CHANGE OF ADDRESS

At this stage in the course, we take a detailed look at how the two index registers, X and Y, are used in indexed addressing. We illustrate the value of indexing by reference to several example programs.

The original concept of the stored program computer was that by storing the program in the same place (and in the same form) as the data on which it was to operate, the program could modify itself as it was running. The major use of this feature was not to modify the actual instructions themselves but to modify the addresses where the instructions got their data. Imagine the problem of having to access a table of several thousand numbers, and having to give separate instructions for each because each instruction could refer only to the one unchangeable address.

This problem was greatly alleviated with the introduction of the concept of *address modification*. In this way, the same instruction could be repeated any number of times and be made to refer to different addresses where data was stored by using the changing value in a register to alter the address. We use this sort of concept all the time in BASIC programs. For example:

```
FOR I = 1 TO N
PRINT TABLE(I)
NEXT I
```

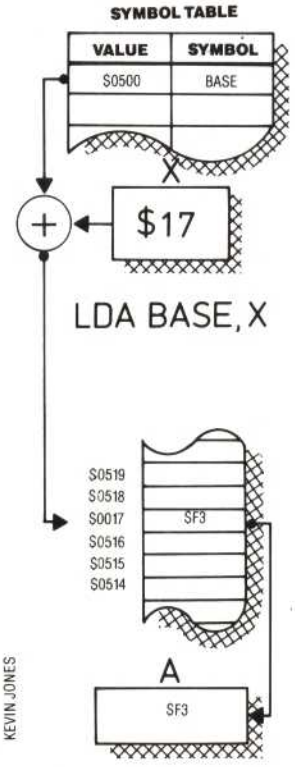
In this case, the same PRINT instruction refers to different data each time it is used by modifying the basic data item (TABLE) using an indexed value (I), which is changed each time it is used.

The fundamental principle of *indexed addressing* is that the contents of the index register are added to the base address given in the instruction to produce the *effective address* — the memory location that is actually accessed. If this instruction occurs within a loop, then the adjustment to the index register (usually an increment or decrement) can be performed within the loop as well, and thus we can easily access a whole table of values.

The 6809 not only has two registers for this purpose, the X and Y registers, but also two further registers — S and U. In special circumstances it is possible to use the program counter as well. To make the whole subject of indexed addressing even more complex, there are a variety of different modes of indexing. However, these do cover nearly all programming requirements. We shall be using indexing, in one form or another, in all our programming from this point, so there will be plenty of opportunity for you to familiarise yourself with the variety of ways in which it is used.

Index Links

The symbol BASE — initialised as $0500 — is the address of the first byte in a table of values. The indexed addressing mode instruction LDA BASE,X takes the value of BASE, and adds to it the contents of the X register, to produce the actual address of the byte whose contents are loaded into the accumulator. If this instruction is inside a loop of which X is the counter, then the entire table can be accesssed, byte by byte, in sequence. Since X is a 16-bit register, the loop could range over the entire memory space ($0000 — $FFFF in an eight-bit system such as the 6809)



KEVIN JONES

Indexed addressing is indicated by adding ,X to the operand field — if the register used is the X register, of course. So the general form of an indexed instruction is:

```
Opcode    Offset,Index Register
  LDA     TABLE1,X
  STA     TABLE2,Y
```

In many situations the offset is zero, in which case it can be omitted. For example:

```
Opcode    ,Index Register
  LDA     ,X
  STA     ,Y
```

Let's see how this works in practice. Suppose we have a table of 64 eight-bit values stored at $3000 and we want to access the bytes in sequence. We can define the base address and reserve space for the table using the directives:

```
          ORG       $3000
TABLE     RMB       64
```

These instructions set the program counter to $3000, define TABLE as beginning at $3000, and reserve the next 64 bytes. We now access the bytes using the next piece of code: the new ORG directive means that our code will be stored in a different part of memory from our data. When you start using indexed addressing this is a sensible precaution against a loop getting out of control and causing your program to overwrite itself.

```
          ORG       $1000
COUNT     FCB       0
          LDX       #0
LOOP      LDA       TABLE,X
```

We now alter the value in the X register:

```
          TFR       X,D
          ADDD      #1
          TFR       D,X
```

This is an awkward way of incrementing X, although it can be useful when we are incrementing or decrementing by numbers greater than two. We will look at the alternatives to the method used here later in the course. The last fragment of code increments the count and checks to make sure that it is not 64 (in which case the program is finished and there is no need to loop again):

```
          INC       COUNT
          LDB       COUNT
          CMPB      #64
          BLT       LOOP
```

There are a number of ways of improving the efficiency of this code. One of the most useful