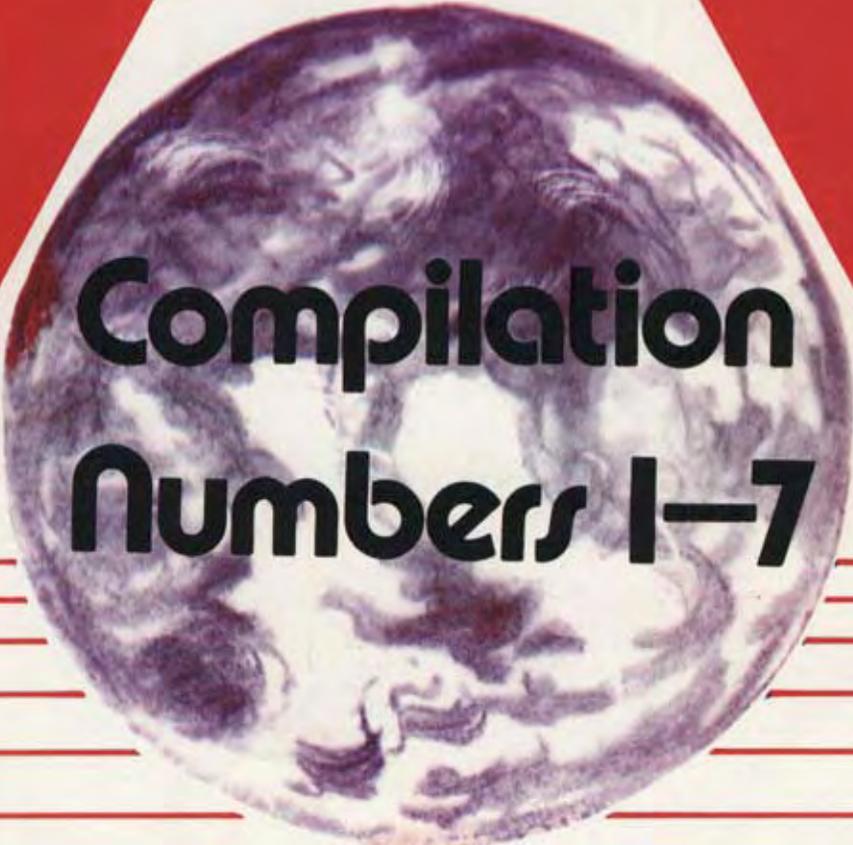best of

nmc news

Compilation
Numbers 1-7

A Special Compilation Issue. Numbers 1-7

Paul had just dropped by, and suggested the idea of a 'Best of INMC' issue. "Would I like to write a small piece as a foreword", he asked. Well, unfortunately, I can't write 'to spec' without considerable thought, and anything I write tends not to be 'that small' anyway. As it was mid evening when Paul dropped by, it was quickly decided that a couple of pints would be ideal for creating the right frame of mind and conducive to nostalgia. Over suitable beverages, we thought about it a bit more, and finally decided that a 'Best of INMC' was perhaps not the cleverest idea, but rather, a 'What's Relevant from the INMC' issue would solve several problems at once. You see, our stocks of the early issues are dwindling, and the recent quote for reprinting them is horrific, so, to save the costly business of reprinting the lot, a compilation of all that's important will likely suit. We still get lots of technical queries, many of which have already been discussed in early issues, so hopefully this compilation issue will help there. New members always tend to buy all the early issues on joining, and much of what is printed refers to early Nasbugs and the like, all of which have long since been superceded. So after much thought, and careful use of the scissors, this is the result, I hope you like it.

Firstly, a bit of early history, going into the depths of time and how Nascoms and the INMC were born.

Let's talk about Nascom first, for without them we would hardly exist. Sometime towards the end of 1976, or early 1977 (I'm guessing here), John Marshall and Kerr Borland were involved in the import of semiconductor devices, mainly discrete transistors. They were looking for a means of adding ICs to their sales, and hit on the idea of a kit microcomputer. Well, both will admit (I hope) that they might have a fair idea of how a micro-computer should be laid out in block form, but as to the actual designing of such a beast, that was beyond them. Dr. Chris Shelton came on the scene, and the original makings of the Nascom 1 were almost literally designed on the back of an envelope. After much development, a couple of prototypes were hand-built on Veroboard, and it was judged that it was time to see what reaction it would have on the great British public. Wembley Conference Centre was booked for November 1977 and the event advertised. It was expected that between two and three hundred people would turn up, and that the reaction would probably be favourable if a little sceptical. In the event over seven hundred people turned up (to the embarrassment of the man who organised the seating) and the reaction was extremely enthusiastic. Orders were taken on the spot for some three to four hundred units, and Nascom 1 was born.

The welcome that the Nascom 1 received was the first of the many problems that have beset Nascom since. It was planned to make five hundred units a year, but to find almost a years production sold in one day caused a major rethink of the market potential of the machine. Production had to be increased by up to ten fold, and if the potential volume were to be that high then a major design rethink would have to take place. All this caused delays, and it wasn't until five months later that Nascom 1s were really in production. Even so, Nascom Is outsold the most optimistic estimates of production, and delays in delivery were common place. In the mean time, thought and funds had to be ploughed into the development of expansion for the Nascom. This at a time when all the available cash could be easily absorbed in the production of the basic product. It was at least a year before expansion memory was available, and when it did arrive, it was soon discovered that it suffered from signal noise on the bus. The phrase 'Memory Plague' was born, although such a disease was never officially admitted by Nascom.

In truth, 'Memory Plague' affected only a minority of Nascom Is, and to be fair to the RAM (A) memory card in particular, RAM (A) has been used with more than a thousand Nascom 2s at 4MHz (double its design speed) with no trouble. Other developments followed, a 'Tiny Basic' was adapted as an interim measure, by an enthusiastic home user, Richard Beal, for use until something better came along. Rights to the popular Microsoft 8K Basic were purchased by Nascom and this was converted for use. Several powerful Z80 assemblers, and a very effective simple text processor (the successor to which prepared this text), all appeared for Nascom. Nascom 1 rapidly became a worldwide success.

Lurking in the background was the successor to Nascom 1, the Nascom 2. This machine, designed with all the original limitations and problems of Nascom 1 in mind is more complex, more complete, employs newer design technology, and (of course) is more expensive. Although what is offered is very much better value for money in the long run than Nascom 1. The trouble was that Nascom 2 was designed on borrowed money, and this was the start of the money problems that then beset Nascom, and at the time of writing, have only just been resolved. Nascom 2 was conceived as a single board machine with 8K of onboard RAM. The initial problem was that the RAM chosen was the MK4118 1K by 8 static RAM, and at the time that was only available from a single source. Any 'hiccough' in the supply of 4118s could completely disrupt the production of Nascom 2, and at the time of the launch of Nascom 2, guess what!! The position was desperate. Here were Nascom with a new machine, potentially as successful as the last, hundreds of orders already placed, stocks of components coming in from all corners, and no Nascom 2s sold because of shortages of one component. The answer adopted was to supply the Nascom 2 with the Nascom 1 RAM (A) card giving the purchaser 16K of RAM for the price of 8K. Very popular with the buyers, but distinctly upsetting to the financial backers as this course was hardly profitable. From then on things escalated. Nascom 2 was (and still is) as popular as its predecessor, but shortages of cash at Nascom continued to get worse. In the end the financial institutions who put up the money for Nascom 2 decided against lending Nascom any more money, recalled their loan, and consequently forced Nascom to call in the Official Receiver.

Nascoms were manufactured by Provemead Ltd., the name the Receiver traded Nascom under for almost a year before Nascom was finally sold to Lucas Logic. What Lucas intend to do with Nascom, only time can tell. In the mean time independent suppliers have not stood still and new products for Nascom have appeared. An independent disk system with CP/M capability is available, and an EPROM card, a colour graphics card, a 64K RAM card, a programmable character generator, a memory mapped graphics card, an intelligent 80x25 video card and lots of other 'add-ons' are already available on the hardware side, with more to come. Software includes an alternative Basic, Pascal, a software 'number cruncher', Sargon chess, and many more.

Nascom has seen it's dark days, but soon the situation is sure to improve. New ideas and new money now back Nascom, and the affects of this suggest an exciting time to come.

And so to the INMC. The idea was mooted by John Marshall and Kerr Borland way back in the very early days of Nascom 1. The idea was to distribute a newsletter containing items of use to Nascom purchasers. The newsletter was to be supported by subscriptions from members, bolstered by Nascom. In much the same way as user's newsletter for other machines are distributed. After about a year of collecting subscriptions from unsuspecting members, the INMC made a shaky start with Issue 1. This was an eight pager containing some items of interest. This issue was mainly prepared by Tony Rundle, who at the time was up to his ears in adapting the Microsoft Basic to Nascom, so from that point of view, it was understandable that the first issue would be short.

It was also apparent that 'something' would have to be done about the INMC. Kerr, as shrewd as ever, decided it ought to be run from outside Nascom, that way it could be more objective. Kerr was also aware of a nucleus of enthusiastic Nascom users who were in the habit of turning up at exhibitions and doing as good a sales job on the stand as Nascom's own staff. What's more, they didn't have to be paid!! Why not try them.

So we went, like lambs to the slaughter. Kerr invited us out for a drink, and after we'd got to the 'matey' stage, he popped the question. Of course we all agreed whole-heartedly. Mind you, we all ended up phoning each other the following morning (or when we came to) to find out whether the bad dream we'd had was true, and if so, what had we let ourselves in for. There we were, a systems analyst, a film editor and a shop manager left with the job of preparing a newsletter for about 2000 avid readers. Paul, then working for Nascom, joined the committee as our technical liaison, and was soon pressganged (What me? - Ed) into being final editor and paste up (final print preparation). Issue 2 followed shortly after, and wasn't much better than issue 1. But the readership came to our aid. Articles started to come in, and we were able to produce a larger issue 3. We started to get lots of letters asking for help with various technical and software problems, and at first we answered them all. These days, pressure of work and sheer volume preclude our answering all but the most important (or unusual) I'm afraid.

And so the INMC grew, membership increased, and the newsletters became more informative. A software library was started and two new members of the committee were co-opted to help. We tried hard to reach our target of six issues a year, but the final preparation of the newsletters is a lengthy process, and we've never made it yet. To compensate, newsletters now average at least 50 to 60 pages an issue instead of the planned 20 to 30 pages. Slowly we drifted away from Nascom as the INMC became more able to support itself, and these days we are to all intents independent of any outside help. Of course, it is to our advantage to foster, and we have the assistance and backing of the various manufacturers and dealers, either in the form of secretarial aid in distribution or advertising, without whose help we would find the task extremely difficult. But hopefully, we are beholden to none.

Now with the recent purchase of Nascom, we look forward to new period of success, and hope that this newsletter will continue support the purchasers of Nascoms and related Nasbus compatible products throughout the world.

D. R. Hunt
Chairman

Editor's bit   Paul Greenhalgh

As David doesn't seem to have quite managed to fill this page I thought that I would take the opportunity to add a bit of waffle myself.

