ReSource 2020

SOME INFORMATION ON THE MTX THAT I WISH I'D HAD IN THE 1980'S….

0 CONTENTS

1 INTRODUCTION

Over its lifetime the MTX was supplied with 2 different manuals, the original one was adequate, but only just. It suffered a little in that although there was an introduction to BASIC and Noddy, and a technical section at the back with lots of info on the hardware and component parts, there wasn't a lot in between the two.

The 2nd manual had a lot more useful detail for the BASIC programmer but was still short of covering the gap between BASIC and the technical appendices.

So back in 1987 Keith Hook wrote a book called "The Source" about some of the more technical aspects of the MTX series of computers, to bridge the gap between the sections in the manual(s).

While it may not be the best book ever written, it is one of the few available for the MTX. During the development of the MTX plus project with Dave Stevenson, Dave provided me with a copy of the book. I found some information of use for the project, I also found some of the information to be presented in a confusing manner, of in a couple of spots just plain wrong.

This therefore is my attempt at correcting some of the errors and presenting things in a way that makes more sense to me. I assume that anyone reading this has at least a basic Knowledge of the MTX and the Z80 CPU.

I'd like to thank Andy Key for adding the screen dump facility in to Memu, and Dave Stevenson for reading the first draft and his many useful suggestions.

2 UNDER THE HOOD

The MTX series of computes were designed using “Off the Shelf” products available in the early 1980’s the major components of the design were

4mhz Z80A CPU

4mhz Z80A CTC

TMS9929A VDP

SN76489A PSG

4MHz Z80 DART - available on the optional RS232 expansion

Memory varied between systems from 32k on the MTX500, up to a theoretical maximum of 784k available as paged memory under CPM. This was supplied as dynamic RAM which was the norm for the time. For the early systems ram was 64k by 1-bit chips later designs used 256k by 1-bit parts. At the time memory chip manufacturers would package “failed” 64k chips where one of the two 32k banks within the device was working as 32k chips and Memotech used these on the MTX500 and the 32k RAM card.

The motherboard has a system of links between the CPU and the RAM to accommodate either high or low bank 32k RAMs on the MTX500 along with a setting for 64 or 32k on board to identify to the memory configuration PAL (Programable Address Logic) whether the system was the MTX500 or MTX512. The RAM expansion card would also have a PAL programmed to the specific type of RAM fitted to the card and for the system it was to be plugged in to.

The 256k MTX series 2 used the larger 256k devices and along with a re-designed PAL needed patch wiring on the reverse side of the board to make them fit.

The 24k of operating system on the MTX, in common with other systems of the era, isn’t anything like what would be considered one today. Rather there is a collection of routines needed to support MTX BASIC. It’s evident from the ROM listing and the state of the error messages, that the programmers were struggling to fit everything they wanted into the space available.

Some parts of the system are written to use a common access point, for other parts however, it’s up to the user to discover “useful” sections within the ROM and call them directly. This then raises potential issues to anyone that wants to modify the ROM contents, for example removing some of the foreign language tables in order to fix the error massages, as software that relies on those entry points would fail.

The majority of the components on the board were standard 74LS series logic. These provided the “ports” for memory mapping, the keyboard and Joysticks, the printer and cassette tape.

The VDP was fitted with 16k of its own dedicated video memory, which was the maximum the device would accept. These used 3 voltage 16k by 1-bit RAM chips which have proved to be a reliability problem as the systems age.

Storage on the basic system was to cassette tape only. There were 2 different floppy disc systems available from Memotech during the life of the MTX.

The FDX was a large system that could accommodate 2 5.25” drives within the chassis and further external 5.25” or 8” if required. The FDX required 64k of RAM to operate as it was essentially a CPM system. An 80 column video card was a standard part of the CPM system, allowing the use of dual monitors for developing programs. A supplied program, FDXB.COM, enabled access to MTX BASIC with 32k of RAM available like the MTX500. The on-board card frame would also accept solid state “silicon discs” under CPM for additional high speed, temporary storage.

The SDX expansion, at least initially, didn’t offer CPM, instead it had an expansion ROM which added a number of commands to MTX BASIC. The underlying disc system however was CPM, allowing the interchange of discs between the FDX and SDX. Later SDX systems had an extra paged ROM on board that allowed CPM in addition to the BASIC extensions. Some also had extra RAM available as a RAM disc, echoing the silicon discs of the FDX.

3 THE MEMORY MAP

The MTX memory map is complex, in order to fit the operating sys and a reasonable amount of memory into the 64k available to a Z80, a memory paging system is used. To make things more complicated, CPM requires another totally different memory layout in order to function.

Memotech assigned the Z80 output port 0 as a write only memory configuration register. Since the port is write only, they also added a system variable PAGE at #FAD2 (64210 decimal) to store a copy of the current memory configuration.

The Z80's 64k memory is split into 4 logical blocks each 16k in size. In ROM mode the first 16k block, #0000 to #3FFF, is allocated to system ROM, the next 2 blocks at #4000 to #7FFF and #8000 to #BFFF are paged RAM. In the MTX500 the lower of those 2 blocks is empty.

In CPM mode, there is no ROM, so all 3 of these blocks are allocated to paged RAM. In large memory systems, this results in each 16k block of RAM within the paging system potentially having 2 different locations.

The final 16k, occupying the area between #C000 and #FFFF is fixed, and always available to the CPU in either mode. It is home to those parts of the system that must be available to the CPU at all times.

As the Z80 is an 8-bit CPU, there are 8 bits available in any memory or I/O port. The paging register is split into 3 sections.

The 4 lowest bits, labelled P0, P1, P2 & P3 select which of the available RAM pages will be accessed making a maximum of 16 pages available in either mode. In RAM mode this is 16 pages 48k in size, in ROM mode it is 16 pages of 32k. The next 3 bits designated as R0, R1 & R2 select the current paged ROM from a total of 8.

The final bit, RELCPMH, determines whether the system has the ROMs enabled, or is in RAM only CPM mode. The name apparently stands for Rom Enabled Low CPM High

3.1 ROM ENABLED MODE

ROM mode is set when the RELCPMH bit is set to a 0. As noted before, the standard MTX has 24k of on-board ROM, in order to fit this into the 16k space allowed for ROMs, Memotech decided to page the upper 8k, leaving the lower 8k fixed. Since there are 3 bits allocated to the ROM paging system this enabled a total of 72k ROM to be fitted. (9 x 8k ROMSs, 8 paged, 1 fixed) The hardware on the motherboard decodes the lower 8k ROM and paged ROMs 0,1 and 7.

ROM 7 isn't used by the motherboard, but this signal is instead made available on the expansion connector for use by external hardware, and is intended for the ROMpak or NODE ROM.

The lower 8k ROM and paged ROMs 0 and 1 then make up the 24k of the operating system. Typically, ROM 4 would be used by CPM, and ROM 5 by the SDX ROM, although some SDX systems used ROM slot 3.

The internal ROM expansions (NewWord, Pascal and the video wall software) used ROM page 2, which because of additional hardware on the expansion card could be sub paged into up to 256 ROM pages.

The MTX 500 has 32k of available RAM, since half of this is fixed by design as occupying the top 16K of the memory map. Memotech made the obvious decision to fit the other 16k directly below to maintain a continuous 32k block of memory, this 16k is in RAM page 0. The 16k between the top of the ROM and the bottom of the RAM is then empty, any attempt to “poke” data into this area will result in the data being lost, “peeking” from this area will return garbage.

MTX500 memory map					
Page	#0000 to #1FFF	#2000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	OS ROM	BASIC ROM	Empty	16k paged RAM	16k Shared RAM
1		Assem ROM		Empty	
2		Expansion			
3		Expansion			
4		DISC (CPM)			
5		DISC (SDX)			
6		Expansion			
7		ROMpak			
8-15	N/A				

The MTX512 on the other hand has 64k of RAM available, however there is only 48k in the memory map allocated to RAM, the extra 16k has to go into page 1 due to the limitations of the PAL that controls the motherboard memory paging, the extra 16k has to occupy the area from #8000 to #BFFF in page 1, this also makes for a continuous 32k block being available in page 1 as the paged RAM adjoins the 16k fixed area.

MTX512 memory map					
Page	#0000 to #1FFF	#2000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	OS ROM	BASIC ROM	16k Paged RAM	16k paged RAM	16k Shared RAM
1		Assem ROM	Empty	16k Paged RAM	
2		Expansion		Empty	
3		Expansion			
4		DISC (CPM)			
5		DISC (SDX)			
6		Expansion			
7		ROMpak			
8-15	N/A				

Adding RAM to either system is done in multiples of 32k. A MTX512, with an additional 128k would have the memory arranged so that the 5 lowest pages were full, and a 6th half full page would be filled from the top. Page 244 of the Phoenix manual has this wrong, though the equivalent table in the original manual was correct.

MTX512 with 128k expansion					
Page	#0000 to #1FFF	#2000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	OS ROM	BASIC ROM	16k Paged RAM	16k paged RAM	16k Shared RAM
1		ASSEM ROM	128k exp. (a)	16k Paged RAM	
2		Expansion	128k exp. (c)	128k exp. (b)	
3		Expansion	128k exp. (e)	128k exp. (d)	
4		DISC (CPM)	128k exp. (g)	128k exp. (f)	
5		DISC (SDX)	Empty	128k exp. (h)	
6		Expansion		Empty	
7		ROMpak			
8-15	N/A				

During start up, the MTX ROM does a very simple check for available paged RAM, and stores the result in the system variable LSTPG at #FA7A (64122 decimal). PRINT PEEK (64122) returns the number of 32k pages of RAM - 1; the MTX500 will return 0, a standard MTX512 returns a 1. The example MTX 512 + 128k above would return 5.

Anyone using an MTX fitted with Andy Key's REMEMOorizer will get a result of 11, indicating there is 384k (32k x 11 +32k) available from BASIC.

3.2 CPM Mode

Setting CPM mode on an MTX500 is possible, but serves no purpose, as all it does is page out the ROMs, however as there is no additional RAM in the system this just serves to create a vacant 32k block in lower memory.

The 64k in the MTX512 is the minimum required for CPM, with the RELCPMH bit set to 1 all 64k RAM is available in page 0, the other pages are empty. Leaving the memory map looking like this:

MTX512 RAM only memory map				
Page	#0000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	16 k Paged RAM	16k Paged RAM	16k paged RAM	16k Shared RAM
1	Empty	Empty	Empty	
2				
3				
4-15				

Note that the lighter shaded page has "moved", in ROM mode it's in page 1 at #8000, in CPM mode it's the very first block in page 0. This moving page only occurs on systems using memory based on the 64k RAM chip, the MTX series 2 using the larger 256k chips is unable to duplicate this behaviour due to the limitations of the 14L4 PAL used by Memotech.

Expansion memory starts in page 1 and grows upward from the lowest point in memory, depending on the amount of RAM fitted, the final page could have a gap in the map between the last block of RAM

fitted and the shared RAM at #C000. The example below as taken from the manual of a MTX512 with an extra 128k illustrates this, Page 3 has memory available in the lowest 32k and the top 16k, however the area from #8000 to #BFFF is empty. It's worth noting that every one of the blocks has moved from the position it had in RAM mode. For example, block e which was in page 3 at #4000 is now in page 2 at the same address. Block a is still in page 1 but has moved from #4000 to #0000

MTX512 RAM only memory map with 128k expansion				
Page	#0000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	16 k Paged RAM	16k Paged RAM	16k paged RAM	16k Shared RAM
1	128k exp. (a)	128k exp. (b)	128k exp. (c)	
2	128k exp. (d)	128k exp. (e)	128k exp. (f)	
3	128k exp. (g)	128k exp. (h)	Empty	
4-15	Empty	Empty		

3.3 The 256k Series 2

The MTX512 series 2 was fitted with 256k by 1-bit RAM chips, however restrictions imposed by the motherboard and the memory PAL meant that not all of that RAM was accessible in either mode. It also meant the moving page from the earlier systems could not be duplicated.

