A Monthly Column

Machine Language

Jim Butterfield
Associate Editor

If I May Interrupt

The 6502 goes about its job, executing instructions at lightning speed. As each instruction is completed, the processor checks: should this process be interrupted?

There are two kinds of interrupt, called IRQ and NMI. They have different features and uses, but they share common characteristics. They may only take effect when the current machine language instruction has been completed. At that time, the address of the following machine language instruction is pushed to the stack together with the Status Register. Then the machine gets an interrupt address stored high in memory, and starts to execute instructions from that address. At a later time, when the interrupt job has been serviced, an RTI instruction will cause the previously stored information to be reclaimed from the stack and the interrupted program to continue.

Interrupt is high priority. It tends to be used where fast response is vital. Don't throw it away on some unimportant job which is not time-sensitive; save these big guns for a real time crunch. I tend to recommend the following priorities: if you can, use straight coding; if you need to, use a timer or two; if you must, use interrupt.

Because an interrupt stops the work in progress to handle a special rush job, users often tend to think of it as instantaneous. Not quite. Don't forget that there's a variable wait to complete the instruction under way (up to seven cycles) in addition to the fixed delay of seven cycles while the interrupt does its bookkeeping work. The effect of the variable wait is "timing jitter" – occasionally important even though the time involved is small.

The Big Two

IRQ – Interrupt Request – is the less powerful of the two interrupts, but it's usually easier for the programmer to handle. It may be locked out with an SEI instruction (Set Interrupt Disable) to prevent interruption from striking at an embarrassing moment; the lockout is released with CLI (Clear Interrupt Disable). Using SEI/CLI adds to the possible timing jitter by a substantial amount, of course.

When an interrupt takes place, an SEI-type lockout automatically takes effect, so that another IRQ interrupt will have no effect until RTI releases the lockout. This is handy for the programmer – he knows that the code in his IRQ type system will be free from further interrupts.

NMI – Non-Maskable Interrupt – is more powerful and less controllable. It cannot be locked out. As a result, the programmer has to be much more careful in sensitive areas: for example, changing the interrupt vector itself can be a ticklish job since the coding cannot prevent the NMI from striking in mid-change with potentially disastrous results. To add to the complexity: an NMI could cause an interrupt, and while it is being handled, another NMI could interrupt again. Careful coding is needed to avoid data corruption if such a multiple-level interrupt is anticipated.

There's another fundamental difference between IRQ and NMI. IRQ is level-sensitive: when the IRQ pin on the 6502 chip receives a low level, interrupt is being requested. NMI, on the other hand, is edge-sensitive: when the NMI pin on the 6502 chip goes from high level to low, a "latch" is triggered within the chip that will signal that NMI needs attention. Think of it this way: if I held the IRQ pin low permanently, the computer would be continuously interrupting. It would go into interrupt, do the job, and upon completing with RTI, the interrupt would take place again since IRQ is still low. In contrast, if I pulled NMI low permanently, I would have only one interrupt – the one that was triggered when the signal went low. A new "edge" would be needed to trigger NMI again.

IRQ Latches

This gives us two seemingly conflicting requirements for the interrupt signal at the IRQ pin. First, it must remain active until the interrupt takes place; too brief an IRQ signal might be missed entirely. Next, it must be turned off before the interrupt coding completes its activity, or RTI will just cause a new interrupt. This seems difficult – not too fast and not too slow – but, in fact, we accomplish the job very easily with the help of extra chips.

Most of the interface chips (the best known to 6502 users are the 6520 PIA and the 6522 VIA) contain latches that may be set by the external interrupting circuits, and reset by the 6502. For example, if a timer counts down to zero and signals an interrupt, this will be latched and signalled to the 6502. When the 6502 gets around to servicing the interrupt, it can switch off the latch.

This system of latches allows many interrupts to be received and forwarded to the processor

chip. The computer can then interrogate the interface chip and find out what caused the interrupt. There might even be two events calling for service at about the same time. The computer can decide to service one of them, turn that particular latch off, and do the job. The moment it gives RTI the other event (whose latch is still locked in) will re-interrupt and be serviced. It works out remarkably elegantly.

The interface chips may have external ports or built-in devices such as timers and shift registers which are allowed to cause interrupts. Each of these may be logically connected to or disconnected from the interrupt line. It seems complex at first; but a little practice will show the system to be straightforward and logical.