Over the three years of INMC (INMC80 since June 1980) we have had the fortune of having a very dedicated following. There has been a small number of people in particular (Richard Beal, Chris Blackmore (alias Dr. Dark), Rory O'Farrell and David Hunt) without whose constant outpourings the newsletters would have been distinctly lacking. But a vast amount of material and comment has also come from many other individuals, and I would like to take this opportunity on the behalf of all INMC/INMC80 readers to thank these people for their efforts. This compilation issue (produced with much assistance from Chris Stone - ta Chris) contains extracts from all of the INMC issues ever published (1 to 7) and represents only a tiny proportion of the amount of effort so many people have put into the club. Having read this issue I hope that you will be sufficiently interested to join INMC80, and to take part in providing material to what has, I believe, the largest club magazine circulation in Britain.

Happy programming.

10 PRINT   "8K BASIC PAGE"

One feature missing on the 8K BASIC which is extremely useful on the (dare I
mention it) Tandy TRS-80 Level II (which is a 12K extended BASIC by the way),
is an 'INKEY$' command, it's a must for interactive keyboard games etc. What
it does is scan the keyboard once, and if a key is down, return with its value.
It doesn't wait for a key press like the 'INPUT' command. A simple bit of
machine code fixes this.

```
For NASBUG T4
0C80  CD 69 00         CALL KBD      Scan the keyboard once.
0C83  38 04            JR C, PUTINB  If a char, go to PUTINB.
0C85  CB FF            SET 7, A      If no char, set bit 7,
0C87  18 02            JR RET        then go to RET.
0C89  47        PUTINB LD B, A       Put the char in B,
0C8A  AF               XOR A         then clear A.
0C8B  2A 0D E0   RET   LD HL, ABRET  Load HL with return address
0C8E  E9               JP (HL)       and jump to it.
```

It's a bit inconvenient to load that by hand, so lets turn it into BASIC and
let it load itself.


```
10 REM REAL TIME INPUT FOR NASBUG T4
20 DATA 27085,14336,-13564,6399,18178,10927,-8179,233
30 DOKE 4100,3200: FOR I9=3200 TO 3214 STEP 2
40 READ 18: DOKE I9,I8: NEXT


100   REM TO USE
110   Z=USR(0): REM IF Z<0 THEN NO KEY WAS PRESSED
120   REM IF Z>0 THEN Z = THE ASCII VALUE RETURNED
```

For those lucky few with NAS-SYS, things are slightly different, and in
consequence the routine is one byte shorter, thus:

```
For NAS-SYS 1
0C80  DF RST SCAL Internal subroutine call
0C81  61 DEFB #61 Table number for KBD
0C82  38 04 JR C, PUTINB      etc.
```

 Then as for the NASBUG routine

```
In BASIC it is similar to the above
10    REM REAL TIME INPUT FOR NAS-SYS 1
20    DATA 25055,1080,-53,536,-20665,3370,-5664,0
30    DOKE 4100,3200: FOR I9=3200 TO 3214 STEP 2
40    READ I8: DOKE I9,I8: NEXT
```

The routine is used exactly as for NASBUG.

The Nascom 8K Microsoft Basic does not support any kind of 'Format statement',
such as 'PRINT USING', a fact which can be annoying if you wish to produce a
column of figures. For example, if you wish to produce a list of items and
prices and the prices are say ... 12.00, 6.56, 145.02, 0.32 ... then using the
straightforward 'PRINT' command this would appear as:-
12
6.56
145.02
.32

rather than:-
12.00
6.56
145.02
0.32

One way to obtain the column on the right is to use a subroutine to do the
formatting for you. The Basic supports a good set of string functions that
makes the implementation of such a subroutine a fairly simple matter. An
example of such a routine is given below. The variable A (amount) has to be
set to the amount (in pounds) to be printed before the subroutine is called.
It returns with the formatted output in A$. It works for both positive and
negative numbers.

```
1000  A$=STR$(INT(A*100+0.5)) : A=LEN(A$)-1
1010  A$=RIGHT$("        "+A$,9)
1020  ON A GOTO 1050,1040
1030  A$=LEFT$(A$,7)+"."+RIGHT$(A$,2) : RETURN
1040  A$=MID$(A$,2,6)+"0."+RIGHT$(A$,2) : RETURN
1050  A$=MID$(A$,3,6)+"0.0"+RIGHT$(A$,1) : RETURN
```

This tip presupposes that you are a person who assembles machine code in HEX,
in memory, and then wants to convert it to DATA statements for use as machine
code subroutines with the 8K BASIC.
Assuming Nas-sys is in use, try this:

Assemble the machine code routines, in HEX, at the correct location in memory
(and perhaps try them if possible). Go into BASIC (making sure that the free
memory space does not overwrite the machine code routines). Then use the
following command:
      WIDTH 40:FOR A = B TO C STEP 2:PRINT DEEK(A);:NEXT
Where B is the start address in decimal, and C is the end address. The BASIC
will print a series of numbers, stopping well short of the right hand edge of
the screen. Using the Nas-sys edit commands, edit in line numbers and commas,
and the job is done. No mistakes through trying to calculate the decimal
equivalents of the HEX, or through copying the numbers wrong. What's more the
data is in double byte form, which may be loaded down using DOKE commands,
taking half the time that it would take to POKE the data down.

Did you know that line number arguments can be appended to the 'RESTORE'
command in the 8K Basic. Thus:
20 DATA 20,20,37,196,20,53,30
30 DATA 20,21,44,44,196,37,77,60

etc.
.
100 RESTORE 30: IF X=Y THEN RESTORE 20
etc.

Note that in the above example the 'RESTORE' command restores to the second
line of 'DATA', not the first as would be more usual. Further, the 'RESTORE'
to the first line is made conditional upon X and Y.
Interesting ain't it!

Just one more for NAS-SYS users, you will have noticed (and been annoyed by) the fact that you can't 'PRINT' on the top 1line of the screen. This is an unfortunate consequence of the cursor control of NAS-SYS. You get round it this way:

```
10 REM   PLACING TITLES ON LINE 16 USING NAS-SYS
20 Z$="TITLE": FOR Z=l TO LEN(Z$)
30 POKE Z+Z1,ASC(MID$(Z$,Z,1)): NEXT
```

Note that Zl is the start address on the top line
   Min. value of Zl = 3017
   Max. value of Zl = Z1+LEN(Z$) < 3065

So enough about the 8K BASIC for one session, it'll be some time before many of you get round to using this bumpf, and by then you'll have used this newsletter for firelighters anyway.

SOFT SPOTS   By Richard Beal (and others)

Suppose you want to test to see if one of the 16 bit registers HL, DE or BC is zero. For example, this could be at the end of a loop that was counting down. A version of this in a magazine had the following:

```
DEC DE
LD A, D
OR A
JR NZ LOOP
LD A, E
OR A
JR NZ LOOP
```
It would be a lot easier to put;
```
DEC DE
LD A, D
OR E
JR NZ LOOP
```

Suppose you want to jump to a certain address in your program, and that this address is currently stored in the HL register. You can just code:
```
JP (HL)
```
This works very well.

Now suppose that you need to have set HL to a certain value when the jump is made. The method above is useless because HL can't be at two different values at once. So code:
```
LD HL, value of HL
PUSH HL
.
.
LD HL, address to jump to
EX (SP), HL
RET
```

or:

```
LD HL, address to jump to
PUSH HL
LD HL, value of HL
RET
```
The second method is the simplest, but often the value of HL is already on the stack, making the first method the most commonly used.

Now suppose that you in fact wanted to call the routine, not to jump to it as above. In this case the return address can be pushed onto the stack in advance. For example:
```
LD HL, value of HL
PUSH HL
.
.
.
LD HL, return address
EX (SP), HL
PUSH HL
LD HL, address of routine
EX (SP), HL
RET
```

This will call the routine and then return to the specified address. The original value of HL has been preserved and passed to the routine.

If the subroutine decides to, it can change its own return address, again using:
```
EX (SP), HL
```
However, this is bad programming practice because it destroys the structure of CALLs.

It is bad practice to set the stack pointer except at the start of the program where it can be useful for re-initialisation. It is very bad practice to ever use the instruction DEC SP because this implies that you have data stored on the stack at an address less than the current stack pointer (I've used DEC SP to 'throw away' unnecessary data on the stack, I don't see anything wrong with that; Ed.). NAS-SYS 1 used this instruction, and is, I'm afraid, noninterruptable. By noninterruptable I mean that an interrupt at the wrong moment will crash the program. In fact close examination of the NASBUG monitors shows that the storing and restoration of registers is noninterruptable so they are no better.

Do you know the correct name for a 'crash', meaning the loss of control by a program resulting in unpredictable changes to memory locations and usually requiring the reset button to be pressed? The correct name for a 'crash' caused by bad code is a 'program fault'. It sounds more impressive but it is just as annoying!

1. Beginners' Corner
How to move data around-quickly!

I have seen a program which put a message on the screen with a piece of code
like this:
```
LD   HL, address on Screen
LD   A, "M
LD   (HL), A
INC  HL
LD   A, "E
LD   (HL), A
INC  HL
```
and so on ..... and on .... and on
This works very well, but it takes a long time and a lot of program just to do
this simple task.   You should see the last newsletter for an easy way to
display a message using code EF, then the message, then a zero.

But quite often, one wants to move lots of data around and not just output a
message at the current cursor address. By far the easiest way is by the LDIR
instruction. Here is an explanation:

1. Set HL to the address of the data you want to move.
2. Set DE to the address you want to move it to.
3. Set BC to the length of the data you want moved.
4. LDIR instruction - this does the work.

For example, suppose you had 48 bytes of data at address 0E80, and you wanted
to put this on the top line of the screen. The address of the top line is
0BCA.
SO:
```
1.  21 80 0E        LD  HL.0E80  (from)
2.  11 CA 0B        LD  DE,0BCA  (to)
3.  01 30 00        LD  BC.0030
(Length of data, 0030 = 48 decimal)
4.  ED  BO          LDIR
```

1.   Suppose you want to compare HL with DE, without changing the contents of either register.
Try this:
```
B7              OR    A
ED 52           SBC   HL, DE
19              ADD   HL, DE
```
If HL = DE, the Z flag is set, otherwise it is reset.
If HL is greater than or equal to DE, the carry flag is reset.
If HL is less than DE, the carry flag is set.

And it only takes four bytes!

2.   Not everyone has realised that the Nascom monitor program uses the Z80 restart instructions to provide some useful features. Print String is an easy way of putting out messages.

```
EF                         RST      28H
48 45 4C 4C 4F             DEFM     /HELLO/
00                         DEFB     0
```
These seven bytes will make the message 'HELLO' be displayed. Don't forget to put the value 00 at the end of the message, or the screen will fill up with the contents of the rest of your program!

3.   Have you wondered about the meaning of the characters which hex values 00 to 1F give you on the screen? Each one is, in fact, a picture which represents the equivalent ASCII code. For example, 07 is a bell!