The RAM itself was mapped into the first 4 pages. Any portions that aren't obscured by the ROM or shared RAM are visible to the CPU.

MTX512 Series 2					
Page	#0000 to #1FFF	#2000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	OS ROM	BASIC ROM	16k Paged RAM	16k paged RAM	16k Shared RAM
1		Assem ROM	16k Paged RAM	16k paged RAM	
2		Expansion	16k Paged RAM	16k paged RAM	
3		Expansion	16k Paged RAM	16k paged RAM	
4		DISC (CPM)	Empty	Empty	
5		DISC (SDX)			
6		Expansion			
7		ROMpak			
8-15	N/A				

With RELCPMH set to 1, the 4 banks of RAM that were previously covered by the ROM are now visible

MTX512 Series 2 RAM only mode				
Page	#0000 to #3FFF	#4000 - #7FFF	#8000 - #BFFF	#C000 - #FFFFF
0	16k paged RAM	16k Paged RAM	16k paged RAM	16k Shared RAM
1	16k paged RAM	16k paged RAM	16k paged RAM	
2	16k paged RAM	16k paged RAM	16k paged RAM	
3	16k paged RAM	16k paged RAM	16k paged RAM	
4-15	Empty	Empty	Empty	

3.4 USING PAGED MEMORY

There are 2 things to be aware of when using the page port:

the first is not to move a page that the CPU is currently using, and

the other is to take precautions for interrupts.

For safety, on start-up, the MTX ROM initializes the Z80's stack pointer to use shared RAM at #FD48. If the system stack were placed in paged RAM, then a system crash would be the likely result of any RAM re-paging. The OS takes care to only re-map the paged ROMs while running from the fixed OS ROM, likewise a user program changing the RAM page should only do so with code that is running in the shared RAM area.

If a paged ROM needs to access routines in another paged ROM, then it needs to do the same as the SDX ROM, copy a short routine into RAM that changes the ROM page and calls the appropriate routine then reverts to the original set up before returning.

When running in BASIC, the MTX interrupt system generates 125 interrupts per second, which needs to be serviced and may or may not involve a different paged ROM. On exit from the interrupt, the system checks PAGE and uses that to reset the memory map to its original setting before returning. Therefore, the following code should be used when changing pages:

```
LD (PAGE),A
OUT (#00),A
```

The system variable needs to be changed first, otherwise an interrupt occurring between the 2 instructions could leave the page port in an incorrect state. With the instructions this way around, if an interrupt were to split them, then the only result would be that the memory would already be re-mapped making the OUT instruction redundant.

For relatively short sequences of code, it's also possible to disable interrupts while paging, though that's not really recommended.

As a practical example, the MTX has 24k of ROM, it's possible to use PEEK from BASIC to examine the lower 8k of ROM, it's also possible to read one of the 2 paged ROMs, but not the other. You can only PEEK the ROM that the PEEK command itself runs from. The same issue arises with PANEL, the only paged ROM that can be examined is the one that PANEL occupies. Should you want to examine the contents of one of the other paged ROMs they need copying to RAM first.

The following program would copy the contents of ROM 2 to #A000, should you want to examine another ROM, just change #4019 to LD A,#30 or #40 etc. the first digit is the ROM number from 0 to 7, the last digit should be a 0 to ensure the copy goes to accessible RAM, using hex numbers in this instance make the intention clearer.

NB While the MTX's built in assembler is functional and reasonably fast, it lacks a decent equates system, so it's not always possible to avoid using numerical references where a label would be clearer. E.g. It is not possible to set a label for PAGE, instead #FAD2 has to be used. The original manual's assembler and panel section is pretty abysmal. The later manual is much better in that respect, but still not great.

```
10 ASSEM
CODE

4007          LD DE, #D000
400A          LD BC,#20
400D          LD HL,START
4010          LDIR
4012          JP #D000
4015 START:   LD A,(#FAD2)
4018          PUSH AF
4019          LD A,#20
401B          LD (#FAD2),A
401E          OUT A,(0)
4020          LD HL,#2000
4023          LD DE,#A000
4026          LD BC,#2000
4029          LDIR
402B          POP AF
402C          LD (#FAD2),A
402F          OUT A,(0)
4031          RET

Symbols:
START    4015
```

The first 4 lines used the Z80 block move to copy the actual code into shared RAM, which is then run by the JP instruction on the 5th line.

Since this code is designed to only change the paged ROM, and as a user program is running in RAM page 0, copying the code to shared RAM first isn't really needed. However, if the same code were running in a paged ROM it would be vital.

The code itself is fairly straightforward.

The code at START gets the current memory layout and saves it to the stack, then sets the new layout in order to access ROM bank 2 and RAM bank 0, then it's another block move to copy the ROM into RAM.

The final lines restore the memory map from the stacked value and then exit.

4 THE VIDEO SYSTEM

The MTX series makes use of the TMS9929A Video Display Processor to generate the display in its PAL models, it was a popular device at the time and was used in a number of home computer systems. The NTSC equivalent is the TMS9928A. From a programmer's point of view, the 2 devices are very similar. The only major difference I'm aware of is because of the different television standard's frame rates, the VDP interrupt occurs 50 times a second on the PAL device and against 60 for NTSC.

The VDP has 16k of its own dedicated video memory, this memory is not visible to the Z80A CPU and all communication between the CPU and the VDP has to go via I/O ports.

The main advantages of the separate memory space means that the CPU does not have to give up RAM to the display nor is any extra circuitry needed to arbitrate between the CPU and VDP potentially slowing the CPU down.

The obvious disadvantage, the CPU has no direct access to the video memory reducing the rate at which new data can be moved to and from the screen. This is mitigated to a certain extent by the availability of hardware sprites and the choice of character mapped or bit mapped display modes.

The MTX ROM provides a number of useful routines for screen handling via the RST 10 and RST 28 interface. However, for maximum performance and some extra facilities, accessing the VDP directly is the way to go.

4.1 ACCESSING THE DISPLAY VIA RST 10 AND RST 28

The majority of the screen handling can be done via RST #10 and those functions are documented to a certain extent in the Phoenix manual as well as Keith's book. However, there are also a couple of useful routines within the functions of the general purpose RST #28 that I discovered from the ROM disassembly that I've not seen documented elsewhere.

Both RST #10 and #28 operate using in-line data. The RST instruction is followed by one or more data bytes. The program code then continues from the end for the data. Obviously, it is vital to get the data statements correct, otherwise a system crash is the likely result if the CPU starts trying to run data as program code.

RST 28 function #42 is used to initialize the video system and is called by the MTX ROM during start up. However, this is after the paged ROMs are set up, if you're programming a paged ROM, and need to do any printing then the following code is a must.

```
RST 28
DB #42
```

This calls the routine VDINIT at #2E85 in the Assem ROM, so there's no way to make the call directly from within a paged ROM. The other RST 28 function that can be used for accessing the display is function #AC (which calls the ROM routine PRINTX at #0CAB). This simply sends whatever is in the A register directly to the screen drivers. It will corrupt the A register on exit so should be used with that restriction in mind. It is however, faster to use than the equivalent RST 10 function.

Control codes, as well as displayable characters, can be sent to the display, as shown in the following example, which uses code 12 (clear screen) as well as 13 & 10 (carriage return and line feed)

```
10 ASSEM
CODE

0x4007           LD HL,DATA
0x400A LOOP:     LD A,(HL)
0x400B           AND A
0x400C           RET Z
0x400D           RST 28
0x400E           DB #AC
0x400F           INC HL
0x4010           JR LOOP
0x4012 DATA:     DB 12
0x4013           DB "HELLO WORLD"
0x401E           DB 13,10,0
0x4021           RET

Symbols:
DATA          4012  LOOP          400A
```

As the routine called by this RST 28 function is in the low ROM, it can be called directly, which is faster, but ends up being 1 byte longer. Since the function is called 14 times in the MTX ROMs, it would have saved the original programmers 14 bytes of ROM space. The RST and Data byte in the above code can be replaced with CALL #0CAB.

The RST 10 call is actually 4 functions accessed through one entry point. The bits 6 and 7 of the data byte determines which function is called. Bit 5 is the continuation bit if this is set then the RST 10 processing code expects to find another data byte after the completion of the current command. If it's clear the CPU will resume processing Z80 code at the end of the current command. Bits 0-4 of the data byte are command specific.

A data byte with the top 2 bits clear (i.e. 00 C xxxxx) are used to send a single byte to the display. As there are only 5 bits available, the value sent is restricted to between 0 and 31, which means control codes only. This may seem a little restrictive, but it is useful for things like clearing the screen or sending a carriage return to the display, as it is both shorter than any of the other options and needs no registers.

Sending a clear screen command (Character code 12 or #0C in hex) to the screen could be done with:

```
LD A,#0C
RST 28
DB #AC
```

Take twice as much space as:

```
RST 10
DB #0C
```

And changes the A register in the process.

Sending a carriage return and line feed to the screen can be done in 3 bytes:

```
RST 10
DB #2D,#0A
```

Note the setting of the continuation bit in the first data byte. There is no limit to the number of continuation items that form a single RST 10 command, nor are there any restrictions in mixing the various functions.

Data bytes with bit 7 clear and bit 6 set deal with the MTX's virtual screen system. Bit 5 is the continuation bit as with all commands, bit 4 isn't used. Bit 3 if set to 1 will clear the virtual screen, set to zero, it won't. Bits 0-2 are the virtual screen number 0-7.

For example, to select VS 4, clear it, and set the continuation bit ready for further printing, the bit pattern for the data byte would need to be 01 1 x 1 100 which is #6C in hex (or #7C). Similarly, selecting VS 5 without clearing it and resuming processing, the data byte is 01 0 x 0 101 or #45.

```
10 CODE

4007        RST 10
4008        DB #4C
4009        RET

Symbols:


20 PRINT "HELLO"
30 LINE 0,183,36,183
40 GOTO 40
```

This code the short assembly section selects VS 4, the graphics screen and clears it. The BASIC lines below print the message and underline it, producing the following output.

It also illustrates the fact that the display system treats all video data in the same way, whether it originates from BASIC or code.

The third function is the most complex, and also the most useful. The data byte has bit 7 set, bit 6 clear, and bit 5 is the continuation bit. The remaining 5 bits are a count from 1 to 31 of the number of bytes to be sent to the screen. Control codes and printable characters can be mixed as required, if more than 31 characters are needed, then the data can be separated into smaller sections and the continuation bit used.

To duplicate the above display with RST 10 the following code could be used.

```
10 CODE

4007        RST 10
4008        DB #6C
4009        DB #A7,"HELLO",13,10
4011        DB #85,2,0,183,36,183
4017        RET

Symbols:

40 GOTO 40
```

The continuation bit is set on the command to select screen 4 and clear it, changing it from #4C to #6C. The next line is the 10 1 bit pattern for embedded data with the continuation bit set, as there are 7 data bytes following which is 00111, making the final byte #A7. The text that follows is the same as the BASIC, along with a carriage return and line feed.

The final line #85 is the bit pattern is 10 0 for embedded data and no continuation and 00101 indicating it is followed by 5 data bytes. The 5 data bytes are 2 which is the control code for LINE, followed by the same 4 positional bytes as the BASIC.

