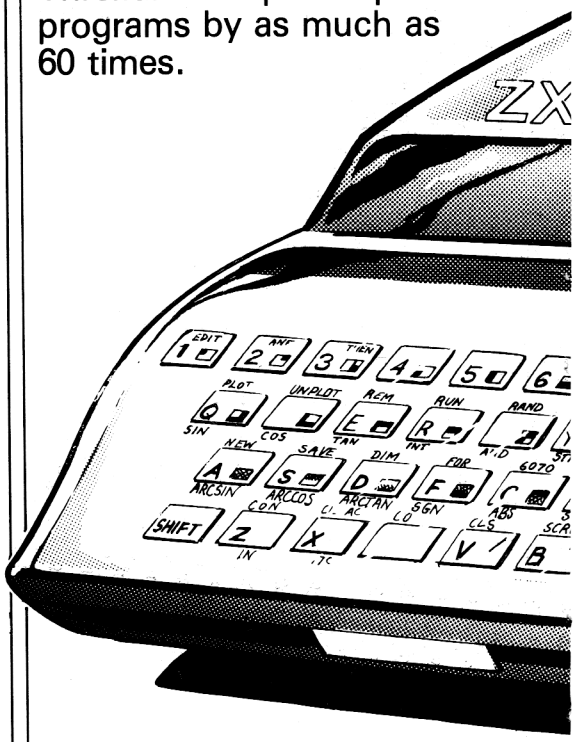
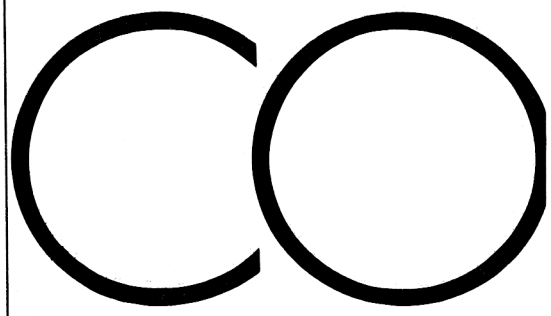
```
(C) Copyright 1983.
Personal Software Services.
Start at location 16514 with the
ZXGT header .The two HALTs cause
the rest of the statement to be
blanked out in BASIC.
         CP A
         CP L
         XOR H
         CP C
         HALT
         HALT
This short section contains a
series of fixed jumping points
which are used to get from the
first half to the second half of
the code.
 L201:   JP L151 ;PRINT
 L202:   JP L131 ;SCROLL
 L203:   JP L51 ;GET KEY
 L204:   JP L53 ;TEST KEY
 L205:   JP L165 ;PAUSE
 L206:   JP L100 ;MULTIPLY
 L207:   JP 0 ;PLOT
 L208:   JP L60 ;RST 10
 L209:   JP L65 ;£B6B
 L210:   JP L20 ;DIVIDE
 L211:   JP L99 ;RANDOM
 L212:   JP L80 ;GETNUM
 L213:   JP L187 ;REM
 L214:   JP L241 ;SAVE
When the break key is pressed we
come to here and do a RST to the
ROM.
 L252:   RST 08
         ADC A,H
Routine to print a 16 bit signed
integer which is held in the HL
register pair. A leading space
is added.
 L151:   PUSH HL
         PUSH DE
         XOR A
         CALL L60
         BIT 7,H
         JR Z,L153
         LD A,22
         CALL L60
         CALL L231
 L153:   LD DE,10000
         CALL L160
         JR NC,L154
         LD DE,1000
         CALL L160
         JR NC,L155
         LD DE,100
         CALL L160
         JR NC,L156
         LD E,10
         CALL L160
         JR NC,L157
         JP L158
 L154:   CALL L120
 L155:   LD DE,1000
         CALL L120
 L156:   LD DE,100
         CALL L120
 L157:   LD E,10
         CALL L120
 L158:   LD E,1
         CALL L120
         XOR A
         POP DE
         POP HL
         JP L60
 L160:   PUSH HL
         AND A
         SBC HL,DE
         POP HL
         RET
Repeatedly Subtract DE from HL
until the result is negative. If
A starts with character code 28
(=char 0) it finishes with the
character code of the left hand
digit of HL. Print this
character.
 L120:   LD A,28
         AND A
 L121:   SBC HL,DE
         JR C,L122
         INC A
         JR L121
 L122:   ADD HL,DE
         JP L60
Negate a 16 bit integer
 L231:   LD A,H
         CPL
         LD H,A
         LD A,L
         CPL
         LD L,A
         INC HL
         RET
Scroll the screen up 1 line
 L131:   LD HL,(16396)
         LD DE,33
         PUSH HL
         LD BC,0
         ADD HL,BC
         INC HL
         PUSH HL
         ADD HL,DE
         EX DE,HL
         LD HL,726
         SBC HL,BC
         EX (SP),HL
         POP BC
         EX DE,HL
         LDIR
         POP HL
         LD BC,£02B6
         ADD HL,BC
         LD (16398),HL
         LD A,33
         LD (£4039),A
         LD A,3
         LD (£403A),A
         RET
Wait for a key to be pressed,
return the value in the A
register and test for break. Rub
out is trapped and waits for
another key to be pressed.
 L51:    CALL £2BB
         INC H
         JR NZ,L51
 L57:    CALL £2BB
         LD A,L
         CP £FF
         JR Z,L57
 L54:    LD BC,£FD7F
         XOR A
         PUSH HL
         SBC HL,BC
         POP HL
         JP Z,L252
         LD BC,£FCEF
         XOR A
         PUSH HL
         SBC HL,BC
         POP HL
         JR Z,L51
         LD B,H
         INC H
         RET Z
         LD C,L
         CALL £7BD
         LD A,(HL)
         CP 0
         RET
See if any key is pressed, works
like INKEY£
 L53:    CALL £2BB
         LD A,H
         CP £FE
         JR Z,L51
         JR L54
Pause for n 1/50ths of a sec
leaving if a key is pressed.
 L165:   SET 7,H
         LD (16436),HL
 L166:   LD HL,(16436)
         LD A,H
         AND £7F
         OR L
```

ZX-81

This compiler by David Threlfall will speed up programs by as much as 60 times.





THE AIM OF this short series is to explain how it is possible to write a compiler for a machine as small as the ZX-81. We will look at the way that some of the Basic statements are translated — compiled — into machine code, and then go on to discuss extending the simple compiler to encompass string handling, multiple arrays and full variable names.

Why compiled Basic? Basic was written as an easy-to-learn language and to make it easy to run it was designed to be interpretative. This means that as each line of code is executed it is "retranslated" into a succession of calls to machine-code subroutines stored in the Basic ROM. This does not mean that Basic is translated into machine code — far from it — that is the purpose of a compiler. This method makes the language very easy to work with, particularly for developing a program, but slow to run.

Many compilers exist for the larger desk-top computers such as TRS-80, Pet, Apple and CP/M-based machines but those offering full floating-point arithmetic produce code only a few times faster than interpretative Basic as

ZXGT MPILER

most of the time is spend executing maths routines in the ROM rather than interpreting the code. ZXGT is the first true compiler small enough to fit into the ZX-81. To provide the necessary speed for games as well as to keep the compiler small — it is only just over 2.3K bytes — it only runs integer Basic and so is restricted to numbers from -32768 to 32767 with no strings or Boolean operations. The result is extremely fast code.

Before we discuss writing a compiler in detail it might be as well to define the purpose of a compiler. Essentially there are two main types of compiler, the first takes each line of Basic and converts it into a very compact set of simple instructions which are interpreted by a run-time system. The second type takes the lines of Basic and generates machine-code statements which carry out the functions of the Basic. Amongst these statements there may be calls to larger hunks of machine code within the operating system.

The first type of compiler produces very concise code but it cannot be executed directly by the microprocessor; instead, it is interpreted. The second type of compiler produces somewhat longer code which is directly executable and much faster.

When the design of this compiler was contemplated, experience of other compilers such as Microsoft and Accel led to the conclusion that "speed meant integer". To get the most speed the generation of an intermediate was rejected in favour of a mixture of pure machine code and calls to a library of fast routines within the compiler run-time system. It was expected that this would lead to an average speed up to about 20 times; in the event it was almost three times better than this.

The ZX-81 is a good target for a compiler for several reasons. First, its interpretative Basic is very slow. On standard benchmarks running in Slow mode it is 10 times slower than the BBC machine making it impossible to write good games in Basic. A compiler would increase the speed and integer arithmetic should be sufficient for most games.

Second, its Basic, in common with many

(continued on page 135)

```
          RET Z
          CALL L53
          JR Z,L166
Multiply   two  16  bit   signed
integers  together.  One in  HL,
the other in DE. Result in HL.
 L100:    LD B,16
          LD C,D
          LD A,E
          EX DE,HL
          LD HL,0
 L101:    SRL C
          RRA
          JR NC,L102
          ADD HL,DE
 L102:    EX DE,HL
          ADD HL,HL
          EX DE,HL
          DJNZ,L101
          RET
Plot  or  unplot a pixel on  the
screen.  Assumes a full 24  line
32 character screen.
 L0:      LD A,£2B
          SUB B
          JP C,L253
          LD B,A
          LD A,1
          SRA B
          JR NC,L1
          LD A,4
 L1:      SRA C
          JR NC,L2
          RLC A
 L2:      PUSH AF
          CALL L15
          LD A,(HL)
          RLC A
          CP £10
          JR NC,L4
          RRC A
          JR NC,L3
          XOR £8F
 L3:      LD B,A
 L4:      LD DE,£0C9E
          LD A,(£4030)
          SUB E
          JP M,L6
          POP AF
          CPL
          AND B
          JR L7
 L6:      POP AF
          OR B
 L7:      CP 8
          JR C,L8
          XOR £8F
 L8:      JP L60
 L15:     LD A,23
          SUB B
          JP C,L253
          LD A,C
          AND £1F
          LD C,A
          PUSH BC
          PUSH BC
          PUSH BC
          XOR A
          RL B
          RL B
          RL B
          LD L,B
          LD H,A
          ADD HL,HL
          ADD HL,HL
          POP BC
          LD C,B
          LD B,A
          ADD HL,BC
          POP BC
          LD B,A
          ADD HL,BC
          LD BC,(16396)
          ADD HL,BC
          INC HL
          LD (16398),HL
          POP BC
```

(listing continued on page 135)

(continued from page 133)

other machines, is tokenised, this means that all reserved words such as Let, Print and so on are represented as a single byte of code. This makes them very easy for the compiler to recognise.

