LERM SAM TOOLKIT

MACHINE CODE UTILITY - FOR ANY SAM

CONTENTS

SAM TOOLKIT 1

EXTRA MANUAL FOR SAM TOOLKIT

Welcome to SAM TOOLKIT. You have been supplied with our SAM ASSEMBLER package of programs PLUS 2 extra programs - CTOS and a second DISASSEMBLER called "auto dis". In addition you also get our GUIDE to writing machine code on the SAM - this appears later in this manual. You should read through the SAM ASSEMBLER manual first, and the rest of this supplement afterwards. If you have ROM2 and were supplied with SAM TOOLKIT on DISK, you can press F9, then SAM TOOLKIT will load in with a simple MENU program called "auto". Press keys 1 to 4 to select the program you want to LOAD. Alternatively you can enter BOOT 1, then enter LOAD followed by the programs name. (The type FONT is changed by AUTO, but you can delete line 150 in the "auto" BASIC to keep the usual SAM font if you wish). IF SAM TOOLKIT WAS PROVIDED ON DISK, WE WILL ONLY HAVE FORMATTED ENOUGH TRACKS NEEDED TO MAKE SAM TOOLKIT SO DON'T SAVE ANY MORE FILES ONTO YOUR LERM DISK.

MANUAL FOR THE 2nd DISASSEMBLER CALLED "auto dis"

This program loads into address 26000 in SAMs memory, as opposed to the one supplied in SAM ASSEMBLER. It is loaded by simply using LOAD "auto dis". We have included this separate DISASSEMBLER because the one in SAM ASSEMBLER is more restrictive, and can't easily be used to DISASSEMBLE ROM 0 or 1, or do DUMPS below 32768. "Auto dis" resides between 26000-32767, so apart from that range of addresses will

* disassemble ROM 0 or ROM 1
* disassemble CODE or use DUMPS for all addresses except 26000-32767.

NOTE: You can use it for addresses 26000-32767 if you use the DISPLACEMENT "x" option.

The instructions are the SAME for both DISASSEMBLERS EXCEPT the following:

1. The ESC key will escape from PRINTING if the printer is NOT switched ON.
2. When you press the "R" key, instead of disassembling the code FROM 49152 to 65535, ROM 1 is put into these addresses and disassembled instead. Pressing the "R" key again toggles back.
   e.g. Load in "auto dis", and when asked to input an address enter 49152. You will see NOPs down the screen. If you press "R" you will see a disassembly of ROM 1 appear. Pressing "R" again will revert back to a disassembly of address 49152.
3. To get from the disassembler back into BASIC press the "Q" key for Quit. TO RE-ENTER use RUN.
4. To make a copy of "auto dis" from SAM BASIC, simply press F0 to DISK, or F1 to TAPE.
5. To change the PALETTE colours from the standard one supplied simply add the appropriate instructions using lines 91 to 120.
6. To disassemble ROM 0 simply enter the address required in the range 0-16384.
7. To send a disassembly to the printer press the "P" key. To stop sending, press "P" again.
8. The ESC doesn't work within the disassembler.

THE MANUAL FOR SOURCE CREATOR (CTOS)

SOURCE CREATOR has been written to allow you to create a source file for the SAM ASSEMBLER from any piece of machine code (or "object code") up to approximately 5000 bytes in length. i.e. an assembler creates machine code from a source file, this program does the reverse. Allowance is also made for bytes which represent blocks of data (e.g. messages to be printed etc). CTOS by the way stands for Code TO Source. The program is in fact a reverse assembler.

If CTOS was provided on DISK then it can be loaded using the F9 key, followed by pressing key "3". Alternatively, you can enter BOOT 1, then LOAD"auto ctos". To load from TAPE press the F7 key and PLAY the tape. To copy the program, break into BASIC - this can be done by entering FFF when asked for the start of the code. Now press F0 to copy to disk, or F1 to copy to TAPE.

When CTOS has loaded it asks you for the NAME of the CODE file that you want the program to work on. If you press ENTER, CTOS assumes that the name of the file on your disk is called "CONVERT". If it isn't you must enter the proper name.

The program naturally requires you to tell it which bytes, if any, and DATA (e.g. spaces, messages, etc), so that it knows that the remaining bytes are genuine machine code. Within your CODE you can have up to 100 blocks which are DATA, and CTOS asks you where each block STARTS and ENDS. It can't do the impossible, and you may well still have some work to do manually, however it will save a great deal of time.

The main uses for CTOS are:-
(a) Moving machine code to a higher or lower address.
(b) As a tutorial in conjunction with the TOOLKIT.
(c) As a tool to enable users to alter or modify pieces of code from ROM's or other sources.
(d) As a way to transfer files from any other assembler.

OPTIONS ARE AVAILABLE TO :-
(i) Create data blocks up to a maximum of 100. These will appear in the final source file as DB's. 10 bytes will be inserted on each line of the source file.
(ii) Create labels within the source file.
(iii) Create EQU's to enable easier alteration of any absolute addresses within the object code.

FILE LENGTH..........

The maximum SOURCE FILE length will be just over 32000 bytes. If CTOS runs out of room while creating a file then you will be told "NO MORE ROOM FOR FILE". You will still be able to save the file CTOS has created up to that point but it would be better to try again with a smaller block of code.

GETTING STARTED......

Before any work should be done on creating a source file, a few words of advice. Any piece of object code you wish to create a source file from should at least be fairly well known to you. There is little point in creating a file if you don't know what it does! First have your section of OBJECT CODE saved to a separate disk or tape. This must be no longer than 5000 bytes in length. There are ways that any length of object code may be processed, but this must be done in small sections and the various pieces of OBJECT CODE joined after assembly. This process is only recommended to those who are familiar with machine code and would involve many hours of work.

Assuming your piece of OBJECT CODE is less than 5000 bytes, we would advise using a small block of code first. Under 1k for a start, then follow the instructions on screen.

CREATING THE SOURCE.

When loaded you will see the prompt "INPUT ORIGINAL START" This must be the ORIGINAL address of the code block. i.e. If the code to be converted usually resides at address 32768, then you must enter this address at the above prompt.

Once entered you will be asked "INPUT ORIGINAL END" so if the block started at 32768 and you were converting 1000 bytes then you would type 32768+999 at this prompt. You add 999 as the addresses are INCLUSIVE. NOTE if the address you type is lower than the original start, the program will wait for a sensible end address. Also if the end address is more than 5000 bytes further on from the start address, then again the program will wait for the proper address.

After this you will be prompted "HOW MANY DATA BLOCKS", just pressing RETURN will enter a zero and zero means NO DATA BLOCKS, indicating all of the CODE is pure machine code. Up to 100 data blocks are allowed and this should be adequate for any piece of code. Data blocks should have been found and noted from your original investigation of the code.

NOTE inserting data blocks at the relevant places will create less source file than defining no blocks. Some detective work will be needed to find the data if any exists, and we recommend using the NUMERIC and ASCII dumps from within the SAM ASSEMBLER to find any relevant data. The ASCII dumps are particularly helpful so that you can easily see messages that should be printed. Then as long as you know what the program is doing, and if you think there are any numeric blocks, try to find them before creating a file.

