PAGED MEMORY

As the next step towards understanding the fundamentals of machine code programming, we must examine the way computers organise and manage their memory. Here, we look at the constraints imposed on both memory pagination and the operation of the CPU by the machine's use of the binary system.

In the first instalment of the Machine Code course we gave an analogy of the way in which a computer stores information in the form of electric current. We used the example of a factory where each worker had an individual switch pattern that lit up four light bulbs in the manager's office, thus identifying who was at work. This showed how information (i.e. the name of the person who is working) could be represented by using a flow of electricity.

In our example, we found that by using four switches and bulbs we could represent the numbers from 0 to 15. In other words, there were only 16 possible patterns. However, if we had used eight switches and bulbs instead, then we could have made 256 unique patterns (2×2×2×2×2×2×2×2=256) and, therefore, have been able to count from 0 to 255.

In your home computer, memory is arranged in individual banks of eight switches, and each of those eight-switch banks is called a *byte*. In general, the CPU handles information one byte at a time, which means, in effect, that it can only add, compare, and store numbers between 0 and 255. This might seem to limit its arithmetical capabilities, but that isn't the case. If you think about doing a sum like 63951 + 48770 = ? then you will see that you actually manipulate the individual digits one at a time. Similarly, the CPU can perform arithmetic on large numbers using one byte at a time.

Because it has eight switches, a byte is a place where an eight-digit binary number can be stored. Each of these binary digit positions is called a *bit* — the smallest possible unit of information. A bit in a byte is either ON or OFF, a binary digit is either 1 or 0.

It's often important to talk about individual bits in a byte, so the convention is to number the bits 0 to 7 from right to left in the byte. If a byte contains the binary number 00000001, then we say that bit0 is 1, or that bit0 is ON, or that bit0 is SET; all the other bits are 0, or OFF, or CLEAR. Thus in the binary number 01001000: bit3 is SET, bit6 is SET, bit4 is OFF, bit7 is 0, bit0 is CLEAR, and so on. In a byte, bit0 is also called the Least Significant Bit (LSB), and bit7 the Most Significant Bit (MSB).

Computer memory, then, can be conceived as a long strip of squared paper, eight squares wide, and thousands of squares long: each row of eight squares is a byte, each square is a bit in a byte. Memory is useless if you can't locate items in it, so each of the bytes has an identifying label called its *address*; the address isn't written anywhere on the paper (or in the byte), it's simply the number of the byte in memory, counting from the start of memory. The first byte, therefore, has the address 0, the next byte has address 1, the next has address 2, and so on. If you want to write something in byte43, then you start at the bottom of the memory (at byte0) and count through the bytes until you reach byte43.

When you get there nothing will identify that byte as byte43 except for its position — you've counted forward from byte0, you've reached 43, so this must be byte43. The bytes of memory are actually minuscule banks of eight-transistor devices (one device per bit, eight devices per byte) etched into the chips inside your machine, and they are identical in everything except their physical position.

However, there is one drawback to this method. This system of memory addressing would be fine if there were only a few hundred bytes. The CPU can count from 0 to 100 in fractions of a millisecond; but computers have thousands of bytes, and counting from 0 to 20000 must take some appreciable time, even for a microprocessor. The way a computer overcomes this problem is by dividing memory into *pages*, just as books are.

If we continue to think of computer memory as a strip of squared paper thousands of squares long and eight squares wide, we can imagine cutting that strip on the boundary of every 100 bytes (i.e. cut across the boundary between byte99 and byte100, cut across the boundary of byte199 and byte200, byte 299 and byte 300, and so on). Each of the strips of paper between the cuts is now a page of 100 bytes. Page0 starts at byte0 and continues to byte99; page1 starts at byte100 and continues to byte199, page2 is byte200 to byte299, etc. Now to find any byte, say byte3518, we don't have to count 3518 bytes from the start of memory because we can see from the address that this byte must be on page 35. Therefore, we need only count 35 pages from the bottom of memory, and then count the bytes from the bottom of that page until we reach byte18 on the page, which must be byte3518. Try it with a strip of squared paper if you haven't followed this.