The final RST 10 function has both bits 6 and 7 set with bit 5 as the continuation bit as usual. The other 5 bits are unused. This function simply sends the 2 characters in the Z80's BC register pair to the screen. The Character in C is sent first, followed by B. The ROM actually uses RST 28 function #AC to do this. So, unless there is a compelling reason, in most cases it is probably simpler to use the RST 28 function. It saves having to remember which order BC is sent, and uses the A register which is probably more natural in most circumstances.

4.2 Accessing the VDP directly

Communication between the Z80 and the VDP is done using 2 I/O ports. Port 1 is used to transfer any data between the two, while port 2 is used by the Z80 to write to the VDP registers and memory pointer. Reading port 2 will access the VDP status register.

That's a little vague, so needs a little detail on exactly what that means.

The VDP has 16k of video memory connected to it, the Z80 only has indirect access to that memory. In order to place data into that memory, or read from it, it has to tell the VDP the address where it wants to access. Once that address is set up, each access to the data register will automatically increment the address pointer.

However, the VDP needs to know in advance whether the access through the data register at port 1 will be a read or a write. It's not possible, when say plotting a single pixel onto the bitmap display, to read a byte, set the pixel and write it back without doing a second address set up in between.

16k of RAM needs a 14-bit address to access each byte individually. Using 2, 8-bit, transfers leaves 2 bits for identification of the type of transfer taking place. The VDP uses bits 6 and 7 of the 2nd byte transferred for this.

In order to send a byte to the VDP RAM, the low 8 bits of the address need to be sent to port 2, followed by the upper 6 bits. Bit 7 of the 2nd byte needs to be a 0, bit 6 needs to be a 1. Any data then set to port 1 will be placed into the video RAM.

There are 2 potential issues with accessing the video memory this way. The first is that when your code starts an address set up its impossible to know whether the VDP is part way through an aborted address set up or not. The first byte you send could be completing a previous address set up. The solution is simply to read form either the data register on port 1 or the VDP status register on port 2 before starting the address set up.

The other issue is interrupts. It's possible for an interrupt to fire between the 2 VDP address setups. If that interrupt were to read the VDP status register, then the address setup would be re-started on return to your code with unpredictable results especially so on VDP writes. Unless you can be 100% certain that any interrupts won't access the VDP, or if you only access the VDP under interrupt control. Then a DI instruction before the address setup and an EI afterwards to temporarily halt interrupts can be a good idea.

It IS possible to write to the VDP from BASIC using the OUT command, and most of the time it will probably work as expected. But it's not guaranteed.

Port 2 is also used to write to any of the 8 VDP registers that control how the device is set up and what each section of video memory is used for. Register's 0 and 1 control the basic setup, which screen mode, sprites, the VDP interrupt etc. The Register 7 controls the colour of the backdrop and text screen. The other 5 registers are pointers to the various lookup tables the VDP uses to create the display.

There are timing restrictions with communicating with the VDP. After setting up the address, the CPU needs to wait 3 micro seconds before attempting to read or write data. With the MTX's Z80 running at 4mhz, this means the delay should be 12 machine cycles. This isn't a major issue, reading the data register with IN A,(1) takes 11 cycles, adding one instruction between the final address set up OUT (2),A and the IN instruction is sufficient. The same applies for writing to the data register as OUT (1),A also takes 11 cycles. There are other I/O instructions available on the Z80, however, they all take more cycles to complete.

What does need a little care is allowing sufficient time between successive data accesses in graphics modes. In text mode there's no problem the minimum time between read or writes is 2 micro seconds, which Is 8 cycles and all the I/O instructions take longer than that. In Graphics mode where the VDP needs more memory access for itself because of the complexity of the display, 8 micro seconds are required between CPU accesses, so a little thought is required when laying out screen access code.

Which mode the VDP is using, is determined by the current setup, and that requires writing to one or more of the 8 write only registers.

In order to write to a register, the register data is sent to port 2 first, then the register number, to separate register writes from the address set up sequence Bit 7 should be set, and bit 6 should be clear. The register number should be between 0 and 7 to maintain compatibility.

Register 0 and 1 between them can best be thought of as a collection flags.

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	BIT 1	Bit 0
0							M3	EV
1	4/16k	Blank	IE	M1	M2		Size	Mag

7 of the 16 bits are un-used and should be set to 0 for compatibility with later VDPs. Of the remaining 9, M1, M2 and M3 select which display mode the VDP uses. None of the modes require more than one bit to be set. It is possible to create "illegal" screen modes by setting more than one mode bit, those modes aren't documented here.

The first 3 modes were inherited from the older TMS9918 VDP used in the TI 99/4. Graphics 2 is only available on the later "A" version devices. (9918A, 9928A and 9929A)

With all 3 bits clear, the VDP will be in graphics 1 mode. The display is 32 by 24 in up to 15 colours, the sprites system is active. Each character can only have 2 colours.

M1 set, M2 and M3 clear puts the VDP into text mode. The display is set to 40 by 24 in 2 colours and the sprite system is turned off. There are 2 colours available for the whole display, setting individual character colours is not possible

M2 set, M1 and M3 clear puts the VDP into Multi colour mode, the display is 64 by 48 in 15 colours, sprites are active. I'm not aware of any MTX software that uses this mode.

M3 set, M1 and M2 clear puts the VDP into Graphics 2 mode, The Display is 32 by 24 in up to 15 colours. Each character row has its own 2 colours, with the correct setup a bitmapped display can be created.

There's only one other useable bit in register 0, EV. Setting this bit enables the external video input in the 9918 and 9918A. Since the MTX uses the 9929A setting it has no effect, and so it should be set to 0 for compatibility.

Register 1 has 5 other, useable bits. Bit 7 should be set on the MTX, as it's used to inform the VDP of the amount of video RAM fitted. With the bit clear the VDP will assume 4k x1 RAM chips are fitted and adjust accordingly.

If the BLANK bit is clear, then the VDP display is turned off, and will show the current border/backdrop colour. Setting it to 1 restores the current display, so possibly of use in some circumstances to hide the display being built.

The IE bit, is the interrupt enable, if this is set the VDP will issue an interrupt at the end of the active display. The interrupt signal from the VDP is connected to the input of counter/timer 0 on the CTC, meaning both the CTC and VDP need to be correctly setup in order to make use of the VDP interrupt. Although its connected to the CTC, the MTX ROMs make no use of the video interrupt. If VDP interrupts are used, the VDP status register must be read during the interrupt routine as reading the register clears the interrupt flag. If the interrupt flag isn't cleared further interrupts can't occur.

Bits 0 and 1 control the sprite settings. If MAG is set, all sprites are drawn double size. If the Size bit is set, then sprites are drawn from 4 consecutive 8x8 sprite characters in memory. If unset, then 8x8 sprites are drawn from a single sprite character.

At reset, the VDP sets both register 0 and 1 to all zero, meaning the display is blanked, interrupts are off and the VDP is in graphics 1 mode with 8x8 single pixel sprites.

The remaining 6 registers have the following functions.

Register	Function	Bit 7	Bit 6	Bit 5	Bit 4	BIT 3	BIT 2	BIT 1	BIT 0
2	Name Table	0	0	0	0	A13	A12	A11	A10
3	Colour Table	A13	A12	A11	A10	A9	A8	A7	A6
4	Pattern Table	0	0	0	0	0	A13	A12	A11
5	Sprite Attribute Table	0	A13	A12	A11	A10	A9	A8	A7
6	Sprite Pattern Table	0	0	0	0	0	A13	A12	A11
7	Text Colour Register	Text Mode, Text Colour				Background/Border colour			

Depending on the display mode some of the registers may not be used. As each table in video memory requires differing amounts of space depending on the amount of data it holds, the VDP designers decided to restrict the amount of address bits available to position the table. I.e. the "Name Table" is 1k in size, the programmer can position it on any kilobyte boundary as just 4 of the 8 bits in the register are used. As usual any un-used high order bits need to be set to 0 for compatibility with later devices.

It's important to note, that though the number of bits available for locating tables in video ram is restricted, overlapping of tables is allowed. For example, the Pattern Table and Sprite Pattern Table could occupy the same position in VRAM, with the programmer then deciding which sections of the table are used as characters, and which as sprites, or both!

The VDP is primarily a character-based device, and in order to create a character-based display 2 tables are required. The Name Table is in effect a character map for the display. So for example, if the display is set with the ASCII character set, placing a byte with a value of #41 (decimal 65) into the first entry in the table will display the character "A" on the top left of the screen.

The Pattern Table contains the actual character pattern that is displayed. There are 8 bytes per character, and 256 possible characters, making a maximum table size of 2k, which is why the register only allocates 3 bits for the pointer. The name table must go on a 2k boundary.

4.3 TEXT MODE

Text mode uses 3 of the VDP registers to control the display. These are register 2 the name table, register 4, the pattern table and register 7 the colour register.

Only 2 colours are available, any "set" pixels will show as the text colour from the upper 4 bits of register 7, everything else is rendered in the backdrop colour from the lower 4 bits of the same register.

The ROM places the Pattern Table at #1800 in VRAM, with the Name table over lapping it at #1C00. This effectively reduces the pattern table to 128 or so entries, of which the ROM sets up patterns 32 to 127 as the ASCII character set. Pattern 0 is also used by the cursor.

The overlapping table setup is required to fit both the bitmap graphics display (I hesitate to call it High Resolution) and text mode into the available ram space without them interfering with each other.

In order to set the display, the MTX rom sends the following data to the 3 registers

```
Register 2: #07
Register 4: #03
Register 7: #F5
```

The binary version of the table pointer values makes it a little clearer what is being set :

```
Register 2; (0000)  01 1100 0000 0000
Register 4: (00000) 01 1000 0000 0000
```

The cleared upper bits (in brackets) are ignored by the 9929 VDP but should be programmed as clear for compatibility. Referring to the table above, for register 2 the lower 4 bits are significant, for register 4 it's only 3,. Adding in the rest of the low order address bits (in red) confirms that register 2 has set the name table to #1c00 and register 4 the pattern table to #1800.

Register 7 is much simpler to understand, as it's setting white text -colour 15 (f) on a light blue background -colour 5.

Unless there's a good reason not to, it's simpler to use the MTX's default VRAM layout than to define one of your own. The following BASIC program illustrates direct screen access

```
0 LET A=INP(2)
10 OUT 2,2*16+1
20 OUT 2,128+7
30 OUT 2,1
40 OUT 2,64+28
50 LET A$+"Hello World"
60 FOR X=1 TO LEN (A$)
70 OUT 1,ASC(MID$(A$,X,1))
80 NEXT
90 GOTO 90
```

```
Line 0 clears any part written port 2 setup.
Lines 10 and 20 set register 7 to #21, for a green on black display.
Line 30 and 40 set the VRAM pointer to #1C01 ready for writing
Line 50 is the message
Lines 60 to 80 pass that message 1 byte at a time to the video ram.
Line 90 prevents the MTX from reverting to blue on white when the
program ends.
```

All characters and sprites used by the VDP are based on an 8 by 8 matrix, however on order to fit 40 columns on a display 256 pixels wide the lowest 2 bits of each row of character data are ignored. The 40 column display is therefore 240 pixels wide, to accommodate this the borders are adjusted to keep the screen more or less central.

Assuming the name table is at its default position at #1C00 the location of any character on the screen can be calculated as #1C00 + row * 40 + column. An additional factor to take into account when writing to the screen is that bit 6 needs to be set on the address transfer, making the calculation effectively #5C00 + row * 40 + column.

