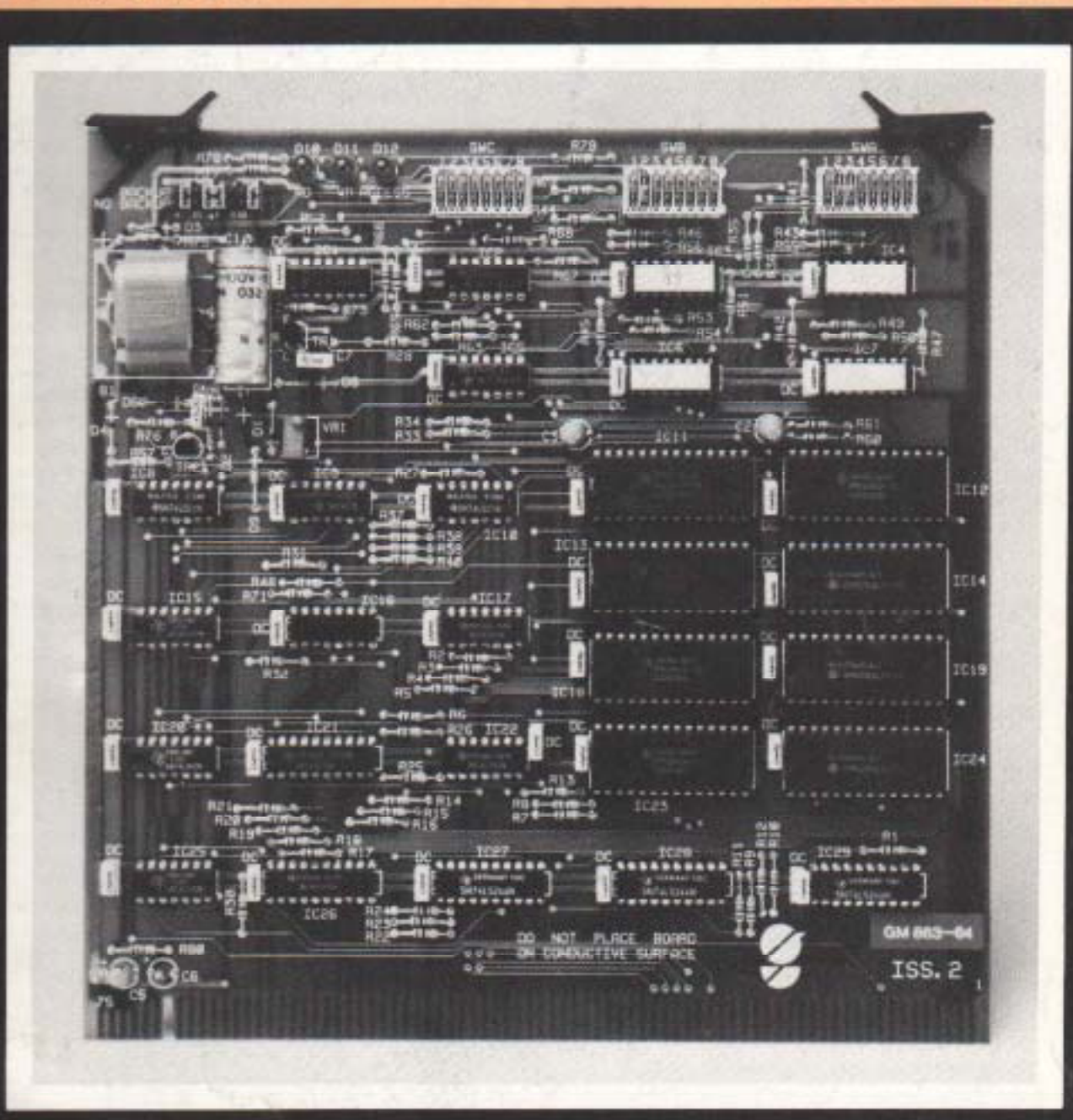
# 8O-BUS NEWS

SUMMER 1985 VOL. 4. ISSUE 2




## GM863 64K STATIC RAM BOARD

SUMMER 1985 **80-BUS NEWS** VOL. 4. ISSUE 2

CONTENTS


| | |
|---|---|
| Page 3 | We are talking "Big" programs here! |
| Page 10 | Private Sales and Wants |
| Page 11 | The Final 80-BUS News |
| Page 11 | An Introduction to CBASIC |
| Page 17 | Ad. |
| Page 18 | A Brief Guide to Mailmerge |
| Page 20 | Shock Horror Headlines - Dave Hunt on: |
| | LU utility |
| | VIRUS program |
| | Gemini GM870 MODEM |
| | UKM720 comms program |
| | Gemini's new BIOS, V3.2 |
| Page 28 | Special deal on back issues |
| Page 29 | Random Rumours (and Truths) |
| Pages 30 - 31 | Ads. |


 Published by Gemini Computer Systems Ltd. Printed by The Print Centre, Chesham.

SUBSCRIPTIONS

Subscriptions for the final two 80-BUS magazines only are available. Once these have been produced subscribers will be put on a Gemini mailing list for a free newsletter. Subscription rates are:

| | | | |
|---|---|---|---|
| UK | £4 | Rest of World Surface | £6 |
| Europe | £6 | Rest of World Air Mail | £10 |

Subscriptions to "Subscriptions" at the address below. Cheques payable to "Gemini Computer Systems Ltd."

EDITORIAL

Editor : Paul Greenhalgh Associate Editor : David Hunt

ADVERTISING

Rates on application to the address below.

PRIVATE SALES

Free of charge for 80-BUS related items by sending to the address below.

ADDRESS: 80-BUS News,
c/o Gemini Computer Systems Ltd.,
Unit 4, Springfield Road,
Chesham, Bucks. HP5 1PW.

# We are talking 'Big' Programs here!

## by Dave Russ

Way back in the twilight of time itself (about 3 – 4 years ago) in a land not too far from here, there could be found a new breed of creature. They were in general a happy band of beings, though they often complained of frequent headaches. You would not see them very often, except maybe first thing in the morning or last thing at night, and if perchance you stumbled upon one in its lair it would seem as if the poor creature was in a trance.

Every now and again they would congregate in cold halls and back rooms and regale each other with stories until the small hours. These tales were not of high adventure or epic endurance but how to coax an extra 1k of RAM from a small home computer thus doubling its capacity and the suchlike. For hour upon hour they would feverishly tell of how they boldly moved in on video RAM and BASIC workspace and worked wonders on a tiny system, that once it had set itself up, left you with an available workspace that was barely into single figures.

This was the heroic age of the early home computer world. Now lost forever, I'm glad to say, though some of the tales still linger on as classics.

However we still have one legacy from those dark days, and I am sure that it is one of Murphy's n laws, and that is whatever system you have, a program will always expand to fill it. Way back then all you could do was read routines into available workspace, wherever it may have been, and through the magic of having read the manual, link it all together to give a running program.

Now that we are living in more enlightened times, and have acquired our 8 bit CP/M computer, we now have oodles of memory to play with (We do???), but still we manage to fill it up somehow. We can now afford to program in high level languages with little regard for the target object code size. Fast, tight assembler type code is now not a prerequisite for an efficient program running on a small computer.

So we still manage to fill our memory up, but these days we have better methods of running programs larger than memory than poking routines into obscure little corners. This is the purpose of this awe inspiring (yawn) article.

It may be worth mentioning that I was talking to an academic type not so long ago who is shortly publishing a book on Artificial Intelligence. He said that any machine in design today should be capable of addressing at least 16 Megabytes of RAM. This seems rather extravagant, doesn't it, but it seems that AI applications are not renowned for their economy with memory.

Getting back to the point, I'm trying to say in my own roundabout way that everyone ought to stop before embarking on a mammoth project and think ahead a little to the time when the program begins to get bigger than the available memory. It may prove difficult to restructure it in six months time, and so we should consider the use of overlaying or chaining even at this early stage.

To give an example, I began a test program for GSX device drivers and at the start I realized that the main menu would have to contain something around 40 options. As it stands this is a prime candidate for overlay techniques, as if I were to add another 20 options along the way I would never have fitted it into memory as a single program. So now I have a single menu/controller program that reads in and executes options as required. Once complete the called subprogram is discarded and the next read into the same memory area.

This approach is by no means revolutionary and is well established. For example take a look at Wordstar and diskpen, these make use of overlays, DR DRAW and DR GRAPH are into double figures with overlay files in their 8 bit versions.

### What to do about it

OK, I hear you shout, so my programs are going to get bigger, what the hell do I do about it??

Well you have a lot of options but they fall into 2 main categories.

1. Program chaining.
2. Program overlays.

### Chaining

Program chaining is simple enough in theory but could prove difficult in practice to implement. When a program is complete it runs the next program in the suite. The first program reads the second into memory over its own memory area and then executes it. Usually no trace of the original program remains.

### Advantages

1. Easy to use if the language used supports a chain function. Program creation using a suitable chain function is simpler than over-laying.
2. Control flow between programs is easy to understand.
3. With the use of some clever code a program management menu could be written to control a number of applications.

**Disadvantages**

1. For application programs chaining could be a slower method of program management than overlays.

2. Variables used in one program cannot (usually) be passed across to the following program.

3. The chain function that you use may error if it does not process the command tail in the same way as the CCP. For example the command:

```
COMP PROG1.COM PROG2.COM
```

when issued normally would call the comparison program COMP to compare the two specified files. Normally the CCP would take this command and load COMP.COM into memory at 100H. The rest of the command tail is placed in memory at 81H with the command tail length at 80H. The command tail is then scrutinized and if it contains filenames then they are put at 5CH and 6CH respectively. The program is then executed.

When run COMP will take a look at the two File control blocks (FCB's) at 5CH and 6CH in order to get the names of the files that it is to compare.

So you can see from this that the chain command will cause an error if it does not perform all the above tasks when invoking a program that requires information in the command tail.

Using a simple chain module that simply read in the COM file and executed it reaked havoc when I chained Wordstar once. Of course the name WS.COM was still in the default FCB at 5C from the reading operation, and so when Wordstar took over it tried to open WS.COM on the disk as a document file, which is no use to anyone. I had forgotten to fill the FCB's with spaces before invoking Wordstar.

**Methods of program chaining**

Of course the simplest method of all is to use a language that has a chain command incorporated in it. Beware here, though, for the very reasons that are mentioned above, they may not work in all cases.

Referring back to Dr. Dark's bit on chaining in *Hisoft* Pascal (80-BUS News, Vol 3. Iss 4), he gave us a method by which programs can run each other via the use of the $$$.SUB file (Read it to find out how) and also suggested that any variables could be passed across by first storing them in a disk file. This is a nice one as it lets the CCP do all the work, but will only work on drive A:.

I had a peep at the chain function in my ECO C library just to get some ideas, and here are the basics of how it works in English.

```
chain(filename)

      open chain file using default FCB at 5CH.

if unable to open then error and return.

read loading routine into area just below BDOS.

set stack pointer below loading routine.

jump to loading routine.

read in chain file.

set stack pointer to BDOS -1.

Jump to program at 100H.
```

You should be able to find a diagram called 'Principles of the CHAIN function' somewhere, if you take a look at that you will see that it represents a memory map type illustration of what is going on inside the computers memory.

Program 1 is an assembler routine, conceived from the above, that will allow you to chain another program. It is only a simple demonstration and will only chain programs that do not expect any information to be contained in the command tail. You will notice that the name of the next program is held in the routine under the label 'fcbh:', this implies that the name of the next program is known. It is possible of course to change this so that any program can be chained from this standard routine. This routine is deliberately simple to demonstrate the basics, if a more complex/universal routine is required then you will have to add the code required to process the command tail and put any valid filenames into the default FCB's. I'm willingly leaving that up to you!!

You will notice that the default FCB at 5CH is filled with spaces after the chain file has been read, this means that Wordstar or Pen can be run without confusing them with odd command tails.

Program 1 will run on its own for testing or may be included in a larger program.

In my search for more information on this subject I took a look at the *Gemini* Utility program KEYCHAIN.COM written by David Parkinson, and later modified by Richard Beal. This program generates a COM file that, when run, will set up the *Gemini* keyboard function keys and then optionally run a named program.

The original version used a 'Naughty but nice' method by which, on exiting to CP/M, the program altered the BDOS pointer at 0006H to point to one of its own routines. When the CCP gained control this routine would intercept all BDOS calls and pass them on to the BDOS as normal until a 'Read console string' came along (Function 10). When this happened the custom routine would immediately return having placed the name of the chain file or command in the CCP command buffer. This let the CCP do all the work which meant that a program could be called using a command tail. However this program has now been modified, and the new version (2.0) uses a method similar to the one that I have outlined above.

I must admit that I have scratched my head a little on where to put the stack pointer before the chain program is executed. (Sensible answers only please.) Keychain V2.0 places it at 80H, and I have put it 1 byte below the BDOS. As the CCP is only required to point the SP at an 8 level stack I suppose 80H would be OK. But in the case of a general purpose chainer the 16 bytes below 80H may be occupied with the second filename in the command tail and thus confuse the issue. I suppose that the ideal answer is to write a chain function to suit your particular purpose at the time and manage the SP accordingly.

And finally, why can't I just read a filename into the CCP command buffer and then call the CCP? Both Steve and I had a look at this a while back, Steve being more conversant with what's what and where it is in CP/M. Well we tried it a few ways calling both the first and second entry into the CCP and neither worked. It seems that the CCP pointer to the command buffer start byte is changed during the process of reading and executing the program. It is not, it seems, restored back to the start of the CCP buffer prior to the command being executed. Thus, the next command that is read into the buffer is either truncated or lost altogether. Regarding this concept do not forget that the previous program may have overwritten the CCP and so the failure of this approach to chaining is guaranteed if this is the case.

**Program overlays**

Program overlays operate on the principle that a controlling module is always present in memory, this is usually called the Root module. If an overlay section is required then the root module will read the contents of the overlay file (which is executable code) into the memory area allocated for overlays. The root module then passes control to the start of the overlay code. On completion the overlay passes control back to the root module.

If, while you are reading all this gumph, you would like to refer to the diagram illustrating overlay techniques, you may find this text more palatable.

**Advantages**

1. Will run faster than a suite of program modules chained together.
2. Will allow the overlay routines to access common variables and utility routines in the root program.
3. You should be able to pass arguments to overlay routines.
4. In some cases you are able to nest overlay sections.
5. The overlay loaders can also be used to load data files into memory if modified.

**Disadvantages**

1. The initial setting up of the overall program is rather complex and can be time consuming.
2. The individual modules cannot be run as separate programs as chained programs can.

**Methods of overlaying**

Now this bit can require a little thought. I think that an efficient overlaid system should have all the common variables and most used utility routines in the root module. This means that they are only read in once at the beginning. This will keep the size of the overlay files down and thus speed up the system. So you are going to have to decide what is going to go into the root. I know that this has little to do with the creation of the files but I put it in to emphasize that there is more to it than hacking the modules together.

Once again I must state that probably the best way of doing it is to begin programming in a language that supports overlays. However this is not always practical and/or efficient. I know of a few languages that will do it, namely CIS COBOL, DR PL1 and Aztec 'C'. There are certainly more but I have not had the pleasure of yet. Overlaying with these is just a matter of reading the instructions.

Assuming that all the files have to be created by hand, and that we are dealing with programs that pass through the .REL stage on their way to the executable .COM stage, here is how we do it.

