Memory mapping and computer number systems — using the VZ200/300

Bob Kitch

This contribution will hopefully stimulate users of the VZ200/300 (or perhaps other small micros) to think about *what* actually lies behind the keyboard or monitor. Therein resides, not simply a collection of electronic components, but a truly creative, near-art form; only restricted by the users' ingenuity. I also hope to provide a firm foundation for users to understand how they should visualise or conceive the internals of their computer. This will lead to more imaginative and rewarding use of their somewhat meagre hardware resources.

THE COMPUTER can be conceptualised (thought of) on two distinct planes: **(i)** the tangible, mechanical or physical level; and **(ii)** the intractable, esoteric or conceptual level. These two "states" are often synonymously associated with the hardware and software aspects of computing but they are not quite analogous as a brief consideration should reveal.

The realisation that the computer can in reality adopt any position between these two end-states sheds some insight into how useful a computer can be as a problem solving tool or as a creative device.

The computer is a *virtual* machine. It is incapable of doing mechanical work such as that done by an internal combustion engine. Furthermore, a computer can be *configured* via suitable programming to carry out any function that we may envisage for it. Again the analogy with a tool, for instance a spanner, is instructive. A shifting spanner has only one use — it is dedicated to that job (although I have seen some tradesmen use it as a hammer!). The important notion in computing is that our imagination is the limiting factor in determining the usefulness of the computer. We may wish to use it to monitor the security of our home or to create fantasies of our mind in intellectual and role-playing games, to carry out tedious and repetitive number crunching, or to correct text for us — etc. The spectrum of jobs is vast, and increasing almost daily.

Transformation

Somewhere between the conception of an idea and the translation of this into a computer-based chore, lies the fundamental task of the programmer. The use of the operation called "transformation" is vital to the succes of this translation. The transformation procedure takes a particular notion in our minds (the "object") and produces a "model" of this in the computer. The model may be termed the "image". A good computer image is a skilful combination of the joint hardware and software aspects of the particular computing configuration.

Often a number of step-wise transformations are required to reach the desired goal or end-point. The distribution of tasks proportioned between hardware and software depends upon

i) the resources available, and
ii) the particular talents of the person undertaking the implementation.

Electrical engineers tend to solve problems with hardware intensive solutions, whilst programmers often develop elaborate algorithmic software solutions.

Not surprisingly, *transformation* has a well developed and rigorous expression in mathematics where the somewhat allied ideas of *correspondence* (between similar objects) and *function* (connecting objects) have relevance. The box entitled "Transformation Concepts" accompanying this article futher elaborates upon some of the powerful transformation concepts — in layman's language.

The way in which "correspondence" occurs in computer science and with which perhaps most programmers are familiar, lies in the various types of codes and coding principles which are employed to connect the diversity of ideas under software control. Note that in transformations from object to image the direction of the conceptual movement may be in either direction or sense.

Thus *encoding* represents transforming the object into the image and *decoding* represents returning the object from the image. Also, multiple levels of coding are often used, depending upon where we are positioned in the hardware-software spectrum.