In assembler there are several ways to calculate this. The simplest is probably to use left shifts to multiply by 2. As 40 is 5 x 8, he following routine will calculate the screen address in HL, assuming the row number is in B and the column in C, B and C are assumed to have been range checked and so B is between 0 and 23 and C between 0 and 39. Multiplying B by 5 is done in the A register as 5 x 39 is 235 and is with the permitted values for an 8 bit register. The final multiply by 8 needs a 16 bit register pair.

```
SCREEN_POS:
PUSH AF       ;save the working registers for neatness
PUSH BC
LD A,B
ADD A,A       ; 2x row
ADD A,A       ; 4x row
ADD A,B       ; 5x row
LD L,A        ; transfer A to HL for the final multiply by 8
LD H,0
ADD HL,HL     ; 10x row
ADD HL,HL     ; 20x row
ADD HL,HL     ; 40x row
LD B,#1C      ; the column value is in C, setting B to #1C here
              ; saves having to add #1C00 later
```

```
ADD HL,BC
POP BC
POP AF
RET
```

Reading or writing the character ad row B, column C is then simply:

```
WRITE_A:
CALL SCREEN_POS
PUSH AF
LD A,L
OUT (2),A
LD A,H
OR #40
OUT (2),A
POP AF
OUT (1),A
RET
```

Reading the character is even simpler:

```
READ_A:
CALL SCREEN_POS
IN A,(1)
RET
```

Since the VDP's internal character pointer is incremented after each access, the further characters could be transferred with just an additional IN A,(1) or OUT (1),A.

To change the appearance of any character, simply update the data in the pattern table. Each character is stored as 8 consecutive bytes, in the default setup, the position is therefore #1800 + 8 * character_no

To set HL to the location of character A one way of doing it is:

```
CHARCTER_POS:
LD L,A       ; character code can be any value 0-255
LD H,3       ; save adding #1800 later, by adding #0300 now
ADD HL,HL    ; HL is now #0600 + 2 x A
ADD HL,HL    ; HL is now #0C00 + 4 x A
ADD HL,HL    ; HL is now #1800 + 8 X A
RET
```

Once the VDP address pointer is set up, then 8 bytes can be transferred with repeated IN or OUT instructions.

I mentioned earlier that there overlapping of the table leaves approximately 128 characters available to the programmer, provided the cursor blink is accounted for, Entries character codes 0-127 are free, then next 960 bytes are taken up by the name table, this leaved a further 64 bytes at the end of the name table that can be used for character definitions. That makes 8 further characters with codes 248 to 255 that can be defined without corrupting the display, making 136 in all.

4.4 GRAPHICS 2 MODE

Graphics 2 is the other mode used as standard on the MTX, despite appearances, it is a character-based mode. In G2 mode there are 24 rows of 32 columns in the display, giving 768 locations. However, with the normal G2 setup, the name, pattern and colour tables are both split into thirds, each third of the name table therefore has 256 entries, allowing each one to be unique. Each byte in the Pattern table has a corresponding byte in the colour table, allowing each group of 8 pixels to have 2 colours.

In order to use the display as if it were bitmapped, each segment of the name table is set up as ascending bytes from 0 to 255 (#FF). The display is then created by accessing the pattern and colour tables directly.

In G2 mode the name table is 768 bytes, making the Pattern table 768 x8, or 6144 bytes in length, as is the colour table. Therefore, there are only 2 possible locations for these tables. Right at the start of Vram in locations 0 -6143 (#17FF) or from the mid-point from 8192 to 14335 (#2000-#37FF).

The Pattern Table and Colour table meanings are modified, only the A13 has any significance in determining the table location. The TI documentation says that all the lower order address bits need to be set to 1 in order to use the full number of patterns. This gives the following 4 “legal” values for the 2 tables

Table	Register	Binary Value	Decimal	Hex	Decimal Location	Hex Location	
Pattern	4	00000 0 11	3	#03	0 - 6143	#0000 - #17FF	MTX setting
Pattern	4	00000 1 11	7	#07	8192 - 14335	#2000 - #37FF	
Colour	3	0 1111111	127	#7F	0 - 6143	#0000 - #17FF	
Colour	3	1 1111111	255	#FF	8192 - 14335	#2000 - #37FF	MTX setting

To avoid complications with overlapping tables, the name table needs to be fitted into one of the available 2k blocks. The MTX rom uses the first “free” area for the text screen, so places the sprites and Name table in the 2nd area. Register 2 is set to 15 (#0f) placing the table at 15386 (#3C00) however once it’s set up it can be forgotten.

The following BASIC code is an example of how to setup and access graphics 2.

```
0 REM GRAPHIC 2 SETUP EXAMPLE
10 LET A=INP(2)
20 OUT 2,3: OUT 2,128+4
30 OUT 2,255; OUT 2,128+3
40 OUT 2,15; OUT 2,128+2
50 OUT 2,15: OUT 2,128+7
60 OUT 2,0: OUT 2,64+60
70 FOR X=1 TO 3
80 FOR Y=0 TO 255
90 OUT 1,Y
100 NEXT
110 NEXT
120 OUT 2,2: OUT 2,128+0
130 OUT 2,64: OUT 2,128+1
```

```
140 OUT 2,0: OUT 2,64+0
150 FOR X=0 TO 6143
160 0UT 1,85
170 NEXT
180 OUT 2,0: OUT 2,64+32
190 FOR X=0 TO 6143
200 OUT 1,INT(X/24)
210 NEXT X
999 GOTO 999
```

The code can be broken down into 2 sections, first the setup sequence:

Line 10 is there to clear any part started transfers. Lines 2- to 40 set up the pattern table, colour table and name table with the MTX default values. Line 50 sets the border to white. Lines 70 to 110 then set up the name table as the required 3 repeating sequences of 0 to 255. Lines 120 and 130 set up the required mode bits for graphic 2, and turn on the display.

The second section sets up a display. Line 140 sets the video memory pointer to the start of the pattern table which is at locations 0 to 6143. The +64 is to ensure the VDP is expecting memory writes. Lines 150 to 170 then poke a vertical line pattern into the screen memory. Line 180 then moves the video memory pointer to 8192 (#2000), again setting up for transfers to vram. Lines 190 to 210 then set 24 consecutive locations (3 characters) to each of the 256 possible colour combinations.

Line 999 is just there to stop BASIC returning to “Ready” on completion.

For a checkerboard display instead of lines, add the following

```
145 LET J=85
160 OUT 1,J
165 LET J=255-J
```

Because the display is character mapped, calculating the screen address is more complicated that it would be if it were arranged on a row by row basis. It’s best explained using the binary representation of the values. As each row of the character is 8 pixels wide, and each character fills 8 rows, the “x” value can be thought of as 2 fields. The upper most 5 bits select which column of characters to access, the 3 x bits, determine which pixel within the character row. The first pixel within the byte that’s displayed being bit 7, so the bit order needs to be reversed, as bi7 7 is pixel 0, bit 6 pixel 1 etc.

Each character is 8 rows high, so the 3 low bits of the Y value select this offset into the character. As each group of 8 rows is 256 bytes long (8 rows x 32 characters) the upper 5 bits determine the high byte of the address.

Colour memory is grouped by 8 pixels, so the calculation is identical except the 3 low “x” bits are ignored, and the offset for colour memory has to be added.

Binary representation of the X value (0-255) ccccc xxx

Binary representation of the Y Value (0-191) rrrrr yyy

Final pixel address 000rrrrrccccc yyy, bit 7-xxx

Final colour address 001rrrrrccccyyy

Drawing a line in BASIC could be done something like this:

```
10 REM GRAPHICS 2 PLOTTING
20 VS 4
30 CLS
40 FOR YPOS=0 TO 191
50 LET XPOS=INT(50+ypos/2)
60 LET C=8*INT(XPOS/8)
70 LET B=7-(XPOS-C)
80 LET R=INT(YPOS/8)
90 LET Y=YPOS-8*R
100 REM PLOT IT!
110 OUT 2,C+Y: OUT 2,64+R: OUT 1,2^B
120 NEXT
999 GOTO 999
```

NB BASIC's OUT command is sufficiently slow the line is almost guaranteed to have glitches!

The equivalent code in assembler is actually a little easier to follow because of the CPU's ability to mask bits in a way BASIC doesn't provide

Assuming on input the B register holds the X position, and the C register the Y one. The following code will set up HL with the byte address.

```
LD A,7        ; mask for the lowest 3 bits
AND C         ; A is now the low 3 bits of the Y value only (yyy)
LD L,A        ; save for later
LD A, #248    ; mask for the top 5 bits
AND B         ; A is now the top 5 bits of the x value (ccccc000)
OR L          ; add back the stored value
LD L,A        ; the low byte is now cccccyyy as above
LD H,C        ; get the Y value
SRL H         ; shift right 3 places to form
SRL H         ; 000rrrrr
SRL H         ; HL now has the full address 000rrrrrccccyyy
```

There's a major issue that still needs to be dealt with, the code above over writes all 8 pixels in the byte to set just the one. What needs to happen is first read in the byte from screen memory, set the pixel and then write it back. However, the screen memory pointer needs to know in advance the type of access being done and it also auto increments after the read, so the address needs to be set up again for the write back. Which leads to the following sequence:

```
Calculate the screen address
Set up the screen address, for a reading
Read the byte
set/reset the required pixel
set up the screen address for writing
write back the modified byte
set up the screen address + #2000 also for writing
write the colour data
```

And that all takes time.

4.4.1 Graphics 2 in text mode

While MTX BASIC uses Graphics 2 as a bitmapped display, the underlying display system is actually a character based one.

Using it as a text display is not unlike the 40 column text mode, the display is 32 characters by 24. Using the default screen set up, the character and colour data needs to be written 3 times, to each of the pattern and colour tables.

Once that is done, a single byte update to the name table updates the display, and unlike the default MTX setup for text mode, all 256 patterns are available.

There is a simpler way to achieve the same effect using some less well documented abilities of the VDP. The TI documentation advises that the “unused” low bits of the pattern and name tables should be set to 1. The reason for this is that those bits are binary ANDed with the memory address when accessing video memory.

Bit 2 selects which of the two 8k halves of the video memory are used for the Pattern table. That is bit A13 of the video memory address, bits 0 and 1 therefore will mask A11 and A12 respectively. Using the default setup both those bits are set to 1, and the full memory range is accessed. However, if they are both set to 0 along with bit 2, all video memory reads for the pattern table will be from the first 2k table only.

From BASIC this can be demonstrated with

```
10 VS 4: CLS
20 PRINT "COMPRESSED PATTERN TABLE"
30 OUT 2,0: OUT 2,128+4
40 GOTO 40
```

Line 30 here sets VDP register 4 to 0, forcing the VDP to use the first table for all 3 segments of the screen. Resulting in the text appearing on the screen 3 times, in the top row of each 1/3 of the screen.

For the colour table, bit 7 of register 3 sets the upper or lower bank, leaving bits 5 and 6 to mask A11 and A12. Masking lower bits will reduce the colour table further but don’t really add anything to the flexibility of the compressed display. Clearing bits 5 and 6 gives a binary pattern of 1001 1111, which is #9F in hex and 159 in decimal.

Changing the default VDP setup this way means that there is effectively one 2k long table in video memory at #0000 to #1FFF for the character patterns and one 2k long table at #8000 to #9FFF for the colours. The table at #3C00 to 3EFF is then the character map for the display.

