The SAM Coupé Technical Manual

CONTENTS

Introduction

Why do we need a new Technical Manual?

All right, I know what you're thinking -- you're thinking "Why on Earth do we need a new SAM Coupé Technical Manual? Isn't the one that SAMCo/MGT gave us originally good enough?"

Unfortunately, the answer to that question is that it is ***not***.

Since the technical manual was first printed (in fact, even up to version 3 of the manual), a lot of new hardware has been produced, and that which has been documented has been documented very sparsely indeed. Bugs and tricks have been discovered on the machines (more by Demo programmers than Games programmers; the tricks tend to be more a case of "but why would I ever even consider doing that" than "Oooh, that'd go down just great in my new Asteroids/PacMan/Word-processor"). Let's face it; the machine is four years old, and it's showing its age. It's time for the "First Generation" of coders as we have been termed to share our obscure but often very useful knowledge of tricks that make people go "How have they managed to get it to do that???"

You'll find exercises at the end of every chapter to try as well – if you're an experienced programmer then you'll probably find them of no use whatsoever. However, if you're not experienced, or you just want to see if you can do them, then the exercises are going to be a lot of help (and hopefully a lot of fun for you to do too). Note: If you find a better way to do things than the one that I put in here, please feel free to be smug, and also to write to me and tell me about it!

Oh, and one last thing. If you find any errors in here, firstly they're not intentional. Secondly (the legal bit), I won't be held responsible for any damage or loss incurred (including loss of data) through the use of the information in this manual, or the programs on the accompanying disc. Thirdly, if you do find errors, *let me know!* That way I can fix them in future versions.

Finally, thanks to you for supporting the SAM and for buying this manual. It'll help keep a poor student/journalist in beer for a good few weeks at least!

Simon Cooke

Credits

Edited by
Simon Cooke.

Written by
Simon Cooke, Stefan Drissen, Bruce Gordon, Roger Hartley, Martin Rookyard, Jonathan Taylor, Steve Taylor, Frode Tennebo and Dr Andy Wright.

With Special Thanks to:
Stefan Drissen, Bruce Gordon, Mark Hall, David Ledbury, Alan Miles and Martin & Maria Rookyard.

While every effort has been made to ensure that the contents of this manual are correct in every particular, it must be appreciated that neither Entropy, Simon Cooke nor any of the contributors listed above can guarantee that the information within is definitive, as technology, and our knowledge of it is always expanding.

An E-Core Production

Accompanying Disk

There is a disk to accompany this manual, which contains handy utilities, as well as source code detailing many of the points shown here. If you have not received this disk with your manual, and would like a copy, send a cheque for £3.00 to Simon Cooke, 18 Braemar Drive, Sale, Cheshire, M33 4NJ. If however, you have received a disk and it develops an error or becomes corrupted in some way, please send a cheque for £2.00 (to cover postage & packing), with your original disk for a replacement, or send the original disk plus an S.A.E. and a cheque for fifty pence.

Each disk is checked before we send it out, and there is a copy of the verification program and logger on the disk, called "VERIFYME".

Disk Instructions

To load the interactive menu, press RESET (without the disk in the drive), then press F9 to boot the disk.

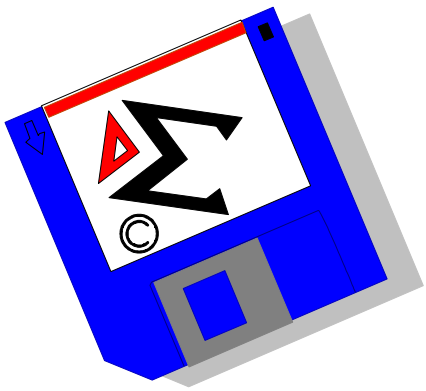
Once the menu has loaded, press the appropriate letter key to load the utility which you require.

More information on files, or the most recent updates to the disk are contained in the READ ME! option.

Disk Contents

On the disk you will find:

- ★ A copy of *Vis!*, the Entropy Hypertext Document reader.
- ★ A copy of SAM Zip and SAM Unzip – compatible with the PC's PKZIP software!
- ★ Documented source code explaining the use of many of the features / points covered in this manual. (*For **MasterDOS** users, these are in the SOURCE directory).*
- ★ Q-DOS v1.1 - this is a patched version of **SAMDOS** which also includes system and hardware identification code (as in this manual) to give a list of hardware / software present when it is loaded. It also cleans the system vectors and allocation tables. Full source code and all the files necessary to recreate it are held in the QDOS directory). Q-DOS is PD.
- ★ The SAM Coupé online directory - this is a list of all the software suppliers and hardware suppliers currently alive and working on the SAM Coupé, as well as a list of the products which they sell. This is updated whenever more information comes in.
- ★ COMET to ASCII. This is a utility to convert Comet source code to ASCII format, and vice versa. Handy for converting code to other computers in a readable format.
- ★ Stefan Drissen's DOS GRAB program. This program allows you to take files stored on MS-DOS 720k formatted disks and use them on the SAM.
- ★ Much, much more – including a demo copy of Termite – the SAM Coupé's most advanced terminal software. (Full version available from B.G. Services)

