Roland Takes a Running Jump

One of the staples of the computer gamer's diet is the platform-type game, which features a lot of leaping about and treasure collecting. Peter Green starts a series showing how to write your own.

One of the games that will go down in computing history as a landmark is Manic Miner, a game which originally appeared on the Spectrum and whose success brought forth an army of clones: Jet Set Willy, Jet Set Willy II, Blagger... Most of these games have been converted to run on the Amstrad computers, and you have all probably seen one version or another.

The basic principle is that you have a number of screens built of various types of platform. The player uses a joystick or the keyboard to control a little figure who can run left and right, or jump into the air. Scattered about each screen are both pieces of treasure, which you have to collect, and nasties of various types, often moving, which you have to avoid.

The object of this series is to give you an insight into the problems involved in writing such a game, and the techniques used to overcome the problems. As an example, month by month we'll build up the code for a demo program called (for obvious reasons) 'Roland Takes A Running Jump'. Our hero will be Amstrad's mascot Roland, who will have to leap about the various floors of the Amstrad company offices collecting 3" floppy discs.

Spritely Moves

Obviously the most important, and therefore the first point to consider, is the graphical representation of each play level on the screen. A screen consists of two basic elements: the background of platforms and fixed hazards which never changes, and the player, treasure, and moving hazards which do change.

The fixed background is no problem. It only has to be drawn once, at the start of a level, and can then be neglected provided we arrange for the moving elements not to corrupt it. It's the moving elements themselves which provide the first obstacle.

The type of gameplay we will incorporate is the same as Manic Miner: the player cannot fall through the platform he is standing on, but he can jump up 'through' a platform above him. Thus we need to draw the player's character over the background in such a way that, when he moves on, theback ground re-appears as it was before.

Some computers have sprites as part of their graphics capability. Normally, the pictures on the TV display are generated from the data stored in screen memory: on the Amstrad, screen memory is a 16K block. Put very simply, a sprite is a graphics shape which appears on the TV display without its data being stored in the screen memory. Its data is actually stored elsewhere in the computer, together with data telling the system where on the screen the sprite is supposed to appear. When this point is reached, the hardware generates the TV frame video signal from the sprite data instead of the screen data, so the sprite seems to appear in front of the screen background. Thus sprite graphics can very easily be put on screen, moved around, and taken off again without any effect on the background.

None of the Amstrad computers have this hardware sprite capability. To make something appear on the screen, its data must be written into screen RAM, even if that means overwriting background data. However, there are two methods we can use to produce a 'software sprite'. The first is really pretty obvious: when we want to draw a shape over a piece of background, first copy the section of the screen which is going to be altered into a safe section of memory. Later, we can erase the shape and restore the background exactly as it was by copying the data back again to its original position. This will give an effect that looks just like a hardware sprite, but we won't be using it for our program.

Over and Out

Instead, we'll be drawing our moving characters using XOR mode. Many of you will know what this means, but I'll briefly explain it for the benefit of readers who don't understand logical operators.

Normally, drawing a shape on-screen means taking numbers rom one part of memory (a table of data for the graphic shapes), and copying it into another part of memory (the screen memory). It replaces whatever was there before. But the Z80 processor at the heart of the Amstrad also lets us combine numbers with the ones already existing in a memory byte, using various logical rules, or operators. The operator we're interested in is XOR. The table below shows what the result bit is for each possible combination of bit pairs in the two original numbers. In English, the XOR rule is 'the result bit is a zero if both bits are the same, and a one if the two bits are different'. The first example shows how to XOR two numbers together, using the table or the above rule:

```
First bit    0011
Second bit   0101
Result bit   0110

Example 1:  11010110      Example 2:  10100010
              XOR                       XOR
            01110100                  01110100
             gives                     gives
            10100010                  11010110
```

But now look at example 2. If we take the answer from example 1, and XOR it with one of our starting numbers, the result is the other number that we started with. This always works with XOR: test a few examples to see for yourself. If we take a number, and XOR it twice with another number, we always get the first number back unchanged.

The point of this mathematical diversion is that, if we draw a shape onto the screen using XOR mode, it appears: drawing it a second time in the same place makes it disappear, and we get our original background back as a bonus! The only disadvantage is that new colours may appear at the points where background and shape overlap, as new bit patterns are produced in the screen memory. However, we (and most commercial games of this type) will put up with this for the sake of convenience. Readers lucky enough to own a copy of the excellent game 'Sorcery' will be familiar with the weird colours that appear when characters cross over each other, and this is the reason.

Speed Limits

Now you might think we're laughing, as the Amstrad TAG option lets us print characters using graphics XOR mode. Our second problem rears its ugly head now: speed. For smooth, flicker-free animation, we've got to add and remove characters from the screen fast. Let's do some experiments. Program 1 is a BASIC listing that fills the Mode 1 screen with characters. Run it and time it and you find BASIC needs about 6.5 seconds to print a thousand characters: about 6.5 milliseconds per character.

But the screen is displayed at 50 frames a second, 20 milliseconds a frame. Chop off the time each frame when the hardware is testing the keyboard, updating the sound chip and doing all the other housekeeping tasks, and you can see we only have time to remove and redraw one or two characters per frame before flicker starts to occur. And we haven't even considered things like checking for legal moves, treasure, scores and time left! We need a speed improvement, or we'll have a very dull game.

Can machine code provide a solution? Surprisingly, the answer is no. Program 2 is a routine that fills a Mode 1 screen with characters (for variety, in vertical strips instead of horizontal rows), using the firmware routines that BASIC would normally use. The routine cycles through all 256 built-in characters, so first the program selects all characters to be taken from ROM, and enables the ROM so that we can read from it. Each character column in Mode 1 is two bytes of screen memory wide, so we move across the screen in two-byte increments (the double CALL to SCR NEXTBYTE) writing vertical strips of characters. For each character we must find the address of the eight bytes that make up its matrix, unpack the matrix into the corresponding set of pixel masks for a Mode 1 screen, then combine each mask with the encoded ink we're using, and copy the result into the right place on the screen.

And the time saving? This routine takes about four seconds to fill the screen, a rate of two to three characters per frame scan. Still no good, and we simply can't go any faster using standard routines. Machine code may be fast, but the complicated organisation of the Amstrad screen means a great deal of work has to be done to print a character, which slows things down again. We have to come up with a new trick.

The Plotting Thickens

Well, the fundamental limit of speed is the time it takes just to write the correct data from one place in memory to another: we can't do that any faster. The real time-waster in Program 2 is the conversion of a character from its stored form into its screen form. Cut out that step and we'll be plotting shapes on screen as fast as we can.

So for smooth animation of a number of shapes, we store our graphics already encoded into their screen data format. An example is given in Program 3, which includes the encoded data for a checker board character. The program plots the character at a point on screen given by the given starting screen address (&F830): the call to KM WAIT CHAR is a machine code 'Press any key to continue', after which the character is re-plotted in the same place and thus disappears, as we showed above.

This is fast, but we can do slightly better. Program 3 uses the proper firmware calls to step right and down through the screen memory map, which is often a complicated process. This is due to the very complex addressing system used by the Amstrad hardware, which becomes even worse if scrolling has taken place. It means a great deal of checking has to be done to move around the screen map.

However, we can make a few short-cuts provided we don't scroll the screen after selecting the screen Mode. In this case, the screen addressing looks like Figure 1. It still looks complicated, but provided we don't try to move off the sides, top or bottom of the screen, we can use the following rules. To move right or left one byte, add or subtract one to the screen address. To move down one row, add 2048 (&800) to the screen address. If the carry flag is clear after this, the answer is the new screen address (for example, the byte below &D800 in Figure 1 is &D800+&800 = &E000).

If the carry is set, though, indicating an overflow past &FFFF (the top of memory), we have to add a further correction factor. This consists of &C000 (which takes us back to the start of the screen memory map), and &50 (which takes us to the next row in the same 2K block). For example, adding &800 to &F801 gives &0001 and the carry flag set. Adding another &C050 gives &C051 as the required address, which Figure 1 shows is the correct answer.

Screen momory map if no scrolling has occured after Mode change

80 (&50) bytes

```
          C000   C001  ......  .C04E   C04F
FIRST     C800   C801  ......  .C84E   C84F
EIGHT     D000   D001  ......  .D04E   D04F
PIXEL     D800   D801  ......  .D84E   D84F
LINES     E000   E001  ......  .E04E   E04F
          E800   E801  ......  .E84E   E84F
          F000   F001  ......  .F04E   F04F
          F800   F801  ......  .F84E   F84F

          C050   C051  ......  .C09E   C09F
SECOND    C850   C851  ......  .C89E   C89F
EIGHT      .      .              .       .
PIXEL      .      .              .       .
LINES      .      .              .       .
           .      .              .       .
           .      .              .       .
           .      .              .       .

          C780   C781  ......  .C7CE   C7CF
          CF80   CF81  ......  .CFCE   CFCE
LAST      D780   D781  ......  .D7CE   D7CF
EIGHT     DF80   DF81  ......  .DFCE   DFCF
PIXEL     E780   E781  ......  .E7CE   E7CF
LINES     EF80   EF81  ......  .EFCE   EFCF
          F780   F781  ......  .F7CE   F7CF
          FF80   FF81  ......  .FFCE   FFCF
```

ALL ADDRESSES IN HEX

Program 4 incorporates these improvements, and takes just under one second to print 1000 characters. As an added bonus, there's no reason why we can't encode our data so that shapes appear multicoloured on the screen. To do that using the firmware routines would have required over printing several characters to produce the extra colours, whereas our system requires no extra time at all.

Typing in the Programs

Since most of the programs have been written in machine code it is usually necessary for you to have an assembler so that you can enter the programs (I use Maxam), so that anyone who is new to machine code, and doesn't have an assembler, can enter the program a small Basic program which reads data and pokes the machine code in is provided.

Program 1

```
1Ø MODE 1
2Ø FOR i=1 TO 4Ø*25
3Ø PRINT "A";
4Ø NEXT
5Ø GOTO 5Ø
```

Program 2 - Machine code

```
        ORG &8ØØØ
        LD DE,256
        CALL &BBAB    ;TXT SET M TABLE
        CALL &B9Ø6    ;KL L ROM ENABLE

filscr  LD A,1
        CALL &BCØE    ;SCR SET MODE
        LD HL,&CØØØ   ;start of screen
        XOR A         ;set A=Ø
        LD B,4Ø

loopØ   PUSH BC       ;save loopØ counter
        PUSH HL       ;save screen address
        CALL vstrip
        POP HL        ;restore screen address
        PUSH AF
        CALL &BC2Ø    ;SCR NEXT BYTE
        CALL &BC2Ø    ;SCR NEXT BYTE
        POP AF
        POP BC        ;restore loopØ counter
        DJNZ loopØ
        RET           ;end of program
        VSTRIP         ;enterwithHLpointingto
                       screen address at top of
                       strip

         PUSH AF
         LD B,25       ;screen height in rows
         LD A,1        ;use ink 1
         CALL &BC2C    ;SCR INK ENCODE
        LD (encink),A ;store encoded ink
        POP AF

loop1   PUSH BC
        PUSH AF
        PUSH HL
        CALL &BBA5    ;TXT GET MATRIX
        LD DE,unpack
        CALL &BC53    ;SCR UNPACK
        POP HL        ;get screen address


        LD DE,unpack
        LD A,(encink)
        LD C,A        ;put encoded ink in C
        LD B,8        ;line counter

loop2   LD A,(DE)     ;get pixel mask
        AND C         ;combine with encoded ink
        LD (HL),A     ;put in screen address
        PUSH HL       ;save screen address
        CALL &BC2Ø    ;SCR NEXT BYTE
        INC DE        ;point to next pixel mask
        LD A,(DE)     ;get pixel mask
        AND C         ;combine with encoded ink
        LD (HL),A     ;save in screen address
        POP HL        ;restore previous screen address
        CALL &BC26    ;SCR NEXT LINE
        INC DE        ;next pixel mask
        DJNZ loop2
        POP AF        ;restore printing character
        INC A         ;next ASCII character
        POP BC        ;restore loop counter
        DJNZ loop1
        RET

encink BYTE Ø
unpack RMEM 16
```