Using BASIC to demonstrate:

```
10 VS 4: CLS
20 CSR 0,1
30 OUT 2,0: OUT 2,128+4
40 OUT 2,159: OUT 2,128+3
50 FOR X=32 TO 126
60 PRINT CHR$(X);
70 NEXT X
80 OUT 2,0: OUT 2,64+60
90 FOR X=0 TO 767
100 OUT 1,32
110 NEXT X
120 OUT 2,0: OUT 2,64+60
130 LET A$= "G2 TEXT MODE DISPLAY"
140 FOR X= 1 TO LEN (A$)
150 OUT 1,ASC(A$(X))
160 NEXT X
170 GOTO 170
```

The section up to line 70, sets up the display, and pokes the standard MTX character set into the definitions for characters 32 to 126.

Line 80 then sets the video memory output pointer to #3C00, then 90 to 110 clear the screen by inserting spaces.

Line 120 then sets the pointer back to the top of the screen for line 130 onwards to output the text in A$ to the display.

Adding the following will show that the colours are defined per character, and not for the whole display as is the case for text mode.

```
55 COLOUR 0,1+RND*15
```

Leaving out one of line 30 or 40 will map one table to 2k and the other to the default 6k allowing for either one set of characters with 3 sets of colours one for each 1/3 of the screen, or the reverse where there are separate characters for each 1/3, but only one set of colours.

5 THE CTC

The MTX uses the standard Z80A CTC to handle the Clocks for the optional RS232 board and cassette tape interface as well as the main system timer. The CTC in the device name highlight that it is both a Counter and Timer Chip.

There are 4 “event” inputs on the CTC. The VDP interrupt is connected to the channel 0 input, but isn’t used by the OS, channels 1 and 2 are the master baud rate clock inputs and are both connected to a 300kHz clock obtained by dividing the main system clock by 13. The final input on channel 3 is the feed from the tape input circuitry.

Due to package limitations, the CTC only has 3 outputs instead of the expected 4. Channel 0 is unconnected, while channels 1 and 2 feed out to the edge connectors as the SER1 and SER2 signals for the RS232 board

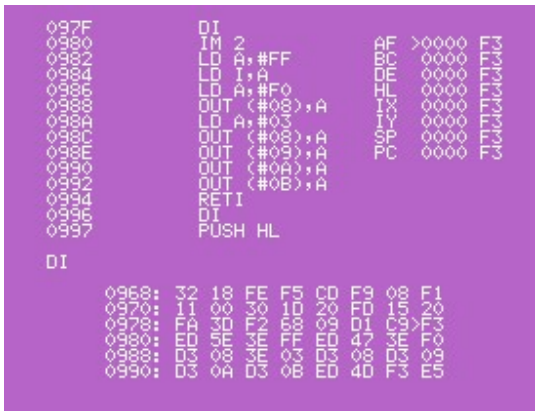
Neither of the MTX manuals goes into great detail on the CTC, however with the Zilog datasheet being readily available on the net, that’s not as much of issue as it might have been in the 80s.

The CTC occupies 4 read/write I/O ports in the MTX running from #08 to #0B (8 to 11 decimal). The CTC also needs 8 bytes of RAM for an interrupt vector table so that each counter/timer can have its own interrupt routine. That table is stored right at the top of memory at #FFF0 (65520 decimal).

Each channel has 2 registers, the channel control word and time constant word. There’s also an additional “common” register, the interrupt vector word.

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	BIT 1	Bit 0
Interrupt Vector	A7	A6	A5	A4	A3	xx	xx	0
Channel Control	Interrupt Enable	Channel Mode	Prescaler Mode	Edge Select	Timer Trigger	Time Constant Follows	Software Reset	1

The MTX sets up the CTC at #097F in the main OS rom and can be viewed in Panel.



The interrupt vector is set in 2 steps. The Z80's I register needs to be set to #FF to set the top byte. The CTC interrupt vector word is set to #F0 to provide the remainder of the base address, bit 0 being clear informs the CTC this is a vector word.

Next all 4 channels are reset in software. Before the routine ends.

5.1 125HZ INTERRUPT

Unlike most systems the main interrupt runs at 125Hz and not anything related to the screen refresh rate. This has the obvious advantage that the interrupt will run at the same speed on all systems regardless of the VDP fitted. That removes the need for separate NTSC and PAL versions of the ROM.

The setup code of the interrupt is at #0996 in the OS ROM

```
0996            DI
0997            PUSH HL          AF >0000 F3
0998            LD HL,#0780      BC  0000 F3
099B            LD (#FFF0),HL    DE  0000 F3
099E            LD HL,#1C11      HL  0000 F3
09A1            LD (#FFF4),HL    IX  0000 F3
09A4            POP HL           IY  0000 F3
09A5            LD A,#A5         SP  0000 F3
09A7            OUT (#08),A      PC  0000 F3
09A9            LD A,#7D
09AB            OUT (#08),A
09AD            EI
09AE            RETI
09B0            LD A,(#FD7E)

DI

     0968: 32 18 FE F5 CD F9 08 F1
     0970: 11 00 30 1D 20 FD 15 20
     0978: FA 3D F2 68 09 D1 C9>F3
     0980: ED 5E 3E FF ED 47 3E F0
     0988: D3 08 3E 03 D3 08 D3 09
     0990: D3 0A D3 0B ED 4D F3 E5
```

This sets up 2 of the 4 interrupt vectors, Channel 0 at #FFF0 is set to #0780 which interestingly the MTX source labels to as vdpint and channel 2 at #FFF4 is set to #1C11 which is referred to as enter in the original sources.

The channel control word for channel 0 is set to #A5 which translates to 10100101 in binary.

Bit 7 set:	Enable Channel 0 interrupt
Bit 6 clear:	Timer mode – ie controlled from the 4mhz clock, and not the channel 0 input pin
Bit 5 set:	Pre-scale value is 256
Bit 4 clear:	Falling edge controlled
Bit 3 clear:	Automatic trigger when the time constant loads
Bit 2 set:	Time constant follows next
Bit 1 clear:	No channel reset
Bit 0 set:	This is a control word update

The main clock is 4MHz, divided by the 256 pre-scale value results in a counting rate of 15625 counts per second. The time constant that follows is 7D (125 decimal) 15625 divided by 125 results in the final 125Hz interrupt which vectors to #0780

The channel 2 interrupt isn't set up, even though the CTC setup code has put it into the vector table.

5.1.1 So why 125Hz?

That's probably because the CTC cannot produce any integer interrupt rates that are lower than that when running from the 4MHz main clock. Using a fractional rate isn't really practical when updating the system clock.

5.1.2 MTX BASIC and Interrupts

The MTX ROMs update the master clock on every interrupt in the routine "vdpint" at #0780. The last byte of the CLOCK system variable at #FD5D is incremented every "tick" from counting up from 48 to 173.The 6 bytes that form the data for CLOCK and TIME$ are updated when that counter hits 173. The count starts at 48 as that's ASCII code for a zero, and the remainder of the clock counts in ASCII to simplify the code required to set and read the clock in BASIC.

Once the clock is dealt with the system variable INTFFF at checked so that the main interrupt code is only run on alternate "ticks" which results in BASIC being a little more responsive as the longer codes sequence is only run 62.5 times a second on average.

If the interrupt code is run, INTFFF is used as a mask to determine which of 5 possible interrupt routines are called.

Bit 0: If set enables the interrupt driven sound
Bit 1: If set enables the break key check
Bit 2: If set enabled the keyboard repeat
Bit 3: If set enables cursor flash and Sprite movement
Bit 4: If set enables a call through USERINT at #FA98
Bit 5: If set enables a call through USERINT at #FA98
Bit 6: If set enables a call through USERINT at #FA98
Bit 7: If set tells the system to bypass the interrupt system this time around.
If more than one of bits 4 5 and 6 are set, the USERINT will be called once for each bit.

A BASIC program that runs in text mode, with no sound can run a around 3% faster if 64862 is poked with 6. For a game that doesn't want the break key operative, then poke in a 13 instead.

5.2 BAUD RATE CLOCKS

The CTC delegates 2 of the 3 active inputs to the baud rate counter.

The master clock for this is 4MHz divided by 13 using ap 74LS193 on the motherboard, which results in a not quite square wave at 307.7kHz (the signal is low for 2000 ns but only high for 1250).

The BAUD command from BASIC then programs the CTC to count those pulses. The baud rate table is stored in rom at #0CFA and defines 10 speeds from 19200 baud down to 75 baud. Each entry is 3 bytes, 2 bytes for the baud rate in binary, followed by the divider value to program into the CTC.

19200 baud is the fastest rate than can be achieved, and is the result of dividing the 307.7kHz input clock with the 16 count pre-scale and triggering on every pulse. The lowest rate of 75 baud requires the maximum 256 counter value with the 16 count pre-scale.

4,000,000 / 13 / 16 gives an output rate of 19231Hz, which is close enough to 19200 not to cause any communications issues. To get a “perfect 19200 would require the input clock to be 307.2kHz, which isn’t possible for a 4MHz main clock.

The output from the CTC on SER0 and SER1 is nothing like a square wave, the output is only low for 1 period of the 307.7kHz clock however that’s enough to clock the Z80 Dart on the communications board.

5.3 VDP INTERRUPT

While the VDP interrupt isn’t used by the MTX roms, it is connected to the highest priority input on the CTC. So games, and anything else that takes over the whole system can synchronise themselves to the VDP in order to use the horizontal blanking period for smooth screen updates.

What it needs is the CTC to be set to count pulses on channel 0, with a count of 1 so that every VDP interrupt trigger a CTC interrupt. The interrupt pointer at #FFF0 will then be called 50 or 60 times a second, depending on the VDP fitted.

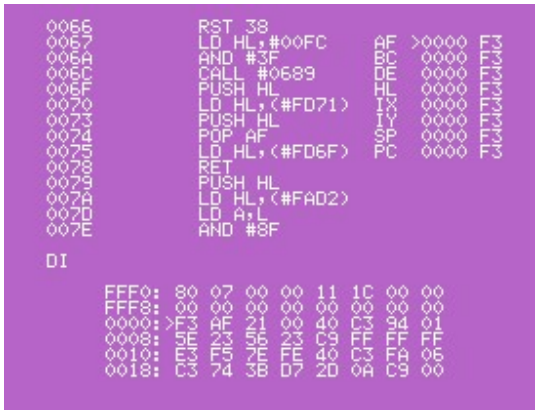
5.4 INTERRUPT PRIORITY

The Z80 has 2 interrupt sources IRQ and NMI pins. IRQ or interrupt request is the only one used by the MTX. The higher priority NMI isn’t used.

The IRQ system has 3 modes of operation, mode 0 which is 8080 compatible, mode 1 which vectors all interrupts through address #0038 and mode 2 which allows Z80 peripherals to supply an automatic vector address

The MTX uses mode 2. Support has been built into the rom for mode 1 operation. Code at #0038 jumps to the system variables at #FD4E which is labelled USRRST. By default, that in turn jumps to a routine called enter30 in the source at #1C03 which drops into Panel.

If an NMI is triggered by external hardware, the “support” provided by the rom is to issue the RST#38 instruction and call the code above.



There is a bit of a “gottcha” with this. Because RST #38 is a call and not a jump, it puts the return address on the stack. Any user routine that intercepts the jump through USRRST will need to remember to adjust the stack on exit otherwise the RETN will “return” control to #0067 instead of the real location.

The Z80 is designed so that there is an “chain” of devices that may issue IRQ interrupts to the CPU. The CTC is wired as the first device in that chain. Its interrupt will always have priority over any other device on the chain. Z80 peripheral chips have an interrupt enable in (IEI) and interrupt enable out (IEO) pin.