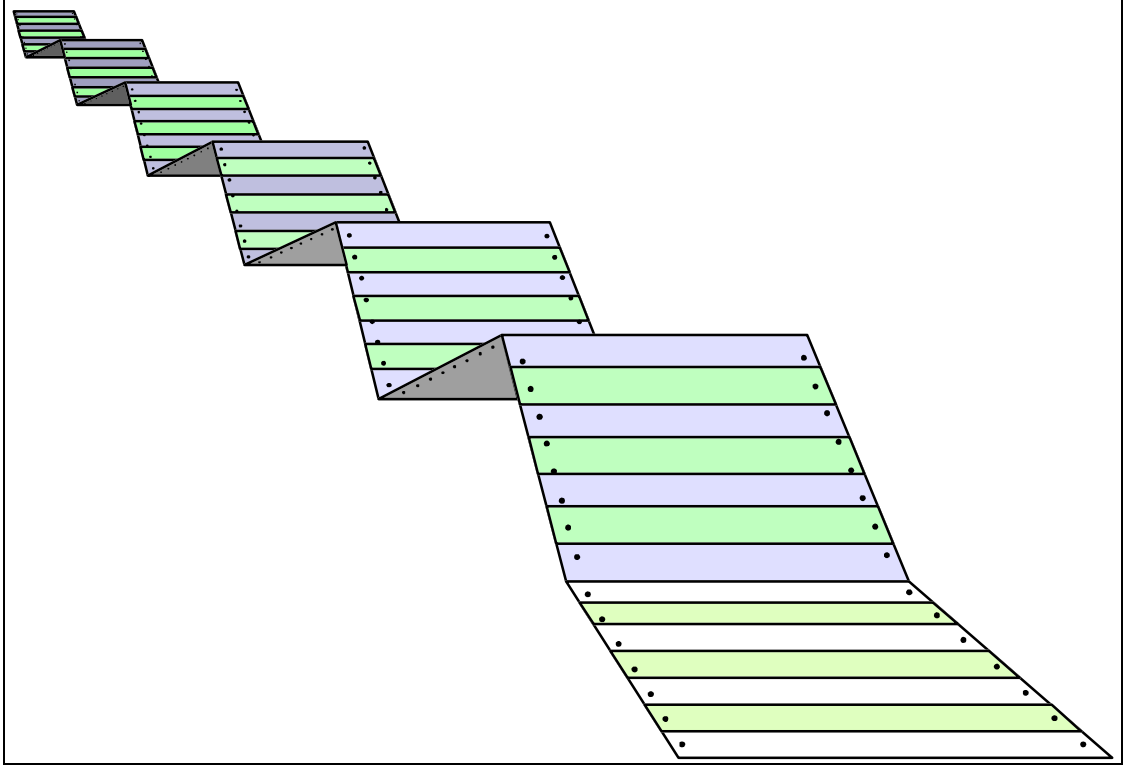


SAM Coupé Specifications

ENGINE	Z80B Microprocessor running at 6MHz.
CONTROL	Customised VLSI 10,000-gate ASIC chip.
ROM	32k including SAM BASIC, disk bootstrap, BIOS. Upgradeable to 544k using SAM MultiROM expansion.
RAM	256k upgradeable to 512k (256x4 100ns DRAM) internal memory. External memory upgradeable to 4Mb using 1Mb interface, to 1Gb using Ryan interface.
SOUND	Phillips SAA 1099 Synthesizer: 6 channel, 8 octave, stereo with envelope and amplitude control, plus wave form choice. 4-bit DAC output through software.
GRAPHICS	Motorola MC 1377P Video Chip performs conversion from RGB to composite video. ASIC serves as graphics processor / engine. All Modes allow 128 colours to be displayed on screen by redefining line interrupt: **Mode 1**: 32x24 character cells per screen, each cell with 2 colour capability; 16 colours selectable from 128; Spectrum attribute and display map compatible. **Mode 2**: as Mode 1, but 32x192 cells, each cell with 2 colour capability; 16 colours selectable from 128; contiguous display map arrangement. **Mode 3**: up to 85 column text display, 512x192 pixel screen with each pixel colour selectable; 4 colours per line selectable from 128. **Mode 4**: 256x192 pixel graphics screen; each pixel colours selectable; 16 colours per line selectable from 128.
INTERFACES	UHF TV Channel 36, through power supply unit. Colour composite video, digital and linear RGB, all through SCART. Atari-standard joystick (dual-capacity with spliiter cable). Mouse interface socket. Light-pen/Light gun, Coupé standard. Domestic cassette recorder. Midi-In, Midi-Out (Midi-Through via software switch). Network, screened microphone cable via external MGT interface to Expansion Port. 64-pin Expansion Port for further peripherals.
DISK DRIVES	1 or 2 removable (originally) and internally mounted 3.5" ultra-slim Citizen drives, 1Mb unformatted, 780k formatted. Due to problems with supplier, Citizen slim-line drives are now unavailable.
DC POWER	Power Supply 4.75 volts to 5.25 volts.
CONSUMPTION	Power consumption 11.2 Watts.
SHOCK	Operating 3 G, non-operating 60 G.
VIBRATION	Operating 5 to 500Hz / 0.5 G, non-operating 5 to 500Hz / 2 G.
ENVIRONMENT	Ambient Temperature, operating 5 to 45C, storage -20 to 50C.
HUMIDITY	Relative Humidity Wet Bulb Maximum 29.4C, nil condensation.
RELIABILITY	MTBF: 10,000 POH. MTTR: 30 Mins. Component Life: >5 years.
WEIGHT	2.26Kg = 4.97 lb.

Memory Handling

Programming within the SAM Environment



Introduction

Alright, I'm warning you now: This will *not* be a guide on how to program the Z80 microprocessor. For that, take a look at the excellent Programming the Z80 by Rodney Zaks, or take a peek at the various tutorials which have been written by people such as Steve Taylor in FRED disk mag. No, this is going to just throw a few tricks and tactics at you which I gathered were the most difficult bits to sort out, in my nine years of programming the Z80. Okay, so I'll admit it, I learned most of it from 1992 onwards (when I got a really *good* assembler – and no prizes for guessing what *THAT* was!), but it's still all very valid stuff that you'll find useful.

Optimizing Z80 code

As long as a program works, there are no good ways or bad ways of doing it. Unless you count two factors, that is. These are aesthetics and speed.

Whereas a nice looking program might please people more than a fast one, on the SAM (due to processor contention, and that it only runs at 6mhz) we really have to go for the fast option. That means using tricks you may not have considered before, so here goes…

XOR A is equivalent to LD A,0 and it also clears the carry flag. Most often it is used to set the accumulator to zero. Saves a byte of memory and 3 t-states of processor time.

OR A and AND A are equivalent to CP 0, and both reset the carry flag. Again, they both save a byte and 3 t-states.

If you need to check if bits are set, then you may like to consider using a combination of RRA's and jumps that depend on the condition of the Carry flag – eg:

```
interrupt.handler:      PUSH AF
                        IN A,(STAT)

                        RRA
                        JP NC,line.interrupt
                        RRA
                        JP NC,comms.interrupt
                        RRA
                        JP NC,midiin.interrupt
                        RRA
                        JP NC,frame.interrupt
                        RRA
                        JP NC,midiout.interrupt

                        POP AF
                        EI
                        RET
```

The code above checks the interrupt status register; if any of the bottom five bits are reset, then it jumps off to service the interrupt concerned. The code above is *much* quicker than using bit tests to do the same thing. The only thing you have to remember is that you are losing the contents of the accumulator by doing this, so you can't use the data in it again unless you keep a copy in another register or in a memory location.

Look carefully at loops – if you can afford the registers, set up data *outside* of a loop, and use it inside, rather than having to set up the data each time round the loop – you'll save a lot of time this way.

eg.

```
put.sprite:             LD IX,image.address
                        LD HL,screen.coord
                        SCF   ;turn screen coord into a screen
                        RR H  ;address (MODE 4 screen),
                              ;located between &8000
```

```
                        RR L  ;and &E000
                        LD B,image.depth
                        LD C,image.width ;in bytes
image.loop:
                        PUSH HL
                        PUSH BC
image.loop1:
                        LD A,(IX)
                        LD (HL),A
                        INC HL
                        INC IX
                        DEC C
                        JR NZ,image.loop1

                        POP BC
                        POP HL
                        LD DE,128
                        ADD HL,DE
                        DJNZ image.loop

                        RET
```

Of course, I wouldn't recommend that you use the index registers for that kind of hefty data moving, but needs must as the devil drives. Anyway, here's a better version of the same code – for long images it takes less time as not only does it not refresh DE each time round the loop, but it also takes into account the image width, and alters DE accordingly so that the PUSH and POP HL are no longer needed. (Note: this alteration of DE will be covered under *Graphics Methods and Techniques: Sprites.)*

```
put.sprite:
                        LD IX,image.addr
                        LD BC,image.width
                        LD HL,screen.coord
                        SCF
                        RR H
                        RR L
                        LD A,128
                        SUB C
                        LD E,A
                        LD D,0
image.loop:
                        PUSH BC
image.loop1:
                        LD A,(IX)
                        LD (HL),A
                        INC HL
                        INC IX
                        DEC C
                        JR NZ,image.loop1

                        ADD HL,DE
                        POP BC
                        DJNZ image.loop

                        RET
```

There's a couple of tricks up there. First of all, if you compare that code with the previous example, we've managed to get rid of a POP HL, a PUSH HL, and we've been able to take the LD DE instruction out of the loop. Now, imagine that we've got a picture 64 bytes across and 90 lines deep. By the time we've finished putting the image on the screen with the original code, we'll have made 64 PUSHes, 64 POPs, and 64 LD DE,128's. Overall, we've made a saving of about 2000 t-states! (In real terms, we've actually saved more than that, due to the freaky way in which the SAM allocates timings to instructions – see *Instruction timings.)*

2000 t-states (or, taking into account the SAM's idiosyncracies, 2300 t-states), by the way, is about a 50th of the screen display. In other words, if you changed the border colour before you entered this routine, and changed it back again afterwards, the coloured area would take up 3 raster lines or so. (This figure, I reckon, has got to be *wrong* – from experience I can tell you that it should take up more like 8 rasters).