Registers

You may recall that only the instruction address (Program Counter) and Status Register (sometimes called the PSW) are saved on the stack during an interrupt. If you plan to use the A, X or Y registers during your interrupt processing, you must save them by pushing them to the stack. Just before giving RTI, bring them back. Your interrupt must be truly "invisible" to the code that was interrupted.

It's quite easy to implement interrupt. You must be especially careful; debugging is much more difficult for this type of code.

Try to keep your interrupt code short, and let the "background" program pick up and do most of the work. The briefer the interrupt program, the more often you'll be able to service interrupts; that will often yield a more powerful system.

Be very careful that a long interrupt doesn't disturb a critical timing process in background code. More than one thermal printer has had its head "smoked" by a sluggish interrupt that didn't know that the background program was waiting to turn the heat off.

A Monthly Column

Machine Language:

Hexed!

Jim Butterfield
Associate Editor

You often find nonsense printed about hexadecimal numbering systems. For example, one source says, "We use hexadecimal numbers when programming in machine language, since that's what the computer uses." Balderdash! There is no such thing as a hexadecimal computer – they're all binary.

It may seem hard to believe at first, but hexadecimal numbers are for human convenience. The computer is happy with binary – in fact, binary is all it's got – but we are not likely to wax enthusiastic if we are asked to place a value of 00001100 into location 1110100001001100. To make it easier for people, we like to condense binary.

Binary

The computer is made up of circuits and wires. Each wire carries either of two kinds of electrical signal – full voltage or no voltage. There's no volume control needed here: it's all or nothing. This two-condition situation is called binary, for its two states: voltage or no voltage, on or off, yes or no, up or down, one or zero.

The one/zero name for the two conditions is handy: it allows us to describe a group of logic signals by a stream of digits. If the computer has a group of eight wires, three of which are carrying full voltage while the others have no voltage, we can describe these wires' states concisely and accurately with the expression 00101100.

Now, there's a very important group of 16 wires called the *address bus*. These wires "call up" a certain part of memory. We might write out such an address as 1110100001001100, giving the condition of each wire of the address bus. The contents of each memory location is delivered on a group of eight wires, called a *data bus*; we might store 00001100 into a location. A group of eight "bits" of information is called a "byte".

But it seems unwieldy to write the individual bits out, one by one.

Enter Hexadecimal

We can shorten these values by grouping the bits together, four at a time. Thus, the address 1110100001001100 may be broken up into 1110-1000-0100-1100. Further, we can give a name to each of the 16 combinations that four bits can have. For example, 0000 can be written as digit 0; 0001 as digit 1; 0010 as digit 2; and so forth. The weighting of the four bits is 8-4-2-1, so that we can quickly see that 0101 can be represented as 4+1 or 5.

This works well for the first ten combinations: 0000 is written as 0 and 1001 as 9. But there are six combinations that total ten or more. Our objective is to write one digit to represent the four bits, so we can't write binary 1010 as 10 for ten; that's two digits. We pick a new scheme for these values: 10 is written as a letter A, 11 as a B, and so on, until we reach 15, which is written as F. The whole table becomes:

0000-0	0100-4	1000-8	1100-C
0001-1	0101-5	1001-9	1101-D
0010-2	0110-6	1010-A	1110-E
0011-3	0111-7	1011-B	1111-F

Now we can write address 1110100001001100 as hexadecimal E84C, which is more compact and easier to remember. We can go the other way easily, too: if we see a value of hex 85 we can write it immediately as binary 10000101 if we need to. Note: this is not the same as the decimal value eighty-five, and we tend to say "eight-five" to keep the two number systems clear.

So we can view hexadecimal notation as a compact way of writing the computer's binary numbers. Hexadecimal, by the way, means "based on 16". You can see that there are 16 combinations, 16 different digits.

Converting To Decimal

If we have a hexadecimal number like 85, we sometimes would like to know its equivalent value in decimal. For example, if we PEEK the number in BASIC, we would see a value of 133 stored in the same location – that's the decimal value. We often need to do conversion. Even to PEEK, we'd need to change the hexadecimal address into decimal so that we could tell BASIC where to look.

In the early days (remember?) we used to be told that a number like 263 means "two hundreds, and six tens, and three units." Same rules for

hexadecimal, except that we use powers of 16 instead of powers of 10. So 85 is "eight sixteens, and five units"; or, to put it mathematically, 8 x 16 +5. This works out to 133, as mentioned before. An address like E84C works out as 14 x 4096 + 8 x 254 + 4 x 16 + 12. The 14 is the value of the E digit, and 4096 is the third power of 16. The whole thing works out to 59468.