Codes

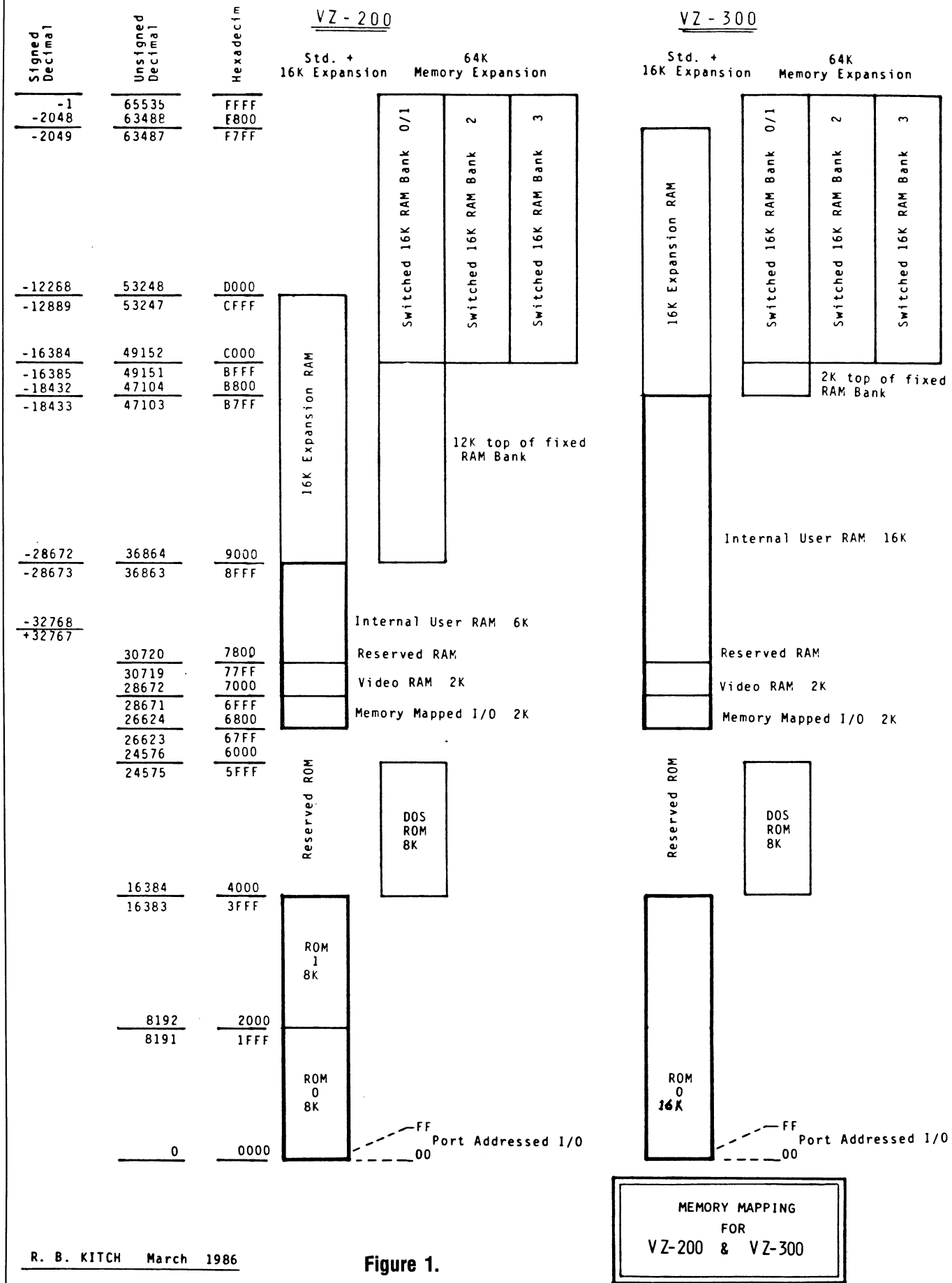
Consider the following code types:

i) Codes used by electronic circuits to perform digital operations e.g: binary codes.

ii) Codes used to convert decimal numbers into binary form e.g: binary coded decimal (BCD) and gray scale.

iii) Codes used to convert decimal numbers and alphabetic symbols into digital form e.g: ASCII, EBCDIC and Baudot code.

iv) Codes used by computers to perform a prescribed series of operations e.g: Z-80 instruction code and PDP8/E.



R. B. KITCH March 1986

Figure 1.

NUMBER BASE CONVERSION & MEMORY MAPPING

In the accompanying article the need to be able to change number representations, according to differing bases, becomes apparent.

Three bases are usually cited and often freely interchanged. These are:

base 10 — decimal (dec./D) uses symbols 0-9
base 16 — hexadecimal (hex./H) uses symbols 0-9, A-F
base 2 — binary (bin./B) uses symbols 0 and 1

The first system is the most familiar to us. The last is the number system of digital computers. The hex system is a convenient intermediate form between decimal and binary systems. (A fourth system to base 8, or octal — using symbols 0-7 — is sometimes employed and is also a convenient intermediate form — see later).

The accompanying table is an indispensable reference for converting base numbers. I always have this chart alongside me when programming — although some people may be fortunate enough to have an electronic calculator with base conversion functions.

Because there are three base numbers, it follows that there are six possible types of conversion. At the conclusion of this box you should be familiar with each conversion and be able to manipulate the resulting numbers.

DESCRIPTION OF TABLE

Table 1 is composed of six columns.

Column 1 (left-hand most) represents single hex digit ranging from OH to FH.

Columns 2 to 5 are labelled Most Significant 3-0 for decimal numbers.

MSO corresponds with 16**0*N (1*N)
MS1 " " " 16**1*N (16*N)
MS2 " " " 16**2*N (256*N)
MS3 " " " 16**3*N (4096*N)

Column 6 is the four-bit binary number corresponding to the hex digit in column 1.