Remember that initialising data from memory costs more than keeping it in other registers – for example:

```
LD DE,128
LD HL,128   ;20 t-states in total
```

This takes up more processor time than the alternative:

```
LD DE,128
LD H,D
LD L,E      ;18 t-states in total
```

What you shouldn't do if you just want to get a copy of another register (unless you're copying to and from the index registers) is this:

```
LD DE,128
PUSH DE
POP HL      ;a mindblowing 30 t-states
```

That's a very wasteful way of doing it indeed, and as each PUSH and POP takes 20 t-states to complete, you should steer clear of things like that where possible.

Where possible, keep everything in registers – it's must faster than using memory.

If you can afford the memory space, don't use loops – instead, repeat the block in the middle as many times as you need. For this, you'll need a good assembler to make it painless (and if any assemblers on the SAM used Macros, it'd be even less heartache to do). This'll also save you a register most times too…

In a similar vein, you can unloop LDIR instructions. Replace them with a string of LDI's. If you wanted to be really fancy, and you knew in advance how many bytes you wanted to move, you could use something like this:

```
ldir.unlooper:
                LDI
                LDI
                LDI
                …
                LDI
                JP PE,ldir.unlooper
```

If you enter this routine with BC holding the number of LDI's you want to do – in this respect it's exactly the same as LDIR – and in the routine itself you have a number of LDIs which will divide exactly into BC (otherwise your LDI routine will crash and run amok through the memory, splattering bytes wherever it goes), then it'll work. And it'll save time too. It works because whenever and LDI occurs, it increments DE and HL, and decrements BC. When it decrements BC, it sets the parity flag to even if the result is zero, hence allowing us to use BC as a counter without much overhead at all. I don't think I have to tell you that you could do this with LDD and LDDR as well – or even the looped OUT and compare instructions.

Oh yes, and you may as well not bother with unlooping your code in parts of your program which aren't time-critical (eg sprite routines), namely because the memory overhead is way too great to contend with.

There's not many more speed-ups that I can tell you about. Of course, there's something called *hard coding*, which is not for the faint of heart – it involves writing routines to

do a single, specific purpose. Good examples of this are the lemmings in Chris White's conversion of Lemmings – for speed, there's a different routine to produce each lemming on the screen. Of course, if you happen to decide that you want to change what the graphics look like later, this causes all sorts of problems… but anyway, what it basically means is that instead of doing something like this:

```
get.gfx.data:     LD A,(IX)
                  LD (HL),A
                  INC IX
                  INC L
```

You now do this instead:

```
get.gfx.data:     LD (HL),&53 ;this is hard-coded graphics data
                  INC L
```

As you can see, you save time this way, by including your data actually inside the routine itself. But it's very memory consuming, and doesn't cope well with changes in graphics.

There's one more speed-up we can do, and most computer science lecturers will tell you that you're dead wrong doing this. Oh, and you can't stick this kind of code in a ROM either. What is it? It's…

Self-modifying code

Self modifying code is tricky to work with, unless you're used to the Z80 instruction set. Even then, you can get caught out. And when you throw paging into the mix… well… you have to know what your program's up to.

What I tend to use self-modifying code for is two things really. The first of which is because I'm too lazy to set up storage space for the BASIC system. So, at the start of one of my routines, you'll often see this:

```
start:            DI    ;disable interrupts, as we're
                        ;going to be playing with LMPR
                        ;and the stack.
                  IN A,(lmpr)
                  LD (lmpr.store+1),A
                  IN A,(vmpr)
                  LD (vmpr.store+1),A
                  LD (sp.store+1),SP
                  LD A,33     ;page 1, ROM0 and ROM1 off.
                  OUT (lmpr),A
                  LD SP,alternate.stack
```

And then, somewhere else, you'll see this:

```
ret.to.basic:
                  DI
lmpr.store:       LD A,&00
                  OUT (lmpr),A
vmpr.store:       LD A,&00
                  OUT (vmpr),A
sp.store:         LD SP,&0000
                  EI
                  RET
```

In the above code fragments, what I've done is stored the contents of the LMPR actually *inside* the LD A,&00 instruction. In hex, the object code of what we've got above is this:

```
lmpr.store:       3E 00
```

```
                    D3 FA
vmpr.store:         3E 00
                    D3 FC
sp.store:           31 00 00
```

What you should be able to see is that if we change the contents of the location in memory which is lmpr.store+1, then we'll be changing it from 3E 00 to 3E NN, where NN is anything. Effectively, by changing the contents of lmpr.store+1, we can change it from LD A,0 to anything we want – and that's how the start routine saves its variables.

Of course, it's not just to save data and program space (which it does – to save a variable using something along the lines of LD (lmpr.store),A and then getting it back using LD A,(lmpr.store) then you'd need an extra two bytes – one for the data itself, and another to form the 3-byte LD A,(address) opcode. Not much, but there will be a time when you'll need it.) It can save time as well. Take this routine for example:

```
                    LD HL,screen.coord
                    SRL H       ;work out screen address for a
                    RR L        ;mode 4 screen, paged in LOW.
                    LD (scr.addr+1),HL
                    …
intensive.loop:
scr.addr:           LD HL,&0000
                    …
                    INC HL
                    LD (scr.addr+1),HL
                    …
                    DJNZ intensive.loop
```

Okay, so that's not so much an example as a lot of teeny fragments, but it'll do for our purposes. Now, if you're short on registers, using self-modifying code as above can save you a lot of t-states. First of all, the actual place you store the data is in the part of the program where the speed-up is needed the most. I don't know where this will be – it's very program-specific – but it's a good guess that it'll be in an output routine of some sort. By storing the data in the opcode itself, you save time that would be otherwise used by fetching a two-byte address, getting the contents of that address and putting it in L, incrementing the address and then getting the contents of that *new* address and putting it in H – which is effectively what the processor does every time you do a LD HL,(address). Every time you need to use that data elsewhere, you can either copy it into *another* self-modifying instruction (unlikely, but you can do it if you need to use the same data in lots of places and you need the speed), or you just refer to it by, eg, LD (scr.addr+1),HL.

It may take a while to get used to this way of working, but it can be worth it in the long run. It's something that you'll get more and more practised in as time goes on. For more examples of self-modifying code, see *Keeping BASIC happy.*

Instruction timings

Interrupt routines

Keeping BASIC happy

Fast Multiplication and Division

Graphics Methods And Techniques

The Graphics Hardware

The Video Memory Paging Register (VMPR)

VMPR	7	6	5	4	3	2	1	0
&FC	MIDI	MODE 1	MODE 0	PAGE16	PAGE 8	PAGE 4	PAGE 2	PAGE 1

The VMPR (Video Memory paging register) is used to select the current screen mode, and also to control which video page the ASIC looks at when it needs to display screen memory. There is also an auxiliary Midi function which is controlled by bit 7 of the register (see *Midi*).