You can do this quickly on your computer (don't forget to use the asterisk for multiplication). If you have a pocket calculator, there's an easier method. Type in the value of the first digit. If there are any more digits, multiply by 16 and add the value of the new digit. Repeat until you run out of digits.

Let's try this with E84C. Type in 14 (that's the E). Multiply by 16 and add the 8. Multiply by 16 and add the 4. Multiply by 16 and add 12 (for C). That's it: you should get 59468 as before.

Decimal To Hexadecimal

You will often have a decimal number that you would like to convert to hexadecimal. There are several different methods of doing this.

An easy manual method is to divide repeatedly by 16: the remainder is the next hexadecimal digit, going from right to left. If we started with 133, dividing by 16 gives 8 with a remainder of 5. The 5 is the right-hand digit. Now divide the 8 by 16: you get zero with a remainder of 8. This goes to the left of the 5 to give a result of 85 hex.

Remainders are hard to do on calculators and computers. Here's a method I prefer that works easily on either:

If the number is less than 256, divide by 16; otherwise divide by 4096. You'll get a number which has a whole and fractional part. The whole value is your first digit; make a note of it and then subtract it. Now multiply by 16 and repeat the whole procedure: you'll get two digits for numbers less than 256, and four for greater numbers.

Suppose we have 59468 on our hand calculator. Divide by 4096; you'll get a number like 14.51855. The 14 is your first digit, E: write it down and then subtract the 14. Multiply the remaining .51855 by 16 and you'll get 8.2968. Note the 8 behind the E, subtract 8, and you're ready for the next multiplication by 16. Keep going and you'll get the 4, and finally the last digit will be 12 (it may be 11.99, but we can stretch a point), for which we write down C. Result: hexadecimal E84C.

Hexadecimal numbers are for our convenience. They are very close to the computer's internal notation – binary – but a little more compact and easier for us.

We've talked about simple conversion methods from hexadecimal to decimal and back. They are useful for small computers. If you are a numbers freak, there's lots more for you to dig into: negative numbers, fractions, and even floating point hexadecimal. But the basics will take you a long way.

Some beginners wonder if machine language programmers know secret spells and incantations to make their programs work. I tell them that it's purely logical – no special secrets are required. But it's nice to know how to deal with a hex... number.

MACHINE LANGUAGE

Jim Butterfield, Associate Editor

Part I

NUMERIC OUTPUT

Outputting strings from machine language is no problem. The programmer takes the characters from memory and sends them out. Numbers need more work: the binary values must be changed to ASCII characters which must be sent out one at a time.

An added complexity is format: numbers often need to be carefully formed into a specific number of characters, so that they will print neatly in columns. Zero suppression is often desirable, so that a number such as 00204 will print as 204. Some of these jobs are fairly straightforward mechanical tasks; the hardest part is often the math routine which is needed to break up a binary number into several digits.

Single Digits

Binary values of zero to nine are easy. All we need to do is to change them to ASCII before sending them out.

We've mentioned before that ASCII represents the character zero, for example, as hexadecimal 30, decimal 48. PRINT CHR$(0) will not print a zero character – indeed, it won't print anything – so that we must do the job with PRINT CHR$(48). So, to print a binary zero, we must change it to hex 30, binary one must be changed to hex 31, and so forth, up to binary 9 changing to hex 39. Binary 10 is a different matter: we must make two digits out of it, one and zero. The easiest way to convert a single digit is with an ORA command: ORA #$30 will insert the desired high bits.

When we move on to more complex numbers, we'll need to remember that each digit, as we generate it, must be converted to ASCII before output.

Let's write a simple program to print several single numeric digits. We'll use $FFD2 for PRINT; this will work on all PET/CBM machines, VIC, and Commodore 64. Our coding goes:

```
      LDX  #$00   (start at zero)
LOOP  TXA         (move number to A)
      ORA  #$30   (convert to ASCII)
      JSR  $FFD2  (print it)
      INX         (go to next number)
      CPX  #$0A   (less than ten?)
      BCC  LOOP   (yes, print it)
      RTS
```

The output looks like a large number – the digits are printed side by side – but, in fact, it's ten independent digits.