4.   The breakpoint command uses a restart to stop the program and display the registers. If you want, you can put the same code, E7 in hex, in several places in your program. You may find it a good idea to fill any empty space with this code, because if you jump to it by mistake, the program will stop, and the register display may give you some clues.

5.   In case all this has been too easy, here is a puzzle for you.
```
AF         XOR  A            Set A to 0
06 00      LD B,0           Set B to 0

3C    LAB1      INC A Increment A
27    DAA        Decimal adjust
10 FC DJNZ LAB1  Repeat, 256 times,
E7    RST BRKPT  Display registers.
```

Now A has been incremented 256 times, and the DAA instruction makes this work in decimal, so A should be 56 at the end.

Why isn't it, and how would you correct the program? (No, the Z80 doesn't have a fault in it!)

SOFTWARE TIPS - SHUFFLING

Suppose you have a list of values which you want to arrange in a random order,
such as shuffling a deck of cards. There are many possible ways to go about
it, and some ways don't perform a truly random shuffle and others take a lot of
programming or a long time to execute. Here are some examples:-

1. SORT METHOD
Attach a random number to each of the items to be shuffled. Then sort the
random numbers into sequence with a sort program.
Then take the original items in this order. This method works correctly, and
could be useful if you have a random number generator, a sort program, and lots
of memory to run it all in and store the random numbers. It is impractical for
our purposes.

2. OBVIOUS REPLACEMENT SHUFFLE
For each item in the list, i, where there are n items (FOR 1=1 to N) generate a
random integer j in the range 1 to N and swap item i with item j. This method
may seem to work but it is incorrect. Some orders are more likely than others.
Never use this method.

3. CORRECT REPLACEMENT SHUFFLE
Here is a correct method!
For each item i in the list of a items, but only up to n-1
(FOR I=1 to N-1), do as follows :-
a) Generate a random integer j in the range i to n.
J = I + INT (RND (1) * (N-I+1))
b) Swap value i with value j.

This method is very fast, is easy to program, works correctly, and runs
quickly. It can be easily programmed in BASIC or in machine code.

```
LIST                                            RUN

100 REM ** DEMONSTRATION OF CORRECT SHUFFLE     10  8  2  1  9  3  7  6  5  4
110 DIM A(10): N=10                             9  2  10  6  3  1  7  4  8  5
130 FOR R=1 TO 8                                1  9  7  4  6  5  8  3  10  2
140 REM ** SET UP VALUES AND SHUFFLE THEM       2  9  3  6  8  7  5  4  1  10
150 FOR I=1 TO N: A(I)=I: NEXT                  5  2  6  7  10  9  3  1  4  8
160 FOR I=1 TO N-1                              10  3  4  6  8  7  5  1  2  9
170 J=I+INT(RND(1)*(N-I+1))                     9  6  7  1  5  8  2  10  3  4
180 T9=A(I): A(I)=A(J): A(J)=T9                 6  1  5  10  8  3  7  4  2  9
190 NEXT I                                      Ok
200 FOR I=1 TO N: PRINT A(I);: NEXT: PRINT
210 NEXT R
Ok
```

Program hints

By N. Ray

<u>ONE OR TWO NOTES ABOUT Z80 MACHINE-CODE PROGRAMMING</u>

Reading other people's Z80 code programs convinces me that many people are unaware of some very useful tricks.

For instance, although it is well-known that XOR A may be used to clear A (and the carry bit) the other operations on A have their uses.

E.g. AND A and OR A set Z and S flags from contents of A and may frequently replace CP 0.

AND A (or OR A) can also be used to clear the carry flag if you don't mind corrupting the other flags. CP A will clear the Z flag.

A warning of general application is in order here.

AND A is <u>not</u> the same as CP 0
INC A is <u>not</u> the same as ADD A,1 (Sim. DEC)
CPL,INCA is not the same as NEG

In each case the code on the left differs from that on the right in the way it affects the carry flag (and certain other flags).

While on the subject of programming dodges, here is an interesting example:

```
E60F   HEXASC:    AND 0FH
C690              ADD A,90H
27                DAA
CE40              ADC A,0H
27                DAA
C9                RET
```

The lower 4 bits in register A are interpreted as a hex digit and the corresponding hex character in ASCII is found in A on exit.
You should try to convince yourself that it works correctly (HINT: watch the half-carry flag).

<u>RECURSION</u>
Recursion can often be used to good advantage in machine-code programming, provided there is ample stack space. A recursive routine is one that calls itself. As a demonstration, consider the 'Tower of Hanoi' problem.

N rings are stacked on one of three poles (as indicated).

The problem is to move the rings from X to Z (using Y) by transferring one ring
at a time from one pole to another, such that a larger ring is never placed on
top of a smaller ring.

A recursive solution is:-
LET   Move N from A to C using B
BE $(IF N=0 RETURN
           Move N-1 from A to B using C
           Take top ring from A to C
           Move N-1 from B to C using A $)

The solution is then a call to 'Move N from X to Z using Y'. The solution is
easily seen to be intuitively correct - 'Take' represents the action of moving
a single ring. We can easily see that for N rings, the routine nests to a
depth N, and Take is called 2^N times.

I have sent a program for inclusion in the INMC library that runs on a minimal
NASCOM and neatly demonstrates the correctness of the routine. It moves one
ring for each press of any key.

In the program BC, DE, HL points at the 'poles' and N is stored in A.

It is arranged that BC, DE, HL always point at the top ring of a
pole.

This program is also an excellent demonstration of the power of the Z80. I am
able to keep all the variables in registers, thus saving on both execution time
and storage space.

<u>MYSTERY PROGRAM FOR NAS-SYS</u>

Type in the mystery program (use the 'L' command with the checksums for error
free loading), don't contract the NOPs at the beginning, they are important.
Execute at 0D00H, and when you get fed up with it, just RESET.

T D00 DB1

```
0D00 00 00 00 00 00 00 00 00 0D
0D08 00 31 00 10 21 00 00 22 99
0D10 7E 0C 21 FF 0F 22 23 0C 27
0D18 3E 0D 52 25 0C EF 0C 20 EE
0D20 20 49 20 61 6D 20 74 68 80
0D28 65 20 4E 61 73 63 6F 6D 1B
0D30 20 44 65 6D 6F 6E 2E 0D 8B
0D38 20 20 50 6C 65 61 73 65 DF
0D40 20 64 6F 6E 27 74 20 74 DD
0D48 6F 75 63 68 20 6D 65 2E 24
0D50 0D 20 20 49 20 6C 69 6B 53
0D58 65 20 74 6F 20 62 65 20 D4
0D60 6C 65 66 74 20 61 6C 6F 74
0D68 6E 65 2E 0D 0D 20 20 57 27
0D70 68 6F 20 61 72 65 20 79 45
0D78 6F 75 20 61 6E 79 77 61 A9
0D80 79 3F 0D 00 DF 63 1A FE AC
0D88 25 20 16 13 DF 79 38 11 A4
0D90 DF 60 7C B5 20 07 21 00 55
0D98 00 22 23 0C C7 22 A5 0D 91
0DA0 E9 2A A5 0D E9 A7 0D EF FE
0DA8 48 65 6C 70 21 20 00 18 97
0DB0 F6 00 00 00 00 00 00 00 B3
```

Get it RIGHT!

Over the last few weeks we have become aware of an appalling ignorance about
NAS-SYS, not only by users like you and me (who should try reading the manuals
even it they are hard to understand at times), but by people who hope to sell
software products for Nascom computers. For instance, we were sent the drafts
of amendments to a book, which incorporated an update for use with NAS-SYS. It
said in effect, 'Throughout the book, whenever you see a reference to one of
the following, change it thus:

```
NASBUG          NAS-SYS
CD 3E 00 (CHIN)   = CD 08 00
CD 3B 01 (CRT)    = CD 4F 01
CD 44 02 (B2HEX)  = CD 19 05
CD 32 02 (TBCD5)  = CD 00 03
etc. '
```

Now this is wrong, WRONG, WRONG !!!!! And one of them is doubly WRONG. If you
can't see anything wrong with that, then you should reread the manuals (go on,
grin, but we bet there are a few red faces out there). The author has missed
the whole point about NAS-SYS. This is not the only instance, we'll be giving
a list of known 'Goodies' and 'Baddies' at the end.

Now one of the major features of NAS-SYS is the fact that to remain compatible
with special versions, and, maybe, later revisions, all calls to internal
routines are handled through a table of addresses. This allows the software
writers freedom to re-assemble NAS-SYS as required, yet still maintain
compatibility with software written for an existing version. It is even
possible to turn NAS-SYS inside-out and for the using software to be unaware of
it.

Here are the rules, they are simple enough:
1) Never ever make absolute calls or jumps of any kind to NAS-SYS.
2) The major routines use the Z.80 restarts, use them properly.
3) Always use the restarts or SCAL (RST 18H) to gain access to all
NAS-SYS routines. STMON is the ONLY exception.
4) If calling routine 'R' using SCAL, make sure you put 52H into ARGX
and that ARGN is set to 00H before the call.
5) Only use routine call numbers 41H to 7CH for full compatibility.
6) If in doubt read the listings and the manuals to understand how the SCAL
routine works.

There, that wasn't difficult was it.

One point, if you are writing software which makes monitor calls, and you don't
know what monitor it is to be used with, make all your CALLs to a table of your
own, using three bytes for each call. Then, to give an example, if you want to
call CHIN in NASBUG, the three bytes will be JP CHIN:
   C5 3E 00
Whereas in NAS-SYS two of the three bytes win become RST RIN, RET, the last is
a 'don't care' byte:
   CF C9 XX
Or in another instance, using B2HEX, the three bytes become JP B2HEX:
   C3 44 02
in NASBUG, whilst the NAS-SYS equivalent is RST SCAL, B2HEX, RET:
   DF 68 C9

A nice example of this approach is XTAL BASIC, which although it appears in our list of 'Baddies', uses just this approach and so, although it makes absolute calls to NAS-SYS, it is very simply corrected (see the review in this issue).

Now for our list of 'Goodies' and 'Baddies'. Richard wrote a special NAS-SYS with all the addresses changed, so we are sure the following do or do not work with other NAS-SYS's (NAS-SYSes, NAS-SYSii ??).

Goodies   Baddies

Nascom ROM and Tape Basic   Xtal Basic
ZEAP 2   Nas chess
Nas Pen   Memory Tests published in Liverpool
Nas-Dis   Software Gazette No. 3
D-Bug

We know of other bits of software coming but haven't yet been able to lay our 'sticky mits on them, so we don't know if they obey the rules or not. So if you are writing software with a view to selling it, DON'T FORGET THE RULES !! Better still, send us a copy, we'll review it, and tell you if any changes are necessary.

All this comes about by failing to understand the manuals, and Nascom must accept some of the blame, there are times when a thorough knowledge of cryptography (and a magnifying glass) would be useful in deciphering some parts of them. Here are some genuine 'howlers' that we have heard uttered by users recently.