Bits 5 and 6 of the register control the current screen mode;

```
Bit 6 Bit 5
0     0     Mode 1
0     1     Mode 2
1     0     Mode 3
1     1     Mode 4
```

Bits 4 to 0 select the screen page. In modes 1 and 2, this can be any value. However, in screen modes 3 and 4, the ASIC looks at the nearest even page (ie in mode 3/4, the ASIC treats bit 0 as if it were reset). Note: Screens may only be displayed from internal memory; the ASIC has no provision for displaying a screen contained in external or other memory (such as the MultiROM's private area).

The VMPR may be written to at any time; you are not limited in any way as to where the change occurs. This may be used to offer split-mode screens (as in **MasterBASIC**) or to change the screen page being displayed part way along the screen. (See *Split-Mode screen displays*)

The CLUT (Colour Look-Up Table)

The colour look-up table (CLUT) is a device used to facilitate easy colour changes, and also to minimize the amount of screen memory required to display a given colour or range of colours on the screen. It is written to using register &0nF8, where ‘n’ is the colour to change, and &F8 is the base address of the CLUT.

It would actually be more accurate to say that the top 4 bits of the 8-bit CLUT address are ignored; for example, writing to port &FFF8 writes to CLUT position 15, even though you might expect it to try (and fail) to write to CLUT position 255.

The format of the CLUT registers are as follows:

CLUT	7	6	5	4	3	2	1	0
&F8		GREEN1	RED 1	BLUE 1	BRIGHT	GREEN0	RED 0	BLUE 0

Bits 6 and 2 control how much green a colour contains, bits 5 and 1 control how much red it contains, and bits 4 and 0 control how much blue. Bit 3 is the bright control; if it is set, it rougly corresponds to adding 0.5 to **all** of the colours.

It may be easier to understand if the colours are thought of as having eight values:

Setting RED1, GRN1 or BLU1 adds 4 units of brightness to that colour. Setting RED0, GRN0 or BLU0 will add 2 units of brightness to the respective colour. Setting the BRGT bit will add 1 unit of brightness to ALL of the colours. In the above diagram, the BRGT bit SET is represented as a shaded area.

For example, if we were to set RED1, GRN1, GRN0 and BLU0 we would effectively have a brightness of 4 for red, 6 for green and 2 for blue. If we were then to set BRGT, these values would change to give a brightness of 5 for red, 7 for green and 3 for blue.

This way of representing the SAM's colour system will be used later. (see *Colour Fading; Representing 8-bit R,G,B on the SAM; Representing the SAM's palette in 8-bit R,G,B; Luminance)*

To output to a specific CLUT register, a 16-bit port address must be written to. The high byte of this determines which register is written to, and the low byte signifies a write to the CLUT (i.e. it must contain &F8).

e.g.

```
write.to.clut3:
                    LD BC,&03F8
                    LD A,colour ;this writes colour to CLUT
                    OUT (C),A   ;position 3
```

If it is necessary to write an entire palette to the CLUT, this may be done using the OTDR instruction. B holds &10, which is the number of colours (16), and HL must point to the end of the palette data (the 16th colour).

```
write.palette:
                    LD BC,&10F8
                    LD HL,palette+&0F
                    OTDR        ;Set up the palette
```

When the OTDR instruction is encountered, B is decremented (to point to CLUT15). The contents of the memory address pointed to by HL are output to this CLUT register, then HL is decremented. If B has not reached zero, it loops around and continues sending data to the CLUT.

The CLUT is a write-only set of registers, so it is necessary to keep a copy of this if you are likely to need to look at the contents of these registers. Basic does this for you (see *Using the palette in BASIC*).

On reset, the CLUT is not affected. (see *Reset screens and how to generate them*)

Screen Memory Mapping

The way that the contents of the SAM's memory relates to what you see on the screen is known as "Screen Memory Mapping". The way that the memory is mapped to the screen varies depending on the screen mode. Modes 1 and 2 are the most difficult to follow, so we shall deal with them last.

Note: Although BASIC tends to number lines with the bottom line on the screen being zero, and the top line being, for example, 191, in machine code it is much easier to work the other way around.

Mode 4 (16 Colour Mode)

This mode gives a screen 256 pixels wide by 192 pixels deep, in sixteen colours. It is the start-up mode of the SAM Coupé, and it is organised as follows:

X co-ordinate

Screen Y address Offset	0	1	2	3	Y co-ordinate
&0000	0000	0001	0010	0011	0
&0080	0100	0101	0110	0111	1
&0100	1000	1001	1010	1011	2
&0180	1100	1101	1110	1111	3
	00		01		

Screen X address offset

In the above example it is easy to see how the memory corresponds to the pixels displayed. For example, the pixel at co-ordinate 0,0 is addressed as follows:

Y address offset: &0000
X address offset: &00
Address: &0000 + &00 = &000

From the diagram, pixel 0,0 corresponds to the left hand nybble of the byte at address &0000.

To find a Y-Offset, take the Y co-ordinate and multiply by 128. To find an X-offset, take the X co-ordinate and divide it by 2, discarding the remainder. Add these together to give you the address in a page which contains the mode 4 byte.

Note: A nybble can contain data with values from 0 - 15; this gives 16 colours in all.

If the X Co-ordinate is originally even, then the CLUT value for the pixel may be found in the left hand nybble; otherwise, if the co-ordinate is odd, then the pixel may be found in the right-hand nybble.

Border Colours and turning the screen off

Border colours and screen on/off status are controlled by the border register (&FE). It is analogous with that of the Spectrum, except where the Spectrum could only display 8 colours (all dimmed), the SAM can display any of the 16 colours in its CLUT in the border area.

BORDER (&FE)			CLUT8			CLUT4	CLUT2	CLUT1

The border colour stems from current contents of the CLUT; the current MODE has no effect whatsoever on it. It is also READ ONLY, so if the border colour needs to be known, a copy of it should be kept whenever it is changed.

BORDER (&FE)	SOFF							

The Screen Off status may be read by reading the border register and checking bit 7, and it may be altered by writing to the border register with bit 7 set to turn the screen off, and bit 7 reset to enable the screen display circuitry.

The ability to turn the screen off is only available in screen modes 3 and 4; in modes 1 and 2 the screen off bit has no effect. It can be used to regain some of the time lost due to video memory contention; **COMET Assembler** turns the screen off during assembly in order to complete the task as quickly as possible. It is important to note that although the LINE and FRAME interrupts will still occur while the screen is off, the HPEN (YSCAN) and LPEN (XSCAN) registers will not update at all so it is not possible to use them for synchronisation in this instance. If the screen has to be restored at a given line, it is necessary to set up the LINE interrupt with the line to restore the screen at, and either to poll the status port (if interrupts are disabled) or to wait for the interrupt to occur. What actually happens when the screen is turned of is that the video circuitry which reads the screen memory is cut out of the system, and no colour (black) is sent out on the RGB lines. This ensures that no screen corruption or flicker occurs due to video sync pulses being sent at the wrong time; the sync pulses continue to be sent, it is just the RGB output which is stopped.

When the border colour register is written to, it will not change until the next screen memory "sync" occurs. It is unknown whether this is due to an I/O wait condition or a memory refresh/video contention wait state. The screen off condition is activated one pixel clock pulse after this in the border area of the screen, and sometimes 8-12 screen pixels later when the screen off occurs inside the memory-mapped screen area. (See *Graphics in the border Area, Hardware Oddities*)

Reset Screens (And how to generate them)

If any of you have seen demos such as **ESI's Surprise**, **The Lyra 3** or **Entropy's Entro 2** (as well as a multitude of others), and on pressing reset have been surprised at the appearance of a picture or message on your screen, then you'll know what I'm talking about.