Program 2 - Basic poker

```
5 valid$="Ø123456789ABCDEF":ln=1ØØØ
1Ø FOR x=&8ØØØ TO &8Ø6Ø STEP 8
2Ø FOR y=Ø TO 7
3Ø READ num$
4Ø IF (INSTR(valid$,LEFT$(num$,1))<>Ø) A
  ND (INSTR (valid$,RIGHT$(num$,1))<>Ø)
   THEN num=VAL("&"+num$):POKE x+y,num
  ELSE 1ØØ
5Ø chk=chk+num
6Ø NEXT:ln=ln+1Ø
7Ø READ check:IF chk<>check THEN 11Ø
8Ø chk=Ø:NEXT
9Ø CALL &8ØØØ
95 END
1ØØ PRINT"Invalid character in line ";ln
   :END
11Ø PRINT"Checksum error in line ";ln:EN
   D
1ØØØ DATA 11,ØØ,Ø1,CD,AB,BB,CD,Ø6, 792
1Ø1Ø DATA B9,3E,Ø1,CD,ØE,BC,21,ØØ, 688
1Ø2Ø DATA CØ,AF,Ø6,28,C5,E5,CD,26, 1Ø82
1Ø3Ø DATA 8Ø,E1,F5,CD,2Ø,BC,CD,2Ø, 126Ø
1Ø4Ø DATA BC,F1,C1,1Ø,EF,C9,F5,Ø6, 1329
1Ø5Ø DATA 19,3E,Ø1,CD,2C,BC,32,6Ø, 671
1Ø6Ø DATA 8Ø,F1,C5,F5,E5,CD,A5,BB, 1597
1Ø7Ø DATA 11,61,8Ø,CD,53,BC,E1,11, 96Ø
1Ø8Ø DATA 61,8Ø,3A,6Ø,8Ø,4F,Ø6,Ø8, 6ØØ
1Ø9Ø DATA 1A,A1,77,E5,CD,2Ø,BC,13, 979
11ØØ DATA 1A,A1,77,E1,CD,26,BC,13, 981
111Ø DATA 1Ø,EE,F1,3C,C1,1Ø,D3,C9, 1176
112Ø DATA ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ, Ø
```

Program 3 - Machine Code

```
        org &8ØØØ
        LD A,1
        CALL &BCØE ;SCR SET MODE

        LD HL,&F83Ø
        LD DE,grafx
        LD BC,&Ø2Ø8
        CALL plot

       CALL &BBØ6  ;KM WAIT CHAR

        LD HL,&F83Ø
        LD DE,grafx
        LD BC,&Ø2Ø8
        CALL plot


        RET

plot    PUSH BC     ;save width and row counter
```

```
        PUSH HL      ;save screen address of start of current row
do_row  LD A,(DE)    ;get byte of graphic data
        XOR (HL)     ;XOR it with the contents of screen address
        LD (HL),A    ;put new screen data back into screen
        CALL &BC20   ;SCR NEXT BYTE
        INC DE       ;point to next byte of graphics data
        DJNZ do_row  ;loop back until whole row plotted
        POP HL       ;fetch screen address of row start
        CALL &BC26   ;SCR NEXT LINE
        POP BC       ;fetch width and row counter
        DEC C        ;decrement row counter
        JR NZ,plot   ;loop back until all rows plotted
        RET
grafx   BYTE &AA,&AA,&55,&55
        BYTE &AA,&AA,&55,&55
        BYTE &AA,&AA,&55,&55
        BYTE &AA,&AA,&55,&55
```

Program 3 - Basic Poker

```
5 valid$="0123456789ABCDEF":ln=1000
10 FOR x=&8000 TO &8060 STEP 8
20 FOR y=0 TO 7
30 READ num$
40 IF (INSTR(valid$,LEFT$(num$,1))<>0) A
   ND (INSTR (valid$,RIGHT$(num$,1))<>0)
    THEN num=VAL("&"+num$):POKE x+y,num
   ELSE 100
50 chk=chk+num
60 NEXT:ln=ln+10
70 READ check:IF chk<>check THEN 110
80 chk=0:NEXT
90 CALL &8000
95 END
100 PRINT"Invalid character in line ";ln
    :END
110 PRINT"Checksum error in line ";ln:EN
    D
1000 DATA 3E,01,CD,0E,BC,21,30,F8, 799
1010 DATA 11,35,80,01,08,02,CD,21, 447
1020 DATA 80,CD,06,BB,21,30,F8,11, 872
1030 DATA 35,80,01,08,02,CD,21,80, 558
1040 DATA C9,C5,E5,1A,AE,77,CD,20, 1183
1050 DATA BC,13,10,F7,E1,CD,26,BC, 1126
1060 DATA C1,0D,20,ED,C9,AA,AA,55, 1101
1070 DATA 55,AA,AA,55,55,AA,AA,55, 1020
1080 DATA 55,AA,AA,55,55,4F,06,08, 688
1090 DATA 1A,A1,77,E5,CD,20,BC,13, 979
1100 DATA 1A,A1,77,E1,CD,26,BC,13, 981
1110 DATA 10,EE,F1,3C,C1,10,D3,C9, 1176
1120 DATA 00,00,00,00,00,00,00,00, 0
```

Program 4 - Machine Code

```
        org &8000

        LD A,1
        CALL &BC0E   ;SCR SET MODE

        LD HL,&C000  ;start address on screen

        LD B,25
Loop1   PUSH BC
        PUSH HL
        LD B,40
Loop2   PUSH BC
        PUSH HL
        LD DE,grafx
        LD BC,&0208  ;character is two bytes wide (B), eight rows deep (C)
        CALL plot
        POP HL
        INC HL
        INC HL
        POP BC
        DJNZ Loop2
        POP HL
        LD BC,&50
        ADD HL,BC
        POP BC
        DJNZ Loop1
        RET           ;end of main program
plot    PUSH BC       ;save width and row counter
        PUSH HL       ;save screen address of start of current row
do_row  LD A,(DE)     ;get byte of graphic data
        XOR (HL)      ;XOR it with the contents of screen address
        LD (HL),A     ;put new screen data back into screen
        INC HL        ;step screen address right one byte
        INC DE        ;point to next byte of graphics data
        DJNZ do_row   ;loop back until whole row plotted
        POP HL        ;fetch screen address of row start
        LD BC,&800    ;offset of 2048 bytes to next row
        ADD HL,BC     ;add offset to screen address
        JR NC,no_cor  ;no carry, so no correction needed
        LD BC,&C050   ;correction factor for wrap-around
        ADD HL,BC     ;add correction
no_cor  POP BC        ;fetch width and row counter
        DEC C         ;decrement row counter
        JR NZ,plot    ;loop back until all rows plotted
        RET
grafx   BYTE &AA,&AA,&55,&55
        BYTE &AA,&AA,&55,&55
        BYTE &AA,&AA,&55,&55
        BYTE &AA,&AA,&55,&55
```

Program 4 - Basic poker

```
5 valid$="0123456789ABCDEF":ln=1000
10 FOR x=&8000 TO &8060 STEP 8
20 FOR y=0 TO 7
30 READ num$
40 IF (INSTR(valid$,LEFT$(num$,1))<>0) A
   ND (INSTR (valid$,RIGHT$(num$,1))<>0)
    THEN num=VAL("&"+num$):POKE x+y,num
   ELSE 100
50 chk=chk+num
60 NEXT:ln=ln+10
70 READ check:IF chk<>check THEN 110
80 chk=0:NEXT
90 CALL &8000
95 END
100 PRINT"Invalid character in line ";ln
    :END
110 PRINT"Checksum error in line ";ln:EN
    D
1000 DATA 3E,01,CD,0E,BC,21,00,C0, 695
1010 DATA 06,19,C5,E5,06,28,C5,E5, 929
1020 DATA 11,41,80,01,08,02,CD,28, 466
1030 DATA 80,E1,23,23,C1,10,EF,E1, 1096
1040 DATA 01,50,00,09,C1,10,E3,C9, 727
1050 DATA C5,E5,1A,AE,77,23,13,10, 815
1060 DATA F9,E1,01,00,08,09,30,04, 544
1070 DATA 01,50,C0,09,C1,0D,20,E8, 752
1080 DATA C9,AA,AA,55,55,AA,AA,55, 1136
1090 DATA 55,AA,AA,55,55,AA,AA,55, 1020
1100 DATA 55,A1,77,E1,CD,26,BC,13, 1040
1110 DATA 10,EE,F1,3C,C1,10,D3,C9, 1176
1120 DATA 00,00,00,00,00,00,00,00, 0
```

Next Month - Part Two

ACU

Roland Takes a Running Jump

This month Peter Green moves in a mysterious way, but Roland doesn't in our on-going arcade programming feature.

Last month we saw how to get chunks of graphic data onto the screen as quickly as possible by storing them in their encoded form (ie, as the actual data bytes required in the screen RAM by the 6845 video display chip), and then copying them as a rectangular block into the screen memory area. This is not only fast, but it means we can display multi-coloured characters just as quickly as monochrome ones, because the size of the screen data block is identical, however many colours it contains, and the encoding is done in advance, not as the screen is being updated.

We decided to use Mode 1 to compromise between screen resolution and variety of inks available. In this mode, a pixel needs two bits in the display memory for a choice of four colours. To display multi-coloured characters by printing them would need up to three operations in transparent text mode (overprinting three characters, one in each foreground ink), or two operations in TAG (text at graphics cursor) mode: printing one character to set one bit of each pixel as required, then overprinting in OR graphics mode with a second character to set the other bits. Neither system is any good for the sort of speeds we need.

On the move

Now that we can place our graphic blocks, or 'software sprites', on the screen, we have to think about animation. We need a systematic way of controlling the sprite movement.

Well, if we go back to O-Level Physics, we have all the tools we need. Let's consider just one sprite. It has a position, represented by two coordinates (in whatever system we choose): an x coordinate across the screen, and a y coordinate up the screen. We can store these as two variables, and, by altering the values of x and y, place the sprite anywhere on screen with an appropriate bit of program (one that converts x and y to a screen address, then calls the routine we ended up with last month).

Suppose we want our sprite to move smoothly in some direction. This means that one (or both) of the variables x and y have to be altered steadily by a constant amount, each time erasing the sprite at its old position and redrawing it at the position given by the new values of x and y. The amount of change in x and y each time round the 'display loop' can be considered to be the x and y velocities, and we can store these in two more variables, vx and vy. Smooth animation, from the program's point of view, now has the form: erase the old sprite, add vx to x, and add vy to y, to obtain new coordinates. redraw the sprite in its new position and loop back. The only problem with this system is that, quite quickly, the sprite vanishes off the screen.

Bouncing checks

We have to include some kind of checking system, so that our sprites can be made to 'bounce' off obstacles or the edges of the screen. A further problem is that Roland's velocity depends partly on what you're doing with the joystick, and partly on his surroundings. In short, we need to have a system of acceleration. Here acceleration is used in its strict scientific sense of 'a change in velocity', not its popular meaning of 'going faster'.

Acceleration is more complicated to deal with than velocity, but the limited nature of our sprite movements means it isn't too tricky to handle. We're basing our game on established favourites like Manic Miner and Blagger, and in these the animated nasties move back and forth in straight lines. This is easy to handle - for each sprite we store the maximum and minimum x and y values for its path. Each time the sprite moves, its new position is checked against these limits. If the sprite has reached an x or y limit, the acceleration is applied by inverting its velocity in that particular plane: the effect on screen is that the sprite 'bounces', and starts moving in the opposite direction. These 'bouncing points' are decided by the programmer when designing a particular game level.

Roland's acceleration is more difficult. Under joystick control he can move left, right, jump up in the air or stand still. But his movement is restricted by his environment in several ways. Again taking our lead from commercial games, there are a number of 'building blocks' which can be encountered. Let's run through them.

First there are solid walls and floors, which Roland cannot penetrate. I'll call these 'thick' floors, to distinguish them from the 'thin' floors which Roland cannot fall through, but is able to penetrate when jumping up. Then there are the melting floors, which gradually disappear if Roland stands on them. There are conveyor belts, moving either to the left or right. Usually the way these work is that, if you try to move Roland against the direction of travel, he just stands still. If you release the joystick or go in the same direction, he moves with the conveyor at his normal walking speed. All right, in real life his speed when walking would be added to the speed of the conveyor belt, but we are trying to keep things as easy for ourselves as possible.

