Around in circles

But seriously: Auntie John is back on the Machine Code business – and he means it.

Up until now every program we have written – and that's not very many to be honest – has been totally linear. By linear I don't mean in the same sense as one of those record decks with the pick-up arm that moves in a straight line instead of being attached to the corner and swinging in an arc.

I actually mean that imaginary 'flow of control' which seeps through the program and only goes in one direction – down. You could imagine it starting at the top of the listing and working its way down, instruction after instruction. All nice and simple. And deadly boring. If all programs were written this way, they would take up fifty billion times more memory to store them and be as predictable as Neighbours. Why? Well read on, gentle reader, and find out.

First, let's tackle the problem of achieving a type of GOTO instruction from machine code. Here we have to learn another bit of magic – the difference between relative and absolute. A certain physicist created quite a stir when he pronounced that nothing was absolute and everything was relative, depending on how fast you were going at the time. This was, of course, obviously complete nonsense. Or so the public thought at the time. It turned out that his theories explained some rather odd natural phenomena and gradually acceptance grew, until today the name of Eamon McPhitrick is known in at least two pubs in Kilkenny. (Professor McPhitrick is convinced that Albert Einstein stole the idea from him after attending his lecture on 'The Potato Soup Model of the Atom' in which he likened sub-atomic particles to vegetables.)

But back to the problem: relative and absolute. Think of a piece of squared paper. Now put a dot in any square. Looks nice, doesn't it? To refer to the position of this dot, we have two choices. We can decide on a fixed point of reference and give the dot a co-ordinate, such as four squares up and three squares right. This is how the normal graph system works, and its an absolute addressing technique. The other way is to stick your finger anywhere on the page and describe the location of the dot in terms of how far away it is from your finger: three squares down and two right. If you move your finger this relative address is useless, but at least it worked when your finger was in the right place. And if you remember where you placed your finger using an absolute system, you could always find the dot. Easy eh?. Now you should understand why computer programmers are all mad.

Essentially that's the difference between relative and absolute, and a Z80 assembler provides both types in a jump instruction. The absolute jump works like this:

```
  JP  2-byte-address  &c3 &nn &nn
eg JP &1234           &c3 &34 &12
```

The relative address looks like this:

```
  JR 1-byte-offset    &18 &nr
eg JR 4               &18 &04
eg IR -1              &18 &FF
```

Immediately an advantage of the relative jump becomes apparent – it only takes two bytes to store it (the instruction and the offset) whereas the absolute instruction takes three. The main disadvantage is that the offset of the relative can only reach about 127 memory locations of either side of the instruction while the absolute jump can access the entire memory – more than 64000 locations. Working out the offset for the relative instruction is always a pain, but if a program is written completely with relative addresses it can be placed anywhere in memory. You pays your money and you takes your choice.

You may remember from your knowledge of Basic that in order to allow a program to make decisions you used the IF and THEN statements, perhaps with the odd GOTO or GOSUB. A condition was checked for between the IF and THEN which controlled what happened next. Such as:

```
If x=62 THEN PRINT "Wow! You
picked the lucky number!"
```

or

```
If x=62 THEN GOTO 2000:REM
My PRINT statement is at line 2000
```

I'm sure you get the idea. In machine

code there are no IF or THEN instructions, which is a great pity. However, making our own is not too difficult – but it's a lot easier if you have bought an assembler by now! To be honest, if you have struggled this far through the wonders of machine code, you are either interested in me or the editor. That being the case, you should have thought about buying one by now. There are some in the Public Domain (ie free) and some supplied with the wonderful CP/M (yawn) operating system. Check Wacci for details.

The major instruction is COMPARE or CP. It always uses the Accumulator (A) register. When you use CP, it checks the contents of the Accumulator with whatever other number you supplied. Not only does it check to see if they are equal, but it decides which is the greater or lesser. All from one instruction – what a bargain! To inform us of its decision, the CP instruction sets or resets some little flags for us.

A flag is an internal signal that we can examine to check a special result. It's the CPU's way of passing information to us. Whenever we press some money into its metaphoric palm, the flags metaphorically whisper what we want to know. Flags are binary – they can only be in one or two states. These states are called Texas and Nebraska. (No they're not. That was a weak joke. I thought things were getting too serious here). These states are called Set and Reset.

The most important flat is the zero flag, usually shortened to Z. The Z flag does a wonderful thing for us. It checks for zero. (Staggers back in amazement and trips over the dog). However, in typical computer style, that does not exactly mean what it sounds like. Look at this short program:

```
LD A,10
CP 10
```

The CP instruction looks at the contents of A (ie 10) and compares it with the value we supplied next to it (ie 10). Compare works by subtracting the value from the accumulator, and in this case the answer is zero.

"Hey!" says the Zero flag, "that's my cue! Here I am, here I am! I'm set! I'm set!"

In the next example, Mr Z Flag is depressed because the answer is not zero, and so he refuses to be set. He is reset.

```
LD A,10
CP 9
```

Now we are on the final stretch. To create an IF/THEN construct, we must somehow combine the jumping instructions with all that testing the flags business. Any guess what? Right – some new instructions.

Relative Ones		
JR Z,n	&28 &nn	Jump Relative if Z flag is SET
JR NZ,n	&20 &nn	Jump Relative if Z flag is RESET
Absolute Ones		
JP Z,&abcd	&ca &cd &ab	Jump to &abcd if Z flag is SET
JP NZ,&abcd	&c2 &cd &ab	Jump to &abcd if Z flag is RESET

So here is our first non-linear program. It is also our first experience of a 'label'. These labels are phenomenally useful. An ingenious concept, invented by a postman called Mr. Label in the mid 1950's.

"Instead of trying to remember lots of tricky addresses," he said, "why not just give them a name, and remember those instead."

The program below is an example of decision making. There are two places where the program may branch depending on the contents of the memory location &9000. As an added bonus, there is a third option which is carried out only if neither of the branches are followed.

```
The program:
          LD A,(&9000)    ;If the contents of &9000 are
          CP 10           ;equal to 10 then jump
          JP Z,LABEL1     ;to this absolute address . .
          CP 11           ;If 11 then jump here
          JP Z,LABEL2     ;to this address . .
          LD A,0          Otherwise put a zero into
          LD (&9001),A    ;this address and then go to
          JP STOP         ;the place to stop.

LABEL 1:  LD A,42         ;This short routine puts 42 into
          LD (&9001),A    ;the memory location &9001
          JP STOP         ;and stops

LABEL 2:  LD A,67         ;This short routine puts 67 into
          LD (&9001),A    the memory location &9001
          JP STOP         ;and stops
```

Next month we'll look at this option and how to get it to loop back on itself. Much more interesting I'd say.