Its Basic is rigorous. You may not leave out Let as in many Basics and after Then the word Goto may not be omitted.

The Basic is syntax-checked on input, so much of the error checking is already done. In fact, the only "errors" which can now occur arise where a particular function — such as strings — is not supported by the compiler.

The decision to restrict the scope of the compiler was taken for several reasons. The first was that the compiler was written in assembler language rather than in machine code, and only about 1.5K's worth of machine code can be held in assembler in a 16K machine. The compiler was written in two roughly equal sections yielding a total of 2,300 bytes of machine code. Also, the ZX-81 has no discs and only a very slow tape interface, so the compiler has to be in memory at the same time as both the original Basic and the machine code generated by the compiler. So keeping it small was important.

The first restriction to be imposed was to limit single letter variables, A-Y. By having only 25 variables a predefined area of memory can be set aside for them which may be accessed by adding the base address of this area to the position of the letter in the alphabet. This position is fixed in the compiler and if a variable is required the code to load it into the accumulator can be written by the compiler.

Next, the compiler was limited to one array — Z. By having only one array, which may be of any size that will fit in the machine, it can be placed above the Basic system and below the machine stack.

Furthermore, strings were omitted to keep the compiler small and to avoid the dynamic allocation of space their use entails. Dynamic allocation means that as the string gets longer more space is made for it; this can be slow.

For loops in this compiler have only positive single increments; the Step keyword is not supported. Also Boolean algebra — And, Or and Not — is not available.

Arithmetic expressions, except in Let statements, must be enclosed in parentheses as this makes the scope of the expression considerably easier to assess. This does not invalidate execution of the code in normal interpretative mode.

As regards the interface to Sinclair's Basic ROM, where possible the standard routines in the Sinclair ROM were used, but it soon became clear that in many cases this would not be possible. Two particular examples where it would at first seem possible to use the ROM routines but where it was decided to write special routines are worth considering.

Try plotting at every point on the screen and by careful observation you might notice that the top right-hand corner takes 70 percent longer to plot than the bottom left. Because of the variable length lines which are possible with the 1K RAM, the machine takes a long time to calculate where a particular point is in

(continued on page 137)

(listing continued from page 133)

```
        LD A,24
        SUB B
        LD (16442),A
        LD A,33
        SUB C
        LD (16441),A
        RET
Print the character ot token that is held in the A register. If we are at the bottom of the screen then display a ? and wait for a key to be pressed. If it is the COPY key make a copy. If it is the SLOW key scroll up at two lines per second. If it is the CONTINUE key then clear the screen and continue. Break returns to BASIC. Any other key causes a fast scroll.
L60:    PUSH DE
        PUSH HL
        PUSH BC
        PUSH AF
        LD A,(16442)
        CP 2
        JR Z.L63
        POP AF
L73:    CP 118
        JR Z,L61
        CP 64
        JR NC,L68
L78:    LD HL,(16398)
        LD (HL),A
        INC HL
        LD (16398),HL
        LD A,(16441)
        DEC A
        LD (16441),A
        LD A,(HL)
        CP 118
        JR NZ,L64
L62:    LD A,(16442)
        DEC A
        LD (16442),A
        INC HL
        LD (16398),HL
        LD A,33
        LD (16441),A
        JR L64
L61:    LD HL,(16398)
L74:    LD A,(HL)
        CP 118
        JR Z,L62
        INC HL
        JR L74
L63:    LD HL,(16396)
        LD BC,760
        ADD HL,BC
        LD A,143
        LD (HL),A
L66:    CALL L53
        JR Z,L66
        CP 40
        JR Z,L67
        CP 63
        JR Z,L75
        CP 41
        CALL Z,L76
        CALL L131
        JR L79
L67:    CALL £A2A
L79:    POP AF
        JR L73
L64:    POP BC
        POP HL
        POP DE
        RET
L75:    CALL £869
        JR L66
L76:    LD HL,10000
L77:    DEC HL
        LD A,L
        OR H
        JR NZ,L77
        RET
Print a sequence of characters. DE points to where the characters are and BC is the number of them there are. Printing is started at the present cursor position.
L65:    LD A,B
        OR C
        RET Z
        LD A,(DE)
        CALL L60
        INC DE
        DEC BC
        JR L65
Expand Tokens into the full keywords adding blanks front and back where necessary.
L68:    CP 67
        JR C,L69
        CP 192
        RES 6,A
        JP C,L78
        AND £3F
L69:    LD HL,£111
        LD B,A
        INC B
        CP 33
        JR NC,L70
        XOR A
        CALL L60
L70:    BIT 7,(HL)
        INC HL
        JR Z,L70
        DJNZ,L70
L71:    LD A,(HL)
        BIT 7,A
        JR NZ,L72
        CALL L60
        INC HL
        JR L71
L72:    AND £3F
        CALL L60
        XOR A
        JP L73
An overflow of some sort has occured so report an error B.
L253:   RST08
        ADC A,D
Divide the signed integer in HL by the signed integer in DE, result is in HL.
L20:    LD A,E
        OR D
        JR Z,L253
        CALL L40
        PUSH BC
        LD A,H
        OR D
        RLCA
        JR C,L253
        LD C,E
        LD B,D
        LD DE,0
        PUSH DE
        EX DE,HL
        INC HL
L21:    ADD HL,HL
        EX DE,HL
        ADD HL,HL
        LD A,C
        SUB L
        LD A,B
        SBC A,H
        EX DE,HL
        JR NC,L21
        EX DE,HL
L22:    EX DE,HL
L35:    XOR A
        LD A,H
        RRA
        LD H,A
        LD A,L
        RRA
        LD L,A
        OR H
        JR Z,L23
        EX DE,HL
        XOR A
```

(listing continued on page 137)

(continued from page 135)

the Display File. Using the entry point given in Mr Logan's book on the ZX-81 ROM it was possible to make the plotting an average four times faster if called from machine code. By modifying the calculation of the Display File address, knowing that ZXGT will only run on a 16K machine, it can be made 35 times faster than Basic.

The second problem is that if a Print causes the screen to be full after a call to Sinclair's printing routine, an error 5 results. When using Basic pressing Continue allows the screen to clear and the program to go on. However, if a machine-code program causes this to happen it will not return to that machine-code program. For this reason new Screen-handling routines are built into ZXGT which do not call the ROM. The final version handles all keyword expansion and, when the screen is full, gives you several options.

You can Press C to continue as before; or press D/Slow and the screen will scroll up at about two lines per second while your finger is on the key. Alternatively press Z/Copy and a copy is made on to the ZX Printer. A fourth option is to press any other key than C, D, Z and Break and the screen scrolls at speed — up to about 40 lines per second. The scroll is much faster than Sinclair's and does not result in a fragmented Display File which is hard to clear. Try a screen of Scrolls in Basic followed by CLS; it takes several seconds to clear.

Figure 1. The column on the right shows how many times faster ZXGT is.

Test	ZXGT (seconds)	ZX-81	
1. For-Next loop * 1000	0.11	19	172
2. If loop * 1000	0.30	27.4	91
3. As 2 + A=L/L*L+1-L	2.6	65	25
4. A$ 2 + A=L/2*3+4-5	5.0	63	13
5. As 4 + GoSub	5.0	79	16
6. As 5 + inner For loop	5.7	206	36
7. As 6 + Array assign	7.0	280	40
			Average 56

The speed improvement of ZXGT over Sinclair Basic is spectacular, varying from 13 to 172 times faster than Basic in the same mode — Slow or Fast.

For example, a For loop of 1,000 steps in interpreted Basic takes 19 seconds in Slow mode while ZXGT takes a little over 0.1 seconds. To get a feel for how quick this is, if this is run in Fast mode under ZXGT it takes 0.03 seconds and, by comparison, a large mainframe computer running Fortran needs 0.001 seconds. In addition, the order of statements no longer matters for speed.

On the standard benchmarks the Slow mode comparisons between ZXGT and Basic were as shown in figure 1.

It is difficult to comprehend what a factor of 60 speed-up on a program means until you realise one hour becomes one minute.

Now for the Assembler listing: those who wish to assemble the ZXGT code given with these articles should note the following. All numbers starting $ are hexadecimal. All RST commands are in hex but the dollar is not given. The labels are L0 to L255. All comment lines should be omitted. This is one third of the code. You will require a Rem of length 2,303 bytes as the first line in the machine. Note that this means that

PEEK 16511 + 256 * PEEK 16512 = 2303

The remaining two-thirds of the code almost fill the machine by themselves, so this third must be assembled and then deleted.

The entry point to the complete compiler is 17389 — use Rand Usr 17389 — for the code to be put in a Rem at line two or 17381 if you wish it to ask where the code is to be put.

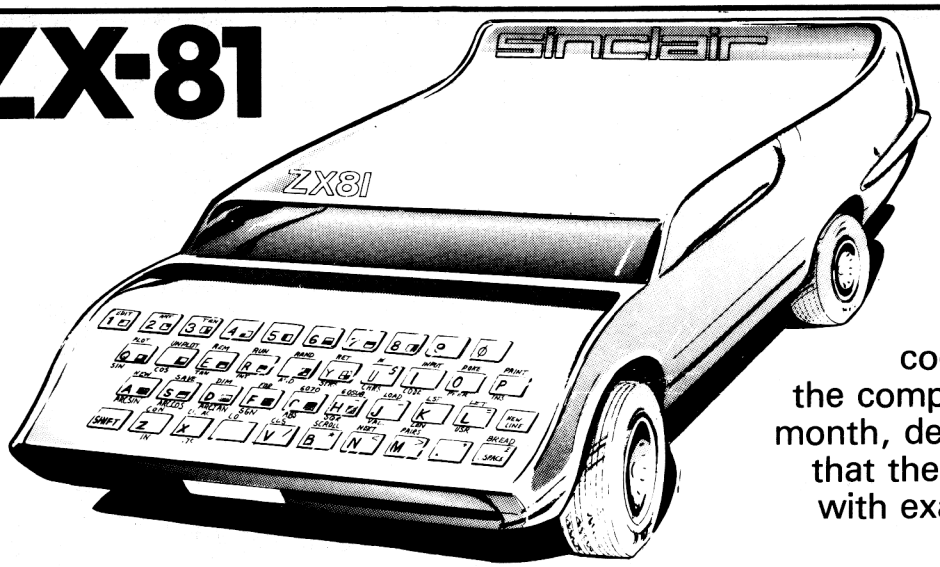
The entry to the code generated by ZXGT is at 18823.

