# ASSEMBLER NOTES

**Laurie Shields**

Introduction

"We have not yet succeeded in answering all your questions ... Indeed; we feel that we have not yet answered any of them. The answers we have found only serve to raise a whole new set of problems. In some ways we feel we are as confused as ever, but we believe that we are confused on a higher plane, about more important things."

Assembler Programming

One school of thought defines assembler programming as the nearest thing to the tortures of the Spanish Inquisition, in its ability to deprive its sufferers from sleep and recognisable human social behaviour; but this opinion cannot truly be relied upon because it totally disregards the self inflicting masochistic aspect of the disease.

Perhaps though we ought to at least consider how the external symptoms are displayed. Usually they take the form of the sufferer ceaselessly keying in at the nearest computer some totally meaningless expressions, that they fondly believe to be 'English type words' and then asking the computer to change the words into equally meaningless numbers. When finished the net effect of their efforts is to achieve the same result as switching the computer off but without the benefit of saving any electricity.

Some of the more expert practitioners in the art form have even been known to fill the video picture with one thousand and twenty four '@' characters and generate a buzzing noise as the penultimate climax to their hours of unceasing toil. Ordinary BASIC programmers, who can't get beyond fifty '@' characters before the dreaded 'Out of String Space' appears on their videos, go in awe and trembling before the amazing powers of the assembler.

The following pages on assembler programming and machine code in general on the TRS80 are the result of some considerable pressure from Brian to put down on paper the content of some of the talks and discussions on Zen that have occurred over the past eighteen months. I am deeply indebted to a large number of people especially John Hawthorne who wrote the original Zen and also to Radio Shack for producing the TRS80 and Scripsit as well as countless others who gave me help when I started and didn't know the difference between a Byte and a Bootstrap.

Laurie Shields.
Chesterfield. September 1981.

# Chapter 1

To get the best use from these notes you really ought to have a live TRS80 with Zen powered up and working. As there are a number of different versions of Zen with differing degrees of sophistication I shall try to refer only to the original fundamental commands which to a great degree are all upwardly compatible. Zen in its original cassette only form is marketed by Newbear for the TRS80 and other Z80 machines (Sharp, Nascom etc) and in its upgraded enhanced format for the TRS80 Cassette, Aculab or Disc by Laurie Shields Software.

Just to get started and at the same time to use Zen to explore the inside of the TRS80, key the following:

Q5712 <E> and also, by itself X <E>.

Now we have some explaining to do as your video looks like a meaningless jumble of numbers and letters. It actually is displaying part of the Leve12 Rom, a few other bits as well and should be:

```
1650 C5 4E 44 C6 4F 52 D2 45  ENDFORRE  .ND.OR.E
1658 53 45 54 D3 45 54 C3 4C  SETSETCL  SET.ET.L
1660 53 C3 4D 44 D2 41 4E 44  SCMDRAND  S.MD.AND
1668 4F 4D CE 45 58 54 C4 41  OMNEXTDA  OM.EXT.A
1670 54 41 C9 4E 50 55 54 C4  TAINPUTD  TA.NPUT.
1678 49 4D D2 45 41 44 CC 45  IMREADLE  IM.EAD.E
1680 54 C7 4F 54 4F D2 55 4E  TGOTORUN  T.OTO.UN
1688 C9 46 D2 45 53 54 4F 52  IFRESTOR  .F.ESTOR

 HL   DE   BC   AF   RI    IX   IY   SP   PC
0000 0000 0000 0000 0000
0000 0000 0000 0000       0000 0000 5514 402D
```

If the numbers you have on display for the last two lines are a bit different, DON'T PANIC. It only means that you're on cassette, or floppy tape or are using a different Dos from the one I was using when this was written.
In order to understand any of what Zen is trying to tell us we must first of all make the effort to comprehend the way information, be it program or data, is stored in the computer and the conventions used to present it at the human interface.

Right down at rock bottom most if not all computers store and evaluate numbers in simple binary 0's and 1's. However most of the time we and the computer operate not with these individual Bits but with eight of them at a time called

Bytes. Invariably however, just to break the first rule straight away and purely for our convenience, these bytes are split into half bytes called nibbles, so that we can avoid having to do mental arithmetic to base 256.

By thinking in terms of 4 bits at a time, the powers-at-be have forced us into straining the brain and understanding hexadecimal arithmetic (base 16), which isn't really too bad, and not much worse than feet and inches. Consider the eight bits below with their numeric values when the bit is set (i.e. equal to 1 rather than 0):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit number: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Decimal val. | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Hex value. | 80 | 40 | 20 | 10 | 08 | 04 | 02 | 01 |

Now it doesn't take an overdose of grey matter to see that the values of bits 0 to 3 can be represented by the right hand half of two hexadecimal digits and the bits 4 to 7 by the left hand one. Even if all the bits 0 to 3 were set giving a value of 1 + 2 + 4 + 8 = 15 this would not exceed the ability of hexadecimal notation, where 15 is represented by F.

Crib Sheet for the newcomers:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decimal: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Hexadecimal: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

Even Better Crib Sheet for beginners and experts:

| | U N I T S | | | | | | | | | | ASCII Value |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tens | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| 00 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | Printer & |
| 10 | 0A | 0B | 0C | 0D | 0E | 0F | 10 | 11 | 12 | 13 | Display |
| 20 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | Codes |
| 30 | 1E | 1F | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | !"#$%&’ |
| 40 | 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F | 30 | 31 | ()*+,-./Ol |
| 50 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 3A | 3B | 23456789:; |
| 60 | 3C | 3D | 3E | 3F | 40 | 41 | 42 | 43 | 44 | 45 | <=>7@ABCDE |
| 70 | 46 | 47 | 48 | 49 | 4A | 4E | 4C | 4D | 4E | 4F | FGHIJRLMNO |
| 80 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | PQRSTUVWXY |
| 90 | 5A | 5B | 5C | 5D | 5E | 5F | 60 | 61 | 62 | 63 | Z[\]^_‘abc |
| 100 | 64 | 65 | 66 | 67 | 68 | 69 | 6A | 6B | 6C | 6D | defghijklm |
| 110 | 6E | 6F | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | nopgrstuvw |
| 120 | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 7F | 80 | 81 | xyz |
| 130 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | BA | BE | |
| 140 | 8C | 8D | 8E | 8F | 90 | 91 | 92 | 93 | 94 | 95 | Graphic |
| 150 | 96 | 97 | 98 | 99 | 9A | 9B | 9C | 9D | 9E | 9F | Codes |
| 160 | A0 | Al | A2 | A3 | A4 | A5 | A6 | A7 | AS | A9 | 128 to 191. |
| 170 | AA | AB | AC | AD | AE | AF | B0 | B1 | B2 | B3 | |
| 180 | B4 | B5 | B6 | B7 | B8 | B9 | BA | BB | BC | BD | |
| 190 | BE | BF | C0 | C1 | C2 | C3 | C4 | CS | C6 | C7 | Space |
| 200 | CB | C9 | CA | CB | CC | CD | CE | CF | DO | Dl | Compression |

```
210 D2 D3 D4 D5 D6 D7 D8 D9 DA DB Codes
220 DC DD DE DF E0 El E2 E3 E4 E5 192 to 255
230 E6 E7 E8 E9 EA EB EC ED EE EF
240 FO Fl F2 F3 F4 F5 F6 F7 FS F9
250 FA FB FC FD FE FF
```

In one eight bit byte we have the ability to store any number from 0 (decimal), OOH (hexadecimal) or 0000 0000 (binary) up to 255 (decimal), OFFH (hexadecimal) or 1111 1111 (binary). We have now gone and confused you by putting a 0 in front of the FFH and someone wants to know why. Well the answer is relatively simple and it doesn't mean that we have acquired an extra ninth bit of value zero. It simply follows on from our normal programming convention that if something begins with a letter it is a variable and something beginning with a numeric digit is a number. So to avoid any possible confusion we always (well nearly always) put a leading zero in front of any hex number that begins with a letter.

Now this convention is purely one way, ie from human to computer. This is because the computer cannot guess as to whether when we key FFH we mean a number, a label or variable or just a plain expletive deletive. When the computer is giving us information the assumption is made that we know what we have asked for and should recognise it when we see it.

So what happened after we keyed Q5712 ? Well the first thing Zen did was to check whether or not we had put an 'H' at the end of our input. Had we done so the Zen would have treated it as hexadecimal; since there wasn't it was treated as an ordinary base ten decimal number and Zen turned the 5712 into hex and rounded the last digit down to the nearest 0 or 8. Decimal 5712 converts exactly to 1650 Hex, in this case no rounding was necessary. The letter Q was interpreted by Zen as a 'Query memory' command so Zen started by printing the value, 1650 and then examined the TRS80's memory starting at 1650 and displaying in hex each byte from 1650 to 1658.

The right hand half of the display is an Ascii representation of the same 8 bytes of memory but shown so that the bytes whose values are greater than 127 and not normally considered as displayable Ascii are depicted in two formats for maximum information. On the extreme, right is the academically correct display where any bytes greater than 127 are shown as full stops. On the left is a much more interesting presentation as all the bytes are displayed as recognisable letters. This conversion from unprintable to printable is achieved by Zen subtracting the value 128 from any number greater than 127.

Now this is no accident or defect in the ROM because if we look closely at the letters we see that what is shown is a listing of some of BASIC's instruction codes such as END FOR RESET etc. except that there are no spaces in between. This is a look-up table used by Leve12 BASIC to check on the spelling in your Basic program and if what you key in doesn't match any of the words in the table you get a syntax error.

By squashing all the words together Microsoft saved themselves quite a lot of ROM that would been just spaces but they then had to figure out a way of knowing when they had come to the end of one keyword and the start of the next. This they did by adding 128 or 80Hex to the first letter of each word and then, simply by testing if any letter was bigger than 127, they knew where each word started. So to get the correct spelling all they had to do was deduct 128 to obtain the correct ASCII value.
This in machine code is the equivalent of starting every word with an enormous 'capital' letter.

Now adding, subtracting and testing for this magical 128 is not very difficult in machine code as it depends solely on whether Bit 7 is set or not, a condition that is very readily detected with the Z80 chip.

So Zen will tell us anything we want to know about the memory in our TRS80, just key the letter Q followed by the address and there it all is. If you want the next 64 Bytes there's no need to give the address again as Zen remembers where the last block finished so just key Q <E>.
Should you require a printout of the Q display then key Shift Q instead of Q and the display will be routed to the lineprinter driver in Zen.

# Chapter 2

Now what on earth are we to make of the second part of the display:

```
 HL   DE   BC   AF   RI   IX   IY   SP   PC
0000 0000 0000 0000 0000
0000 0000 0000 0000      0000 0000 5514 402D
```

Well lets think of a BASIC program that has the line:

```
100 LET Y = 4 + 3 * 3
```

and imagine that we stopped the program with the Break key just as the Z80 had finished working out that line.

We could then tell the computer to PRINT Y and get the answer 13. Obviously, the variable Y has had it's value stored somewhere, after it was evaluated and, of course that somewhere is in a few bytes of RAM that BASIC has allocated from the memory that was not needed for other things. Now in the middle of processing that line of code the computer had first of all to get the number 4 on one hand, then calculate 3 times 3 (probably using it's toes); add the two numbers together (somebody else's toes); check whether there were any more sums to do; and then, and only then, put the result into RAM. Thus releasing the fingers, toes and brain for the next sum.