The whole process depends upon which linking loader you are using at the moment. The *Microsoft* LINK-80 (L80) is the most popular, it seems, but unless I'm wrong it does not allow you to directly create overlay files. The equivalent *Digital Research* product which also seems to go under the name of LINK-80 does give you this facility.

Due to this confusion in names I will call the *Microsoft* product L80 and the DR loader LINK. These seem to be the more commonly used names.

L80 is much easier to use, which probably accounts for its popularity, but does not give you the same facilities as LINK. LINK, for example, will allow you to generate overlay files (OVL), page relocatable files (PRL), and will output a symbol table to disk which could be useful. I only ever use LINK when I need to generate any of these special filetypes for the simple reason that the copy I have demands that all the program modules are entered at the start of the loading process and thus is not very forgiving of the person that forgets to enter the odd module name or two.

Back to overlays now, I recommend that you get hold of the LINK program as it will make your life a lot easier. A word of warning before we start. It must be realized that whatever source language has been used the first byte of the overlay file must be an executable instruction and not a data area as some compilers will create. All my 'C' programs compile down into MAC files and so I can alter the data areas to suit before I create the overlays. Not all of you will have this luxury so beware!

I will now explain the ins & outs of overlaying using the DR LINK linking loader. Let's assume that we have created a program called ROOT and 3 overlay modules called OV1, OV2 & OV3, that will be used by ROOT and are to be run in the same area of memory. The source code has been compiled and we have ROOT.REL, OV1.REL, OV2.REL and OV3.REL on the disk ready to be turned into a COM file with overlays. We now invoke LINK to do its stuff by entering the following:

```
LINK ROOT (OV1) (OV2) (OV3) <RET>
```

And off it goes. Easy isn't it? On completion you will have created ROOT.COM and 3 overlays, OV1.OVL and so on. LINK will have resolved any global references between ROOT and the overlays allowing an overlay to call routines or access variables in the ROOT module. All the overlays will have the same base address and that will be at the start of the next 128 byte boundary above the top of the root module.

It is possible to nest overlays using LINK so that overlays themselves can call in and execute their own overlays (Complex eh?). I will give you the example in the book as I have not yet used this facility and had better stick to safe ground. If you take a look at the system memory map on the overlay techniques diagram it may make the following clearer. By way of an example let's say we have a six overlay system; OV1 to 4 are to be run in the main overlay area, while OV5 & OV6 are to be run in the secondary overlay area. 5 & 6 are to be called from OV2. If you enter the following:

```
LINK ROOT (OV1) ((OV2) (OV5) (OV6)) (OV3) (OV4) 186
                  |_ _ _ _ _ _ _ _|
                        |
                  5&6 nested above 2
```

all the overlays will be sorted out for you. Note the parentheses are nested to indicate the relationship between the overlay sections.

So you can see how easy it is once you have got to this stage but we still have to understand some more on the nitty gritty to enable us to program the overlay loader that handles it all from the root module.

When LINK creates a .OVL file it reads the REL file and resolves the references as normal. It will generate overlay code beginning from the start of the next 128 byte boundary above the root end. The output file (fname.OVL) is in effect a .COM file but LINK will have added a 256 byte header to the beginning of the OVL file. This header area is zero filled except for four bytes. Bytes 1 and 2 (0 based) will contain the length of the executable overlay code, and bytes 7 and 8 will contain the base address of the overlay code. What you as the writer of the overlay loader have to do is extract this information from the header and then read the code into memory starting at the given base address. Once the load is complete then you simply transfer control to the base of the overlay code using a CALL and away you go. As each OVL file contains its own base address a single overlay loading routine is all that is required to load any OVL file.

The above process is a bit of a cheat as the generation of overlay files by LINK was designed specifically to interface with the DR PL-I language, which has an overlay manager in its library. We can use it however if we take note of the little idiosyncracies.

You will notice that the sample overlay loaders (programs 2 & 6) have the variables ?MEMRY and ?OVLA0 in them apparently doing nothing. These are looked for by LINK and reported as undefined if they do not exist as globals. The ?MEMRY word is set to the overall top of program address by LINK as it is creating the overlays. In other words it tells you what the top address of the biggest overlay is and may be used to determine free space or whatever without jeopardizing any of the program memory.

I have not got a clue what ?OVLA0 is meant to do, if anyone knows, please let me in on the secret. I have put it in the programs as a dummy global just to get rid of those undefined global messages from LINK.

Program 2 is an assembler source listing of a working overlay loader, it can be used in conjunct-

ion with programs 3, 4, & 5 to create an overlay demo. If you assemble progs 2,3,4 &5. and then enter.

```
LINK ROOT,OVLAY (OV1) (OV2)<RET>
```

The demo will create itself. It only outputs messages and will not win any prizes but it proves it works.

Program 6 is an overlay loader written in 'C'. This is the one I use for my GSX tester. It uses 'C's pointers to functions that allow you to call things like overlays at absolute addresses and pass parameters to them if necessary. Don't worry if it all seems like gibberish as 'C' can sometimes confuse, it performs in the same manner as the assembler version.

**Using L80 to create overlays**

There is to my knowledge no facility for creating overlay systems from within L80. It can be done but is somewhat complex and should be avoided. You have to hack the code around a lot in memory and I don't like having to admit that I have done it this way on occasion.

The basic principle used here is that you load the root module into memory, and then you load an overlay in at a predetermined address. All global references will now have been resolved. You now exit L80 having saved the program. Using Gemdebug, ZSID or DDT you take the program file and move the overlay code down to 100H and SAVE it as a separate file. Messy but it works. May I suggest at this juncture that while you are messing about with the overlay, you create overlays that are compatible with the LINK format (see previous for details).

Using the sample assembler programs lets create the programs using L80.

1. Assemble all programs to REL format.
2. Enter the following:

```
L80 <RET>              Run L80

ROOT,OVLAY<RET>        Load root module
                       and overlay loader
```

At this point you will see where the top of the root module is by looking at the Data readout from L80 (in this case it is 01BE). We have to decide where the overlay base is to be. Lets put it at 200H and keep things nice and simple. To do this enter:

```
/P:200<RET> Meaning load next bit at 200H.

OV1<RET>          and load in the 1st overlay.

ROOT/N/E          Save ROOT.COM and exit.
```

Now we have to create the overlay from the COM file using a debugger. I have used Gemdebug to create LINK lookalike overlay files. This is how it is done:

```
BUG ROOT.COM<RET>

F100,1FF,0  Fill 256 byte header with 0's
```

Now move the overlay section down to 200H using the M command. In this case we have been lucky as the overlay was originally loaded into 200. If it had started at 0760H for example we would have to enter:

```
M760,800,200<RET>
```

assuming that the overlay code finished at 800.

To maintain LINK compatibility we now have to insert the information concerning the length of code and the base address. To do this use the 'S' (set) command to insert the details. Our overlay is 12 bytes long and its base is at 200H. We set 101 and 102 to 12 and 00, then 107 & 108 to 00 and 02.

We now have a .OVL file in memory and we are ready to save it, enter:

```
SAVE 2 OV1.OVL
```

and the process is complete.

Repeat this for OV2.

Now create the economy version of the COM file by entering:

```
L80 ROOT,OVLAY,ROOT/N/E
```

and its all ready to go.

Not very pleasant is it? It involves a lot of effort and no mean amount of Hex arithmetic to calculate the sizes of the files for the move and save commands. As all this is done for you by the LINK program may I suggest again that you go out and get it and avoid the above.

## Program listings for chain and overlay functions

### Program 1. A routine to chain another program.

```
.z80
bdos    equ     0005h           : BDOS jump address
fcb     equ     005ch           : Default file control block
currec  equ     fcb+20h         : Current record byte in FCB

chain::                         : Start of routine, double colon
                                : flags routine as global to
                                : M80 assembler

    ld      de, fcb             : load chain file dets into default
    ld      hl, fcbh            : file control block at 05ch
    ld      bc, 13
    ldir
    ld      de, fcb             : open up the chain file
    ld      c, 000fh
    call    bdos
    inc     a                   : return on open error (FF hex returned)
    jp      z, opener
    xor     a                   : 0 to current record
    ld      (currec), a
    ld      de, (bdos+1)        : point DE to top of mem (get from 0006h)
    dec     de
    ld      hl, codend          : HL point to end of transfer code
    ld      bc, codend-codbeg   : BC contains length of transfer code
    lddr                        : read transfer code into stack area
    inc     de                  : transfer code addr to hl
    ex      de, hl
    jp      (hl)                : .. and execute it

                                : Following is transfer code
codbeg:
    ld      sp, hl              : sp nc    code beginning
    ld      de, 100h            : DE points to start of program area
rdsec:                          : Read a sector in to memor"
    push    de
    ld      c, 01ah             : set dma to current memory addr
    call    bdos
    ld      de, fcb             : read next sector
    ld      c, 14h
    call    bdos
    pop     de
    or      a                   : last sector ??
    jr      nz, filred          : ... yes so exit loop
    ld      hl, 80h             : move memory pointer up 128 bytes
    add     hl, de
    ex      de, hl
    jr      rdsec               : not finished so read another sector

filred:
    ld      a, 0                : 0 length command tail
    ld      (80h), a
    ld      de, 80h             : reset DMA to default
    ld      c, 1ah
    call    bdos
    ld      hl, 5CH             : fill FCB with spaces
    ld      a, 20H
    ld      b,?2
    ld      (ovcall+1), hl  ; and write it into the call data area
ovcall:
    call    0000            ; call the overlay. The 0000 address will
                            ; have been altered to the overlay base
                            ; addr by the previous instruction.
    ret
getsec:                     ; read a sector
    ld      c, 14h
    ld      de, fcb
    call    bdos
    ret
filerr:                     ; file error routine
    ld      c, 09
    ld      de, errmes
    call    bdos
    ret

errmes: defm    'Unable to open overlay file',10,13,'$'

?memry::                    ; LINK will put the top of program
    defs    2               ; address here

?ovla0::                    ; Link looks for this - Reason not known
    defs    2
    end
```

### Program 3. Root module for overlay demo. (ROOT.MAC)

```
    .z80
root::
    ld      de, mess            ; output root message
    ld      c, 09
    call    0005
    ld      de, 5ch             ; load fcb with 1st overlay filename
    ld      hl, fil1
    ld      bc,13
    ldir
    call    ?ovlay##            ; call overlay loader
    ld      de, 5ch             ; load fcb with ... overlay filename
    ld      hl, fil2
    ld      bc,13
    ldir
    call    ?ovlay##            ; call loader again
    jp      0                   ; exit to CP/M


fil1:   defb    0,'OV1     OVL',0  ; overlay filenames
fil2:   defm    0,'OV2     OVL',0

mess:   defm    'In root',10,13,'$'
    end
```

### Program 4. 1st overlay for demo program. (OV1.MAC)

```
    .z80
ov1:
    ld      c, 09               ; output id message
    ld      de, mess
```

ion with programs 3, 4, & 5 to create an overlay demo. If you assemble progs 2,3,4 &5. and then enter.

```
LINK ROOT,OVLAY (OV1) (OV2)<RET>
```

The demo will create itself. It only outputs messages and will not win any prizes but it proves it works.

Program 6 is an overlay loader written in 'C'. This is the one I use for my GSX tester. It uses 'C's pointers to functions that allow you to call things like overlays at absolute addresses and pass parameters to them if necessary. Don't worry if it all seems like gibberish as 'C' can sometimes confuse, it performs in the same manner as the assembler version.

**Using L80 to create overlays**

There is to my knowledge no facility for creating overlay systems from within L80. It can be done but is somewhat complex and should be avoided. You have to hack the code around a lot in memory and I don't like having to admit that I have done it this way on occasion.

The basic principle used here is that you load the root module into memory, and then you load an overlay in at a predetermined address. All global references will now have been resolved. You now exit L80 having saved the program. Using Gemdebug, ZSID or DDT you take the program file and move the overlay code down to 100H and SAVE it as a separate file. Messy but it works. May I suggest at this juncture that while you are messing about with the overlay, you create overlays that are compatible with the LINK format (see previous for details).

Using the sample assembler programs lets create the programs using L80.

1. Assemble all programs to REL format.
2. Enter the following:

```
L80 <RET>            Run L80

ROOT,OVLAY<RET>      Load root module
                     and overlay loader
```

At this point you will see where the top of the root module is by looking at the Data readout from L80 (in this case it is 01BE). We have to decide where the overlay base is to be. Lets put it at 200H and keep things nice and simple. To do this enter:

```
/P:200<RET> Meaning load next bit at 200H.

OV1<RET>         and load in the 1st overlay.

ROOT/N/E         Save ROOT.COM and exit.
```

Now we have to create the overlay from the COM file using a debugger. I have used Gemdebug to create LINK lookalike overlay files. This is how it is done:

```
BUG ROOT.COM<RET>

F100,1FF,0  Fill 256 byte header with 0's
```

Now move the overlay section down to 200H using the M command. In this case we have been lucky as the overlay was originally loaded into 200. If it had started at 0760H for example we would have to enter:

```
M760,800,200<RET>
```

assuming that the overlay code finished at 800.

To maintain LINK compatibility we now have to insert the information concerning the length of code and the base address. To do this use the 'S' (set) command to insert the details. Our overlay is 12 bytes long and its base is at 200H. We set 101 and 102 to 12 and 00, then 107 & 108 to 00 and 02.

We now have a .OVL file in memory and we are ready to save it, enter:

```
SAVE 2 OV1.OVL
```

and the process is complete.

Repeat this for OV2.

Now create the economy version of the COM file by entering:

```
L80 ROOT,OVLAY,ROOT/N/E
```

and its all ready to go.

Not very pleasant is it? It involves a lot of effort and no mean amount of Hex arithmetic to calculate the sizes of the files for the move and save commands. As all this is done for you by the LINK program may I suggest again that you go out and get it and avoid the above.