Next month we shall give details of the way that the compiler translates Basic with examples from the standard Basic repertoire. We shall also supply a complete listing of the compiler in machine code. ■

(listing continued from page 135)

```
        RR H
        RR L
        LD A,C
        SUB L
        LD A,B
        SBC A,H
        JP M,L22
        LD A,C
        SUB L
        LD C,A
        LD A,B
        SBC A,H
        LD B,A
        EX (SP),HL
        ADD HL,DE
        EX (SP),HL
        JR L22
L23:    POP HL
        POP BC
        BIT 7,B
        JP NZ,L231
        RET

Find the sign of the division
and make both divisor and
dividend positive. Sign is in B.

L40:    LD B,H
        LD A,H
        RLA
        CALL C,L231
        EX DE,HL
        LD A,H
        XOR B
        LD B,A
        BIT 7,H
        JP NZ,L231
        RET

Generate a pseudo random integer
number using the standard
Sinclair Seed. The result is
always between 0 and 32768.

L99:    LD DE,(16434)
        LD H,E
        LD L,£FD
        LD A,D
        OR A
        SBC HL,DE
        SBC A,0
        SBC HL,DE
        SBC A,0
        LD E,A
        LD D,0
        SBC HL,DE
        JR NC,+1
        INC HL
        LD (16434),HL
        RES 7,H
        RET

Make a REM statement to contain
the code which we are
generating. This REM always
contains two 118's at the start
to stop BASIC trying to list the
machine code. There is a call to
ROM in here and this routine
must be executed in FAST mode.
The length of code arrives in
BC. On return HL is the address
of the first location available.
18816 is the location of the
first character in the REM.

L187:   INC BC
        INC BC
        PUSH BC
        LD HL,6
        ADD HL,BC
        LD B,H
        LD C,L
        LD HL,18816
        CALL £9A3
        INC HL
        LD A,118
        LD (DE),A
        LD (HL),0
        INC HL
        LD (HL),2
        INC HL
        POP BC
        INC BC
        INC BC
        LD (HL),C
        INC HL
        LD (HL),B
        INC HL
        LD (HL),234
        INC HL
        LD (HL),A
        INC HL
        LD (HL),A
        INC HL
        RET

Get a signed integer from the
keyboard - used for INPUT and to
ask user where the code should
be put (see the second half of
the code).

The result is returned in the HL
register pair.

L80:    LD HL,0
        PUSH HL
L82:    PUSH HL
L89:    CALL L51
        CP 22
        JR NZ,L87
        POP HL
        POP DE
        PUSH AF
        PUSH HL
        CALL L60
        JR L89
L87:    CP 118
        POP HL
        JR Z,L88
        ADD HL,HL
        PUSH HL
        ADD HL,HL
        ADD HL,HL
        POP DE
        ADD HL,DE
        SUB 28
        LD B,0
        LD C,A
        ADD HL,BC
        ADD A,28
        CALL L60
        JR L82
L88:    POP AF
        CALL Z,L231
L92:    RET

Save the code away into the the
location pointed to by location
16507. If the SLOW key is
pressed display the location and
contents - useful in debugging.

L241:   PUSH BC
        PUSH HL
        LD HL,(16507)
        PUSH AF
        LD (HL),A
        PUSH HL
        INC HL
        LD (16507),HL
        CALL L53
        CP 41
        JR NZ,L242
        LD A,118
        CALL L60
        POP HL
        CALL L151
        POP AF
        LD H,0
        LD L,A
        CALL L151
        JR L243
L242:   POP HL
        POP AF
L243:   POP HL
        POP BC
        XOR A
        RET
```

ZX-81



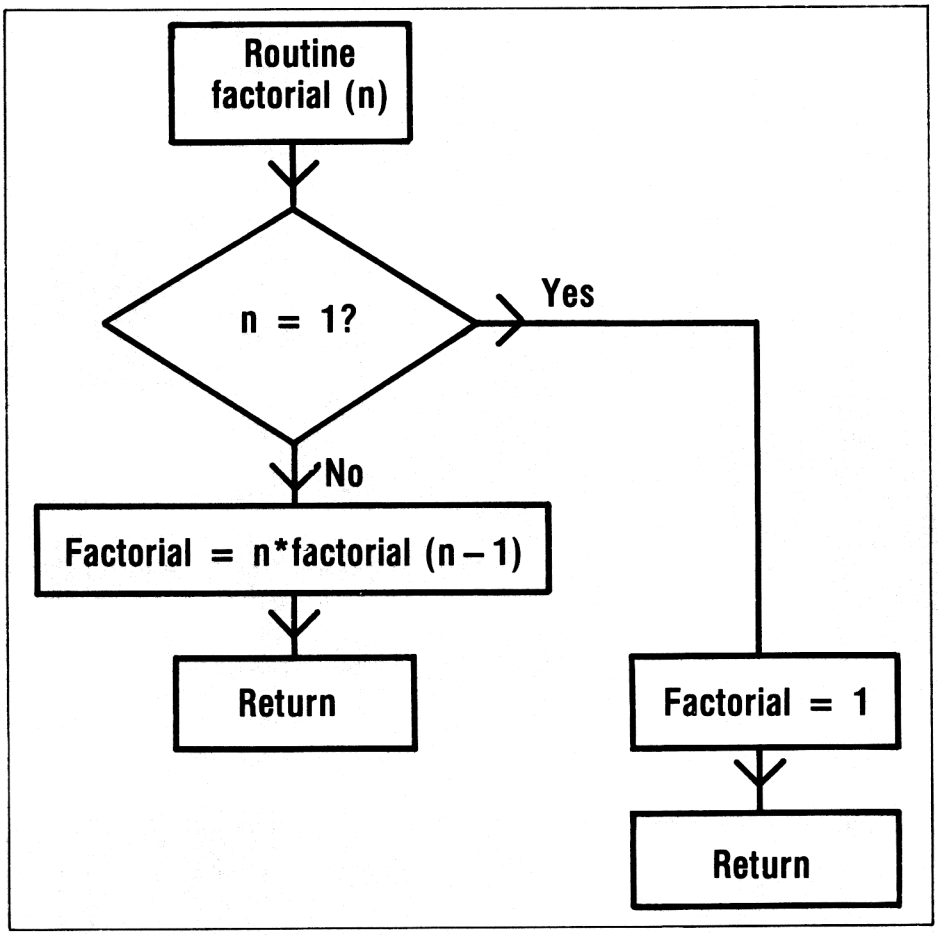
David Threlfall continues his short series with the complete machine code for ZXGT, a true compiler for the ZX-81. ZXGT is only just over 2.3K bytes. The fast code is the result of limiting the compiler to integer Basic. This month, details are given of the way that the compiler translates Basic with examples from the standard Basic repertoire.

GT COMPILER

LAST MONTH we considered the philosophy behind ZXGT, my ZX-81 Basic compiler. This month we move on to the intricacies of integer arithmetic evaluation and see how some statements compile.

For those uninitiated to Z-80 machine code here are a few preliminaries. ZXGT uses the Z-80 registers A,B,C,D,E,H and L. A is the eight-bit accumulator. H and L may be considered as a single 16-bit accumulator. All the registers may be used for eight-bit storage but the pairs BC and DE may also be used in 16-bit manipulations. Putting a register pair or 16-bit number in brackets means that the value in brackets should be taken to point to the location required. For example:

LD HL,n	load HL with the value n
but	
LD A,(HL)	means load register A with the data in the location pointed to by the value in the HL register pair.



Here are a few simple examples to start the description of the compiler.

CLS

This results in a call to Sinclair's ROM at hexadecimal address 0A2A.

RETURN

This one is very easy, requiring the Z-80 instruction Ret — return.

PEEK n

This causes HL — the double-precision accumulator — to be loaded with the contents of location n, thus:

LD HL,n	load HL with n
LD A,(HL)	load the accumulator with the contents of location HL
LD L,A	move A into L
LD H,0	zero H
	HL now contains the contents of location n.

The next example is:

ABS X

Load HL with X and test the top bit of H — the sign bit. Call a negate routine if this bit is set, that is, if the number is negative. Negating a number entails taking the 2's complement but there is no Z-80 instruction for this. Instead we must take the 1's complement of H and L independently and then increment HL.

POKE x,y

This means put the lower byte of y in location x. As we need x and y simultaneously they cannot both be in the HL register pair. Therefore we get x in HL and y in DE then do LD (HL),E as required remembering that Poke acts on only one byte.

GOTO n

This will be translated as a Jump — JP — instruction; n must be a number and not a variable. The compiler has two passes. On the first, it generates a table of line numbers and their addresses in the machine code. On the second pass, the correct addresses will be available for both forward and backward Gotos.

(continued on page 79)

(continued from page 77)

Note in particular that, in the compiled code, the nearness of n to the start of the program does not affect the time taken to execute the Goto.

GOSUB n

This works just like a Goto except that it results in a Call instead of a JP.

USR x

This results in the machine code at location x being executed. It looks as if it should result in just a Call to location x. However, there is no machine code statement for "call to the address given by a register pair" that is,

CALL (HL)

so subterfuge is necessary. Consider the following code:

```
LD BC,BACK   load BC with the address of
             label BACK.
PUSH BC      keep BC on the stack
LD HL,x      get x into HL
PUSH HL      and push HL on to the stack

BACK: continue code
```

The first four lines get the address of Back and the address to which we wish to go on to the stack. The Ret instruction makes the machine "return" to the address at the top of the stack which is x, just as we wanted. At the end of the routine starting at x, a Return causes a jump to the next address on the stack which is Back and there we are.