Nascom 1 & 2 with 8K Basic:
"Isn't a pity that Nascom Basic always LISTs from one end to the other without being able to break it 'mid-stream'."
Answer:
Use the 'ESC' (shift newline) once to stop the list at any point (any other key will restart the LISTing), and 'ESC' again to break the list.
The same applies whilst running a program.

NAS-SYS:
"How on earth did you open up that line and insert those extra characters?"
Answer:
Shift cursor right and shift cursor left keys will open and close a line.

"I didn't realise that you could make the Tabulate command tab in blocks of say ten, nor that you could escape halfway through a tab."
Answer:
Try T 0 FFFF A, newline, then hit any key a few times followed by an 'ESC'.

Anything except NASBUG T2:
"I'm a typist, and I'm not used to computer keyboards. I find normal lower case is better for me, can I change my keyboard for a lower case one please?
Answer:
Type Kl then carry on. That will cost you the 50.00 I've just saved you on a new keyboard, for the information please. Ta !!

ZEAP 2.0

MODIFICATIONS TO ZEAP 2.0

Here are details of some "unofficial" modifications to ZEAP which we have found useful. These cure two minor bugs, and provide a valuable new feature :-

Bug 1: After an assembly, the margin to the right of the bottom line is corrupt, which enables the cursor to be moved off the screen, and can cause the top line to move when editing the bottom line. This is cured.

Bug 2: When using the H command to set the number of lines, and then repeatedly using the Z command for editing, it becomes necessary to press a key to display the line to be edited. This is cured.

Additional feature: Type a colon (:) and press ENTER. The message "Command?" is displayed, and the system waits for you to enter a NAS-SYS command, which is then obeyed. On completion of this command, control remains within ZEAP. This makes it very easy to use the Read and Write commands to save the source file or object program. It can also be useful to use the Tabulate and Modify commands to examine memory locations. To provide this feature it has been necessary to remove the Q command from ZEAP, but this command has been superceded by the J and K commands.

MODIFICATIONS TO ZEAP 2.0 RAM VERSION
```
1057 3A
1058 CC
1059 1F
1493 FF
182B 00
182C 20
1BBD CD
1BBE B5
1BBF 1F
1BE9 CD
1BEA C2
1BEB 1F


1FB0 1D C3 DD 1D A3 32 FE 0F
1FB8 21 BA 0B 06 10 77 23 10
1FC0 FC C9 32 FF 0F 3A 14 0F
1FC8 32 77 0F C9 EF 43 6F 6D
1FD0 6D 61 6E 64 3F 0D 00 DF
1FD8 63 01 2B 0C 1A FE 20 C8
1FE0 FE 41 38 0D FE 5B 30 09
1FE8 02 32 0A 0C 13 DF 79 30
1FF0 03 DF 6B C9 DF 60 DF 5C
1FF8 C9 00 00 00 00 00 00 12
```

MODIFICATIONS TO ZEAP 2.0 ROM VERSION
```
D057 3A
D058 CC
D059 DF
D814 00
D815 E0
DBA6 CD
DBA7 B5
DBA8 DF
DBCF CD
DBD0 C2
DBD1 DF


DF98 C3 DD 13 00 00 00 00 00
DFA0 00 00 00 00 00 00 00 00
DFA8 00 00 00 00 00 00 00 00
DFB0 00 00 00 00 00 32 FE 0F
DFB8 21 BA 0B 06 10 77 23 10
DFC0 FC C9 32 FF 0F 3A 14 0F
DFC8 32 77 0F C9 EF 43 6F 6D
DFD0 6F 61 6E 64 3F 0D 00 DF
DFD8 63 01 2B 0C 1A FE 20 C8
DFE0 FE 41 38 0D FE 5B 30 09
DFE8 02 32 0A 0C 13 DF 79 30
DFF0 03 DF 6B C9 DF 60 DF 5C
DFF8 C9 00 00 00 00 00 00 00
```

RRRepeat KBDs

Repeat keyboard for Nas-sys   by Richard Beal

The routine itself is from 0C80 to 0CF5H. After the program is executed it
returns to Nas-sys. You can then use Nas-sys or execute ZEAP 2.0 or Basic and
enjoy the benefits of a repeating keyboard. Just hold down any key, and after
a delay it will repeat quite quickly, which is ideal for cursor movements. Try
holding down several keys at once!! Note that the @ key does not repeat
because of its dual function on Nascom 1. We also attach a tabulation of the
code to make it easier to type in. Execute the program at 0C90H.

Memory usage and changing the speed

We hereby declare the following memory locations will for all time have the
specified meanings:-

Location	Description
0C2C - 0C2DH	KCNT counter for repeat keyboard
0C2E - 0C2FH	KLONG delay before repeating starts
0C30 - 0C31H	KSHORT speed of repeating keys

These were unused locations in the Nas-sys work area. To change the initial
delay before repeating starts, and the speed of repeat, change the values at
KLONG and KSHORT.

Suggested values:	4MHz	2MHz
KLONG	0280H	0140H
KSHORT	0050H	0028H

The version overleaf is for 4MHz (Nascom 2).

Repeating keyboard from Basic

You can also set up a repeating keyboard within Basic, by typing in the Basic
subroutine at lines 24000 to 24090. To execute it, type in: RUN24000

You will find that you then have a repeating keyboard. Note that line 24030
sets the speeds, we suggest the following values:

	4MHz	2MHz
Initial delay	DOKE 3118,640	DOKE 3118,320
Repeat rate	DOKE 3120,80	DOKE 3120,40

THIS NEWSLETTER CONTAINS ONLY THE FINEST MATERIALS AND
WORKPERSONSHIP. THE EDITOR HAS AT HIS DISPOSAL ADVANCED
EQUIPMENT SUCH AS TINS OF SPRAY-GLUE AND TRENDY PENKNIVES
THE PUBLISHERS THEREFORE CONSIDER THAT ANY COMMENTS MADE UPON
THE STRAIGHTNESS OF THE LAYOUT AND ACCURACY OF THE TYPING
ARE SO MADE WITH MALICE AFORETHOUGHT, PROBABLY BY RIVAL
PUBLISHERS; SUCH COMMENTS WILL THEREFORE BE FED WITHOUT
FURTHER LET OR HINDRANCE TO THE OFFICE GOAT.

```
0C90 21 F3 0C DF 72 21 B0 0C
0C98 22 7B 0C 21 80 02 22 2E
0CA0 0C 21 50 00 22 30 0C DF
0CA8 5B 00 00 00 00 00 00 00
0CB0 DF 61 30 07 2A 2E 0C 22
0CB8 2C 0C C9 2A 2C 0C 2B 22
0CC0 2C 0C 7C B5 C0 21 02 0C
0CC8 01 00 08 16 FF 7D FE 06
0CD0 20 02 16 BF FE 09 20 02
0CD8 16 C7 7E A2 28 06 0E 01
0CE0 7A 2F A6 77 23 10 E4 79
0CE8 B7 C8 2A 30 0C 22 2C 0C
0CF0 DF 61 C9 76 70 00 00 00
0CF8 00 00 00 00 00 00 00 00
```

```
24000 REM ** SET UP REPEAT KEYBOARD
24010 FORI=3248TO3316STEP2:READJ:DOKEI,J:NEXT
24020 DOKE3195,3248:DOKE3189,3315
24030 DOKE3118,640:DOKE3120,80:END
24040 DATA25055,1840,11818,8716,3116,10953
24050 DATA3116,8747,3116,-19076,8640,3074,1
24060 DATA5640,32255,1790,544,-16618,2558,544
24070 DATA-14570,-23938,1576,270,12154,30630
24080 DATA4131,31204,-14153,12330,8716,3116
24090 DATA25055,30409,112
```

NOTES ON PIO OPERATION
_____

The Nascoms 1 and 2 have on board two totally uncommitted 8 bit parallel I/O
ports complete with handshake lines, in the shape of an MK3881 Z80 - PIO. The
PIO is, in itself, a fairly complicated processor, which needs programming
before it will operate in any of its 4 modes:

Output   Mode 0
Input   Mode 1(Automatically set on PIO reset)
Bidirectional   Mode 2
Control   Mode 3

It is not the purpose of these notes to describe in detail these operational
modes, but to help clear up a few common problems encountered in controlling
the PIO.

One very important fact to note is that on the Nascom 1 the PIO is not reset by
the reset button on the keyboard - this resets the CPU only, NOT the PIO.The
problem has been rectified on the Nascom 2 but the Nascom 1 PIO may be reset in
two ways. The simplest is to switch the power off and on again; a bit drastic
but the PIO does have automatic power on reset. The second method (shown in
the diagram below) is to apply an Ml without either RD or IORQ and this is how
it is done on the Nascom 2. It should, however, be pointed out that, since the
CPU can be reset, it is always possible to regain control of the PIO in
software, by simply reprogramming it but more of this later.

Now to 'interrupts'. Don't forget that the PIO is designed to operate in the
Z80 interrupt mode 2, so before doing anything put the CPU into this mode by
executing 'IM 2' (hex code ED 5E). Remember that a CPU reset puts the Z80 back
to interrupt mode 0, clears the I register, and disables CPU interrupts (having
no effect on the PIO on a Nascom 1).

In Interrupt Mode 2, the CPU finds the address of the interrupt routine, by
loading the Program Counter (PC) with the contents of the memory address. This
is formed by the I register (high byte), and the interrupt vector sent from the
interrupting port (low byte).

For example, let us suppose that an interrupt routine for port A starts at
OE12H, and that the interrupt address table will be stored at 0F80H. In order
that the routine should be found correctly, the I register should contain 0FH,
the value 80H should be sent to the control register of port A, and finally,
memory locations 0F80H and 0F81H should contain 12H and 0EH respectively (low
byte first). At an interrupt, CPU interrupts are automatically disabled and
must be re-enabled, if required, by the programmer.

Always end an interrupt service routine with the RETI instruction, as this is
the only way to indicate to the PIO port that the service routine is finished.
This feature can cause some dismay to the unwary. Take the following example:
everything is set up correctly, and the port interrupts correctly. However,
unfortunately the interrupt routine crashes. No problem to our intrepid
experimenter, he presses reset, debugs the interrupt routine and tries again,
remembering to reset IM 2, I register and interrupt enable. Dismay! Nothing
happens. No interrupt.

The problem is that the PIO still thinks that it is being serviced for its
initial interrupt, and is internally inhibited from causing another. A useful
routine to get out of this sort of problem is as follows:-
21 00 00   LD HL,0000H
E5   PUSH HL
ED 5E   RETI

This will tell the port that its service routine is finished and then restart the monitor by executing from 0000H. It can be used at any time, if there is any doubt as to the status of a PIO.

Once the mode and interrupt control have been set, the port interrupt may be enabled or disabled by sending 83H or 03H to the control register. This feature could form the basis of a generalised interrupt control program for a given system. However, it should be noted that the correct way to disable a port interrupt is to first of all disable CPU interrupts before the port interrupt. This is because an interrupt by that port, during the execution of the instruction to disable its interrupt, would cause a system crash.

