# MEAN machine CODE

***If machine code makes as much sense to you as all that techno-babble they spout on Star Trek: The Next Degeneration, then Jason Finch is here to act as interpreter.***

Together we can beat this thing. With my expert tuition and your thirst for knowledge, we can make machine coders out of you yet. In this month's epic extravaganza I'm going to reveal a few things about indexing. I'll also be explaining accumulator and implied addressing modes, together with jumping and returning so that you can create sub-routines. And because I'm so generous, I'm going to tell you all about the zero page and throw in some useful ROM routines into the bargain. What's more, all the sample proglets are written exclusively for 6510+ Assembler which we featured on the Power Pack two issues ago.


## THE ASSEMBLER

Some information has been given about 6510+ in *CF* already, but I'm going to go into a little bit of detail on labels and numbers for you. In the past, my examples have given numbers in hex format. It is vital you know how to convert between this and decimal, whether by using the Action Replay or by doing some amazing mental arithmetic that would impress even a child prodigy.

Let's take as an example the loop program that I gave you last month. Branch instructions in 6510+ can reference labels. By that I mean that you can give a line a name such as LOOP. From then on, the assembler knows that whenever you refer to LOOP, you are referring to the machine language command at that line. Also, we can get rid of the hex notation so that things will be a bit clearer for you. So, the loop program, would become:

```
10 *=49152; START ADDRESS
20          LDY #0
30 RESET    LDX #0
40 LOOP     INX
50          BNE LOOP
60          INY
70          BNE RESET
80          RTS
```

## ENOUGH IS ENOUGH

Well you've got plenty of examples programs this month and there's been quite a lot for you to take in. Experiment with indexed addressing because I'm going to introduce you to the joys of post-indexed indirect and pre-indexed indirect addressing modes at some point shortly. That'll be enough to finish you off if you don't understand things like `LDA 1024,X`. Try using the ROM routines to print your name on the screen, and the stuff you learned last month about changing colours of characters on the screen. Then use indexing to copy your name and the colours to different parts of the screen.

Labels are used because you don't need to know the actual address of the instruction `LDX #0` when you're using an assembler. You simply enter the above program and type `ASSEMBLE` to, surprisingly enough, assemble it. Enter `TABLE` and press the Return key; this will give you a list of labels that your program has used and the address to which they refer. From now on I'll use labels to make things easy for you. Kind or what?

## LOOK IN THE INDEX

One of the most important things to learn about machine language is indexing. You know that the X and Y registers are called index registers; these are the things you use to do your indexing. But what is indexing? Well I mentioned last month that indexing was adding a number to an address to make a new address. Let's look at another rather nifty and somewhat complicated example:

```
100 *=49152; START ADDRESS
110          LDX #0
120          LDY #0
130 MAINLP   LDA 1024,X
140          STA 1024+24*40,Y
150          LDA 55296,X
160          STA 55296+24*40,Y
170          INX
180          CPX #5
190          BNE NORESET
200          LDX #0
210 NORESET  INY
```

## MORE ON ADDRESSING

There are two modes to cover this month are accumulator addressing and implied addressing. They are both unbelievably simple to understand. Accumulator addressing simply means that instead of a number or address, you do something to the accumulator. For example, there is an instruction called `LSR` which stands for Logical Shift Right – I'll be covering it in next month's instalment. You can do the normal `LSR 1024` (like `LDA 1024`) or `LSR A` which will perform the operation on the accumulator instead of a memory location. A silly example would be `LDA A`. This is not actually a legal command and would generate an error when you tried to assemble it. However, the implication is that it would do `A=A`.

Implied addressing was mentioned in last month's Mean Machine Code and is used when the number following the instruction is implied from the instruction itself. `DEX` implies that the X register is to be decremented by just one. `RTS` implies that you are returning to somewhere that you needn't specify.

```
220         CPY #40
230         BNE MAINLP
240         RTS
```

This takes the first five characters from the top left of the screen and repeats them all the way along the bottom of the screen. The Basic equivalent of that listing would be this:

```
100 X=0
110 FOR Y=0 TO 39
120 POKE 1024+24*40+Y,PEEK(1024+X)
130 POKE 55296+24*40+Y,PEEK(55296+X)
140 X=X+1:IF X=5 THEN X=0
150 NEXT Y
160 END
```

There are plenty of other methods of addressing, but we'll look at those in a future issue (well, we've got to keep you coming back somehow). I can't really explain all the uses od addressing here because there are literally hundreds, believe me, you will instinctively know when you need to use it!

## MAKING IT JUMP

In Basic you use GOTOs and GOSUBs to jump about in a program. In machine language you use JMP and JSR. So, let's say you had a machine language program that started at 49152 and you had written another one that did something spiffy which was located at address 50000 onwards.