FOR-NEXT

The For-Next pair is compiled into directly executable code — not calls to other routines — and so a For-Next loop is extremely fast. The For statement has the form:

FOR K=M TO N

where M and N may be parenthesised expressions. M is moved into variable K and (N+1) is stored in the next word/two-byte location. During compilation the address of the next location after For — let us call it zzzz — is also stored. The next K statement is compiled thus:

```
LD HL,(nnnn)     where nnnn is the location
                 where variable K is stored.
INC HL           make K one bigger
LD (nnnn),HL     store this value
LD DE,(nnnn+2)   get value of end of loop
AND A            clear carry flag
SBC HL,DE        subtract DE from HL
JP M,zzzz        if HL-DE is negative jump to
                 the next address after FOR.
                 Otherwise execute the next
                 instruction.
```

This arrangement results in extremely fast execution of the final code — about 170 times faster than Basic. Machine-code enthusiasts might care to consider what limitations the test places on the values of M and N.

Now for some arithmetic. Wherever a variable may be used in Basic an expression may be substituted, so some means has to be found to evaluate that expression. The method which has been chosen for ZXGT uses an often-mentioned but rarely-used mathematical function called recursion.

For those who have not come across recursion before, consider evaluation of n!, that is, n factorial; n! is defined as:

$$n! = n \times (n-1) \times (n-2) \ldots\ldots\ldots \times 1$$

and we may rewrite this as:

$$n! = n \times (n-1)!$$
$$= n \times (n-1) \times (n-2)!$$

etc.

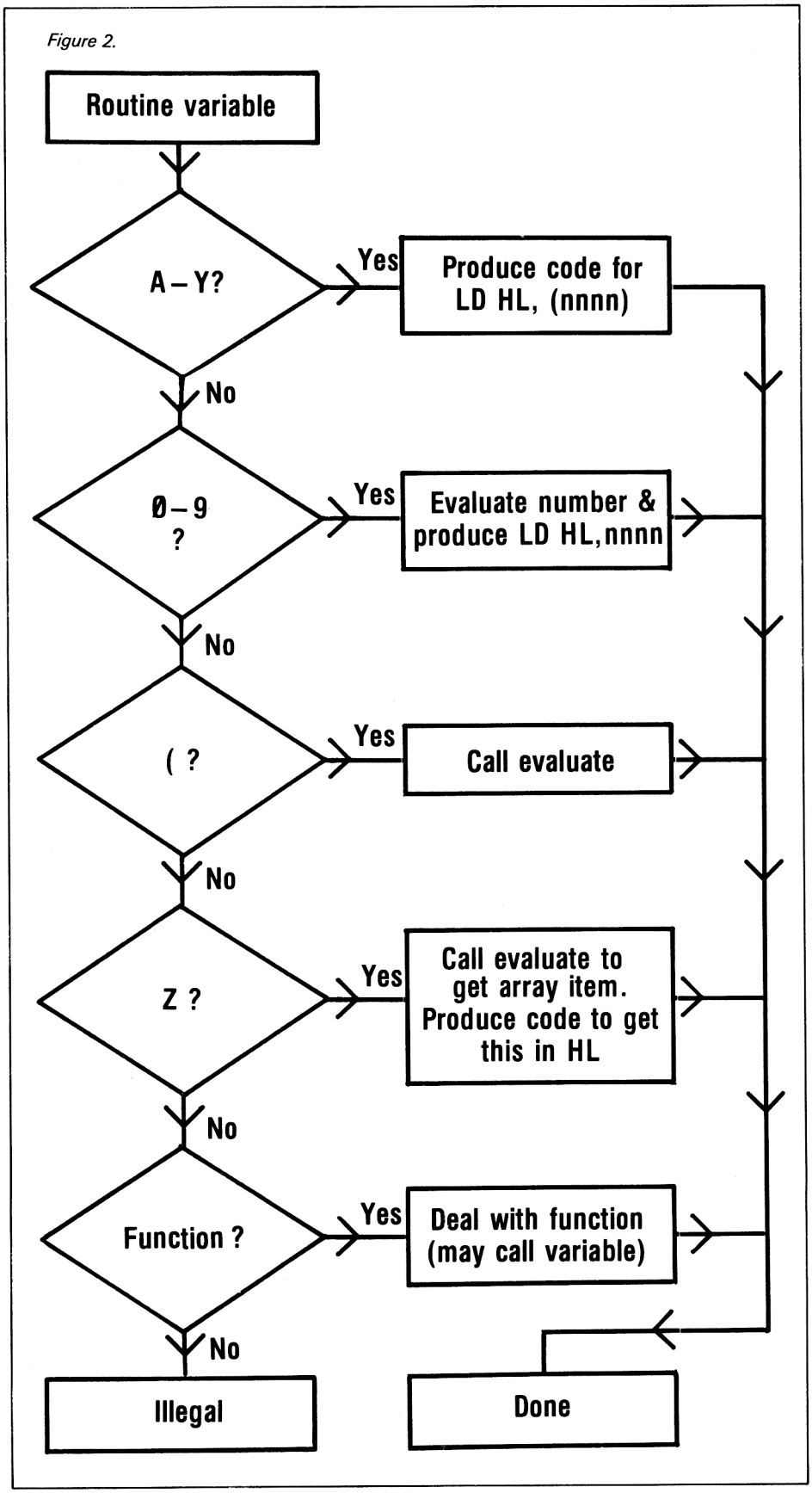
To calculate n! it is necessary to multiply n by (n–1)!. To calculate (n–1)! we multiply (n–1) by (n–2)! This process is continued until we arrive at 1! which is 1. Figure 1 shows a flow diagram for this process. The routine Factorial calls itself repeatedly.

The process of evaluating an expression uses a similar technique which is shown in figures 2 and 3. We see that Variable calls Evaluate and Evaluate calls Variable, but the way out may not be clear. Each time we enter Evaluate, a marker — 0 — is pushed on to the compiler stack and when we reach the end of the line or a right parenthesis, the stack is popped back to see what "pending operations" are left.

Operations are performed until an "operator" 0 is encountered. We placed this

(continued on page 81)

Figure 2.



(continued from page 79)

there to mark the end of the expression when we entered the routine. The exit is taken with the result in HL.

Two other points should be noted. Firstly, the right and left parentheses will match exactly, because of Sinclair's syntax checking. Secondly, we are using two different stacks. The operands of the expression are pushed on to the stack of the compiled program by code generated in the compiler. The operators are kept on the compiler stack and are used by the compiler to cause the correct code to be generated for combinations of operand in HL and DE.

In effect, infix notation is changed into postfix or reverse Polish.

The Let statement calls Evaluate directly. Many other Basic statements are supported such as: Fast-Slow, Input, Pause Rand/Rnd to seed and use the random number generator, Print, Scroll, Stop Unplot/Plot.

The ZX-81 does not have the ability to store on tape anything except Basic statements so where can the machine code generated by the compiler be put so that it may be recalled from tape for later use?

The general answer to this is "in a Rem statement" and that is the solution adopted by the compiler. The one twist is that the compiler generates its own Rem statement into which it puts the code. To accomplish this the code is first compiled over the ROM — and so not stored — and the resulting length of code is used in forming the Rem. It does not delete any old compiled code.

One of the best features of ZXGT is that the Basic may be run and tested under the interpreter before the compiler is invoked. One writes a program bearing in mind that eventually it will be compiled. When you are satisfied with your code a single Usr command runs the compiler and puts your code into the Rem statement. This is clearly a very powerful feature and one which should eventually become standard on all small machines.

Some people may not be aware of the way to make the large Rem required for ZXGT. A possible procedure is as follows: first, type a line 1:

```
1 REM ABCDEFGHIJKLMNOPQRSTUVWXY
ABCDEFGHIJKLMNOPQRSTUVWXY
ABCDEFGHIJKLMNOPQRSTUVWXY
ABCDEFGHIJKLMNOPQRS
```

This makes the total length of the line including the terminator — 118 — exactly 100 bytes. Check that

```
PEEK 16511
```

is 96. Second, edit line 1 changing the line number to 2. Third, edit line 2 to make it line 3. Repeat up to line 23. Fourth, edit line 1 adding 1234567 immediately after Rem. We now have the correct number of characters. Fifth, check that

```
PEEK 18815
```

is 118. Steps 6 to 10 of the procedure are:

```
POKE 16512,8
POKE 16511,255
```

(continued on page 83)

Figure 3.