One hex digit can represent half-a-byte (one-*nibble*) of binary information. Hence the close relationship between hex and binary representations. A 16-bit (two-byte) binary number maps onto four hex digits. A single byte maps onto two hex digits. (Octal or base-8 numbers map onto three bits of binary hence an eight-bit binary number can be represented by three octal digits.)

CONVERSION PROCEDURE

A. We will start converting a hex address value into its corresponding decimal and binary values.

1. Converting hex to dec. We will do this using an example. For instance, what is the decimal mapping of address 345CH? Note that the Most Significant Byte (MSB) is 34H and the Least Significant Byte (LSB) is 5CH.

The corresponding decimal for 3H (actually 3000H) appears in column MS3 and maps as 12288D. Similarly, the 4H (400H) in position MS2 maps as 1024D; 5H or 50H maps as 80D in MS1 and finally, CH corresponds to 12D from MSO.

Thus,

```
 3000H ──► 12288D
  400H ──►  1024D
   50H ──►    80D
+   CH ──► +   12D
 345CH ──► 13404D
```

So 345CH maps as 13404D. A little involved, but easy with the table.

2. Converting hex to bin. Remember I said that hex and binary systems are closely related. Again, what is the binary mapping of address 345CH?

```
 3    4    5    C   H — from column 1
 |    |    |    |
0011 0100 0101 1100 B — from column 6
```

So the binary address for 345Ch would be —

MSB 00110100B LSB 01011100B

It could hardly be simpler!

See how difficult it would be to remember binary, but hex is much more concise and memorable?

B. Let us now take a decimal number and convert it into hex and then binary.

3. Converting dec to hex. What is the hex mapping of 22010D? This involves a little scanning of MS3-MSO of the table.

First scan down MS3 for a decimal number which is equal to, or just less than, 22010D. This is seen to be 20480D which maps as 5000H. Subtract this value from 22010D and look for the number just lower than this is MS2. For example 22010D — 20480D = 1530D. The number just lower than this in MS2 is 1280D which maps as 500H. The remainder from this operation is 250D which corresponds to 240D or FOH in MS1. The final remainder is 10D which maps as AH in MSO.

Thus:

```
 22010D
-20480D ──► 5000H
-  1280D ──►  500H
-   240D ──►  FOH
-    10D ──► +  AH
     0D ──► 55FAH
```

It should be easy to convert this hex number into binary equivalent.

55FAH maps as 01010101 11111010 B

C. Let's now start with a binary number and convert it to hex and then to decimal (as previously done).

4. Converting bin to hex. By now you should be getting the idea. Simple isn't it? For example, convert the two-byte address 10011111 11010011B (looks horrible doesn't it?) into its hex value and then decimal value.

```
1001 1111 1101 0011 B — from column 6
 9    F    D    3   H — from column 1
```

Furthermore,

```
 9000H ──► 36864D
  F00H ──►  3840D
  D0H  ──►   208D
+  3H  ──► +   3D
 9FD3H ──► 40915D
```

For those that have been following closely, 40915D is an unsigned decimal and mapped as a signed decimal it is

40915 — 65536 = -24621D
(see later in main article if unsure)

So in summary, we now have four ways of mapping the same address:

hex	9FD3H
unsigned decimal	40915D
signed decimal	-24621D
binary	MSB 10011111B LSB 11010011B

As a final comment and for completeness, it should be said that all the examples given herein are for unsigned decimal numbers in the range of 0 to 65535D. These map onto two-byte numbers ranging from 0000H to FFFFH in hex and 00000000 00000000 to 11111111 11111111 in binary.

The same principles apply for single-byte numbers except that the range of unsigned decimals is reduced to 0 to 255D and 00H to FFH in hex. Only MS1 and MS0 need be used in converting single-byte numbers.

Given this background then, it should be easy to calculate the appropriate values to POKE into addresses 30862D (788EH) and 30863D (788FH) to initialise the USR() command on the VZ. But more of that next time.

If you want some practice in number base conversion and require some additional confidence in following the procedures set out herein then take some addresses from the memory map and practise converting them. (I hope I get them right!)

v) Codes used by programmers to describe a problem to the computer e.g: BASIC, FORTRAN, and SAS.

vi) Codes used by the populace to have work done by a computer which is often transparent to the user. Everyday-type language is often used to communicate to the computer. (i.e: no special skills are required) e.g: POS ('Point-of-sale') terminals or pushbutton data entry panels on microwave ovens etc.