You will need to note the start and the end of the blocks. If you have entered a specific number to the prompt above, then the program will now enter a loop and ask you for each start and end of block in turn until all blocks are entered. Blocks can be defined outside of the code but they will be ignored when the source is created. Once again the end address of a block must be higher than the start of that block. Blocks can be entered in any order - the program will still read through them. During source creation if a block of data falls half way through a legal instruction (meaning that you have defined the start wrongly), then the block will be ignored, so try to make sure that you have the actual start of the block.

If you allow any ASCII data to be converted into source without defining a data block for it, then a lot of meaningless labels will be created. This is because 2 (jump relative) instructions are within the ASCII code range, and the program will attempt to create labels that are not really needed.

Once all the data blocks have been entered (if any), the program will create the source file without LABELS or EQU's. Once this has been done you will be asked "DO YOU WANT LABELS Y or N". Also shown on screen will be the start address and the length of the SOURCE FILE that CTOS has created for you so far. If you answer "Y" to the prompt then the program will insert as many labels as it can. If a label falls half way through an instruction then it will not be inserted. If CTOS created lables for you, it will then ask if you want it to create EQUs and again to simply press "y" for yes, or "n" for no. EQUs are created for any numbers that are needed which are NOT labels.

Finally CTOS will ask if you want to SAVE the source file or LOAD THE ASSEMBLER. Pressing "S" will prompt you for a filename and the SOURCE FILE will be saved for you. Pressing "A" will ask you to put your ASSEMBLER disk into the drive, and having pressed any key, it will load in the ASSEMBLER for you. At this point the source file that has been created by CTOS will reside in page 1 of the assembler. You should press the "a" key to go into the assembler at this point.

When creating labels you will be told how many have been found inside the code, and how many have been found outside the code. Also shown is how many labels have been inserted - this is the most time consuming part of the program so go and make a cup of coffee if you are working on a large block! The labels found outside of the code can be inserted as EQU's, and as stated above, you will be asked, once the main labels have been inserted "DO YOU WANT EQU's Y or N". CTOS will always try to put EQU's at the top of the file if it can, otherwise it will put them at the end of the file.

DETAILS OF THE SOURCE FILE.

LINE NUMBERS........

SAM ASSEMBLER uses line numbers and usually defaults to line 10 and steps of 10. When a file created from CTOS is loaded into the assembler the first thing to note is that the line numbers take the form of addresses. This means that if the start of the code block you converted was at 32768 then the first line number in the source file will be 32768 (unless you also created EQU's), the rest of the file will look just like a disassembly of the code. NOTE DO NOT RENUMBER THE FILE UNTIL IT ASSEMBLES PROPERLY. It should have an appropriate ORG added to the source file.

LABELS...............

The labels will take the form "L + ADDRESS" i.e: CALL 12345 will be seen as CALL L12345, To find L12345 just use (LIST 12345) this will list line 12345 onwards and a label should be present at that line number.

EQU's...............

EQU's will be seen much the same as labels i.e. if label L12345 is outside of the file it will be seen as : L12345 EQU 12345, thus when the assembler assembles the file then label L12345 will be converted to 12345 - in other words CALL L12345 will be assembled properly as CALL 12345.

Sometimes a label will be assigned where an absolute address was meant - for instance the code below is a sixteen bit timing loop. Listing 2 is the file you would write using the assembler, listing 3 is the file that CTOS would create. The 32768 in line 32768 is a timing constant that would not want to be changed. This would not be a problem if you were just re-assembling the code to the same address that it came from (i.e. 32768), but if you decided to move the code to address 30000 by adding so:

10 ORG 30000

Then on re-assembly the label L32768 would point to address 30000 and the constant would be changed to LD BC,30000. The way round this is to edit out the "L" so leaving the instruction as LD BC,32768. This sort of thing has to be watched for throughout the CTOS source file. BC could have been any of the register pairs HL, DE, IX, or IY.

LISTING 2 NORMAL FILE.	LISTING 3 CTOS CREATED FILE.
`00010 ORG 32768`	
`00020       LD BC,32768`	`32768 L32768 LD BC,L32768`
`00030 LOOP  DEC BC`	`32771 L32771 DEC BC`
`00040       LD A,B`	`32772        LD A,B`
`00050       OR C`	`32773        OR C`
`00060       JR NZ,LOOP`	`32774        JR NZ,L32771`
`00070       RET`	`32776        RET`

In the event that you are creating a source file starting at address zero, then to put an ORG into it you would have to create a multi-statement line i.e. 00000 ORG ????? : LD A,10 or whatever.

JR INSTRUCTIONS.....

If you create a source file with no data areas defined, then all JR instructions will show the instruction plus the label pointing to the jump address as well as the actual displacement byte shown after the "*". The displacement acts on the two's complement notation. The reason the byte is shown in this way is so that you can see if the byte is really ASCII data.

As an example the instruction JR NZ,12345 could actually be data it might be seen in the CTOS file as JR NZ,L12345 ;* 65. The instruction JR NZ is byte 32 (space in ASCII). The byte 65 in this example is the offset byte and would be added to the address of the instruction JR NZ to form a new address to jump to. It just might be some ASCII data though and if it was it would mean " A" (space then A). If on investigation you decide that this instruction is data then you could convert the source file to read DEFB 32,65.

Again if you only intend to re-assemble the code back to its original address then it would not matter as the offset byte would not change. If you inserted some more instructions into the source file between the instruction and the destination label then the byte would change, and what should read "A" will now be something totally different. For instance if you inserted just one instruction, say a NOP, then on assembly the JR NZ 12345 would be changed to JR NZ,12346. Thus the sequence 32,65 would have been changed to 32,66 (space B).

Just defining one data block will suppress this option and of course create less source file. So if you want to create a file with no data blocks, and at the same time suppress the displacement byte being printed in the file, then define one data block and give it START and END addresses that are outside the block you are working on.

CTOS SUMMARY.

Once the source file has been created it can be loaded into the SAM ASSEMBLER and edited in the same way as a normal source file.

* Do investigate the code to be converted before creating a file.
* DO LOOK for embedded data and ASCII strings to define as data blocks.
* DO check for constants that have had labels assigned to them.
* DO NOT re-number the file until you have finished editing it.
* DON'T try altering the code until you can assemble and run the code as it originally was. Then and only then, make any alterations that you may want.
* DO TRY to change labels that are more meaningfull to you.
* WATCH out for those JR's as they might be data.
* EXPERIMENT with small blocks until you get used to using CTOS.
* PLEASE DON'T USE CTOS TO PIRATE SOMEBODY ELSES CODE.

CTOS is most useful for converting those small routines that appear in various magazines to an address more convenient for you. Most useful is the ability to take an old SPECTRUM routine and convert it to a source file before modifying it to work on the Sam Coupe. Also code produced from other assemblers can be converted to a source file that will load into the SAM ASSEMBLER.