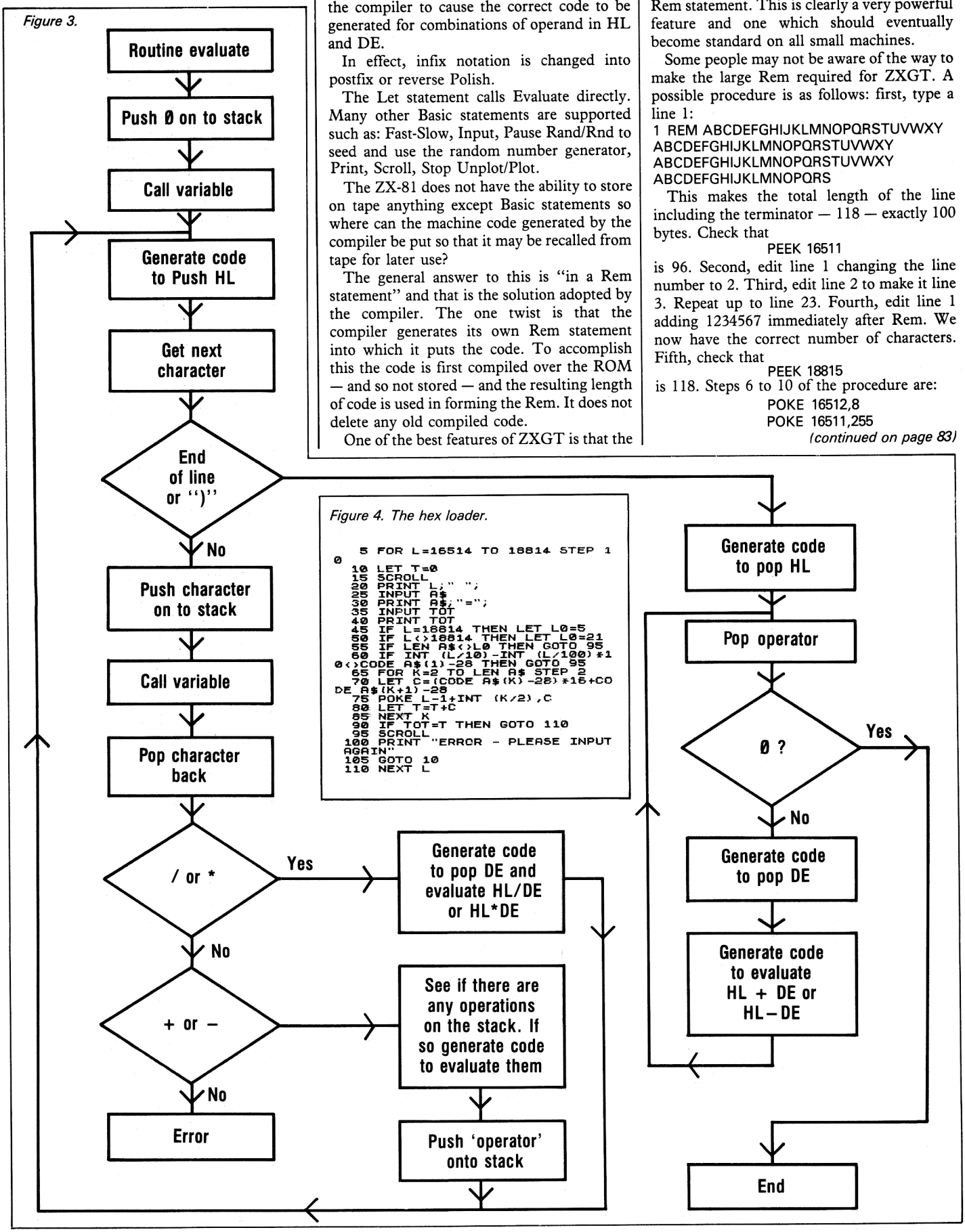


Figure 4. The hex loader.

```
  5 FOR L=16514 TO 18814 STEP 1
0
 10 LET T=0
 15 SCROLL
 20 PRINT L;" ";
 25 INPUT A$
 30 PRINT A$;"=";
 35 INPUT TOT
 40 PRINT TOT
 45 IF L=18814 THEN LET L0=5
 50 IF L<>18814 THEN LET L0=21
 55 IF LEN A$<>L0 THEN GOTO 95
 60 IF INT (L/10)-INT (L/100)*1
0<>CODE A$(1)-28 THEN GOTO 95
 65 FOR K=2 TO LEN A$ STEP 2
 70 LET C=(CODE A$(K)-28)*16+CO
DE A$(K+1)-28
 75 POKE L-1+INT (K/2),C
 80 LET T=T+C
 85 NEXT K
 90 IF TOT=T THEN GOTO 110
 95 SCROLL
100 PRINT "ERROR - PLEASE INPUT
AGAIN"
105 GOTO 10
110 NEXT L
```

(continued from page 81)

```
POKE 16514,118
POKE 16515,118
POKE 16510,0
```

You now have a Rem called line zero of length 2,303 bytes as required. Note that there are minor differences between the assembler code and the version of ZXGT in the hex dump.

Figure 4 shows the hex loader that will enable you to enter ZXGT. Note that the Rem to contain the code must be exactly 2,303 bytes in length, that is the total line length as defined by Sinclair. The loader will prompt with the address to be loaded and you should refer to figure 5 to see the hex string that should be entered.

You should input the 21 characters up to, but not including, the equals sign followed by Newline. You will then be prompted for the check number by an equals sign. If the hex and decimal agree you will be prompted for the next line of input. If they disagree you will be asked to re-enter the data. The last hex string only contains five characters.

The entry point to the complete compiler is 17389 — use Rand Usr 17389 or Let L=Usr 17389 — for the code to be put in a Rem at line 2 or 17381 if you wish it to ask where the code is to be put.

The entry to the code generated by ZXGT is at 18823. Use

```
Let L=USR 18823
```

Do not use

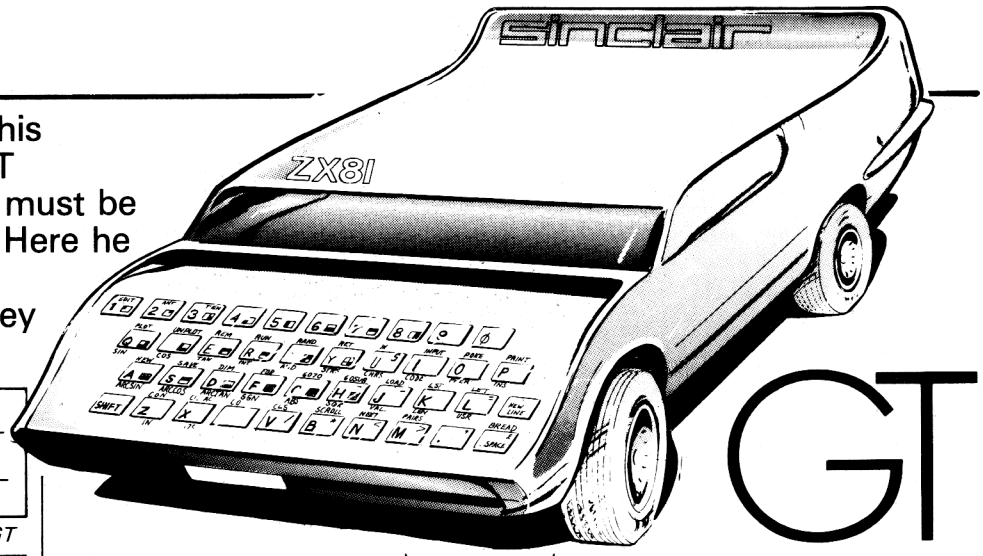
```
RAND USR 18823
```

In future issues, we shall give the remaining third of the ZXGT assembler code and discuss how to avoid some of the restrictions mentioned in part 1. If you find the listing too daunting to type in the compiler is available on cassette for £8.95 from Personal Software Services, 452 Stoney Stanton Road, Coventry.