All of these forms of transformation (or coding) describe a relation or function between any object (the notion) and its corresponding image (the programme). Flowcharting is often an intermediate coding step in the transformation process.

The memory image

Towards the hardware end of the spectrum previously alluded to lies the memory or storage system of the computer. Both the programme (or driver) and data are stored in memory which is sequentially addressed in the present generation of Von Neumann machines. Often a successful programmer "needs to get close" to this physical device — particularly in a small microcomputer environment where the memory resource is usually limited. 4K of memory usually requires some smart coding to get a worthwhile programme running — and often in machine code. Larger machines sometimes use a virtual or paged memory system so that the programmer does not need to get close to the hardware limitations. Such things as programme and storage overlaying can be done to make the memory system appear larger than it actually is. The new generation of 16- and 32-bit microprocessors include on-chip memory management functions (e.g: the 80286) to handle memory paging.

The usual way of describing the memory system of a particular computer is via the "Memory Map". This is a transformation of the actual (object) memory chips contained in the computer. This conceptual diagram (image) is an aid for the programmer. It is not a map in the same sense as a geographic (or road) map, but rather it has a one-to-one correspondence with the actual memory system. It does not actually point up any directions in the memory, in the way that a road map does. The memory map is simply a useful programmers' image of the storage which can be accessed by the CPU and the way it is organised.

VZ memory maps

(You thought I was never going to get to it!) Figure 1 is a *Universal Memory Map)* for all the VZ-200 and VZ-300 compluters. These are expandable machines in that additional memory modules, disc systems and various other peripherals can be added onto the standard system. Eight distinct types of machine are detailed:

a) standard "8K" VZ-200 and
b) standard "18K" VZ-300 (both shown in the dark outline)

In the standard machine an area of 10K is reserved for plug-in ROM cartridges. To each of the types can be added:

i) a 16K memory expansion module or
ii) a 64K memory expansion module, and additionally
iii) a disc system containing an 8K DOS can be added which utilises portion of the reserved ROM area.

Thereby eight types of VZ configuration are possible and shown in Figure 1.

A study of the range of memory expansion modules added to the VZ-200 or VZ-300 indicates that they occupy different areas of memory. This clearly shows why expansion modules are *not* interchangeable between models. Fortunately all of the "system areas" are compatible across models — otherwise software would not be transportable. All memory addresses below the reserved RAM (communications area) are the same on either system. This includes video RAM, memory mapped I/O, port addressed I/O and DOS ROM. As most of the peripherals are mapped into the I/O areas, these devices are also compatible between models.

Numbering systems for memory mapping

The three columns extending down the left-hand side of the map are the memory address ranges in the computer that are handled by the Z-80 microprocessor. Again the concept of "mapping" is worth noting — because the CPU uses none of the techniques shown in the columns to actually address memory! The actual (object) addressing method is a 16-bit wide binary sytem which, with suitable decoding, can resolve all the addressing functions necessary. A binary view of the addressing is unnecessarily complicated to obtaining a clear image of the VZ's address space.

An explanation of the three numbering systems used on the memory map follows.

Two forms of decimal (base 10) notation and one of hexadecimal (base 16) are shown. These are image numbering systems of the actual (object) 16-bit binary (base 2) method used by the Z-80 (Port addressed I/O uses only eight-bits of the Least Significant Byte of the address, to uniquely identify the 256 I/O ports).

If you are not particulary familiar with converting or dealing with numbers derived from differing bases, then read the boxes called "Number Base Conversion" accompanying this article.

Unsigned decimal addressing

This number system is shown in the central column of the memory map. It is perhaps the easiest to understand and explain. With a 16-bit binary number as used on the address bus, it is possible to uniquely map 2**16 or 65536 memory locations. These addresses may furthermore be mapped into a one-dimensional vector with memory location OD (2**1-1) mapped at the bottom and memory location 65535D (2**16-1) mapped at the top. This convention of "top" and "bottom" may be inverted — but top of memory is conventionally referred to as the bigger decimal number — so it makes little logical sense to have "top" at the bottom! (Note that some memory maps are drawn in this inverted sense).

Another sense of mapping is apparent and worth mentioning here. This type of map is a byte-mapped transformation as each address is actually eight-bits wide. Most data processing programming deals with bytes as the fundamental units of information. However, the Z-80 can be addressed down to bit level and hence another bit-mapped image containing 524288 (65536*8) bits could be conceived. Some controller applications make use of bit mapping because often the available RAM for programme use is rather restricted and usually the definition or resolution of the process is two-state and can be aptly modelled by a single-bit.