GUIDE TO WRITING SAM MACHINE CODE

USING THE SAMS MEMORY PAGING SYSTEM

The following GUIDE has been written to help you get started with SAM MACHINE CODE. It assumes that you already know how to write CODE using a Z80 chip, but are not familiar with how the SAM works. A full technical manual is available from SAM COMPUTERS.

THE MEMORY - PAGES AND BLOCKS

The 256k SAM has 16 PAGES each of which contains 16k (16384 bytes) of memory - if you multiply 16k by the 16 pages you get the 256k! The first PAGE is numbered 0, the next is 1, and the last is PAGE 15.

The 512k is the same except that it has 32 PAGES numbered from 0 to 31. Simply regard a PAGE of memory as like a tray that can be slid out of a rack and replaced by another.

In both machines these PAGES are RAM - i.e. memory that can be changed by the computer or yourself. You can READ it using PEEK, or change it using POKE.

In addition there is the ROM - this can be READ but you can't change it at all with a POKE. The ROM contains BASIC and uses TWO 16k BLOCKS called ROM 0 and ROM 1.

The SAM is an 8 bit computer- i.e. in each address you can only put (or POKE) a number from 0 to 255. In BINARY this is from 00000000 to 11111111 - there are 8 BITS that can 0 (or RESET or LOW or OFF). Any of the bits can be 1 (or SET or ON or HIGH). Now the Z80 central chip that runs the computer can only use addresses from 0 to 65535 - yes we know that you can POKE 131000 with a number but this isn't really the case at all as you will see later.

So the question is how does the SAM use the extra memory available? Let us imagine the whole of the memory from 0 to 65535 set out in BLOCKS of 16k. The SAM has therefore 4 BLOCKS which we will label A, B, C, and D.

0-16383	16384-32767	32768-49151	49152-65535
BLOCK A	BLOCK B	BLOCK C	BLOCK D

Now it is possible to take any of the PAGES of RAM and put them into any of the 4 BLOCKS that are shown. Indeed ROM 0 or ROM 1 can be put into any of the BLOCKS. The SAM constantly takes out a PAGE from one BLOCK and substitutes another.

Incidentally, we are going to use DECIMAL in our explanations. In HEX 0-16383 is 0-3FFF, 16384-32767 is 4000-7FFF, etc.

When you switch on the SAM and start typing in BASIC the following happens:

BLOCK A	BLOCK B	BLOCK C	BLOCK D
ROM 0	PAGE 0	PAGE 1	PAGE 2

The SAM puts ROM 0 into BLOCK A (0 to 16384). In BLOCK 2 are placed several items, and the start of your BASIC program (16384-32767). BLOCKS C and D are available for BASIC or CODE. When you use BASIC BLOCK D it is constantly changed by sliding in ROM 1 while it is needed and then putting PAGE 2 back again when it isn't. The lower ROM 0 takes care of this for you.

To appreciate how this might happen let us imagine that you have a fairly large piece of BASIC in your SAM of 27k length. The SAM will store the start of this in PAGE 0, at about 23760. As the BASIC is 27k long it will use all of PAGE 1 and go into PAGE 2 so:

BLOCK A	BLOCK B	BLOCK C	BLOCK D
0-16383	16384-32767	32768-49151	49152-65535
ROM 0	PAGE 0	PAGE 1	PAGE 2
Your 27k of	Start at	---------------	End at approx
BASIC	23760		51408

(27k is 27*1024 bytes = 27648 - so end of BASIC is 23760+27648 which is 51408).

Now when you type in a new line of BASIC, or edit an old one, the SAM slides PAGE 2 out of BLOCK D and replaces it with ROM 1. When you type in your new line of BASIC it is placed somewhere in PAGE 0 which is in BLOCK B so:

BLOCK A	BLOCK B	BLOCK C	BLOCK D
ROM 0	PAGE 0	PAGE 1	ROM 1

Now when you have finished entering the new line of BASIC and the syntax is ok, ROM 0 will move ROM 1 out of BLOCK D and replace it with PAGE 2. The machine code in ROM 0 will then copy the bytes of your new line of BASIC into the correct position in PAGE 2.

All this memory switching is very fast and is done by a very simple piece of machine code: OUT (251), A
where the value of A selects the PAGE number. This can only be done within machine CODE - not BASIC.

LOOKING AT PAGE 0
Let us look at PAGE 0 for a moment. This normally resides in BLOCK B but you could put it into another block if required. In BLOCK B the PAGE starts at 16384 and ends at 32767. BASIC makes use of it so:

NORMAL PAGE 0 at BLOCK B

16384-?? HEAP Approx 3k	??-?? Storage area needed by BASIC	20736-20885 PAGE allocation	20886-21647 Character Patterns
21648-21975 UDG patterns	21976-22015 Palette table	22016-22527 Colour table	22528-23039 Keyboard tables
23040-23733 System variables	23734-?? Channels area	Approx 23760- BASIC	

The above has been written to READ from left to right. At 16384 is the start of the HEAP area. This is a SPARE area that you can use to put some machine code - up to approx 3k, and run it. So write your code, and ORG it to run from this area.
EXAMPLE:

```
10          ORG 16384:PUT 32768
20          LENGTH EQU END-START
20 START    LD HL,(&5AA0):LD BC,16384
30          AND A:SBC HL,BC:PUSH HL:POP BC
40          RET
50 END      NOP
```

Use our ASSEMBLER to write the above. Having assembled it, type

SYM, and you will see that LENGTH is equal to 11 - showing you that you have produced 11 bytes of m/code. Do a QUIT and save this OBJECT code with the name BASTART from 32768, and length of 11. Now reload the code into your SAM using LOAD "BASTART" CODE 16384. Now type PRINT USR 16384, or LET r=USR 16384:PRINT r. This code is placed into the HEAP area, below BASIC, and is therefore a handy area to use as it doesn't interfere with CODE placed above BASIC either.

EXPLANATION:
The system variable at &5AA0 (which is HEX 5AA0 as our assembler copes with & or a HASH to denote HEX) is the start of BASIC but is exactly 16384 bytes too long. So to find the address of the START of BASIC you can use PRINT ((DPEEK &5AA0) - 16384) OR use the above machine code. This CODE puts the start of BASIC into HL at line 20, then subtracts 16384 from it. It then puts the value into the BC register using PUSH HL:POP BC, and returns to BASIC. This is because when you RETURN to BASIC, the value held in the BC register is given to BASIC. So using LET r=USR 16384 makes the value of r EQUAL to the BC register.

OTHER AREAS OF INTEREST:
The SAM stores the bytes that creates the FONT for characters 32-127 starting at 20886. The USER defined characters start at 21648 - each character needs 8 bytes. The Palette table holds the bytes associated with each Palette number. If you do Palette 0,34 this makes colour 0 set to HELLFIRE. So if you do PAPER 0 and CLS, the screen will have a colour of HELLFIRE. To change paper 0 back to black you can enter

(i) PALETTE 0,0 OR
(ii) POKE 21976,0 then POKE 21996,0

At address 21976 is the bytes for Palette 0, 21977 is for Palette 1 and so on up to Palette 15. The same numbers must be POKED 20 bytes higher if no flashing is to occur - that is why we POKED both 21976 and 21996 with 0. If there are different bytes then you will flash between them so:

	Address colour1	colour 2
Palette 0	29176	29196
Palette 1	29177	29197
Palette 2	29178	29198
etc.		