Finally, we need to keep track of deadly objects, which cause instant death when touched: and treasure objects and the exit from the level, which do not affect Roland's freedom of movement but still have to be recognised. All of these various game elements need to be stored in a kind of map, so that the program knows where Roland is and how he should be moving.

```
rolacc LD A,(rx)      ;get Roland x coord
       LD C,A         ;put it into C reg
       LD A,(ry)      ;get Roland y coord
       LD B,A         ;put it in B
       INC B          ;go down one square (ie check the 'ground' underfoot)
       CALL peek      ;see what's there
       AND &CC        ;lose irrelevant bits (see text)
       JR Z,fall      ;if zero, then Roland is falling, so ignore joystick
       PUSH AF        ;push 'ground' onto stack
       CALL &BB24     ;KM GET JOYSTICK
       POP DE         ;pop 'ground' off stack into D
       LD HL,rvy      ;point HL to variable for Roland's y velocity
       LD B,0         ;assume zero velocity to start with
       BIT 4,A        ;fire button pressed?
       JR Z,newrvy    ;if not, OK to load y velocity with zero
       LD B,-2        ;else Roland jumps upwards with velocity rvy=-2
newrvy LD (HL),B      ;store new y velocity in (rvy)
       OR D           ;combine 'ground' type and joystick data
       AND &C         ;mask off unwanted bits
       LD HL,rvx      ;point HL to variable for Roland's x velocity
       LD B,0         ;assume zero velocity to start with
       JP PE,newrvx   ;and load this if parity is even (see text for reason)
       LD B,1         ;else load B with 'right' x velocity
       CP 8           ;check if Roland is moving right
       JR Z,newrvx    ;and store this if he is
       LD B,-1        ;otherwise he's moving left, so make B = 'left' velocity
newrvx LD (HL),B      ;store new x velocity in (rvx)
       JR newpos      ;now calculate new positions for all animated objects
fall   LD HL,vy       ;point to y velocity
       INC (HL)       ;simulate gravity
newpos ;the routine to calculate the new positions of all animated objects
       ;goes here (next month)
;**********************************************************************
peek   CALL locate
       LD A,(HL)
       RET
poke   CALL locate
       LD (HL),A
       RET
locate PUSH AF        ;save AF
       PUSH BC        ;save x,y coords
       LD DE,40       ;width of one floor
       LD HL,fldat-41 ;point to start of floor data less 40
addrow ADD HL,DE      ;add on row length
       DJNZ addrow    ;once for each row
       ADD HL,BC      ;B now zero, so this line adds column no.
       POP BC         ;restore BC
       POP AF         ;and AF
       RET            ;HL now points to required byte in fldat table
```

Of Maps and Memory

In the good old bad old days of the 8K PET (you know you're getting old when you can remember 8K PETs), screens were character-mapped. This means that a 40 by 25 text screen literally occupied 40 x 25, or 1000 bytes of memory: the screen memory held the ASCII codes for the characters to be displayed, rather than the actual pixel patterns making up the displayed shapes. It was the job of the hardware to send the right patterns to the electron gun on each pass through the frame, the patterns being fixed in ROM. This was useful as memory was so expensive, but made user-defined characters and all the pixel-based graphics like line-drawing, which you take for granted on your Amstrad, completely impossible. The great thing about it was that the display layout was identical to the 'map' we mentioned above. If you wanted to see what your playing piece was standing on, you just PEEKed the code out of the screen location below him.

In contrast, it is extremely difficult to check out Roland's surroundings by examining the Amstrad's screen memory. One character occupies many bytes in the display map, and the firmware routine to read characters from the screen needs them to be in one ink, not multicoloured like our graphics. Thus for convenience, we set up a section of memory to represent the rows and columns of our playing area: this becomes our x and y coordinate system for placing sprites. We can fill this block of memory with our own invented number codes to represent the layout of floors, walls, treasure, nasties and so on, and use it to check on Roland's possibilities for movement.

Bits and pieces

The numbering system for our game elements is completely arbitrary, so we can choose it to make life convenient for the rest of the programming. First, remember that we need to combine the conveyor belt data with the joystick data to arrive at Roland's true x velocity. The firmware CALL to fetch the current state of the joysticks is &BB24, KM GET JOYSTICK, which returns the state of joystick 0 in the A register of the Z80 CPU. Bit 3 is set if the stick is pushed right, bit 2 is set for left. (What a pity this makes right and left the wrong way round in Figure 1). Note that it's impossible to push the stick left and right at the same time, so these bits cannot both be set to 1, which is important later.

What I've decided to do is to set bits in the memory byte representing a given location to indicate the contents of that location. And obviously the first choice is to set bit 3 for a right-moving conveyor, or bit 2 for a left-moving conveyor. (It isn't obvious? All will be revealed...) Notice again that both bits cannot both be set at the same time. Figure 1 shows the remainder of the assignments, which are completely arbitrary.

As far as Roland's physical speeds go, I chose 1 and -1 to be the horizontal (x) velocities, for left and right respectively. The vertical velocity is somewhat more complicated because we have to consider the force of gravity. If Roland is on solid ground, obviously vy is zero. If there is nothing underfoot, Roland has to fall, and I've done this by arranging for his y velocity to increase by 1 unit each time round the program loop (Negative velocity representing upwards). Note how this makes Roland fall faster and faster until he hits something: we can use the speed of impact to determine how far Roland has fallen and, if necessary, kill him from the impact.

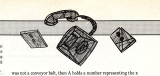
Jumping is now easy. If the player presses the joystick button, make vy -2; this represents an upward leap. The gravity we've programmed in automatically makes the jumping Roland reach a peak and fall back to the ground. Of course, while Roland is jumping or leaping, the joystick is ignored because his legs can't do anything while he's in mid-air.

We are now in a position to calculate Roland's true x and y velocities based on,

1) what he's standing on;
2) what the player is doing with the joystick.

Listing 1 shows the chunk of machine code I've called rolacc, which works out Roland's acceleration and stores the required values of vx and vy (prefixed r because they're Roland's velocities).

First, the program checks the map to see what's underfoot. The B and C registers are loaded with the current values of ry and rx (Roland's position), and B increased by 1 (coordinates increase downwards). Because we're using a system similar to the PET screen PEEK I described above, the subroutine to examine a map location is also called peek. It first CALLs another subroutine, locate, which converts the coordinates in BC to the right memory byte to look at. I've decided to make the playing area 40 columns wide by 16 deep, with the top left corner having coordinate 1,1. (This makes the playing area the whole screen wide, and gives a space at the bottom for the 'time remaining' display, number of lives left and so on). Obviously the required byte is (map + 40*row + column - 41), the -41 being required because the origin is 1,1, not 0,0.



The routine peek then just fetches the contents of the location into the A register and returns. We're going to need to alter the contents of the map as the game progresses, so I've included the analogous routine to store the A register at a given row/column, and called it poke for obvious reasons.

If Roland's in mid-air, the program jumps to deal with the 'fall'. Notice that we have to remove bits 0,1, and 4 before deciding if a location is 'empty', because Roland is able to move onto treasure, death objects or the exit. As described above, fall simply leaves rvx as it was, and increases rvy by 1.

Otherwise, the type of ground is saved on the stack and the joystick state fetched. The 'ground' is popped into the D register, HL pointed to the variable rvy, and B loaded with zero velocity. If the fire button hasn't been pressed, this is the correct value for rvy, so the program jumps to store it. Otherwise rvy is made equal to a jump value of -2.

Next, the ground and joystick states (D and A) are combined by ORing them together. This is best explained by reference to Figure 2, which shows all the possible starting combinations and results for bits 2 and 3, which we then mask off with AND &C. If the ground was not a conveyor belt, then A holds a number representing the x velocity: 0 for stationary, &8 for right, &4 for left. Furthermore, the F register parity bit will be even for zero velocity, and odd if an x velocity exists (because only bit 2 or bit 3 can be set, never both). If the ground were a conveyor belt and the joystick neutral, the same state applies. But if the joystick direction opposed the conveyor belt, both bits 2 and 3 are set to 1. Two bits set to 1 means even parity: thus the parity test can be used to choose the correct x velocity, which is zero. Of course a zero in A also passes the 'parity even' test, but since the result is to zero A, the x velocity is still the right value.

The various tests following this piece of bit-manipulation trickery simply put the right x velocity into B, and store it in rvx. The program then jumps over the fall routine and into the new position part of the program, which I'll tackle next month.

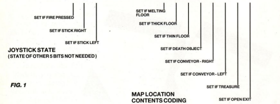


FIG. 1

IF BYTE IS ZERO, MAP LOCATION IS EMPTY SPACE.
IF BIT 7 IS SET, THEN ABOVE SCHEME IS OVERRIDDEN.
BITS 2-6 ARE SET, AND BITS 0-1 FORM A TWO-BIT COUNTER IMPLICATING HOW MUCH THE FLOOR HAS MELTED. THE LOCATION CONTENTS ARE INCREMENTED EACH TIME ROLAND STEPS ON THE LOCATION, AND AFTER FOUR COUNTS THE BYTE CLOCKS ROUND TO ZERO (SPACE); IE. THE FLOOR HAS COMPLETELY MELTED.

FIG. 2

BITS 3&2 OF GRAND TYPE (D)	00	00	00	01	01	01	10	10	10
BITS 3&2 OF JOYSTICK STATE (A)	00	01	10	00	01	10	00	01	10
A OR D	00	01	10	01	01	11	10	11	10
PARITY	EVEN	ODD	ODD	ODD	ODD	EVEN	ODD	EVEN	ODD
BYTE VALUE AFTER PARITY TEST & JUMP	00	01	10	01	01	00	10	00	10
VX (0 = STATIONARY) (-1 = LEFT) (1 = RIGHT)	0	-1	-1	-1	-1	0	1	0	1

(ALL OTHER BITS IN BYTE RESET TO ZERO BY AND &C INSTRUCTION)

Roland Takes a Running Jump

PART 3

This month it's one small leap for Roland, one giant program for mankind, as we hit you with all the remaining machine code in one go. Bytes by Keith Wilson and Marcus Sharp. Words by Peter Green.

The editor (all fall on bended knee) was on the phone. "This Roland series, can I have a program in the next part that actually does something?" The only permissible answer to questions like that is "Yes, sir," so we're abandoning the piecemeal, one-routine-at-a-time approach and presenting all of the remaining machine code in one go. The advantage of this is that people who prefer fun to information can just type in the rest of the listing and start leaping. The rest of you will have rather a lot to plough through, but "Running Jump" is neatly written in well-defined blocks, like all good machine code programs, so it should be easy to follow what's happening. Everyone involved in the game is very pleased with the final result, and we're sure you will be too. Why, even the editor is happy!

The Listings List

We have several chunks of program to consider this month. If you want to type-it-in-and-go, you will be interested in Listings 1,2 and 3. Listing 1 is the short BASIC program to load and run the machine code routines. Listing 2 is the BASIC hex loader for the machine code program, while Listing 3 is a room designer, for those among you who feel creative and want to produce your own customised version. It uses some of the routines from the main game (so that has to be loaded into the computer for the program to work), and allows you to paint in floors and walls of the various types wherever you wish, define the starting points for Roland and the two monsters, the monsters' paths, a title and the ink colours for the screen, and the level number. The resulting data file is called SCREENS.BIN and is used by the ROLAND program. The designer program is very easy to use, but actually producing a screen which is challenging, but not impossible, is something that will require a lot of practice. The nine screens provided here should give you some ideas.

Next month we will print Listings 4 and 5. Listing 4 is a set of nine feindishly difficult screens which show what can be done with the designer. Listing 5 is the source code for the machine code program, for those readers who are interested in how it all works.

The Running, Jumping, Standing Still Show

To produce a working version of the game, first type in Listing 1 and save it to tape or disc under the filename "ROLAND". Then type in Listing 2 and run it: this generates the machine code program and saves it under the filename "GAME.BIN". For this month you will need to work with screens you have designed yourself. Type in listing 3. Save it on a seperate tape then use it to create a file called "SCREENS.BIN". This must go on the tape after "ROLAND" and "GAME.BIN". Tape users must have the files on cassette in that order, but obviously this is irrevelant for disc users.If you now load and run "ROLAND", it will set HIMEM to reserve sufficient space for the machine code, and load the two binary files. Then you're asked which level you wish to start at: type in a number from 1 to 9. After setting up the required tone and volume envelopes for the sound effects and setting a rapid ink flash speed for the death routine, the main machine code program is called.The game itself then follows the classic pattern. On each level you must collect five treasures (3" floppy discs) and then make your way to the exit to the next level (the blocks with crosses). To make (programming) life simple the conveyor belts are not animated, but they do have arrows on them to indicate the direction of travel. Melting floors . . . well, they melt . . . and don't touch the deadly plants! Finally, the large animated monsters represent the twin banes of an editor's life: the telephone, and the man in the MD's chair. You've heard of the Stay-Puft Marshmallow Man from Ghostbusters? Here's the Sugar Monster . . .

Once you've died five times (and believe me you will), control returns to the BASIC program so you can play again. Pressing the small ENTER key will perform GOTO 30, running the program again from the same level. To start from another level, type GOTO 20 and press the large ENTER key. To stop during the game, press ESC and you'll drop back into BASIC: but there's no pause facility (evil, aren't we!).

Designs On You

The screen designer can be used to make small changes to the existing screens, or start from scratch with your own ideas. Note that it also requires two of the routines from the main game, which it loads in line 30. Your various options are selected by single letter keypresses as shown in Table 1.

LOAD loads in an old set of screens that you want to edit. When you're happy with the screen data, SAVE saves it under the filename SCREENS.BIN ready to use with the main game. You could keep a whole set of different screen files on tape, changing to the required cassette after the main game has loaded.Both the current graphic to plot (see Table 2) and the current level to edit, are selected by pressing keys 0 to 8 (the program adds one to the current number to set the level range from 1 to 9 as required), First select and display the level you want to edit by pressing its number (minus 1), then P. This displays the current design of that level.The cursor keys move the cursor around the playing area: pressing the COPY key places the currently-selected graphic at the cursor position. Alter the current graphic as required by pressing keys 0 to 8. To save time, it's possible to leave a TRAIL of the current graphic behind the cursor as you move it: this feature is toggled on and off by pressing T.

Each level can be displayed in its own set of colours, which are chosen by selecting the INKS option and then entering your chosen colours in response to the prompts. You have to enter two numbers, (separated by a comma) for each prompt, and you are strongly recommended to have both numbers the same (eg 16,16), which gives a non-flashing colour. Putting in two different numbers gives a flashing ink, and

bearing in mind that the initial BASIC control program makes the flash speed extremely fast, this is only recommended to fans of eyestrain. Nevertheless the option is provided in case someone can come up with a good use for it. Also, INK 1, which makes up most of the Roland character, is always white, so you should choose a darkish colour for the backgound, INK 0, otherwise you won't be able to see him!If you get in a complete mess and want to start again, the currently displayed screen can be wiped clear by pressing *.