In the unsigned decimal mapping methods, magnitude or size of the address number uniquely defines the location of the address in memory. Relational operators such as "greater than" and "less than" work correctly. This image of addressing is most easily visualised but it bears a difficult relationship to the 16-bit object addressing.

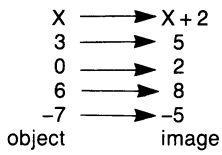
Hexadecimal addressing

This system is shown in the third column and has a stronger relationship to the two-byte wide addressing used by the CPU ▶

TRANSFORMATION CONCEPTS

In a *transformation*, the point being transformed is called the *object*. A transformation *maps* an object onto its' image according to some relation.

An *image* is the result when an object is transformed. e.g:



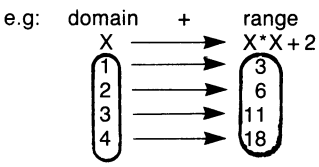
"the image of 3 is 5"

Relations are a way of connecting sets of numbers — a mapping is a special relation.

In a *mapping*, any number in the set being mapped is an object, but the entire set being mapped is usually called the *domain*.

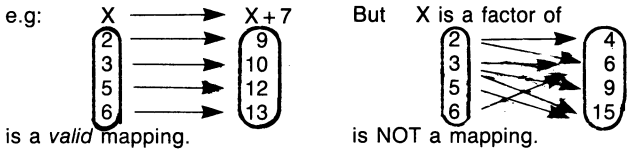
The *domain* of a *function* is a set of numbers mapped by the function.

The domain is the object set.

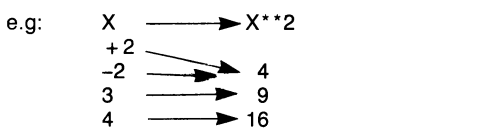


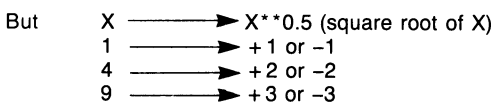
"the set (1, 2, 3, 4) is the domain"

A mapping is a *relation* in which, for every object mapped, there is one, and only one, image.



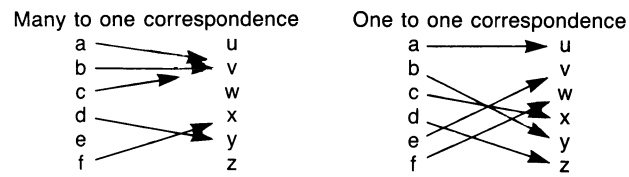
Functions are special relations in which each object is uniquely mapped onto one image.



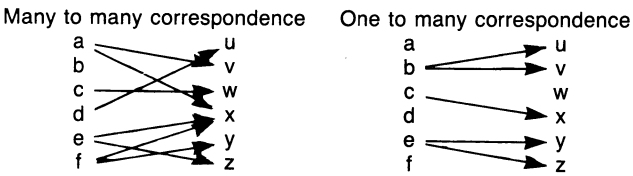


Correspondence has four types:

Mappings are:



NOT mappings are:



bus system. Each nibble (half-a-byte or four-bits) of the address is mapped onto one hexadecimal digit.

Whilst this system may appear a little unfamiliar, it has magnitude and sense — the same as the unsigned decimal notation. Therefore, similar connotations apply to the hexadecimal system as to the unsigned decimal system.

The correspondence between "top of memory" in an unexpanded VZ-200 as being 36863D or 8FFFH should be obvious from the memory map. It is simply a different way (by virtue of the number base difference) of image-mapping the same object.

In certain applications it is more convenient to use decimal notation — and in others it is clearer to use hexadecimal. If it is necessary to get close to the hardware, such as when designing the address decoding for a peripheral expansion, then hexadecimal, with its closer relationship to bus addressing, is better. Alternatively, when a programmer is wanting to locate a routine in memory, there is less need to get close to the machine, (e.g: when PEEKing or POKEing), and the more familiar decimal system is easier. In reality, experienced programmers or engineers readily flip from one to the other — particularly if they have a "smart" electronic calculator with base conversion functions.

Up to this point, all should appear to be logical, orderly and comprehensible. Unfortunately, the people who wrote the Microsoft version of the BASIC interpreter resident in the VZ (and previously used in the TRS-80 Level II, System-80 and PET) must have thought that unsigned decimal and hexadecimal were too logical and easily understood! If you try to PEEK into an address higher than 32767D or 7FFFH you will obtain an "OVERFLOW ERROR" message during run time. A look at the Reference Manual informs you that the valid address range is from -32768D to +32767D. Fair enough, but can one now assme that "top of memory" is +32767D and "bottom of memory" is -32768D. A reasonable deduction, but unfortunately, entirely incorrect! Is our faith in mathematics and logic (relational operators) misplaced?

