the PC to obtain the *effective address*. The problem with this addressing mode is how to calculate the offset correctly: this requires calculating the difference between the address of the data and the current value of the PC, remembering that the PC is incremented as soon as an instruction is loaded into the processor. When the instruction is being executed, therefore, the PC points to the following instruction.

This method is complicated by the wide variation in the lengths of 6809 instructions — from one to five bytes long. For example:

```
LDX OFFSET,Y
```

takes up one byte for the op-code, and one byte for the *post-byte*, which is used for any indexed instruction to specify the index register being used, and whether or not indirection is to be taken into account. The offset can take up zero, one or two bytes depending on its size. Zero offsets and offsets that can be expressed in five bits can be incorporated in the post-byte (though some assemblers cannot handle the choice very accurately). Larger offsets require an extra byte (if they can be expressed in eight bits) or an extra two bytes. Special zero or five-bit offsets are not allowed when the PC is used for indexing. The instruction:

```
LDY OFFSET,X
```

would require yet another extra byte because the op-code for LDY is two bytes long.

If you enjoy writing Assembly language programs, and you are familiar with deciding what data addresses to use and where to locate your subroutines, then the associated tasks of looking up (and sometimes working out) the op-codes, converting addresses into two-byte format and manually calculating jump offsets will soon become second nature. A much simpler alternative to doing this assembly by hand, however, is to buy an assembler and let it do the work for you, since it must calculate the length of every instruction anyway. Most assemblers use the special notation PCR (Program Counter Relative), which makes the assembler use the PC as the index register and calculate the offset. For example:

```
DATITM  FCB  0
..... .....
        LDA  DATITM,PCR
```

TERMINAL EMULATION

We give a subroutine that uses this technique to allow emulation of a variety of terminals, so that a program written to use a particular type of terminal can be run on your system. The differences between terminals are most apparent in the codes that are used to control the various screen functions, such as clearing the screen and positioning the cursor. These may be control codes (characters whose ASCII code is less than 32) or escape sequences, which consist of the Escape character (ASCII 27) followed by any other character or sequence of characters. Our simple routine allows only for the substitution of one control character by another, or a single character following Escape by another single character. But the routine clearly shows how such an emulation is carried out. Two tables are kept: one contains control characters; the other the Escape characters. If a program issues a control character, for example, then this character is used as an offset into the table to pick up the actual character that should be displayed.

Being fully relocatable, the routine can be added on to any other program in any position. We assume the existence of an operating system routine (OUTCH), which sends the character in the A accumulator to the screen, and we use JMP to access this routine — which should be at a fixed position in memory. Note that the ORG directive must still be given, although it has no effect. The character to be displayed should be in A.

INSTRUCTION LENGTHS

The problem of calculating the length of instructions is not confined to using 'program counter relative' addressing. It is often necessary to know the total length of a routine to be fitted into a restricted memory space — for example, in a ROM. Any book on 6809 Assembly language, or the manual of an assembler, should include a table of mnemonics along with associated data. For each mnemonic, this data would include a corresponding op-code, the total length of the instruction (though this may not be possible, in which case the minimum length will be given followed by a '+' sign), the number of clock cycles that the instruction takes to perform, and the effect of the instruction on the condition code flags.

The general rules for calculating the lengths of instructions — and hence for writing compact code — are:

1) Most op-codes are single byte; those directly affecting the contents of S and Y (except for LEA) and some affecting U (such as LDY and STS) are two bytes long.

2) Any indexed addressing will necessitate the use of a post-byte, and possibly a further one or two bytes depending on the size of the offset.

3) Data following the op-code for immediate mode will be one or two bytes long, depending on the size of the register used.

4) Addresses should be one byte long if in the direct page (usually locations zero to $FF), and two bytes otherwise. Not all assemblers make proper use of direct addressing, so an address may turn out to be two bytes when only one was expected.

The problem of calculating the time taken by each instruction is equally complicated, if for no other reason than that the time depends on the number of bytes that must be fetched, i.e. the length of the instruction. This is important in real-time applications and for driving some peripherals. The time for each instruction is given as the number of clock cycles — or, at least, the