Once the background is to your liking, you need to place the moving characters on it. Roland is simple enough: you simply move the cursor to the place where you want his head to be at the start of the level (remember that Roland is TWO characters high), then press @. The monsters are slightly more tricky. You are allowed per level, and the following procedure is done for each. Move the cursor to the position where you want the TOP LEFT corner of the monster graphic to start (remember that each monster is two characters high and two characters wide). Now press M.

Following the prompts, enter the monster number (1 or 2), the monster graphic number (1 for the Sugar Monster, 2 for the telephone), the amount of movement (ie the initial velocity) in the X and Y directions (this is -1 for left or up, +1 for right or down, 0 for no movement in that plane) and the distance in characters that the monster travels before reversing direction. You will need to be careful here to make sure you don't try and move the monster off the screen. In the game proper, setting the X velocity zero will make the monster move up and down only, setting Y zero will make it move left and right only, while if both velocities are non-zero the monster will move diagonally. Of course if both velocities are zero the monster will stay still, but that's not much fun, is it?Finally, choosing the NAME option lets you assign a silly name for the current level, up to 40 characters long.

TABLE 1

0-8	Select current number (for level to edit or block to display).
Cursor keys	Move cursor.
P	PRINT; displays the level corresponding to the current number and allows it to be edited.
COPY	Place current number graphic at cursor position.
T	TRAIL; leave a trail of the current number graphic behind cursor (toggles on/off).
I	INKS; allows selection of colours for INKs 0, 2 and 3.
*	CLEAR; wipes currently displayed screen clear.
N	NAME; allows entry of the title string for current level.
@	Sets start position of Roland's head at cursor for current level.
M	MONSTER; allows monster data to be entered for current level.
L	LOAD; load a SCREENS.BIN file into screen editor.
S	SAVE; saves current screens as file SCREENS.BIN.

TABLE 2

Current Number	Graphic
0	Exit
1	Treasure (disc)
2	Conveyor left
3	Conveyor right
4	Deadly plant
5	Thin floor
6	Thick floor
7	Melting floor
8	Empty space

Listing 1

```
10 MEMORY &3FFF:LOAD "game",&8000:LOAD "
   screens",&4000
20 MODE 1:INPUT "Which start screen";lev
   el
30 POKE &8392,level
40 ENT -1,1,-2,1:ENV 1,15,-1,4:SPEED INK
    2,2
50 CALL &8354:MODE 1:KEY 11,"GOTO 30"+CH
   R$(13):PRINT "PRESS THE SMALL ENTER K
   EY TO PLAY AGAIN...":PRINT
```

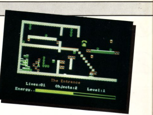
Listing 2

```
10 MEMORY &7FFF:ln=1000
15 PRINT"Please Wait"
20 FOR n=0 TO 783:POKE n+&8000,0:NEXT:PR
   INT n
30 FOR col=0 TO 7:READ a$
40 IF a$="**" THEN 120
50 v=VAL("&"+a$)
60 z(col)=z(col)+v
70 chk=chk+v
80 POKE &8000+n,v:n=n+1:NEXT
90 READ A:Z(8)=Z(8)+a
95 ln=ln+10:PRINT"*";
100 IF chk<>a THEN PRINT"Error in line "
    ;ln:STOP
110 chk=0:GOTO 30
120 FOR col=0 TO 8:READ chknum:IF chknum
    <>z(col) THEN PRINT"Error in column
    data" ELSE NEXT
130 PRINT"All OK"
140 SAVE"game.bin",b,&8000,&940
150 END
1000 DATA 00,00,00,4C,69,76,65,73, 515
1010 DATA 3A,30,4F,62,64,65,63,74, 705
1020 DATA 73,3A,4C,65,76,65,6C,3A, 735
1030 DATA 45,6E,65,72,67,79,2E,2E, 710
1040 DATA 01,01,01,30,01,00,0F,1E, 97
1050 DATA 00,04,01,00,64,00,00,0F, 120
1060 DATA 3C,00,07,00,00,00,09,0F, 219
1070 DATA 0E,C8,00,07,00,01,30,01, 399
1080 DATA 00,0E,96,00,CD,91,83,CD, 850
1090 DATA DD,83,06,0E,CD,19,BD,10, 887
1100 DATA FB,CD,2D,84,3A,00,83,D6, 1047
1110 DATA 03,0C,42,84,CD,0C,85,CD, 1200
1120 DATA 23,86,CD,2D,84,CD,19,BD, 970
1130 DATA CD,3D,84,CD,E5,85,CD,7D, 1295
1140 DATA 84,CD,3D,84,CD,BF,86,3E, 1074
1150 DATA 42,CD,1E,00,28,CC,C3,00, 927
1160 DATA 00,3E,01,32,00,00,3E,04, 494
1170 DATA 32,0E,83,21,79,88,22,11, 536
1180 DATA 83,3E,01,CD,0E,BC,01,00, 602
1190 DATA 00,CD,38,BC,3E,01,01,1A, 539
1200 DATA 1A,CD,32,BC,3E,01,CD,90, 881
1210 DATA 00,DD,21,13,83,06,07,21, 637
1220 DATA 17,05,CD,11,84,06,00,21, 443
1230 DATA 17,11,CD,1F,84,06,06,21, 453
1240 DATA 17,1E,CD,1F,84,06,00,21, 468
1250 DATA 19,01,C3,1F,84,AF,32,00, 622
1260 DATA 83,32,0C,83,32,00,83,3E, 578
1270 DATA FE,32,0F,83,3E,02,32,10, 500
1280 DATA 83,21,99,88,22,11,83,CD, 840
1290 DATA CF,86,0E,FF,21,90,C7,11, 1003
1300 DATA 00,40,CD,47,BC,CD,F9,86, 1124
1310 DATA 34,07,00,32,07,83,3A,0D, 452
1320 DATA 00,32,00,83,3E,01,CD,90, 729
1330 DATA 00,CD,2D,84,C3,3D,84,CD, 1162
1340 DATA 75,00,DD,7E,00,CD,5A,00, 1133
1350 DATA DD,23,10,F6,C9,2A,01,00, 890
1360 DATA CD,1A,BC,ED,5B,11,83,01, 896
1370 DATA 10,02,C3,64,84,FD,21,03, 734
1380 DATA 00,CD,48,84,FD,21,09,00, 960
1390 DATA 11,40,00,21,09,00,AF,ED, 847
1400 DATA 52,FD,46,05,19,10,FD,EB, 939
1410 DATA FD,66,00,FD,6E,01,CD,1A, 950
1420 DATA BC,01,10,04,C5,E5,1A,AE, 835
1430 DATA 77,23,13,10,F9,E1,01,00, 664
1440 DATA 00,09,30,04,01,50,C0,09, 351
1450 DATA C1,0D,20,E8,C9,21,10,83, 851
1460 DATA 35,C0,36,02,3A,0F,83,5F, 600
1470 DATA 16,00,21,40,00,19,EB,21, 412
1480 DATA 01,00,01,06,00,3E,00,CD, 275
1490 DATA 62,BC,21,0F,83,35,C0,CD, 915
1500 DATA 53,85,3A,02,00,41,3A,01, 542
1510 DATA 00,47,04,04,CD,39,85,E6, 832
1520 DATA EC,28,73,CB,7F,28,1B,CD, 993
1530 DATA 59,85,E6,7F,FE,7F,28,66, 1070
1540 DATA CD,FF,C6,04,77,F5,61,68, 1225
1550 DATA CD,1A,BC,EB,F1,F5,CD,7A, 1467
1560 DATA 87,F1,F5,3A,0C,83,D6,05, 1041
1570 DATA 30,79,AF,32,0C,83,CD,24, 778
1580 DATA 00,01,21,09,83,06,00,CD, 778
1590 DATA 67,28,11,F5,E5,21,30,83, 846
1600 DATA CD,AA,BC,3E,04,32,00,83, 821
1610 DATA E1,F1,06,FF,70,5F,02,CB, 1315
1620 DATA 7A,28,01,70,E6,0C,21,0A, 571
1630 DATA 83,06,00,EA,24,85,06,01, 547
1640 DATA 11,99,00,ED,53,11,83,FE, 1028
1650 DATA 00,28,09,06,FF,11,79,88, 592
1660 DATA ED,53,11,83,70,C9,3E,01, 844
1670 DATA 32,09,83,3A,00,83,A7,C0, 749
1680 DATA AF,32,0A,83,21,0C,83,34, 594
1690 DATA C9,CD,43,85,7E,C9,CD,43, 1285
1700 DATA 85,77,C9,F5,C5,11,28,00, 952
1710 DATA 21,E7,7F,04,19,10,FD,09, 698
1720 DATA C1,F1,C9,CD,2D,84,E1,E1, 1467
1730 DATA 21,42,83,CD,AA,BC,3E,01, 856
1740 DATA 01,19,01,CD,32,BC,06,04, 656
1750 DATA CD,19,BD,10,FB,3E,01,01, 750
1760 DATA 1A,1A,CD,32,BC,AF,32,0C, 732
1770 DATA 83,3A,0E,83,A7,28,04,3D, 612
1780 DATA 32,0E,83,CD,CF,86,C3,57, 1023
1790 DATA 83,C3,0E,83,3A,04,83,A7, 965
1800 DATA 28,31,FE,01,21,02,00,7E, 633
1810 DATA 20,16,FE,27,28,25,E5,4E, 731
1820 DATA 0C,3A,01,00,47,CD,00,86, 793
1830 DATA E1,CB,77,20,16,34,18,13, 696
1840 DATA A7,28,10,E5,4E,0D,3A,01, 602
1850 DATA 00,47,CD,00,86,E1,CB,77, 1269
1860 DATA 20,01,35,3A,09,83,47,3A, 613
1870 DATA 01,00,00,F5,47,3A,02,00, 761
1880 DATA 4F,CD,00,86,CB,77,C1,20, 1149
1890 DATA 06,78,32,01,00,21,00,83, 478
1900 DATA 7E,A7,C8,35,C9,FD,21,03, 1036
1910 DATA 00,21,07,83,CD,F6,85,FD, 1136
1920 DATA 21,09,00,21,00,83,FD,7E, 721
1930 DATA 00,47,FD,7E,02,00,FD,77, 952
1940 DATA 00,FD,7E,01,47,FD,7E,03, 833
1950 DATA 00,FD,77,01,35,C0,FD,7E, 1125
1960 DATA 04,77,FD,7E,02,ED,44,FD, 1062
1970 DATA 77,02,FD,7E,03,ED,44,FD, 1061
1980 DATA 77,05,C9,3A,02,00,4F,3A, 648
1990 DATA 01,00,47,CD,00,86,21,0D, 769
2000 DATA 83,CD,4F,C4,60,86,CB,67, 1145
2010 DATA C4,53,85,CB,47,C0,7E,FE, 1266
2020 DATA 05,C0,3A,00,00,3C,FE,0A, 707
2030 DATA 20,02,3E,01,32,00,00,E1, 500
2040 DATA 21,4B,83,CD,AA,BC,06,02, 938
2050 DATA CD,19,BD,10,FB,C3,57,83, 1099
2060 DATA 34,C5,CD,CF,86,21,39,83, 1016
2070 DATA CD,AA,BC,C1,CD,39,85,CB, 1354
2080 DATA 7F,20,06,CB,4F,28,02,18, 513
2090 DATA 0A,05,CD,39,85,CB,7F,C0, 932
2100 DATA CB,6F,C8,36,00,61,68,CD, 942
2110 DATA 1A,BC,EB,AF,C3,7A,87,2A, 1118
2120 DATA 05,00,CD,9C,86,2A,09,00, 805
2130 DATA CD,9C,86,C9,3A,02,00,0D, 1073
2140 DATA 28,03,2C,DD,C0,3A,01,00, 655
2150 DATA BC,C4,55,85,24,BC,CA,53, 1115
2160 DATA 85,25,3C,BC,CA,53,85,C9, 1037
2170 DATA CD,39,85,CB,7F,C4,CD,86, 1260
2180 DATA F5,04,CD,39,85,D1,CB,7F, 1183
2190 DATA C4,CD,86,02,C9,AF,C9,21, 1323
2200 DATA 17,0C,CD,75,00,3A,0E,83, 747
2210 DATA C6,30,CD,5D,00,21,17,19, 812
2220 DATA CD,75,00,3A,0D,83,C6,30, 957
2230 DATA CD,5D,00,21,17,24,CD,75, 899
2240 DATA 00,3A,00,00,C6,30,C5,5D, 907
2250 DATA 00,21,00,40,11,34,03,AF, 531
2260 DATA ED,52,3A,00,00,47,19,10, 617
2270 DATA FD,E5,DD,E1,AF,DD,4E,00, 1402
2280 DATA DD,46,01,CD,32,BC,DD,23, 991
2290 DATA DD,23,3E,02,DD,4E,00,DD, 840
2300 DATA 46,01,CD,32,BC,DD,23,DD, 991
2310 DATA 23,3E,03,DD,4E,00,DD,46, 690
2320 DATA 01,CD,32,BC,DD,23,DD,23, 956
2330 DATA DD,E5,E1,11,01,80,01,06, 828
2340 DATA 03,ED,00,E5,3E,02,CD,90, 1058
2350 DATA 00,21,15,01,CD,75,00,E1, 976
2360 DATA 06,28,7E,E5,C5,CD,5D,00, 1003
2370 DATA C1,E1,23,10,F5,21,0F,00, 890
2380 DATA 11,00,C0,01,F8,02,7E,CD, 789
2390 DATA 7F,28,03,3E,EF,77,CD,7A, 917
2400 DATA 87,23,13,13,00,78,01,20, 548
2410 DATA ED,C9,E5,C5,A7,28,0E,CB, 1288
2420 DATA 7F,06,00,28,03,2F,18,09, 256
2430 DATA 04,07,30,FC,78,07,07,07, 452
2440 DATA 07,4F,06,00,21,E9,87,09, 502
2450 DATA D5,D5,DD,E1,06,00,11,00, 903
2460 DATA 00,7E,DD,77,00,23,7E,DD, 856
2470 DATA 77,01,23,DD,19,10,F2,D1, 868
2480 DATA C1,E1,C9,3E,00,21,00,00, 714
2490 DATA F5,CD,1A,BC,F1,EB,C3,7A, 1457
2500 DATA 87,57,52,49,54,54,45,4E, 692
2510 DATA 20,42,59,20,40,45,49,54, 520
2520 DATA 48,20,57,49,4C,53,4F,4E, 580
2530 DATA 20,41,4E,44,20,4D,41,52, 499
2540 DATA 43,55,53,20,53,48,41,52, 569
2550 DATA 50,00,00,00,00,00,00,00, 80
2560 DATA 00,00,00,00,00,00,00,00, 0
2570 DATA 00,F0,F0,F0,F0,AA,AA,55, 1385
2580 DATA 55,CC,CC,33,33,CC,CC,CC, 1207
2590 DATA CC,F0,F0,1E,87,F0,F0,C3, 1524
2600 DATA 0F,F0,F0,0F,69,F0,F0,3C, 1155
2610 DATA 0F,0F,0F,0F,0F,00,10,40, 283
2620 DATA 20,20,40,10,00,0F,0F,0F, 317
2630 DATA 0F,55,00,22,AA,AA,CC,AA, 848
2640 DATA 99,44,AA,55,22,33,CC,11, 782
2650 DATA 00,0F,0F,0F,0F,F4,F4,F2, 790
2660 DATA F2,F1,F1,F2,F2,F4,F4,0F, 1711
2670 DATA 0F,0F,0F,0F,0F,F2,F2,F4, 805
2680 DATA F4,F8,F8,F4,F4,F2,F2,0F, 1727
2690 DATA 0F,78,E1,78,E1,0F,0F,0E, 749
2700 DATA 07,0E,07,0F,0F,1E,07,1E, 253
2710 DATA 87,F0,F0,05,DE,D5,0E,E6, 1649
2720 DATA 7E,E7,76,D7,0A,07,DC,F0, 1519
2730 DATA F0,00,CC,11,EE,77,EE,10, 1072
2740 DATA 60,10,E0,00,C0,01,0E,03, 546
2750 DATA C2,12,86,30,0E,33,EE,33, 748
2760 DATA EE,77,77,77,77,60,30,C0, 1082
2770 DATA 70,33,00,77,88,77,EE,60, 871
2780 DATA 80,70,80,30,00,07,00,54, 483
2790 DATA 0C,16,84,07,C0,77,CC,77, 807
2800 DATA CC,EE,EE,EE,EE,C0,60,E0, 1668
2810 DATA 70,00,07,0C,00,01,0C,00, 155
2820 DATA 00,03,00,0E,0C,05,0F,06, 74
2830 DATA 0E,06,00,0F,06,00,78,69, 290
2840 DATA 0F,0F,C0,90,00,0D,E0,90, 790
2850 DATA 0F,16,F0,F0,83,16,70,30, 830
2860 DATA 06,05,40,00,0E,01,00,C0, 280
2870 DATA 0C,00,00,01,00,11,CC,02, 252
2880 DATA 00,FF,FC,F1,EE,FF,FE,F1, 1736
2890 DATA FF,11,FF,FF,88,77,FF,FF, 1547
2900 DATA EE,FF,88,11,FF,FF,00,00, 1156
2910 DATA FF,FF,00,00,FF,44,00,00, 833
2920 DATA 00,22,00,00,00,44,22,22, 170
2930 DATA 00,88,77,FF,00,44,FE,F3, 1075
2940 DATA 88,88,ED,3D,88,55,CB,1E, 1024
2950 DATA CC,99,CB,1E,CC,55,ED,3D, 1177
2960 DATA CC,77,FE,F3,CC,11,FF,FF, 1551
2970 DATA CC,00,00,00,00,00,00,00, 204
2980 DATA 00,00,00,00,00,00,00,00, 0
2990 DATA **
3000 DATA 20948,19082,21617,20363,21748,
     19481,22334,19678,165251
```

