BASIC, the working area required can be thirty or forty bytes, in addition to temporary stack storage of the results from within each and every set of parentheses.

In this version of the Game of Life, 3072 bytes of display memory represent 12,288 screen display points. In other words, each screen display point requires two bits of a byte. These quarter bytes provide for economical use of memory, but are more time-consuming to handle in a program because they have to be shifted left or right, or masked, or whatever, to retrieve their information. Point #0 on the screen is byte zero, bits 7 and 6; point #2 is byte zero, bits 5 and 4; point #12287 on the screen is byte #3071, bits 1 and 0. The relationship isn't difficult, but program handling can be.



Exclusively for reasons of speed, then, I chose to set up a scratchpad memory 12,288 bytes long — one byte for each point on the screen. Although it's wasteful of memory, it's very speedy because my "neighbor" information is immediately accessible in raw form. Since there are from 0 to 8 neighbors for every point, I could have used nybbles, but I chose to use the whole byte to avoid the time required for rotating and masking.

I've also made an arbitrary decision to choose **$1000** to **$3FFF** for these 12,288 bytes of scratchpad memory. That's hex **$3000** bytes. So the screen display runs from **$0000** to **$0BFF** and the scratchpad runs from **$1000** to



$3FFF. All the memory that's left for the program itself is **$0C00** to **$0FFF**. I'll put the program at **$0C80**.

So I'd like you to begin by writing a program beginning at **$0C80** to perform the setup, and to clear scratchpad memory to zero. Once again, the setup is in four steps: disable interrupts, select color set #1, point display memory to begin at **$0000**, and choose color graphics mode 3. Scratchpad memory runs from **$1000** to **$3FFF** and must be filled with zeroes.

Before I leave you with this project, I'd like to suggest that although there is a simple way to fill memory, there is a faster but less obvious one. The simple way is to load an accumulator with the value needed to fill memory, to point the X or Y register to the start of that memory, and to store-and-increment your way through.

The less obvious method is to use the 6809's fast and powerful stack instructions. Re-examine the stack instructions in the MC6809E data book, including their opcodes and speed, and — without looking ahead to my solution in the next lesson — work out both ways of filling up that scratchpad memory.

```
* In display mode CG3, how many
display positions are
represented by one byte?

Four display positions are
represented by one byte.

* How many bits of a byte are
necessary for each display
position in mode CG3?

Two bits are necessary for each
display position.

* Why two bits?

Because mode CG3 displays four
colors, and all combinations of
two bits are necessary to
display four colors.

* What addressing mode is LDX
#$1234?

Immediate addressing.

* What addressing mode is LDX
$1234?

Extended addressing.

* What addressing mode is LDX
($1234)?

Indirect addressing.

* What specific addressing mode
is LDX $1234?

Extended addressing.

* What specific addressing mode
is LDX ($1234)?

Extended indirect addressing.

* What specific addressing mode
is LDA ,Y?

Zero-offset indexed.

* What specific addressing mode
is LDA (,Y)?

Zero-offset indexed indirect
addressing.
```