Figure 5. Hex dump of ZXGT compiler.

```
16514 1BFBDACB97676C3B440C3=1607
16524 22341C35741C34F41C385=1114
16534 341C39741C3AB41C32142=1201
16544 4C39C42C3DE42C33943C3=1414
16554 57A43C35743C3B843CF8C=1331
16564 6E5D5AFCD2142CB7C2808=1296
16574 73E16CD2142CD1B411110=718
16584 827CD0741301A11E803CD=847
16594 907413015116400CD0741=535
16604 030131E0ACD07413012C3=645
16614 1FC40CD0D4111E803CD0D=1069
16624 241116400CD0D411E0ACD=710
16634 30D411E01CD0D41AFD1E1=1001
16644 4C32142E5A7ED52E1C93E=1497
16654 51CA7ED5238033C18F919=931
16664 6C321427C2F677D2F6F23=886
16674 7C92A0C40112100E50100=599
16684 8000923E519EB21D602ED=1019
16694 942E3C1EBEDB0E101B602=1544
16704 009220E403E213239403E=449
16714 103323A40C9CDB8027CFE=1148
16724 2FE200ECDBB022420FACD=1217
16734 3BB027DFEFF28F8017FFD=1492
16744 4AFE5ED42E1CAB24001EF=1616
16754 5FCAFE5ED42E128DD4424=1549
16764 6C84DCDBD077EFE00C9CB=1462
16774 7FC2234402A34407CE67F=1041
16784 8B5C8CD4F4128F306104A=1109
16794 97BEB210000CB391F3001=731
16804 019EB29EB10F5C93E2B90=1247
16814 1DADC42473E01CB283002=931
16824 23E04CB293002CB07F5CD=1020
16834 3EB417ECB07FE103007CB=1164
16844 40F3002EE8F47119E0C3A=762
16854 5304093FAE141F12FA018=1271
16864 602F1B0FE083802EE8F18=1144
16874 7363E1790DADC4279E61F=1169
16884 84FC5C5C5AFCB10CB10CB=1486
16894 910686729290C1484709C1=843
16904 04709ED4B0C400923220E=560
16914 140C13E1890323A403E21=754
16924 291323940C9D5E5C5F53A=1459
16934 33A40FE02283AF1FE7628=1129
16944 42AFE403072ZA0E407723=796
16954 5220E403A39403D323940=523
16964 67EFE76204A3A3A403D32=895
16974 73A4023220E403E213239=471
16984 84018382A0E407EFE7628=802
16994 9E62318F82A0C4001F802=906
17004 009368FCD4F4128FBFE28=1140
17014 12815FE3F281BFE292008=780
17024 22110272B7DB420FBCD23=959
17034 3411803CD2A0AF1189AC1=961
17044 4E1D1C9CD690818D378B1=1485
17054 5C81ACD2142130B18F5FE=1083
17064 6433809FEC0CBB7DA3542=1301
17074 7E63F2111014704FE2130=754
17084 804AFCD2142CB7E2328FB=1138
17094 910F97ECB7F2006CD2142=1063
17104 02318F5E63FCD2142AFC3=1271
17114 12D42CF8A7BB228FACD29=1293
17124 243C57CB20738F14B4211=1028
17134 30000D5EB2329EB297995=1070
17144 4789CEB30F6EBEBAF7C1F=1605
17154 5677D1F6FB42818EBAFCB=1227
17164 61CCB1D7995789CFAFE42=1376
17174 779954F789C47E319E318=1199
17184 8DDE1C1CB78C21B41C944=1517
17194 97C17DC1B41EB7CA847CB=1260
17204 07CC21B41C9ED5B324063=1152
17214 12EFD7AB70600ED5298ED=1318
17224 252985F50ED5230012322=846
17234 33240CBBCC9030303C503=915
17244 403030321804 9CDA30923=655
17254 53E767123360223C17123=760
17264 6702336EA2377237723C9=979
17274 7210000E5E5CD5741FE16=1124
17284 82009E1D1F5E5CD214218=1277
17294 9F0FE76E1281F29E52929=1260
17304 0D1194FCD214279D61C38=1036
17314 1084F060009FE0A38D3F1=874
17324 23EE6CD214218C7F1CC1B=1291
17334 341C9C5E52A7B40F577E5=1514
17344 423227B40CD4F41FE2920=932
17354 5123E76CD2142E1CDB440=1176
17364 6F126006FCDB4401802E1=1090
17374 7F1E1C1AFC9C9C93E0FD7=1729
17384 8CD7549181221000CD12=693
17394 944CDF74803CD7849E5CD=1427
17404 00649E1CD1244CD5949CD=1167
17414 12044CD0D44CF7F3EC9C3=1178
17424 27B49227040ED5B1C4013=845
17434 3217240CD09452A704036=766
17444 41823366823227940116 8=592
17454 50019227B40217D40CDD4=885
17464 6442323E52322164 0CD44=795
17474 745FEF2CCCB44FEF9CC96=1897
17484 848FEFACCC848FEDE28EA=1802
17494 9FEE9CC1D49FEECCC2B45=1599
17504 0FEEDCC4045FEEBCCC747=1791
17514 1FEF3CCF447FEE4CC0049=1775
17524 2FEF4CCB647FEE5CCF148=1955
17534 3FEEECC5D48FEF6CC6C48=1745
17544 4FEFCCC9248FEF1CC0346=1700
17554 5FEFBCC9F47FEE3CA9647=1843
17564 6FEF5CC5545FEEA2815FE=1660
17574 7E7CCA447FEFECC0D44FE=1717
17584 800200ECD4445FE762007=799
17594 9E1CD0E49C236443EB8CD=1284
17604 06849CD5949CF9BCD4D47=1259
17614 1215F49C3A947E556235E=1080
17624 2D5EBE5222340220A40CD=1123
17634 3EC45CD5349E1CD13451B=1211
17644 41BEBD1CD0945ED5B7B40=1269
17654 5CD0945E5ED5B7240A7ED=1422
17664 652E13803227240E1C973=1119
17674 7237223C95E235623C9E5=1065
17684 8C12A1C4023CD0E45EBA7=1052
17694 9ED42D02A7240ED52D8EB=1501
17704 02318ED3EC3F5CD4445CD=1345
17714 12A48CD1345EBCD0E45EB=1165
17724 2F1C3AB473ECD18E9E5E7=1662
17734 3FE762809FE7E2805F5CD=1296
17744 46849F1E1C9CD4445FE76=1558
17754 52877FE0B2813FEC1284C=1046
17764 6FED62874CD5047215349=1169
17774 7CDA947184F3E18CD7B49=1035
17784 82A7B40E5CD7B4901FFFF=1370
17794 92A164023037EF5CD6849=919
17804 0F1FE0B2805CD7B4918EF=1215
17814 1221640E1C571233E11CD=974
17824 2AB473E01E1CDAB47216B=1117
17834 349CDA9471812CD3747CD=1096
17844 4C14721434DCDAE4721F5=1169
17854 508CDA947CD4445FE1A28=1115
17864 68CFE19288821EC45CDA9=1307
17874 7472A16402B221640AFC9=738
17884 8CD4D473E7DCD7B492168=1078
17894 949CDA94718D63E76C368=1235
17904 049FE40D2C144D6261717=1160
17914 1ED4B794026006F09C9CD=1061
17924 24445CDF145FE64200ACD=1253
17934 32A493EE5CD7B4926FFE5=1329
17944 4CD4445FE14C2C144CD3D=1337
17954 546CDD345E17CFEFF200C=1457
17964 621D1EBCDAE47210945C3=1233
17974 7A9473E22C3AB47AFF5CD=1398
17984 84D473EE5CD7B49CD4445=1182
17994 9FE112850FE76284CF5CD=1329
18004 04D47F1FE152804FE1620=1016
18014 123C16778FE002004F5E5=1215
18024 218D8E521444DCDAE47CD=1302
18034 37E463EE5216960CDAB47=1168
18044 418C4F5C5181CFE182810=1048
18054 5FE17C2C144CDC6462162=1336
18064 649CDA94718ACCDC64621=1220
18074 76E4918F33EE1CD7B49F1=1379
18084 8FE00C8FE152811CDC646=1259
18094 921A7EBCDAE4721ED52CD=1442
18104 0AE4718E7CDC6463E19CD=1265
18114 17B4918DD3ED1C37B49FE=1357
18124 2D3200FCD4D47217E6FCD=1086
18134 3AE47212600C3AE47FED4=1222
18144 42012CD4D4721EB46C3A9=1105
18154 54701F146C5E5C96069C9=1412
18164 6FEC4C20F47CD4445FE41=1391
18174 7C2C144215C49CDA9473E=1160
18184 82621006FC3AB47FE40C2=1131
18194 91A47217249C3A947FED2=1216
18204 02011CD4D4721CB7CDDAE=1141
18214 1473EC4217A47C3AB47FE=1246
18224 2CFC2C144C34D47CD4D47=1358
18234 3CD4445F53EE5CD7B49CD=1484
18244 44D473ED1CD7B49F1C9CD=1467
18254 54445FE16F5CC4445FE10=1269
18264 62816FE40300DFE26D482=1075
18274 747DC8F47F1CC7547C9CD=1544
18284 8CB4618F6CD3D4618F121=1177
18294 97A47182F7C2F677D2F6F=821
18304 023C9CDF145FE64CA3F49=1443
18314 13E2AC3AB47CD2A483E21=955
18324 21815CD0D44E1ED4B7B40=1055
18334 3C9212A0A180521564918=531
18344 4003ECDCD7B497DCD7B49=1194
18354 57CC37B49CD3747CDC147=1315
18364 621EB7318EDFE1AC2C144=1379
18374 7C9CD4445CDF145E5E5CD=1721
18384 84445CD37473E23CD7B49=966
18394 921ED53CDAE47E1CDAE47=1478
18404 03E222323CDAB47E1ED5B=1166
18414 17B40CD0945C9CD4445CD=1218
18424 2F145E5E53E2ACDAB4721=1352
18434 32322CDAE47E1CDAE4723=1229
18444 4233EEDCD7B493E5BCDAB=1264
18454 5473EA721ED52CDAB473E=1161
18464 6FAE1CD0E45EBCDAB47C9=1646
18474 7210000180FCD4445FE7E=794
18484 8281AF5110A00CD6249F1=955
18494 9FE26D2C144FE1CDAC144=1524
18504 0D61C06004F0918DFE52A=854
18514 116401105001922164 0E1=478
18524 2C9217549CDA947CD4445=1211
18534 3CDF145C338463E9BF5CD=1503
18544 43747CDC147214B45CDAE=1151
18554 5473E3ECD7B49F1CD7B49=1238
18564 63E32213040CDAB472165=838
18574 749C3A9473EA018D83E2A=1074
18584 8213440CDAB473E222132=775
18594 940C3AA47FEDD2812FE14=1307
18604 028141F3001EB17A7ED52=884
18614 137F817D01808ED5237C0=1132
18624 21804ED5237C83FC9CD37=1126
18634 347672E3ECDAE4721A648=1003
18644 4CDA9472A0A4023CD1345=889
18654 5EBCD0E45EB3ED2CDAB47=1477
18664 6CD4445FEDEC2C144C921=1507
18674 7F748C3A947CDE702213B=1284
18684 840CBB6C9210649C3A947=1197
18694 9213B40CBF6C307024E23=922
18704 04609233A0C409DC03A0D=668
18714 1409CC9CD4445FE3FC2C1=1467
18724 244CD4D47AFC9CD4D472 1=1183
18734 3ED4BCDAE47211C40CDAE=1266
18744 447212909C3AE47CD2A49=914
18754 53ED5CD7B49210E45CDA9=1166
18764 64721EBD1C3AE47C38840=1383
18774 7C38B40C38E40C3914 0C3=1398
18784 89440C39740C39A40C39D=1387
18794 940C3A040EBC3A340C3A6=1501
18804 040C3A940C3AC40C3AF40=1357
18814 1AF40=239
```

ZXGT COMPILER



David Threlfall continues his insight into how the ZXGT compiler works and what must be considered in the design. Here he looks at the compiler's shortcomings and how they have been overcome.

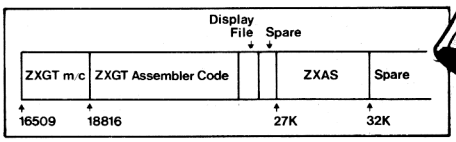


Figure 1a: Memory map during work on ZXGT

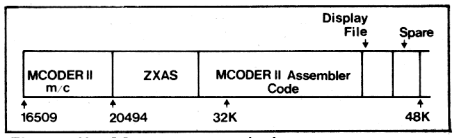


Figure 1b: Memory map during work on MCoder II

A FEW MONTHS after ZXGT had been put on the market as MCoder a few of its shortcomings became apparent. As it stood it worked extremely well, was very fast and far exceeded its original specification, but there are several areas where improvements would be welcome. Problem areas include;

■ No strings — this can be circumvented but string handling would make more a much more versatile compiler.

■ Only 25 variables — programs written for ZXGT were getting larger and this restriction was proving a major handicap to writing understandable code. Multi-character names were necessary, not even two-character names as on the Pet could be considered.

■ Only one numeric array — the array was added as an afterthought primarily to help fulfil the role of data logger on scientific experiments. A full 26 arrays, even if one dimensional, would be very useful.

■ Printing was not *à la* Basic — each number was printed with a leading space and "," did not tabulate.

■ Lprint was not supported.

■ No Boolean algebra at all — a little would be useful.

■ Input was totally non-standard, it appears on the main screen not on the input area and there is no rubout. Input at the bottom of the screen with the ability to correct mistakes was necessary.

■ At about 56 times faster than Basic it was not slow but work on the Spectrum version had shown that more speed was possible — the final version is almost 70 times faster than Basic.

■ Brackets are necessary except in Let — this is compatible with Basic but annoying.

■ Last but still relevant was size. The original ZXGT had been aimed at 2K bytes and only just ran over but most users gave the impression that to obtain the items above they would sacrifice a little more space and longer loading time. 4K was felt to be optimum if all these features could be added.