Listing 3 - Room Designer

```
10 MODE 1:BORDER 0:INK 1,26:SPEED INK 2,
   2
20 WINDOW 1,40,1,21:WINDOW #1,1,40,22,25
   :MEMORY 16383
30 LOAD "game",32768:level=1:x=1:y=1:FOR
    n=16384 TO 22944 STEP 820
40 POKE n,0:POKE n+1,0:POKE n+2,15:POKE
   n+3,15:POKE n+4,18:POKE n+5,18:NEXT
50 tr=0:blk=6:LOCATE #1,1,1:PRINT #1,blk
60 POKE &8000,level:CALL &86F9:PEN 1:LOC
   ATE x,y:CALL &BB8A:adr=15584+level*82
   0
70 i$=LOWER$(INKEY$):IF i$="" GOTO 70 EL
   SE i=ASC(i$)
80 IF i=&E0 OR tr=1 THEN GOSUB 270
90 IF i=&F0 AND y>1 THEN CALL &BB8A:y=y-
   1:LOCATE x,y:CALL &BB8A
100 IF i=&F1 AND y<19 THEN CALL &BB8A:y=
    y+1:LOCATE x,y:CALL &BB8A
110 IF i=&F2 AND x>1 THEN CALL &BB8A:x=x
    -1:LOCATE x,y:CALL &BB8A
120 IF i=&F3 AND x<40 THEN CALL &BB8A:x=
    x+1:LOCATE x,y:CALL &BB8A
130 IF i>47 AND i<57 THEN blk=VAL(i$):LO
    CATE #1,1,1:PRINT #1,blk
140 IF i$="t" THEN tr=1-tr
150 IF i$="i" GOTO 300
160 IF i$="@" THEN POKE adr-14,y-1:POKE
    adr-13,x-1
170 IF i$="p" THEN level=blk+1:GOTO 50
180 IF i$="s" THEN WINDOW SWAP 0,1:CLS:S
    AVE "screens",b,16384,7380:GOTO 260
190 IF i$="l" THEN WINDOW SWAP 0,1:CLS:L
    OAD "screens",16384:GOTO 260
200 IF i$="*" THEN CALL &BB8A:FOR n=adr
    TO adr+799:POKE n,0:NEXT:GOTO 50
210 IF i$="m" GOTO 330
220 IF i$<>"n" GOTO 70
230 CALL &BB8A:FOR n=adr+760 TO adr+799:
    POKE n,32:NEXT:WINDOW SWAP 0,1:CLS
240 LINE INPUT n$:n$=LEFT$(n$,40):s=adr+
    760+(40-LEN(n$))\2
250 FOR n=s TO s+LEN(n$)-1:POKE n,ASC(MI
    D$(n$,n-s+1,1)):NEXT
260 CLS:WINDOW SWAP 0,1:GOTO 50
270 IF blk=0 THEN a=0 ELSE IF blk=7 THEN
     a=239 ELSE a=2↑blk
280 POKE adr-41+x+y*40,a:POKE &8784,a:PO
    KE &8786,y-1:POKE &8787,x-1
290 CALL &8785:CALL &BB8A:RETURN
300 CALL &BB8A:WINDOW SWAP 0,1:CLS:INPUT
     "INK 0";a,b:POKE adr-20,a
310 POKE adr-19,b:INPUT "INK 2";a,b:POKE
     adr-18,a:POKE adr-17,b
320 INPUT "INK 3";a,b:POKE adr-16,a:POKE
     adr-15,b:GOTO 260
330 WINDOW SWAP 0,1:CLS:INPUT "Monster n
    o.";mn:INPUT "Which graphic";mg
340 INPUT "X movement";mx:INPUT "Y movem
    ent";my:INPUT "Distance to move";md
350 s=adr-(3-mn)*6:POKE s,x-1:POKE s+1,y
    -1:POKE s+2,(256+mx) MOD 256
360 POKE s+3,(256+my) MOD 256:POKE s+4,m
    d:POKE s+5,mg:GOTO 260
```

Roland Takes a Running Jump



PART 4

Last month we produced the core of a game which ran and a program which allowed you to design screens for the game. Now it is time to see what makes 'Roland takes a running Jump' tick. Bytes by Keith Wilson and Marcus Sharp. Words by Peter Green.

Roland takes a running jump was written to include a set of nine sheets which make up a data file like the one you will have from the room designer program listed last month. We had intended to list these rooms this month but the program turned out to be so long that it would have taken over the magazine. As a compromise we have listed the source code, explaining how the machine code works and left the rooms out. If you want the original rooms then we will send you a copy of the listing. Just drop an SAE in the post to us and we will let you have a copy of a program to create the rooms.

Decoding the code

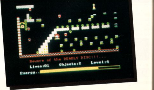
Normal people can stop reading now, and start typing: the rest of you masochists can carry on with the explanation of how the remaining machine code works. Listing 1 is the beast in question, and it is assembled at &8000: the data for all the levels is loaded at &4000. The total length of data for one floor is 820 bytes, consisting of the values for the three variable inks (six bytes), the start coordinates for Roland, two sets of monster data as described above, and the 760 bytes which hold the block map of the level.We now have a 40 by 19 play area, not 40 by 16. We managed to squeeze a bit more into the screen, which makes the design possibilities more versatile.

The entry point for the code is at line 400. We don't strictly need the ENT directive here, as the routine isn't self-running but is always called from the Basic program. Maxam users will have to drop it anyway because it isn't a recognised directive. They'll also need the dots in front of the labels print and end as these are reserved words.

The first set-up routine simply initialises the start level (this value is POKEd into place by the Basic program, allowing you to start at any point in the game), the number of lives (cheats can alter to suit), and the pointer to the data for the Roland sprite to be displayed (there are two, one facing left, the other right). The mode and border colour is set, INK 1 is set to white, the text pen is set to INK 1 and the four headings for the game information printed; that is, the number of lives left, the number of objects collected, the current level and the amount of energy Roland has left.

Each of these text strings is printed using the subroutine print, which is entered with HL containing the screen coordinates at which printing is to start, IX pointing to the first letter of the text, and B containing the number of letters to print. The routine just sets the text cursor to the HL value and prints B letters: nice and simple. The JP in line 930 means that the RETurn in line 1280, at the end of the print routine, acts as a return for the setup1 routine. If XXXX is a subroutine, then JP XXXX has the same effect as CALL XXXX: RET but saves a byte (unless you're doing naughty tricks with the stack that mess up the return addresses). Routine setup1 is only called once, at the start of the whole game.

At line 410 setup2 is called, and this is done at the start of each new level. It resets the counts for objects collected, and Roland's fall status (some exits are only accessible if you drop onto them). The energy level is reset to its maximum of 254, and the energy count to 2: we'll see what happens to these values later. The numerical values are printed out by the subroutine info at line 5130, which is also called at various times during the game to update these values as necessary (when an object is collected, for example). This routine takes the binary values of the variables (which are always in the range 0-9) and adds 48 to obtain the correct Ascii code for the digits 0-9. The firmware routine TXT WR CHAR is then used to print the numbers at screen positions set by TXT SET CURSOR.



Next, setup2 calls a firmware routine SCR FLOOD BOX to draw the rectangular strip that displays how much energy is left (line 1090). Then the routine is called that draws the background for this level onto the top 19 rows of the screen. This routine is called putbak and it starts at line 5320, but its action needs a lengthy explanation so we'll return to it later. After drawing the background, the counts for the path lengths of each monster are initialised from the monster data at m1 and m2 (lines 130-140). Finally the text pen is set to INK 1, and the Roland and monster sprites are drawn on-screen by the putrol and putmon routines. This concludes the setup2 routine.

