MEAN machine CODE

Part 6

It's the beginning of the end for the Machine Language series as Jason Finch ties up all those loose ends to bring everything to a neat and tidy conclusion.

Don't all cry at once, but this is the last in the series of Mean Machine Code. We've been through a lot together and we've struggled through the hard times but hopefully you've had a few laughs on the way. Like that time the screen went blank and the smoke started coming out of the back of the computer. Ahh, those were the days.

In this final part we'll cover only one new instruction. Though that'll leave you a few short; there are aspects of machine language I'm not going to tell you about – this is meant to be a beginners' course and you don't need any extra hassle. The fact I don't understand them myself doesn't come into it! No, I jest, it's just unlikely you'll ever need to use them.

SPRITE FLIGHT

Let's start at the beginning with the *=49152. You know that simply means the code is to be assembled at location 49152 onwards in memory. In this section I'll break the program down into its component parts and explain what they do and why. Remember that the entire program was given last month, or you could just type in these bits to make up one long working program.

```
1000 *=49152
1010 ;
1020 INIT !
1030 JSR $E20E
1040 JSR $AD9E
1050 JSR $B7F7
1060 LDA 21
1070 STA SPEEDX
1080 LDA 20
1090 STA SPEEDY
```

The INIT routine uses ROM calls to read a value placed after a comma in the SYS call. This value is returned in the normal low/high byte format at locations 20 and 21. The machine language then reads in the values and stores them back at locations referenced by SPEEDX and SPEEDY. These are just memory locations and are reserved at the end of the program.

MAKESPRITE

```
1100 ;
1110 MAKESPRITE ;
1120 LDX #0
1130 LDA #%11111111
1140 MAKESPR1 ;
1150 STA 832,X
1160 INX
1170 CPX #64
1180 BNE MAKESPR1
```

The Makesprite routine uses a counter to fill 64 locations from 832 onwards with the binary value %11111111 which is 255 in decimal. This will become the square block sprite that is used by the program.

SETUP

```
1190 ;
1200 SETUP !
1210 LDA #<150
1220 STA SPRXLO
1230 LDA #>150
1240 STA SPRXHI
1250 LDA #120
1260 STA SPRY
1270 LDA #7
1280 STA SPRCOL
1290 LDA #(832/64)
1300 STA 2040
1310 LDA #1
1320 STA 53269
1330 JSR $E544
```

The bit of SETUP code stores a number of values in machine language variables. Of course, variables don't really exist in the true sense of the word; they are memory locations used by the program. The fact you've given them a label is irrelevant, you could equally do something like STA $C539 but labels mean you don't have to know the exact number. The assembler works it out.

Lines 1210-1260 deal with the horizontal and vertical position of the sprite, lines 1270-1280 with the colour and lines 1290-1300 with the definition. The 832/64 is the calculation for the sprite pointer. The final three lines switch the sprite on and clear the screen using a built-in ROM routine at $E544.

READJOY

```
1340 ;
1350 READJOY !
1360 LDA 56320
1370 LSR A
1380 BCS READ1
1390 PHA
1400 JSR JOYUP
1410 PLA
1420 READ1 ;
1430 LSR A
1440 BCS READ2
1450 PHA
1460 JSR JOYDOWN
1470 PLA
1480 READ2 ;
1490 LSR A
1500 BCS READ3
1510 PHA
1520 JSR JOYLEFT
1530 PLA
1540 READ3 ;
1550 LSR A
1560 BCS READ4
1570 PHA
1580 JSR JOYRIGHT
1590 PLA
1600 READ4 ;
1610 LSR A
1620 BCS READ5
1630 PHA
1640 JSR JOYFIRE
1650 PLA
1660 READ5 ;
1670 JSR POSITION
1680 JMP READJOY
```

It's big and meaty; it's the main loop. You may find this routine useful for your own programs. It reads the joystick at line 1360 and then shifts the bits along,