Reset screens are generated when the reset button is pressed (or on power-up) because the SAM ASIC sets most of its internal registers to zero. When the VMPR is set to zero, it can be seen that what should be displayed is a Mode 1 screen, found in page zero of the memory map. As the CLUT is not altered by reset, the picture will assume whatever palette was set up before the reset occured. The screen remains displayed because the memory refresh and video circuitry in the ASIC are allowed to run constantly when the RESET line goes low.

With careful planning, spectacular and colourful effects can be produced -- the Reset screen has even been used to display a winking SAM robot!

To install a RESET screen (sample code):

```
            ORG HIGH    ;this code is assumed to be paged in high

set.reset:
            LD A,32     ;page 0, with RAM in section A
            OUT (LMPR),A
            LD HL,reset.screen
            LD DE,&0000
            LD BC,6912
            LDIR
```

And that is really all there is to it. Note: Reset screens overwrite the BASIC system area, and so are not feasible for use with BASIC programs. If reset screens are needed for menu programs, it is usually a good idea to copy the 6912 bytes which will be overwritten to a storage space for safe-keeping, and then to replace it when a return to BASIC has to be made.

Hardware Oddities

While the screen is turned off, both vertical and horizontal synchronisation pulses are created by the ASIC and fed through to the composite video generation circuitry.

It is generally known that the video circuitry can affect the speed at which the processor goes about its business. What is less well known is why.

When Bruce Gordon designed the ASIC, Mode 1 was included so that Spectrum software could be run with a minimum of hassle and overheads. As the SAM's standard processor (the Z80B) runs at 6MHz as opposed to the 3MHz Z80A in the Spectrum, it was necessary for Bruce to include extra delays in the ASIC to make the SAM run at approximately Spectrum speeds. Even though the SAM still runs roughly 10-20% faster than the Spectrum in Mode 1, there are extra delays inherent in this mode.

Although it is hard to test, from results gained by experimenting with border pixel routines, it appears that Mode 2 operates slower than Mode 3 or 4 as well, but not to the same degree as Mode 1.

When writing border pixel generation routines, it is interesting to note that writes to the border register and to the CLUT in the border area are automatically synchronised to the display itself. In modes 1 and 2, this synchronisation appears erratic, but it is in fact stable (experiment with this yourself!). In modes 3 and 4, colour changes are synchronised to approximately the nearest block of 8 pixels (mode 4 size), even though it is not possible to change colours faster than in 16 pixel blocks. The use of, for example, LD R,A instruction to delay for a block of 8 pixels, may be used to give smooth border scrollies. (See *Graphics in the Border area*)

Representing 8 bit R, G, B values on the SAM

```
;Convert N-bit palette to SAM palette
```

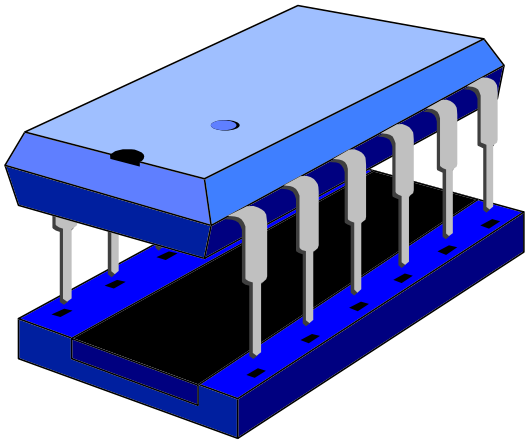
```
;-----------------------------------
;Entered with HL = SAM Palette to output
;             IX = RGB Palette input
;              C = Bits per rgb byte
;              B = Number of palette entries

;RGB palette is stored as R,G,B,0

;Rotates RGB values until their MSb is at bit 7, then masks off top 3
;bits bit 5 then becomes a BRIGHT counter (if 2 or more are set,
;bright is used)

;Falls over on palettes with less than 3 bits per RGB, but they're
;not that common. Will still work, but colors won't be exactly
;correct.

;Copyright (c) 1994 Simon Cooke

rgb_to_sam:
            PUSH BC

            LD A,8
            SUB C

            LD D,(IX+0)
            LD E,(IX+1)
            LD C,(IX+2)

            OR A
adjust_rgb:
            JR Z,end_adj

            SLL D
            SLL E
            SLL C
            DEC A
            JR adjust_rgb

end_adj:
            LD B,0            ;calculate half-bright bit

            BIT 5,D
            JR Z,notbrite1

            INC B
notbrite1:
            BIT 5,E
            JR Z,notbrite2

            INC B
notbrite2:
            BIT 5,C
            JR Z,notbrite3

            INC B
notbrite3:
            AND %00000010 ;mask bit 1 - if this is set, color has
                          ;BRIGHT
            RLCA
            RLCA          ;bright is now in right place
                          ;now convert the rest of the data
            BIT 7,D
            JR Z,no_rh

            OR %00100000
no_rh:
```

```
            BIT 6,D
            JR Z,no_rl

            OR %00000010
no_rl:
            BIT 7,E
            JR Z,no_gh

            OR %01000000
no_gh:
            BIT 6,E
            JR Z,no_gl

            OR %00000100
no_gl:
            BIT 7,C
            JR Z,no_bh

            OR %00010000
no_bh:
            BIT 6,C
            JR Z,no_bl

            OR %00000001
no_bl:                  ;finished converting palette to SAM format
            LD (HL),A
            INC HL
            LD DE,3
            ADD IX,DE
            POP BC
            DJNZ rgb_to_sam

            RET

;Converts raw SAM palette data to 24 bits...
;Enters: A=SAM Palette number
;Exits:  C=RED 8bit, D=GRN 8bit, E=BLU 8bit

; SAM PALETTE: 76543210
;              xGRBHgrb

expand_sam:
            PUSH HL
            PUSH AF
bright:
            AND %00001000
            RRCA
            RRCA
            RRCA
            LD C,A
            LD E,A
            LD D,A            ;BRIGHT=bit 0 of all...
            POP AF
red:
            BIT  5,A
            JR Z,not_hired

            SET  2,C
not_hired:
            BIT 1,A
            JR Z,blue

            SET 1,C
blue:
            BIT 4,A
            JR Z,not_hiblu
```