The IEI signal has to be high for an interrupt to occur. The IEI for the CTC is connected to the 5v rail through a resistor and is always high.

The IEO signal will be pulled low while the CTC is servicing an interrupt, any devices further down the chain will only issue an interrupt if their IEI pin is high. If their IEI pin is low, they set their own IEO pin low to maintain the integrity of the chain.

If there are too many devices (typically more than 3) on the chain, the time taken for the IEO to traverse the chain can cause issues, that’s not a problem with the MTX as the standard system only has the CTC on the chain. With the RS232 board connected the DART will also be on the chain.

6 THE PSG

The MTX uses the 4 channel Texas Instruments SN76489A Programmable sound Generator. The A version indicates it's the later 4MHz capable version. The PSG was used in a number of systems in the late 70's early 80 most notably in the BBC Micro. The PSG mono device so although the MTX has a "HiFi" port on the back, it's only a single channel, and not stereo.

The chip itself is relatively slow as it's basically a 500kHz device (The original SN76489 with a divide by 8 on the clock input.) and needs 32 cycles at 4mhz to complete a read of the data bus. Rather than hold up the CPU while the data is transferred, Memotech devised a hardware solution that doesn't use "ready" from the sound chip to pause the CPU.

Instead OUT (6),data writes to a 74LS374 8 bit latch. IN A,(3) then starts the PSG data read from the latch. The Data returned from reading port 3 is irrelevant, it's the action of reading the port that starts the access, there's actually no valid data being output at that point in time to read.

Since the PSG is reading from the latch it can take as much time as it needs, the CPU is then free to carry on processing. The manual warns that there should be at least 32 T states (or CPU cycles) between reads. That's not quite accurate, there needs to be 32 T states between the IN (3) and the next OUT (6). If the data in the latch is updated before the PSG completes the read, the data is corrupted.

The MTX ROMs have a routine at #093A (2362 decimal) to turn off all 4 sound channels. Calling #0953 (2387 decimal) will sound the bell, without needing to go through the overhead of using the VDU routines.

6.1 PSG REGISTERS

The PSG has 8 internal registers, each of the 4 channels has its own volume and frequency register. Internally the volume is stored in 4 bits, and the frequency counter in 10 bits. The nose channel treats the received data differently from the 3 tone channels but the update method is the same.

The data sheet refers to one and 2 byte data transfers, with a 2 byte transfer to update the frequency registers and a one byte transfer to update the noise and volume registers. Because of the way that the PSG latches the channel information the frequency can also be updated with a one byte update.

Which part of a PSG update is in progress depends on bit 7 of the data byte sent to port 6.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Frequency update	0	Don't care	F9	F8	F7	F6	F5	F4
Source update	1	Ch1	Ch0	D/V	X3	X2	X1	X0

If bit 7 is high indicating a source update the PSG will latch the 2 channel bits and the D/V bit, and then sent the 4 low bits to the appropriate internal register.

Channel bits	Destination Channel
00	Tone 1
01	Tone 2
10	Tone 3
11	Noise Generator

If the D/V bit is low, then the destination is the frequency, if it's high the volume register is selected. The 4 low bits then set the low 4 bits of the frequency count or the volume.

The volume control actually works in reverse, the 4 bis control the amount of attenuation of the signal. So, 0 is maximum volume and 15 turns that channel off.

The 10 bits of frequency control the internal counter that flips the output when the counter is reached. Low values therefore produce a high note and high values a low note.

The frequency produced is the 4MHz MTX main clock divided by 32 times the register value. Which simplifies to:

Frequency = 125000 / Count, or

Count = 125000/frequency

To output the ISO standard note A4, which has a frequency of 440Hz, the count value would be 125000/440, the result isn't an exact integer so the closest value is 284. To play that note on the tone 1 channel at maximum volume, the following values need to be sent to the PSG #8C, #11 and #90

Examining those in detail:

#8C in hex is 1 000 1100 in binary. The 3 sections of the value highlight it's a source update, the channel is 00 for tone 1, D/V is zero as it's the frequency update and 1100 is the low 4 bits if 284 which is 0100011100 in binary.

#11 is 0 0 010001 in binary. The first zero is the frequency update marker, the 2nd one is a don't care bit, it could have been a 1 without altering the output, and 010001 is the top 6 bits of 284.

#90 is 1 001 0000 in binary, so another source update, this time it's the tone 1 volume register, and the attenuation is zero for the loudest output.

6.2 Driving the PSG

MTX BASIC has a rich set of sound controls, however the PSG can also be driven through the port 6/port 3 update process in the same way as would be done in assembler. The bit fields are clearer in hex, but that's not available in BASIC, so the examples have to use decimal.

```
 LIST
10 OUT 6,144
20 LET A=INP(3)
30 OUT 6,128
40 LET A=INP(3)
50 OUT 6,63
60 LET A=INP(3)


Ready
```

Lines 10 and 20 set the volume for tone 1 to maximum (144 is #90 hex or 1 001 0000 binary)
Lines 30 and 40 set the low counter bits for tone 1 to zero (#80 hex or 1 000 0000 binary)
Lines 50 and 60 set the upper counter bits to 63 so the count is 1008 and the output frequency 124hz or so. (#3F or 00 111111 binary).
Entering Ctrl G from the keyboard, to sound the bell, will kill the output.

The BASIC program can the be extended to illustrate the one byte frequency update. The extra 4 lines progressively reduce the count to zero, increasing the output frequency.

```
 LIST
10 OUT 6,144
20 LET A=INP(3)
30 OUT 6,128
40 LET A=INP(3)
50 OUT 6,63
60 LET A=INP(3)
70 FOR X=63 TO 0 STEP -1
80 OUT 6,X
90 LET A=INP(3)
100 NEXT


Ready
```

Changing the code to output to the volume register instead of the frequency produces a different effect.

```
 LIST
10 OUT 6,144
20 LET A=INP(3)
30 OUT 6,128
40 LET A=INP(3)
50 OUT 6,63
60 LET A=INP(3)
70 FOR X=144 TO 159
80 OUT 6,X
90 LET A=INP(3)
95 PAUSE 50
100 NEXT


Ready
```

BASIC is sufficiently slow, that there is no consideration of the timing between port 6 updates.

The single byte frequency update illustrated in the 2nd example needs the previous source update to be the tone register. Swapping lines 10 and 30 over so that the volume register is the previous source can produce some “interesting” effects.

7 THE KEYBOARD

The MTX has a 79 key keyboard. 2 of those keys are the reset keys, the other 77 are the input keys. The 2 sections are totally independent. Pins 1 to 18 of the 20 way keyboard cable handle the main section which is arranged as a 10 by 8 matrix, with 3 unused keys. Pins 19 and 20 are connected to the reset keys.

7.1 RESET KEYS

The reset keys cannot be read from assembler or BASIC, and because they are attached directly to the CPU's reset pin, they cannot be blocked or intercepted. While this ensured that the user can always regain control from a run-away program, it does have issues for program security.

The MTX only clears the 16k common RAM on reset, the memory detection system also corrupts a small amount of RAM in the MTX512, but not the MTX500. Loading a typical early game on the MTX500 and resetting then allows the game code to be examined in the panel and saved out.

The 2 reset keys are connected in series so that both have to be pressed to activate the CPU reset. The hardware on the motherboard deals with any key-bounce etc. to ensure the reset pin is activated in line with the manufacturer's recommendations.

7.2 THE MAIN KEYBOARD

BASIC provides the INKEY$ function to read the keyboard, in assembler the equivalent is CALL #0079. The keyboard scanning code is the same for both options, and will stop at the first key ASCII detected. The scan will not detect the shift or control keys, nor will it detect multiple keypresses. It will however process the control or shift key's effect on the key detected.

Reading multiple keys, requires accessing the hardware directly. Care has to be taken when doing this, as the break key is read during the 125hz interrupt processing and so can result in an incorrect read if precautions aren't taken.

The keyboard matrix is based on an 8 by 10 grid, which is too big to read through a single 8 bit wide I/O port therefore 2 ports are used.

The matrix itself consist of 8 "Drive" lines and 10 "Sense" lines. Sending a value to output port 5 will set the drive lines. The reading from input port 5 will read 8 of the 10 Sense lines, input port 6 is used to read the other 2, input port 6 has other functions so only the bottom 2 bits are relevant to reading the keyboard matrix.

The sense lines are normally pulled high by resistors. If a key is up, or the drive line is high the resistor ensures the input is a high. To be detected the key attached to that line has to be down AND the drive line for that key also has to be set low. That makes it possible to detect an individual key.

The ROM's keyboard scanning routine therefore sets each drive line low in sequence and then reads the 2 input ports. If port 5 reads as anything other than #FF (255 decimal 1111 1111 binary) then there is a key down. Similarly, if the lowest 2 bits of port 6 reads as anything other than #03 (xxxx xx11 binary) there is a key down on that part of the keyboard.

For something simple, like Magrom when it checks for the space bar being pressed on boot, the code simply needs to set 1 drive line low, and check one sense line. For the space bar the code is:

```
LD A,#7F
OUT (5),A
IN A,(6)
AND 1
RET NZ
```

Breaking that down #7F is 0111 1111 in binary so drive line 7 is being set low. There's no requirement to wait a minimum number of cycles, the keyboard data is immediately available. In this case the space bar is on sense line 8, port 6, bit 1 needs to be tested. If the key isn't being pressed the AND 1 instruction will return 1, and the code will exit RET NZ. If the key is down, the result of the AND 1 is zero and the code continues after the RET NZ and starts up the games ROM.

This particular snippet doesn't need to take any precautions for interrupts as it's run before the system sets those up. However, once the interrupts are running, there is a system variable LASTDR at #FD7E specifically set up to preserve the drive status across an interrupt. Adding one line to the above snippet is all that's needed.

```
LD A,#7F
LD (#FD7E),A
OUT (5),A
IN A,(6)
AND 1
RET NZ
```

On exit from the break key test, the interrupt code will put that value stored in LASTDR back on the port 5 drive lines.

The keyboard matrix looks like this:

Sense line	Port 5 Drive line							
Port 5	Bit 7	Bit 6	Bit 5	Bit 4	BIT 3	Bit 2	Bit 1	Bit 0
Bit 0	**Z**	**L shift**	A	caps	Q	ctrl	esc	1
Bit 1	**C**	X	D	S	E	W	2	3
Bit 2	**B**	V	G	F	T	R	4	5
Bit 3	**M**	N	J	H	U	Y	6	7
Bit 4	.	,	L	K	O	I	8	9
Bit 5	_	/	:	;	@	P	0	-
Bit 6	Ins	**R shift**	Ret	]	Linefeed	[	^	\
Bit 7	Cls	**Down**	**Home**	**Right**	**Left**	**Up**	Eol	Page
Port 6								
Bit 0	**Space**				Del	Tab	BS	BRK
Bit 1	**F4**	**F8**	**F3**	**F7**	**F6**	**F2**	**F5**	**F1**

The emboldened keys highlight some of the decisions made by Memotech when the keyboard was laid out.

The 2 shift keys share a common drive line so can be read in a single read of port 5.

The cursor keys (and the right joystick port that mimics them) have a common sense line, as do the 8 function keys.

Like the shift keys, the 5 keys returned by the left joystick port (Z C B M and space) share a common drive line making that port a little easier to read, though most games seem to have gone with right joystick port.

The 3 unused spots in the matrix are all in bit 0 of port 6.

8 EXPANSION POTENTIAL

The standard MTX console has plenty of expansion potential. There are edge connectors on both sides of the main circuit board as well as a 20 pin DIP socket in the centre of the PCB with 8 input and 8 outputs, that can be read/set using I/O port 7.