## Program listings for chain and overlay functions

### Program 1. A routine to chain another program.

```
.z80
bdos    equ     0005h           ; BDOS jump address
fcb     equ     005ch           ; Default file control block
currec  equ     fcb+20h         ; Current record byte in FCB

chain::                         ; Start of routine, double colon
                                ; flags routine as global to
                                ; M80 assembler

    ld      de, fcb             ; load chain file dets into default
    ld      hl, fcbh            ; file control block at 05ch
    ld      bc, 13
    ldir
    ld      de, fcb             ; open up the chain file
    ld      c, 000fh
    call    bdos
    inc     a                   ; return on open error (FF hex returned)
    jp      z, opener
    xor     a                   ; 0 to current record
    ld      (currec), a
    ld      de, (bdos+1)        ; point DE to top of mem (get from 0006h)
    dec     de
    ld      hl, codend          ; HL point to end of transfer code
    ld      bc, codend-codbeg   ; BC contains length of transfer code
    lddr                        ; read transfer code into stack area
    inc     de                  ; transfer code addr to hl
    ex      de, hl
    jp      (hl)                ; .. and execute it

                                ; Following is transfer code
codbeg:
    ld      sp, hl              ; sp nc   code beginning
    ld      de, 100h            ; DE points to start of program area
rdsec:                          ; Read a sector in to memor"
    push    de
    ld      c, 01ah             ; set dma to current memory addr
    call    bdos
    ld      de, fcb             ; read next sector
    ld      c, 14h
    call    bdos
    pop     de
    or      a                   ; last sector ??
    jr      nz, filred          ; ... yes so exit loop
    ld      hl, 80h             ; move memory pointer up 128 bytes
    add     hl, de
    ex      de, hl
    jr      rdsec               ; not finished so read another sector

filred:
    ld      a, 0                ; 0 length command tail
    ld      (80h), a
    ld      de, 80h             ; reset DMA to default
    ld      c, 1ah
    call    bdos
    ld      hl, 5CH             ; fill FCB with spaces
    ld      a, 20H
    ld      b,Z2

fil5ch:
    ld      (hl), a
    inc     hl
    djnz    fil5ch
    ld      hl, (ovcall+1)      ; get the overlay call address
    ld      (ovcall+1), hl      ; and write it into the call data area
ovcall:
    call    0000                ; call the overlay. The 0000 address will
                                ; have been altered to the overlay base
                                ; addr by the previous instruction.
    ret
getsec:                         ; read a sector
    ld      c, 14h
    ld      de, fcb
    call    bdos
    ret
filerr:                         ; file error routine
    ld      c, 09
    ld      de, errmes
    call    bdos
    ret

errmes: defm    'Unable to open overlay file',10,13,'$'

?memry::                        ; LINK will put the top of program
    defs    2                   ; address here

?ovla0::                        ; Link looks for this - Reason not known
    defs    2
    end
```

### Program 3. Root module for overlay demo. (ROOT.MAC)

```
    .z80
root::
    ld      de, mess            ; output root message
    ld      c, 09
    call    0005
    ld      de, 5ch             ; load fcb with 1st overlay filename
    ld      hl, fil1
    ld      bc,13
    ldir
    call    ?ovlay##            ; call overlay loader
    ld      de, 5ch             ; load fcb with 2nd overlay filename
    ld      hl, fil2
    ld      bc,13
    ldir
    call    ?ovlay##            ; call loader again
    jp      0                   ; exit to CP/M


fil1:   defb    0,'OV1     OVL',0  ; overlay filenames
fil2:   defm    0,'OV2     OVL',0

mess:   defm    'In root',10,13,'$'
    end
```

### Program 4. 1st overlay for demo program. (OV1.MAC)

```
    .z80
ov1:
    ld      c, 09               ; output id message
    ld      de, mess
```

```
filfcb:
    ld      (hl), a
    inc     hl
    djnz    filfcb
    ld      sp, (0006h)          ; set SP to bdos -1
    dec     sp
    jp      100h                 ; execute program
codend:
    nop

opener:                          ; outputs error mess & returns
    ld      de, errmsg
    ld      c, 9
    call    bdos
    ret

errmsg: defm    'Chained file open error$'

fcbh:   defb    0                ; details of the file to chain.
        defm    'CHAINFILCOM'    ; Note- this is in standard CP/M
        defb    0                ; FCB format
        end
```

### Program 2. Assembler version of an overlay loader. (OVLAY.MAC)

```
.z80
fcb     equ     05ch            ; declare default fcb
bdos    equ     05h             ; entry to BDOS
currec  equ     fcb+32          ; current record byte in fcb

?ovlay::
        ld      c, 0fh          ; open overlay file
        ld      de, fcb
        call    bdos
        inc     a               ; error if file not found
        jp      z, filerr
        xor     a               ; put 0 to current record
        ld      (currec), a
        ld      c, 1ah          ; set dma buffer to 80h
        ld      de, 80h
        call    bdos
        call    getsec          ; read 1st 128 bytes
        ld      hl, (87h)       ; get base page of code
        push    hl              ; .. and store it, once as buffer addr
        push    hl              ; and again for the overlay call later on
        call    getsec          ; read 2nd 128 bytes
                                ; we are now at start of code
        pop     de              ; restore base addr of overlay area
rdovl:
        push    de              ; store current buffer addr
        ld      c, 1ah          ; set dma current buffer area
        call    bdos
        call    getsec          ; read a sector
        pop     hl              ; restore buffer addr
        ld      de, 128         ; and add 128 to it so that the next
        add     hl, de          ; overlay sectors are read into memor"
        ex      de, hl          ; sequentially
        or      a               ; was it the last sector
        jr      z, rdovl        ; ... no so go get another
        pop     hl              ; pop the base addr into hl
        ld      de, mess

        call    0005
        ret                             ; return to root

mess:   defm    'In ovl',10,13,'$'
        end
```

### Program 5. 2nd overlay for demo program. (OV2.MAC)

```
        .z80
ov2:
        ld      c, 09                   ; output id message
        ld      de, mess
        call    0005
        ret                             ; return to root

mess:   defm    'In ov2',10,13,'$'
        end
```

### Program 6. An overlay loader written in 'C'. (ECO 'C' used).

Please note that ECO C translates all underscores in variable names to '?' and so __memry is translated to its correct form ?MEMRY.

```
#define READ_ONLY 0
#define ERROR 0
#define OK 1

int (*_ovla0)();                        /* ?0vla0 used as pointer to func */
int _memry;                             /* set to top of mem by LINK */
                                        /* These are declared as global  */
                                        /* so that they are identified to */
                                        /* the LINK program when linking. */

_ovlay(fname)                           /* overlay file name passed as arg */
char *fname;
p37
FILE *ofd;                              /* declare overlay file descriptor */
int ovlen;                              /* used to store length of ov code */
static char currovl[16];                /* stores name of current overlay */
char buff[128];                         /* local input buffer */
int *data_ptr=&buff[1];                 /* integer pointer used to access */
                                        /* int values in input buffer */

        /* Start of overlay routine */

if((ovlen=strcmp(currovl,fname))==0)    /* if the current overlay in memory */
                                        /* is same as that requested  */
        p37
        (*_ovla0)();                    /* then call the function */
        return(OK);                     /* and return to caller */
        p38
                                        /* else we have to read it in */
ofd=open(fname,READ_ONLY);              /* open overlay file */
if(ofd==EOF) return(ERROR);             /* if not found return error code */
read(ofd,buff,128);                     /* read 1st 128 bytes */
ovlen=*data_ptr;                        /* .. & extract code length from it */
data_ptr+=3;                            /* point to byte 7 of input buffer */
```

# Oops!

This page should have contained a diagram to illustrate the text of the surrounding article! Unfortunately it has been mis-placed, and the author has not kept a copy. However, he assures me that it is not essential and so, rather than leaving a blank page, we will take a short commercial break!

## Private Advertisements

FOR SALE

GM803 EPROM Card, Real Time Clock Module by EV Computing for GM812, GM805 case with 2 Pertec FD250 disk drives and mains PSU. Software and data available if required. Tel: C. Bowden 0209-860480.

Nascom 2 with RAM (cased). EPROM blower. Level 9 BASIC, ZEAP, Assembler, Disassembler, NasPen, Toolkit, SYS 3. ROM or EPROM. Most of a Nascom 1, C/B i/f, Merseyside ROM board, no PSU. Tandy DMP110 & CGP115 printers, little used. APF 10" monitor. Sanyo and Sharp cassette recorders. Manuals where appropriate. All in good or very good condition. Sensible offers please. Tel: J. Turner, South Elmsall (0977) 45424.

Nascom 2 in Kenilworth case, 5 card frame, 3 inch fan, 32K RAM A board, 3A PSU, Nas-Sys 3, BASIC ROM. ZEAP 2, Nas-Dis and Nas-Debug in EPROMs. CCSoft Graphpac on cassette. 12" B/W TV converted for use as monitor/TV. Complete sets of INMC, Micropower and 80-BUS News magazines + Nascom Program Books + some tapes. £250 or offers. Also Bits and PCs EPROM Programmer for 2708 or 2716's, built but not tested. £20 ono. Gemini GM803 EPROM board kit - Brand new, still boxed. £50 ono. Tel: Steve Wright, Horsham, W. Sussex (0403) 69148 evenings or weekends.

IVC with EV beeper and installed EV Real Time Clock/Calendar. Fully programmable, inc. documentation and disk demo, £110; CPU card (GM811), RP/M V2 installed, £50; MAP FDC/Video card, FDC only populated, £90; Juki 6100 tractor paper feed (New - boxed), £70; Cannon 210, 40 track double sided disk drives 500K unformatted, two at £60 each. All ONO and in excellent condition. Tel: Dave Birch on 051-709-4465 (evenings).

Micropolis 1015 400K (new drive belt, good order) £90 ono, 2 x 48K RAM B Boards with Page Mode £85 ono, GM805 Henelec Disk System & ROMs for Nascom 2 with CP/M 1.4, Utilities, Assemblers (M80 etc), Languages, Forth, Fortran, C, Pascal, BASIC, £75 ono. Also PHG Sound Board very cheap, Philips DCR Cassette Drive & Controller - even cheaper. Any offers considered. Tel: Ian - Ipswich (0473) 831353.

Galaxy 2 Micro, 2 x 800K diskettes, amber screen, two years old but only lightly used. Offer includes 20 diskettes in protective boxes and CP/M and Wordstar/Mailmerge software. Offers around £750 please. Tel: Worcester (0905) 830612, evenings or weekends.

WANTED

AVC, IVC and disk system + CP/M 2.2. Eric Wright 091 285 3762.

```
__ovla0=*data__ptr;          /* ..& write base addr into the */
                            /* function pointer */
read(ofd,buff,128);          /* read the next 128 bytes. We have */
                            /* no use for these */
read(ofd,__ovla0,ovlen);     /* Read ovlen bytes into area */
                            /* pointed to by __ovla0  */
close(ofd);                  /* close overlay file */

(*__ovla0)();                /* call overlay */
strcpy(currovl,fname);       /* update the current overlay name */
return(OK);                  /* return alls well to main prog */
p38
```

## Final Issue

Please note that we intend the final issue of 80-BUS news to be a little silly. (Yes, OK, I know that virtually every issue has been silly, but that's not what I mean.)

If you have any Nascom/Gemini etc related tales that you think will give other readers a chuckle then please send them in. Nothing libellous though please!

You might like to relate an anecdote on the troubles you have had building up your system, getting to grips with software, mis-understandings with the Mrs., or how YOU had to explain to the dealer what a RAM chip is!

I can certainly remember certain things; like the user who soldered his Nascom 1 together with all the components on the wrong side of the board. And another who melted a load of solder in a frying pan, and then dipped his Nascom 1 into it to "flow solder" it!!

---

# An introduction to CBASIC

## by P.D.Coker

CBASIC is one of several dialects of BASIC available to users of 80-BUS machines. It originated from the fertile mind of Gordon Eubanks Jr. and first became available to micro users in 1977, quickly followed by an enhanced version, CBASIC2 in the latter part of 1978. Dr Eubanks had developed the idea of a type of compiled BASIC as part of his doctoral research, soon after CP/M had been released on an unsuspecting public, and this became known as EBASIC but has now been almost completely superseded by CBASIC which has many enhancements and is more suitable for programming purposes. CBASIC2 was originally a product of *Compiler Systems Inc.*, the firm which Eubanks founded in California but is now under the tender care of *Digital Research Inc.* In this article, CBASIC refers to the second release, CBASIC2. An expensive but versatile version including a true 'native code' compiler, known as CB-80 is also available from *DRI.*

Most people will have heard of *Microsoft* BASIC (MBASIC) in one or other of its forms, and once one becomes familiar with the quirks and deficiencies of the particular version one has available, one's style of programming adapts accordingly. So why bother about another version of BASIC? Certainly, for many purposes, MBASIC performs quite adequately as an interpreted language, with the option of a compiler, while CBASIC is a 'pseudo-code' compiled (sometimes known as 'semi-compiled') language.

Unlike compilers, interpreted BASICs allow the programmer to develop and test sections of a program immediately using the built-in line editing facility, and syntax or other errors can be spotted when the program or segment is RUN. On-screen editing is available to enable offending lines to be corrected. Additionally, MBASIC has a means of compacting the code as it is entered, to save time and memory when the program is executed. From almost every point of view, a satisfactory situation. What are the drawbacks to this paragon of virtuosity? Essentially, the interpreter has to evaluate each line as it is executed, and this can slow down the execution speed of programs which handle large amounts of data or, more importantly, those in which loops or calls to subroutines, more familiarly known as 'GOSUB ... RETURN' occur. This can be significantly changed by converting the program after it has been fully tested in to a 'compiled' form, where the program is efficiently compacted and (in a true compiler), converted into machine code instructions. CBASIC does not have a true compiler, but instead produces an efficiently packed code from a source program which is

entered using a text editor such as ED (if you must), WordStar or PEN. From the source code program (which must have the extension type .BAS), it produces an .INT file in which, for example, 'PRINT' is stored as 1Ah. The logic behind this is that it takes a shorter time to interpret 1 byte such as 1A when the semi-compiled program is run than it does to deal with 5 letters as in PRINT. To use the language, you need to:

1. Write a CBASIC source program.
2. Use CBASIC to produce the intermediate (.INT) file.
3. Use CRUN to execute the .INT file.

CRUN is a run-time interpreter which takes the .INT file produced by CBASIC after checking the source code for errors, and converts it to machine code. The pseudo-code compiler/run-time interpreter combination is a compromise between the readily alterable but slow-running interpreted and the fast-running but not easily alterable compiled versions of BASIC.

The same procedure could be used for EBASIC, which, I believe, is available as Public Domain software at a very modest charge from the CP/M Users Group.

CBASIC appears to be more cumbersome in use than MBASIC, but it has a number of very useful features which the latter lacks. The major use for it has been in the field of business applications software where its speed of execution and facilities have been most favoured. Unfortunately, it is only available for disk-based CP/M systems.

The language takes a little getting used to and I would not recommend it to anyone who has less than a good grasp of BASIC: until one is reasonably familiar with its syntax and little peculiarities, it is best not to use some of the more advanced features. In the following pages, I have attempted to provide some information on CBASIC and MBASIC (disk version) statements and commands, where they differ or where they are peculiar to CBASIC. CBASIC has an excellent, if expensive *User Guide* by A. Osborne, G. Eubanks and M. McNiff, published by Osborne/McGraw-Hill in 1981. It is rather costly at £12 (in 1983) for 215 pages but is essential for the serious user. The Guide deals with version 2.07 of the compiler; I have used version 2.02 only, but I have not found any major problems or incompatibilities. The version number of the run-time interpreter CRUN used in the book is not stated but mine was 2.04.

### Data types

Both BASICs support integer and floating point (real) numbers, with exponential notation for the latter. The range for integers is the same in both (+32767 to -32768), but real numbers are stored by CBASIC with an accuracy of 14 digits, compared with 7 for single precision and 17 for double precision MBASIC. Exponential range for MBASIC is 2.9387E-38 to 1.70141E38, whereas CBASIC has a range from 9.9999999999999E62 to 1.0E-64, which could be very useful!! Both BASICs allow hexadecimal notation, but CBASIC allows binary and MBASIC permits octal notation in addition. All three are, of course, special ways of representing integer numeric data in BASIC.

Names of variables may be up to 31 characters long in CBASIC; although in MBASIC they can be of any length, only the first two alphanumeric characters are recognized; some characters are not allowed but who wants to use ★,!,@ or & in a variable name? Both BASICs use the $ sign to denote string variables, and integers may be explicitly declared. This is done in CBASIC by adding a % sign to the end of the variable name (with not more than 30 preceding characters); in MBASIC, the DEFINT statement does this. The ability to use variable names like 'microcomputer', 'collywobble' or 'knightsbridge' may appeal to the whimsical but unfortunately, 'supercalifragilisti-expialidocious' is too long and would be truncated! Variable names should be distinct and descriptive, which will help you understand the flow of each program you write.