In the Z80 chip the arithmetical brain is known as the Accumulator or, A register, where any numbers in the range 0 to 255 can be added, subtracted, compared or fiddled about with. Since the A register can only hold numbers up to the value 255, consider what will happen if the Z80 tries to evaluate the sum 250 + 20. The result is like adding say 95 and 8 pence. We get 3 pence and an overflow or carry of 1 into the pounds.

The result of 250 + 20 in the Z80 is 14 with carry i.e.

```
              250 + 20 = 270 = too big.
         therefore 270 - 256 = 14 + Carry.
```

the same in hexadecimal:

```
Carry     Sixteens  Units
              F      A   (15*16 + 10 = 250)
add           1      4   (01*16 +  4 =  20)
   =====================
                     E   ( A+4 = 10+4 = 14)
              0          (15+1 = 16   = 0 + Carry)
  1
=========================
              0      E   with carry.
```

Since there isn't any room in the A register (remember there's just 8 bits) to store the information about the carry condition we have to have another register alongside where interesting things such as Zero, Carry, Positive or Negative states are brought to the Z80's attention. As this register is effectively waving flags at the Accumulator, it is known as the Flag or F register.
These two registers are the best of friends and very little activity in the A register is not known about in the F. However there is one very important exception and that is the simple loading of data into the A register. F couldn't care a fig - he's only bothered about exiting events like adds, subtracts and compares.

That's sorted the brain out, where are the fingers and toes. Well these are the other registers and we have quite a few. Just to make life simple forget completely about RI (that's something to do with refresh and interrupts - all very nasty), forget nearly completely about. IX and IY, and just put to one side for the moment SP (this stands: for stack pointer - very important).

PC stands for program counter and is simply the address of the next byte of machine code that the Z80 would execute. That leaves HL, BC and DE with a repeat underneath alongside another AF.  Those on the bottom line are there just to confuse you.  They are known as the alternative set of registers and represent the fact that the Z80 is very nearly two microprocessors in one with a duplicate set of registers available for use if necessary. Since Microsoft didn't find it necessary to use them to get the TRS80 to work I doubt if we will need them either.

So we are left with HL, BC and DE. These can be used as 6 independent 1 Byte (8 bit) registers H, L, B, C, D and E or as 3 combined 16 bit ones depending on what you want them for. The HL pair ( H stands for High and L for Low - tens and units if you like except the tens are two hundred and fifty sixes) is the cleverest of the three and has a limited brain of it's own. Liken it to the fingers, right hand for H and left for L. The others BC and DE are relatively dumb but we couldn't get along without them. B is perhaps the most useful and has a great propensity for counting. Just the thing for stepping through a program and keeping a count of the number of steps.

Unfortunately it's a little bit more complicated than just pressing the break key to stop in the middle of a machine code program and find out just what are the values held in the various registers but it can be done when using the de-bugging facilities that Zen offers us. It is then that the

'X' display becomes meaningful because it shows us just exactly what the Z80 was up to at the stage of the program when we set the breakpoint to interrupt it. So until we have used the de-bugger to jump into and break out of a program the values displayed by the X command are just as meaningless as telling the computer to PRINT Y would be before we ran the BASIC program that gave Y its value.

Just before we leave the registers we should note that as the register pairs HL, BC, and DE can perform as 16 bit registers, it is possible for them to comprehend a number as big as FFFFH or 65535 and as little as 0. This makes them very useful in pointing to a particular byte in memory for storage or retrieval. HL, remember the pair with very little brain, can also do adds and take-aways in 16 bit arithmetic provided the other number is stored in either BC or DE.
PC (program counter), SP (stack pointer), IX and IY (index registers) are all inseparable 16 bit registers intended for storing memory addresses.

It is best not to try and grasp the whole complexities of the Z8O chip in one go. Instead just get to know each register in turn as you use them in your programs. To start with it is sufficient to know that A holds a small number and can do clever sums with the help of the F, and that HL and BC can hold big numbers. If we need to store a lot of small numbers HL and BC can be treated separately as H, L, B and C.

# Chapter 3

Negative Numbers

Negative numbers are a concept that the Z80 would rather do without. Either there's a 1 there, in which case something is there, or there's a 0 and there's nothing there. The preposterous idea that there is less than something there is just too ridiculous.
But since we humans insist on using this hair brained idea then the Z80 tries its best to comply.

Picture the number 255 Dec or OFF Hex in binary notation, (remembering that that leading zero is just to stop the computer thinking you're swearing at it):

0FF Hex = Binary 1111 1111    Beautiful!

Since we always write our minus signs on the left of a number lets turn the leftmost bit over on its side like this:
-111 1111. Now we are getting somewhere. That's a negative number if ever I saw one; but what is it? The easy way to find out is to add 1 to it and see what happens. Ready?

```
     - 1 1 1   1 1 1 1
add  0 0 0 0   0 0 0 1
     ---------------- Left blank for you to fill in.

     ----------------
```

Stand by to check your answer:
One plus one equals zero with carry.
One plus zero plus carry equals zero with carry.
One plus zero plus carry equals zero with carry.
One plus zero plus carry equals zero with carry.
That's half of it done.

One plus zero plus carry equals zero with carry.
One plus zero plus carry equals zero with carry.
One plus zero plus carry equals zero with carry.
Remember that the minus sign is really just a Bit = 1 lying on its side so the last one:

One plus zero plus carry equals zero with carry falling off the end (advanced students might guess that it didn't really fall off it went into the Carry flag, more about that later).

Answer:   0 0 0 0   0 0 0 0   = 0 Dec.

So if adding 1 makes the original number 0 then the original number must have been -1. (Get out of that.)

OK I know it didn't look like minus one but I did warn you that the Z80 wasn't happy about it either.

The following is the way the Accumulator would store the number -128 if you wanted it to: 1 0 0 0 0 0 0 0 ie 80 Hex. (Try adding 1 a hundred and twenty eight times.)

Minor Diversion to talk about flags:
-----------------------------------
As we have a potentially dangerous situation here, since a number in the A register could be treated as a large positive one (between 121 and 255) or a smaller negative one we had better do some flag waving to check on where we are. To help us is the friendly unassuming F register with a very useful flag which is waved whenever one of the instructions that can affect the flags detects that Bit 7 is set at 1, i.e. negative.

We can use this flag in our assembler program decision making by utilising instructions like:
JUMP if POSITIVE to ADDRESS, or
CALL SUBROUTINE if MINUS.

The trio of flags Zero, Sign and Carry form the backbone of virtually all the logic in most machine code programs.
There are three other flags:

Parity/Overflow or P/V which some clever people understand.
Half Carry or E used in decimal arithmetic I think.
Add./Subtract or N again used in decimal type arithmetic, but as I've never had occasion to use them, their exact function and usefulness hasn't sunk in yet.

Back to working out negative numbers
------------------------------------
Right you've got the picture. Treat Bit 7 (the leftmost) as a sign bit, work out what the others would come to if positive, subtract it from the value of Bit 7 and you've got the negative decimal equivalent. Just to show off the Z80 can do the same sort of thing with 16 Bit (two byte) numbers like 0FFFF Hex or 65535 as well. In binary it looks like:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 as before for the sign:
- 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

If we added 1 to it like last time we would get 0 so this must be the way -1 Dec would be stored as a 16 Bit two byte number.
Now sixteen bit numbers are used a lot in the Z80 for memory addresses and one of the things that baffles most BASIC programmers for a while is the problem of PEEKS and POKES above the 16K boundary. Lets examine the numbers in that

region; firstly the last byte in 16K and then the next one and finally the last byte in 48K (These sizes of course refer to RAM, we must add another 16K to them for the ROM, keyboard, video etc).

```
Top of 16K - 7FFFHex  = 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Next Byte  - 8000Hex  = 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Top of 48K — 0FFFFHex = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

If the Z80 has been told to interpret these as 'human' numbers then the second two are obviously negative. Setting these numbers out with the values of each bit, we get:

```
                           B I T S
Sign    14   13   12   11   10    9   8   7   6  5  4 3 2 1 0
32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

7FFF =
   0     1    1    1    1    1    1   1   1   1  1  1 1 1 1 1
      = Positive = 32767

8000 =
   1     0    0    0    0    0    0   0   0   0  0  0 0 0 0 0
      = Negative = -( 32768 - 0 ) = -32768

FFFF =
   1     1    1    1    1    1    1   1   1   1  1  1 1 1 1 1
      = Negative = -( 32678 - 32767 ) = -1
```

Don't worry if it doesn't all sink in first time. But when it eventually does penetrate the grey matter you will have a sound grasp of how BASIC handles its Integers, known in the trade as 'Signed 16 Bit Integers'.  And you will also appreciate why in BASIC, the range of values that an integer can hold is limited to -32768 to +32767.

FOR ADVANCED BASIC PROGRAMMERS ONLY

```
10 DEFINT A - Z
20 Y = 7 + 256 * 8
30 V = VARPTR( Y )
40 PRINT PEEK( V), PEEK( V + 1 )
50 INPUT "NEW VALUE "; Y
60 GOTO 30
70 END
```

Explanation.

10 Make everything Integers
20 Give Y the value 87Hex and store it in RAM.
30 Find out where Y is stored.
40 Have a peek at what's there.
50 Try your own numbers.

Alternatively you could try POKEing some values into V and V+1 and then finding out what Y had become.

What you should observe is that any value of Y is held in RAM such that the remainder from dividing by 256 is stored at the byte in memory pointed to by VARPTR(Y) and the number of 256's in Y is stored at the next byte i.e. VARPTR(Y)+1.

This is **VERY IMPORTANT** and we shall use this fact in PART 2 with USR routines.

# Chapter 4

Assembler Programming

Like all good stories we must have a beginning, an end and something in the middle. It's the middle part that's rather awkward so let’s dispose of the easy bits first.

First Simple Rule:
------------------
All assembly language programs must finish with the simple statement END. This tells the assembler (more about him later) that it has come to the end and nothing else is of any consequence and can be ignored. Without an END to your program you will get an error message EOF. Of course if you are using the dreaded E....m you will know that just as it was being put on the shelf for sale somebody spotted there was a bit missing and it had to be added on after the END so the END wasn't the end after all and it set the scene for all the other incomplete bits of software from 2.0 to the M....3 Dos.

Second Simple Rule
------------------
All assembly language programs need a beginning which tells the assembler (he's here again) the position, (location or address) in memory for which the machine code program is written. This beginning or origin statement takes the form:
ORG value
Whence 'value' can be any decimal, hexadecimal number in the range 0 0FFFFH. (It could also be a label with an associated numeric value known to the assembler).

This is the big difference in thinking between Basic and machine code.

Basic programs are completely position independent and to a large extent machine or even type of computer independent. Machine code is written not just for a particular central processor but also very likely for a particular configuration of computer and further more to occupy and work in only one part of that computers memory.
This often raises the question "Why bother ?" but we won't go into that just yet.

The value used for the ORG of your program depends on the purpose of the program. If you were re-writing the whole of the Level 2 Rom you would start with an ORG 0, not caring whether it was Hexadecimal or ordinary numbers and then grow old gracefully. Most of the time however we want to create a

program to work in a section of unused memory and a convenient place is the upper part of the 16K Ram as this is not used by the Aculab, cassette or any of the Disc operating systems.
All our programs therefore will be based on an ORG of 7800Hex but of course you could change it if you wanted to.

Our assembly language source code (notice the crafty way these new words creep in) now looks like this:

```
ORG 7800H
?
?
?
END
```

and as some folk might say: "It certainly looks impressive but what does it do ?".
There's a definitive answer to that question, that goes back to time immemorial or at least to Babbage, and it is: "Well it doesn't do anything yet ... but wait till I get the bugs out."

Third Simple Rule

Assembly language programming isn't a bit like BASIC. Don't cry about it - there's nothing anybody can do - and remember you couldn't understand ON ERROR GOTO and PRINT USING once.

Sooner or later we are going to want to change a Hex number into decimal so lets do it with 7800H:

```
7000H =  7   x  4096 =    28672
 800H =  8   x   256 =     2048
  00H =  0   x    16 =        0
   0H =  0   x     1 =        0

7 8 0 0 Hex    =    30720 Decimal
```

Now that wasn't too hard was it ? I leave you to work out for yourself how to do get back from decimal to hex.
There's a tradition build up with the TRS80 that the first machine code program example should be whiting out the screen in no-time-at-all, so just to be that bit different we will fill the video with a chequer-board pattern. In order to achieve this laudable objective we need to know how the video is connected to the computer ... AND ... I don't mean that silly little plug on the end of a bit of wire.

Fourth simple rule
------------------

Most apparently straight forward bits of English don't mean, what they used to mean when talking about computers.
What we mean is the logical connection between the Z80 and the video character generator. Fortunately the Level 2 manual comes to our aid and informs us that the video is treated as if it were a chunk of ordinary RAM of 1024 bytes starting at 3000 Hex (or 15360 Dec) and finishing at 3FFFHex (16383 Dec).

So our object is to write in assembler the nearest equivalent of the following BASIC program:

```
10 FOR X = 15360 TO 16383: POKE X,161: NEXT
```

Now just hold on there a little! That's far too complicated for an assembler programmer to understand. Can't we simplify it a little first ?
Well we could try:

```
10 FOR X = 15360 TO 16383
20   POKE X ,161
30 NEXT X
40 END
```

That's getting better but I'm afraid all this FOR / NEXT business is just a little too .. how shall we say .. intricate .. can't we get there in singles.
Third time lucky ?

```
10 REM Simple BASIC
20 LET HL = 15360
30 LET BC = 1024
40 POKE HL,161
50   HL = HL + 1
60   BC = BC - 1
70 IF BC <> 0 THEN GOTO 40
80 END
```

For those who didn't skip the earlier chapters, the letters used for the variables might seem familiar; they are of course the same as those used for two of the Z80 registers.

Before we can go much further there is one important concept. that we must master and that is the complete irrelevance of line numbers in assembler programming. Whereas in BASIC we have the ability to refer to line numbers in the program to change the flow of control by GOTO's and GOSUB's, in assembler we must identify the particular line with a label and then refer to that label.
Labels or symbols can be called anything you like, they can be as long as you like provided they begin with a letter and aren't one of the assembler's reserved words like END or LOAD etc.
In our BASIC program that we are going to copy the program keeps jumping back round in a loop until BC is exhausted so let us use the label LOOP. To convey to the assembler that a particular collection of letters is to be referred to later

we must add a colon immediately after the label as a means of declaring it to the assembler.

First GOLDEN rule of assembler programming.
------------------------------------------
You will never learn ANYTHING by reading until you have actually keyed it into the machine yourself.

So stop reading, put the cat out, switch on the computer, get your copy of Zen, and System load, @Run or Zen/cmd, now ! Make sure you have a clean cassette, formatted floppy tape or disc with some room on it as we shall save the source file for future use. From here on in <E> means the Enter (TRS80) or New Line (Video Genie) key.

To switch Zen into the text entry mode key: E <E>
(Enter the Editor). Zen responds in a similar fashion to the AUTO mode in BASIC by printing the number 1 for the first line and waits for you to key in anything you like, so key in the following except for the line numbers, which Zen does for you.

```
1 ORG 7800H <E>
2 LD HL, 15360 <E>       (let HL = 15360)
3 LD BC, 1024 <E>        (let BC = 1024)
4 LOOP: LD (HL),161 <E>  (poke HL,161)
5 INC HL <E>             (HL = HL+1)
6 DEC BC <E>             (BC = BC-1)
7 LD A, B <E>            (; test BC in two steps)
8 CP 0 <E>               (if B <> 0)
9 JP NZ,LOOP <E>         (goto LOOP)
10 LD A,C <E>
11 CP 0 <E>              (if C <> 0)
12 JP NZ,LOOP <E>        (goto LOOP)
13 END <E>
14 . <E>
```

The full stop tells Zen that we have come to the end and there's no more text input. Zen ignores the full stop and switches control back to the command level.

Let's step through the program, nice and gently, to see what we've done, forgetting all about ORGS and ENDS for the time being.
2 The LD instruction loads the HL register pair with the value 15360 which is the address of the top left hand corner of the video. The LD part of the instruction is known as the OPCODE and the 15360 part is the OPERAND.
3 Likewise BC becomes 1024, which is our Byte Count.

4 This is the tricky line. LOOP: is a label LOOP because it has a ':' after it. LD we know means load but what does (HL)

imply. Well its beautifully simple (it needs to be) once you get the hang of it. HL by itself means the register pair HL. (HL) means the byte in memory that HL is looking at, so for the first time round HL=15360 and the load instruction means that particular byte, next time round it will be 15361 etc. 161 is simply the value of the graphics character that is to be loaded into the byte pointed to by HL.

5 & 6 Add one (INCrement) to HL and subtract one (DECrement) from BC. Since the Flag registers generally go on strike when we are doing 16 bit arithmetic the Z flag will not be triggered when BC becomes zero, so the only way to find out if the whole of the 1024 in BC has gone is to check B and C separately and if either of them are not zero then carry on looping.

7 Load the clever A reqister with the value held by B.

8 CP means for the Z80 'test for the difference between the A register and' in this case 0. If in testing the Z80 finds no difference then the Z flaq will be set.

9 Is it, because if not there's work still to be done.

10 Check the value in C the same way.
11 Compare.
12 Keep up the good work.
13 Everybody happy ?

Fifth simple rule
----------------
Condition testing causes most of the headaches in assembler.

At the end of our program we would have filled the video completely and the Z80 would then execute the next machine code instruction which, if we do nothing about it would be just whatever the next byte in Ram just happened to be. The fact that we have put an END in our code does not mean that the computer will respond with >READY the way it would in Basic, when control is passed back to the keyboard.

In machine code the only way to stop the Z80 chip is to switch the computer off, this being rather a drastic measure. We can however hand control back to Zen provided we arrange it before we go blasting the video with graphics. How to do it you will find out later.
Meanwhile the video is displaying Z>, which is Zen's way of saying 'Master I am here to do your will - What is your command ?".
We must now understand that Zen has got a singularly one track mind when it comes to manipulating source code. If we want Zen to display, print or alter any line we have to tell

Zen in simple language whether to go to the Top line, or Up so many, or Down so many lines. To review our text which conveniently, is less than one video page enter the following:

```
T <E> (This tells Zen to point to the Top line)
<E>  (Enter by itself tells Zen to clear the Video)
P13 <E>   (Print thirteen lines)
```

At this point lets assume that we have an error in line 4, say there's a semicolon ';' instead of a colon ':' How do we fix it ? First get Zen's mind concentrating on line 4 by keying U9 <E> (Up nine lines) and line four is displayed. Now key N <E> (for New) and depending on what version you have, either line four will be zapped and you can key it in afresh or it will be displayed with the cursor at the end so you can backspace to the error and retype the line from the error onwards. Our program should now look like this:

```
1 ORG 7800H
2 LD HL,15360
3 LD BC,1024
4 LOOP:LD(HL),161
5 INC HL
6 DEC BC
7 LD A,B
8 CP 0
9 JP NZ,LOOP
10 LD A,C
11 CP 0
12 JP NZ,LOOP
13 END
```

To turn these Mnemonics into actual machine code bytes we activate the assembler in Zen by keying A <E>. Disc users will be prompted by Zen for Source File> to which the answer is simply <E>, unless they want to assemble direct from disc and then they wouldn't be reading this chapter.
Then Zen asks OPTION.
Since you might want to just check for errors, or display the assembled code on the video, or write it to cassette as a System tape etc, this is the stage at which you tell Zen where to send the output. Since we want it on the Video, key V <E> and you will get the complete program assembled as shown below except for the headings, or an error message such as OPERAND or HUH ? where Zen can't decipher your typos.

```
Line Memory Object Label Opcode Operand
  # Address Code
  1                       ORG    7800H
  2 7800 21003C           LD     HL,15360
  3 7803 010004           LD     BC ,1024
  4 7806 36A1      LOOP:   LD    (HL),161
  5 7808 23                  INC    HL
  6 7809 OB                  DEC    BC
  7 780A 78                  LD     A,B
  8 780B FE00                CP     0
  9 7800 C20678              JP     NZ,LOOP
 10 7810 79                  LD     A,C
 11 7811 FE00                CP     0
 12 7813 C20678              JP     NZ,LOOP
 13                          END
```

Note that Zen has understood the decimal numbers but in generating the machine code program these have all been changed into hexadecimal.

Now somebody at the back is bound to ask whether it really takes as much effort to find out if something has been done (lines 7-12) as it does to do it (lines 2-6). In all honesty the answer must be 'not quite' but we don't want to run before we know which direction we should be walking.

Lets prove that the program actually works before we go any further. To do that with E.... m or M....80 or very nearly any other assembler the technique is to assemble to cassette or disc, go back to the operating system, (system) load the object code, power up Debug or Supermon and jump to your program start address. Now that is just too much like HARD WORK so we will get Zen to do it all for us.

First we need to tell Zen that the object code bytes should be loaded into memory as they are evaluated during assembly. To do this we need to insert one line of code into our source file between line 1 and 2. Here's how:  Goto line 2 (T <E> for Top, then D1 <E> for Down 1) Zen should be displaying line 2. Key E <E> to enter the editor and we get the prompt 2.   Key LOAD 7800H <E> followed by the full stop . <E>. Assemble the program again, using if you have a printer the E option, so that you can frame the printout for posterity as the your first working machine code program. The result of the assembly should look exactly as before except the line 2 to 13 now have the numbers 3 to 14 and there's the new line 2 'LOAD 7800H'. To check that the code is in memory use Zen's Query command

Q7800H <E>. You should get:

```
7800 21 00 3C 01 00 04 36 Al    !.<...6. !.<...6!
7808 23 0B 78 FE 00 C2 06 78    #.x....x #.x..Q.x
7810 79 FE 00 C2 06 78 xx xx    y....x.. y..Q.x..
```

The xx figures aren't important, except that if we don't intercept the Z80 after its done our program then they are what will happen next. The Ascii representation on the right will be different as Scripsit can't print the graphics that you get on your video.

To execute the program we want to instruct Zen to Jump or Goto the address 7800 Hex and let the program do its thing and when address 7816 Hex is reached to return to Zen. There's a minor complication here as very early versions of Zen used the single command letter G whence you were then prompted for the address, later versions included the address with the G (Goto) like this G7800H <E> and finally the versions of Zen with the Global replace facility use the command format J7800H <E> (Jump). Since you can all refer to your manuals I don't expect that this will be too problematical.

Depending on your version, key one of the following: J7800H <E>, or G7800H <E>, or G <E> followed by 7800H <E>.
Zen will then ask for a Breakpoint, key 7816H <E>, and faster than the eye can blink your video will be filled with spots.

Unfortunately if you didn't buy your Zen from me, the screen will, even more quickly, be cleared so that you might not catch it; but you know the answer to that.
Before we dash off and celebrate lets write the source code to tape or disc. Prepare your magnetic bulk storage device for recording and make sure the cassette, floppy tape or disc has some space on it and key W <E>. Zen will then ask you for a name or filespec as appropriate and then record your source file.     Cassette and floppy tape users have the V command to verify that the recording is a good one and does not generate any checksum errors.

Sixth simple rule.
------------------
Assembler is so fast that five minutes of programming will take at least five hours to get right. This is known as the advantage in speed of Assembler over Basic.

# Chapter 5

## Now that it works lets change it
--------------------------------

But first we will review what we have learnt.

1. The registers A, B, C, H, L, etc. or the register pairs HL, BC, etc. can be loaded with a value using the LD instruction (LD BC,1024).

2. The contents of the registers can be incremented or decremented (INC HL).

3. Data can be transferred between the registers, again using the LD Instruction (LD A,B)

4. The contents of the A register can be compared with some other value (CP 8), and if they are the same the Zero flag is set (1), otherwise it is reset (0).

5. Instead of executing the next instruction the program can jump to a different location that has been identified with a label.

The next few pages are going to be the hardest to grasp for some time, so make sure you've got some peace and quiet before you start; and until you've grasped OR, XOR and AND you'll never make the grade.

These three very powerful instructions do Bit by Bit checks and comparisons between the A register and some other data. The other data can be virtually anything from a fixed numeric value, the contents of any register (including the A itself) or the contents of a byte in memory pointed to by either the HL or one of the index registers (IX or IY)

Let us assume that the A register contains 49 decimal and we will see what happens if we carry out OR, XOR and AND with 26 decimal. Now 49 Dec = 31 Hex and 26 Dec = 1A Hex
As the instructions are going to operate at Bit level we had better convert to binary 0's and 1's.

```
49 Dec = 31 Hex =   0 0 1 1 0 0 0 1
26 Dec = 1A Hex =   0 0 0 1 1 0 1 0
```

With all three instructions we ignore the 0's and fix our minds on the 1's.

The OR instruction means take the 1's from either the first value OR the second OR both and put them in the A register.

```
So we get 0 0 1 1 0 0 0 1     = 31H
     OR   0 0 0 1 1 0 1 0     = 1AH
          ---------------
          0 0 1 1 1 0 1 1     which is 3B Hex.
```

Its something like the 'lowest common multiple'. In that the result is bigger than either of the two starting numbers and it has got 'set' bits common to both.

The XOR instruction, abbreviated from EXCLUSIVE OR, means take the 1's from either the first value OR the second value but EXCLUDE those where both values have a 1, and put the result in the A register.

```
So we get 0 0 1 1 0 0 0 1     = 31H
     XOR  0 0 0 1 1 0 1 0     = 1AH
          ---------------
          0 0 1 0 1 0 1 1     which is 2B Hex
                ^
                this bit was excluded.
```

This operation can be considered most like a two-way switch. Any bits that are set in the second number will ensure that the result for those particular bits will be the dead opposite of whatever they were with the first number and visa-versa. The most commonly seen example is blinking 'star' on the cassette load.

The AND instruction is simpler and means take the 1's where they occur in BOTH the first value AND the second value with the result of a 1 into the A register.  Otherwise a 0.

```
So we get 0 0 1 1 0 0 0 1  = 31H
     AND  0 0 0 1 1 0 1 0  = 1AH
          ---------------
          0 0 0 1 0 0 0 0   which is 10H
                ^
                this is the only bit set in 31H AND 1AH
```

This operation is very akin to a filter. Any bits set in one of the numbers that don't match those set in the 'filter' are not set in the result.

Now part of the importance of these operations lies in the fact that the Zero flag in the F register is triggered to set or reset depending on whether the resultant value in the A register is zero or not. Also as none of the three instructions can generate a carry condition the C flag is automatically reset irrespective of the final value.

Back to our program. We left it with a lot of coding just to check if the BC register pair had reduced to zero. Consider now what would be the result of doing an OR operation between the B and C registers. If either of the registers had just one bit not equal to zero the final result would reflect this condition and it would not be zero. So we now have a means of checking BC in one go .... or have we? Well we nearly have; remember one of the values for the OR command needs to be in the A register first, so it looks like this:

```
7 LD A,B  (load the contents of the B register into A)
8 OR C         (test if either of them is not zero)
9 JP NZ,LOOP
10 END
```

You have now been introduced to the true assembler's abiding passion - trying to save the odd byte here and there. In general the main objective of Basic programmers is to get their creations working without any error messages. Until the OM appears very few ever worry about how much memory is being used.

With true machine code it is very different. Whilst the program is being assembled the hard facts are there on the screen and every byte that is used is open to criticism. There are of course the people who use macros, relocatable object modules and linking loaders who neither know nor care about such niceties, but we will have none of their ways of working here.

The designers of the Z8O haven't finished with us yet though. Being true men of spirit, they have provided yet another means of saving one byte out of our program. The instruction JP NZ,LOOP takes up a hulking three whole bytes just to tell the Z80 to jump back six or so steps. We have available a two byte instruction that will do just that. It uses the OPCODE JR (Jump Relative) and is used in exactly the same way. Since in two bytes we cannot include all the details for an instruction to jump and the address of any of the 65536 addressable memory locations the use of this command is limited to jumps of plus 127 or minus 128 bytes.

Now we have a plan so power up Zen and read in the old source file by keying R <E> and answering the prompts for a filename. Since there are only a few lines to change at the end of the program the simplest way is to zap the lines we don't need and key in the new ones. Get Zen pointing to line 8. CP 0, which is the first of the ones we don't want and key Z5 <E> (zap five lines). This should leave you with just

END for line 8. Press E <E> to enter the editor and key in the following lines:

```
8 OR C <E>
9 JR NZ,LOOP <E>
10 . <E>
```

Again, note the use of the fullstop to terminate text entry. We have now got it into the most compact form so just to make life difficult we are going to incorporate this extremely useful routine (?) into a USR function from Basic.

There are, as you might have guessed just a couple of problems.
 i) How do we link our program to Basic ?
ii) How do we ensure our program will not upset Basic ?
Lets tackle the second one first. When Basic hands control over to a USR routine it always ensures that there is nothing important in the AF or HL registers, so we can do what we like with those. As for the BC and other registers we do not know whether what they contain is of any consequence or not so we had better save the contents of BC before we use it and restore whatever was there when we are finished.

Fortunately the Z80 has a very useful technique of saving and restoring the contents of a register pair. It is known as the 'stack' and works a bit like a spring loaded money-box allowing you to push values into storage and pop them back again afterwards. The stack is also used by the Z80 itself for remembering return addresses from subroutines.
Most machine code programmers at one stage or another get their pushes and pops crossed with their calls and returns generating something that resembles more of an infestation than a bug, but more of that later. Sufficient to say that with our program all we need to know and do is PUSH BC at the start and POP BC at the end.

But what happens at the end ? How do we get back to Basic ? Well the way Basic processes a USR statement is to call it like a subroutine. A subroutine in machine code works just like one in Basic and needs at least one happy return, abbreviated to RET.

Make the necessary changes to your source file so that after assembling it looks like this:

```
1                        ORG  7800H
2                        LOAD 7800H
3 7800 C5                PUSH BC
```

```
4 7801 21003C           LD   HL,15360
5 7804 010004           LD   BC,1024
6 7807 36A1     LOOP:   LD   (HL),161
7 7809 23               INC  HL
8 780A 0B               DEC  BC
9 7808 78               LD   H,B
10 780C B1              OR   C
11 780D 20F8            JR   NZ,LOOP
12 780F C1              POP  BC
13 7810 C9              RET
                        END
```

If you haven't got a printer write down the object code bytes: C5, 21, 00, 3C etc. as we are now going to change the ORG to another value, say 8700H and re-assemble. Comparing the object code generated for the two addresses should show no difference.
This is **VERY IMPORTANT**. It means that our program, unlike most m/c ones, will function correctly anywhere in memory. We are not tied down to a particular block of Ram that might be needed for something else.

Now as for linking it with Basic we have a number of different techniques available but all of them require us to know the individual values of our code in decimal rather than hex. If you are using E.... m, M....80 or one of the other very powerful assemblers then you will have no doubt large reference tables in the manuals to help with the conversion process. That's not how we do it with Zen however, simply get back to Basic and the TRS80 will do it for you.

To get to Basic from cassette or Aculab do a Jump or Goto memory address 114, from disc Zen return to Dos first and then Basic. Having got to >Ready prompt, key the following to extract from memory the decimal values of the program that Zen loaded into memory on the last assembling operation:

```
AD = 30720 :FOR X = 0 TO 16 :PRINT PEEK(AD + X); :NEXT
```
and you should get:

```
197 33 0 60 1 0 4 54 161 35 11 120 177 32 248 193 201
```

This is obviously some very important data so let’s turn it into Data statements:

```
60 DATA 197,33,0,60,1,0,4,54,161
70 DATA 35,11,120,177,32,248,193,201
```

If you haven't forgotten we discovered earlier that Basic stores integers in memory with each one being allocated two bytes. Well since we have seventeen bytes to put somewhere lets reserve some memory for 9 integers by setting up an array US%(8) - US for USR, % makes it integers and subscripts 0 to 8 give nine values i.e. eighteen bytes of Ram, so:

```
10 DIM US%(8)
```

OK we've got 18 bytes - where is it? Well the problem is that it is likely to move about depending on whether or not any other simple variables come along asking Basic for storage space. But wherever it is, Basic will always give us its position via VARPTR, and simply asking for VARPTR(US%(0)) will point to the first byte of the 18.

To put the decimal values back into memory we simply execute, with Pokes, something similar to the Peeks that got them out.