As an exercise, let's convert the above program to BASIC POKEs and run it. Our BASIC equivalent goes:

```
100 DATA 162, 0,   138,  9,48
110 DATA 32,210,255,   232,  224,10
120 DATA 144,245,  96
200 FOR J = 848 TO 861:READ X
210 POKE J,X:NEXT J
300 FOR J = 1 TO 10:SYS 848:NEXT J
```

The first three lines give the machine language program in decimal. The individual instructions have been separated by spaces to make them more visible. Lines 200 and 210 POKE the program into the cassette area. Finally, line 300 invokes the machine language program ten times; you'll get a hundred digits printed.

Hexadecimal Output

Hex output, like input, is fairly easy. Hexadecimal might be viewed as a compact way of representing binary, and since the computer has binary, the conversion must be easy. It is. All we need to do is grab four bits at a time. Each group of four bits is a hex digit value, which can be converted to ASCII and then output. For example, a decimal value of 225 (hex E1) can be converted this way: take the high four bits, binary 1110, and convert and print as a hex character. That works out to a letter E. Now take the low four bits, binary 0001, and do the same, giving us the digit 1. We've printed E1, the hex value.

Let's get technical. How do we get the four high bits? By giving four shift-right instructions. The bits obligingly move over to the low order side, and zeros are left in the vacated space. Later, how do we get the four low bits? By taking the original value and performing an AND #$0F, which wipes out the high bits.

When the four-bit group is extracted, how do

we change to ASCII? If the four-bit value is zero to nine, we can use the simple ORA #$30 as mentioned before. For the six high values, ten to fifteen (A to F), we would need to use arithmetic, usually the ADC command. Of course, we could bypass the whole question by setting up a table of digits and looking up each digit. Most programmers go for the arithmetic.

Multiple bytes are no problem for hex. We just convert them starting at the high order end: each byte generates two hex digits. Let's write a program to convert some memory bytes into hex and display them. First, a subroutine to convert and output a four-bit value in the A register as two hex digits:

```
HEXDIG CMP  #$0A      (alphabetic digit?)
       BCC  SKIP      (no, skip next part)
       ADC  #$06      (add seven)
  SKIP ADC  #$30      (convert to ASCII)
       JMP  $FFD2     (print it)
```

There are a couple of curious coding quirks above. We need to add seven to the alphabetics: why does the coding say ADC #$06? Because the carry bit is set, that's why. Adding six plus a carry makes a total increase of seven. Another oddity: the subroutine doesn't return with RTS. Instead, it goes to another subroutine; when the other subroutine (FFD2) returns, it will return directly to the caller.

Now an outer subroutine. This one breaks a byte in the A register into two four-bit numbers and prints the two digits. It uses HEXDIG, above:

```
HEXOUT PHA            (save the byte)
       LSR  A
       LSR  A         (extract four...)
       LSR  A         (... high bits)
       LSR  A
       JSR  HEXDIG    (print hex char)
       PLA            (bring back byte)
       AND  #$0F      (extract low four)
       JMP  HEXDIG    (restore ASCII)
```

Again, we save an RTS by doing a JMP direct to a subroutine.

Now we can do the main job: displaying a number of memory locations:

```
   JOB LDX  #$00      (counter)
 JLOOP LDA  $FFC0,X   (get a byte)
       JSR  HEXOUT    (print it)
       LDA  #$20      (space char)
       JSR  $FFD2     (print it)
       INX
       CPX  #$0A      (ten bytes yet?)
       BCC  JLOOP     (no, do another)
       LDA  #$0D      (RETURN char)
       JMP  $FFD2     (print it)
```

We've written the program to display a specific range of addresses. You may change it to display what you wish.

The four LSR instructions may be considered the equivalent of dividing by 16. That's what the word "hexadecimal" means, of course: hex for six and decimal for ten, giving a total of 16.

Sneaky Hex

You may have decided that hexadecimal output is quite easy. It is, compared to decimal, and that gives us an interesting possibility.

Could we write hex numbers that looked like decimal numbers? In other words, could we print decimal 22 by somehow converting it to look like hex 22, and then printing it? It sounds complex: decimal 22 would be written as hex 16, and hex 22 has a decimal value of 34. Not much in common there. But there's a gimmick.