DIMension statements have virtually the same functions in both BASICs; a little appreciated fact is that it is possible to use a constant or a variable to declare the highest subscript for an array or vector:

```
Z%=50
DIM SORTCODE$(Z%),HOUSENUMBER(Z%)
```

or

```
Z%(1)=50
Z%(2)=5
DIM SORTCODENUMBERS$(Z%(1),Z%(2))
```

### Statement construction

Line numbers are optional in CBASIC except where the line is referenced by a statement elsewhere in the program. An additional and somewhat bizarre feature is that the numbers can be in any format — integer, real or exponent, but there must be less than 31 digits.

A single CBASIC statement can be spread over several lines by the use of the backslash (\) and lower case letters are converted to upper case unless the 'D' toggle is used.

```
IF REPT = 0 \
THEN GOTO 999
```

There seems to be no limit to the length of a statement but keep it within reasonable bounds otherwise your source will look like one of those

nasty Beeb programs! An additional, useful facility is that text can be put after the backslash — a subtle form of REM, and, as in MBASIC, two or more statements on one line must be separated by a colon.

```
IF REPT = 0 \ Check for repetition
THEN GOTO 999: PRINT"REPEATING TEST"
```

The statement can start anywhere on the line — but it is obviously better to use the first 80 columns.

### Data I/O

Both BASICs use the conventional PRINT statement with semi-colon or comma separators; in CBASIC, the comma indicates a move to the next printing field which is every 20th print position and the semi-colon leaves one space after the last printed item and then starts printing again. PRINT USING, WIDTH and TAB are also supported by both but 3 statements, LPRINTER, CONSOLE and POS are specific to CBASIC. LPRINTER directs all PRINT statement output to the printer until it is cancelled by the CONSOLE statement which routes it back to the screen. LPRINTER assumes a print WIDTH of 132 characters unless a lower figure is set (e.g. 80):

```
LPRINTER WIDTH 80
```

CONSOLE assumes a screen width of 80 characters unless otherwise directed.

The POS function is used to identify how many characters have been printed or displayed on a line — useful if you want to see how much space remains. I haven't used this facility very often!

### File handling

In CBASIC there are two types of file, sequential and relative. Their use is rather more straightforward than in MBASIC. The sequential file type is better in terms of disk space usage but the relative file can be read or written to at random since it assigns a fixed space to each possible record. The following remarks apply broadly to both types of file but OPENing or CREATEing relative files involves the RECL parameter in which the record length must be specified:

```
100 OPEN "A:ADDRESS.LIS" RECL 128 AS 3
```

Files are opened using CREATE, OPEN or FILE statements and closed by CLOSE (and CHAIN or STOP if these statements occur in a program). CREATE, as its name suggests, sets up a new file, deleting any previous file with the same name, OPEN — well, it just opens a file which has already been created in the directory. FILE is very useful since it will CREATE a new file and OPEN it if it does not currently exist — and it will OPEN a file for I/O if it exists. All these commands can be used to operate more than one file at a time.

OPEN and CREATE have the same syntax, but FILE does not need the identifier to be specified — it does this automatically:

```
OPEN "B:TEDIOUS.DOC" AS 2
CREATE "A:TEST.DOC" AS 1
FILE "A:DAVEHUNTSBITS"
```

In the first two cases, a file and drive are specified and given a numeric identifier — thus TEDIOUS.DOC was originally set up on disk B and has the identifier 2 — which will be used for subsequent file operations using READ # and PRINT #:
the file on drive A called DAVEHUNTSBITS will be assigned 3 as its identifier if it was in the same program; this is because it is the third file to be specified.

```
READ #2;A$ - reads string variable
             A$ from file 2 which in
             this example, is TEDIOUS.DOC
PRINT #1;A$ - writes the same string
             variable to file 1 - TEST.DOC
```

PRINT USING # is a very useful facility if you have a number of different types of output format to include; both PRINT # and PRINT USING # are limited to a maximum number of 20 file identifiers! To CLOSE a file, simply type CLOSE <file identifier> — very neat and just like MBASIC.

Other useful statements are DELETE <identifier> which enables one to delete files when they are no longer required in a program and INITIALIZE which allows one to change a disk without causing the system to become bewildered or risking the scrambling of directories. One can also RENAME a file, determine its SIZE in 1 Kbyte increments, or employ the IF END # statement to do something mind-boggling when the contents of a file have been read and Marvin or whatever you call your machine is about to become seriously confused!

```
RENAME (NEWFILENAME$,OLDFILENAME$)
PRINT SIZE("FILENAME$")
IF END  2 THEN 999 \STOP THE PROGRAM
```

CBASIC defaults to a record length of 128 bytes for RECL; additionally, one can specify the data buffer length and sector length using the BUFF and RECS parameters; these follow the numeric file identifier. Further information on file handling can be gleaned from the Guide.

### Numerical operations

The major differences between the two BASICs are in the field of relational operators (less than, greater than etc.). The conventional combinations of <, > and = are supplemented by letters:

| *MBASIC* | *CBASIC* |
|---|---|
| = | = or EQ |
| < | < or LT |
| <= | <= or LE |
| <> | <> or NE |
| > | > or GT |
| >= | >= or GE |

The same letters are used for relational operators in FORTRAN which also uses the same set of logical operators — NOT, AND, OR and XOR — as both the BASICs.

### Intrinsic functions

Most of the intrinsic functions in CBASIC are found in MBASIC. The exceptions are as follows:

FLOAT — converts an implicitly declared integer to its real number equivalent; if a real number is used instead, it is converted to an integer and then reconverted to its real equivalent.

INT% — this truncates a real number at the decimal point and returns an integer result. If a real number is to be converted, it is first converted to a real number before truncation. Note that this is not the same as the INT function which actually produces a real result.

This highlights the fact that unnecessary conversions and other operations will slow down a program, and to minimize this, one should make sure that any conversions are done using the right type of number.

### User-defined functions

These are almost identical in both BASICs; the only major difference is that, in addition to the letters FN, up to 29 other alphanumeric characters may be added in CBASIC.

### String functions

Compared with MBASIC, CBASIC has several more string functions; the following will give some idea of the flexibility of this aspect of the dialect:

COMMAND$ — returns the CP/M command line without the name of the CP/M program being invoked. You probably won't use this a lot.

MATCH(A$,B$,I%) — looks for string A$ starting at the I%th character in string B$. If it is present, the function returns the position in B$ of the first A$ character; if it is not present, the value 0 is returned.

SADD(A$) — returns the starting address in memory where CBASIC has stored A$.

UCASE(A$) — converts lower case letters to upper case in string A$.

### Program logic

In addition to the normal IF ... THEN ... <ELSE> statement, CBASIC has an amazing feature called the Compound IF statement where many comparisons can be made. A program which illustrates this and the use of other CBASIC features may help explain this very useful facility:

```
REM Silly program in CBASIC to illustrate
REM ompound IF statement and other features.
    INPUT "ENTER THE VALUE OF THE COIN ";
              COIN.VALUE%
    COIN.COLOUR1$="GOLD-COLOURED":COIN.
              COLOUR2$="SILVER"
    COIN.COLOUR3$="BRONZE"
    PRINT"YOU INPUT A VALUE OF ";COIN.VALUE%
    IF COIN.VALUE% GT THAN 100 \
    THEN PRINT"CAN'T DO THIS ONE!":GOTO 999
    IF COIN.VALUE%= 20 OR 100 THEN GOTO 10
    IF COIN.VALUE%=5 OR 10 OR 50 THEN GOTO 20
    IF COIN.VALUE%=1 OR 2 THEN GOTO 30
10  IF COIN.VALUE%=20
    THEN PRINT \
    "YOUR COIN IS A 20p PIECE AND IT IS A":
    PRINT"DIRTY YELLOW COLOUR":GOTO 999
    IF COIN.VALUE% EQ 100 \
    THEN PRINT"YOUR £1 COIN IS ":
              COIN.COLOUR1$: \
    GOTO 999
20  IF COIN.VALUE% EQ 50 \
    THEN PRINT"YOUR 50p COIN IS ":\
    COIN.COLOUR2$:GOTO 999
    IF COIN.VALUE% LT 50 \
    AND GT 5  THEN PRINT\
    "YOUR 10p COIN IS ":COIN.COLOUR2$:GOTO 999
    IF COIN.VALUE% = 5 \
    THEN PRINT"YOUR 5p COIN IS ";\
    COIN.COLOUR2$:GOTO 999
30  IF COIN.VALUE% LT 5 THEN PRINT
              "YOUR COIN IS A ";\
    COIN.COLOUR3$:
    IF COIN.VALUE% NE 1 \
    THEN PRINT" 2p PIECE"
    IF COIN.VALUE%  EQ 1 THEN PRINT" 1p PIECE"
999 END
```

Another useful pair of statements found in both BASICs, is WHILE/WEND; they are essentially a variation of the FOR/NEXT statements and will allow statements inserted between them to be continually executed as long as the expression following WHILE is logically true. As soon as it is logically false, the sequence terminated and the next statement after WEND is executed. This is quite a nifty idea and additionally, Compound IF statements can be included between them. The following example illustrates the action of the WHILE/WEND pair:

```
    N=0
1   WHILE N LT 10
    N=N+1
    PRINT"N = ";N
    GOTO 1
    WEND \SKIPS OUT WHEN N = 10
    PRINT"THAT'S ALL, FOLKS!"
    END
```

WHILE/WEND loops can be nested in the same way as FOR/NEXT loops.

**'Subroutines'**

The use of GOSUB...RETURN as a form of 'subroutine' in BASIC has been the subject of much hot air and plonking arguments, culminating in the DEFPROC/PROC/ENDPROC to be found in BBC BASIC. If used properly, there is nothing wrong with GOSUB, whether straight or computed. CBASIC has an additional feature over MBASIC and this is the multiple-line function. This clever device enables those who are allergic to the use of GOSUB to define the 'subroutine' function as follows: (note the full stop between the FN and the dummy argument)

```
DEF FN.CROSS.SECTION(AREA)
```

In this case, the function name (in this case, CROSS.SECTION) is followed by a dummy argument, AREA, which is a real numeric variable used within the function and CROSS SECTION is treated as a variable which may have values assigned to it within the function itself — in this way, values of variables needed within the 'subroutine' can be input without the need to define them in assignments. Constants or expressions cannot be used as dummy arguments — only variables can be used in this way. The variable can be a string, integer or real type and is treated as local to the function, and only able to be modified within the function. The function must be terminated with a RETURN, followed by FEND; FEND is not executed under normal circumstances but if RETURN is missed out, then CRUN will send an error message and return to CP/M.

Initially, the use of multiple line functions seems a little difficult but with experience, the advantages of the facility become apparent. Essentially, the CBASIC function follows the FORTRAN method. The User Guide is reasonably clear on the use of this type of function, and provides some examples of its application.

**Overlaying and Chaining**

In spite of the suggestive names, these have nothing at all to do with S★X, kinky or otherwise! Overlaying is employed when a program is too big to fit into the available memory; the program is broken down into smaller units, called overlays and these are stored as separate, compiled programs. An overlay is called when the previous program reaches a CHAIN statement which explicity names the overlay in question before it completes its execution. The new overlay can end by calling yet another overlay, or the previous one or even by stopping program execution. Two important points should be borne in mind before using overlays in CBASIC.

1. An overlay overwrites the previous program but the previously used program can pass data to the overlay by means of COMMON statements which leave the data in memory; these COMMON statements must be in both the calling program and overlay.

2. Memory usage is slightly restricted by the use of overlays; full details of this are to be found in the CBASIC User Guide.

CHAINing allows COMMON variables to remain intact while the area of memory used for storage of file buffers, strings and arrays is compacted. If the %CHAIN directive is used, the system cunningly instructs the first of a series of programs to determine the maximum storage capacity of all following program overlays; this maximum figure will be used to determine how much memory will be allocated to each of the areas required by the program suite.

In its simplest form, CHAIN is used as follows:

```
REM FINISH PREVIOUS PROGRAM THEN
REM LOAD NEXT OVERLAY
CHAIN "NEWPROG"
```

An arrangement which allows an overlay to be loaded from a different disk drive follows; this assumes that the logged in drive is A and that the next program resides on drive B:

```
DISK.DRIVE$ = "B:"
INPUT"NAME OF NEXT PROGRAM";PROGRAM.NAME$
CHAIN DISK.DRIVE$+PROGRAM.NAME$
```

The COMMON statement is not found in other BASICs, and its use follows the FORTRAN precedent as being the first statement in a program, agreeing in type and sequence in both sending and receiving programs. Thus if variables A, B$ and C% are produced by the sending program, the receiving program must have either the same variables in the same order or different named variables of the same type in the same order, i.e. X, Y$ and Z%.

Array variables in COMMON must be followed by one parameter which specifies the number of array subscripts, and these must appear in a DIMension statement.

```
COMMON A,B(1),C%,D$,E%(3),F(2)
DIM B(10),E%(2,3,4),F(10)
```

### Creating and using library programs

In CBASIC one has the ability to include programs — such as graphics or screen handling routines which one may have written or acquired. In this sense, CBASIC operates in much the same way as the better versions of PASCAL or FORTRAN. Any library programs may be incorporated by means of the %INCLUDE statement; it is important to make sure that the INCLUDEd program (or program segment) does not include a STOP or END statement — for obvious reasons. Up to 6 levels of %INCLUDE nesting are permitted which should be enough for most people!

### Other statements, functions and directives

CBASIC is full of other goodies to excite the keen programmer or to depress the less able! Some useful ones are listed briefly.

#### *Statements*

CALL — summons a machine code routine, and has the syntax:

CALL <integer expression> — the expression may be in hex

INPUT ... LINE — reads an entire entry from the display and assigns it to a special string variable.

INPUT"Message"; LINE SILLY.ANSWER$

OUT — puts out an 8 bit integer value to an I/O port.

OUT <integer expression,integer expression> sends a low order byte of the second expression to the output port addressed by the low order byte of the first expression.

RANDOMIZE — seeds the random number generator called by the RND function; the seeding depends upon the operator's response time to a request from a previously executed INPUT statement.

```
INPUT"Do you need to use a random
number";A$
RANDOMIZE
```

READ # ... LINE — reads one record from the selected file and assigns it to a string variable.

READ #2; LINE X$ — reads one record from file no. 2 and places it in X$

SAVEMEM — reserves space for a file which will be loaded when SAVEMEM is executed; normally, the file will be a machine code type.

#### *Functions*

CONCHAR% — waits for and accepts one character from the keyboard.

CONSTAT% — returns a -1 (true) if a key has been pressed or 0 (false) if not.

INP — returns a byte from a selected I/O port

INP(port.no.)

#### *Compiler directives*

There are six of these; two, %INCLUDE and %CHAIN have already been covered and the rest deal with the format of the listing produced by the compiler. Directives do not interfere with the running of the program and they must always be at the beginning of the line.

%LIST and %NOLIST — turn the program listing on or off which could be handy if you only want to see part of a program: they affect both console and printer listings and also operate on disk files.