So for no flash PEEK 29177 and PEEK 29197 should be the same. POKE 29178,48 and POKE 21998,34 sets PALETTE 2 to flash between WOAD and HELLFIRE. Doing PRINT PAPER 2;"fred" will print "fred" on a flashing paper colour.

THE STACK
This is placed in PAGE 0 of memory at around address 20180. You must be very careful if you MOVE PAGE 0 out of BLOCK B as that is where the stack normally lives. If you always leave it in this position, then you can forget about the stack completely.

GETTING STARTED with using the memory
It is easiest if you simply leave BLOCKS A and B alone containing ROM 0 and PAGE 0, and restrict the size of your BASIC to lie between 23760 (approx) and 32767 - i.e. keep it in PAGE. So keep the size of BASIC down and use CLEAR 32767 as the first instruction in BASIC. You can now:

(a) Use PAGES 1,2,3,4, etc in BLOCKS C and D
(b) Use the HEAP area from 16384 to approx 19500
(c) Have enough room for SOME BASIC.
(d) You can forget about the stack.

BLOCK A	B	C	D
ROM 0	System+ your BASIC	PAGE 1	PAGE 2

When you want to put different PAGES into BLOCKS C and D this can only be done TWO PAGES at a TIME.

LD A,3:OUT (251),A

Port 251 controls which memory pages are BLOCKS C and D so the above code will put Page 3 into Block C and Page 4 into Block D.

	BLOCK C	BLOCK D
Before	Page 1	Page 2
After LD A,3:OUT (251),A	Page 3	Page 4
After LD A,10:OUT (251),A	Page 10	Page 11
etc.		

You can now write your m/code to ORG from 32768, and you have 32k available in PAGE 1 and PAGE 2.

AN EXAMPLE OF MEMORY PAGING:

Lets imagine that you are writing some code that is over 32k long - perhaps 70k. When you get near the end of your 32k you will want to start with some new memory and will need to remove PAGES 1+2 and put in PAGES 3+4 into BLOCKS C and D. This is how it is done:

We might have some CODE in the HEAP area that is never moved out so:

```
            10  ORG 16384:PUT 32768
LIST        20  START      OUT (251),A
ONE         30             JP (HL)
            40  END        NOP
```

OR simply have, in your BASIC a line POKE 16384,211,251,233 as this will put the above code at 16384 for you.

Now let us imagine that you have ORGd your code to 32768 so:

```
            00010                  ORG 32768
            00020 ; 1st 32k of my program - Pages 1+2
            00030 BEGIN1           CALL SETUP
           ...........
                      etc
LIST       ...........
TWO         02000                  LD HL, 40000
            02010                  CALL JOHN:CALL STORE
            02020 ;  Now you want to move memory
            02030 ;
            02035 MEMEND           LD HL,BACKHERE:LD (27000),HL
            02040 MOVEMEM          LD A,3:LD HL,32768
            02050                  JP 16384
            02060 BACKHERE         LD HL,34000; returns from PAGE3+4
            etc
```

This is the source file for the next 32k of CODE, again ORGd at 32768, but will be placed into PAGES 3+4. The start of PAGE 3 in BASIC is 65536 (see later for explanation of BASIC page boundaries.)

```
                10010                       ORG 32768
                10020 ; 2nd 32k of my program; load into 65536
                10021 ; as it runs from PAGE 3.
                10025 BEGIN2                LD IX,34000
LIST            10030                       CALL ALAN
THREE           10040                       LD HL,4
                etc

               101000 ;  Now to move back to PAGES 1, and 2
               101010 MOVEMEM2              LD HL,(17000):LD A,1
               101020                       JP 16384; moves back to "BACKHERE"
```

EXPLANATION:
The CODE (LIST ONE) is put into the HEAP area at 16384 that is in PAGE 0. This allows us, when running CODE in BLOCKs C+D with PAGE 1+2 to jump OUT of BLOCKs C+D into BLOCK B, switch the memory to PAGES 3+4 in BLOCKs C+D, and then jump back again to the new code in BLOCKs C+D. Also the HEAP can be used to store data needed for all the CODE you run in BLOCKs C+D.

So having POKED 16384,211,251,233 and placed the CODE produced by LIST TWO into 32768, and the CODE produced by LIST THREE into 65536 we can explain what would happen. You would begin from BASIC with the CLEAR 32767, then do a CALL 32768.

The code produced from LIST TWO, starting at BEGIN1, would then be running with the memory so:

BLOCK	A	B	C	D
	ROM 0	PAGE 0	PAGE 1	PAGE 2

The CODE calls SETUP, and runs until you get to CALL JOHN, then CALL SETUP. At the line MEMEND we want to stop running the CODE in pages 1+2, and switch for a while to pages 3+4, and then return back to Pages 1+2 again from BACKHERE.

At MEMEND we store the value of BACKHERE in the HEAP area at address 17000. At MOVEMEM we set the "A register to 3 for PAGE 3, and HL at 32768, as this is the address we want to start running our new code from in Page 3. Line 2050 does a JUMP 16384, so the Z80 chip moves to 16384 with A=3 and HL=32768. At 16384 the SAM is in BLOCK B, and the CODE switches BLOCKS C and D to PAGES 3+4 so:

	BLOCK C	BLOCK D
Before	Page 1	Page 2
After LD A,3 OUT (251),A	Page 3	Page 4

Then the SAM does the instruction JP (HL), which, as HL=32768 in this case, causes SAM to run from 32768 in BLOCK C which now has PAGE 3 in it!

So the CODE from LIST THREE would now run from BEGIN2. The IX register will be loaded with 34000, etc. The code will run until you get to MOVEMEN2. At this point you want to move back to PAGES 1+2, so line 11010 loads HL with the contents of address 17000 which was previously used to store the value of "BACKHERE". The A register is loaded with ONE, for PAGE 1, and the JP 16384 at line 11020 causes SAM to move out of BLOCKs C+D and back into BLOCK B. The CODE in the HEAP area now switches PAGES 1+2 back into BLOCKS C+D, and the JP (HL), makes the SAM run from line 2060 - "BACKHERE".

We hope that you have understood what is going on. Clearly any data needed by all PAGES of your code must be stored in BLOCK B, and we suggest that valuable HEAP area for this.

There is still more to come, like how to access the screen memory, using BLOCKS A and B, and so on, but for the moment, we suggest that you get used to the above.

PAGES ALREADY USED

The TOP THREE pages of your SAMs memory should be left alone:

PAGES 13, 14, and 15 for the 256k SAM
PAGES 29, 30, and 31 for the 512k SAM

This is because the TOP TWO pages are used by the SCREEN and the PAGE 13 (29 for the 512k) is where DOS is placed.

PAGE BOUNDARIES

As we said earlier, BASIC pretends that you can have addresses above 65535. This is how it works.