Back to putbak

Now to cover the printing of the current background. First, putbak calculates where in the data table the 820 bytes for the current level can be found (lines 5320-5400). Then the values for the three variable INKs (0, 2 and 3) are obtained from the table and sent to the hardware using the firmware routine SCR SET INK (lines 5410-5570). This means each level can have its own colour scheme. Then the next 774 bytes of data are copied from the data table into the variable work area (lines 110-150). Next, the title for the screen is displayed, 40 characters (including leading and trailing spaces added by the screen designer program to centre it) which are printed underneath the playing area (lines 5650-5790).

Now the program has to place all the background graphics on the screen. It does this in a similar way to the sprite-drawing routine back in Part 1, by copying pre-encoded data directly into the screen RAM. Again, this allows multi-coloured characters to be displayed with the minimum of fuss. The screen data for the background blocks appears at lines 6570-6820 in the listing. The routine that puts a block on screen is called putblk (no kidding!) and starts at line 6010. It expects to find a block number in the A register, and the screen address of the top left-hand corner of the block in DE.

The action of putblk is as follows. After saving the HL and BC counters from putbak, the type of block is examined to calculate how far into the block data we have to index to find the right graphic. If the block is zero, representing empty space, we can skip the next bit of program because the index is zero (first entry in the block table). Otherwise, we check bit 7 and if it is not set, the floor isn't of the melting type. So we now go into a loop starting at bk_lp1, rotating the A register to the left and incrementing B (initially zero) until the carry is set. As you'll remember from Fig. 1 last month ordinary objects are represented by a single bit set in the block code. So once this loop terminates, B contains a number corresponding to the position of the block in the block table. For example, B is 2 for a thick floor, because a thick floor has the second bit from the left set.

Now B is put into the A register and multiplied by 16, because each graphic in the block table contains 16 bytes. At bk_lp3 this index value is moved into BC as a 16-bit unsigned number and added to the start of the block table in HL, to give the actual address for the graphic data. Finally the code at lines 6220-6350 transfer the graphic data into the screen, before restoring the saved registers and returning.

Suppose, however, that the block was a melting floor. Here we do something different, and slightly at variance with the information given in the December issue in Fig. 2. If bit 7 is set, it does mean that the floor is melting, but also that the whole block number is the complement (inverse) of an offset into the block graphic lookup table (that is, the offset which we had to calculate for all the other 'normal' blocks above). Line 6080 recovers this offset with a CPL command (this inverts all the bits in the A register), then jumps straight to the absolute address calculation at bk_lp3, line 6180.

The point of all this is that an 'unmelted' melting floor is stored with a character value of 239, &EF, or %11101111, which inverts to %00010000, or 16. This is the correct offset into the block character table for the melting floor graphic, so the whole graphic block gets printed. However, for each time period that Roland stands on a melting floor, 4 is added to the block number in the fldat work area. So the block numbers for a melting floor cycle through 239, 241, 245, 249 and 255. The inverse of these gives 16, 12, 8, 4 and 0, so as the floor melts, the table offset decreases: thus the block gets drawn as partly the empty space (the character before it in the table), and partly the melting floor graphic. This gives the effect of the floor gradually sinking, and when it is finally gone (value 255), the offset is zero, the whole of the space graphic is printed, and the floor is now just an empty space. Neat though this is, it does slightly complicate the 'see what's there' routines as both 0 and 255 represent empty air.

One last small point: Lines 5840-5870 aren't really necessary. They're there as a leftover from an early version of the screen designer program, which stored the block data as powers of two only. For the above system to work, a virgin melting floor block had to be converted from 128 to 239 in the fldat map. If you look at the final version of Designer (listed last month), line 270 does this conversion too.

The game of the name

And so to the main game loop for Roland (lines 420-610). First off, we count 14 TV frames to slow the game down to manageable speeds, then erase Roland (putrol does this job too because we're EXORing the screen: see Part 1 of the series). Next we look at a variable, jmpcnt. Roland's jump is split into two parts: the parabolic curve as he leaps into the air and falls back to the same level, and thereafter a vertical plunge (x velocity zero). This is not textbook physics but makes programming easier! The curved part is measured by jmpcnt: if Roland is moving up, there's no point in CALLing the joystick input and 'fallen to his death' routine (line 480 checks this). If Roland is moving downwards or on the level, this routine checks out what's underfoot and the state of the joystick, adjusts Roland's velocities accordingly (as we discussed last time), and kills Roland if he's fallen too far onto a solid surface.

If Roland isn't dead, the rest of the loop is straightforward. In turn, the program: moves Roland's coordinates; checks if Roland has hit a disc, the exit, a plant or one of the monsters, and if so deals with it; puts Roland back on screen at his new position; for smoothness, waits for the next flyback; erases the monsters; moves them on their predetermined paths; decreases the energy bar (killing Roland if time has run out); replaces the monsters in their new positions; checks if Roland has hit a monster (killing him if so); tests the ESC key and quits if it's pressed; and then loops back to the label 'game' to do it all again. Now we'll examine each of the routines mentioned above.

The fall guy

The cunning part of the rolacc routine (lines 1980-2760) was discussed in December. The additional lines you see here deal with melting floors and falling. Melting floors are handled by lines 2090-2240. First, if the floor block character from fldat is all 1s, ie wholly melted, it counts as empty space and a jump is made to the falling routine at line 2670. Otherwise the value of the A register is altered (by 4) to represent the next stage of the melting process, this value replaced in the fldat table, and the putblk routine used to draw the new floor graphic on screen.

If Roland is standing on solid ground, lines 2260-2300 check if he got there by falling more than four character heights. If he did, the death routine is invoked because he's broken his neck! Otherwise the fall counter is reset to zero. Finally, if Roland is dropping in free fall, the routine at line 2670 is used. The fall velocity is set at 1 (this is not the system suggested in part 2, with an accelerating drop, but makes the programming much easier). If Roland is falling back from the peak of his jump towards the level he started from, the routine quits here (so Roland also travels left or right if he was moving that way when he jumped). Otherwise, Roland is on the straight, vertical section of his fall, so the x velocity is zeroed in line 2730 and the fall count incremented.

Move it all about

The Roland sprite movement routines at 3270-3780 are quite straightforward. Checks are made to prevent Roland moving off the right or left edges of the screen (lines 3300-3350 and 3470-3480). Then, the routine labelled 'find' is called to check the contents of the two locations Roland is attempting to move onto (as Roland is two characters high, remember): this is done at lines 3360-3460 and 3490-3580 for right and left movement respectively. Roland isn't allowed to move onto a solid block, or 'thick floor' as we've called it, which is indicated by bit 6 of the fldat entry being set to 1.

Finally, if the variable jmpcnt isn't zero, showing that Roland is jumping, its value is decremented.

Check it out

The check routines are equally straightforward. The find routine is used again to see what Roland is now actually touching (again, he covers two character positions). If bit 1 is set, a disc is being touched: lines 4420-4660 increase the object count and displays the change in the information box, makes a suitable sound, checks whether the disc was under Roland's head or feet, and removes it from the corresponding location on the screen. If a deadly plant is being touched, the death routine is invoked without further ado. Finally, if the exit is being touched, the object count is checked and, if all five discs have been collected, the level is incremented (wrapping back to 1 if we're on Level 9 already). The victory sound is made, there's a long pause, and the program jumps back to gm_lp1 to set up things for the new level.

Mobile monsters

The monsters move in straight lines and are able to cross over any type of background character, so their movement routine at lines 3830-4080 is quite simple. Each monster has its x and y positions altered by the magnitude of its x and y velocities. The count (which is the length of the monster's path) is decremented and, if it has reached zero the monster is at the end of its path. The count is reset to its start value and the x and y velocities have their directions reversed (using NEG), which sends the monster back in the other direction.

Energy losses

Decene, another simple routine at lines 1760-1930, knocks a strip one pixel wide off the end of the energy bar every two passes round the overall game loop (ecount counts the number of passes). As the strip starts out 254 pixels long, each level must be completed within 508 game loops (about one and a half minutes). If the time runs out, Roland croaks.

Touch and go

Finally, lines 4680-4900 check whether Roland is overlapping either of the monsters. For each monster, the routine compares Roland's x coordinate with the monster's. If it isn't either the same, or one larger (remember Roland is only one character wide, the monsters are two characters), then Roland can't be overlapping and the routine exits. Otherwise, at line 4800 the y coordinates are retrieved and a similar comparison made. A match here means Roland and a monster have at least one character position in common, and a jump is made to the death routine.