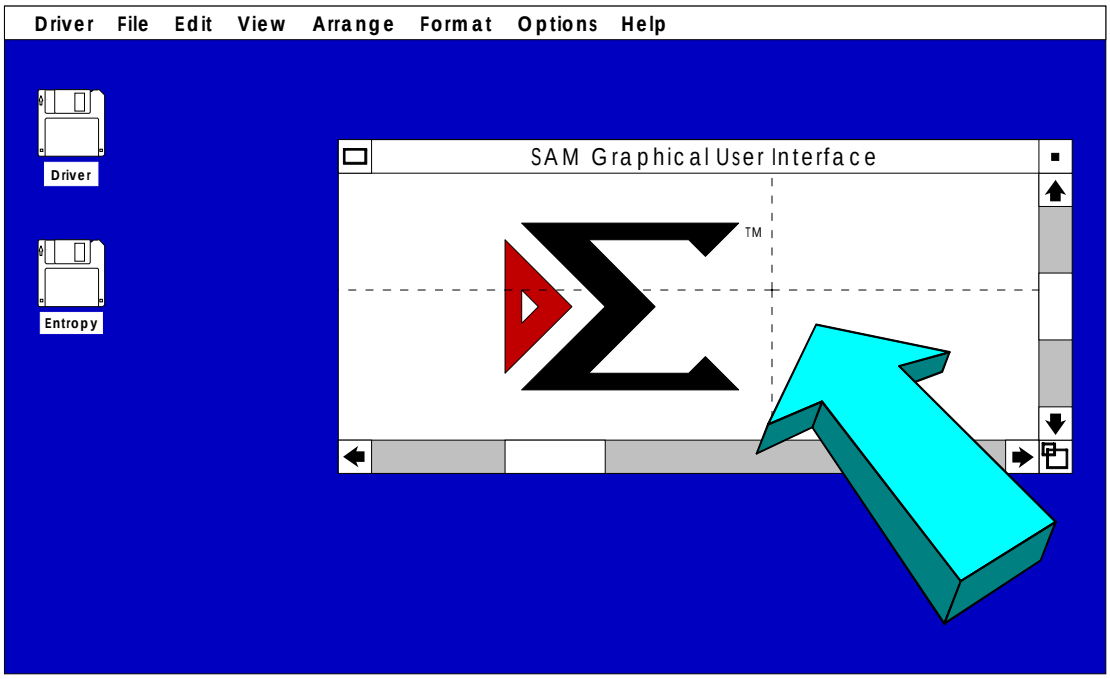
```
            SET 2,E
not_hiblu:
            BIT 0,A
            JR Z,green

            SET 1,E
green:
            BIT 6,A
            JR Z,not_higrn

            SET 2,D
not_higrn:
            BIT 2,A
            JR Z,expand_rgb

            SET 1,D

;We now have 3 separate RGB values, at 3 bits a piece.
;Turn these into full 8-bit values using a table.

expand_rgb:
            LD L,C
            LD H,0
            PUSH DE
            LD DE,rgb_spread
            ADD HL,DE
            POP DE
            LD C,(HL)

            LD L,D
            LD H,0
            PUSH DE
            LD DE,rgb_spread
            ADD HL,DE
            POP DE
            LD D,(HL)

            LD L,E
            LD H,0
            PUSH DE
            LD DE,rgb_spread
            ADD HL,DE
            POP DE
            LD E,(HL)

            POP HL
            RET

;Conversion table for SAM rgb to 8-bit.

rgb_spread: DEFB 0,36,73,109,146,182,219,255
```

Sound Methods And Techniques

SAM BASIC, Master BASIC and the SAM Coupé BIOS

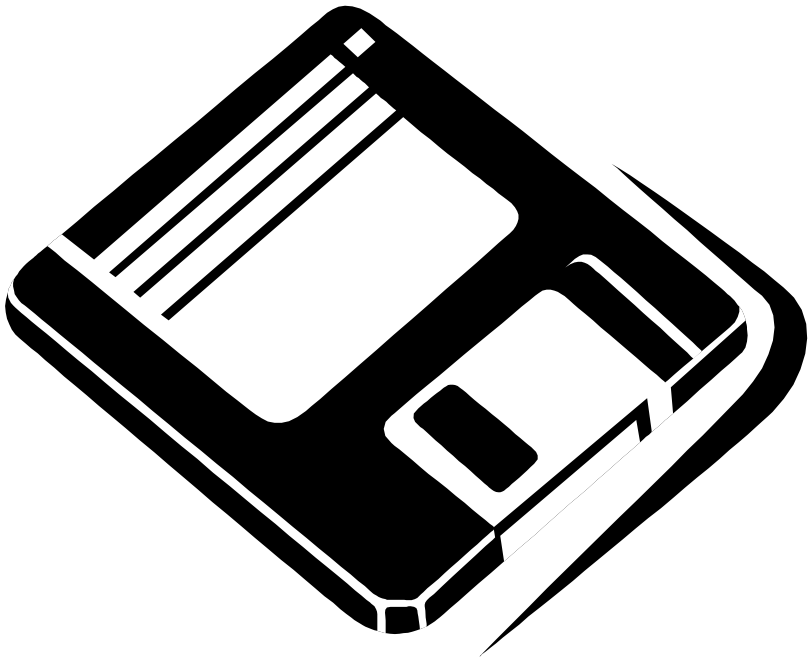


SAM Graphical User Interface



Driver

DOS, Disks And Disk Controllers



ADDRESS SELECTOR		STATUS (&F9)			BORDER (&FE)				
		7	6	5	4	3	2	1	0
11111110	FE	F 3	F 2	F 1	V	C	X	Z	SHIFT
11111101	FD	F 6	F 5	F 4	G	F	D	S	A
11111011	FB	F 9	F 8	F 7	T	R	E	W	Q
11110111	F7	CAPS	TAB	ESC	5	4	3	2	1
11101111	EF	DELETE	+	-	6	7	8	9	0
11011111	DF	F 0	"	=	Y	U	I	O	P
10111111	BF	EDIT	:	;	H	J	K	L	RETURN
01111111	7F	INV	.	,	B	N	M	SYMBOL	SPACE
11111111	FF				RIGHT	LEFT	UP	DOWN	CNTRL

Keyboard port address map

MasterDOS Hook Codes

MasterDOS provides hook (command) codes which enable the machine code programmer to use the DOS's facilities without having to return to or call SAM Basic.

If an error occurs, MasterDOS puts an error number into the A register; otherwise the A register will contain zero on return.

Hook codes are accessed by an RST &08 instruction followed by the operation code. eg.

```
load.data:  RST &08     ;hook code
            DEFB HLOAD  ;HLOAD operation code
            ;program continues here
```

INIT 128 (&80) Look for an AUTO file on the current disk. No action (or error) occurs if there is no AUTO file, otherwise it is loaded (and executed if it is an auto-running Basic or CODE file). This Hook can only be used in sections B and C of the memory map.

HGTHD 129 (&81) Get file header. This routine should be called with IX pointing to the UIFA, which should hold the file type required (at IX+0) and the file name (at IX+1 to IX+10). The routine looks for the file in the current directory on the current drive and either returns with an error code, or transfers the data from the file directory to IX+80, in UIFA form. The calling code and the UIFA can be in sections B, C or D of the memory map. (Note: this hook works correctly in SAMDOS, provided that IX=&4B00.)

HVERY 131 (&83) Like HLOAD, but verify the data on the disk against the data in memory. Error code 93 returned if verify failed.

HSAVE 132 (&84) Save the file whose UIFA is pointed to by IX. All relevant data in the UIFA must be complete - for a CODE file, type, name, start, length and execute address. If in doubt, try a SAVE from BASIC and then look at &4B00-&4B47 to find the required values.

HSKSF 133 (&85) Seek Safe. On some machines, pressing the Reset button can corrupt the disk sector under the drive head. This is often on the track containing the last sector of the last file loaded. MasterDOS tries to minimise the problem by parking the drive head on the last track in the directory, after a LOAD or a SAVE. This track will be unused unless the directory is fairly full. Using the HSKSF hook will move the head of the current drive to the last track in the directory, unless this would be track 4 (which contains the first sector of DOS) in which case track 3 is used instead.

HAUTO 136 (&88) Like Hook 128, but an error code of 101 is returned if there is no AUTO file.

HSKTD 137 (&89) Seek Track D. Move the drive head of the current drive to the track specified in the D register.