The 6502 processor has an arithmetic feature called "decimal mode." When we invoke it (with the SED, Set Decimal, command), decimal arithmetic takes place using numbers that look like hex. In other words, the decimal value of 22 is stored as hex 22. The proper name for this kind of number is not hexadecimal, of course. This numbering system is called "binary coded decimal."

We can't go into the inner mysteries of BCD at this time, but a few facts can be noted. Decimal mode affects only the ADC (add with carry) and SBC (subtract) instructions; all other instructions still deal with binary numbers. If you're going to play with decimal mode, kill the interrupt for the moment; your interrupt routines may not be able to cope with "new math." And remember to put everything back (clear decimal mode, restore the interrupt) when you've finished doing the task at hand.

Decimal mode arithmetic is great for things like keeping score in video games. The scores can be easily translated and delivered to the screen. But decimal mode is not too good for serious mathematics: multiplication, division, square roots and such become much harder to handle. For most applications, stick with binary.

We'll be talking about how to convert binary numbers to decimal in the next session.

MACHINE LANGUAGE

Jim Butterfield, Associate Editor

Part II

NUMERIC OUTPUT

This is the second in a three-part series on techniques of handling numeric displays or printouts in machine language.

Preparing decimal output can be done in a number of ways. The methods for converting binary integers to decimal can be summarized by direction: right-to-left or left-to-right. In both cases, there is usually a need to perform division. And don't forget that each digit must be converted to ASCII before it is output.

No matter which way we do the job, we need to plan the output format. A one-byte number might require three decimal digits to be printed (e.g., 255), but a two-byte number might need five digits (e.g., 65535). It's often a good idea to plan to output a fixed number of digits, since numbers may need to be printed neatly into columns or onto specific parts of the screen. We might also find it desirable to suppress leading zeros on a number so that 00307 becomes 307, with leading spaces.

Right-To-Left

The method goes something like this: divide by ten. The remainder is the rightmost digit. If the quotient is non-zero, repeat. Thus, a binary value of 287 is calculated: divide by 10, remainder 7; divide quotient 28 by 10, remainder 8; divide quotient 2 by 10, remainder 2. The quotient becomes zero at this point, so we have the three digits – 2, 8, and 7.

The digits come out backwards, however. In the above example, we can't print the 7 the moment we calculate it, since we must work out two earlier digits. That's not a problem, since the digits can be placed into a buffer area – or on the stack, for that matter.

Right-to-left is attractive because it automatically finds the number of digits that need to be printed; the procedure stops when a quotient of zero is reached. You can immediately spot numbers that are too big. It's also very easy to insert leading spaces to fill out the number to any desired length. You'll need a good divide-by-ten routine, of course.

Left-To-Right

This method takes a little more effort to set up, but generates digits in the "normal" order, which allows you to output them directly. Zero suppression adds a little extra code.

We must start by assuming the number of digits that we wish to output. Let's say, for example, that we expect up to three digits. We would follow roughly the following procedure:

> Set FACTOR to 100;
>
> Divide the number by FACTOR;
>
> The quotient is the next digit;
>
> Take the remainder, set FACTOR to 10, and repeat;
>
> Then set FACTOR to 1 and repeat; or for that matter, the remainder from the last calculation will be your last digit.

To convert 287, we divide by 100; the quotient of 2 is our first digit. Take the remainder (87) and divide by 10; the quotient of 8 is the next digit. Finally, the remainder of 7 is our last digit whether or not we divide it by 1.

We can achieve this without a formal division routine; repeated subtraction will work efficiently enough for most purposes. We might change our algorithm to read:

> Set FACTOR to 100;
>
> Set COUNTER to 0;
>
> If the number is greater than or equal to FACTOR, then subtract FACTOR from the number, add 1 to COUNTER, and repeat this step;
>
> COUNTER now contains the first digit; you may print it.
>
> Now set FACTOR to 10, COUNTER to 0, and repeat.

Our example of 287 would have 100 subtracted from it until it reached 87. The counter would have counted 2 subtractions, so we can

send the digit 2 to output.

The various factors (1000, 100, 10, 1, or whatever is needed) may be stored in a table for quick reference rather than calculated. Using true division would be faster than our subtraction algorithm. But since we'll never need to subtract more than nine times for each digit (and since we're likely to spend much more time delivering the output digit to its destination), it's not much of a worry.

Mathematics fiends will tell you that the left-to-right procedure may be easily extended to generate decimal fractions. Useful, but only if you are using binary numbers with fractional parts in the first place.