You could do a JSR 50000 in your first program to call up the second program. At the end of it you do RTS to get back. Let's do some comparisons, just for the hell of it, yeah? What do I care? I'm young and reckless – I want to be free, to do what I want to do, to be what... (*yes, okay, Jason, calm yourself down and let's get back to the machine coding, shall we? – Dave*)

*A Shaq's eye view of the '64.*

Imagine the following Basic program:

```
10 A=PEEK(1024)
20 IF A=1 THEN 40
30 GOSUB 50
40 END
50 POKE 1024,48
60 RETURN
```

The exact equivalent in assembly language would be:

```
5  *=49152
10         LDA 1024
20         CMP #1
25         BEQ LINE40
30         JSR SUBR
40 LINE40  RTS
50 SUBR    LDA #48
55         STA 1024
60         RTS
```

You'll notice that there are two RTS instructions in there – okay, for the terminally lazy and completely braindead I'll point out that there's one at line 40 and another one at line 60. The one at line 40 simply returns to Basic after you do a SYS 49152. In effect, the Basic SYS 49152 command is the same as a machine language JSR 49152 and therefore to RTS makes complete, total, logical sense (don't argue – it does). The RTS at line 60 returns to the address immediately following the JSR SUBR line. It works exactly the same as a GOSUB...RETURN combination.

## ZEROING IN ON THE ZERO PAGE

Zero page is simply 256 bytes of memory right at the start of your C64's chunk of brain cells. The memory of the C64 consists of 65,536 bytes which are divided into 256 blocks of 256 bytes. Each block is known as a page. The first page covers locations 0 to 255 and it is this page that we call zero page. I suppose page zero would have been more logical, but Commodore like to do everything backwards.

The C64 uses these locations to store important information concerning Basic programs, file transfers from tape and so forth. But there are a few that you can use yourself – for instance, locations 251 to 254 are free. This enables you to do things like:

```
100 *=49152
110       LDA #0
120       STA 251
130       LDA #39
140       STA 252
150       LDY #0
160 MAIN  LDX 251
170       LDA 1024,X
180       LDX 252
190       STA 1984,X
200       INC 251
210       DEC 252
220       INY
230       CPY #40
240       BNE MAIN
250       RTS
```

You should be able to work out why the first line of the screen is duplicated along the bottom in reverse. Or at least it should be if you've typed the assembly program in correctly! It uses locations 251 and 252 to keep track of two index pointers which are incremented and decremented by one each time through the loop. The Y register keeps track of how many times the loop has been done – 40 makes it do the whole of the top line. So now you know.

## I'VE GOT THE KEY

An extremely useful thing to be able to do in machine language is to read from the keyboard and send stuff to the screen (okay, so it doesn't sound like a major quake on the excitement Richter scale, but you'll have to take my word for it). By that I mean that the C64 checks out which key is being pressed and displays some corresponding characters on the monitor. To do both of these things you can use ROM routines. These are special pieces of machine code that are part of your C64's memory system; this means that they cannot be changed, but they can be accessed and used. Take the following example, for, er, example:

```
100 *=49152; START ADDRESS
110       LDY #0
120 READ  JSR $FFCF
130       STA 1024,Y
140       INY
150       CMP #13
160       BNE READ
170       RTS
```

This example uses indexing together with a ROM routine and will simply expect you to press some keys, ending in the Return key. Imagine you wanted instead to display some text on screen. There are plenty of ways to do that, but I'll show you just a couple. First:

```
100 *=49152; START ADDRESS
110 LDA #'C
120 JSR $FFD2
130 LDA #'F
140 JSR $FFD2
150 LDA #13
160 JSR $FFD2
170 RTS
```

The above uses a ROM routine at $FFD2 to display a character whose ASCII code is given in the accumulator. The LDA #'C is equivalent to LDA #67 but the 6510+ Assembler is able to convert the 'C into the ASCII code for that character. The LDA #13 in line 150 is the code for the Return key. A slightly more sophisticated version is shown here:

```
100 *=49152; START ADDRESS
110       LDA #<TEXT
120       LDY #>TEXT
130       JSR $AB1E
140       RTS
150 TEXT BYT "FORMAT",0
```

This is yet another ROM routine – at $AB1E – which displays a string of text starting at a location defined by the accumulator and Y register. The accumulator is the low byte and the Y register is the high byte. The < and > symbols allow the 6510+ Assembler to calculate the correct values for you. You must ensure that the strings finishes with a null byte (,0).

For a complete list of Kernel ROM routines (ones like $FFCF and $FFD2) it'd be worth getting hold of a Programmers' Reference Guide as there are too many details to list here.

## NEXT MONTH

"What joyous snippets do I have for you next month?" I hear you ask (*they must have blimmin' loud voices then – Dave*). Well, I'll be telling you all about the rest of the branching instructions, together with zero page addressing and relative addressing. The arithmetic, logical, shift and rotate instructions will also get a look in and I'll be backing this all up with a scattering of example programs so you can get that all important hands-on experience. Be here – you know it makes sense.