Finally, when a port has been disabled, an interrupt may be pending, so that when the port is again enabled it will at once interrupt the CPU. This pending interrupt may be cleared, if required, by sending an interrupt control word with bit 4 set. This is effective in all modes.

Please let us know of any interesting applications for your PIO, or better still write an article for YOUR newsletter.
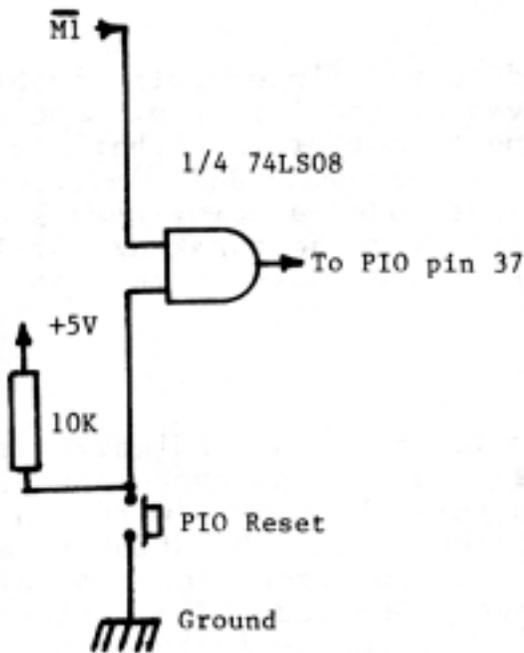
Solution to Puzzle

Nobody sent in a correct explanation of the problem, but I give half marks to those who made the program work by putting in code B7, which is OR A, before the INC A. The reason for the problem is that although INC A and ADD A, 1 might seem the same, INC A does not set the Carry flag, or change it at all.

Once the carry flag gets set, the DAA instruction gets completely confused! To understand all this, use the S command to step through, keeping an eye on the Carry flag. Also, have a look at the Z80 programming manual, under INC, ADD, and DAA. If you don't understand the table describing DAA, you are not alone!

By the way, perhaps the Z80 should set Carry when INCing - but if it did it wouldn't be 8080 compatible, so you really have to blame Intel for the 8080 design. Anyway, these little quirks make programming interesting.

PIOs

A few newsletters back we did an article about the PIO (MK3881), and tried to get readers out of the 'Catch 22' situation, where the PIO technical manual could only be understood if the reader knew something about PIO's. The article dealt with the general aspects of the PIO, and how to use it. Well Richard has been busy, and has sent us this:

Understanding the PIO
The PIO manual provides a good description of how this powerful device works, but there are some aspects of it which are not easy to understand. Perhaps the most confusing of all is the use of the 'handshake' signals, known as STROBE and READY lines. This article will briefly explain how input and output modes work, and then will give an example showing how two PIO' s can communicate, when one is the output and the other is the input.

Input mode

The PIO port must be set to mode 1 and interrupts enabled.
As usual the I register, interrupt vector and interrupt mode must be set up correctly, and the PIO must not think that it is already processing an interrupt, so a dummy RETI instruction should be executed. Also, when in input mode, a dummy read should be issued once at the start to initialize handshaking. At last you are ready to enable the CPU interrupts with an EI instruction.

The sequence of events is as follows:

The input STROBE line goes from low to high, indicating that data is ready in the PIO input register. This change causes an interrupt, and the interrupt routine gets the data with an IN instruction. Issuing this instruction gets the data and also causes the PIO READY output line to go from low to high, indicating that the data has been received. The interrupt routine ends by enabling the CPU interrupts again and then immediately returns with a RETI, which re-enables the PIO interrupts.

Output mode

The PIO port must be prepared as for input mode, except that it is set to mode 0.

The sequence of events is as follows:

The input STROBE line goes from low to high, indicating that the previous output has been received. This change causes an interrupt, and the interrupt routine outputs the data with an OUT instruction. Issuing this instruction not only outputs the data to the PIO, but also causes the PIO READY line to go high, indicating that data is ready in the PIO. The interrupt routine ends by enabling the CPU again and then immediately returns with a RETI, which re-enables the PIO interrupts.

You may have wondered when the READY line goes low. In both modes the READY line goes low when an interrupt is generated. This may be used by the external device to acknowledge that an interrupt has been received. It could be used to inhibit further interrupts from the external device, particularly where a number of PIO's are 'daisy chained' together, and higher priority interrupts are already being processed, hence there will be a delay before the external device is processed. In other words, it says to the external device "OK, seen you, you are in the queue, don't bother me again until I go high." It has no internal effect on the PIO, save to get it ready to go high again when processing of the interrupt is complete.

Example of handshaking sequence

For two PIO's to handshake, the STROBE (input) line of each is connected to the READY (output) line of the other. This diagram shows a typical sequence of events, starting with a dummy read from the input port to start the process.

Output port

Input port

                                        IN issued once after PIO initialized.
                                        READY goes high. CPU interrupts enabled
                                        with EI.

The incoming STROBE goes high,
so there is an interrupt.
READY goes low.
Interrupt routine issues an
OUT, and the READY goes high.
Interrupt routine ends with
EI, RETI, so it is ready for
the next interrupt.

                                        Incoming STROBE goes high, so
                                        there is an interrupt.
                                        READY goes low.
                                        Interrupt routine issues an
                                        IN, and the READY goes high.
                                        Interrupt routine ends with
                                        EI, RETI, so it is ready for
                                        the next interrupt.

The incoming STROBE goes high,
so there is an interrupt,
READY goes low.
Interrupt routine issues an
OUT and the READY goes high
                                        And so on
And so on

You could try this with just one PIO, by making the two ports of your PIO talk to each other.

A note for Nascom 2 owners. The memory card joins the 'daisy chain' (IEI and IEO) lines together, and under these conditions the interrupts won't work. Cut one of the tracks on the vero bus connector. Do not cut the link on the memory board, as this would be required if that card were used on a bigger bus with 'daisy chain' priority interrupts.

THE STORY OF THE NASCOM ONE-TWO

Or Why Two Computers are Better than One   by Richard Beal

Is the Nascom One-Two a marvelous product of the distant future? No, it is already here, and consists of a Nascom 1 connected to a Nascom 2 to form a new and versatile computer system with the following features:

(a) Ability to read data from Nascom 1 tapes directly into a Nascom 2. This works even with an unexpanded system.

(b) Intelligent giant print buffer for serial printers such as the Nascom Imp or Teletype. This enables output from ZEAP, BASIC or NASPEN to be routed to the Nascom 1 memory virtually without delay, and with automatic compression of blanks, to save valuable RAM space. Printing is completely independent of the Nascom 2, and can proceed at the same time as more data is being sent across at high speed. If the print buffer becomes full, the Nascom 2 will automatically wait until printing starts, or if required the print buffer can be cleared if the output is not, after all, needed. Printing can be paused at any time, and the Nascom 1 display shows the number of characters waiting to be printed.

(c) Almost instantaneous transfer of the whole of the contents of memory from one machine to the other, allowing very fast recovery of programs and data when testing machine code programs. For example ZEAP could be used on the Nascom 2, and the source code and generated machine code both held in memory. Then before testing the program, which could well crash and change the contents of any part of the memory, the whole memory is copied in a few seconds to the Nascom 1. When the Nascom 2 crashes, it is simply reset, and the data brought back. Within seconds the ZEAP source can be edited and re-assembled and the process repeated.

These facilities are very useful, and mean that if you have bought a Nascom 2, your Nascom 1 can continue to be used instead of putting it in the attic, or trying to sell it, which is rather sad after all the effort needed to build it. It also gives you another reason for buying a Nascom 2 if you already have a Nascom 1!

At this point you may be wondering what sort of equipment you need for all this. If you have a Nascom 1 and Nascom 2, all you need is ...... NOTHING!!!

Obviously if the Nascom 1 is not expanded, only feature (a) is possible. The only hardware work required is to connect various lines from the Nascom 2 26 way ribbon cable to the Nascom 1 PIO sockets. Only one port on each machine is used, with all data transfer being interrupt driven using the handshake lines. To understand this, read the PIO technical manual until you are sure you know it all. (I have read it about 30 times and am still not completely confident). Alternatively, read on.

Connect the 8 data lines of port B on the Nascom 2 to the corresponding 8 data lines of port B on the Nascom 1. Connect the ground line of the N2 to pin 9 (ground) of the N1. Connect the B READY line from the N2 to the B STRB on the N1 and the B READY line on the N1 to the B STRB line on the N2. Do not connect the power lines together! You may choose to wire a switch so you can see the N1 or N2 display on the monitor screen, but you don't really need the N1 display. If you wire such a switch, wire the earth lines together.

One further point on hardware is that for some strange reason the memory board links together the IEI and IEO. This doesn't matter on the N1, but the N2 brings these lines out correctly to NASBUS, and the PIO won't interrupt unless IEI is high. Simply look at the edge of the memory board, next to line 19, and you will see that it is connected to the next line. Break this join by scraping away with a screwdriver. (Ed. Don't do that, the memory board is correct, it is the mini Vero bus on the N2 which is wrong. Break track 19 on the Vero board. See note under the article about PIOs.)

As you will have realised, the key to the Nascom One-Two is in the software. We will just cover feature (a) the cassette input via Nascom 1. This is extremely useful for converting your tapes from N1 to N2 format.

Since giving you the program listings would make it much too easy, here is a description of the little programs which you need to put into each computer. (The listings will be put in the library in due course).

Cassette input transfer program for Nascom 1

This program initialises the PIO and then goes into a loop which does nothing at all. However, each time there is an interrupt the interrupt routine waits until there is a character from the cassette input, and then outputs it to the PIO and returns. The full list of actions is:

Disable the CPU interrupts with a Dl instruction.
Disable PIO (03H to control port).
Ensure PIO is happy by pushing address of next instruction on
to the stack, and then using a RETI.
Load I register.
Set interrupt mode 2, with IM2.
Output the interrupt vector (low order half address of the
interrupt address table).
Output 0FH to PIO to set it to output mode.
Set the interrupt control word by outputting 87H to the PIO.
Enable the CPU interrupts with an EI instruction.
Go into a tight loop (actual code 18 FEH).

That is the end of the initialisation program, which is executed.
Then you need an interrupt address table. The high order part of the address of this table has been stored in the I register, and the order part has been sent to the PIO as the interrupt vector (this must be an 'even' number, ie. last bit = 0). The interrupt address table actually contains just one address, which is the address of the interrupt routine itself. This address is placed at the start of the table, low order byte first, as usual.