```
10 DIM US%(8)
20 FOR X = 0 TO 16 30 READ C
40 POKE (VARPTR( US%(0) + X), C
50 NEXT X
60 DATA 197,33,0,60,1,0,4,54,161
70 DATA 35,11,120,177,32,248,193,201
80 rest of program
```

There's a 1 in 5000 chance that this routine would fail should the result of (VARPTR(US%(0) +X) be greater than 32767 and what we ought to do is test it first before executing the Poke, but I leave that to you.

When line 80 is reached the whole of our machine code program is safely in memory and we now set up a subroutine at say line 1000 which can be called at any stage in the main Basic program. Disc users have it easy in setting up USR routines and their subroutine would be:

```
1000 DEFUSR = VARPTR( US%(O))
1010 X = USR(0):RETURN
```

Level 2 programmers have a bit more work to do, which involves, as you might have guessed, dividing the VARPTR value by 256 and Poking the result and the remainder in the USR vectors given in the manual:

```
1000 X = INT( VARPTR( US%(0))/256)
1002 PORE 16527, X
1004 POKE 16526, VARPTR( US%(0)) - 256 - X
1010 X = USR(O):RETURN
```

It is necessary for us to set up the USR address every time we need to access this routine due to Basic's memory allocation procedures mentioned earlier.

# Chapter 6

## Using the ROM Routines

So far we've learnt how to load registers and memory, count, jump and perform certain logical operations like ADD and XOR etc.
Now for the machine language equivalent of Basic's GOSUB.
In machine code these are still called subroutines but instead of accessing them with a GOSUB we use a CALL instruction. As with the jump commands we must provide a reference to the memory address either absolutely in hexadecimal or by means of a label such as CALL SUBR1 or CALL OUTPUT. Just like in Basic when the subroutine has done its job it needs to instruct the Z80 to return to the command immediately following the call and this is achieved with a RET instruction.

Now the Level 2 ROM is jam-packed with hundreds if not thousands of subroutines, most of which are inextricably interlinked and which form the Basic Interpreter. Some of these routines however are available for general use and the first ones we shall meet are to do with data input from the keyboard and data output to the video and printer.

The easiest to get to know is the routine for printing a character on the video at the current cursor position and moving the cursor on to the next. It starts at memory address 0033H and as the leading zeros aren't necessary it is nearly always referred to as simply 33H.

This routine will save us a lot of work as it not only finds the current cursor location, puts our character there and increments the cursor but it also looks after any scrolling of the video that should be needed if we do any printing on the bottom line.
To use 33H we simply load the A register with the character we want to display and then CALL 33H.
On encountering our CALL instruction the Z80 pushes the address of the next instruction onto the stack so that it knows where to come back to, and then it jumps to 33H and starts following the machine code instructions there.

Let’s test this routine to prove that it really works. Use R to kill off any source file you have in memory and then enter the editor and key in the following code:

```
1              ; TESTING ROM 33H
2              ORG 7800H
3              LOAD 7800H
4 7800 3E2A    LD A,'*'
```

```
5 7802 CD3300  CALL 33H
6 7805 00      NOP
7              END
```

The semicolon in line 1 is the Assembler's equivalent to REM in Basic, and the NOP pseudo-op generates a 00 byte, which is a No Operation code as far as the Z80 is concerned. We've put it there just to give us an address for the breakpoint when we test the routine.

Assemble the program and check, with Q7800H, that it has loaded into memory. Now before we jump or goto 7800H to test it, use X to examine the registers first and pay particular attention to the values of AF
and DE. Now jump to 7800H and set a breakpoint at 7805H. Key X again to examine the registers and the video should look like this:

```
Z>X
 HL   DE   BC   AF   RI   IX   IY   SO   PC   Flags
0000 0000 0000 0000 0000                     -------
0000 0000 0000 0000 0000 0000 0000 5627 402D -------

Z>J7800H
Brkpnt>7805H
Z>X

 HL   DE   BC   AF   RI   IX   IY   SO   PC   Flags
0000 401D.0000 2A20 7200                     --1----
0000 0000 0000 0000 0000 0000 0000 5627 7805 -------
```

Well apart from the fact that only the most recent disc version of Zen has the Flag register decoding, what are we to make of this ?
Firstly, the routine worked and if we look hard enough we can see that the "*" got printed at the cursor position following the last input before Zen jumped to our program. Which, since our last keystroke was an <Enter> or carriage return, was at the start of the line after the 'Brkpnt>7805H".
Secondly the AF, DE, PC and R registers came back with different values to those they initially had on leaving Zen. The A register was of course altered by our program but what about the others ?
We should have expected the F register to alter as that register responds to most sorts of activity, except loading memory and registers, and there must have been a fair bit of it going on with checking on cursor position, displaying the character, incrementing the cursor etc. Likewise the Refresh register and Program Counter have simply behaved in the

normal way they would be expected to, but what is 401D doing in the DE register pair ?

Well it's a long story but here goes: Just imagine that you were designing the TRS-80 and were writing the Level 2 ROM. You are faced with the problem of implementing a video display routine using just upper-case characters but you know that sooner or later the computer will be fitted with a lower-case character generator about which you have no information. It is impossible to write a display routine that will handle a chip about which you know nothing and once the Rom is finalised there's no way of altering it.

The solution is to have dedicated address in Rom which will point to your fixed upper-case only Rom video driver program. So that when we want to display a character we call a fixed address in the computer, such 33H, this in turn looks to an address in Ram, say 401DH, to find out where the video driver routine is really located.    Normally of course the answer will be the address of your routine in Rom but if some time hence a computer has been fitted with lower-case or there's some other reason for not using the Rom routine then all that is necessary is to put the details of the new display program into 401DH and everything will still work perfectly.

There are actually 8 bytes starting at 401DH dedicated to the video display routine. This block of memory is called 'Device Control Block' usually abbreviated to DCB. It contains some identifying information, the address of the actual subroutine to handle the character output, the current cursor location and a few other bits and pieces.
There are two other DCB's in that area doing much the same thing, at 4015H there is the keyboard DCB and at 4035H the lineprinter DCB.

Keyboard debounce was a problem on the early TRS-80's and a number of programs were written that intercepted the keyboard scan by picking up the address from the DCB and simply executing a delay count before carrying on with the Rom routine. Likewise if your printer needs some special treatment then all you have to do is to put the address of your printer driver module into the DCB at 4026H and all printing from Basic or machine code programs will go to your routine. (all that is, that do their printing properly by calling the ROM).

So what happens when we call 33H. Well the Rom loads the DE registers with the address of the Video DCB (401DH) and jumps to a sort of DCB handler routine that looks at the DCB, checks on whether it is input or output, most of the other registers are saved on the stack, the actual driver program address is loaded into HL and then we jump to where HL is pointing. On return all the saved registers are restored and we are back to our program - except that DE is different.

This highlights one of the problems of using somebody else's subroutines, in that occasionally they use some registers, changing the values in them that we had put there and then not restoring them on the return to our program. We must therefore be very certain, before we use any ROM routines, which registers they use and whether or not they are restored on return.     Now, if we refer to Tandy's documentation they tell us that the IY register may be altered as well, so we must be prepared to look after both DE and IY if we want to use 33H.

Since we will probably use it over and over again during a program we may as well set up a general purpose subroutine labelled, say, OUTPUT that we can simply CALL for output. It will look like this,

```
OUTPUT:PUSH DE
       PUSH IY
       CALL 33H
       POP IY
       POP DE
       RETURN
```

By pushing the contents of DE and IY onto the stack, we can allow the ROM to do what it wills with them as after it has finished we simply pop them back off the stack, being careful not to get them crossed over, and thus we are in total control.

There are two other routines concerned with data input/output that are used in a very similar fashion. The first is located at 3BH and outputs to the lineprinter. The difference between this and the one for the video is that 33H is bomb-proof and if we call it then as sure as eggs is eggs the character in the A register will appear instantly on the video even if the whole screen needs to be scrolled, if we call 3BH and a printer isn't connected then the computer will hang up for ever and a day waiting for one to be attached.    And what's more even if the printer is there and working some of them don't get round to actually doing any printing until a carriage return is sent to signal the end of the line.

The second routine is, of course, to do with keyboard input and is located at 2BH. Keyboard input is somewhat different from data output in that should a key not be being pressed when we call this routine then there will not be any data. This keyboard scan is therefore identical in approach to the INKEY$ function available in Basic.

To use it we must first save the DE and IY registers, then call 2BH, restore the registers and test the contents of the A register to see if it contains any data. It is then up to us to decide if we want to hang around repeating the operation until a key is depressed or to carry on with something else and come back to the keyboard again later.

A typical scan the keyboard / display on the video routine with our usual ORG 7800H and LOAD 7800H at the start might look like this:

```
1                        ORG 7800H
2                        LOAD 7800H
3 7800 CDOA78  GETCRAR:  CALL KEYSCAN
4 7803 B7                OR A
5 7804 28FA              JR Z,GETCHAR
6 7806 CD1478            CALL OUTPUT
7 7809 00                NOP   ; Just to give a breakpoint
8                              ; Rest of Program
9 780A D5     KEYSCAN:   PUSH DE
10 780B FDE5             PUSH IY
11 780D CD2B00           CALL 2BH
12 7810 FDE1             POP IY
13 7812 D1               POP DE
14 7813 C9               RET
15 7814 D5     OUTPUT:   PUSH DE
16 7815 FDE5             PUSH IY
17 7817 CD3300           CALL 33H
18 781A FDEl             POP IY
19 781C D1               POP DE
20 781D C9               RET
21                       END
```

Test this routine as we did the others by Jumping or Going to 7800H and setting the breakpoint at 7809H. What should happen is that the TRS-80 will appear to go into suspended animation until a key is depressed whence the character corresponding to the keystroke will be displayed and then control returns to Zen from the breakpoint.   Let's look at the program in more detail.

Line 3 : This is a call to our subroutine KEYSCAN. Note we have not saved any registers so should there have been anything important in A or F then we have lost it.

Lines 9 to 14 : This is the subroutine KEYSCAN. First the DE and IY registers are pushed onto the stack and the ROM keyboard scan is called. If a key is pressed whilst 2BH is executing the character will
be in the A register, otherwise it will contain a zero. The state of the flags in the flag register are not meaningful.

The IY and DE registers are popped off the stack and then the routine returns.

Line 4 : We must now check on whether the A register contains a zero which means no key depression, or non-zero. To do this is it necessary to trigger the Flag register to the contents of the Accumulator. The quickest and easiest method is simply to OR the A register with itself. If A is zero then the Z (zero) flag will be set, otherwise it will be NZ.

Line 5 : If the A register is zero then jump back and do it all again, otherwise carry on. This call, test and jump back loop, is typical of a keyboard input routine.

Line 6 : The A register now has the character for the video so call the display routine.

Line 7 : Just there to give a breakpoint address.

Lines 15 to 20: The video display routine. Saving DE & IY and calling 33H, then restoring IY & DE before returning.

Examining the registers before and after we test this program should show that the only ones that do not return their initial values are the A and F ones, where the A will contain the keystroke character and the F could be anything.

The problem with this routine is that it is limited to just one keystroke and more often that not we will want to input something more meaningful such as a name or a quantity. The first thing we need to do is to keep on accepting characters until the <Enter> key is pressed.
Insert the extra lines to the program so the beginning looks like this:

```
1                       ORG 7800H
2                       LOAD 7800H
3 7800 CD0E78 GETCHAR:  CALL KEYSCAN
4 7803 B7               OR A
5 7804 28FA             JR Z,GETCHAR
6 7806 CD1878           CALL OUTPUT
7 7809 FEOD             CP 13
8 780B 20P3             JR NZ,GETCHAR
9 780D 00               NOP
10                      : Rest of Main Program etc.
```

Now test it as before but this time the program will carry on accepting characters from the keyboard until the <Enter> key is pressed. We have made some progress but if you think about it not all that much, firstly you will find that by repeated pressing of the back-arrow key you can wipe out everything on the video and secondly, even if we key in something useful, our program hasn't remembered it.

All the characters were only 'known' by the computer for a brief instant between the key being pressed and their appearing on the video. To be really useful our program must store each key depression somewhere so that they can all be handed onto the next part of the program. It must also keep a count of the characters so that backspacing beyond the first one is not allowed. Later on there will be further complications about such things as unprintable characters or those keys which mean special things such as <Clear>, but all of that will wait for the next session.

# Chapter 7

## Putting it on the Video

Now that we are all at home calling the ROM lets develop a few more complete routines to display and input messages. Display first as that's the easy one.

The simplest technique involves a straight forward message ending with a carriage return or CR as it is usually abbreviated. Assembler has a command similar to the LET in Basic for allocating a value and takes the form EQU. In future we will always refer to ASCII 13 as CR and use the EQU pseudo op at the beginning of our programs to initialise it.

OK we want to display a message, say, "PRESS ENTER" in our machine code masterpiece. Obviously somewhere in memory we will have to have these 11 characters stored and we will need to refer to them so we will use the label MESSAGE to have the start address of the message, i.e. the address in Ram of the letter "P". Since our program might also want to display other messages of differing lengths we must include some means of identifying the end of the message and for that purpose we will use the CR.

Review the problem:

1, Put the characters "PRESS ENTER", followed by CR into memory.
2, Start displaying the characters on the video one by one.
3, Keep checking on whether or not a CR has been displayed.
4, If it’s a CR then STOP displaying and return.

It should be obvious by now that what we want is to set up a register pair to point to the message and go round a loop getting the character pointed at, displaying it, checking for a CR and if not incrementing the pointer and looping back. The program looks like this:

```
1                       ORG 7800H
2                       LOAD 7000H
3                 CR:   EQU 13
4 7800 210778           LD HL,MESSAGE
5 7803 CD1378           CALL VIDEOUT
6 7806 00               NOP  ; For Breakpoint
7 7807 50524553 MESSAGE: DB   "PRESS ENTER",CR
7 780B 5320454E
7 780F 5445520D
8 7813 7E      VIDEOUT:  LD A,(HL)
```

```
9 7814 23                INC HL
10 7815 CD1D78           CALL OUTPUT
11 7818 FEOD             CP CR
12 781A 20F7             JR NZ,VIDEOUT
13 781C C9               RET
14 781D D5     OUTPUT:   PUSH DE
15 781E FDE5             PUSH IY
16 7820 CD3300           CALL 33H
17 7823 FDE1             POP IY
18 7825 D1               POP DE
19 7826 C9               RET
20                       END
```

By now you should be so expert at assembling these programs, that jumping to them with the de-bugger and getting back from the breakpoint will be taken for granted, but note the value returned in the HL register of 7813H. This is the next byte in memory after the last character displayed. I suppose that it is worth mentioning that there is really no need to be pushing and popping DE a IY as neither are being used in our programs for anything important, but it should become second nature whenever calling 2BH, 33R or 38B to save these two registers otherwise one day you WILL be caught out.

The fun starts, I suppose, with line 4 where the HL register is given the label Message to take with it on line 5 where VIDEOUT is called. VIDEOUT doesn't get involved with the complications of actually outputting to the video, that's left to OUTPUT. There's enough to be getting on with just picking up the characters pointed to by HL and checking, after OUTPUT has done with them, on whether or not to get some more.

Line 7 is also of interest as we have used Zen's all purpose DB (Define Bytes) pseudo-op. This supercedes the DEFM (DEFine Message) and the DEFB (DEFine Byte) of the more primitive assemblers, and allows the mixture of different operands as long as they are separated by commas. Also notice that for each character in the DB statement there is a corresponding hexadecimal value generated in the object code listing.

VIDEOUT, starting at line 8, is the important routine and has a few points of note. Firstly the HL register arrives with the address of the start of the string of characters for output. The first character is loaded into the A register and immediately HL is incremented. This is a fairly universal convention of maintaining a pointer, be it stored either in memory or in a register, as always pointing to the next character.     Once the current character has been loaded from, or to memory or wherever and reference to it is no longer required, the pointer is 'bumped' to the next one. The output subroutine is then called and whatever was in the A register is now on the video.

We know from earlier programs that the output routine will return with the value still in the A register so we check on whether it was a CR and if not, loop back for the next character. If it was a CR then VIDEOUT finishes with a return.

There is a weakness with this program in that all our strings for output must finish with a CR otherwise VIDEOUT doesn't know when to stop. To make our routine more useful it would be nice if we could detect the end of a string in mid-line so to speak. Since our old friend zero is an easily detected non printable character we can use it to flag the end of a string where we do not want a CR. We must make a small addition to our VIDEOUT program, first: immediately after loading A with each character execute an OR A command and if it is zero then return. Now we meet one of the many aspects of the Z80 that makes life easy for us.  We want to return if the zero flag is set, i.e. the equivalent of: IF Z = 1 THEN RETURN. In assembler it's as simple as RET Z.

Thus we have two exits from VIDEOUT, the first is to leave immediately if the next character it gets is zero and the second exit is following the output of a CR.
The program, with a few added messages now becomes:

```
1                        ORG 7800H
2                        LOAD 7800H
3              CR:       EQU 13
4 7800 211378            LD HL,MESS1
5 7803 CD3378            CALL VIDEOUT
6 7806 211F78            LD HL,MESS2
7 7809 CD3378            CALL VIDEOUT
8 780C 212878            LD HL,MESS3
9 780F CD3378            CALL VIDEOUT
10 7812 00               NOP  ; For Breakpoint
11 7813 50524553 MESS1:  DB   "PRESS ENTER",CR
11 7817 5320454E
11 781B 5445520D
12 781F 54455354 MESS2:  DB   "TESTING»» ",0
12 7823 494E473E
12 7827 3E3E3E00
13 782B 3C3C3C2E MESS3:  DB   "<<.>>>",CR
13 782F 3E3E3E0D
14 7833 7E      VIDEOUT: LD A,(HL)
15 7834 23               INC HL
16 7835 B7               OR   A
17 7836 C8               RET Z
```

```
7837 CD3F78           CALL OUTPUT
783A FE0D             CP   CR
783C 20F5             JR   NZ,VIDEOUT
783E C9               RET
783F D5       OUTPUT: PUSH DE
7840 FDE5             PUSH IY
7842 CD3300           CALL 33H
7845 FDE1             POP IY
7847 D1               POP DE
7848 C9               RET
                      END
```

Testing this version in the usual way will give the following on the VDU. Note that Mess3 follows immediately after Mess2:

```
Z>J7800H
Brkpt>7812H
PRESS ENTER
TESTING>>>><,<.>>>> >>>
Z>
```

There are of course other ways of achieving the same objective. If you should examine a machine code program generated by a compiler or program generator then you will find that usually each message is preceded by a single byte value giving the number of characters for output. This quantity is loaded into the B register and the output routine goes round a loop displaying and decrementing B until B becomes exhausted. There's a nice little Z80 instruction, specifically for the B register, called Decrement and Jump If Not Zero which does it all for us in one go.

Taking our original example and rewriting this way it becomes

```
                       ORG 78008
                       LOAD 78008
                  CR:  EQU 13
7800 210778            LD HL,MESSAGE
7803 CD1478            CALL VIDEOUT
7806 00                NOP  ; For Breakpoint
7807 0C505245 MESSAGE: DB   12,"PRESS ENTER",CR
7808 53532045
780F 4E544552
7813 0D
7814 46      VIDEOUT:  LD B,(HL)
7815 23      VID1:     INC HL
7816 7E                LD A,(HL)
7817 CD1D78            CALL OUTPUT
781A 10F9              DJNZ VID1
781C C9                RET
781D D5      OUTPUT:   PUSH DE
781E FDE5              PUSH IY
```

```
7820 CD3300          CALL 33H
7823 FDEl            POP IY
7825 Dl              POP DE
7826 C9              RET
                     END
```

If we didn't want the CR at the end of the message it could easily be omitted provided we reduce the counter from 12 to 11. This particular technique is used in Scripsit to display all the various prompts and messages on the bottom line. It suffers, as does the earlier example, in needing an extra byte, be it the counter at the beginning or the special marker at the end, to indicate the length of the message string.

To do the job really efficiently we need a technique that somehow can manage without any extra bytes. The secret is in appreciating that all displayable ASCII characters have values less than 80H (decimal range 32 to 127), so if we add 128 to the last character of each string then simply by testing if greater than 127 we will know whether or not the whole message has been displayed.

To add 128 or 80H to an ASCII character causes just the seventh bit to become set, everything else stays the same so by simply testing the condition of Bit 7 will be enough to indicate the end of a string. We must of course make sure that we clear this bit before we try to display the character otherwise we would get some weird graphics instead of letters. Obviously our display routine is going to get a little bit longer and more involved, but the reward is in the saving in memory by not needing an extra byte with every message.

So once again we will rewrite our simple example:

```
                     ORG 7800H
                     LOAD 7800H
7800 210778          LD HL,MESSAGE
7803 CD1278          CALL VIDEOUT
7806 00              NOP  ; For Breakpoint
7807 50524553 MESSAGE: DB   "PRESS ENTE","R".80H
7808 5320454E
780F 5445D2
7812 7E       VIDEOUT: LD A,(HL)
7813 E67             AND 7FH
7815 CD1E78          CALL OUTPUT
7818 CB7E            BIT 7,(HL)
781A 23              INC HL
7818 28F5            JR   Z,VIDEOUT
```

```
13 781D C9                RET
14 781E D5     OUTPUT:    PUSH DE
15 781F FDE5              PUSH IY
16 7821 CD3300            CALL 33H
17 7824 FDE1              POP IY
18 7826 D1                POP DE
19 7827 C9                RET
20                        END
```

Line 6 looks interesting and may take some working out.
First of all the CR has gone just to make it easier to explain.
DB "PRESS ENTE","R".80H the fun comes after the comma "R".80H.
Well here is the best explanation you're going to get.
As we want to manipulate the last character 'R' we have to separate it from the rest of the letters "PRESS ENTE". To generate the letter 'R' with its bit 7 set could have been achieved by using "R"+80H but there is a convention with some sense behind it (that I wont go into here) that it is better to use logical operators wherever possible in preference to arithmetical ones. So we are using the assembler's equivalent to the Z80's OR operation. The full stop '.' is used to signify that the bits corresponding to the letter 'R' are to be ORed with 80H, which will always result in bit 7 being set.

VIDEOUT, starting at line 7, gets the character pointed to by HL but before displaying it must ensure that bit 7 of the A register is not set. This could be done either by the command RES 7,A (RESet bit 7,A) but again the logical operation AND is preferred, so we have AND 7FH,
thus filtering out bit 7.
The character in A can now be displayed, but after being output the A register is of no use to us as a check on the bit 7 end of string marker. To do that we have to refer to the actual string of characters and fortunately the HL register is still pointing to the particular character we are interested in. To test bit 7 then is quite easy with the questioning command BIT 7,(HL). The result will be either a Zero or Non-Zero condition in the flag register. If it is zero we go back for the next character having incremented HL first, if it is non-zero then that was the last character and the subroutine having done its job can return.

There are of course, as one might expect, equivalent complete routines available in the ROM and also for disk users in the DOS, which will output complete character strings to the video with just a single call. But there are problems in that the conventions for treating the end of string markers are not consistent, the ROM routine at 28A7H doesn't work under Newdos and as Tandy do not declare these routines as being available they might change them one day and not tell anybody.

As the amount of coding involved in setting up one's own general purpose output using the official supported 33H is quite minimal, then I would always recommend doing it. That way your programs will need little updating when the Model 99 TRS-80 comes out and should you change computers altogether then the alterations needed, will be kept to a minimum.

# Chapter 8

## Getting it from the Keyboard

Now that we are all at home putting it on the video, the next step is to get it from the keyboard. We got an inkling of the possible complications earlier and now it’s for real.

Prising the cap of one of the keys (only if you have an early TRS-80) is the nearest you will ever come to seeing a 'bit'. The keyboard is logically connected to a block of memory and each key, arranged in sets of eight, makes or breaks a little contact. This sets or clears one bit of the eight that make up the byte at its memory address. It all sounds very complicated and believe me decoding the keyboard is no game for amateurs. Fortunately we don't have to as Tandy have provided 2BH for us to save all the trouble.

Now saving us all the trouble is perhaps not quite as you would express it if you knew what was coming, as all that Tandy have given us is a very fundamental call that just instantaneously scans the keys and returns with the value of whatever key was being pressed. If we weren't quick enough with our fingers then hard luck and what's more we have to save DE & IY first. That's the sum total of the official machine code keyboard input information.

Let’s set ourselves the problem of accepting, from the keyboard, a string of characters that we need to save for use later in our program. As the characters are keyed in we will arrange to display them on the video as that isn't automatic either. We must consider carefully just which keys on the keyboard we are prepared to recognise.  As apart from the normal visible characters from a space (value 20H) all the way through the numbers (30H to 39H) and the alphabet both upper and lower case, there are the arrow keys - penny plain and two pence shifted.  If we don't set our thoughts out carefully at the beginning our programming will have some unpredictable effects.

This Basic program will demonstrate the decimal value returned by these keys:

```
10 X$ = INKEY$
20 IF X$ = “” THEN 10
30 PRINT ASC(X$);
40 GOTO 10
```

With the exception of the up-arrow by itself, which returns a value of 91 as it is a displayable up-arrow character, all of the others including <Clear> and the big white one give values less than 32. These are known as the various control codes for the output devices, such as the video or a printer, and do weird things like clearing the screen, or throwing a page, or blanking out a line, etc. Handling all of that first time is just too much, so to keep things relatively simple we shall restrict our input so that it recognises just the back-arrow (value 8) as a backspace/delete and the Enter key as the end of input.

As we want to retain the input, assume it's the name of the person who is playing our 'Vacuum Deserters' game, we need to allocate some memory space for storage.  As soon as we do this however we are then faced with making sure that the number of characters keyed in does not exceed the size of this buffer.  Just to keep up the excitement we must also provide a prompt on the video such as Enter Name :, we must make sure the rest of the line has been blanked out before we ask for keyboard input and last but not least make sure the cursor is switched on !. Don't laugh, just see Zap 22 in Apparat's Newdos80 version 2 for their Edtasm fix.

A crafty peek at your Leve1 2 manual gives us the three very useful numbers 14, 30 and 31. The value 14 will switch the cursor on, 30 will clear from the current cursor position to the end of the line and 31 will clear from the cursor to the end of the video. We will use just 14 and 30 for the time being.

Taking our previous routine we can easily alter it to display our new message "ENTER NAME :" but lets add the 14 and 30 to the message and see what. happens. We may as well add the storage buffer as well, which introduces us to a new pseudo-op DS or DEFS for Define Storage. This must be followed immediately by the number of bytes to be reserved so let's fix it now at say 25 bytes - just under half a line.

Now we get the tricky bit. We will use the C register to hold the maximum number of characters to key in and HL will have the address of the buffer, so it goes like this:
Set B reg. equal to zero, call keyboard and wait for key to be pressed. If it's a CR jump to end of the routine.  If it's a backspace, deal with it. If it's a character we are prepared to accept (if not ignore it), is there room in the buffer for one more and if so put it there. Display it and go back for another one.

```
                       ORG 7800H
                       LOAD 7800H
                  CR:  EQU 13
                  BS:  EQU 8

7800 213D78            LD HL, MESS1
7803 CD6578            CALL VIDEOUT
```

```
7806 214C78           LD HL, BUFFER
7809 OE19             LD C, 25; Length of BUFFER
780B 0600   TEXTIN:   LD B, 0; Character count
780D CD7B78 KEYIN:    CALL KEYSCAN
7810 B7               OR   A
7811 28FA             JR Z, KEYIN
7813 FEOD             CP   CR
7815 2825             JR Z, TEXTEND
7817 FE08             CP   BS
7819 2000             JR NZ, CHECK
781B 05               DEC B ; If 1st key, makes B neg
781C FA0B78           JP M, TEXTIN ; so back to beginning

781F CD7178           CALL OUTPUT    ; Display backspace
7822 2B               DEC HL    ; Go back one
7823 3620             LD (HL)," "    ; clear character in Buffer
7825 18E6             JR KEYIN

7827 FE20   CHECK:    CP 32     ; If less then 32 dec.
7829 38E2             JR C, KEYIN    ; we don't want it

                 ; We now need A reg to check if Buffer full
                 ; so save the character on the stack first
782B P5               PUSH AF
782C 78               LD   A, B
782D B9               CP   C
782E 3803             JR C, NOTFUL   ; Carry if B is less than C
7830 F1               POP AF         ; Restore the stack
7831 18 DA            JR   KEYIN     ; Wait for CR or BS

7833 F1     NOTFUL:   POP AF         ; Restore character
7834 77               LD (HL), A     ; Put in Buffer
7835 23               INC HL         ; Bump the pointer
7836 04               INC B          ; Add 1 to the counter
7837 CD7178           CALL OUTPUT    ; Put it on the video
783A 18D1             JR KEYIN       ; Go for next one
783C 00     TEXTEND:  NOP            ; For Breakpoint
783D 454E5445 MESS1:  DS   "ENTER NAME :`, 30,14,0
7841 52204E41
7845 4D45203A
7849 1E0E00
            BUFFER:   DS 25

7865 7E     VIDEOUT:  LD A, (HL)     ; As we had before
7866 23               INC HL
7867 B7               OR   A
7868 CS               RET Z
```

```
53 7869 CD7178           CALL OUTPUT
54 786C FEOD             CP   CR
55 786E 20F5             JR   NZ, VIDEOUT
56 7870 C9               RET
57 7871 D5     OUTPUT:   PUSH DE
58 7872 FDE5             PUSH IY
59 7874 CD3300           CALL 33H
60 7877 FDE1             POP IY
61 7879 D1               POP DE
62 787A C9               RET
63 7878 D5     KEYSCAN:  PUSH DE
64 787C FDE5             PUSH IY
65 787E CD2B00           CALL 2BH
66 7881 FDE1             POP IY
67 7883 D1               POP DE
68 7884 C9               RET
69                       END
```

The first point to watch with this program is for disc users in particular but it applies in a general way to all others. With the source file growing larger the end is getting very close to 7800H (use Q to ask Zen for the Start & End) and should the end of the file go beyond 7800H some funny things will happen when we assemble. As the object code is generated the Load command will place it in memory from 7800H onwards obliterating the letter part of the source file and, whilst being perfectly understandable it is damned annoying when it happens. So whenever you are using the Load pseudo-op ALWAYS check on memory usage to ensure you are loading into free memory. Free memory being between the end of the source file and the top limit of high memory, everything below the start of the source file is in use, broadly as follows:

| Start | Stop | Used by |
|---|---|---|
| 0000H | 3FFFH | Rom, Keyboard, video, Printer etc. |
| 4000H | 4350H | (Mod 1) |
| | 4450H | (Mod 3) Level 2 cassette operating system. |
| 4450H | Start of Source Cassette Zen. | |

Or for disc

| Start | Stop | Used by |
|---|---|---|
| 4000H | 51FFH | DISC operating system. |
| 5200H | Start of source | Disc Zen. |

Meanwhile back at the program, TEXTIN starts with setting B to zero and calling the keyboard until a key is pressed. Check first of all if it is the big white one as in that case we can pack up. Next check for a back-arrow as this is the only other key less than 32 that we will accept. Now if it is a back-arrow there are two possibilities, either it is the first keystroke and there isn't anything there to rub out or it isn't and there is. We will decrement B anyway and see what happens.

If B was zero to begin with then we dropped a clanger and it has gone minus so the simple way to recover is to jump back to the very beginning of TEXTIN and start afresh. Note we have to use JP M, instead of JR C, as just DECrementing a register does not trigger the Carry flag.  If B hasn't gone negative then there must be a character to delete, so first of all CALL OUTPUT to do it on the video and then we must move the value in HL back one and zap the character in the buffer, before going round again for the next character.

That has disposed of the keys less than 32 that we are interested in so now to check on what is in the A register and simply ignore anything less by going back to KEYIN.
OK we have a valid character but have we got room in the buffer to store it? To find out means doing sums with the A register but we can't do that without losing the important character in A, so the best thing to do is PUSH it on to the stack. Comparing B with C by putting one of them in A first will give the answer in the Carry flag to the state of the buffer.

We now have two things to do and if we get them in the wrong order we are in trouble. The wrong way would be to POP AF to get the character back and then jump back to KEYIN on No Carry if the buffer was full.  Why wrong?  Well the important carry condition in the F register would be altered when the old values off the stack were popped, so we must decide on whether or not there is room in the buffer before Popping AF.

If there is room, then we put the character into the buffer using the neat instruction load the contents of the memory address in HL with the A register.  Increment the value in HL to point to the next available space, add one to B as our character count and go back for the next character. Nothing to it really, so assemble it and jump to 7600H with breakpoint at 783CH to see it all work.

We can of course tidy up TEXTIN to a little subroutine that simply needs HL and C initialising to come back on return with the string input into the buffer and the number of characters in B:

```
TEXTIN:         LD B,0
KEYIN:          CALL KEYSCAN
                OR A
                CP CR
                RET Z; This is the exit from the routine.
                CP BS
                JR NZ, CHECK
                DEC B
                JP M, TEXTIN
                DEC HL
                LD (HL),' '
                JR PUTONVDU
     CHECK:     CP 32
                JR C, KEYIN
                PUSH AF
                LD A, B
                CP C
                JR C, NOTFUL
                POP AF
                JR KEYIN
     NOTFUL:    POP AF
                LD (HL), A
                INC HL
                INC B

     PUTONVDU: CALL OUTPUT
                JR KEYIN
```

That was the easy one, in case you hadn't noticed, now it gets a little bit trickier as the next problem is inputting numeric data. Instead of any character from the keyboard we now only want to accept the digits 0 to 9, but once having keyed them in convert the digits in decimal format to a numeric quantity that the Z80 can understand. Actually it's not going to be that bad considering what we've already been through.

First of all we can re-use most of the preceding program for reading the keyboard with some additions and alterations to label names etc. The first change is to reject any characters less than '0' with CP '0' rather than CP 32; we must also add an extra test to reject anything bigger than '9' so Check becomes:

```
CHECKNUM: CP '0'
          JR C,  KEYIN2 ;The new name for KEYIN
          CP '9'+1      ;Anything bigger than a 9
          JR NC, KEYIN2
          PUSH AF etc
```

This will allow input from the keyboard and only accept the 0 to 9 digits. If you want to use plus, minus and decimal points then you're reading the wrong chapter. Having got them in though they are still just characters and we have to do some nifty footwork to turn them into a useful numeric quantity.

What we shall do is to start with zero and work from left to right through the string of digits multiplying the value we already have by ten and then adding the number represented by the digit to our value. As the final answer could end up bigger than 255 we shall have to use the HL register pair (remember they can do adds) for most of this operation. Let's address ourselves to the problem of multiplying the contents of HL by ten.

Doubling HL is easy as we have the instruction ADD HL,HL and if we did it again we would have four times. If only we had saved the first value we could add that to get five times and then double it again for ten. Well the DE registers have been lazing about doing nothing let's use them:

