AUNTIE JOHN

Nibbles and bytes

John Kennedy assembles the pieces of machine code madness

In the last few thrill packed months we have explored the exciting world of machine code and how to go about using it on your trusty CPC. We even took a break for a quick game of Life.

However, now is the time to start getting serious. Ok, you know a bit about Z80. You know how to type the listings into your computer using a Basic program. You even know that you must put a RETurn statement in at the end or your code will crash.

If you have been experimenting, you will know that converting all those mnemonics into the hexadecimal codes was a difficult task. And as for calculating the relative jump address! Yuck!

No-one has to suffer these injustices any more, because there are programs that will do the suffering for you automatically. I am of course, referring to Assemblers.

Assemblers are jolly useful programs to have. They use two 'files' – a Source code file and an Object code file. A file is simply a collection of data all stored in memory or on disk or tape. A letter saved from your word processor could be a file, or some data produced by a database. There is nothing special about files, they are just useful ways of bundling information into easy to handle packets.

To use an assembler you first produce the source file, which consists of all the mnemonics you wish to be incorporated into your program. The assembler gets to work on this file, thinks for a bit, and then produces the Object file. This file contains all the actual code, complete with all the jump addresses and labels calculated for you.

The source file could be entered from within the assembler itself, or be a separate file produced by another program – a word processor for example.

Of course in practise there are usually one or two mistakes in the source code file which you must edit and correct before you are finished. Eventually you will have a completely error-free program, all ready to save and then run. If you are lucky, it might even work. Just because the assembler produces the object file for you, doesn't mean your program is perfect. For example, the assembler doesn't know that a RET is needed at the end of the program, so if you left one out everything would appear to be ok except for when you actually ran your program. Nasty. Moral – always, always, always save your source code before you execute your object code.

Within the source file, you are allowed to put one or two extra commands. These Directives provide information needed by the assembler, or ask for something to be displayed on the screen for you. The directives change depending on which assembler you are using, but here are some of the more common ones.

Common Assembler Directives

ORG <address>
(short for 'ORIGIN').
This directive tells the assembler where you want the object code to be stored in memory. For short programs, addresses around &8000 are usually ok.

LOAD <address>
Using this command, you can fool the assembler into putting the object code somewhere else. For example, say your program had to run at a certain address, but the address was taken up by the actual assembler program. Bit of a problem eh? Not with the LOAD command! The code will still be assembled as though it was going to the ORG address, but in actual fact it will be stored in the LOAD address.

<label> **EQU** <address>
(short for 'EQUATE')
This is an instruction whose only purpose in life is to make your life easier. Imagine your program was always calling a subroutine at &8032. Instead of having lots of instructions such as

```
        JP &8032
or      CALL &8032
```

if you had placed the command

```
        MyRoutine EQU &8032
```

at the start of the program, from then on you could say things like

CALL MyRoutine
and JP MyRoutine

READ <file>
Instead of taking a source file from memory where it has been typed in with an editor, take it from a file stored on tape or disc.

WRITE <file>
Send the object code directly to a file where it can be loaded and executed later. This is useful if your program is soooooo big that it won't fit in memory at the same time as the assembler and the source code.

DUMP or **LIST**
Produce a table with all the labels and their addresses. A table such as this is useful to see where the various parts of your program are stored, or for PEEK and POKING values directly into memory.

PRINT <text>
Display some text on the screen so you can tell what stage the assembler has reached.

DB
Now we reach a good one. This command allows byte values to be stored directly in memory. For example, if you had the line:
DB 42,42,42
in your source code, then at some point in the object file there would be three bytes with the value 42 all in a row. You can also use DB with text, such as:
DB "HELLO"
which will convert the text into the ASCII codes representing each letter. Text and graphics can be mixed like this:
DB "Hello",32,"Colly",32,"Wobbles"
which uses the ASCII code for a space. DB is useful for storing text such as 'GAME OVER' and 'WELL DONE EARTHLING' inside your source files.

DS
This is similar to DB except it just reserves some bytes for your program to use later. If you needed a table of 100 bytes to store names in, you could put the line
DS 100
in your program.

HEXADECIMAL

Right back at the start of our tutorials, I mentioned hexadecimal, and what a jolly useful thing it was. I also promised that I would explain it, so here goes.

Most humans can count to ten fairly well, using their fingers if necessary. We use a Decimal numbering system, because once the numbers reach ten strange things happen.

It is best explained by looking at digits, so please consider the sequence below.

```
tens column----  ----units column
                00
                01
                02

and so on, up to

                09
                10
                11
```

As soon as ten is reached, we put a one in the tens column and then start again from zero in the units column. All very primary school stuff. But have you ever wondered why we chose to make ten the number where all the changes occur? Why not five or seven or thirty-two? The answer probably has something to do with how many fingers we have, so consider a race of aliens from the planet Zarqyl who have eight fingers on each hand. These aliens are excellent pianists, and are well known for their brilliant computer programming. How do they do it?

Well, they have a head start, because instead of using a Decimal (base ten) numbering system, they use a hexadecimal (base sixteen) system. When they are counting and reach nine, they don't stop to put a one in the tens column, instead they count on until they run out of fingers. THEN they put a one in the next column, and start again. Obviously they would run out of numbers to use if they used our old-fashioned earth numbers, so they have invented their own new numbers. I think you will find that they look surprisingly familiar.

```
16's column----  ----units column
                00
                01
                02

and so on, up to
                09
                0A
                0B
                0C
                0D
                0E
                0F
                10
                11
```

And there you have it. But why does hexadecimal make computing easier? Well, now each byte can be represented by a pair of hex digits. Even if the decimal version is three digits, the hex one is two. Here are some examples:

decimal	hexadecimal
00	00
19	0A
15	0F
16	10
42	2A
100	64
200	C8
255	FF

If you want to split each byte into 'nibbles', that is use two groups of four bits, you can easily tell their values if you are dealing in hex. Another example:

```
binary              hexadecimal
0000  1111    0F          decimal
0001  1000    18          15
 /     /      \ \__       24
1st   2nd     \    \
nibble nibble  1st   2nd
               nibble nibble
```

Which brings us neatly to the end of our introduction to computer mathematics. Bye for now!