PAGE	BASIC start of page
1	32768
2	49152
3	65536
4	81920
5	98304
6	114688
7	131072
8	147456
9	163840
10	180224
11	196608
12	212992
13	229376
14	245760
15	262144

Indeed this simple BASIC will calculate the START (or PAGE BOUNDARY) of any PAGE:

```
LET START=16384*(PAGE+1)
```

For the 512k SAM you carry on up to PAGE 31. In the 256k SAM DOS is in PAGE 13 which starts at 229376. When you copy DOS this is done from 9 bytes higher up at 229385. When DOS runs this is what happens to memory:

	BLOCK B	BLOCK C	BLOCK D
Before	Page 0	Page 1	Page 2
During DOS	Page 13	Page 14	Page 2
After	Page 0	Page 1	Page 2

When you load/save bytes the machine code in the DOS loads the bytes into BUFFERS (memory spaces) in Page 13, then memory is switched as appropriate, and copied into correct PAGES by putting those PAGES into BLOCK C+D. After use, the PAGES are restored to their usual BLOCKS.

So you can ORG some machine code at 32768 (up to 32k in length), to run in, for example, PAGE 8. To run this code load it into address 16384*9 (using our formula), and then do CALL 16384*9 to run the CODE. BASIC will automatically put PAGE 8 into BLOCK C and PAGE 9 into BLOCK D. When you return to BASIC, then SAM will automatically restore BLOCKs C+D to PAGES 1+2.

To test this out assemble the following CODE.

```
10          ORG 32768
20          LD HL,40000
30          LD  (STORE),HL
40          RET
50 STORE    DS 2
```

If you use SYM you will see that STORE EQUALS 32775. Now do QUIT and save the CODE as "TEST" from 32768 with a length of 10. Now QUIT to BASIC and do CALL 32768, then PRINT DPEEK 32775. You should get 40000 printed on the screen. Now do LOAD"TEST"CODE 9*16384 followed by CALL 9*16384. This time do PRINT PEEK (9*16384+7), and again 40000 should appear on the screen. This is because although the value of STORE is 32775 it should really be regarded as the "START OF THE PAGE PLUS 7 BYTES" - i.e. the PAGE BOUNDARY with an OFFSET of 7. So when running the CODE in PAGE 1 you do PRINT DPEEK 32775, and in PAGE 8 it is DPEEK (9*16384+7), but in BOTH CASES you are really looking at the START of the PAGE plus an OFFSET of 7 bytes, and the CODE is run at 32768, but with different PAGES in BLOCK C.

USEFUL VARIABLES

ESC key

To disable the ESC key do POKE SVAR &141,1

To enable it again do POKE SVAR &141,0

CAPS LOCK

To put CAPS LOCK on do POKE 23658,8

To switch it back to lower case POKE 23658,0

HOW TO SECURE BASIC.

The ON ERROR GOTO fred or whatever is easily stopped from within BASIC by simply pressing the NMI button. We are going to show you how to secure BASIC.

When you are in BASIC it is easy to stop the ESC key working using the POKE indicated above. However, you can BREAK into BASIC using the NMI button at the back. The variable at &5AE0 is the VECTOR that can be POKED to change this. When you press the NMI button, SAM does DPEEK &5AE0, and runs from that address, doing the break into BASIC. If you DPOKE &5AE0,addr where addr is an address from which you want to run your code then no break to BASIC will occur. Try the following.

```
10                  ORG 18000:PUT 32768
20 START            DI:LD DE,START:LENGTH EQU END-START
30                  LD SP,20224
40                  PUSH DE:LD (23613),SP
50                  XOR A:LD (23610),A
60                  LD HL,(STORE):EI
70                  JP 271
75 STORE            DS 2
80 END              NOP
```

This code can be assembled and saved - it is 25 bytes long, and STORE is 18023. Save it with the name "ONERR". Having loaded in the "ONERR" CODE into 18000, your BASIC program should have something like the following:

```
10000 DEF PROC onerr
10010 POKE SVAR &141,1:DPOKE &5AE0,18000:DPOKE 18023,2
10020 END PROC
```

To set this up from within your BASIC do the following:

```
1 ONERR:CALL 18000
2 REM:Rest of your BASIC program
```

Run this BASIC and LINE 1 does the ONERR command. Every time an error is met or you press the NMI button, the program will run from the LINE number in address 18023 - we made it line 2 in the above example. So FROM within your own BASIC, to change the line number from which you want to run if an error or break is met, simply DPOKE 18023 with the new line number.

You could use the command LET a$=MEM$(18000 to 18024) and save this m/code from within BASIC. Your BASIC could then be, as its FIRST INSTRUCTION, POKE 18000,a$, then do the above PROCEDURE called ONERR. The new ON ERROR routine will then be in place.

The CODE works by setting up a NEW stack at 20224, and it then puts DE=start of the routine. This address is pushed onto the stack, and the system variable ERROR STACK POINTER (23613) is loaded with the value of the STACK POINTER. Zero is placed into 23610, then HL is loaded with the contents of address STORE, which contains the line number from which BASIC should run. The final JUMP is to the ROM location - ROM moves to the BASIC line number given by HL. From now on, when an ERROR of any sort happens, or the NMI button is pressed the error is trapped.

If there is an error in BASIC, then SAM looks at the address inside 23613, and loads the STACK POINTER with this address, and does a RET. Therefore the SAM picks up the value we pushed onto the stack (=START), and runs from that address - i.e. 18000.

If the NMI button was pressed, even when using DOS, the SAM looks into address &5AE0, in this case 18000, and runs from there.

So in both cases, the SAM is forced to run from address 18000 which resets up the values of the STACK, and addresses 23613, and 23610, and then looks up the contents of 18021 to find out from which BASIC line number we should run from.

USING THE SCREEN

The following can be done from BASIC easily enough.

```
        10 MODE 4:CLS:PRINT AT 10,3;PAPER 1;PEN 5;"FRED"
        20 PRINT !0;AT 0,5;"JOHN";TAB 20;"BOY":PAUSE
```

Sorry, my printer won't print a HASH so I have used an ! instead. This will print "JOHN", but also a message on the bottom 2 lines normally used by the INPUT command. How can you do this in m/code?

```
        10          ORG 32768:LENGTH EQU END-START
        20          OPEN EQU &112:AT EQU 22:PAPER EQU 17
        24          PEN EQU 16:LASTK EQU 23560
        25 START LD (STACKSTORE),SP         ;STORE RETURN ADDRESS
        28;
        30          LD A,2:CALL OPEN        ;OPENS UPPER SCREEN
        40          LD HL,MESS1:CALL PRINT  ;PRINTS MESS1
        45;
        50          LD A,0:CALL OPEN        ;NOW PRINT MESS2
        60          LD HL,MESS2:CALL PRINT
        65;
        69          CALL PAUSE
        70          LD SP,(STACKSTORE):RET  ;BACK TO BASIC
        80;
        90 MESS1 DB AT,10,3,PAPER,1,PEN,5
       100          DM "FRED":DB 255
       105;
       110 MESS2 DB AT,0,5
       120          DM "JOHN":DB AT,0,20
       125          DM "BOY":DB 255
       130;
       140 PRINT LD A,(HL):CP 255:RET Z
       150          RST 16:INC HL:JR PRINT
       160          STACKSTORE DS 2
       165 PAUSE XOR A:LD (LASTK),A ; PUTS 0 into LASTK
       166    PA1 LD A,(LASTK):CP "a:RET Z
       167          LD A,(FLAGS):RES 5:LD (FLAGS),A
       168          JR PA1
       170 END   NOP
```

Assemble the CODE, then do the following from BASIC.

```
   9000 MODE 4:CLS:CALL 32768:STOP
```

and then goto 9000.