%EJECT — as its name suggests, operates on the printer when it is enabled and performs a control-L function to start printing at the top of the next page.

%PAGE — sets up the number of printed lines per page on the printer. It defaults to 64 and must be altered if another page length is needed. Thus PAGE(60) specifies 60 lines per page of print.

Apart from compiler directives, there are six "toggles" which may be included as required in the command line. The toggles are preceded by a $ sign.

B — suppresses source file listing at the console, but allows compilation statistics to appear.

C — suppresses the creation of an .INT file which helps in initial error checking for a new program since compilation is a lot faster.

D — suppresses the conversion of lower to upper case letters when these are used in variable names.

E — a useful debugging facility which pinpoints source lines in the .INT file in which an error occurred. It must also be used if you wish to employ the TRACE facility.

F — produces a listing of the program on the printer as well as the console.

G — produces a listing on a disk file as well as the console.

A typical CBASIC compilation command line would be:

```
A>CBASIC TESTPROG  first run of the
                        revised version $EF
```

A typical CBASIC compilation command line would be:

```
A>CBASIC TESTPROG  first run of the
                        revised version $EF
```

This compiles TESTPROG, lists it on the printer and enables error checking to take place in the .INT file and providing a message on the screen (the bit in lower case, although it can be in upper case if you want, or even omitted!).

TRACE is a useful facility which can be called upon when a compiled CBASIC program is run using CRUN. If TRACE is called without parameters, the trace of execution of all lines in the program is carried out but the extent of tracing may be altered by including the line numbers as appropriate:

```
CRUN TESTPROG TRACE 1,10 - traces the
          execution of lines 1 to 10
CRUN TESTPROG TRACE 9 - traces the
          execution of all lines
          after and including 9
CRUN TESTPROG TRACE - traces the lot!
```

### Error messages

CBASIC produces (when appropriate) both COMPILER and RUN-TIME error messages which are self explanatory. It also produces two-letter error codes which are expanded upon in a closely printed 6 page appendix in the *User Guide* — and if the E toggle is employed at compilation time, a little more information is produced including an indication of the offending line and the probable position of the error on that line.

In spite of what the User Guide says, the simple error messages are not immediately intelligible. When the expanded error message is produced it says:

```
ERROR SE IN LINE 123 AT POSITION 14
```

OK, SE means syntax error, line 123 is the 123rd line of the compiler listing, not statement label 123 and position 14 means the 14th column on the screen or printout. In fact, the error may have been caused by something in a preceding line or by the omission of something earlier on in the program (such as a variable which has not been previously declared in a DIM statement). It might be on the line which is indicated, or even its predecessor(s), particularly if the \ feature is used to break up a line — MORAL — don't use the \ feature unless you are confident, and for the first few trial compilations, do use the F toggle to get hard copy of your errors.

### Conclusions

CBASIC is not really designed for the inefficient, impulsive or non-structured programmer, but it has some excellent features which render it very useful for commercial or scientific applications. Until Microsoft dropped their absurd royalty claims on published programs written in compiled MBASIC a couple of years ago, it was the preferred BASIC language used in commercial software and the CP/M User Group library. Benchmark tests show that it is somewhat slower in single precision arithmetic than compiled MBASIC (but it works to nearly double the level of precision). It appears to be more accurate in handling numbers with several significant figures, but I have not tested this exhaustively yet; a future article on benchmarks will deal with this aspect and will include comparisons of both BASICs and some other languages (including P*SC*L and F*RTR*N) on 80-BUS machines. Although it is not available from Gemini, several software houses will supply it at round about £120 — about half the price of the MBASIC interpreter. DRI's CB-80 which has a true compiler currently costs about £400.

All things considered, CBASIC would be an excellent choice in terms of cost and facilities for the user who is updating from a cassette-based to a disk-based system for serious applications. Before committing oneself to its purchase, it would be worthwhile getting hold of copies of EBASIC and ERUN to see how the idea of a semi-compiled language appeals to your style of programming. EBASIC is said to be compatible with CBASIC but lacks some of the more advanced features, so EBASIC programs should run satisfactorily on your CBASIC system.

---

# A Brief Guide to MailMerge

## by P.D.Coker

Many readers of 80-BUS News will have heard of or used WordStar — one of a series of very useful programs produced by Micropro. Apart from WordStar, a sophisticated word-processing package, there is SpellStar, a spelling check facility, and MailMerge.

Both MailMerge and SpellStar can only be accessed from within WordStar; the appropriate files (MAILMERG.OVR and SPELSTAR.OVR) which are supplied separately from WordStar, must be available on the logged-in disk [Ed. — or the disk specified in the WordStar patch area, normally A:]. The appropriate program is called up from the 'no file' WordStar menu. SpellStar has a comprehensive range of facilities which are selectable from a menu. Further information is provided in section 13 of the main WordStar manual. Mailmerge is also well-documented — sections 9 – 12 of the same manual deals at length with all aspects of the facility.

For most purposes, only a small proportion of the documentation is needed but the relevant bits are often difficult to locate; this article aims to provide a simple guide to the more commonly used features of MailMerge for the non-expert user.

Richard Beal (80-BUS News vol. 2 issue 3) has dealt with the installation of WordStar on Gemini machines with the IVC (and SVC), and the same installation patches will work for the Map-80 Systems VFC.

MailMerge is, as its name suggests, mainly used for creating and printing personalised letters (the sort which Dave Hunt consigns to the rubbish bin, unopened and unread!) for multiple mailings, but it can also be used for creating letters or documents in which standard clauses feature regularly — which would be useful in Solicitors offices, for example. The ability to personalise a standard letter would be useful for many small firms and clubs, and just the job for aspiring authors to send their manuscripts (printed by WordStar) to several dozen publishers!

Both MailMerge and WordStar use 'dot' commands for processing text. They take the form of a full stop followed by two letters such as **.pa** which inserts a page break into the document (similar to Control L in CP/M or PEN). Some dot commands are followed by a number or by text; many such commands exist in WordStar and MailMerge but normally, only a few are needed and when used, they must occur at the beginning of a line. A few dot commands are specific to either WordStar or MailMerge but many are common to both.

WordStar operates in two modes — Document or Non-document and the mode required is identified by typing D or N in response to the initial display. The Document mode is used for text work and has automatic text justification among other features while the Non-document mode is used for listings and source code programs (and for setting up MailMerge data and command files). For simple mail list work, three files are needed — the standard letter, a data file with names, addresses and other information, and a command file (optional but makes life easier). The command file works in a similar fashion to a SUBMIT file in CP/M and enables the use to control the progress of the letter printing more effectively. It only contains dot commands, unlike the letter file which contains dot commands and text.

The following dot commands are most commonly used in MailMerge:
(The file names or variables enclosed in angle brackets '<' and '>' must be supplied by the user.)

**.av** <variable> (Ask for Variable)

For most purposes, this will prompt for a variable such as the date which may be altered from one mailing to another, and which would not normally be stored in a data file. The text following the .av command must match the field name in the letter file, and it is automatically inserted into the letter during each printing cycle.

**.cs** (Clear Screen)

Self explanatory — removes any previously displayed material from the screen.

**.df** <file name> (Define data file name)

This indicates the name of the data file which will be used during the MailMerge operation. It is placed in the letter file.

**.dm** <message> (Display Message)

This is followed by appropriate text, which will be displayed on the screen. It is normally only used in a MailMerge command file.

**.fi** <file name> (Define file name)

MailMerge needs to know what file is to be merged with the data file: .fi is followed by the name of the letter file and the command is usually placed in a command file.

**.op** (No page numbering)

WordStar defaults to numbering pages which may be unnecessary — particularly for a single page document. .op disables this facility.

**.pa** (Page break/go to next record)

A very useful command which instructs the program to insert a page throw and go on to use the next record in the data file.

**.rv** <list of variable(s)> (Read a variable)

This specifies what data fields are in the data file, and their order.

The standard letter will have data fields inserted into it and bracketed by ampersands (&); by placing a /O at the end of a data field, one can suppress the space which would be left by any field such as a missing post code or county which may have been omitted from the data file.

Three or four dot commands are required for all letters which will be processed by MailMerge. These are .op (optional), .df, followed by .rv and .pa; the first three are placed at the beginning of the letter file and .pa at the end. A typical letter file would be as follows, where the dot commands should **always** start in column 1.

```
.op
.df silly.dta
.rv name,company,street,town,county,
postcode,goods

                              123 High Road,
                              London E99 4YY.
&data&

&name&
&company/O&
&street&
&town&
&county& &postcode&

Dear &name&

Would you kindly send me a copy of your
current catalogue of export model &goods&?

I would also be interested in receiving
details of discounts available for cash.

Yours faithfully,


A.N.Other
```

File this as BEGGING.LET.

Then set up a data file with the names and addresses of all the firms you wish to bother. Use the **NON-DOCUMENT** mode of WordStar for this. The data must be present in the same order as it is required by the letter file and if any information such as the name of the street, or the county is missing, its place in the data file must be marked with a comma. Each name and address must take up one line of the file and is terminated with a carriage return — the record may be well over 200 characters long. The following example will show how it is done:

```
Top Cat,Tiger Computers,Zoo Road,Panthertown,Lynx,
ZZ1 1ZZ,mice,pl86
Deputy Dawg,Newbone Computer Store,,Wuff,
Avon,,floppy disks,pl86
Chris Tandoori,Beebac,,Oxbridge,,OX99 BBC,
plastic computers,pl86
```

Store it as SILLY.DTA

Note that record 1 is complete, record 2 has no street name, or postcode, hence the double commas after Store and Avon, and record 3 has no street name or county, hence two lots of double commas.

The command file comes next.

In Non-document mode type the following, noting that the dot commands must start in column 1.

```
.cs
  .dm Printing your letters
  .av Date
  .fi BEGGING.LET
  .dm Finished (Thank goodness!) - or
whatever you want to say.
```

File this as DAFT.CMD

You now have three files, BEGGING.LET, SILLY.DTA and DAFT.CMD — and you are ready to try out the all singing, dancing and generally frolicsome MAILMERGE!

Connect the printer, and select M from the WordStar menu. The program will prompt you for the name of the file to merge/print. In the example given above, this is DAFT.CMD. You then have to give some simple answers to profound questions about the output. If you want to, you can save the results of your labours in another file; MailMerge prompts you for the name. Otherwise, the default settings will be quite satisfactory for a trial run. Don't make the mistake of specifying more than 1 copy, since this means that MailMerge will go on churning out as many **sets** of letters as you have asked for **copies**!

Having established a reasonably simple format for the use of MailMerge, I have found it quite easy to use, although a minor bug sometimes manifests itself. If one creates new data, letter or command files using WordStar and then uses MailMerge without exiting to the system, and re-entering WordStar, the whole system may lock up, and has to be reset.

# SHOCK HORROR HEADLINE

## by David Hunt

In this set of DRH ramblings he looks at a utility called LU, a deadly little program called VIRUS, the new *Gemini* GM870 80-BUS MODEM board and its software, the new UKM720 program, and *Gemini's* latest BIOS, version 3.2.

### DH disk storage crisis

Having sat down to write this sessions' pile of disjointed bits, I'd done about a page's worth and for some reason decided to save it.

```
ERROR Destination disk full
```

Well perhaps it's not a crisis, I didn't lose anything and I've plenty of secondhand disks knocking about, but I had a look at the full disk anyway, sure enough 784K with 4K free, and not a .BAK file in sight. Oh well, start a new disk; format it and label it. The label, -80BUS4 ??? Did that mean I'd completely filled three disks? It sure did! Almost 2.4M bytes worth, and that's exclusively this mag and its predecessors, and then only what I've bothered to keep. Just as an exercise — a word count; I put everything into one big library using LU on the winnie, and then set SPELGARD the problem of counting the words whilst I went for a bite [Ed. — byte ?] to eat. Well of course, SPELGARD gave up at 65000 odd words, not surprising really, and I learned for the first time that SPELGARD has a 16 bit word counter. Well by now I really did want to know how many words had been written, so I truncated an average file to exactly 8K using a debugger and fed that to SPELGARD, then multiplied the result by 295. The result, I know you're all dying to know – just over 300,000 words. Was it really that much? Well a re-count on another file came out about the same, so I guess it must be something like that. Just as well 80-BUS doesn't pay me at a commercial word rate, I'd be quite rich and 80-BUS would be broke.

I mentioned LU; well this program has been lurking around the CP/M libraries for some while now, but I only came across it a few months ago. It's marvelous for archive stuff, you know, the sort of thing you are loath to erase, but at the same time don't ever intend to look at again whilst begrudging the disk space it takes up. In fact your typical DH 80-BUS article. No in fact DH type articles are too long and rambling to make much advantage of stuffing them into a library, the only advantage being in this case, keeping bits of things together. It's much more useful at archiving lots of little things, letters, dBASE .CMD files and the like.

As you know, CP/M allocates space to files in blocks. In the good old days of single density 8" disks with only about 240K of usable capacity, life was easy. CP/M builds a map of where things are put on the disk, it's part of the directory entry for the file, and saved along with the name. That's what file control blocks (FCBs) are all about, but then if you know about file control blocks, you know about directory allocation blocks. Anyway, the disk is carved up into nice small chunks, and allocated an 8 bit number. Now there can only be 256 discrete 8 bit numbers, so the maximum number of blocks is 256. If the disk has a capacity of 240K, 240 1K blocks can be allocated. But like Topsy, disks growed, and double sided double density 35 track disk formats came along. These had a capacity of about 340K, and you can't have 340 1K blocks, so the block size was doubled, 170 2K blocks. Then someone increased the number of tracks available by halving the track pitch, and 80 track disks appeared with a capacity of 788K. So now we have 197 4K blocks. Yes I know I've dodged the issue of single sided 80 track disks and the 'double byte' block allocation numbers used in winnies, but the fact remains that on the *Gemini* high capacity disks systems, we have 197 4K blocks.

The snag, you ask? Well, it doesn't matter if a program is two bytes long, it's still allocated 4K of disk space, and so appears as a 4K file. More typically, letters and dBASE .CMD files are usually about 1K long, so 3K goes wasted. LU gets round this rather neatly. It constructs one file called a library file from a number of files, adding them end to end, and then puts an internal directory into the file. Yes, LU also has an internal block size, but it's only 128 bytes, so not much space is lost.

Of course LU must have snags, the biggest is that once a file is in the library you can't execute it or even look at it, it has to be extracted from the library before you can do anything with it. Having said it's for archive stuff anyway, this isn't too much of a limitation. More annoying perhaps is it's quirky syntax, which is also rather unhelpful and unforgiving.

Having said that you can't do anything with a file in the library without extracting it, LU came with a couple of utilities, LDIR, so you quickly look at the file names in the internal directory, and LRUN so you can execute a xxx.COM file from within the library. Unfortunately if a program is executed from the library using LRUN, the program still can't access any overlays it might want, they're still buried in the library. The one utility that should have been there but wasn't was LTYPE, as much of my library stuff is ASCII files. Still perhaps there wasn't one or it had got lost or something. Or worse still, someone out there thinks DH should write one — no way!

I made a major mistake when I first got LU. I played with it a bit and thought it would come in useful

later, so my first library file was made up of the bits of LU, including the documentation. No I didn't then erase LU from the system disk, it stayed there until I wanted to use it, then the problem. The documentation was in the library, and I couldn't remember the syntax to make it work. After about half an hour trying all the tricks I could think of that wouldn't take too long, (including trying to load the library into a debugger but it was too big) I resorted to phoneing Richard who promptly reminded me that all commands are prefixed with a '-'. That solved the problem and away I went.