HFMTK 138 (&8A) Format Track. Format the track under the drive head, using the D register to supply the track number and the E register as the number of the first sector (1-10). Later sectors will be numbered 1 higher until 10 is reached and numbering goes back to 1.

HVAR 139 (&8B) Supply the address of a DVAR by putting it on the floating point calculator stack. On entry, the FPCS should hold the desired DVAR number. Note: it is probably easier to page in DOS (the DOS page is held at &5BC2) and read the disk variables directly. DVAR 0 is at an offset of &0220 within the page - this will not change.

HEOF 140 (&8C) Supply the End-Of-File status (1 or 0) of a specified stream. The stream number should be on the FPCS. It will be replaced by the EOF status.

HPTR 141 (&8D) Supply the PTR value for a specified stream. The stream number on the FPCS is replaced by the PTR value.

HPATH 142 (&8E) Supply the current PATH$ on the FPCS. Use CALL &0124 (JSTKFETCH) to get page (A) offset (DE) and length (BC) of the string.

HLDPG 143 (&8F) As Hook 130, but on entry the A register should hold the page number of the destination address. This need not be paged in.

HVEPG 144 (&90) As Hook 131, but on entry the A register holds the page to verify against.

HSDIR 145 (&91) Select Directory. Similar to DIR="name" in Basic. On entry, the registers hold details of the location and length of the desired subdirectory name. DE is the offset, A is the page of the name start, and BC is the name length.

HOFSM 146 (&92) Open a File Sector Map for an OPENTYPE file. IX must point to the UIFA. The routine will create the map and clear the disk buffer.

HOFLE 147 (&93) Open a file on the disk. IX must point to the UIFA. The routine will create a sector address map, and save a 9-byte header to the disk buffer.

HSBYT 148 (&94) Save the byte in the A register to the disk file (If the buffer is full it will be written to the disk and the byte will go into the start of the next buffer).

HWSAD 149 (&95) Write Single Sector. On entry, the A register is the drive number (1-7) which is used to access the table at DVAR 111 to get the actual drive to use. D holds the destination track, and E the sector number. HL points to the source in memory, which must be in sections B, C or D of the memory map. 512 bytes will be written to disk.

HKSB 150 (&96) Save a blockof data to the disk file. The A register holds the length to save in pages, and DE holds the length MOD 16K. HL points to the start of the data to save, paged into section C of the memory map.

HDBOP 151 (&97) Save BC bytes to the disk file. DE points to the start of the data to save, paged into section C of the memory map. Used by DOS to write strings to OPENTYPE files.

HCFSM 152 (&98) Close a file. This routine writes the last buffer to a disk file and creates a directory entry for it. IX should point to the UIFA.

HORDER 153 (&99) Sort list into ASCII order. HL should point to the start of the list in sections B, C or D of the memory map. The BC register should hold the length of each item in the list, and the DE register the number of items. The A register

specifies the number of characters to sort on. No paging is performed so the entire list must be paged in by the user before this hook is called.

HGFLE 154 (&9E) Get a file from the disk. The IX register must point to the UIFA. The return is made with the first sector of the file loaded into the disk buffer and RPT pointing to the first byte.

HRSAD 155 (&A0) Read Single Sector. On entry, the A register is the drive number (1-7) which is used to access the table at DVAR 111 to get the actual drive to use. D holds the source track, and E the sector. HL poitns to the destination in memory, which must be in sections B, C or D of the memory map. 512 bytes will be read from the disk.

HLDBK 161 (&A1) Load a block of data from the current disk file. HL points to the destination of the data in memory, paged into section C of the memory map. The A register is the length to load, in pages, and DE holds the length MOD 16K.

HMRSAD 162 (&A2) Read Multiple Sectors. Equivalent to READ AT in Basic. The A register is the drive to user (1-7 using DVAR 111 table), D holds the track, E the sector, C the page and HL the offset (&8000-&BFFF) of the destination. IX holds the number of sectors to load.

HMWSAD 163 (&A3) Write Multiple Sectors. Equivalent to WRITE AT in Basic. As above, but C and HL hold the source address, rather than the destination.

HREST 164 (&A4) Restore. Move drive head to track 0. The disk need not be formatted.

HPDIR 165 (&A5) Print directory. If the A register holds 2, print a simple directory. If it holds 4, print a detailed directory. Neither option does a CLS first. The current stream is used to output.

HERAZ 166 (&A6) ERASE a file from disk. The file name should be at IX+1 to IX+10.

HCHRD 168 (&A8) Read character from the disk file whose UIFA is pointed to by IX. The character and flags are passed out in the alternate BC register: EXX, PUSH BC, EXX, POP AF gives the character in A, and the carry flag set if the read was OK, else we hit the end of file.

(Information provided by Dr. Andy Wright)

PicView SuperSCREEN$ File Format

PicView is a file format devised by Entropy member Colin Piggot. It caters for screens bigger than the standard SAM screen size.

PICVIEW SUPER SCREEN$ FILE

128 byte header:

```
000-015 (ASCII)   "PICVIEW: (C) CGP:"     <PV Identification>
016-019 (ASCII)   "PVS$"                   <Picture type>
020-021 (WORD)    WIDTH                    <Image Width (Pixels)>
022-023 (WORD)    HEIGHT                   <Image Height (Pixels)>
024-039 (BYTE)    PALETTE                  <Palette (colour) value for
                                           each of the 16 colours that
                                           may be displayed>
040-059 (ASCII)   NAME                     <Image Name>
060-079 (ASCII)   NOTES                    <Image Notes>
080-099 (ASCII)   PNOTES                   <Palette Notes>
100-102 (MIXED)   RVERSION                 <Version     of     PicView
required
                                           to display picture>
103-105 (MIXED)   MVERSION                 <Version of PicView which
                                           created this picture>
106-127           RESERVED FOR FUTURE EXPANSION

128-End of file   Graphics data (as for Mode 4 SAM Screen, length
                  (WIDTH+1) DIV 2 * (HEIGHT*128))
```

Bytes 100-102 provides information detailing which version of PicView is required to display the PVS$ file, whereas bytes 103-105 detail which version of PicView created the PVS$ file.

The version numbers are stored as a byte giving the main version number (0-255), followed by the ASCII period character ("."), followed by a byte giving the part or minor version number (0-255). For example, revision 10.23 of the PicView software would be stored as a string of bytes, value 10, 46, 23.

To determine whether a file is in PicView format or not, and if your software should be able to display it, check that bytes 16-19 hold "PVS$" -- if they do not, then you are not looking at a PicView SuperScreen$ file. Also, ensure that the RVERSION number is not higher than your software can cope with. The above PicView file format information is correct for version 2.0; your software may assume that files created with version 2.0 or less (that means that byte 100 of the header contains 0, 1 or 2, byte 101 contains 46 and byte 102 contains zero) will be compatible with the information above.