Signed decimal addressing

The culprit is the signed decimal numbering system shown in the left hand column of the memory map. This number system is closely derived from the 16-bit binary system. The signed decimal numbering is developed from the two's complement binary system which is a method that facilitates the ▶

TABLE 1.

CONVERSION DECIMAL — HEXADECIMAL — BINARY

	Dec.				
	MSB		LSB		
	4096	256	16	1	
Hex.	MS3	MS2	MS1	MS0	Bin.
0	0	0	0	0	**0000**
1	4096	256	16	1	**0001**
2	8192	512	32	2	**0010**
3	12288	768	48	3	**0011**
4	16384	1024	64	4	**0100**
5	20480	1280	80	5	**0101**
6	24576	1536	96	6	**0110**
7	28672	1792	112	7	**0111**
8	32768	2048	128	8	**1000**
9	36864	2304	144	9	**1001**
A	40960	2560	160	10	**1010**
B	45056	2816	176	11	**1011**
C	49152	3072	192	12	**1100**
D	53348	3328	208	13	**1101**
E	57344	3584	224	14	**1110**
F	61440	3840	240	15	**1111**

manipulation of negative numbers. *Do not be overwhelmed if the terms are unfamiliar as it is not essential to understand their derivation.* There exists a simple relationship between the familiar unsigned decimal and the signed decimal systems.

The simplest way of expressing the relationship is that if the unsigned decimal address is greater than 32767D then subtract 65536D from the unsigned decimal value — thereby obtaining a (negative) signed decimal. If the unsigned decimal is less than or equal to 32767D then the signed decimal value maps directly. Expressing this in BASIC is as follows:

UD = unsigned decimal value
SD = signed decimal value

To convert UD to SD:

```
15 IF UD > 32767 THEN SD = UD — 65536
                 ELSE  SD = UD
```

To convert SD to UD

```
25 IF SD < 0 THEN UD = SD + 65536
             ELSE  UD = SD
```

Refer to the mapping in the extreme left hand column of the memory map where the signed decimal system is detailed. Bottom of memory is still OD but top of memory is –1D. A very important discontinuity occurs in the numbering system at mid-memory, where adjacent bytes are numbered 32767D and –32768D. Relational operators do not work in this mapping system.

Suppose one wanted to PEEK into each consecutive memory address over the entire range of memory from OD to –1D (note!). As remarked previously, it is necessary to use signed decimals when PEEKing.

The loop written in BASIC —

```
10 FOR SD = –32768 TO +32767
20 V = PEEK (SD)
30 PRINT SD, V
40 NEXT SD
```

will not provide a consecutive listing of memory. It will commence at the base of the upper half of memory (SD = 32768D) and proceed to the top of memory (SD = –1). It will then leap to the bottom of memory (SD = OD) and proceed to the mid memory (SD = +32767D) position. Not quite what was intended!

To achieve the desired result, the following loop could be written:

```
10 FOR UD = 0 TO 65535
20 SD = UD: IF UD > 32767 THEN SD = UD — 65536
30 V = PEEK (SD)
40 PRINT SD, UD, V
50 NEXT UD
```

This will correctly step-up through memory consecutively from bottom to top (but slowly!)

Uses of the memory map

Having worked thus far through this exposition, what are some of the uses to which the memory map can be put? The first use is when it provides the programmer with a clear image (that word again) of how the addressable memory of the computer is *organised.* A number of advanced programming techniques for the BASIC interpreter also become available. For example, the *utilisation* of the memory by a BASIC programme can be determined. Overlaying of the Programme Statement Table by another routine but with retention of the Variable List Table, becomes possible. Also Assembly Language routines can be loaded into Free Space and called by the USR statement. Overwriting and corruption of programmes (images) can be avoided by reference to the map during loading. If, however, this does inadvertently occur, then the memory map becomes an important load map for debugging purposes.

A more detailed description of the I/O area (including the video RAM) mapping for the peripheral devices, and the communications area would provide more information for advanced programming techniques. Perhaps, with the Editor's indulgence, we may be able to explore these interesting areas at a future date? Meanwhile, get to *understand* your VZ'd, practise number base conversions and let your imagination run with applications for the VZ.