The documentation for LU is 'something else'. It's 32K long, and if you reckon I go on a bit, you should try reading the LU documentation! I think the command summary should take up about half a page, after all there are only 9 simple commands. And perhaps another page of warnings, tricks, etc. That's about 4K at best. I can't say I've read all the guff that came with LU, so perhaps I've missed the one vital command which gives you a list of commands from inside the program, but I doubt if there is one.

To give an idea of space saving with LU, take my dBASE master disks, there's five in all and tot up to about 900K (allocated on 4K blocks). Squeeze all that lot using SQ or SWEEP brings it down to about 700K. Then shove the squeezed file into a library, the library ended up at about 330K. In fact I got the whole of dBASE 2.41 and dBASE 2.43 on to the same disk with room to spare. So the space saving with LU was quite fantastic.

The other problem LU cures is getting things mixed up. Lots of different suites of programs have a program call INSTALL (or other similarly named files). Try running the wrong install on a program, and brother, have you got problems. Previously I kept different suites of programs on different disk user areas, but this still wasted space. With LU, all the different parts of a suite of programs are all tucked away in one file and can't get lost or mixed up.

### VIRUS

Back to the wasted space at the end of a block in a CP/M directory, someone came up with a real lulu of an evil idea. I don't know how far he went with the idea, for all I know, it might be out there lurking, but it should have been (or will be) a real block buster if it's ever done.

The program was to be called VIRUS, and like a virus, it was to be self propagating and infect the world silently until the symptoms struck an unsuspecting world computer population almost at once. The idea goes something like this:

VIRUS is introduced into an executable file by the perpetrator of the crime, and that file is given to someone else. The next time that program is executed, the program loads into the TPA as usual, but before the program executes, VIRUS executes first, because the start jump of the program has been changed to VIRUS instead of the program. VIRUS throws a random number and goes and gets a directory entry from the unsuspecting persons disk. VIRUS then checks four things, first that the directory entry is a .COM file, secondly it has a start jump (most CP/M programs have, and if it has VIRUS saves it), third, that there is space at the end of the last block for VIRUS to lurk (could be difficult that, but a clever programmer could probably crack it), and fourth, VIRUS is not already resident in that space.

Given that all four conditions are satisfied, VIRUS copies itself into the wasted space at the end of the last block using the CP/M 2.2 random access ability, changes the start jump to itself and changes its exit jump to the jump at the start of the program so the program will execute after VIRUS. One last thing. In copying itself, it decrements an internal counter by one.

Get the idea, VIRUS has now infected another program. Next time either that program or the previous program is executed, VIRUS will infect another program and so on. The infection process would only take a few tenths of a second, and the user would be totally unaware of what was happening. So the guy with infected disks gives a copy of some program to another person, who also catches VIRUS, who in turn passes it to someone else. All the time the counters are decrementing.

When the counters get to zero, the program refuses to run and up pops a message:

```
YOUR PROGRAM HAS VIRUS.  SEND   £5.00 to
P.O. BOX 1 BRIGHTON FOR DECONTAMINATION PROGRAM!
```

If the counters were set to a fairly large number, and VIRUS were introduced into the right places, like NASTUG and the Merseyside Users Group, the natural process of swapping disks between folks could go on for some considerable time before the symptoms break. Exit our hero with a small fortune and a lot of angry guys on his tail. Angry guys? Yes probably, as VIRUS would have infected proprietory programs as well as rip-offs. And as for chasing him, well I'll be one jump ahead of the pack, 'cos I know whose idea it was in the first place. Nice thought though.

### The Gemini GM870 MODEM

They've arrived at last. The *Gemini* GM870 MODEM cards that is. What a delight, both for me and *BT*. Me because I'd been waiting for one for some time, and *BT* because my usually frugal phone bill has been taking a walloping.

So what do you get? An all 'bells and whistles' auto-dial, auto-answer MODEM all on one card

with enough software to suit all but the most fastidious user. The board is the usual high quality 8" x 8" card, with a fair sprinkling of big chips and some miniature relays as well. This all comes with the documentation, a lead fitted with the 'new-type' *BT* rectangular plugs to connect it to the phone and a disk with enough 'comms' software to keep most users quiet for a long time. In fact everything you need to plug it in and go.

**The Line Side**

Close examination of the board reveals good adherence to the *BT* ideas of line isolation using both a line transformer (to the more rigid *BT* 5KV spec.) for line coupling and an opto-isolator for ringing detection. The whole area of the board associated with the phone line part is ringed by a heavy earth track, and a plate through M3 hole is provided for the isolating earth. Tracking is provided for spark gaps, but these are missing, the manual emphasizing that the system is for connection as a secondary device within the *BT* 'new-type' six pole plug and socket scheme. If the MODEM is connected as a secondary, then it is protected by the spark gaps in the primary socket, but with proliferation of diy telephone wiring, there is always the danger of someone doing a short cut job and not using a primary socket for the phone input. This danger is also present if the MODEM were connected direct to the older type *BT* wiring and the original phone removed. Three fuses are provided, one each on the A and B pair, and a third on the bell suppress line. I've never seen one there before, but useful none-the-less.

Having said that the board appears to adhere to *BT* specs. with regard to the line side of the board, why the big red triangle saying 'Not approved for ... etc.'?

Well here lies an interesting story, typical of the muddle headed thinking which tends to go with big bureaucracy. A few years ago, with the idea that *BT* was to go public, HMG considered the business of *BT* approval. *BT* could hardly carry out the approvals itself as this could smack of monopoly, and the only one allowed to run monopolies is HMG. So the approval was farmed off to an independent body, *BABT*, the *British Approvals Board for Telecommunications*, along with draft specs. from *BT*. It appears they also had a brief to tighten the specs. a bit and were to be allowed to charge for approval (expenses you know). So *BABT* beavered away, and did what it was told and tightened the specs.

Now what follows is hear-say, but sounds rather likely. When *BT* heard what *BABT* had done, and this was before *BABT* got hold of the approval monopoly, *BT* hastily rushed through a few projects of their own. Could this batch of approvals have included the 'new-type' telephone socket, as by no means would that socket get *BABT* approval now. For starters, you can shove the British Standard finger into it, and the ringing voltage is 150 volts!! Oh dear, oh dear.

Another bit of spec. tightening involved the situation of MODEM cards. They don't actually approve a card any more, they approve the card and the system it will live in. So we have the very odd situation where the *Gemini* GM870 meets the spec., so the card will actually get (or by the time you see this, maybe have got) approval; BUT, only when fitted in a specified *Gemini* machine by *Gemini* themselves. Under any other circumstances the card is not approved, and unless you submit your own system, including the card, and fork out quite a bit of loot, the card will never have approval in your system. Even more wierd, if *Gemini* made the card stand-alone, with its own PSU and serial interface, and put the lot in a pretty box, then it would likely gain approval, despite the fact that you might have the live side of the mains connected to system ground! So stand-alone MODEMs get approval. Built-in MODEM cards don't.

So what's the point, why sell an unapproved MODEM? None really, it's not illegal to sell them, and it's up to the conscience of the purchaser who buys and uses it. The red triangle notice only says '... action MAY be taken ... ' and *BT* are fully aware of the situation regarding the use of non-approved phones in this country. If someone were mad enough to try and make a stand on the matter, the courts would be tied up from here to kingdom come with petty flouting of *BT* regulations, and no-one in authority likes to see the law made a fool of, least of all *BT*. So provided the card does everything it needs to to ensure it's safety, and provided you have done everything to ensure that your system is electrically safe, and provided the MODEM doesn't draw attention to itself by acting in an unapproved fashion (which it won't), then the chances of being done for using it are vanishingly small. Not that I would encourage anyone to flout *BT* regulations, it's up to you.

To finish off the coupling arrangements the line transformer is coupled to the AMD 7910 'World Series' MODEM chip via a terminating pad for output, and via a 741 op-amp with carefully controlled gain to reduce the input on V.21 operation, thereby improving the integrity of the signal. No in-line filtering is incorporated, as the AMD 7910 includes its own line filtering, although I have a sneeking suspicion that the inclusion of a bandpass filter may improve the performance under marginal conditions. That said, none of the other MODEMs in the same price bracket using the AMD chip include bandpass filtering, nor the gain control, so the performance should be at least as good as the competition, and possibly better on V.21 due to the gain controlled amplifier. The

transformer is sampled via a high impedance pad and fed to an LM386, which is a small audio amplifier. An 8R speaker output is thus provided to monitor the line. Useful if you keep dialing engaged numbers or keep finding noisy lines. A software on/off switch is provided for the audio output, and the audio level is adjustable from zero to full volume using a preset pot. Maximum audio output was not measured, but was certainly loud enough!

**The Computer Hardware**

So on to the system side of the hardware, the bit the computer side of the coupling transformer and isolator. It's an 8" x 8" bus card like all the rest, and the height of the coupling transformer and line inductor are such that the card will fit in to a normal Vero frame or the reduced pitch (0.95") *Gemini* frame used in the Galaxy series computers. The major interface devices are a CTC, a PIO and a DART. The CTC and the PIO were familiar, the DART was a new one on me, and unfortunately little detail was provided about any of these devices in the manual. Not a problem as we shall see, but unusual, as *Gemini* manuals tend to go into detail describing unfamiliar devices. The usual I/O buffers were provided, handing the usual bus signals and providing NASIO. More typical of present technology, I/O decoding was provided by a PAL rather than discrete logic, with no options for alternative addressing. The board uses 12 ports, from 80h to 8bh, which overlaps the Belectra Arithmetic card. This I understand is due to a *Gemini* 'internal communications failure' as the decoding was intended to be 84h to 8fh. Not that it matters as the Belectra card may be re-addressed and the dedicated software for the Belectra card may also be re-addressed.

Three of the four counters of the CTC are used for internal clock signals, two for the DART Rx and Tx clocks, and one to control the dialling. The PIO device does most of the card control, and all 16 bits are used. Port A produces a 3 x 4 matrix which is directed to the dialling decoded (effectively it pretends it's a numeric key pad of a telephone). Port B pulls in the relays as required and selects the MODEM chip mode. The DART I don't know a lot about, except that it is a dual UART of some kind. Only one side is used, the same register decode is used for status in either direction and another port for data either direction.

Dialling is quite sneaky, it uses a *Mostek* telephone dialling chip, which is normally used with 'touch-dialling' phones. Give it a matrix of numbers and it just gets on with it. Board tracking is included for tone dialling to CCITT standards using a second *Mostek* chip, but this chip and crystal are not provided. This means the MODEM may be equipped for use with UK tone dialling PABX switch boards as well as the normal direct line connect.

The guts of the matter revolves around the MODEM chip, I haven't got the full words for this device (I borrowed and read the words some time ago) and information in the MODEM manual is sparse (like the CTC, PIO and DART). This chip is the all laughing dancing singing thingy capable of CCITT and BELL protocols. This is all controlled by five address lines, the first two being connected to the PIO, the remainder being returned to patch plugs which are sealed with paint. Now the words say that if you remove and change the links, the card will be forced to work outside *BT* regulations, which translated means it will start working in the BELL mode. What the manual does not say is what exactly happens when you do swop the plugs about. So for that you need the AMD manual. What's on offer with the standard setup as supplied:

| | | |
|---|---|---|
| V.21 300/300 | originate | full duplex |
| V.21 300/300 | answer | full duplex |
| V.23 75/1200 | originate | asymmetrical duplex |
| V.23 1200/75 | answer | asymmetrical duplex |

This is standard for most things found in the UK, and the likelihood of BELL protocols being encountered is remote.

On test a strange anomoly has been observed which must be something to do with the hardware. I wonder if it has occurred elsewhere? I haven't looked into it yet, but it goes as follows:

Conditions:

Computer mains off but not unplugged from mains
MODEM connected to line (plug in socket)
Video recorder on play back in another room
Incoming call, phone ringing

Symptoms:

Ringing 'burr' heard quite loudly from the TV set when an incoming call occurs??? Now, as I say I haven't looked for it, I just unplug the MODEM when not in use, but I haven't the foggiest idea what causes it!!

**The Manual**

A pretty comprehensive manual is supplied which runs to 52 pages, and covers all the major aspects of the card and its software. Details are missing about the I/O devices used, referring you to the respective device manuals. Instead of blow by blow accounts of the devices used, source listings of the setup and I/O procedures are provided. As can be imagined, with a PIO, a CTC and DART to set up, the routines are complicated and tedious. With the source listings provided, they can all be seen and analysed. These are much more useful than accounts of the workings of the chips. The

hardware description is short but with enough detail to allow the user to cope. The majority of the manual very adequately covers the software provided. One day *Gemini* will get round to employing someone who can wield a drawing pen, as once again, any illustrations in the manual have been composed using the printer and look tatty.

**The software**

There are three parts to the software. A demo with its source file for auto-answering, not very useful, but invaluable for those few who will go on to write proper answering software. The two main parts are GEMTERM a 'comms' program written for the card, and a suite of programs modified around Richard Beal's DIAL program.

GEMTERM is a complete entity, being a terminal emulator with the ability to send and receive ASCII (not binary) files. All parameters needed by the MODEM and program can be set manually or from a library of parameters, items such as baud rate, the mode, the number to ring (if any), whether the loudspeaker should be on or off, the number of data bits to be used and the parity (if any). The first entry in a line of library data is a name, so GEMTERM can enter and dial the service required direct with no intervention by the user. Other parameters available from the command menu include memory save of incoming data, XON/XOFF protocol select, printer echo, number of nulls to use after line feed, disconnect line, and many others. This is stand-alone software dedicated to the GM870 and it makes good use of all the features of the card, particularly the ability to select a service, set up the MODEM, dial the service and then let you get on with it. It works well, but being new and unfamiliar to me I had a problem remembering the commands at first. I particularly liked the library facility which saves all the hassle of remembering the MODEM parameters for a particular service and the telephone number. The snags, it can't handle the 1200/75 *PRESTEL* service properly or the Viewdata compatible bulletin boards (these require interpretation of graphics symbols), and it will only send or receive ASCII files.

DIAL has been mentioned before. A similar idea to GEMTERM as far as the library goes, but otherwise, completely different. What DIAL does is to fetch an appropriate 'comms' program for the service required, set up the MODEM for that service, dial the number and wait for reply, and then jump straight into the 'comms' program. On exit, DIAL disconnects the line. DIAL may be patched for a number of different MODEMs, not only the GM870 and may be purchased as a separate entity if required.

DIAL always expects a second command parameter, if you don't give one, it will prompt for one, so the command:

```
>DIAL TARGET
```

would execute DIAL which goes to the library, looks up TARGET, then fetches UKM7, the appropriate 'comms' software, gets the MODEM parameters and sets them, dials the number, waits for a reply, and when a reply is received, jumps into UKM7. There are no terminal controls from DIAL itself, UKM7 is the terminal program and that does the business. This means the DIAL can be used with appropriate software for the service to be called, and two 'comms' programs are provided. Snags? None encountered but would be nice if it sent service ID's instead of leaving this to the 'comms' program. But that perhaps would complicate matters somewhat. Another thought for DIAL would be an internal password, as it's so easy to use, I caught my kids trying to raise the Titanic at 2p a frame on *PRESTEL*.

