PROGRAMMING IS JUST A GAME

Auntie John gives away those game design secrets.

Now we have spent the last few months looking at the basics of machine code programming on the Amstrad CPCs it is time we got down to doing something useful.

You will no doubt recall that we have looked at reading the keyboard, writing to the screen, printing numbers and making sounds. That seems like everything you could possibly need to write a game in machine code. Anyway, using all the routines we developed we have sufficient information to write a simple arcade game.

When I mention something like The module called PRINT-A-NUMBER I mean a piece of assembly language we have looked at previously: Now we reach the first major problem. What game are you to write? That is where the more optimistic programmers will say three dimensional programmers will say three dimensional Zombie attack from alternative galaxies with Hardware scrolling and the more realistic will mutter something like "What about a game of squash?".

SQUASH!

Having decided on a really simple game such as squash, its time to telephone several of the biggest software companies and get some contracts to sign. Perhaps not. After spending ages writing your first game you will even be able to get granny to play it. First games are always dull. Do not be discouraged – everyone has to write first games. One of my first was written on a ZX-81 and was so dull that it put everyone who saw it into a two-week coma. Those who are too young to remember the ZX-81 era may count themselves lucky; older folk will sigh with nostalgia and remember long nights huddled round the little black box for warmth, waiting for the game to load.

The first game you will write will be terrible. It will take a long time to produce, it will crash, and will hold the attention of anything with a higher IQ of a house-brick for less than 10 seconds – but it will be your game.

Your second game will be better and the third better still. By the fourth? Well, perhaps time to think of setting up your own software house? Do not be disheartened; just realise that things will get better, easier and more commercially feasible with time.

We now come to the wonderful task of designing the game. I know you all want to roll up your sleeves and start hammering away at the keyboard but that way leads to messy programming, millions of bugs and probable madness. Unfortunately, it is time for a large piece of paper, a trustworthy Biro and perhaps even one of these little plastic flowchart template things stationery shops stock.

There are as many ways of designing a program as there are ways of coding it. It is for you to decide what style of design you will use, whether it is top-down, bottom-up, middle-out or some other way. Top-down involves starting at the highest level of the program and working down until you reach the really nitty-gritty stuff. It has the advantage that your program will be beautifully-structured and easy to de-bug.

With bottom-up design you start at some tiny piece of code, say a scrolling routine or a way of printing an object, and write more and more complicated code until you have finished. The advantage is that you do not write an entire game before you realise that you will not be able to scroll something fast enough, or for some other reason the game is impossible to write.

Middle-out is a term I conjured on the spur of the moment but it describes the way in which I program very well. It is a kind of cross between top-down and bottom-up.

If you ever get a job as a commercial programmer – programming mini and mainframe computers for a living – you will be compelled to use a design methodology and when using it you must plan your program in a pre-defined order, keeping detailed notes about averything. Design is a very important aspect of programming, so remember not to treat it too lightly. That should not deter you from doing little drawings; a good design is one someone who has not seen a computer previously will understand.

Se we have a large sheet of paper and a pen. For our game design let us use top-down approach, to get the overall structure correct. If we are writing a simple squash game – bat hits ball hits wall – it would look a little like figure one.

Coding

Now we can add more detail. Figure two has expanded on the first few parts of figure one. Notice that we have not even mentioned switching on the computer yet.

It is at that time we can think about coding. Is there part of the design which looks a little tricky? How about moving the ball? Well, try and expand that even further, such as in figure three. You can continue in this way until you have written the entire program in a form both easily understood by you and, at the same time, is almost machine code. This is known as step-wise refinement.

Now you can switch on the computer and start entering the code. Use plenty of calls to subroutines and modules and plenty of comments, as in figure four. The subroutines called in figure four can be written later and each of them may themselves call other subroutines.

Notice how 'flag' is used to store the current game status. If the bat failed to hit the ball, 'flag' would be set to a non-zero value – say 1. An Escape key option is also a good idea; somewhere in the main loop a piece of code checks for key number 66 being pressed and sets flag to another non-zero value – e.g. 2. Thus when the program exists its main loop 'flag' can be examined to see what is happening.

Some may be wondering how to use variables in

```
Set Variables


Set Screen
```

Figure 1. Overall design for Squash

```
                         Set Variables

Set SCORE to zero        SetLIVEs to three        Set initial BAT and
BALL positions.

                         Set Screen

Choose MODE              Choose INKS              Print SCORE and LIVES
                                                  using the PRINT-A-
                                                  NUMBER MODULE
```

Figure 2. Some of figure one in more detail.

Figure 3. Move ball in more and more detail.

```
Move Ball

        Erase old Ball
```

Figure 2. Some of figure one in more detail.

```
;Squash Game (c) Me 1989
        org &4000               ;Start address of code.
        call Set_Variables      ;Routine to initialize SCORE etc
        call Set_Screen         ;Routine to set MODE and stuff
LOOP
        call Move_Bat           ;Print and move the Bat
        call Move_Ball          ;Print and move the Ball
        ld a,(FLAG)             ;FLAG is a variable that is zero
        cp 0                    ;if the game is continuing normally
        jp nz,LOOP              ;but if something happens, it is
                                ;set to a non-zero value.
        ret                     ;RETurn to BASIC.
```

Figure 4. Coding the game.

```
        org &4000               ;Start of code
        call set_variables      ;A subroutine to initialize variable
                                ;values
        call print_score        ;A subroutine that prints the score
        jp loop

;Subroutines

        ld hl,(score)
        call PRINT_A_NUM        ;Our previously defined code module.
        ret

        ld hl,0
        ld (score),hl
        ret

;variable storage space
```

Figure 5. Variables in machine code.

machine code in the same way they are used in basic. Nothing could be simpler; variables are just a few bytes of memory set aside with a label telling you where you are. Let us look at a small piece a program which sets up a variable to keep the score – figure five. The assembler you are using may be more fussy about what you can and cannot define as a label, so check the instructions if you are in doubt. The variable 'score' is defined to be two-bytes long, which means it can store values from 0 to 65535. If 'score' was defined to by only one byte long – using the DB, define byte, pseudo-op instead of DW, define word – the A register would be used to set its value, not the HL pair.

After two days you should have finished typing in your game and be in a position to test it. If you have used modules you have coded and tested previously; things will be much easier than if you wrote the code specially for this program. Testing will also be easier if you have a Monitor program; it allows you to stop the program at any point and print-out the contents of the registers.

You will notice that I have give you no code to type-in; you will have to work out that yourself. When you get the simple game working, why not try adding extra features? Sound is easy if you remember to read the article I wrote about it. You could change the game into a version of *Breakout* or *Arknoid* or even something different.