Listing 1

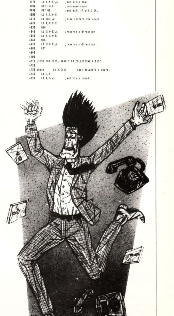
```
10 ; ****** ROLAND TAKES A RUNNING JUMP ************
20 ; ** Code by Keith Wilson & Marcus Sharp ********
30 ; *******************************************
40
50
60    ORG &8000
70
80 bakdat     EQU    &4000   ;start of floor data
90 baklen     EQU    820     ;length of data for one floor
100 level     DEFB   0       ;current floor number
110 ry DEFB   0
120 rx DEFB   0       ;Roland's coordinates
130 m1 DEFS   6       ;monster 1's data
140 m2 DEFS   6       ;monster 2's data
150 fldat     DEFS   760     ;screen work space
160 count1    DEFB   0       ;monster 1's count
170 count2    DEFB   0       ;monster 2's count
180 rvy       DEFB   0
190 rvx       DEFB   0       ;Roland's x and y velocities
200 jmpcnt    DEFB   0       ;count for jumps
210 falcnt    DEFB   0       ;characters Roland has fallen
220 objcol    DEFB   0       ;number of objects collected
230 lives     DEFB   0       ;Roland's lives left
240 energy    DEFB   0       ;Roland's energy
250 ecount    DEFB   0       ;count for energy
260 rolspr    DEFW   0       ;Roland's sprite address
270
280 text1     DEFM   "Lives:0Objects:Level:Energy.."
290
300                  ;table of sounds
310 sd.jmp    DEFB   1,1,1,48,1,0,15,30,0
320 sd.get    DEFB   4,1,0,100,0,0,15,60,0
330 sd.die    DEFB   135,0,0,0,9,15,14,200,0
340 sd.end    DEFB   135,0,1,48,1,0,14,150,0
350
360
370 ;MAIN GAME LOOP
380
390    ENT $
400    CALL setup1     ;set up variables for start of game
410 gm.lp1     CALL setup2     ;set up more variables
420 game       LD B,14         ;count to slow the game down
430 gm.lp2     CALL &BD19      ;MC WAIT FLYBACK - wait for a frame flyback
440    DJNZ gm.lp2     ;and again until count runs out
450    CALL putrol     ;erase Roland
460    LD A,(jmpcnt)
470    SUB 3
480    CALL C,rolacc   ;handle input if Roland's not going up
490    CALL move       ;move the sprites
500    CALL check      ;check for any special circumstances
510    CALL putrol     ;print new Roland
520    CALL &BD19      ;MC WAIT FLYBACK
530    CALL putmon     ;erase the monsters
540    CALL monmov     ;move the monsters
550    CALL decene     ;decrease energy
560    CALL putmon     ;print the monsters
570    CALL monchk     ;check for Roland hitting a monster
580    LD A,66         ;is the ESC key pressed?
590    CALL &BB1E      ;KM TEST KEY
600    JR Z,game       ;if not pressed, loop round
610 .end       JP &BB00        ;clear key buffer
620
630
640 ;SET UP VARIABLES FOR GAME
650
660 setup1     LD A,1
670    LD (level),A    ;set start level
680    LD A,4
690    LD (lives),A    ;set Roland's lives
700    LD HL,rollft
710    LD (rolspr),HL  ;set Roland's sprite address
720    LD A,1
730    CALL &BC0E      ;SCR SET MODE - select Mode 1
740    LD BC,&0000
750    CALL &BC38      ;SCR SET BORDER - to black
760    LD A,1
770    LD BC,&1A1A
780    CALL &BC32      ;SCR SET INK - ink 1 = white
790    LD A,1
800    CALL &BB90      ;TXT SET PEN - to ink 1
810    LD IX,text1     ;message text address
820    LD B,7
830    LD HL,&0517     ;data for 'Lives' text
840    CALL print      ;print text
850    LD B,8
860    LD HL,&1117     ;data for 'Objects' text
870    CALL print      ;print text
880    LD B,6
890    LD HL,&1E17     ;data for 'Level' text
900    CALL print      ;print text
910    LD B,8
920    LD HL,&0119     ;data for 'Energy >'
930    JP print        ;and print it
940
950 setup2     XOR A
960    LD (objcol),A   ;set objects collected to 0
970    LD (falcnt),A   ;reset fall count
980    LD (jmpcnt),A   ;and jump count
990    LD A,254
1000   LD (energy),A   ;set initial energy
1010   LD A,2
1020   LD (ecount),A   ;set energy count
1030   LD HL,rolrgt
1040   LD (rolspr),HL  ;set initial Roland sprite
1050   CALL info       ;print information
1060   LD C,255
1070   LD HL,&1800     ;set up registers for
1080   LD DE,&4000     ;size and colour of energy bar
1090   CALL &BC47      ;SCR FLOOD BOX - paint energy bar
1100   CALL putbak     ;print the background
1110   LD A,(m1+4)
1120   LD (count1),A   ;start monster 1's count
1130   LD A,(m2+4)
1140   LD (count2),A   ;and same for monster 2
1150   LD A,1
1160   CALL &BB90      ;TXT SET PEN - to ink 1
1170   CALL putrol
1180   JP putmon       ;plot some dummy sprites
```

```
1190
1200
1210 ;PRINT A MESSAGE, LENGTH B, POSITION H,L AND WITH TEXT AT IX
1220
1230 .print     CALL &BB75      ;TXT SET CURSOR - set cursor position
1240 pr.lp1     LD A,(IX)       ;get character
1250    CALL &BB5A      ;and print it
1260    INC IX          ;increase pointer
1270    DJNZ pr.lp1     ;and carry on until all done
1280    RET
1290
1300
1310 ;XOR ALL SPRITES TO SCREEN
1320
1330 putrol     LD HL,(ry)      ;get Roland's coords
1340    CALL &BC1A      ;SCR CHAR POSITION - get screen address
1350    LD DE,(rolspr)  ;sprite address
1360    LD BC,&0210     ;sprite size
1370    JP plot         ;print the sprite
1380
1390 putmon     LD IY,m1
1400    CALL monplt     ;XOR monster 1
1410    LD IY,m2
1420 monplt     LD DE,64        ;size of each graphic
1430    LD HL,montab    ;start of graphics
1440    XOR A
1450    SBC HL,DE       ;minus one graphic
1460    LD B,(IY+5)     ;get monster graphic number in B
1470 mn.lp1     ADD HL,DE       ;add offset for 1 graphic
1480    DJNZ mn.lp1     ;for each monster number
1490    EX DE,HL        ;put graphic address in DE
1500    LD H,(IY+0)     ;get x coord.
1510    LD L,(IY+1)     ;get y coord.
1520    CALL &BC1A      ;SCR CHAR POSITION - find screen address
1530    LD BC,&0410     ;width and height of monster
1540 plot       PUSH BC         ;save width and row counter
1550    PUSH HL         ;save address of current row start
1560 do.row     LD A,(DE)       ;get byte of graphic data
1570    XOR (HL)        ;XOR it with the contents of screen
1580    LD (HL),A       ;put new screen data back into screen
1590    INC HL          ;step screen address right one byte
1600    INC DE          ;point to next byte of graphics data
1610    DJNZ do.row     ;loop back until whole row plotted
1620    POP HL          ;fetch screen address of row start
1630    LD BC,&800      ;offset of 2048 bytes to next row
1640    ADD HL,BC       ;add offset to screen address
1650    JR NC,no.cor    ;no carry, so no correction needed
1660    LD BC,&C050     ;correction factor for wrap-around
1670    ADD HL,BC       ;add correction
1680 no.cor     POP BC          ;fetch width and row counter
1690    DEC C           ;decrement row counter
1700    JR NZ,plot      ;loop back until all rows plotted
1710    RET
1720
1730
1740 ;DECREASE ENERGY BAR AND CHECK IF TIME HAS RUN OUT
1750
1760 decene     LD HL,ecount    ;point HL to energy counter
1770    DEC (HL)        ;decrease it
1780    RET NZ          ;if still some time left, return
1790    LD (HL),2       ;recharge count
1800    LD A,(energy)
1810    LD E,A
1820    LD D,0          ;put energy in DE
1830    LD HL,64
1840    ADD HL,DE       ;get x position
1850    EX DE,HL        ;put it into DE
1860    LD HL,1         ;start y coord.
1870    LD BC,6         ;end y coord.
1880    LD A,0          ;encoded ink
1890    CALL &BC62      ;SCR VERTICAL - remove pixels from end of e-bar
1900    LD HL,energy    ;point HL to energy left
1910    DEC (HL)        ;decrease it
1920    RET NZ          ;if still energy left, exit
1930    CALL dead       ;else kill Roland
1940
1950
1960 ;MOVE ROLAND ABOUT
1970
1980 rolacc     LD A,(rx)       ;get Roland's x coord.
1990    LD C,A          ;put it into C reg
2000    LD A,(ry)       ;get Roland's y coord.
2010    LD B,A          ;put it in B
2020    INC B
2030    INC B           ;go down 2 squares
2040    CALL peek       ;see what's there
2050    AND &EC         ;lose irrelevant bits
2060    JR Z,fall       ;if zero, then Roland is falling
2070    BIT 7,A
```

```
2080    JR Z,mr.lp1     ;if not on a sinking floor, exit
2090    CALL peek       ;re-check position
2100    AND 127
2110    CP 127
2120    JR Z,fall       ;if floor has sunk, fall too
2130    SET 7,A         ;make floor melting again
2140    ADD A,4         ;and move it down
2150    LD (HL),A       ;then put this in the map
2160    PUSH AF
2170    LD H,C
2180    LD L,B          ;put position into HL
2190    CALL &BC1A      ;SCR CHAR POSITION - and find its address
2200    EX DE,HL        ;put it into DE
2210    POP AF
2220    PUSH AF
2230    CALL putblk     ;and print the new floor
2240    POP AF
2250 mr.lp1     PUSH AF         ;push 'ground' onto stack
2260    LD A,(falcnt)   ;get fall count
2270    SUB 5
2280    JR NC,dead      ;if fallen more than 4, die
2290    XOR A
2300    LD (falcnt),A   ;else reset count
2310    CALL &BB24      ;KM GET JOYSTICK - test the joystick
2320    POP DE          ;pop 'ground' off stack into D
2330    LD HL,rvy       ;point HL to y velocity
2340    LD B,0          ;assume zero velocity to start with
2350    BIT 4,A         ;fire button pressed?
2360    JR Z,newrvy     ;if not, load y velocity with zero
2370    PUSH AF
2380    PUSH HL
2390    LD HL,sd.jmp
2400    CALL &BCAA      ;SOUND QUEUE - else make sound
2410    LD A,4
2420    LD (jmpcnt),A   ;set count for the jump
2430    POP HL
2440    POP AF
2450    LD B,-1         ;set jump velocity
2460 newrvy     LD (HL),B       ;store new y velocity
2470    LD E,A          ;store joystick byte
2480    OR D            ;merge with conveyor belt bits
2490    BIT 7,D
2500    JR Z,notmlt     ;if not on melting floor, skip over
2510    LD A,E          ;if so, only use left & right
2520 notmlt     AND &C          ;mask off unwanted bits
2530    LD HL,rvx       ;point HL to Roland's x velocity
2540    LD B,0          ;assume zero velocity to start with
2550    JP PE,newrvx    ;and load this if parity is even
2560    LD B,1          ;else load B with right x velocity
2570    LD DE,rolrgt
2580    LD (rolspr),DE  ;set sprite address
2590    CP 8            ;check if Roland is moving right
2600    JR Z,newrvx     ;and store this if he is
2610    LD B,-1         ;else make B=left velocity
2620    LD DE,rollft
2630    LD (rolspr),DE  ;set sprite address
2640 newrvx     LD (HL),B       ;store new x velocity
2650    RET
2660
2670 fall       LD A,1
2680    LD (rvy),A      ;set velocity for falling
2690    LD A,(jmpcnt)   ;get jumpcount
2700    AND A
2710    RET NZ          ;exit if jumping
2720    XOR A
2730    LD (rvx),A      ;zero x movement
2740    LD HL,falcnt    ;fall count address
2750    INC (HL)
2760    RET
2770
2780 peek       CALL locate
2790    LD A,(HL)
2800    RET
2810
2820 poke       CALL locate
2830    LD (HL),A
2840    RET
2850
2860 locate     PUSH AF         ;save AF
2870    PUSH BC         ;save x,y coords
2880    LD DE,40        ;width of one floor
2890    LD HL,fldat-40  ;point to 1 floor before level data
2900    INC B
2910 addrow     ADD HL,DE       ;add on row length
2920    DJNZ addrow     ;once for each row
2930    ADD HL,BC       ;add column number
2940    POP BC          ;restore BC
2950    POP AF          ;and AF
2960    RET             ;HL points to byte in fldat table
```

```
2970
2980 dead       CALL putrol     ;plot Roland
2990    POP HL
3000    POP HL          ;clear stack
3010    LD HL,sd.die
3020    CALL &BCAA      ;SOUND QUEUE - make dead noise
3030    LD A,1
3040    LD BC,&0119
3050    CALL &BC32      ;SCR SET INK - flash ink 1
3060    LD B,100        ;count for pause
3070 dd.lp1     CALL &BD19      ;wait for 1/50th second
3080    DJNZ dd.lp1     ;and repeat until count runs out
3090    LD A,1
3100    LD BC,&1A1A
3110    CALL &BC32      ;SCR SET INK - reset ink 1
3120    XOR A
3130    LD (falcnt),A   ;reset fall count
3140    LD A,(lives)    ;get lives left
3150    AND A
3160    JR Z,vdead      ;if none, he is kaput!
3170    DEC A
3180    LD (lives),A    ;else store new number of lives
3190    CALL info       ;and print them
3200    JP gm.lp1       ;and start again
3210
3220 vdead      JP end          ;clear key buffer
3230
3240
3250 ;MOVE SPRITES ABOUT
3260
3270 move       LD A,(rvx)      ;get Roland's x velocity
3280    AND A
3290    JR Z,mv.lp1     ;if he is still, skip over
3300    CP 1
3310    LD HL,rx        ;pointer to x coord.
3320    LD A,(HL)
3330    JR NZ,mv.lp3    ;skip over if not going right
3340    CP 39
3350    JR Z,mv.lp1     ;if already at boundary, ignore
3360    PUSH HL
3370    LD C,(HL)       ;get x coord.
3380    INC C           ;plus 1
3390    LD A,(ry)       ;and y coord.
3400    LD B,A