UKM7 is a public domain program and is a UK version of the original all bells and whistles American MODEM7. MODEM7 was very system dependent, and not really appropriate to UK use so David Back put it through a mincer and came up with UKM7, a version entirely suited to UK use. Richard Beal has removed the UART initialisation routines as the initialisation is carried out by DIAL, although the full source and documentation is supplied so UKM7 could be put back to its original state with no difficulty. This is a very powerful program, not only a terminal emulator, but it incorporates down load and up load of files (singly or in batches) in ASCII or binary mode, using both old and new versions of XMODEM binary self checking protocol. It also has the ability to trap all I/O and keep this on a separate file, so that mistakes or stuff arriving too fast to be read at the time can be analysed later. All clever stuff, and improving all the time (see below). Snags? Touchy about syntax and also seems to get it's knickers in a twist when up-loading and down-loading in the wrong mode. It you're going to use this one, take David Back's name out of the sign-on before use. (The manual doesn't mention this.)

The last program supplied is TERMB, a little terminal emulator with buffered I/O written up in 80BUS some time ago. Very fast and simple. Dead easy to use with no vices.

Not supplied on the disk is PRETZEL2, Dave Ryder's *PRESTEL* terminal emulation program available from Henry's. The new version 3.0 is fully compatible with DIAL and may be directly used with the GM870. Works well.

**In use**

Couldn't be simpler, I plugged it straight in and it all worked. Well I haven't yet tried the auto-answer program. People tend to phone me, not the computer. Other than that, what can I say, it all worked!! All the software works and GEMTERM and DIAL libraries are easily set up. PRETZEL2 worked, but two versions were required for use

with DIAL, one for *PRESTEL* itself with the 14 character ID, and one without the ID for other Viewdata services. (I don't want to give other people my sign-on and passwords.) Never mind, what's on offer is one of the most effective and complete 'comms' packages I've seen for any machine.

**UKM720**

Always on the lookout for something new, and browsing through the down-load section of the *CBBS* London West Bulletin Board, I happened across a revised version of UKM7 call UKM720. Again written by David Back, this one is two generations younger than UKM7, and is put together rather differently.

Instead of one huge source file, UKM720 has an uninitialised .COM file, and an installation program for a number of computers ranging from *Gemini* GM811/813 cards through *Tandy* TRS80 to *Epson* portables. The source file of the installation data file is also provided so you can add routines for any that aren't in the list, and of course the GM870 is too new for the list. Actually, there's a small problem here, the install program offers only 10 options, one of which is *Gemini*. If you want to install something else, then you must delete one from the installation data source and substitute the new. I didn't want to do this, as everything there was useful to someone, and anyway, it looks greedy to have two *Gemini* options. I opted to make the two *Gemini* options conditional, enclosed in an 'IF - ELSE' assembler directive, and you reassemble for either option. The modified source for the GM870 bit is published here, but remember, if you do down load this from CBBSLW, then the source will also require an equate somewhere to indicate which MODEM source to assemble. Sorry about the 8080 mnemonics, I did shove the thing through TRANSLAT to convert it to Z80, but it wouldn't assemble, so rather than spend half an evening mucking about to find out why, I simply added my bit in 8080 mnemonics.

This proves the advantage of having the sources in the GM870 manual, as it was a straight copy job to include it. The bit tests are done differently to the manual, but there were 10 other examples in the source showing how to do it, so no problems.

Since then I've had trouble with a couple of the more local bulletin boards which are supposed to auto select 1200/75 or 300 baud, they just don't seem to work. My first suspicion was the software, but that was exonerated by trying different software, which pointed to the MODEM card, but as similar results happen with a WS2000 MODEM, I guess the bulletin boards are at fault. *DISTEL* does say (at 300 baud) that the auto select is experimental, so there you go! [Ed. — I have used the *Gemini* GM870 MODEM on an 'auto-select' bulletin board, and am pleased to say that it worked at either rate. Unfortunately I can't remember which board!]

Enough for this session about MODEMs, they're fun but tough on the pocket. Unlike yacking on the phone, you don't tend to be aware that the minutes are ticking away. Now there's another idea, reprogram the *Gemini* SVC/IVC to tick away in money rather than minutes at the top of the screen, it could pick up the charges for the phone call from DIAL. That would show just how bad on the pocket it really is.

**Another New Goodie — Gemini BIOS V3.2**

Another new goodie has appeared from *Gemini*, and not too tough on the wallet, all things considered. BIOS 3.2. Those who remember SYS will remember one of its features was the ability to run with a selection of drives. You know, *Gemini* DDDS on drive A and RML on drive B. Or, substituting *Teac* TD55-F (double sided 80 track) for the standard *Gemini* arrangement of the time of Micropolis 1015-V (single sided 80 track) drives. A lot of SYS'es saw use for just this purpose, limited though the feature was. BIOS 3.2, which is only eighteen months late, supercedes and exceeds SYS for this purpose.

Over a period of time, *Gemini* have gone through four disk formats, GEMSDSS, GEMDDDS, GEMQDSS and GEMQDDS; at the same time less significant features of the BIOSes has been improved, changed, added or just done away with. One such improvement is again associated with drives. Early *Gemini* BIOSes know nothing about RAM-disks, later versions you hand patch to turn them on, and the latest finds out for itself what's there and turns it on for you. Likewise, if you upgrade your drives, to a different format, the BIOS has to be changed to suit. *Gemini* would always do this for you, but the number of permutations is getting out of hand. All arguments for a different way of doing things, hence BIOS 3.2.

BIOS 3.2 consists of a full feature BIOS, (one or more of several versions being supplied depending on your current CP/M or system, one called BIOSF.SYS (for the floppy drive only version), a BIOSFW.SYS (for a floppy as the boot drive plus Winchester drivers, BIOSW.SYS (for the winnie and floppy version), and BIOSN.SYS for MultiNet systems), a system configuration file, a serialized CP/M core with the *Gemini* version of CCPZ, and a CP/M constructor called GENSYS. The system

configuration file is edited with a text editor to suit the drive configuration required, and the whole lot put together with GENSYS which goes to the BIOS file and pulls out the bits it wants. GENSYS then gives you the option of putting the results on to a system track or saving it as a file for use with SYSGEN later. The attraction from *Gemini's* point of view is that when significant changes have been made to the BIOS or new permutations of drive and format take effect, all they have to do is issue new versions of BIOSx.SYS and everyone can reap the benefit without all the hassle of having to send disks back to *Gemini* for upgrade.

So what does BIOS 3.2 offer? Apart from all the drive permutation twiddles, basically all the normal *Gemini* features, screen editing, screen dump, etc. Also the usual I/O byte support for serial and parallel printers, serial transmission protocols and all the rest. Nothing radically new, just the latest versions with extensions to the patch area so that all three different IVC/SVC cursor types may be defined, and screen edit mode may now be disabled.

The fun really starts around the drives. If BIOSFW or BIOSW for winnies is included, then the winnie can be partitioned as required, for instance you could make five 1M drives out of 5M winnie (I can't think why you should want to, but you can) or, more usefully, for example an 8M and a 2M out of a 10M winnie (8M being the maximum CP/M 2.2 can address).

The permutations of drives can be fun, one problem with SYS was that you couldn't easily mix drive types beyond 5" 48 tpi with 8", or 5" 96 tpi with 8". So mixtures of 48 tpi and 96 tpi were out as you couldn't make SYS double step 96 tpi drives to read 48 tpi formats. With BIOS 3.2 you can. The menu is impressive for home use, yet not powerful enough to make any competition to *Gemini's* own MFB series of machines. All the past and present *Gemini* formats are supported, naturally, that's both 96 and 48 tpi double density and the original double sided single sided GM805 format. Additionally, *Superbrain* 48 tpi single sided, *IBM* PC 48 tpi single and double sided 8 sector formats, *DEC* Rainbow, both *Nascom* formats and dear old fashioned 8" single sided single density. What's more you can assign more than one logical drive assignment to one physical drive. So for a person like me who gets a lot of disks in various *Gemini* formats, the following works a treat.

```
Logical Drive A:  Gemini 5M winnie
Logical Drive B:  Gemini QDDS   Physical drive 0
Logical Drive C:  Gemini QDDS   Physical drive 1
Logical Drive D:  Gemini DDDS   Physical drive 1
Logical Drive E:  Gemini SDDS   Physical drive 1
```

By sticking disks in either drive 0 or drive 1 as appropriate, I can copy anything that comes my way to either the other drive or the winnie. Similarly, if people expect things returned, I can copy stuff back to their original disks with no problem. The only problem comes when formatting disks, BIOS 3.2 does not supply a format program which will pick up the disk dph's and format to the standard in current use by the drive. Pity that, I'd have loved to have sent this disk to Paul in say *DEC* Rainbow format and then see how long it would take him to sus what I'd done. Never mind, I'll invent a wierd format of my own using ALLDISC, that should fox him completely and give *Gemini's* new MFB2 something to get upset over.

BIOS 3.2 is even more intelligent than it already appears. Lets take an instance where you are using a 96 tpi machine to construct a system track for a 48 tpi machine. It wouldn't be a lot of good if the result sent to disk was 96 tpi, as the 48 tpi drives couldn't read it. No if the BIOS constructed is a 48 tpi BIOS, then GENSYS will write the result to either a 48 tpi drive or double step (still equalling 48 tpi) to a 96 tpi drive, the boot sector being correctly set up for the 48 tpi format. This brings me to the second big feature of BIOS 3.2. Each physical drive can be specified separately. One drive could be an 8" for instance, another, a double sided 96 tpi with 3ms head stepping and no head load delays, another, a 96 tpi single sided with 20mS head stepping and 50mS head load delay and spectacularly long head settle times, whilst yet another drive could be a double sided 48 tpi drive with 10mS head stepping and no head delays. A maximum of four physical drive may be specified by type 8" or 5", the number of tracks per inch, the number of tracks. the number of sides, the step, head settle and load times.

BIOS 3.2 expects you to be logical about your logical to physical drive assignments, it no good specifying 96 tpi formats to a 48 tpi drive for instance, on the other hand it a 48 tpi format is specified to a 96 tpi drive, then the drive is forced to double step to read and write like a 48 tpi.

Reading 48 tpi disks with a 96 tpi drive works well, writing needs care, as the disk should be wiped clean with a magnet and reformatted using the 96 tpi drive before data is written, using the 96 tpi drive (hence the need for the format program). This is because the heads of a 48 tpi drive are wider than those of a 96 tpi drive and therefore track a wider area. If a disk previously written using a 48 tpi drive is rewritten using a 96 tpi drive, there will be areas of the original 48 tpi recording left between the tracks recorded by the double stepping 96 tpi drive. When the disk is placed in the 48 tpi drive, the head will track both the intended track recorded by the 96 tpi drive, and also the residual crud left from the original 48 tpi recording. This will degrade the signal to noise ratio, often to the degree that the disk can not be read.

with DIAL, one for *PRESTEL* itself with the 14 character ID, and one without the ID for other Viewdata services. (I don't want to give other people my sign-on and passwords.) Never mind, what's on offer is one of the most effective and complete 'comms' packages I've seen for any machine.

**UKM720**

Always on the lookout for something new, and browsing through the down-load section of the *CBBS* London West Bulletin Board, I happened across a revised version of UKM7 call UKM720. Again written by David Back, this one is two generations younger than UKM7, and is put together rather differently.

Instead of one huge source file, UKM720 has an uninitialised .COM file, and an installation program for a number of computers ranging from *Gemini* GM811/813 cards through *Tandy* TRS80 to *Epson* portables. The source file of the installation data file is also provided so you can add routines for any that aren't in the list, and of course the GM870 is too new for the list. Actually, there's a small problem here, the install program offers only 10 options, one of which is *Gemini*. If you want to install something else, then you must delete one from the installation data source and substitute the new. I didn't want to do this, as everything there was useful to someone, and anyway, it looks greedy to have two *Gemini* options. I opted to make the two *Gemini* options conditional, enclosed in an 'IF - ELSE' assembler directive, and you reassemble for either option. The modified source for the GM870 bit is published here, but remember, if you do down load this from CBBSLW, then the source will also require an equate somewhere to indicate which MODEM source to assemble. Sorry about the 8080 mnemonics, I did shove the thing through TRANSLAT to convert it to Z80, but it wouldn't assemble, so rather than spend half an evening mucking about to find out why, I simply added my bit in 8080 mnemonics.

This proves the advantage of having the sources in the GM870 manual, as it was a straight copy job to include it. The bit tests are done differently to the manual, but there were 10 other examples in the source showing how to do it, so no problems.

Since then I've had trouble with a couple of the more local bulletin boards which are supposed to auto select 1200/75 or 300 baud, they just don't seem to work. My first suspicion was the software, but that was exonerated by trying different software, which pointed to the MODEM card, but as similar results happen with a WS2000 MODEM, I guess the bulletin boards are at fault. *DISTEL* does say (at 300 baud) that the auto select is experimental, so there you go! [Ed. — I have used the *Gemini* GM870 MODEM on an 'auto-select' bulletin board, and am pleased to say that it worked at either rate. Unfortunately I can't remember which board!]

Enough for this session about MODEMs, they're fun but tough on the pocket. Unlike yacking on the phone, you don't tend to be aware that the minutes are ticking away. Now there's another idea, reprogram the *Gemini* SVC/IVC to tick away in money rather than minutes at the top of the screen, it could pick up the charges for the phone call from DIAL. That would show just how bad on the pocket it really is.

**Another New Goodie — Gemini BIOS V3.2**

Another new goodie has appeared from *Gemini*, and not too tough on the wallet, all things considered. BIOS 3.2. Those who remember SYS will remember one of its features was the ability to run with a selection of drives. You know, *Gemini* DDDS on drive A and RML on drive B. Or, substituting *Teac* TD55-F (double sided 80 track) for the standard *Gemini* arrangement of the time of Micropolis 1015-V (single sided 80 track) drives. A lot of SYS'es saw use for just this purpose, limited though the feature was. BIOS 3.2, which is only eighteen months late, supercedes and exceeds SYS for this purpose.

Over a period of time, *Gemini* have gone through four disk formats, GEMSDSS, GEMDDDS, GEMQDSS and GEMQDDS; at the same time less significant features of the BIOSes has been improved, changed, added or just done away with. One such improvement is again associated with drives. Early *Gemini* BIOSes know nothing about RAM-disks, later versions you hand patch to turn them on, and the latest finds out for itself what's there and turns it on for you. Likewise, if you upgrade your drives, to a different format, the BIOS has to be changed to suit. *Gemini* would always do this for you, but the number of permutations is getting out of hand. All arguments for a different way of doing things, hence BIOS 3.2.

BIOS 3.2 consists of a full feature BIOS, (one or more of several versions being supplied depending on your current CP/M or system, one called BIOSF.SYS (for the floppy drive only version), a BIOSFW.SYS (for a floppy as the boot drive plus Winchester drivers, BIOSW.SYS (for the winnie and floppy version), and BIOSN.SYS for MultiNet systems), a system configuration file, a serialized CP/M core with the *Gemini* version of CCPZ, and a CP/M constructor called GENSYS. The system

configuration file is edited with a text editor to suit the drive configuration required, and the whole lot put together with GENSYS which goes to the BIOS file and pulls out the bits it wants. GENSYS then gives you the option of putting the results on to a system track or saving it as a file for use with SYSGEN later. The attraction from *Gemini's* point of view is that when significant changes have been made to the BIOS or new permutations of drive and format take effect, all they have to do is issue new versions of BIOSx.SYS and everyone can reap the benefit without all the hassle of having to send disks back to *Gemini* for upgrade.