IT'S BUGGED

Remember that wonderful sprite routine I gave you on the Proggy Selector? Well, this month we'll take a look at that and see why it all works. It uses a lot of the instructions you have learned and therefore is a good place to end the series. In the meantime, you will hopefully remember the short assembly language program that I gave to demonstrate indexed addressing. The one that allegedly cleared the bitmap screen. Of course, you all noticed the somewhat accidental bug in it – line 240 should in fact read BNE LOOP-3 instead of just BNE LOOP. But unless you're a complete lemon you will have worked that one out for yourself.

checking whether each one is set or clear. It does this by shifting them into the carry flag and then checking the status of that. You will need five routines called JOYUP, JOYDOWN, JOYLEFT, JOYRIGHT and JOYFIRE as well. The jump to the POSITION routine at line 1670 simply repositions the sprite on the screen after any movement that has been done. You wouldn't need that in a normal joystick reading routine.

POSITION

```
1690 ;
1700 POSITION !
1710 LDA #0
1720 RASTWAIT ;
1730 CMP $D012
1740 BNE RASTWAIT
1750 LDA SPRXLO
1760 STA 53248
1770 LDA SPRXHI
1780 STA 53264
1790 LDA SPRY
1800 STA 53249
1810 LDA SPRCOL
1820 STA 53287
1830 RTS
```

The POSITION subroutine takes the values from SPRXLO and SPRXHI and stores them in the horizontal position register for sprite zero (53248) and the MSB register (53264). If you're using more than one sprite you would have to use a more sophisticated method for storing the high byte. The vertical position is taken from SPRY and stored in the appropriate register (53249), as is the colour information; it's taken from SPRCOL and stored in the colour byte for sprite zero (53287).

The less-than-fancy loop at the start, RASTWAIT, waits for the raster scan line to stop drawing the main display. This simply means your sprite won't flicker or wibble as you move it around. Technical, sweetie, technical. You see, the computer could have drawn half of your sprite before you decide to move it left. Then the bottom half appears a bit to the left of the top half on your display. Not good.

JOYUP

```
1840 ;
1850 JOYUP !
1860 LDA SPRY
1870 SEC
1880 SBC SPEEDY
1890 CMP #50
1900 BCS JOYUP1
1910 LDA #50
1920 JOYUP1 ;
1930 STA SPRY
1940 RTS
```

The JOYUP subroutine is called when the computer detects you've pushed the joystick up, funnily enough. It reads in the SPRY value and subtracts SPEEDY from it. With any luck you'll remember that the SEC in line 1870 is necessary to make the subtraction at line 1880 correct. It then checks this value against the decimal 50 and if it is greater than or equal to 50 it branches to JOYUP1 where the new value is stored back to SPRY. Eventually the SPRY value is stored at location 53249 (see the POSITION subroutine) which results in the sprite being repositioned on the screen. If it happened that pushing the joystick up would have taken SPRY below 50, the code simply ensures that doesn't happen; the LDA #50 in line 1910 takes care of that.

JOYDOWN

```
1950 ;
1960 JOYDOWN !
1970 LDA SPRY
1980 CLC
1990 ADC SPEEDY
2000 CMP #230
2010 BCC JOYDOWN1
2020 LDA #229
2030 JOYDOWN1 ;
2040 STA SPRY
2050 RTS
```

Working in much the same way as JOYUP, this subroutine is called when the joystick has been pushed in the down direction. This time, however, SPEEDY is added to SPRY. The intention is that SPRY should never be more than 229 and therefore if it is less than that value the code will simply go ahead and store it back. If it ends up being 230 or more, the branch at line 2010 will not occur and the value 229 will be stored instead and everyone's happy.

JOYLEFT

```
2060 ;
2070 JOYLEFT !
2080 LDA SPRXLO
2090 SEC
2100 SBC SPEEDX
2110 STA SPRXLO
2120 BCS JOYLEFT1
2130 LDA SPRXHI
2140 EOR #%00000001
2150 STA SPRXHI
2160 JOYLEFT1 ;
2170 LDA SPRXHI
2180 BNE JOYLEFT2
2190 LDA SPRXLO
2200 CMP #24
2210 BCS JOYLEFT2
2220 LDA #24
2230 STA SPRXLO
2240 JOYLEFT2 ;
2250 RTS
```

The JOYLEFT routine is hideously complicated in comparison with JOYUP and JOYDOWN. If you know anything about sprites, you should know that there is a line down the right hand side of the screen called the MSB line. You can't actually see it, but it exists. If the sprite crosses from the right hand side of the line to the left of it when you move the joystick, the computer has to update the high byte of the sprite's horizontal position as well as the low byte.