3410    CALL find       ;find out contents of new location
3420    POP HL
3430    BIT 6,A
3440    JR NZ,mv.lp1    ;if solid, ignore
3450    INC (HL)        ;else increase x coord.
3460    JR mv.lp1
3470 mv.lp3     AND A
3480    JR Z,mv.lp1     ;if already at boundary, ignore
3490    PUSH HL
3500    LD C,(HL)       ;get x coord.
3510    DEC C           ;minus 1
3520    LD A,(ry)       ;and y coord.
3530    LD B,A
3540    CALL find       ;get contents of new location
3550    POP HL
3560    BIT 6,A
3570    JR NZ,mv.lp1    ;if solid, ignore
3580    DEC (HL)        ;else decrease x coord.
3590 mv.lp1     LD A,(rvy)      ;get Roland's y velocity
3600    LD B,A
3610    LD A,(ry)       ;and his y coord.
3620    ADD A,B         ;add them
3630    PUSH AF
3640    LD B,A
3650    LD A,(rx)       ;get x coord.
3660    LD C,A
3670    CALL find       ;find what is there
3680    BIT 6,A
3690    POP BC          ;get new y back in B
3700    JR NZ,mv.lp4    ;if solid, go and check count
3710    LD A,B
3720    LD (ry),A       ;else store the result
3730 mv.lp4     LD HL,jmpcnt    ;jump count address
3740    LD A,(HL)       ;get the jump count
3750    AND A
3760    RET Z           ;if it's zero, exit
3770    DEC (HL)        ;else decrease the count
3780    RET
3790
3800
3810 ;MOVE THE MONSTERS
3820
3830 monmov     LD IY,m1
3840    LD HL,count1
3850    CALL movmon     ;move monster 1
```

```
3860    LD IY,m2
3870    LD HL,count2
3880 movmon     LD A,(IY+0)     ;get x coord.
3890    LD B,A
3900    LD A,(IY+2)     ;and x movement
3910    ADD A,B         ;add them
3920    LD (IY+0),A     ;and store the result
3930    LD A,(IY+1)     ;get y coord.
3940    LD B,A
3950    LD A,(IY+3)     ;and y movement
3960    ADD A,B         ;add them
3970    LD (IY+1),A     ;and store that
3980    DEC (HL)        ;decrease count
3990    RET NZ          ;and exit if still OK.
4000    LD A,(IY+4)
4010    LD (HL),A       ;else restart the count
4020    LD A,(IY+2)
4030    NEG
4040    LD (IY+2),A     ;reverse x direction
4050    LD A,(IY+3)
4060    NEG
4070    LD (IY+3),A     ;reverse x direction
4080    RET
4090
4100
4110 ;TEST FOR EXIT, DEADLY OR COLLECTING A DISC
4120
4130 check      LD A,(rx)       ;get Roland's x coord.
4140    LD C,A
4150    LD A,(ry)       ;and his y coord.
```



```
4160    LD B,A
4170    CALL find
4180    LD HL,objcol    ;set objects collected pointer
4190    BIT 1,A
4200    CALL NZ,disc    ;if a treasure, go to routine
4210    BIT 4,A
4220    CALL NZ,dead    ;if a deadly object, die!
4230    BIT 0,A
4240    RET Z           ;exit if not an exit?!?
4250    LD A,(HL)       ;else get object collected count
4260    CP 5
4270    RET NZ          ;exit if not all collected
4280    LD A,(level)    ;else get level number
4290    INC A           ;increase it
4300    CP 10
4310    JR NZ,ch.lp1    ;if a legal level, store it
4320    LD A,1          ;else back to screen 1
4330 ch.lp1     LD (level),A    ;store the level
4340    POP HL          ;clear the stack
4350    LD HL,sd.end
4360    CALL &BCAA      ;SOUND QUEUE - make sound
4370    LD B,150
4380 ch.lp2     CALL &BD19      ;wait for a flyback
4390    DJNZ ch.lp2     ;until loop runs out
4400    JP gm.lp1       ;and use the new screen
4410
4420 disc       INC (HL)        ;increase objects collected
4430    PUSH BC         ;store coords.
4440    CALL info       ;print information
4450    LD HL,sd.get
4460    CALL &BCAA      ;SOUND QUEUE - make sound
4470    POP BC          ;restore coords.
4480    CALL peek       ;check bottom position
4490    BIT 7,A
4500    JR NZ,ds.s1     ;if melting floor, ignore
4510    BIT 1,A
4520    JR Z,ds.s1      ;if no treasure, ignore
4530    JR remchr
4540 ds.s1      DEC B           ;get head's position
4550    CALL peek       ;check top position
4560    BIT 7,A
4570    RET NZ          ;if melting floor, ignore
4580    BIT 1,A
4590    RET Z           ;if no treasure, ignore
4600 remchr     LD (HL),0       ;remove treasure from map
4610    LD H,C
4620    LD L,B          ;put coords. in H,L
4630    CALL &BC1A      ;SCR CHAR POSITION - get screen address
4640    EX DE,HL
4650    XOR A
4660    JP putblk       ;and remove the treasure from screen
4670
4680 monchk     LD HL,(m1)      ;get monster 1's coords.
4690    CALL moncol     ;check for collision with monster 1
4700    LD HL,(m2)      ;get monster 2's coords.
4710    CALL moncol     ;check for collision with monster 2
4720    RET
4730
4740 moncol     LD A,(rx)       ;get Roland's x coord.
4750    CP L
4760    JR Z,mc.s1      ;if same as monster's, go ahead
4770    INC L
4780    CP L            ;now test for being one along
4790    RET NZ          ;and exit if not the same
4800 mc.s1      LD A,(ry)       ;get Roland's y coord.
4810    CP H
4820    JP Z,dead       ;if same as monster's, die
4830    INC H
4840    CP H
4850    JP Z,dead       ;if same as monster's + 1, die
4860    DEC H
4870    INC A
4880    CP H
4890    JP Z,dead       ;if same as monster's-1, die
4900    RET
4910
4920
4950 ;FIND OUT WHAT ROLAND IS ON (NEEDS COORDS C,B)
4940
4950 find       CALL peek       ;get first byte
4960    BIT 7,A
4970    CALL NZ,zeroa   ;if melting floor, ignore
4980    PUSH AF         ;store it
4990    INC B
5000    CALL peek       ;get next byte
5010    POP DE          ;get old byte in D
5020    BIT 7,A
5030    CALL NZ,zeroa   ;if melting floor, ignore
5040    OR D            ;merge the two
5050    RET
5060
5070 zeroa      XOR A   ;make A=0
5080    RET
5090
5100
5110 ;PRINT LIVES, OBJECTS AND LEVEL AT BOTTOM OF SCREEN
5120
5130 info       LD HL,&0C17
5140    CALL &BB75      ;TXT SET CURSOR - set cursor for lives
5150    LD A,(lives)
5160    ADD A,48        ;convert to ASCII code
5170    CALL &BB5D      ;TXT WR CHAR - and print it
5180    LD HL,&1917
5190    CALL &BB75      ;TXT SET CURSOR - for objects
5200    LD A,(objcol)
5210    ADD A,48        ;convert to ASCII code
5220    CALL &BB5D      ;TXT WR CHAR - and print it
5230    LD HL,&2417
5240    CALL &BB75      ;TXT SET CURSOR - for level
5250    LD A,(level)
5260    ADD A,48        ;convert to ASCII code
5270    JP &BB5D        ;TXT WR CHAR - and print it
5280
5290
5300 ;PRINT THE CURRENT BACKGROUND
5310
5320 putbak     LD HL,bakdat    ;get start of floor data
5330    LD DE,baklen    ;get length for one floor's data
5340    XOR A           ;reset carry
5350    SBC HL,DE       ;find address of one floor before start
5360    LD A,(level)    ;get floor number
5370    LD B,A          ;use it as a count
5380 pb.lp1     ADD HL,DE       ;find address of next floor
5390    DJNZ pb.lp1     ;loop until at correct floor
5400    PUSH HL
5410    POP IX          ;put address in IX
5420    XOR A
5430    LD C,(IX+0)
5440    LD B,(IX+1)     ;get ink 0 colour
5450    CALL &BC32      ;SCR SET INK - and set it
5460    INC IX
5470    INC IX
5480    LD A,2
5490    LD C,(IX+0)
5500    LD B,(IX+1)     ;get ink 2 colour
5510    CALL &BC32      ;SCR SET INK - and set it
5520    INC IX
5530    INC IX
5540    LD A,3
5550    LD C,(IX+0)
5560    LD B,(IX+1)     ;get ink 3 colour
5570    CALL &BC32      ;SCR SET INK - and set it
5580    INC IX
5590    INC IX
5600    PUSH IX
5610    POP HL
5620    LD DE,ry
5630    LD BC,774
5640    LDIR            ;store other variables and working map
5650    PUSH HL
5660    LD A,2
5670    CALL &BB90      ;TXT SET PEN - set pen ink to 2
5680    LD HL,&0115
5690    CALL &BB75      ;TXT SET CURSOR -set cursor position
5700    POP HL
5710    LD B,40
5720 pb.lp4     LD A,(HL)       ;get character from the title string
5730    PUSH HL
5740    PUSH BC         ;save registers
5750    CALL &BB5D      ;TXT WR CHAR - print the character
5760    POP BC
5770    POP HL          ;restore registers
5780    INC HL          ;increase pointer
5790    DJNZ pb.lp4     ;and loop back until finished
5800    LD HL,fldat     ;start of map data
5810    LD DE,&C000     ;start of screen
5820    LD BC,40*19     ;size of screen
5830 pb.lp2     LD A,(HL)       ;get block type
5840    BIT 7,A         ;test if a melting floor
5850    JR Z,pb.lp3     ;jump if not
5860    LD A,239
5870    LD (HL),A       ;else change to correct block code
5880 pb.lp3     CALL putblk     ;print the block
5890    INC HL          ;point to next block
5900    INC DE
```

```
5910    INC DE          ;increase screen address
5920    DEC BC
5930    LD A,B
5940    OR C            ;check if BC=0
5950    JR NZ,pb.lp2    ;and loop back if not
5960    RET
5970
5980
5990 ;PUT BLOCK A TO SCREEN AT ADDRESS DE
6000
6010 putblk     PUSH HL
6020    PUSH BC         ;save counts
6030    AND A           ;check if block is a space
6040    JR Z,bk.lp2
6050    BIT 7,A         ;check for a melting floor
6060    LD B,0
6070    JR Z,bk.lp1     ;jump if not
6080    CPL             ;get offset for floor
6090    JR bk.lp3       ;and go and print it
6100 bk.lp1     INC B           ;increase count
6110    RLCA            ;rotate high bit into carry
6120    JR NC,bk.lp1
6130    LD A,B
6140 bk.lp2     RLCA            ;A contains table index
6150    RLCA
6160    RLCA
6170    RLCA            ;multiply by 16 (length of a block)
6180 bk.lp3     LD C,A
6190    LD B,0
6200    LD HL,blocks    ;start of block data
6210    ADD HL,BC       ;HL points to start of correct entry
6220    PUSH DE         ;save screen address
6230    PUSH DE
6240    POP IX          ;move it to IX register
6250    LD B,8          ;row count
6260    LD DE,&0800     ;displacement between rows
6270 bk.lp4     LD A,(HL)       ;fetch data byte
6280    LD (IX+0),A     ;plot to screen
6290    INC HL
6300    LD A,(HL)
6310    LD (IX+1),A     ;and next byte
6320    INC HL
6330    ADD IX,DE       ;move down a row
6340    DJNZ bk.lp4     ;and repeat until all printed
6350    POP DE
6360    POP BC
6370    POP HL
6380    RET
6390
6400
6410 ;PUT A BLOCK ON THE SCREEN (FOR USE BY THE ROOM DESIGNER)
6420
6430 txtput     LD A,0          ;set block type
6440    LD HL,&0000     ;set screen position
6450    PUSH AF
6460    CALL &BC1A      ;SCR CHAR POSITION - get screen address
6470    POP AF
6480    EX DE,HL        ;put address into DE
6490    JP putblk       ;and print the character
6500
6510
6520 DEFM       "WRITTEN BY KEITH WILSON AND MARCUS SHARP"
6530
6540
6550 ;BLOCK CHARACTER TABLE
6560
6570 blocks     DEFB &00,&00,&00,&00,&00,&00,&00,&00
6580    DEFB &00,&00,&00,&00,&00,&00,&00,&00    ;SPACE
6590
6600    DEFB &F0,&F0,&F0,&F0,&AA,&AA,&55,&55
6610    DEFB &CC,&CC,&33,&33,&CC,&CC,&CC,&CC    ;MELTING FLOOR
6620
6630    DEFB &F0,&F0,&1E,&87,&F0,&F0,&C3,&0F
6640    DEFB &F0,&F0,&0F,&69,&F0,&F0,&3C,&0F    ;THICK FLOOR
6650
6660    DEFB &0F,&0F,&0F,&0F,&80,&10,&40,&20
6670    DEFB &20,&40,&10,&80,&0F,&0F,&0F,&0F    ;THIN FLOOR
6680
6690    DEFB &55,&00,&22,&AA,&AA,&CC,&AA,&99
6700    DEFB &44,&AA,&55,&22,&33,&CC,&11,&00    ;DEATH OBJECT
6710
6720    DEFB &0F,&0F,&0F,&0F,&F4,&F4,&F2,&F2
6730    DEFB &F1,&F1,&F2,&F2,&F4,&F4,&0F,&0F    ;CONVEYOR-RIGHT
6740
6750    DEFB &0F,&0F,&0F,&0F,&F2,&F2,&F4,&F4
6760    DEFB &F8,&F8,&F4,&F4,&F2,&F2,&0F,&0F    ;CONVEYOR-LEFT
6770
6780    DEFB &78,&E1,&78,&E1,&0F,&0F,&0E,&07
6790    DEFB &0E,&07,&0F,&0F,&1E,&87,&1E,&87    ;TREASURE
6800
6810    DEFB &F0,&F0,&03,&0E,&05,&0E,&E6,&7E
6820    DEFB &E7,&76,&07,&0A,&07,&0C,&F0,&F0    ;OPEN EXIT
6830
6840 rollft     DEFB &00,&CC,&11,&EE,&77,&EE,&70,&60
6850    DEFB &10,&E0,&00,&C0,&01,&0E,&03,&C2
6860    DEFB &12,&06,&30,&0E,&33,&EE,&33,&EE
6870    DEFB &77,&77,&77,&77,&60,&30,&E0,&70    ;ROLAND FACING LEFT
6880
6890 rolrgt     DEFB &33,&00,&77,&88,&77,&EE,&60,&00
6900    DEFB &70,&00,&50,&00,&07,&08,&34,&0C
6910    DEFB &16,&04,&07,&C0,&77,&CC,&77,&CC
6920    DEFB &EE,&EE,&EE,&EE,&C0,&60,&E0,&70    ;ROLAND FACING RIGHT
6930
6940 montab     DEFB &00,&07,&0C,&00,&01,&0C,&09,&08
6950    DEFB &03,&00,&0E,&0C,&05,&07,&06,&0E
6960    DEFB &06,&0D,&0F,&06,&00,&78,&69,&0F
6970    DEFB &0F,&C0,&90,&00,&00,&E0,&00,&0F
6980    DEFB &16,&F0,&F0,&83,&16,&70,&30,&06
6990    DEFB &03,&40,&00,&0E,&01,&00,&C0,&0C
7000    DEFB &00,&08,&01,&08,&11,&CC,&02,&00
7010    DEFB &FF,&FC,&F1,&CC,&11,&FE,&F1,&FF    ;ALAN SUGAR MONSTER
7020
7030    DEFB &11,&FF,&FF,&88,&77,&FF,&FF,&EE
7040    DEFB &FF,&00,&11,&FF,&FF,&00,&00,&FF
7050    DEFB &FF,&00,&00,&FF,&44,&00,&00,&00
7060    DEFB &22,&00,&00,&00,&44,&22,&22,&00
7070    DEFB &88,&77,&FF,&00,&44,&FE,&F3,&00
7080    DEFB &88,&CD,&3D,&88,&55,&CD,&1E,&CC
7090    DEFB &99,&CB,&1E,&CC,&55,&ED,&3D,&CC
7100    DEFB &77,&FE,&F3,&CC,&11,&FF,&FF,&CC    ;TELEPHONE
```