The original Memotech memory expansion fitted internally as did the RS232 communications board. The RS232 board also provided the pin header to connect the FDX disc system which had additional internal space for the various boards of the CPM system. There is sufficient space under the keyboard for the RS232 board and one of the ROM or RAM boards.

The external edge connector was used by the SDX, ROMpaks and Speculator. It's the easiest expansion for the home user to access, as it doesn't require opening the case.

The internal and external connections are mirror images of each other, with the A1 (component side) and B1 connection closest to the front of the keyboard.

They are not quite identical. The circuit diagrams in both versions of the manual show a link between position A25 and the IEO (input enable output) from the CTC. That link is actually only present on the external connection (which is J1) and is open by default. The internal connection (j10) does not have the link as the connection is always made.

The Z80 DART on the RS232 board needs to be connected to the Z80 interrupt chain. The IEO signal from the DART is then passed on to the FDX if connected. If there is an expansion on the external connector that needs to be added to the interrupt chain, then a modification the PCB is needed.

The RS232 communication board would then be incompatible with the external hardware, as the DART and new interrupt source would both think they were 2nd in the priority chain after the CTC.

The MTX used a 30 x 2 way 0.1" edge connector. This was a pretty standard thing at the time. To save money, the connections are not gold plated and are simply solder "plated" by the PCB assembly process. That can cause ussies with poor connections between boards because of dirt or tarnishing. It's not unusual to see systems where the main PCB and an expansion board have the edge connector replaced by a permanent soldered connection.

Another difference between internal and external expansions. The board around J1 the external is shaped to take a closed ended edge connector, while J10 has no cut-outs and requires an open-ended connector.

This makes the keyway in position 5 redundant for J1 when using a closed ended connector. For J10 it's very much needed if using less than 160mm full height expansion board. For a full height board, the case rails maintain the correct position for the connector.

The layout of the edge connector, and the probable reason for the keyway, is set up to allow the ROMpak to use a sub set of the connections, and therefore a smaller (and cheaper) 15 x 2 way edge connector. The ROMpak itself just needs to be big enough to hold a single 64kbit ROM or (E)EPROM. Which conveniently is not much bigger than the edge connector making a nice compact package.

8.1 CONNECTOR LAYOUT

Component (A) Side	Position	Solder (B) Side
SER #2	30	0V
RE/CPM	29	SER #1
R1	28	R2
P3	27	R0
P1	26	P2
IEO (J10 only)	25	P0
*NMI	24	*INT
*WAIT	23	*BUSREQ
*HALT	22	*BUSAK
PHI	21	*RFSH
*WR	20	*M1
*IORQ	19	*RD
*RESET	18	*MREQ
0V	17	0V
12V	16	-V
5V	15	5V
D6	14	D7
D4	13	D5
D2	12	D3
D0	11	D1
A14	10	A15
A12	9	A13
A10	8	A11
A8	7	A9
A6	6	A7
Keyway	5	Keyway
A5	4	0V
A3	3	A4
A1	2	A2
*GROM	1	A0

8.2 CONNECTING A ROMPAK OR EXTERNAL 8K (E)EPROM

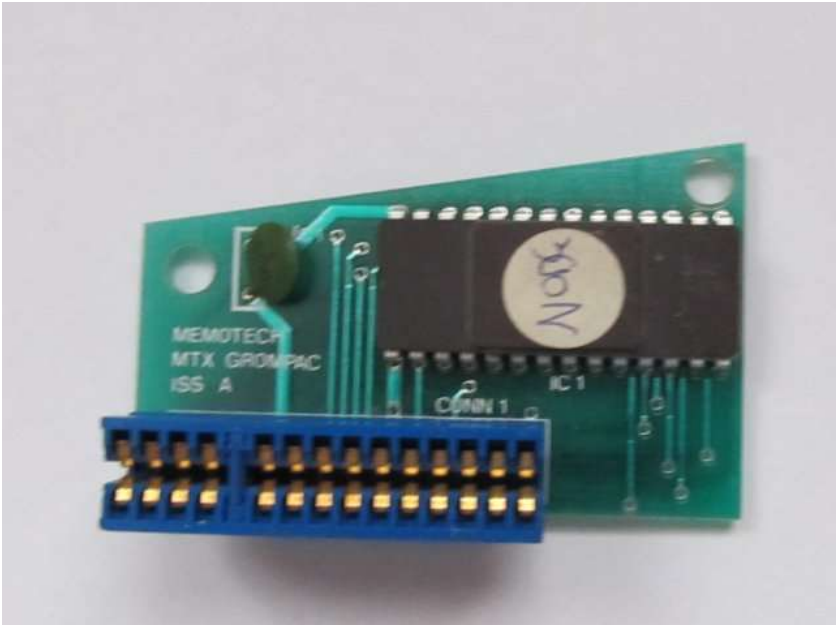
The coloured sections are the signals used by the ROMpak The PAL within the MTX decodes the memory range used by the paged ROMs but doesn't have enough output pins to provide a chip select for both the internal paged ROM (*CE64B) and the external one (*GROM).

Instead, the PAL will output a low if the correct memory range is used and either paged rom 1 or 7 is active. Paged ROM 7 decoded separately by 3 diodes, the rest of the hardware combines that, with the dual chip select from the PAL to determine whether *CE64B or *GROM is active.

Because the PAL includes the *RD signal from the Z80 in its equation, neither rom select will be active if the Z80 is writing, which means there is no need for *RD itself to be passed to the ROMpak. For *GROM to be active the Z80 is reading data.

An (E)EPROM connected externally would need both the chip select and output enable to be connected to *GROM. Depending on the device being used the *WR (write enable) on an EEPROM or Vpp (programming voltage and *P (program enable) pins on an EPROM will probably need to be tied to 5v.

The power, data and address pins are wired 1 to 1. The only other component would be an optional capacitor across the power rails, the one in the MTX ROMpak in the photo is 47nf, 100nf would also be a common option. The visible traces show how simple the connections are. And how short the 15 x2 edge connector is.



Anything other than a ROM board is likely to need multiple signals from the other half of the edge connector so in most cases it makes sense to fit the full 30 way connector. For mechanical stability if nothing else.

8.3 CONNECTING RAM TO THE EXTERNAL CONNECTOR

In addition to the full 16 bit Z80 address bus, the expansion connector also carries all 8 of the page port signals discussed in the section on the Memory Map. When the MTX was designed, static ram would have been too expensive so the RAM boards used dynamic memories. The MTX memory boards required 6 support chips alongside 16 Memory chips to add 128k. Modern static RAM in the sort of sizes useful for expanding the MTX is cheap, 128k can be added using just 1 memory chip and one GAL to control the memory mapping.

The simplest, and probably most useful memory expansion would be to add the extra 32k to a MTX500 to give it the full 64k of the MTX512

The extra RAM in the MTX512 has 2 possible locations in the memory map controlled by the RE/CPM signal in position A29. If that signal is low, the MTX is in ROM mode, and the 32k is split between the 16 to 32k block in page 0 and the 32 to 48k block in page 1. When RE/CPM is high the MTX is in CPM mode and the extra memory fills the gap from 0 to 32k in page 0.

Connecting the RAM is straight forward, A0 to A14 and D0 to D7 on the edge connector connect to A0 to A14 and D0 to D7 on the ram chip. For static RAM, there is no action requirement for A0 to be connected to A0, or A1 to A1. As long as all 15 address pins connect to 15 different address lines, that's all that is required. The same applies to the data lines. If swapping the connections around makes for a simpler board layout there's no reason not to do so.

Static RAM is directly compatible with the Z80's *RD and *WR signals so they connect to output enable and write enable respectively. It's the job of the GAL to provide the chip select. 2 inputs are required to take the A14 and A15 address lines from the CPU, plus 5 more to take RE/CPM and the 4 RAM page signals from the page port. One additional signal is needed and that *MREQ as the memory only needs to respond to memory requests, and not to any I/O port (or the dynamic RAM refresh signal). With one output that easily fits into a 20 pin 16V8 GAL.

CPM mode is the easiest to decode, the chip select needs to be active when all of the following conditions are met:

RE/CPM is high – as this is a CPM mode decode
A15 is low - indicating memory in the 0000 to #7FFF bank is being accessed
P0 to P3 are all low – The chip should only respond in page 0
*MREQ is low – only activate the chip on a memory access

For ROM mode there are 2 possible circumstances in which the chip should be activated:

RE/CPM is low – as this is a ROM mode decode
A15 is low, A14 is high - indicating memory in the 4000 to #7FFF bank is being accessed
P0 to P3 are all low – The chip should only respond in page 0
*MREQ is low – only activate the chip on a memory access

RE/CPM is low – as this is a ROM mode decode
A15 is high, A14 is low - indicating memory in the 8000 to #BFFF bank is being accessed
P0 is high, P1 to P3 are all low – The chip should only respond in page 1
*MREQ is low – only activate the chip on a memory access

And that's it, 8 inputs, 1 output and 3 equations. The exact format of the equations depends on the programming environment being used.

In WinCUPL supplied by Atmel for programming the ATF series of CPLD's the code would look something like this.

```
Name     MTX32K ;
```

```
PartNo   01 ;
Date     24/03/2018 ;
Revision 01 ;
Designer Engineer ;
Company  None ;
Assembly None ;
Location  ;
Device   g16v8 ;

/* 32k RAM decode for MTX500 to MTX512 expansion */

/* *************** INPUT PINS *********************/
PIN    1  = RECPM  ;
PIN    2  = A15    ;
PIN    3  = A14    ;
PIN    4  = MREQ   ;
PIN    5  = P0     ;
PIN    6  = P1     ;
PIN    7  = P2     ;
PIN    8  = P3     ;

/* *************** OUTPUT PINS *********************/
PIN    15 = RAMCS
 ; /* Pin 15 and 16 are outputs in simple mode */

FIELD ADDRESS = [A15..0] ;
FIELD PAGE = [P3..P0] ;

!RAMCS =
   RECPM & !MREQ & ADDRESS:[0000..7FFF] & PAGE:0    /* CPM mode     */
# !RECPM & !MREQ & ADDRESS:[4000..7FFF] & PAGE:0    /* ROM mode page 0 part */
# !RECPM & !MREQ & ADDRESS:[8000..BFFF] & PAGE:1    /* ROM mode page 1 part */
 ;
```

For memory expansion beyond 32k the GAL also needs to control the additional memory address lines, however for 32k, that's not required, and the chip select is the only output.

8.4 Connecting Input/Output devices to the External connector

To maintain compatibility with the Intel 8080 the Z80 CPU maintains a separate address space for input/output device. On the 8080 there were 256 "port" addresses, as only 8 bits were made available to address them. The Z80 extends this to 16 bits, giving the 64k of possible ports.

The MTX hardware only decodes the 8 bit Intel compatible addresses. The first 32 ports are defined as internal to the keyboard unit, the rest are allocated to the FDX.

This can cause compatibility issues with the RS232 board. The Z80 DART on the expansion is mapped to ports #0C to #0f (12 to 15 decimal) in line with the internal allocation guidelines. The decode PAL then passes any access to ports in the FDX range to the 60 pin expansion header.

With one of the alternative storage systems attached (CFX, REMEMOorizer etc), which use high numbered ports the actions of the PAL causes contention on the data bus, as both the storage device and the RS232 board are loading the bus.