■ Floating point. Slight consideration was given to full floating-point arithmetic. However, work with Microsoft's compiler on a larger machine had shown that a speed up of only five or so could be expected particularly as Sinclair's ROM routines are very slow. Put another way the balance between interpretation and evaluation is heavily on evaluating. For this reason primarily it was decided to stay with integer arithmetic on all the compilers for Sinclair computers.

Readers may wonder why these features were not implemented in ZXGT. ZXGT was not written directly in machine code but in assembler, a listing of which has been appearing in parallel with this text. It is hard

(continued on next page)

```
The Sinclair functions are
examined here and code
appropriate to each is
generated.
PEEK ?
 L16:   CP 211
        JR NZ,L161
        CALL L108
        LD HL,#6F7E
        CALL L36
        LD HL,#0026
        JP L36
USR ?
 L161:  CP 212
        JR NZ,L162
        CALL L108
        LD HL,L163
        JP L39
 L163:  LD BC,L160
        PUSH BC
        PUSH HL
        RET
 L160:  LD H.B
        LD L.C
        RET
CODE INKEY$ ?
 L162:  CP 196
        JP NZ,L168
        CALL L240
        CP 65
        JP NZ,L252
        LD HL,L53
        CALL L39
        LD A,38
        LD HL,#6F00
        JP L38
RND ?
 L168:  CP 64
        JP NZ,L172
        LD HL,L117
        JP L39
ABS ?
 L172:  CP 210
        JR NZ,L201
        CALL L108
        LD HL,#7CCB
        CALL L36
        LD A,196
        LD HL,L231
        JP L38
                              10
INT ?
 L201:  CP 207
        JP NZ,L252
        JP L108
 L244:  CALL L108
Get a pair of arguments and
generate the code
    ld hl,nnnn or ld hl,(nnnn)
    push hl
    ld hl,nnnn or ld hl,(nnnn)
    pop de
        CALL L240
 L246:  PUSH AF
        LD A,229
        CALL L241
        CALL L108
        LD A,209
        CALL L241
        POP AF
        RET
Work out what the next thing is.
Is it a variable, a number or a
function?
 L108:  CALL L240
 L102:  CP 22
        PUSH AF
        CALL Z,L240
        CP 16
        JR Z,L218
        CP 64
        JR NC,L97
        CP 38
        CALL NC,L245
        CALL C,L86
 L98:   POP AF
        CALL Z,L230
        RET
 L97:   CALL L16
        JR L98
 L218:  CALL L200
        JR L98
Generate negation code.
 L230:  LD HL,L231
        JR L39
Actually do the negation.
 L231:  LD A,H
        CPL
        LD H,A
        LD A,L
        CPL
        LD L,A
        INC HL
        RET
                              11
Get variable then code is
generated for ld hl,(nnnn).
 L245:  CALL L37
        CP 108
        JP Z,L234
        LD A,42
        JP L38
Get integer then code generated
for ld hl,nnnn.
 L86:   CALL L81
        LD A,33
        JR L38
STOP statement seen
 L110:  CALL L96
        POP HL
        LD BC,(16507)
        RET
CLS statement seen. Generate a
call to the ROM.
 L52:   LD HL,#A2A
        JR L39
SCROLL statement seen. Generate
a call to the scroll routine.
 L130:  LD HL,L131
        JR L39
A routine to save either HL as
code, HL and A as code or to
generate a call to the number
presently in HL.
 L39:   LD A,205
 L38:   CALL L241
 L36:   LD A,L
        CALL L241
        LD A,H
        JP L241
POKE , get the two arguments and
make sure that a comma separates
them.
 L141:  CALL L244
        CALL L107
        LD HL,#73EB
        JR L36
Check that the char in A is a
comma, if not error.
 L107:  CP 26
        JP NZ,L252
        RET
                              12
FOR statement seen. Generate
code to load the variable with
the first number and keep the
second number incremented by one
in the next location.
 L100:  CALL L240
        CALL L37
        PUSH HL
        PUSH HL
        CALL L240
        CALL L244
        LD A,35
        CALL L241
        LD HL,#53ED
        CALL L36
        POP HL
        CALL L36
        LD A,34
        INC HL
        INC HL
        CALL L38
        POP HL
        LD DE,(16507)
        CALL L170
        RET
NEXT statement. Find the
variable name and generate code
to increment it and test if we
have reached the end. See Text.
 L101:  CALL L240
        CALL L37
        PUSH HL
        PUSH HL
        LD A,42
        CALL L38
        LD HL,#2223
        CALL L36
        POP HL
        CALL L36
        INC HL
        INC HL
        LD A,237
        CALL L241
        LD A,91
        CALL L38
        LD A,167
        LD HL,#52ED
```

(listing continued on next page)

(listing continued from previous page)

```
            CALL L38
            LD A,25Ø
            POP HL
            CALL L171
            EX DE,HL
            CALL L38
            RET
                                    13
Get a number from the code
skipping over the floating point
form which follows it.
 L81:   LD HL,Ø
        JR L83
 L85:   CALL L24Ø
        CP 126
        JR Z,L84
        PUSH AF
        LD DE,1Ø
        CALL L112
        POP AF
 L83:   CP 38
        JP NC,L252
        CP 28
        JP C,L252
        SUB 28
        LD B,Ø
        LD C,A
        ADD HL,BC
        JR L85
 L84:   PUSH HL
        LD HL,(16406)
        LD DE,5
        ADD HL,DE
        LD (16406),HL
        POP HL
        RET
INPUT statement. Find the
variable and generate a call to
read the keyboard and then code
to put the result in the slot
reserved for that variable.
 L75:   LD HL,L8Ø
        CALL L39
        CALL L24Ø
        CALL L37
        JP L1Ø5
PLOT statement. Find the pair of
arguments and then generate a
call to the Plotting routine.
 L6Ø:   LD A,#9B
 L61:   PUSH AF
        CALL L244
        CALL L1Ø7
        LD HL,#454B
        CALL L36
        LD A,#3E
        CALL L241
        POP AF
        CALL L241
        LD A,#32
        LD HL,#4Ø3B
        CALL L38
        LD HL,L113
        JP L39
                                    14
UNPLOT statement. Much like PLOT
but with a different argument
generated.
 L65:   LD A,#AØ
        JR L61
RAND statement. Gets number and
puts it into location 16436.
 L31:   LD A,42
        LD HL,16436
        CALL L38
        LD A,34
        LD HL,16434
        JP L38
IF tests start here. Arrive with
DE=first argument, HL=second and
A=operation. Result is carry set
if condition true.
 L18Ø:  CP 221
        JR Z,L182
        CP 2Ø
        JR Z,L184
        RRA
        JR NC,L181
        EX DE,HL
 L181:  RLA
        AND A
        SBC HL,DE
        SCF
        RET M
        RLA
        RET NC
```

(listing continued on page 123)

(continued from previous page)

to write very compact machine code mainly because changes and corrections are nearly impossible so space is always left just in case.

However, it is impossible to fit an assembler, the target code and the source for more than 1.5K of machine code in a 16K machine. That is why the code was written in two longish sections — figure 1a. This left two options — write the code in more than two sections — two were hard to manipulate, three or more would be devilish — or bootstrap from Basic.

The concept of bootstrapping is to write a small compiler with features that allow you to write a larger one. At least one Algol compiler has been written that way. Just a kernel of machine code is necessary to get started and this already existed in ZXGT.

This route was avoided since the code generated by ZXGT is 16-bit integer and therefore would not be suited to the mainly eight-bit character handling of a compiler. Perhaps somone would like to write an eight-bit version of ZXGT — it would be even faster.

Now, you are probably asking why I did not buy a 32K or larger memory pack. The answer is that I had one but that ZXAS — the assembler I had — sat in the 27 to 32K slot so stopping Basic from crossing the 32K word boundary. For those who do not already know, it is quite permissible for Basic code — not just variables and arrays — to cross this boundary but at no time must the display file straddle the line, either side is equally alright. Also, you cannot run machine code which sits above 32K.

Salvation finally came when DCoder arrived on the market. DCoder is a disassembler but more important it is a relocator so it was possible to relocate ZXAS to sit above the compiler machine code — figure 1b.

Relocatable code is code which may be put and run anywhere in the machine. It is easy to write relocatable sub-routines but whole programs are harder. Suffice to say that neither ZXAS or ZXGT are relocatable.

So a relocator is a smart disassembler which can spot which jumps and calls are to within the code to be moved and change them appropriate to the new location. This does not make the relocated code relocatable but merely changes where it will run. The twist to this story is that DCoder was written independently by someone using MCoder — alias ZXGT.

The third major limitation was the Sinclair tape interface. I have rarely had any trouble with this but by the time MCoder II was completed 32K bytes of code had to be run back in each time the machine crashed. With the standard tape interface that takes about 15 minutes. This time the remedy was QSave which records 32K bytes in a little under one minute onto a standard tape recorder.

The final piece in the jigsaw, which arrived when MCoder II was nearly complete, was Ian Logan's ROM disassembly for the ZX-81. This allowed several new features to be added by using ROM routines to save space. ROM routines were only used where time was not critical eg. in Print. It also explained some of the clever but slow features of Sinclair's Basic.

In this section problem areas will be discussed in detail.

■ Strings: The major difficulty with strings was to avoid garbage collection when the string was lengthened or shortened. Eventually this was solved by decreeing that all strings would have a maximum length of 32 bytes. This length is held in a location that is known to the user so that it may be Poked from Basic.

The value is reset to 32 when the compiled program is run and must be Poked from your program. By this means the maximum length for individual strings can be set as the string is used for the first time. Thereafter the length of that string cannot be increased beyond that limit. There are no string arrays.

All strings — and numeric arrays for that matter — are allocated space at run-time and so sizes need not be preset. These strings/arrays are stored above Stkend. Consider this short program:

```
1 INPUT N
2 DIM A(N)
3 LET B$="QWERTYUIOP"
4 LET C$="BROWN FOX"
5 DIM X(N+N)
```

The program would prompt for N, allocate the array A — 2*N bytes to be zeroed — allocate B$ and C$ and finally allocate the array X of length 2N. Note the use of addition rather than multiplication to form 2N. Multiplication takes 10 times longer than addition and division 50 times longer. The result is the memory map shown in figure 2.