The interrupt routine itself calls its own copy of the SRLIN routine and loops
until the routine returns with the Carry set. It then outputs the character to
the PIO, enables the CPU interrupts and immediately returns. This last bit of
code reads:

```
OUT (5), A
EI
RETI
```

Cassette input program for Nascom 2

The other half of the software lives in the Nascom 2. The program initialises
the P10 just like the other program, except that the PIO mode is 4FH, for input
mode. After setting up the PIO, the CPU interrupts are not enabled. Instead a
dummy READ to the PIO is made to ensure that handshaking starts. Then a
version of the READ routine, copied out of NAS-SYS, is executed. If the
routine ends, control returns to NAS-SYS. Instead of the need for changing
tables and RST RIN to get an input, normal calls are made to a new routine
called CIN.
This routine reads as follows:

```
CIN   OR A
      EI
SLP   JR NC SLP
      RET
```

This loops until an interrupt occurs, and the interrupt routine sets the Carry
flag, and puts the input character into A. The interrupt routine reads simply:

```
PROC  IN A, (5)
      SCF
      RETI
```

Note that it does not re-enable CPU interrupts.

As you can see, the use of the PIO requires a bit of thought, but if you read
the notes above, and have a look at the PIO manual, you should be able to write
these routines for yourself, exactly as you wish, which is much more
interesting than typing in some HEX listings from the library.

Please write in and let others know if you have any other uses for two (or
more) machine systems. How about a controlling machine loading other slave
machines with data, programs, perhaps even interpreters as well, then doing
other work while the slave machine performs the subordinate tasks. This would
be one way to tackle a multi-user operating system with resources such as
printer or disc attached to only some machines.

Richard Beal

PS You could use two Nascom 2 computers, if you don't already have a Nascom 1.
PPS Don't blame me if you get your wires mixed up and blow up both computers.

The End of 'PLAGUE'

THE FINAL DEFINITIVE MEMORY PLAGUE NOTE.

In the last issue we promised that we would collect together all thoughts about 'Memory Plague', and publish them. We are indebted to many members of the IMNC for writing to us detailing various symptoms and possible causes. For more recent members of the INMC let us first define 'Memory Plague': this was a euphemism coined to cover the small percentage of Nascom Series 1 memory cards (that's the one with the four EPROM sockets) that proved to be unreliable through causes not otherwise due to faulty or slow chips. 'Plague' soon became apparent after the release of the Series 1 card, and the symptoms are unreliability when working machine code programs in the expansion memory. 'Plague' is not usually revealed by Tiny Basic or the memory these represent data stored in memory and not op-code, affects the op-code fetch (Ml) cycle as the timing is more critical, and results in the misreading of the op-code byte, causing programs to crash. The probable cause is noise generated on the Nascom 1 busses, although this has never been conclusively proved. A number of 'cures' were published, which consisted of basically three things:

1) Gridding the back of the memory card to reduce power supply noise.
2) Pulling up the outputs of the RAMs to increase operational speed.
3) Adding a time constant in the form of a capacitor to the RAM pullups to damp any noise on the output buffer.

On a Nascom 1 these work. On a Nascom 2, step 5 represents overkill, and can cause poor operation at 4MHz, the time constant being calculated for 2MHz.

Over that intervening period other possible causes have come to light. One is the 33R damping resistors in series with the address and CAS lines are too low to be properly effective. Another possible is the DBDR signal coming off too early. Yet another is the MREQ signal (applied to IC31) arriving too late, related to the system clock which latches it.

With the advent of Nascom 2, which is buffered on board, bus noise has been much reduced, and plague is almost nonexistent. However, Nascom 2 has revealed that the memory card will not run reliably at 4MHz without 'WAIT states'.

We therefore now recommend the following:
Nascom 1
1) For each row of logic chips, grid the back of the pcb, connecting the +5 volt rai1 from the termination on the logic half of the pcb to the +5 volt termination of the l00n decoupling capacitor immediately opposite. For each row of logic chips connect the 0 volt rail likewise; where a row is adjacent to pin 12 of each of the EPROM sockets incorporate this into the 0 volt grid.

2) Change R7 - 14 and R17 to 68R, and RAS links LK1 and LK2 should also be 68R resistors.

3) Where the Basic ROM is fitted, add a 150pF capacitor from pin 1 to 0 volts of IC9 on the buffer board, this will slightly increase MREQ, and thereby increase DBDR.

Nascom 2

1) Try the board and see if it runs Basic reliably. If so go to step 3. Don't
bother to investigate further unless the board reliability is suspect.
2) Grid the back of the pcb and change the resistors as for Nascom 1.
3) If you want the board to run at 4MHz without 'WAIT states':
a) Cut the clock line from the bus connector, 5, to pin 3 of IC31.
b) Connect a wire link from the clock side of the cut track to pin 9 of IC35.
Connect a 1K pullup between pins 8 and 14 of IC35.
c) Connect a wire link from pin 8 of IC35 to pin 9 of IC34. Connect a wire
link from pin 8 of IC34 to pin 3 of IC31.

A number of boards have been fitted with the above, and all and have behaved
perfectly. If you have a board which has been fitted with the earlier Nascom 1
mods, and is working correctly, then leave well alone. If for some reason you
wish to incorporate the above, then remove any earlier mods first.

LITTLE KNOWN FACTS THAT NO-ONE SEEMS TO CARE ABOUT

Did you know that the 8K Basic can be persuaded to accept 'INPUT' strings with
commas in them? Try this:

First set up the machine code input.
```
10 DATA 25055,1080,-53,536,-20665,3370,-5664,0
20 DATA 27085,14336,-13564,6399,18178,10927,-8179,233
30 RESTORE 20:Xl=31:X2=29
40 IF PEEK(1)=0 THEN RESTORE 10:Xl=13:X2=8
50 DOKE 4100,3200:FOR I9=3200 TO 3214 STEP 2
60 READ 18:DOKE 19,18:NEXT
```

Now go and get the string (including commas)
```
100 GOSUB 200
110 PRINT A$:GOT0100
```

This subroutine replaces the normal 'INPUT' command
```
200 A$=""
210 A=USR(0):IF A<0 THEN 210
220 IF A=X2 AND A$="" THEN 210
230 PRINT CHR$(A);
240 IF A=X1 THEN RETURN
250 IF A=X2 THEN 270
260 A$=A$+CHR$(A):GOTO 210
270 A$=LEFT$(A$,LEN(A$)-1):GOTO 210
```

Two things here, firstly we have setup a machine code 'INKEY$' command, then
using that, we continually add to A$ until a 'new line' is found in line 240.
All keyboard characters will be accepted except 'new line', and 'back space'
which are treated separately. Note that line 30 assumes that NASBUG is in use,
whilst line 40 resets the pointers if NAS-SYS is present.

The Nascom 1 on board modulator is capable of producing a good picture on a domestic TV, although results in the field vary. If you have a poor picture and need a stronger signal try reducing the value of R8. The winding of the coil can also be critical and seems to account for much of the variation between systems. Rather than spend too much time on this, we would recommend that you obtain a separate add-on screened modulator from your local distributor.

If the TV sync slips when displaying your Nascom, or if you are lacking video drive to your monitor you could hang a 470R preset pot across the cathodes of Dl and D2 (a diode 'points' to the cathode by the way) and, having removed R8 and R9, feed the wiper of the pot to the modulator or monitor. Tweak for best sync versus best contrast.

Due to an artwork error on the earlier Nascom 1 boards, pin 35 of the UART (parity inhibit) has been left floating. It should be tied to +5V, and this can be done by adding a small link on the solder side of the board between pins 34 and 35. In practice we have found that the absence of a connection to this pin generally makes no difference, but it can occasionally cause intermittent tape loading or incompatibility between systems.

In certain circumstances it may be possible to increase the data transfer rate of the cassette load and dump. This can be done by increasing the frequency to pins 17 and 40 of the UART (IC29). The standard frequency at this point is 3.9 kHz, which is taken from IC2 pin 11. The transfer rate may be doubled by taking the UART clock from IC2 pin 12 (7.8 kHz) or quadrupled by taking the UART clock from IC2 pin 13 (15.625 kHz). With each increase in speed, however, the record/replay levels, the quality of the tape and recorder used, and the variations between records become much more critical. Using a #40 portable cassette recorder and good quality tapes we have been achieving a 100% load rate at 4 times the standard speed, but it should be noted that these modifications are not 'guaranteed' to work.

We have found that the tolerance of some manufacturers' components can lead to problems with the timing of IC18 (74LS123). The problem is centred around the timing of the character decoding in IC16 and manifests itself when a character with bit 6 set (eg. any capital letter) is followed by a character with bit 6 clear (eg. any number). If these are displayed on the screen, the first character will disappear! The problem can usually be alleviated by (i) interchanging ICs 7 & 18 (ii) bending pin 5 of IC18 straight so that it is not connected when IC18 is in the socket and connecting pin 5 to pin 12 of IC18 on the underside of the board (iii) changing the device for that of another manufacturer.

Floating inputs to port 0 (keyboard user inputs). Although the software ignores spurious characters which may appear on port 0, the keyboard still carries out a search to determine whether the input is still valid. If, as is likely, the two user inputs on SKT1 are left unconnected, this could have a detrimental effect on the running of any program with interactive keyboard routines. This flaw may be easily rectified by connecting the two user inputs to +5V, forcing them permanently 'high'. Under these circumstances, no spurious inputs occur.

Adjustment of the 1760 Hz (10 char/sec.) clock without test equipment. Firstly it should be noted that this clock need not be adjusted until such time as a printer or other serial peripheral is added. Adjustment is effected by VR1. Clockwise rotation reduces the clock speed. With a printer attached via the RS232 or 20 mA outputs, a short test program may be written that will output continual text in the form '1234567890123...' etc. Adjustment is made by observing the printed output;