```
TIMESTEN:      PUSH HL
               POP DE    ; DE has original HL value
               ADD HL,HL ; HL is now doubled
               ADD HL,HL ; HL is now four times
               ADD HL,DE ; HL is now five times
               ADD HL,HL ; HL is now ten times
```

Just hold it a minute. HL is doing sums, DE is helping, B knows how many digits to process - who's left to tell us where the buffer is with the characters in it? Time to use IX or IY, I suppose and since we've been pushing and popping IY like there was no tomorrow we'll use that one. Assume IY points to our string of digits - this is what you do:

Get the character pointed to by IY into the A register.
Subtract the 30H from it so that we're left with just 0 to 9
Add it to HL - no that isn't allowed!
Add L to A, load L with A, if Carry then increment B
That is all just too messy! Try it this way:

Load the E register with A, make sure D equals zero
Add DE to HL - much tidier.

The only point to watch is that IY never expects to be pointing in the right direction and always thinks that you're going to want the one just to the left or half a dozen to the right like this: (IY-1) or (IY+6). If we really do want the memory address that IY is actually pointing to then we must use (IY+O); so the next bit:

```
     LD A, (IY+O)
     SUB 30H
     LD E, A
     LD D, 0
     ADD HL, DE
     INC IY
```

again for the next digit, so:

```
     DJNZ TIMESTEN
```

This will cause a monumental disaster if for some reason the user had just pressed <Enter> instead of inputting some numbers, as B would start at zero, after the first DJNZ would get the value 255 and we would do the conversion on the whole two hundred and fifty six bytes starting at the buffer address and ending up, who knows where.
We can handle that though by checking on B first; so putting it all together with a prompt, say 'ENTER QUANTITY :' it looks like this:

```
1                          ORG 7800H
2                          LOAD 7800H
3                     CR:  EQU 13
4                     BS:  EQU 8 5
6 7800 211978              LD HL, HOWMANY
7 7803 CD8678              CALL VIDEOUT
8 7806 212C78              LD HL, NUMBUFF
9 7809 0E05                LD C, 5
10 780B CD5578             CALL NUMBERIN
11 780E 00                 NOP       ; For breakpoint halfway
12 780F 212C78             LD HL, NUMBUFF
13 7812 CD3378             CALL CONVERT
14 7815 223178             LD (QUANTITY), HL
15 7818 00                 NOP       ; For Breakpoint 16
17 7819 454E5445 HOWMANY:DB   'ENTER QUANTITY :',14,30,0
17 781D 52205155
17 7821 414E5449
17 7825 5459203A
17 7829 0E1E00
18              NUMBUFF:  DS 5
19 7831 0000   QUANTITY: DW 0
20
21 7833 FDE5   CONVERT:  PUSH IY
22 7835 E5               PUSH HL
23 7836 FDEl             POP IY
24 7838 210000           LD HL, 0

25                    ;Check first if B=0 i.e. no input

26 7838 76               LD A, B
27 783C B7               OR   A
```

```
783D 2813             JR Z, CONVERTEND

783F E5     TIMESTEN: PUSH HL
7840 D1               POP DE
7841 29               ADD HL, HL
7842 29               ADD HL, HL
7843 19               ADD HL, DE
7844 29               ADD HL, HL
7845 FD7E00           LD A, (IY+0)
7848 D630             SUB 30H
784A 5F               LD E, A
784B 1600             LD D, 0
784D 19               ADD HL, DE
784E FD23             INC IY
7850 TOED             DJNZ TIMESTEN
7852 FDEl CONVERTEND: POP IY
7854 C9               RET

7855 0600 NUMBERIN:   LD B, 0
7857 CD9C78  KEYIN2:  CALL KEYSCAN
785A B7               OR A
785B 28FA             JR Z, KEYIN2
785D FE0D             CP CR
785F C8               RET Z
7860 FE08             CP BS
7862 2009             JR NZ, CHECKNUM
7864 05               DEC B
7865 FA5578           JP M, NUMBERIN
7868 2B               DEC HL
7869 3620             LD (HL)," "
786B 1814             JR NUMBVDU

786D FE30   CHECKNUM: CP '0'
786F 38E6             JR C, KEYIN2
7871 FE3A             CP '9'+1
7873 30E2             JR NC, KEYIN2
7875 F5               PUSH AF
7876 78               LD A, B
7877 B9               CP C
7878 3803             JR C, NOTFUL
787A Fl               POP AF
787B 18DA             JR KEYIN2

787D Fl     NOTFUL:   POP AF
787E 77               LD (HL), A
787F 23               INC HL
7880 04               INC B
7881 CD9278 NUMBVDU:  CALL OUTPUT
7884 18D1             JR KEYIN2
7886 7E     VIDEOUT:  LD A,(HL)
                      etc.
```

An alternative approach might be to use the original TEXTIN routine for any input and check it for numeric validity before conversion. This would probably produce a more universal solution but would have added to the complexity. For storing the final answer we have asked for two bytes of memory to be reserved with the pseudo-op DW (Define Word). A Word in Z80 jargon is two bytes (one high order, one low) and in our case we have initialised them at zero. When we test this routine we will discover the peculiar way the Z80 puts the contents of a register pair into memory. In human terms it appears to be back-to-front, i.e. the most significant byte comes second when displayed left to right.   A better way to think of it though is to realise that the highest order (most significant) byte goes into the highest address of the two bytes of storage and the lowest (least significant) byte goes into the lowest address.

Right oh - assemble and jump to 7800H with a breakpoint at 7818H (or 780EH if you want to stop half way), key in any number you want and then check with Q7831H on what has been stored at the location labelled Quantity.

Now that we've assembled some programs with a reasonable number of labels in them let's look at something else Zen has got for us. The first is the simple sorted Symbol Table listing which tabulates all the labels we have used and their hexadecimal values:

```
Label    value  Label    value  Label    value  Label      value

BS        0008  CR        OOOD  CONVERT   7833  CONVERTEND 7852
CHECKNUM 786D   HOWMANY   7819  KEYIN2    7857  KEYSCAN    789C
NUMBUFF  782C   NUMBERIN 7855   NOTFUL    787D  NUMBVDU    7881
OUTPUT   7892   QUANTITY 7831   TIMESTEN 783F   VI DEOUT   7886
```

With the later versions of Zen there is a cross-reference listing capability that lists every label in the symbol table with its value, the line number where it is defined and the line numbers of every reference to it in the whole of the source file:

```
Label     value     occurrence     references
CR         000D          3              50 83
BS         0008          4              52
HOWMANY    7819         17               6
NUMBUFF    782C         18               8 12
QUANTITY   7831         19              14 17
CONVERT    7833         21              13
TIMESTEN   783F         30              42
CONVERTEND 7852         43              28
NUMBERIN   7855         46              10 55
KEYIN2     7857         47              49 61 63 69 76
```

```
CHECKNUM   786D          60              53
NOTFUL     787D          71              67
NUMBVDU    7881          75              58
VIDEOUT    7886          78               7 84
OUTPUT     7892          86              75 82
KEYSCAN    789C          92              47
```

Anybody getting this far without the help of Assembler's Aspirin (Pt No 26-XXXX at your local Tandy shop) is entitled to celebrate.

## Appendix 1

## R O M   C A L L S
---------------------

The following list is simply a catalogue of the Model 1 Rom calls that have come to my attention through various publications and other means. Some of these may be Model 3 compatible and some are bound to be totally different.

```
Address   Action         Comments
------    ------         -------
OOOBH     Where am I?
                         Pops the return address into HL and
                         Jumps to (HL)
                         Used to determine where in memory
                         you are:
                              CALL 0BH
                         HERE:PUSH HL
                              POP IX;   IX now has value HERE

0013H     Device Input
                         Usually used for 1 byte read from
                         disc file. LD DE, File control block
                         of open file CALL 13H; Byte returned
                         in A Zero flag set if no error JP
                         NZ, Disc to error routine.

001BH     Device Output
                         Similar to above for 1 byte to disc
                         ;Character in A register
                         LD DE, File control block
                         CALL 1BH
                         JP NZ, Disc error routine.

002BH     Keyboard scan
                         Scans keyboard and returns character
                         in A Very similar to Inkey$ from
                         Basic.
                         Uses DE and IY registers, char in A
                         reg and Zero set if key pressed.
                         Does not hang around waiting,
                         usually used with a loop:
                         KBSCAN:   PUSH DE
                                   PUSH IY
                                   CALL 2BH
                                   POP IY
                                   POP DE
                                   OR A
                                   JR Z,KBSCAN
                                   RET ; character in A
```

| | | |
|---|---|---|
| 0033H | Video Output | Outputs character in A to current cursor pos. Uses DE & TY:<br>`;character in A`<br>`DISPLAY:  PUSH DE`<br>`          PUSH IY`<br>`          CALL 33H`<br>`          POP IY`<br>`          POP DE`<br>`          RET` |
| 003BH | Printer Output | Exactly the same as for video but this routine will hang up if printer not connected. On Model 3 the hang up can be broken with <Break>. |
| 0040H | Line Input | Keyboard line input terminated with <Entry>.Entry HL points to your buffer, B - No bytes. Exit HL -> buffer and B = No of characters. Uses DE register. |
| 0049H | Key Input | Similar to Keyscan but waits for key stroke. |
| 0060H | Delay | Decrements BC until = 0 then returns. Uses just BC and AF registers. |
| 0072H | Basic Ready | Jump to 72H instead of IA19H. Jumping to 1A19 is OK if you've tidied everything up first. If you haven't and you want to get back to Basic from a System load or other m/c routine 72H does all the housekeeping for you. |
| 01C9H | Screen clear | On model 3 all registers altered. Usually switches cursor off on Model 1. |
| 01F8H | Cassette Off | Switches cassette off. |

| | | |
|---|---|---|
| 0212H | Cassette On | Switches cassette on. It is probably best to XOR A before calling either of these. I don't know how the second cassette via the expansion interface is addressed. 022CH Blink Flashes the '•'. |
| 0235H | Byte in | Reads byte from cassette into A register. |
| 0264H | Byte out | Outputs byte in A register to tape. |
| 0287H | Write sync | Outputs the leader and sync byte to tape. Uses BC registers. |
| 0296H | Reads sync | Reads the leader and sync byte from tape. 0314H Address read Reads two bytes into HL. Used for the load address in system tapes. |
| 37E8H | Printer addr. | Model 1 parallel printer connected here. Model 3 puts printer status here. |

| | |
|---|---|
| For | Bit 7 = 0 Not busy. |
| both | Bit 6 = 0 Paper OK. |
| Model 1 | Bit 5 = 1 Device select (whatever that means) |
| & Model 3 | Bit 4 - 1 No fault. |

Keyboard Matrix

| Address | Bit0 | Bit1 | Bit2 | Bit3 | Bit4 | Bit5 | Bit6 | Bit7 |
|---|---|---|---|---|---|---|---|---|
| 3801 | @ | A | B | C | D | E | F | G |
| 3802 | H | I | J | K | L | M | N | O |
| 3804 | P | Q | R | S | T | D | V | M |
| 3808 | X | Y | Z | | | | | |
| 3810 | | ! | " | # | $ | % | & | ` |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 3820 | ( | ) | * | + | < | = | > | ? |
| | 8 | 9 | : | ; | , | - | . | / |
| 3840 | Enter | Clear | Break | UA | DA | LA | RA | Space Bar |
| 3880 | Shift | | | | | | | |

```
3C00H  Video   If Bit 7 = 1 (is Set) then Graphics
 to
3FFFH  Memory
```

Note :- UA, DA, LA and RA are the arrow keys!
   Viz. up, down left and right