EXPLANATION:

You can PRINT to any of 3 CHANNELS. The TOP of the screen, the bottom few lines (used by INPUT), or to a printer. In machine code, before printing you must set up the correct CHANNEL. To do this simply load the A register with 0, 1, or 2 so:

```
  LD A,0 for the bottom lines (normally used by INPUT)
  LD A,2 for the top part of the screen
  LD A,3 for output to the PRINTER.
```

The CALL &112 - a ROM routine that sets up the channels area to output to the place required. So LINE 25 stores the STACK in a space called STACKSTORE. Line 30 opens up the TOP of the screen, and LINE 40 prints the first message. LINE 50 opens up the bottom part of the screen, and LINE 60 prints the message. LINE 69 calls our "are you pressing any key" routine, and LINE 80 restores the STACK POINTER to the correct position, and RET does a return to BASIC. It is handy to store the STACK POINTER just in case your stack gets moved wrongly, or you want to return to BASIC in the middle of a machine code CALL.

Note the DATA held in MESS1 and MESS2. You must get the ROM to print AT the correct position, with the appropriate PAPER and PEN numbers.

Our PRINT routine starts by loading the A register with the contents of HL - the first byte to print. It then does the ROM routine RST 16 - this does the PRINT. After this the next address is found using INC HL, and that in turn is printed. The routine continues until byte 255 is found - this is used as an END MARKER - after which printing stops.

The PAUSE routine pokes the SYSTEM VARIABLE called LASTK with 0. It is then scanned until it changes, after which it RETurns. LASTK is used by ROM to store the last key number that was pressed.

PUTTING BYTES DIRECTLY INTO THE SCREEN MEMORY AREA

The screen lies in the TOP 2 pages - for the 256k these are PAGES 14 and 15. Depending upon which of the 4 modes you are in will dictate how much of the 32k is used, and the affect of changing any of the bytes.

In MODE 1, the SPECTRUM mode, 6192 bytes are used. The screen is divided into 3 parts, each of which has 8 lines. So lines 0-7 are in PART A, lines 8-15 in PART B, and lines 16-23 in PART C. Using BASIC lines 0-21 are in the TOP part of the screen, and lines 22 and 23 are in the BOTTOM part - normally used by INPUT.

Returning to our 6192 bytes - how are they configured? Well it is not easy to understand. The screen is a GRID of 24 lines DOWN and 32 columns ACROSS. Every LINE of the screen has 32 spaces which can be illustrated so:

```
LINE 0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
LINE 1     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
.....
LINE 23    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

This GRID of 24 lines with 32 columns needs 8 bytes of memory for each SPACE in the grid. Memory used = 24 x 32 x 8 = 6144 bytes. The remaining 768 bytes are used to control the COLOURS of the GRID.

Each X is used to indicate a PRINTING SPACE. To print in any SPACE requires 8 bytes. The first prints the TOP EIGHTH of the character, then next the second EIGHTH and so on - that is why you need EIGHT BYTES (each with 8 BITS), to make a USER DEFINED GRAPHIC.

Let us imagine that PAGE 15 is in BLOCK B from 16384 onwards. This is achieved by doing from machine code

LD A,14:OUT (250),A

provided that you are in BLOCK C or D. PORT 250 controls the BLOCKS A+B, so that doing this piece of machine code puts PAGE 14 into BLOCK A and PAGE 15 into BLOCK B. ROM 0 will no longer be available nor the HEAP area.

	BLOCK A	B	C	D
BEFORE	ROM0	PAGE0	PAGE1	PAGE2
AFTER LD A,14 OUT (250),A	PAGE14	PAGE15	PAGE1	PAGE2

Now you can POKE anything from 16384 to (16384+6144) with a BYTE and it will place a mark on the screen 8 pixels WIDE and 1 pixel DEEP - that is it prints one EIGTH of a SPACE with the bits associated with the byte you are poking.

e.g. POKE 16384,255 would print a thin BAR as all 8 bits of 255 are SET (255 = BIN 11111111)

POKE 16384,60 would print a "-" (60 = BIN 00111100)

The above can be seen using the following BASIC. We don't POKE 16384, but the memory for PAGE 14, which is 15*16384. For the 512k SAM the screen is in PAGE 30 so change the below to 31*16384.

LIST FOUR

```
10   LET START=15*16384:BORDER 7
20   LET PBYTE=255:MODE 1:CLS
30   FOR A=START TO (START+31)
40   POKE A,PBYTE
50   BEEP .03,0
60   NEXT A
```

Run this BASIC and you will see that it makes thin lines across LINE 0 of the screen. Try it again but EDIT LINE 20 making PBYTE=BIN 00111100

What you are REALLY doing is to poke the first 32 bytes of PAGE 14 with the value of PBYTE.

The same can be achieved using the following CODE which must run from BLOCK C. It puts the SCREEN PAGE into BLOCK B, and POKES 16384 onwards. It could have been put into BLOCK A using LD A,14+32 and having HL=0. 32 must be added to the PAGE NUMBER if you use BLOCK A - see later.

```
10         ORG 32768
20         DI                          ;stops the interupts.
30         LD (STORE),SP               ;stores the stack pointer
40         LD SP,50000                 ;puts stack into BLOCK C
50         LD A,7:OUT (254),A          ;border 7 (=white)
55         IN A,(250):LD (ST2),A       ;store page in BLOCK A
60         LD A,13:OUT (250),A         ;PAGE13/14 into BLOCK A/B
70         LD B,32:LD HL,16384:LD A,255
80 F1      LD (HL),A                   ;POKES 255 thirty
90         INC HL:DJNZ F1              ;two times
100        LD A,(ST2):OUT (250),A      ;PAGE 0/1 into BLOCK A/B
110        LD SP,(STORE):EI            ;Returns stack to Block B
120        RET                         ;and back to BASIC
130 ST2    DS 2
```

For the 512k SAM change LD A,29 in line 60. You must put the SAM into MODE1 before running this code. Now if you change list FOUR LINE 30 to:

FOR A=START TO (START+8*32-1)

and run the program you will see that the first 256 bytes of the PAGE contain the memory for the FIRST 8th of LINES 0 to LINE 7 - i.e. the top THIRD of the screen.
Now change line 30 to: FOR A=START TO (START+16*32-1)
and run the program. You will see again that the FIRST 256 bytes fill up the FIRST EIGHTH of LINES 0 to 7, and that the next 256 bytes fill up the SECOND eigth of LINES 0-7. So the first 256*8 bytes makes up LINES 0-7. Change line 30 to

FOR A=START TO (START+64*32-1)

and delete line 50 to see the whole of the top third completed. You can probably guess the rest. The middle lines 8-15 are controlled exactly the same way as the top third. 256 bytes print the first eighth of lines 7-15, and so on. Finally the last third, lines 16-23 follow the same pattern. Change line 30 to FOR A=START TO (START+6143)
but add a line 70 so: 70 PAUSE
to see the complete screen fill up.

Now for the last 728 bytes of the screen. These control the COLOURS of each space. The first does space 0,0 and the next 0,1 and the next 0,2 until line 0 is complete. The next is 1,0 then 1,1 and so on. The last 728 bytes give the colours of all the SPACES on the 24x32 GRID, starting at the top left and moving across the screen, one row at a time. To see this add line 70 so:

```
70 FOR A=(START+6144) TO (START+6911):POKE A,45:NEXT A
80 PAUSE
```

The 45 sets the screen to be CYAN. To select a colour you use the following formula:

PEN + 8*PAPER + 64 IF BRIGHT ON + 128 FOR FLASH

So to have pen cyan, paper yellow, with BRIGHT on use

5 + 8*6 + 64

(blue=1, red=2, magenta=3, green=4, cyan=5, yellow=6, white=7, black=0)
To make flash ON simply add 128.

To calculate the 8 OFFSET addresses associated with a point of the GRID with LINE A, COLUMN B use:

FOR LINES 0-7:	FIRST ADDRESS = 32*A + B
	2nd address = first + 256
	3rd address = first + 2*256
	8th address = first + 7*256
FOR LINES 8-15	FIRST ADDRESS = 2048 + 32*(A-8) + B
	2nd address = first + 256, etc
	8th address = first + 7*256
FOR LINES 16-23	FIRST ADDRESS = 4096 + 32*(A-16) + B
	2nd address = first + 256
	8th address = first + 7*256

These OFFSET addresses are to be added to the BASE address of the screen so:
To calculate the BASIC addresses of the 8 required for the point 9,12 you would do the following:

BASE address = 16384*(PAGE+1)
for the 256k SAM BASE = 16384*15 = 245760
but for 512k SAM BASE = 16384*31 = 507904

Now we must add the OFFSET - as the line is between 8-15 we use the middle formula with A=9 and B=12 so

OFFSET for 1st address = 2048 + 32*(9-8)+12=2092

So the FIRST address is BASE+OFFSET, which for the 256k SAM is 245760+2092 = 247852 (or 2092 bytes into PAGE 14). The remaining 7 addresses are calculated by adding on 256, then another 256, then another. i.e. 247852, 248108, 248364, etc.

Complicated isn't it!!!

All the above is for MODE 1 only. Different configurations apply for MODES 2, 3, and 4 which we won't go into. You can get the technical manual for the SAM to explain these.

HOW TO COPY THE ROMS + FURTHER EXPLANATION OF PORT 250

To copy ROM 0 all we need to do is to move 16384 bytes from address 0 to 32768 so:

```
10            ORG 50000
20 START      LD HL,0:LD DE,32768
30            LD BC,16384:LDIR:RET
40 END        NOP
50 LENGTH EQU END-START
```

Assemble this, quit to BASIC, then enter CALL 50000. You can save the rom so: SAVE "ROMO" CODE 32768,16384

Now how can we copy ROM1?

```
10            ORG 16384:PUT 32768
20 START      DI:IN A,(250):LD (STORE),A
25            SET 6,A:OUT (250),A
30            LD HL,49152:LD DE,32768
40            LD BC,16384:LDIR
50            LD A,(STORE):OUT (250),A
60            EI:RET
70 END        NOP
80 LENGTH EQU END-START
```

First more explanation of PORT 250. This controls which PAGES of memory go into BLOCKs A+B. So

LD A,4:OUT (250),A

will put PAGES 4+5 into BLOCKs A+B. In addition however if you ADD 64 (i.e SET BIT 6 of A) to the value of A, BLOCK D is changed to make it ROM 1 so:

	BLOCK	A	B	C	D
	Before	ROM 0	PAGE1	PAGE2	PAGE3
After	LD A,66 OUT (250),A	PAGE4	PAGE5	PAGE2	ROM1

Set the first 5 bits (i.e. BITS 0,1,2,3,4) of the value of the A register to decide which PAGE goes into BLOCKS A+B.

BIT 6 if set ON (or by adding 64 to the page number), makes BLOCK D have ROM 1 in it. Block C is unchanged and is only affected by port 251.

BIT 5 (or by adding 32 to the page number) if set ON makes the RAM in BLOCK A act as a ROM - i.e. you can put your own ROM into a page of memory, and then get the SAM to regard it as ROM 0 rather than SAMs ROM 0. Indeed if you EVER want to WRITE to BLOCK A, then BIT 5 must be ON and BIT 7 OFF - whether you use it as a ROM or just for running some CODE. By setting BIT 7 (adding 128 to PAGE number) as well you WRITE PROTECT the ROM just like a normal ROM, but if this bit is left OFF you can change (or POKE) the CODE. So if you put a ROM into PAGE 3 (i.e. address 65536 in BASIC), by doing LD A,163:OUT (250),A you make the following happen:

BLOCK	A	B	C	D
	PAGE3 (WRITE protected ROM)	PAGE4	same as before	

The 163 is made up so: THREE for PAGE3 + 32 TO make it ROM
+ 128 to write protect the ROM.

If you had used A=35 then the ROM can be poked. We will illustrate this further by showing how you can put the SPECTRUM ROM into the SAM and run it. To do this there is one further complication. The screen is normally in PAGE 14 for the 256k or 30 for the 512k SAM. We are going to do the following:

BLOCK	A	B	C	D
	PAGE3 (SPECCY ROM)	PAGE4 (must be screen)	PAGE5	PAGE6

Block B starts at 16384 which is where the Spectrum needs its screen. So in addition to putting the correct pages into the correct blocks, we must also tell the SAM that the screen is in PAGE4. This is easy to do as PORT 252 is used. So if we do

LD A,4:OUT (252),A

this will do the trick.

BLOCK	A	B	C	D	TV port
Start	ROM0	PAGE0	PAGE1	PAGE2	14 or 30
Step1	PAGE3 (but as ROM)	PAGE4	PAGE1	PAGE2	14 or 30
Step2	PAGE3-ROM	PAGE4	PAGE1	PAGE2	PAGE4
Step3	PAGE3-ROM	PAGE4	PAGE5	PAGE6	PAGE4

Now we can plan our m/code. We need to put 163 out of PORT 250, 4 out of PORT 252, and 5 out of PORT 251. There is still one problem as you will see.

Write the folowing code:

```
      10 ORG 50000
LIST  20 LD A,163:OUT (250),A     ;Puts PAGE3 into BLOCK A as ROM
FIVE  30 LD A,4:OUT (252),A       ;Puts PAGE4 as screen PAGE
      40 JP 15000
```

Save it as "mover1" CODE 50000,11

Now write the following code:

```
      10 ORG 15000:PUT 32768
LIST  20 LD A,4:OUT (251),A      ;Puts PAGE4/5 into BLOCK C/D
SIX   30 JP 0                    ;Restarts the computer
```

Save it as "mover2" CODE 32768,7

Switch you SAM off then on again and boot your DOS using the command BOOT 1. Enter CLEAR 49999.
Now load in your SPECTRUM ROM into 65536.
Then LOAD "mover1"CODE 50000
Then LOAD "mover2"CODE (65536+15000)
Finally CALL 50000

EXPLANATION:
First we CLEARED 49999 to protect our CODE above 50000.
Second we loaded the Speccy ROM into PAGE 3 at 65536.
Then we put our "mover1" CODE into 50000 and "mover2" CODE into 65536+15000. This "mover2" code was put into 15000 as this is a SPARE AREA in the Speccy ROM - this is VITAL for memory moving. Now when we did CALL 50000 the m/code did step1 and step2 - it made the SCREEN PAGE 4 and put PAGES 3+4 into BLOCK A+B and made BLOCK A into ROM. The SAM was doing this from address 50000 in BLOCK C.

Now comes the problem. We want to change BLOCK C+D BUT the SAM is already there at 50000 or so. That is why we put our "mover2" CODE into 15000 - so the JP 15000 causes the SAM to move from 50000 or so to 15000 which is in BLOCK A. Now we are safely in BLOCK A we can change BLOCK C+D so the CODE at 15000 puts PAGES 4+5 into BLOCKS C+D for us. This is how we achieve STEP3.

To get out of being a SPECTRUM use the RESET button on your SAM.
To change ALL 4 BLOCKS with different PAGES you should
(a) CALL M/CODE set up in BLOCK C or D
(b) this code switches pages in BLOCKS A+B and sets the TV PAGE
(c) now JUMP to some code setup in BLOCK A or B
(d) this code in BLOCK A or B switches the CODE in BLOCKS C+D.

Incidentally, the STACK is now important as you have changed all the PAGES in the BLOCKS. The SAM is set up with the STACK at around 20100 or so in PAGE 1. You will need to set a new stack somewhere. e.g. LD SP,60000

SUMMARY OF PORTS
PORT 252 - SCREEN PORT
This is used to indicate which PAGE of memory the screen is in. In a 256k SAM it is PAGE 14+15, and the 526k SAM it is 30+31.
From BASIC you can do PRINT IN(252) BAND 31
You can have a screen in a different PAGE(S). Simply do this.

```
10    LD B,NEWPAGE   ; PUT B=NEW PAGE NUMBER
15    IN A,(252)     ; GET CURRENT PORT DATA
20    AND 224        ; SETS BITS 0 TO 4 TO ZERO
30    OR B           ; PUTS BITS FROM A REG TO B
40    OUT (252),A    : SWITCH TO NEW PAGE
```

Summary of PORT 252:

BITS 0-4	PAGE NUMBER
BIT 5	First BIT of screen mode
BIT 6	Second BIT of screen mode
BIT 7	Used by MIDI channel

To the PAGE number add 0 for MODE 1, 32 for MODE 2, 64 for MODE 3, and 96 for MODE 4.

e.g. To set the SCREEN to PAGE 14 in MODE 2 do LD A,14+32

```
PORT 251 - CONTROLS BLOCKS C+D
           BITS 0-4  PAGE NUMBER
           BIT  5-7  Used in Modes 3+4 for colour settings

PORT 250 - CONTROLS BLOCKS A+B
           BITS 0-4  PAGE NUMBER
           BIT  5    When ON RAM replaces ROM 0 in BLOCK A.
           BIT  6    When ON Block D is ROM1, Block C unchanged
           BIT  7    When ON it write protects BLOCK A.

PORT 254 - Controls BORDER colour
           BITS 0-2  Border colour
           BIT  3-7  MIC, BEEP and other OUTPUTS.
```

If you do LD A,7:OUT (254),A the Border will change to colour 7.

USING THE JUMP BLOCKS

So that you can use ROM routines, the SAM was set up with JUMP BLOCKS. If you leave BLOCKS A+B alone as we suggest, then it is easy to access the ROM routines. If you put your own CODE into BLOCKS A+B then you can switch IN ROM 0 and PAGE 1 for a while to access the ROM routines, but you must be careful with the STACK, and the INTERUPT ROUTINE if interupts are enabled. We are going to assume that you leave ROM 0 and PAGE 1 in BLOCKS C+D.

e.g JCALLBAS (&10F)

If you are running some machine code in BLOCK C or D and want to CALL a BASIC routine do the following:

```
20000  ........
20010  LD HL,30     ;LOAD HL WITH THE BASIC LINE NUMBER
20020  CALL &10F    ;CALL THE JUMP BLOCK
20030  .........
```

So to goto BASIC LINE 30, load HL with 30, and do CALL &10F. To return back to your CODE at 20030 ensure that an ERROR occurs in the BASIC. e.g. STOP, or RETURN.

e.g. JCLSBL (&14E)

Clear entire screen if A register is zero, else upper screen only.

e.g. JCLSLOWER (&151)

Clears the lower screen.

e.g. JMODE (&15A)

Put the A register equal to the MODE required then do a CALL &151

e.g JKBFLUSH (&166)

Even when working a routine the keyboard can be READ and the keys STORED before being acted upon. You can CLEAR this buffer using CALL &166

We hope that the above has proved to be a useful introduction to machine code writing on the SAM. The TECHNICAL MANUAL available from SAM COMPUTERS goes into much more detail, but is short on examples and is certainly not easy to follow. We wish you well. Do explore fully the superb BASIC that SAM has - machine code isn't always necessary!

OTHER LERM PRODUCTS FOR THE SAM

SAMDISK - A superb DISK MANAGER/DOCTOR to REPAIR bad disks, a VERY FAST and EASY to use COPY/FORMAT/ERASE/HIDE/PROTECT, as well as UNERASE. There is an extra FREE "BASIC BOOT" program, and much more. It is ESSENTIAL - even if you have the new MASTERDOS!! Sold ON DISK.

SAMTAPE 3 and 4 - The main Spectrum emulator used by SAM owners. Allows 1000's of Spectrum programs to run on a SAM including utilities like Tasword, and Desk Top Publisher by PCG. Version 3 is for all SAMs with a DISK. Version 3T is for all SAMs WITHOUT a disk drive. Version 4 comes on disk and is for DISK owners who have ROM2 - it has many extra features including COMPRESSION of memory, selecting your OWN palette colours, the Spectrum COPY command works, and much more.

SAM ADDRESS and PHONE manager.
A superb program to keep track of addresses and telephone numbers - up to 5000 on a single disk. Has alphabetic sort, prints to labels, SEARCH, AMEND, and can even be used to store prices paid by customers with a product code. In lister mode it will print out on sheets of A4 all names, addresses and phone numbers. Ideal for mail shots, X-mas cards, etc.

FOR DETAILS INCLUDING PRICE, SEND A STAMPED ADDRESSED ENVELOPE TO LERM SOFTWARE, 11 BEACONSFIELD CLOSE, WHITLEY BAY, TYNE AND WEAR. NE25 9UW. TELEPHONE (091) 2533615.

WE ALSO PROVIDE AN UPDATE SERVICE.