If the clock is too fast, random garbage will appear thus:
123c5u789,P234+z7   etc.
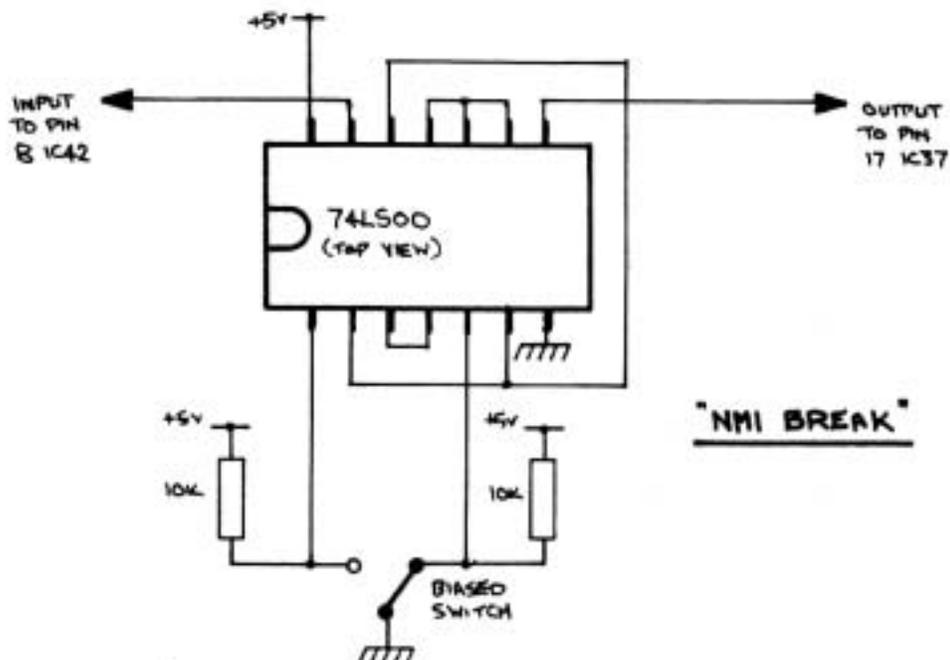If the clock is too slow, characters will be missed, thus:
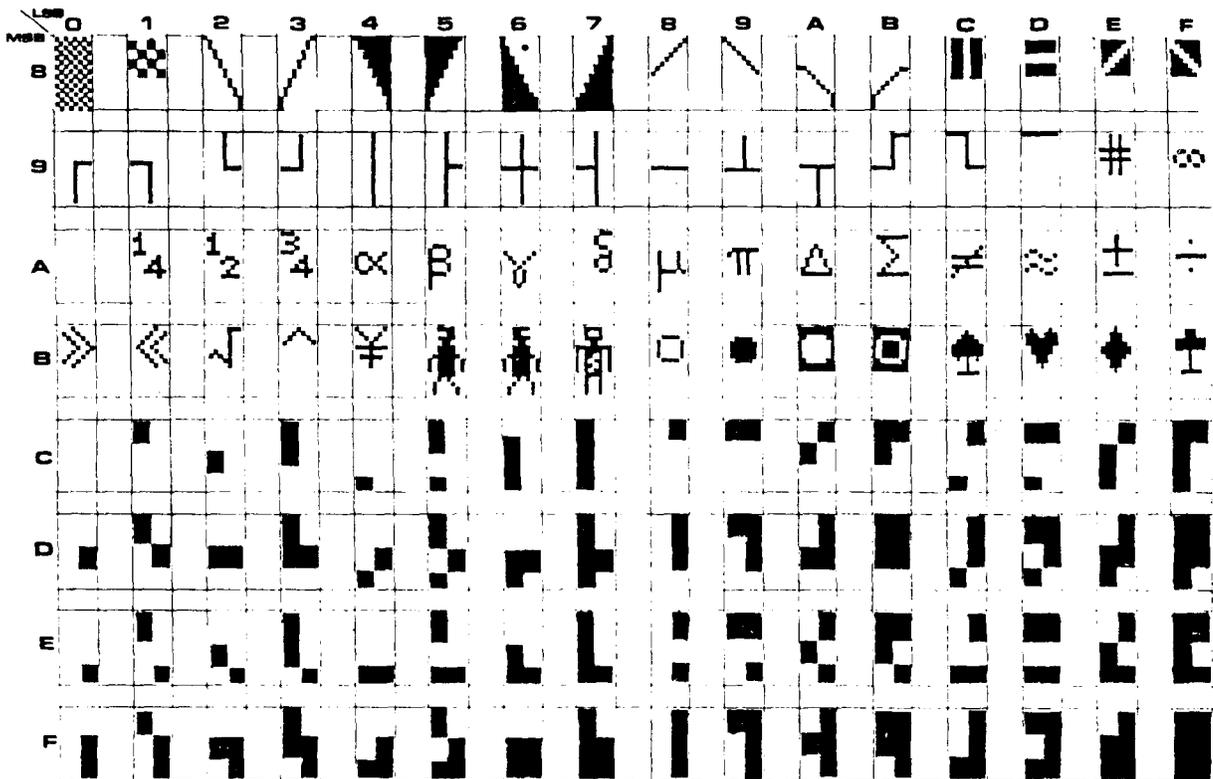1235679013457891   etc.
Note that VR1 is a multiturn (20 turns) preset, and that the end stops are
detected by an increase in rotational torque at the ends of the track. No harm
can be done by over 'turning' the preset. Correct adjustment is the mid point
between garbage and missing characters, this is a latitude of 4 to 5 turns of
the pot.

The serial clock speed may be changed to 4800 Hz (30 char/sec.) by changing C12
to an 8n2 10% polyester capacitor. Setting of the 4800 Hz clock is as above.

The snow plough is used in conjunction with IC11 to increase the VDUSEL
blanking time to eliminate 'snow' on the screen during memory access to the
video RAM. See the diagram below. The simplest method of construction is to
take one of the spare 16 pin DIL plugs (supplied) and cut off two pins, making
it a 14 pin plug. Then cut a piece of 0.1" pitch Veroboard about 1" wide by
about 1.5" long, with the tracks in the longer direction. Mount a 14 pin
socket at one end (breaking tracks as appropriate) and build the LS123 circuit
at the other, connecting the output of the 123 to pin 5 of the 14 pin socket.
Then solder the 14 pin plug pin for pin to the underside of the 14 pin socket
(except pin 5). Link pin 5 of the plug to the input of the 123 circuit, and
low!! a little plug in module which carries 1C11 and the 123, with a plug that
fits directly into the 1C11 socket on the board. Neat, tidy and effective.
Don't forget to connect power to the 123, in parallel with pins 7 and 14 of
1C11. Plug in the module, and a TV display should appear usual. Tab from 0 to
FFFF and adjust the preset pot such that the 'snow' just disappears.

The NMI 'break' (which will not work with Nasbug T2) should be made on a small
piece of Veroboard and mounted somewhere appropriate. To connect it, a wire
should be run from pin 8 of IC42 (under the board) to the input of the circuit.
The CPU should be lifted from the board and pin 17 carefully bent out
horizontal, the CPU may then be replaced. The output of the circuit is
connected to pin 17 of the CPU, using a 'Soldercon' pin. DO NOT SOLDER TO THE
CPU. If you fit the NMI generator and make the leads too long, then spurious
NMIs can be caused. Every time I turned the tape recorder off, so the damn
thing would do multiple NMIs in the keyboard routine. Now you can't restart
from an NMI in the keyboard routine!!! Sod's law says that by having a
multiple NMI you'll have screwed up something.

+5v

25K

5K6

680P

74LS123
(TOP VIEW)

TO VDUSEL

+5v

O/P TO
PIN 5 IC11

"SNOW PLOUGH"

PLANNING YOUR DISPLAY   By V.P.Lipton

1. The chart below has the decimal equivalents of the hex addresses of the
screens for Nascom 1 & 2. When using BASIC, one has to POKE data into the
decimalised addresses on the screen, so the little chart will help all you
BASIC people to get your little men into the right positions easily.

(Ed.'s Note: Any position on the screen can be calculated by the BASIC using
the equation $P=1993 + X + 64 \times Y$, where X is the column (1-48), Y the line (1-
16), and P the decimal address of the desired location. (Ed.'s Note: The note
above was from the original Editor who obviously forgot to mention the fact
that using this calculation line 16 is the TOP line on the screen with lines 1-
15 below and not as shown in the table. Nested 'Ed.'s notes' eh? ... Just wait
till the next reprint comes out!))

2. Also below is a listing of the Nascom Graphics ROM. If a lot of key-bashers
have been scratching their heads trying to get their little men to appear, then
this amended chart is sure to relieve their dandruff, as the chart printed in
the original Nascom 2 handbook was far from useful!

1	3018	3019	etc...	3042		3065
2	2058	2059	etc...	2082		2105
3	2122	2123	etc...	2146		2169
4	2186	2187	etc...	2210		2233
5	2250	2251	etc...	2274		2279
6	2314	231S	etc...	2338		2361
7	2378	2379	etc...	2402		2425
8	2442	2443	etc...	2466		2489
9	2506	2507	etc...	2530		2553
10	2570	2571	etc...	2594		2617
11	2634	2635	etc...	2658		2681
12	2698	2699	etc...	2722		2745
13	2762	2763	etc...	2786		2809
14	2826	2827	etc...	2850		2873
15	2890	2891	etc...	2914		2937
16	2954	29S5	etc...	2978		3001

```
 L.H. Edge                MID SCREEN           R.H. Edge
--------------------- 48 LOCATIONS -------------------
```

LITTLE KNOWN FACTS THAT NO-ONE SEEMS TO CARE ABOUT

Did you know that NASBUG T4 and NAS-SYS 1 are specifically designed to ignore
nulls output to the CRT routine? Therefore any program which is designed to
output nulls via SRLOUT, using the 'X' command will not be able to do so, as
CRT is called as part of SRLOUT, before the character is sent to the output
port, and (of course) CRT throws the nulls away.

So, what to do? Well the simplest way is to write a routine which will:
1) Push AF
2) Call SRLX
3) Pop AF
4) Jump to CRT in T4
4a) or do a return if you don't want to print on the screen

Don't set any 'X' command, but manually change $CRT to the start address of the
routine (don't forget this must be a double byte change).

Using NAS-SYS, put DF 6F at 0C79 and activate the 'J' command.

Of course this need not only apply to printers, for instance, if you wanted to
change a tape recorded for Nascom ROM Basic to run on Nascom Tape Basic, you
would have to make a 'LIST' to tape and output about 20 nulls between each line
(to allow the text buffer time to clear before the next line appears), and this
is the only way to do it.
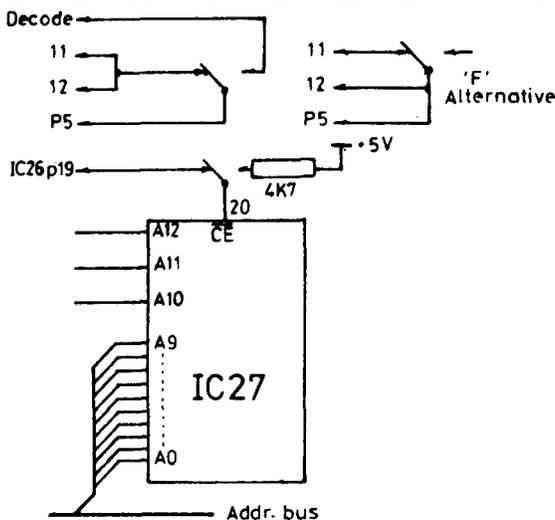
BASIC CHIPS ON NASCOM 1's

So there you are with your brand new (and expensive) MK36271 BASIC ROM, and
wondering where to fit it. There don't seem to be any holes for it. When the
memory board was designed it wasn't envisaged that anyone would want to fit an
8K ROM to it, so no holes were provided. Never mind, a little bit of surgery,
and all will soon be working.

Take a careful look at fig. 1, and carefully cut the tracks as shown. Then
connect wire jumpers as follows:
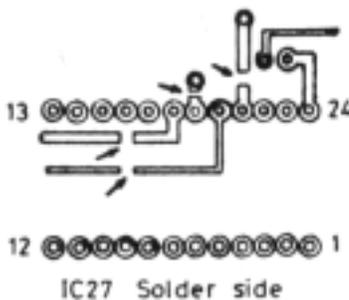
1) IC26 pin 19 to IC27 pin 20.
2) IC24 pin 2 to IC27 pin 19.
3) IC24 pin 3 to IC27 pin 18.
4) IC25 pin 7 to IC27 pin 21.
5) Decode pads 11 and 12 together to pad P5.
6) Check it!
7) Plug the board back in the Nascom and try
it. Lo! You should have BASIC.