So what does BIOS 3.2 offer? Apart from all the drive permutation twiddles, basically all the normal *Gemini* features, screen editing, screen dump, etc. Also the usual I/O byte support for serial and parallel printers, serial transmission protocols and all the rest. Nothing radically new, just the latest versions with extensions to the patch area so that all three different IVC/SVC cursor types may be defined, and screen edit mode may now be disabled.

The fun really starts around the drives. If BIOSFW or BIOSW for winnies is included, then the winnie can be partitioned as required, for instance you could make five 1M drives out of 5M winnie (I can't think why you should want to, but you can) or, more usefully, for example an 8M and a 2M out of a 10M winnie (8M being the maximum CP/M 2.2 can address).

The permutations of drives can be fun, one problem with SYS was that you couldn't easily mix drive types beyond 5" 48 tpi with 8", or 5" 96 tpi with 8". So mixtures of 48 tpi and 96 tpi were out as you couldn't make SYS double step 96 tpi drives to read 48 tpi formats. With BIOS 3.2 you can. The menu is impressive for home use, yet not powerful enough to make any competition to *Gemini's* own MFB series of machines. All the past and present *Gemini* formats are supported, naturally, that's both 96 and 48 tpi double density and the original double sided single sided GM805 format. Additionally, *Superbrain* 48 tpi single sided, *IBM* PC 48 tpi single and double sided 8 sector formats, *DEC* Rainbow, both *Nascom* formats and dear old fashioned 8" single sided single density. What's more you can assign more than one logical drive assignment to one physical drive. So for a person like me who gets a lot of disks in various *Gemini* formats, the following works a treat.

```
Logical Drive A:  Gemini 5M winnie
Logical Drive B:  Gemini QDDS   Physical drive 0
Logical Drive C:  Gemini QDDS   Physical drive 1
Logical Drive D:  Gemini DDDS   Physical drive 1
Logical Drive E:  Gemini SDDS   Physical drive 1
```

By sticking disks in either drive 0 or drive 1 as appropriate, I can copy anything that comes my way to either the other drive or the winnie. Similarly, if people expect things returned, I can copy stuff back to their original disks with no problem. The only problem comes when formatting disks, BIOS 3.2 does not supply a format program which will pick up the disk dph's and format to the standard in current use by the drive. Pity that, I'd have loved to have sent this disk to Paul in say *DEC* Rainbow format and then see how long it would take him to sus what I'd done. Never mind, I'll invent a wierd format of my own using ALLDISC, that should fox him completely and give *Gemini's* new MFB2 something to get upset over.

BIOS 3.2 is even more intelligent than it already appears. Lets take an instance where you are using a 96 tpi machine to construct a system track for a 48 tpi machine. It wouldn't be a lot of good if the result sent to disk was 96 tpi, as the 48 tpi drives couldn't read it. No if the BIOS constructed is a 48 tpi BIOS, then GENSYS will write the result to either a 48 tpi drive or double step (still equalling 48 tpi) to a 96 tpi drive, the boot sector being correctly set up for the 48 tpi format. This brings me to the second big feature of BIOS 3.2. Each physical drive can be specified separately. One drive could be an 8" for instance, another, a double sided 96 tpi with 3ms head stepping and no head load delays, another, a 96 tpi single sided with 20mS head stepping and 50mS head load delay and spectacularly long head settle times, whilst yet another drive could be a double sided 48 tpi drive with 10mS head stepping and no head delays. A maximum of four physical drive may be specified by type 8" or 5", the number of tracks per inch, the number of tracks. the number of sides, the step, head settle and load times.

BIOS 3.2 expects you to be logical about your logical to physical drive assignments, it no good specifying 96 tpi formats to a 48 tpi drive for instance, on the other hand it a 48 tpi format is specified to a 96 tpi drive, then the drive is forced to double step to read and write like a 48 tpi.

Reading 48 tpi disks with a 96 tpi drive works well, writing needs care, as the disk should be wiped clean with a magnet and reformatted using the 96 tpi drive before data is written, using the 96 tpi drive (hence the need for the format program). This is because the heads of a 48 tpi drive are wider than those of a 96 tpi drive and therefore track a wider area. If a disk previously written using a 48 tpi drive is rewritten using a 96 tpi drive, there will be areas of the original 48 tpi recording left between the tracks recorded by the double stepping 96 tpi drive. When the disk is placed in the 48 tpi drive, the head will track both the intended track recorded by the 96 tpi drive, and also the residual crud left from the original 48 tpi recording. This will degrade the signal to noise ratio, often to the degree that the disk can not be read.

```
                org     500h
                .phase  100h
;***************** GEMINI OVERLAY ********************
                if      GM811
;
; *** Gemini GM811/813 overlay

; THE GEMINI SOURCE AS IS

; *** End of Gemini GM811/813 overlay ***
                else
;
; *** Gemini GM870 Overlay ***
; For ready initialised DART using DIAL type program.
                jmp     300h            ; Absolute start of program
modctlp         set     085h            ; Modem control port for send
moddatp         set     084h            ; Modem data port for send
modcrcv         set     modctlp         ; Modem control port for receive
moddrcv         set     moddatp         ; Modem data port for receive
msndb           set     4               ; Modem send bit (xmit buffer empty)
msndr           set     4               ; Modem send ready
mrcvb           set     1               ; Modem receive bit (rx buffer ready)
mrcvr           set     1               ; Modem receive ready
;It is important not to alter the addresses of labels below
fastclk:        db      true            ; 4 MHz or greater processor speed
bakupbyte:      db      false           ; true=make .BAK file
xprflg:         db      true            ; true=menu initially off
twidth:         db      79              ; Max. terminal columns
saveflg:        db      false           ; true=terminal filesave initially on
echoflg:        db      false           ; true=terminal echo initially on
initflg:        db      true            ; true=modem port already initialised
ansbak:         db      true            ; true=answerback on ^E
inmodclp:       in      modctlp         ; Get send port status
                ret
incrcv:         in      modcrcv         ; Get receive port status
                ret
outmoddatp:     out     moddatp         ; Send data
                ret
anisnd:         ani     msndb           ; Bit to test for send ready
                ret
cpisnd:         cpi     msndr           ; Value of send bit when ready
                ret
inmoddatp:      in      moddrcv         ; Get received data
                ret
anircv:         ani     mrcvb           ; Bit to test for receive ready
                ret
cpircv:         cpi     mrcvr           ; Value of receive bit when ready
                ret
logmssg:        db      'Dave:Hunt',cr,lf,0
                .dephase
                org     550h
                .phase  150h            ; Do not alter this org
initmod:        jmp     geminit
olid:           db      2               ; Overlay identity
slo:            jmp     gmsetlo
shi:            jmp     gmsethi
txwait:         jmp     gmtxw
hexprt:         jmp     0               ; UKM7 fills in address
geminit:        ret
; Dynamic split baud sets
gmsetlo:        ret
gmsethi:        ret
gmtxw:          ret
; *** End of Gemini GM870 overlay ***
                endif
;********** END OF GEMINI OVERLAY *************
                .dephase
```

An optional facility for use with BIOS 3.2 is a utility called IBMCOPY. Now this program allows a system that can cope with 48 tpi, 40 track single or double sided disks to initialise, read and write, *IBM* PC 8 sector or 9 sector, single or double sided, PCDOS disks. This will be of great use to people who have a PCDOS machine at work, and a *Gemini* at home, as they will be able to transfer data to and from those different machines, or even to *Superbrain* or *DEC*! Remember though that the PCDOS programs themselves will be of little use, unless you have a *Gemini* GM888 board and have implemented PCDOS on that.

BIOS 3.2 has been supplied with standard *Gemini* CP/M's from about June this year. Owners of earlier CP/M's can get theirs upgraded by returning their original *Gemini* master disk to their dealer who will return it to *Gemini* for upgrade. Cost £30.00 plus VAT.

Well, that just about wraps it up for this time, back to the beginning, this is another 6500 words on the total assuming no-one goes mad with a blue pencil. Let me think, at fourpence a word, thats ...

---

## Back Issues

Please note that back issues of all 80-BUS News magazines are still available, except for Volume 2 Issue 6.

In order to reduce our stocks any issue from Volumes 1, 2 or 3 may be purchased for £1 each. Volume 4 issues are available at £1.50 each.

Postage must be added to these prices. Allow 30p per issue UK, 50p per issue overseas.

---

## Random Rumours (and Truths?)

**by S. Monger**

So, 80-BUS News is soon to cease? For those to whom this is news, let me just set the scenario. With the last issue of 80-BUS News, subscribers to this wonderful rag were sent a letter from our Editor, saying that once volume 4 was completed (this issue plus 2 more) there would be no more 80-BUS News. Oh woe, oh woe! Does this mean there will be a "Son of 80-BUS"? Well, our Ed. did go on to say that those that remained subscribers to the bitter end would receive FREE, GRATIS and FOR NOTHING a regular (since when has anything to do with 80-BUS been regular?) newsletter from Gemini. Well, as I'm sure that such an esteemed publication will have nothing to do with the likes of me, I thought that I had better write what looks like being my pen-penultimate rambling. And as you haven't heard from me since Volume 3 Issue 3 I know how incredibly thrilled that this must make you.

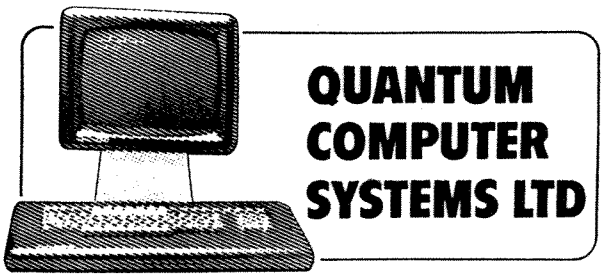
So what has happened since I last penned a masterpiece? [Ed. - an awful lot, as I can't remember ever receiving a masterpiece from you!] Well, let's quickly go through the 80-BUS/Nasbus manufacturers. Lucas - no sounds at all. Apparently the Nascom 2 still continues, unmodified from the original 8 year old design, and is largely used within the Lucas organisation. IO Research - Pluto 2 seems to be going very well, and at £2500 a time I'm sure that IO are going very well also, thank you. Belectra, Cotsgold, EV Computing, MAP80, Maas Computer Consultants - their products continue to be available. Microcode - I understand their products are now discontinued. Newburn - four new boards available from this new company to the 80-BUS manufacturing scene; A/D, D/A, multi-input, and multi-output boards. Also two new I/O boards soon from MRFS Ltd. Further details on these six boards when I have them.

And what about Gemini. Good news - GM802 RAM, GM813 CPU/RAM, GM832 SVC, and various systems are down in price. Also, glancing through their most recent MultiBoard catalogue (No.5 - the colour one) I note the following changes. The GM863-32 32K battery-backed static RAM board, GM863-64 64K bb s-RAM board, GM853 EPROM board, and GM870 auto-dial auto-answer V21/V23 MODEM, all marked as "in design" have now all been in production for some time. The GM851 12-bit A/D board, similarly marked, is just entering production. There is no sign yet of the GM855 tape streamer that is mentioned, and the GM723 ultra high capacity floppy drive has never appeared as the manufacturer went bankrupt! However, there is no mention of the GM842 D/A daughter board for the GM816, but this should be "available soon". Finally, the GM829 FDC/SASI board has been replaced by the GM849 FDC/SCSI board; the main reason for this change being the decreasing availability of the FDC chip set that was used on GM829.

On the Gemini system front: 20 Mbyte Winchesters have been available for some time - as a stand-alone system (GM924), as a MultiNet Fileserver (GM927), and as an add-on sub-system (GM835-20). Certain systems are now being shipped with 1/2 height drives. MultiNet 2 is available on all new network systems, or as an upgrade, and gives many worthwhile improvements. M-F-B 2 (the system for reading and writing disks of many different system types) has also been available for some time, and is currently being shipped with a library of formats rapidly approaching the 500 mark!

And finally, even though it is unrelated to 80-BUS, I just have room to make mention of the bomb-shell dropped by Gemini recently. In a word, "68K-BUS". Although Gemini have no intention of stopping development of 80-BUS products (as I'm sure the above must illustrate) they decided that they must also have a "proper" 16/32 bit range. They have therefore been beavering away quietly behind the scenes and have launched a range of 68000 based system and board products, and a new bus - 68K-BUS. Good luck to Gemini with this range (grovel, grovel), and can I write for 68K-BUS News please ??!!

---




Springfield Road Chesham Bucks HP5 1PU
Telephone (0494) 771987 Telex 837788

## TURBO PASCAL V3

* Absolute address variables
* Bit/byte manipulation
* Direct access to CPU memory & data ports
* Dynamic strings
* Free ordering of sections within declaration part
* Full support of CP/M facilites
* In-line machine code generation
* Include files
* Logical Operations on integers
* Overlay system
* Program chaining with common variables
* Random access data files
* Structured constants
* Type conversion functions
* Wordstar type editor
* Twice as fast as TURBO V2
* One step compile
* Installed for Gemini IVC/SVC

### WHAT THE CRITICS SAY

**PC MAGAZINE** "Language deal of the century...TURBO PASCAL. It introduces a new programming enviroment and runs like magic."

**POPULAR COMPUTING** "Most Pascal compilers barely fit on a disk, but TURBO PASCAL packs an Editor, Compiler, Linker, and Run time library into just 29k Bytes of RAM."

**BYTE** "What I think the computer industry is headed for: Well documented, standard, plenty of good features, & a reasonable price."

TURBO PASCAL IS AVAILIABLE NOW
GIVE YOUR SYSTEM A TURBO BOOST

| | | |
|---|---|---|
| TURBO PASCAL V.3 | CP/M 80 | 69.95 |
| " " " | CP/M 86 | 102.95 |
| TURBO 8087 version | CP/M 86 | 109.95 |

### TURBO TOOLBOX

Designed to compliment the power and speed of Turbo Pascal, TURBO TOOLBOX consists of three modules created to save you from "rewriting the wheel" syndrome.

**TURBOISAM Files using B+ Trees**
Makes it possible to access records in a file using keys (e.g. "Smith" or "Rear Bumper") instead of just a number. Even though you have direct access to individual records in the file you also have easy access to the records in a sorted sequence. A must if you plan to work with direct access files. Source code is included in the manual.

**QUICKSORT ON DISK**
The fastest way to sort your arrays. Preferred by knowledgeable programmers. Available for you now with commented source code.

**GINST (General Installation Program)**
Now...the programs you write with Turbo Pascal can have a terminal installation mode just like Turbo's!. Saves hours of work and research, and adds tremendous value to everything you write.

TURBO TOOLBOX 54.95

### TURBO TUTOR

Don't know Pascal?..... Let Turbo Tutor teach you. Supplied as a disk of demonstration programs and a very informative and entertaining manual this package will show you how to use your Turbo Pascal to best advantage.

TURBO TUTOR 34.95

All disks supplied in Gemini QD-96tpi format as standard, please state your format if different. All Prices are exclusive of Vat (15%) and Carriage & Packing 1.50. Personal Callers by appointment Only.