An Example

Let's do some very quick code to output a dozen numbers from memory in decimal. We'll use the left-to-right method. Zero suppression won't be used. Address FFD2 will be used for output (PET/CBM/VIC/C64 compatible).

```
OUTPUT LDX  #$00       (number counter)
       STX  COUNT
NXNUM  LDA  $0350,X    (get mem value)
       LDY  #$02       (2+1 digits)
LOOP   CMP  TABLE,Y
       BCC  DONE
       SBC  TABLE,Y
       INC  COUNT
       BNE  LOOP
DONE   PHA             (add seven)
       LDA  COUNT
       ORA  #$30
       JSR  $FFD2
       LDA  #$00
       STA  COUNT
       PLA
       DEY
       BPL  LOOP
       LDA  #$0D
       JSR  $FFD2
       INX
       CPX  #$0A
       BCC  NXNUM
       RTS
TABLE  .BYTE           1,10,100
```

It's fun to do this in a practical example. Let's POKE it from BASIC:

```
100 DATA 162,0, 142,144,3, 189,80,3
110 DATA 160,2, 217,132,3, 144,8
120 DATA 249,132,3, 238,144,3
130 DATA 208,243, 72, 173,144,3, 9,48
140 DATA 32,210,255, 169,0, 141,144,3
150 DATA 104,136, 16,225, 169,13
160 DATA 32,210,255, 232, 224,10
170 DATA 144,210, 96, 1,10,100
200 FOR J=848 TO 902:READ X
210 T=T+X:POKE J,X
220 NEXT J
230 IF T<>6199 THEN STOP
300 SYS 848
```

It will take a few moments to POKE the program in place; after that, the decimal numbers come out with blinding speed (especially if you have cleared the screen so that there is no need for scrolling). The numbers, by the way, are the same values as in the DATA statements in line 100 and part of 110.

But there's more.

These are the conventional methods, and they have a number of variations that we haven't mentioned.

But there's a very fast and radically different method available on the 6502. It uses Decimal mode in an unusual way to generate decimal number output super fast.

More on that the next time around.

MACHINE LANGUAGE

Jim Butterfield, Associate Editor

A Bagel Break

Let's walk through an example of programming a complete game, including machine language. We'll make it a simple one: "Bagels," a guessing game that has appeared under other names, including the commercially packaged game, *Master Mind*.

We'll make this one simple, with few frills. We could do it entirely in BASIC, of course; we're using machine language for the practice and for the thrill of seeing the answers come up instantly. You can judge for yourself whether or not machine language handles the job more efficiently.

Ground Rules

We will assume that BASIC will generate the random codes. Yes, you can generate pseudo-random numbers in machine language, too, but we'll shorten the job with BASIC. Once we're into a game, we'll stay entirely in machine language.

The program is written to work on all Commodore machines up to and including the VIC and 64. This means that we need to be careful about memory, since different machines have differently arranged memories. We'll avoid this problem by using the cassette buffer area that is located in the same area in all these machines. And of course, we'll use the built-in Kernal routines that work on all Commodore units: FFD2 to print, FFE4 to get a character.

Planning

We'll need the following work areas:

- A counter which keeps track of the number of guesses (let's put this at $0240 hexadecimal);
- A counter which says how many "exact" matches have been found on this guess (let's use $0241);
- A counter which says how many "inexact" matches have been found (use $0242);
- A counter to keep track of the number of characters typed by the player (we'll use $0243);
- A place to keep the mystery code (four locations from $0244 to $0247 hex);
- A place to put a copy of the mystery code (from $0248 to $204B);
- A place for the user's guess (from $024C to $024F).

Why do we make a copy of the mystery code? Because we will destroy parts of this copy as we test for matches. That way, we will never count the same item twice as a match.