To concatenate strings, and that includes CHR$, INKEY$ or "string", it is necessary to form the new string elsewhere than its final resting place in case a string appears both on the left and right hand side of the assignment. For example:

```
LET A$ = A$(1 TO 3) + A$(6) + CHR$
128 + B$ + "*"
```

results in the new A$ being built above Last —

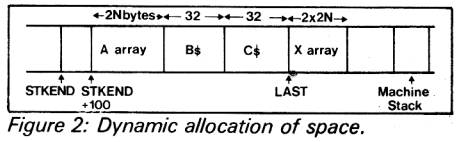


Figure 2: Dynamic allocation of space.

see figure 3 — and then locations Last to End are copied into A$. The length of the string is simply the number of bytes between Last and End.

Printing strings is a similar process except that the final copy is to the screen. A more difficult task is to compare two strings, for example:

```
IF A$ <> B$ THEN PRINT "NOT EQUAL"
```

A copy of A$ is made above Last — which is reset first — and a copy of B$ is made above End. Each is terminated with the code 255. With HL pointing to A$ and BC to B$, incrementing and comparing and using the results flags it is possible to do all six standard comparisons with one short routine.

■ Full variable names: Sinclair Basic is rare in allowing variable names of any length which may contain spaces if desired. Apart from disallowing spaces this is fully implemented in MCoder II. The variable names are stored as shown in figure 4. The names are entered downwards from just below the machine stack — actually 256 bytes clear of RAMtop — and the last letter is stored in inverse video, ie., bit seven is set.

For strings the names are considered to be A$ to Z$ and for arrays A< to Z< thereby differentiating between the three sorts of variable. An example might be to store the names Fred, A$, One and an array C, these would be stored down from just below the stack thus:

```
F
R
E
Inverse D
A
Inverse $
O
N
Inverse E
C
Inverse <
0
```

The zero on the bottom marks the end of the list. There can only be 255 variables as a single byte is used to store the number of variables. A careful note should be kept during compilation of the maximum number of concurrent variables together so that space may be reserved for them at the end of the compiled code.

In Basic the command Clear removes all the

(continued on page 123)

(continued from page 120)

variables, but in MCoder the command merely zeroes all the variables used in the program. Zeroing the locations reserved for a string variable actually zeroes the pointer to the string implying that it has not yet been allocated.

■ Numeric arrays: As the DIM statement for an array is encountered by the compiler, code is generated to load the HL register pair with the length of the array — always one dimensional — and a call is generated to the "allocate space and zero" routine. This generates and stores a pointer to where the array is to be put and then zeroes the array as follows:

```
LD HL, where array is to be put (=LAST)
LD, BC, length of array*2
LD D, H
LD E,L
INC DE
LD (HL),0
LDIR
LD (LAST),DE
```

If the array already exists then the old one is forgotten and a new one created. This failure to collect old, unused space can be wasteful but is not considered too detrimental.

Array elements are accessed by calculating the index, doubling it — two bytes per element — and adding this number to the base address of the array. No array bound checking is done as the code is assumed to be fully tested under Basic.

■ Printing to the screen: To attain the same result as Basic is really only a matter of attention to detail. However, the Tab command may be of some interest. Sinclair stores the column number where printing is to take place next but stores it backwards. The actual column number stored is 33 — the number the user inputs. For example Tab 5 makes the system variable S_POSN (16441) take the value 28 and Tab 25 yields the value 8.

■ Printing to the ZX printer: If printing to the screen is clear, but perverse, consider Lprint. The printer buffer is in locations 16444 to 16476. Tab 5 makes PR_CC take the value 69 and Tab 25 makes it 89, these numbers being the lower byte of the address in the buffer. Otherwise the change from Print to Lprint commands requires only that bit one of location 16385 be set and then Sinclair's routines for Print, Tab, At etc may be used.

■ Boolean operations: Only two Boolean operations have been made available in MCoder II:

B AND C which is true if B and C are non zero
B OR C which is true if either B or C or both are non zero.

These are only implemented in the If statement using routines for Or and And which are called with HL containing one 16 bit argument and DE the other. They return a set carry flag if the result is true.

```
AND:  LD A,H      ;test if
      OR L        ;HL is zero
      RET Z       ;return if it is

      LD A,D      ;now do DE
      OR E
      RET Z
      SCF         ;set carry flag
      RET         ;return
OR:   LD A,H      ;test if HL zero
      OR L
      JR Z,+2     ;jump on if it is
      SCF         ;HL non zero, set carry
      RET
      LD A,D
      OR E
      RET Z       ;both HL and DE zero
      SCF
      RET
```

■ Input: This had to be completely rewritten to be like Basic. For simplicity of coding input is limited to 31 characters plus the prompt so that it all fits on one line. This avoids the problem of compressing the display file. Invalid characters will not print, incorrect ones may be rubbed out, but you cannot step over characters using the cursors.

The screen is used as the input buffer and a string input is copied directly to the space reserved for the variable. The length can be ascertained from the position of the cursor. At the end the buffer is cleared using part of the array zeroing routine described above.

■ Speed: In ZXGT the particular If test to be carried out was decided at run-time. This resulted in very short code for the six functions. In MCoder II code is generated at compile time. As before HL and DE contain the two operands in the general form.

IF operand1 operator operand2 THEN
and the code generated is:

```
LD HL,operand1   ;this may be the result of an
                  evaluation
PUSH HL
LD HL,operand2
POP DE
AND A            ;clear carry flag
SBC, HL, DE      ;subtract the two operands
                  from each other
JP 'flag', NEXT
.....            ;code to be executed if the
                  condition
.....            ;is true
NEXT: .....      ;next statement after IF
```

where 'flag' depends on the operator. For example <> requires the use of the zero flag. This results in considerably quicker code than in ZXGT, often by almost 50 per cent.

■ Brackets: Originally an arithmetic expression was terminated either by the end of line character — 118 — or by a right bracket. The change required to evaluate an expression where brackets are not necessary — ie, like Basic — turned out to be very easy but needed the logic to be stood on its head. The expression is now not terminated by the four operators +,—,/,* and everything else means that the end has been reached, be it with a comma, semicolon, bracket etc.

Sinclair makes a distinction between the way that various functions decide where their arguments stop. For instance Peek 16388+1 means (Peek 16388)+1 not Peek (16388+1) but Poke 16388+1,n means the same as Poke 16389,n.

It seemed as if all the functions just picked up the next item, enclosed in brackets if necessary, while statements such as Poke, Plot etc., took expressions treating the comma or end of line as terminator. However, all was not so simple as Tab, which seems to be a function, takes the whole expression up to the next delimiter eg., Tab 5+3 means Tab 8.

■ Size: The final size of MCoder II at 3980 bytes is considerably larger than ZXGT/MCoder showing how expensive in space the refinements proved but it is a much easier program to use and its versatility is greatly enhanced.

I hope that this short series has given some insights into the way that a compiler works and what it is necessary to consider in the design.

ZXGT is marketed by Personal Software Services, 452 Stoney Stanton Road, Coventry CV6 5DG. ■

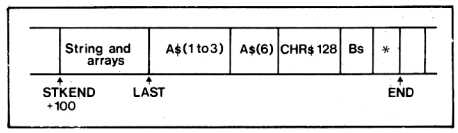


Figure 3: Forming a concatenated string.

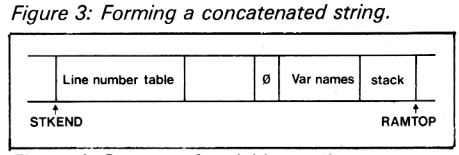


Figure 4: Storage of variable names.

(listing continued from page 120)

```
        JR L188
L182:   SBC HL,DE
L183:   SCF
        RET NZ
        JR L185
L184:   SBC HL,DE
L188:   SCF
        RET Z
L185:   CCF
        RET
IF statement. Generate code to get the two operands to be compared and put operation in A register. Generate call to " If tests". On return test carry and jump to next line of code if its not set.
L186:   CALL L244
        LD H,A
        LD L,62
        CALL L36
        LD HL,L188
        CALL L39
        LD HL,(16394)
        INC HL
        CALL L91
        EX DE,HL
        CALL L171
        EX DE,HL
        LD A,210
        CALL L38
        CALL L240
        CP 222
        JP NZ,L252
        RET
FAST and SLOW statements. Because the direct calls to the ROM moved in the new ROM some of the ROM code is duplicated here.
L40:    LD HL,L103
        JP L39
L103:   CALL #02E7
        LD HL,#403B
        RES 6,(HL)
        RET
L38:    LD HL,L104
        JP L39
L104:   LD HL,#403B
        SET 6,(HL)
        JP #207
Get address of next line. Enter with HL pointing to the length of the present line. Return with HL pointing to the new line.
L50:    LD C,(HL)
        INC HL
        LD B,(HL)
        ADD HL,BC
        INC HL
        LD A,(16396)
        SBC A,L
        RET NZ
        LD A,(16397)
        SBC A,H
        RET
DIM statement. Make sure argument is only single dimensional. The size is ignored as there is only one array. Note that the array is not zeroed.
L236:   CALL L240
        CP 63
        JP NZ,L252
        CALL L108
        XOR A
        RET
                                16
These last two routines are used in calculating the location of an array element. Basically the subscript is doubled and added to the contents of STKEND (16412).
L233:   CALL L108
        LD HL,#4BED
        CALL L36
        LD HL,16412
        CALL L36
        LD HL,#929
        JP L36
L234:   CALL L233
        LD A,#D5
        CALL L241
        LD HL,L171
        CALL L39
        LD HL,#D1EB
        JP L36
Finally a jump list which links to the jump list at the start of the first section. Although relatively expensive in space this method is pretty fool proof if any code is moved.
L151:   JP #4088;PRNT
L131:   JP #408B;SCROLL
L51:    JP #408E;GETK
L53:    JP #4091;TESTK
L111:   JP #4094;PAUSE
L112:   JP #4097;MULT
L113:   JP #409A;PLOT
L114:   JP #409D;RST10
L115:   JP #40A0;#B6B
L116:   EX DE,HL
        JP #40A3;DIV
L117:   JP #40A6;RND
L80:    JP #40A9;GETNUM
L187:   JP #40AC;REM
L241:   JP #40AF;SAVE
```