Lines 2080-2110 subtract SPEEDX from SPRXLO and store the new value back. If this has not resulted in a negative answer the routine will branch to JOYLEFT1. When I say a negative answer, I actually mean looping back to 255. You know that if you subtract 1 from numbers the computer stores ..3,2,1,0,255, 254,253,.. and continues doing that loop. If the branch doesn't take place, then the value has looped from 0 to 255 which means the sprite has crossed the invisible MSB line. Shock horror. So lines 2130-2150 flip the appropriate bit of the high byte to show this has occurred.

Lines 2170-2180 check which side of the line the sprite is now on. If it's on the right, SPRXHI will be one, and the branch will go to JOYLEFT2 which simply returns from the subroutine. No further checking of how far left you have made the sprite go is necessary. On the other hand, if you're on the left side of the MSB line, lines 2190-2230 make sure the sprite isn't pushed so far left that it disappears off the left of the screen. Useful.

JOYRIGHT

```
2260 ;
2270 JOYRIGHT !
2280 LDA SPRXLO
2290 CLC
2300 ADC SPEEDX
2310 STA SPRXLO
2320 BCC JOYRIGHT1
2330 LDA SPRXHI
2340 EOR #%00000001
2350 STA SPRXHI
2360 JOYRIGHT1 ;
2370 LDA SPRXHI
2380 BEQ JOYRIGHT2
2390 LDA SPRXLO
2400 CMP #65
2410 BCC JOYRIGHT2
2420 LDA #64
2430 STA SPRXLO
2440 JOYRIGHT2 ;
2450 RTS
```

Are you getting the hang of this yet?

JOYFIRE

```
2460 ;
2470 JOYFIRE !
2480 LDX SPRCOL
2490 INX
2500 TXA
2510 AND #%00001111
2520 STA SPRCOL
2530 RTS
```

Guess what? JOYFIRE is called when the joystick fire button is pressed down. It reads the sprite colour into the X register and then adds one on to it. This value is transferred into the accumulator at line 2500. Line 2510 makes sure we only get the first four bits which restricts the value between 0 and 15. This is not strictly necessary but is useful in many areas. The value is stored back at line 2520; from the accumulator.

VARIABLES

```
2540 ;
2550 ! VARIABLES
2560 SPRXLO BYT 0 ;HORIZ POS LOW BYTE
2570 SPRXHI BYT 0 ;HORIZ POS HIGH BYTE
2580 SPRY BYT 0   ;VERT POS
2590 SPEEDX BYT 0 ;HORIZ MOVEMENT SPEED
2600 SPEEDY BYT 0 ;VERT MOVEMENT SPEED
2610 SPRCOL BYT 0 ;SPRITE COLOUR
```

This is an easy way to do variables in machine language. Obviously they're not, strictly speaking, proper variables, but they do the same job. You use memory locations referenced by labels to make life even easier.

NOP INSTRUCTION

What may at first strike you as a useless instruction is NOP. It stands for No OPeration and simply means the computer does nothing for two cycles, or two millionths of a second. Building up a timing loop with these things is going to fill the memory, okay, so don't even think about it! NOP will come in usaful when you start playing about with raster interrupts and I've explained it here just so that you know why they appear when you see them. It is usually used so that colour bars and scrolling bits of the screen don't flicker. Timing is all important, and it's NOP that helps solve many of the problems.

NEXT MUNF?

Something tells me there isn't going to be any more of this series next month so I'm afraid that's it folks. Assuming I've done my job properly and you're not a plum tomato, you should be able to grasp at least the basics of machine language. You need to experiment with the commands you've learned and keep disassembling the routines I give away for free in Techie Tips. Try to do things where you can predict what is likely to occur and you should have hours of fun.

If you've got any follow-up questions to ask then please direct them to me at Techie Tips. Also feel free to furnish me with comments on how the series has gone. Take That and take care (well, I had to get them mentioned somewhere!).