Writing The Program

We lay out a blank piece of paper and try to write the logic. We assume that the BASIC program has placed the mystery code (alphabetic characters from A to F) into hex addresses 0244 to 0247 before it calls upon our program to play the game. Here we go: we'll write a "main routine" first. Although we plan to put it into the cassette buffer (starting at hex 033C), we don't need to write in the addresses – yet.

```
START   LDA #$00
        STA $0240
```

We set our "number of guesses" to zero for starting. Now, on to the next guess:

```
GUESS   INC $0240
        LDA $0240
```

Our guess-number is set one higher, and we bring it into the A register.

```
        CMP #$0A
        BEQ QUIT
```

If we've had nine guesses, we quit here and let BASIC take over. By the way, we don't know exactly where to branch ahead, so we give the branch location a name rather than an address. We'll fill this in soon. In the meantime, if we don't branch, it's time to play:

```
        JSR PLAY
```

This subroutine will do the whole job of receiving one guess from the user and accounting for it. If the user guesses perfectly, the Z flag will be set. In any other case, we'll need to go back:

```
        BNE GUESS
QUIT    RTS
```

Again, we may not know the exact address to which we're looping back at the time we scribble down our first program outline. We'll fill it in later. Sometimes we do this by "hand," and sometimes an assembler program will do it for us. A full-scale assembler will take the "labels" we have used – GUESS, QUIT, and PLAY – calculate their addresses, and make the substitution for us. If we have a smaller assembler, or are assembling by hand, we'll need to write in the addresses. We do this in two columns:

```
        033C   LDA  #$00
        033E   STA  $0240
GUESS   0341   INC  $0240
        0344   LDA  $0240
        0347   CMP  #$0A
        0349   BEQ  $0350
        034B   JSR  $0351
        034E   BNE  $0341
QUIT    0350   RTS
```

The programmer will quickly learn to convert the program into whatever form his development programs need.

We'll assume this translation (at least in part) and continue with subroutine PLAY. First, we must print the guess number. The binary number in the A register must be converted to ASCII, and printed, together with a following space:

```
0351  PLAY     ORA  #$30
               JSR  $FFD2
               LDA  #$20
               JSR  $FFD2
```

Now, on to the main play. Let's zero the counters, including the player input count:

```
               LDX  #$00
               STX  $0241
               STX  $0242
               STX  $0243
```

Here comes another loop, as we wait for each character to be input. We test each character to make sure that it's a letter from A to F:

```
0366  INLOOP   JSR  $FFE4
               CMP  #$41
               BCC  INLOOP
               CMP  #$47
               BCS  INLOOP
```

We have a legal letter; echo it to the screen and put it to memory.

```
               JSR  $FFD2
               LDX  $0243
               INC  $0243
               STA  $024C,X
```

We must also copy the "secret" code into a work area, so that we can destroy it as we test for matches:

```
               LDA  $0244,X
               STA  $0248,X
```

Have we received all four letters of the guess yet? If not, go back:

```
               CPX  #$03
               BNE  INLOOP
```

Now we may check for exact matches. X is conveniently at three, so we may count it down as we compare:

```
0381  COMPAR   LDA  $0248,X
               CMP  $024C,X
               BNE  SKIP
```

If they don't match, we'll skip the next part. If they do, we must count the match and destroy the values so that we don't use them again:

```
               INC  $0241
               LDA  #$00
               STA  $0248,X
               STA  $024C,X
```

Now, our coding rejoins. We move along to test for the next match:

```
0394  SKIP     DEX
               BPL  COMPAR
```

We have logged any exact matches. Now we must look for the out-of-place matches. We may use X and Y to move through the two values, remembering to skip zeros.

```
               LDY  #$00
0399  RETRY    LDX  #$00
039B  CHECK    LDA  $0248,Y
               BEQ  PASS
               CMP  $024C,X
               BNE  PASS
```

Again, if we see a zero (already counted) or no match, we skip the next bit and go to PASS. Otherwise, we've got a match; we count it and destroy the entry, as before:

```
               INC  $0242
               LDA  #$00
               STA  $0248,Y
               STA  $024C,X
```

Our code comes together again. We have two loops to pick up:

```
03B0  PASS     INX
               CPX  #$04
               BCC  CHECK
               INY
               CPY  #$04
               BCC  RETRY
```

Now we may print the two results, stored in $0241 and $0242. A loop will save a little time and space:

```
               LDX  #$00
03BC  PLOOP    LDA  #$20
               JSR  $FFD2
               LDA  $0241,X
               ORA  #$30
               JSR  $FFD2
               INX
               CPX  #$02
               BCC  PLOOP
```

Now a carriage return to end the line. Finally, we must check for a "correct" solution (exact matches = 4) so that the calling routine will know whether to quit or not:

```
               LDA  #$0D
               JSR  $FFD2
               LDA  $0241
               CMP  #$04
               BNE  PLAY
               RTS
```

That's it for our machine language part; we'll start to put it together next time. ©