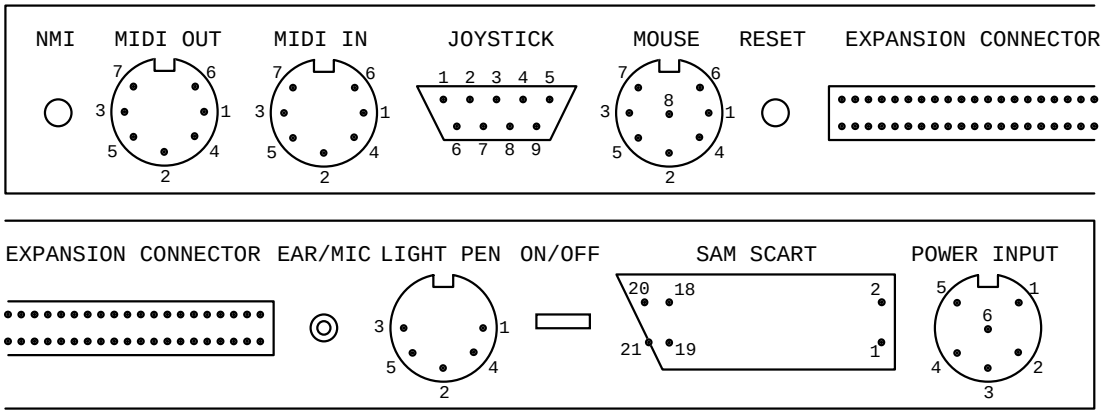
(Information provided by Colin Piggot)

SAM Coupé ASCII Code Table

The SAM Coupé uses a slight variation of ASCII as its standard method of encoding text. This stems from the ZX Spectrum, which used certain codes to change colours and other attributes, including moving the PRINT position (AT). Another addition was the use of character code 127 (the DEL control code in ASCII, the greek letter Delta on the PC) as a copyright symbol.

Mouse Driver Source Code

Back Panel Help Sheet



MIDI IN

1 Net - Loop
2 No Connection
3 Net + Loop
4 Midi + In
5 Midi - In
6 Net - Loop
7 Net + Loop

MIDI OUT

1 Net - Loop
2 GND
3 Net + Loop
4 Midi + Out
5 Midi - Out
6 Net - Loop
7 Net + Loop

ATARI JOYSTICK

1 Up
2 Down
3 Left
4 Right
5 0 Volts
6 Fire
7 +5 Volts
8 STROBE1 (active high)
9 STROBE2 (active high)

MOUSE

1 Down
2 Up
3 CNTRL
4 Left
5 Right
6 Int.
7 RDMSEL
8 +5V
Scrn - GND

LIGHT PEN

1 +5 Volts
2 Audio Left
3 0 Volts
4 SPEN Input
5 Audio Right

SAM SCART

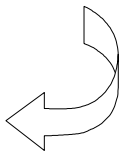
1 Audio Right
2 SPEN Input
3 Audio Left
4 Audio Ground
5 Blue Earth
6 Blue TTL Out
7 Blue Lin Out
8 Red TTL Out
9 Green Earth
10 Green TTL Out
11 Green Lin Out
12 +5v Power In
13 Red Earth
14 CSYNC Earth
15 Red Lin Out
16 CSYNC
17 Cmp Vid Earth
18 +12v Power In
19 Cmp Vid Out
20 Bright TTL Out
21 GND

POWER INPUT

1 +5 Volts
2 0v Signal GND
3 0v Digitl GND
4 Comp. Vid O/P
5 +12 Volts
6 Sound Output (mono)

Euroconnector (Expansion Connector)

Rear View Of SAM



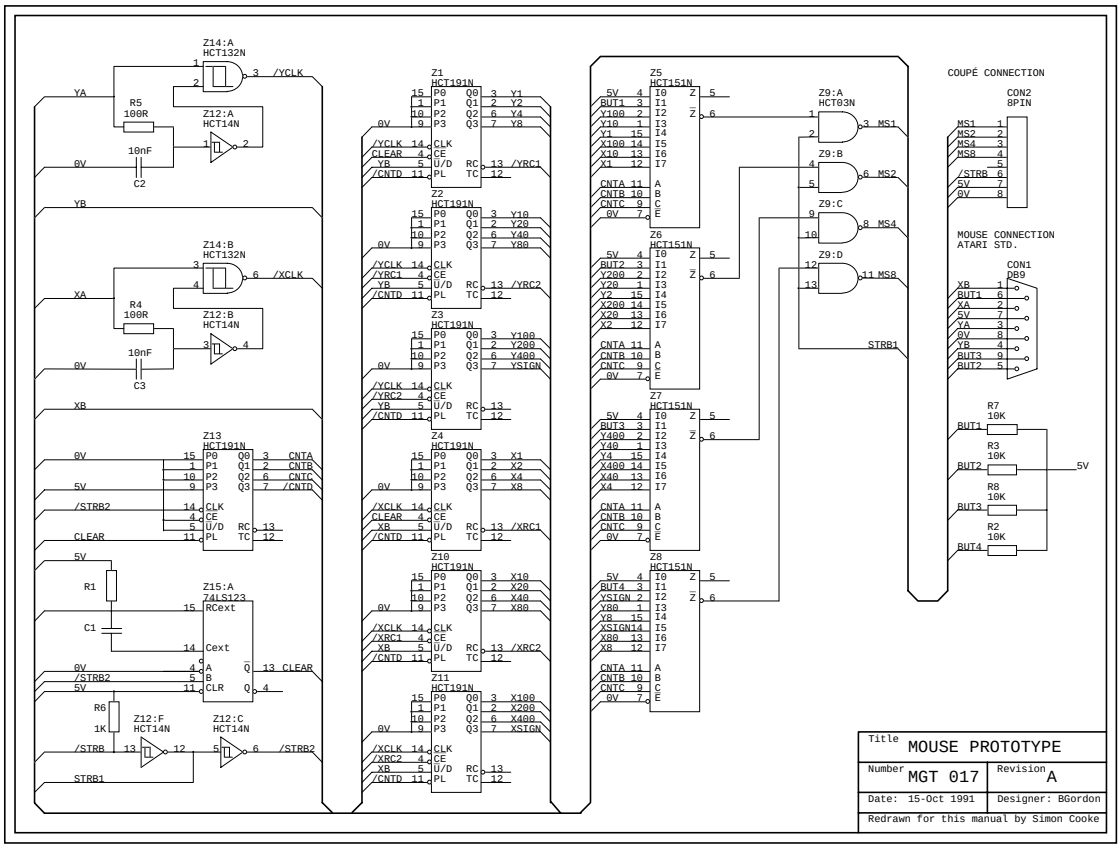
	A	C	
DBDIRL	1		IORQL
RDL	2		MREQL
WRL	3		HALTL
BUSACKL	4		NMIL
WAITL	5		INTL
BUSREQL	6		CD1
RESETL	7		CD0
CM1L	8		CD7
REFRESHL	9		CD2
0 VOLTS*	10		+5 VOLTS*
A0	11		CD6
A1	12		CD5
A2	13		CD3
A3	14		CD4
A4	15		CPU CLK
A5	16		A15
A6	17		A14
A7	18		A13
A8	19		A12
A9	20		A11
A10	21		DISC 2L
EXTINTL	22		ROMCSL
XMEML	23		EARMIC
8 MHz	24		DISC 1L
RED 1	25		PRINTL
GREEN 1	26		BLUE 1
CSYNCL	27		ROMCSRL
SPEN	28		AUDIO RIGHT
BLUE 0	29		AUDIO LEFT
RED 0	30		COMPOSITE VID
BRIGHT	31		GREEN 0
+5 VOLTS	32		0 VOLTS

Pin out of Expansion Connector

*Not to be used as a supply rail - reference only.

The SAM Coupé edge connector is a standard Euroconnector socket with rows A-C fitted. All identifiers ending with L are active low.

SAM Mouse Interface (Schematic)



SAM Hardware And Peripherals