IC27 Solder side

Now this mod. means that you have lost the use of the other 3 sockets, so for
those more adventurous souls, you can get them back again. The tracks to pins
19 and 21 were the +12 and -5 volt supplies to the EPROM sockets so these must
be restored to make ICs 28, 29 and 39
operable. Two jumpers from the points where
the tracks were cut to pins 19 and 21 on
IC28 will restore the juice. But this will
also put juice back on IC27, as these pins
are linked on the top side of the pcb. Some
deft knife work is required to cut these two
tracks between ICs 27 and 28 sockets (and
get it right 'cos you'll never repair a
mistake!!!). Now the way we have mod'ed the
chip select on the Basic ROM means that it
will come on regardless, so a switch is
called for to disable the ROM when not
required. So make it a double pole switch,
and use it for selecting the decodes as
well. Note that alternative arrangements
are shown, for any other decode apart from
'E' and 'F', and decode for 'F'. Why? Well some idiot will want to run Big
BASIC and Super Tiny BASIC on one board. Apart from that our version of Naspen
is an early one, and runs from F800 to FFFF. Naspen usually resides between
B800 and BFFF.

Can we make one thing quite clear, these mods have been tried and work, but the
INMC can not take responsibility for any failures, or chewed up pcbs as a
consequence of trying them. If you don't think you are capable of attempting
these mods, DON'T TRY. Seek the help of someone competent, then you can blame
him if he chews up half the tracks on the pcb.

STOP PRESS: 100nF capacitor required across pins 24 and 12 of the BASIC ROM.

AUTHOR FINDS OWN BOOBS IN INMC ARTICLE   by Ms D. R. Hunt

No, not a refugee from page 5 (we got told off by readers last time we let a naughty word onto these hallowed pages), but to draw your attention to the fact that we all make mistakes. Last issue, we published a bit about adapting the series 1 memory card to carry the 8K Basic ROM, and went on to describe how to put the remaining three EPROM sockets back into use. Well the Basic ROM bit works, but the EPROM bit doesn't!!!

Having fitted the Basic ROM to the card, and written it up for the INMC, in a most un-natural fit of enthusiasm, I modified the board for the EPROMS, 'zapped' in a couple of EPROMS, gave it a quick go and wrote it up, but because of the closeness of the copy date, I didn't get round to actually testing it thoroughly. Later, (over Christmas), horror of horrors, fit some EPROMS and the Basic does funny things. Close examination of the circuit reveals that enable of 1C24 (pin 1) comes on when the Basic ROM comes on, so with EPROMS fitted they are enabled along with the Basic ROM resulting in rubbish on the bus. The strange thing is though, that in the conflict between the EPROMs and the ROM, the ROM usually won, so Basic didn't just crash as you'd expect, but became unreliable.

At first sight this problem looks very difficult, requiring a three pole switch (very hard to find) to switch the 1C24 enable along with the Basic enable, and the decodes. However, look at the circuit again, it won't take long to realise that the enable line to IC26 (which we already use for the Basic ROM) could just as well be used to enable 1C24 instead of the signal currently used.

So, now to the correct version, refer back to the previous article, and notice that a 4K7 pull up resistor (on the right hand connector) is used to disable the Basic ROM when not in use, leave this where it is, but reverse the connections to the other side of the switch. Viz: IC26 pin 19 is connected to the pole of the switch, and IC27 pin 20 is connected to the left hand connector. Further, another 4K7 pull up resistor has to be fitted to IC27 pin 20 to pull the Basic ROM 'off' when the switch is over the other way. Carefully cut the track to 1C24 pin 1 (adjacent to the pin), and connect pin 1 to the right hand connector of the switch, leaving the existing 4K7 pull up resistor in place to turn 1C24 'off when the switch is over to the Basic.

All this seems to work satifactorily, but we have heard of a few cases of difficulty (even without EPROMs on board), a 'Stop Press' was added to the previous article about fitting a 100n capacitor across the back of the ROM to decouple the supply lines, I have found that 2u2 bead tant. is better. You wouldn't believe the 'crud' the Basic ROM generates on the supply line as it's enabled, if you've got a 50MHz scope to hand, have a look at it across pins 12 and 24. A further aid to quieting down and speeding up the system is to decouple each EPROM socket with 10n, and to pull the data outputs of the ROM and EPROM block high with 10K resistors.

We've mentioned memory plague before, and a recent note in the INMC newsletter gives a more consistent cure. However, whilst looking for the troubles with the Basic ROM with EPROMs in situ, a remarkably naughty bit of tracking on the earth line of the EPROM block was discovered. The earth track is connected to the ground plane close to pin 12 of IC27, but nowhere else!!! So I now suggest when gridding the back of the pcb that pin 12 of ICs 28, 29 and 30 be incorporated in the grid thereby incorporating the EPROM sockets firmly into the ground plane. It can't do any harm, and seems to do some good (but difficult to prove).

By this time the back of the memory board is beginning to look like the classic 'rats nest' with bits of wire and resistors hooked all over it. Don't worry, finding the cures, and incorporating the mods was fun wasn't it? And after all the thing works reliably (or should anyway).

NASCOM 1 and 8K BASIC   a bit more
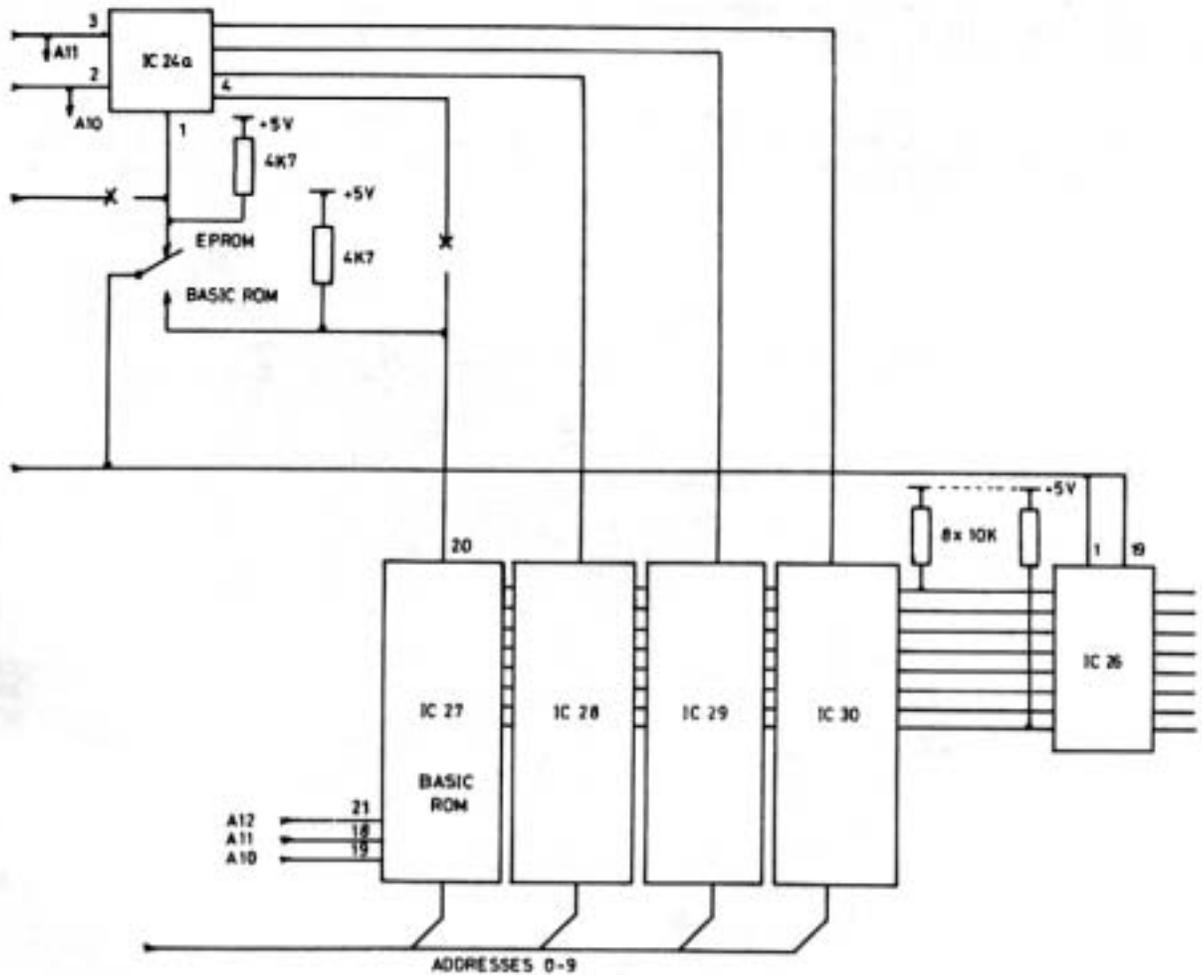
In the last two issues we have published details on how to fit the Nascom Basic
ROM to a series 1 memory card. We have since learned that a good few hundred
are working successfully, but that a couple of dozen did some inexplicable
things. We tended to ignore this as 'finger trouble' until last week when one
of the committee's Nascom 1 suffered an unfortunate accident. (It was
discovered that the PIO input protection is not proof against 240V AC !!!).

As only about three chips were still working, it was decided to re-chip the
Nascom from square one. As the Nascom was pretty well fully expanded this was
expensive, but all worked well except the Basic which normally wouldn't
initialise, or when it did, only worked a few minutes without crashing. We
poked around and found the only difference made when re-chipping was that IC9
on the buffer was originally a 74S04 (see memory plague notes in INMC 5), and
as we didn't have any spare S04s, a 74LS04 was used instead. Well we never
believed that the extra 5nS the S04 gave to the leading edge of MREQ was of any
significance, so how about the extra capacitance an S04 imposed on MREQ?

We tried zapping 150pF across pin 1 of IC9 to ground, and lo !!!, working
Basic. Just to prove this wasn't a fluke, we got in touch with a dealer we
knew had three boards supposedly not working, borrowed them, and without the
capacitor the Basic didn't work, with the capacitor it worked.

So if you have a Basic ROM fitted to a Nascom Series 1 memory, and its giving
you trouble try 150pF between pin 1 and earth of IC9 on the buffer board.
Don't ask us what it's doing, we haven't dared look (something horrible to MREQ
we bet), but it's worth a try.

*STOP PRESS: This preliminary board layout was found by our own Investigative Journalist
on the desk of NW's ace designer Heath Robinson-Crusoe. Concealed beneath a copy of
International Times was also a breadboard prototype, but this disintegrated before it
could be photographed. Robinson-Crusoe was unavailable for comment, but sources close to
the industry suggested that the device was actually available some time last year.*