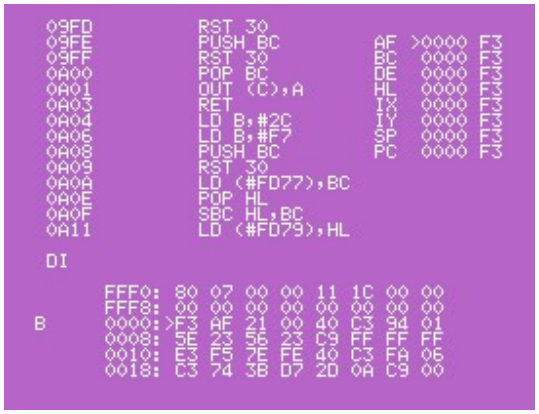
Alternatives to the PAL on the RS232 board are available if it needs to coexist with one of the modern storage devices.

While the MTX hardware only uses 8 bit port, the MTX ROM used 16 bit port, well sort of.

In BASIC, the OUT (port),data will accept a 16 bit value for the port. Try OUT 257,65 to see this in action. The port address is truncated to 8 bits by the hardware, but the full 16 bits address is actually put onto the Z80 address bus.

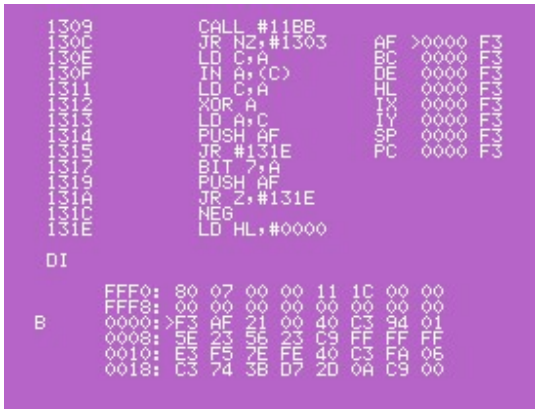
However, the INP(port) function will only take an 8 bit address, entering PRINT INP(257) as a command will stop with an error.

The reason for this it the OUT command, which is processed at #09FD in the ROM, reads 2 16 bit values from the BASIC line (the 2 RST 30 instructions showing in the panel listing reads a 16 bit value to BC and copies the low byte to A) and uses them in the OUT (C),A instruction.



The input function on the other hand uses totally different code. Which range checks the value. The MTX programmes could actually have saved some space if INP hadn't been restricted to 8 bits.

The INP code at #1309 first calls a routine to read a small integer, which tests the top byte for zero before returning. The INP code branches to an error routine if the top byte isn't zero before reading a 16 bit address anyway. Had the code simply called the read a 16 bit integer routine it would have enabled the full input range and removed the code for the branch.



So, a potential “upgrade” to 16 bit input would be to replace the 2 bytes at #130C and #130D with zero. For most MTX users that’s not really an option. However, with access to a rom programmer and a system with the 2 rom, 4000-04 main board which has configurable links, replacement (E)EPROMS could be fitted to update the “OS” portion of the roms.

9 PROJECTS

Starting with “Magrom” in 2013 I’ve designed a number of add-ons for the MTX, some of which have been made available to the public and some that haven’t. Since initially, they were all for my own use, the information available for them varies from the comprehensive pages on Dave Stevenson’s Primrosebank.net, to the non-existent.

I won’t claim any of them are ”perfect” designs, however, they do stand as examples of what can be done, and so documenting them may give ideas as to what can be done better!

9.1 MAGROM

The original Magrom idea, was simply to cram as many games as possible onto a large capacity EPROM and make them available via some sort of menu system. That would reduce the loading time from minutes to seconds and remove the need for an unreliable and increasingly hard to find tape cassette player. The name was Dave’s invention initially I referred to it as just the Games ROM.

There were 5 hand-built prototypes, before the design was finalised and Dave has subsequently produced 3 versions of the PCB. Because the design has evolved over so many steps, it probably isn’t the best. However, a re-design at this stage wouldn’t serve any purpose.

The original concept design arose from 2 observations on the nature of the MTX

- Most games were written to run on the 32k MTX500
- The MTX memory map typically has a half page of ram in the “last” page

MTX500 games only use memory from #8000 upwards, as that is all that is fitted on the 32k MTX. That leaves room on the MTX500 for 16k of data ROM to occupy the normally empty area from #4000 to #7FFF. The MTX 512 has 64k of RAM, however, due to the paging of that memory on page 1, there is only ram from #8000 upwards, the same gap from #4000 to #7FFF is available.

The Magrom board provide a manual system select jumper so that the hardware knows which RAM page to place the data ROM in. Though “unusual” this is quite safe, the system itself doesn’t apply any conditions as to what appears in the various places in the memory map. As long as the electrical and timing requirements are met.

However, a data ROM sitting at #4000 won’t be detected on start up. For the board to auto run it must be found as a paged rom at #2000. One way of doing this would be so fit 2 memory devices, an 8k control ROM in a paged rom slot, and a data ROM in main memory.

Fitting 1 rom in the paged rom area, and then dividing that up into subpages is an option. However, that has the disadvantage of needing an area of RAM to put the data transfer code in, otherwise the system would crash as soon as a sub page is activated.

Since I didn’t know if there would be any RAM areas than no game used to put that code into it seemed best to run the control code as a paged ROM, and the put the data in to the empty memory area where it could be paged without risking a crash, or using any RAM.

Fitting 2 ROMs increases both the cost and the size. The Magrom design therefore uses a single ROM and has it respond to access requests to both memory areas.

The MTX's default rom design allocates ROM7 as the "games rom" and allocates a signal on the expansion connector labelled GROM to activate it. Early prototype versions of the Magrom decoded the ROM page for themselves but this was removed before the first PCB version to save components. It then had to be added back in on the V1.1 PCB because of issues using ROM 7 on some "international" versions of the MTX.

The size of the data rom is dictated by part availability, 512k flash ROM is available in an easy to handle DIP package and at a MTX compatible 5 volts. 512k would leave room for 15 32k long games plus the controlling software. As it turned out, none of the MTX500 games are close to being that large and so there was room for more than double that.

9.1.1 Magrom Software

The Control ROM occupies almost the full 8k allocated to a paged ROM. However, over 7k of that is the Graphics mode 2 screen that appears at start up. The controlling code is only 600 bytes or so, and is included in full below, typos and all.

```
NAME GraphicRom105
TYPE rom

;Change History
; Version 1.05 Rom6 comatibility changes for production version V1.1
; version 1.02 Further changes to accomodate REMEMOrizor compatibility
; version 1.01 ROM 7 entry point now in use, do nothing skeletom removed.
; Version 1.00 release version, cosmetic changes from 0.22 only
; Version 0.22 I/O port changed to &FB to match hardware revision 5.1
; version 0.21 Spacebar check added on startup.
; Version 0.20 CTC changes rolled back to V0.18 due to causing crashes
; Version 0.19 CTC and other setup changes
; Version 0.16 Keybaord range extended for up to 38 games
; Version 0.15 Graphics 2 mode used, redundant code deleted.


; temporary storage, there arw 3 unsued system variable bytes at FA8C

sector EQU &Fa8D
key EQU &fa8E

port EQU &FB
lstpg equ &fa7a
sstack equ &fa96
page equ &fad2
setcall EQU &fd48
```

Once past the revision notes and the equates, the actual ROM starts with a standard MTX auto run header, the paged rom ID pointer at #2008 isn't populated, just the auto-run and ROM x entry points

```
; Code starts at &2000, as only the upper half of the 16k rom space is paged.
ORG &2000
```

```
;if bytes 0 - 7 are 8-1 resp. autoboot rom via &2010 on power up/reset
;but after high memory cleared, any variables set will be retained
DB 8
DB 7
DB 6
DB 5
DB 4
DB 3
DB 2
DB 1

; BASIC's ROM command enters at &200C
ORG &200C
JP run
```

The auto-run entry point checks for the spacebar, and will exit if not pressed, so that the MTX can be used normally while the MAGROM board is fitted, which allows for internal fitting. Changing the RET NZ to RET Z would have the MAGROM run unless spaces is presses, and was included in a one off build for a system used to display the MTX at retro shows.

```
;boot entry point &2010
org &2010
; first test for space pressed
; can exit via a return as the startup code
; is entered via a call.

LD A,&7F
OUT (5),A
IN A,(6)
AND 1
ret NZ
```

Because the auto-boot rom is called very early in the start up sequence the control rom needs to duplicate some of the hardware setup that BASIC performs. Turning off the sound and setting up the VDP are necessary, initialising the system VDU driver was required by some, but not all of the games, to get them to run.

```
;no spacebar check if entering from "ROM 7" command
;enters here to ensure a clean stack.
.run
;setup the stack
ld hl,setcall
ld (sstack),hl
ld sp,(sstack)

;the sound chip seems to default to making a noise at startup,
; and we're loading before the OS kills the volume
;so kill all 4 channels
call sound_off
```

```
;this is the MTX VDINIT routine, which is run immediately after
;the autostart rom check.
rst &28
db &42


; select and clear VS 4 using RST 10 functions, which now work,
;thanks to VDinit
rst &10
db &4C
;set the border to black
LD a,&F1
out (2),a
ld a,&87
out (2),a

; make sure the CTC iterrupts are off just in case
call ctc_off
```

Having set up the hardware, the memory map now has to be configured. The memory test run on boot will have identified how much ram is fitted, and so the RAM page has to be set so that the “current” page is the one that has the data ROM at #4000.

```
; make sure we're in the correct page (0 for 32k, 1 for 64k,
; 3 for 128k; 4 or 11 for 384k
; (Andy's revised rom on REMEMOrizor sets lstpg to 4, instead of
; 11. However all 11 full pages still exist)
; or the data rom wont be acessible
ld a,(lstpg)
and &0f
cp 4
jr nz,change_page
ld a,11
.change_page
ld b,a              ;V1.05 changes: save the ram page number
ld a,(page)
and &70             ;get the current rom page - won't always be 7 any more
or b                ;add back the ram page
ld (page),a         ;save to the OS first in case of interrupt
out (0),a           ;set the hardware
```

Once the configuration so done, the menu needs to be displayed and the start-up beep sounds

```
CALL welcome_page
call welcome_beep
```

The main loop is exceedingly simple. It reads the keyboard, the keycode is used to access the list of where each game starts in the data ROM, and then load & run that game if there is one allocated to that key.

```
.key_loop
call readkey        ;returns an index value in A only accepts 0-9 and A-Z
and &3f
LD HL,directory
add a,l
ld l,a
ld a,(hl)
ld (sector),a      ;save the location for later
inc a
jr z,key_loop      ;loop if empty entry
call loadgame      ;wont exit unless the game tries to exit to basic (ie reversi)
jp run             ;so re-start if it does
```

The subroutines:

The first one sounds the double beep on starting.

```
.welcome_beep
;now make the hello! beep
ld hl,20000
call beep_hl
ld hl,0
call delay
call sound_off
ld hl,20000
call delay
ld hl,28000
call beep_hl
ld hl,30000
call delay
call sound_off
ret
```

The game loading routine is pretty straightforward. All the games are in the RUN format used by the SDX and FDX disc systems. On the disc USER RUN “xxx.RUN” is used the access a lot of the early Continental Softwaree titles. They’s basically binary images of the game, with a 4 byte header containing the size and load/start address. Being a binary dump made them ideal for Magrom.

The 512k of data rom is split up into 256 “sectors” each 2K long. This allows the catalogue enry to be a single byte. The loading code just copies up to 2k at a time from the data ROM to main RAM, and jums into the entry address once completed.

```
;catalogue entry in HL
.loadgame
call find_sector
```

```
ld e,(hl)
inc hl
ld d,(hl)         ;DE now holds the loading address
inc hl
ld c,(hl)
inc hl
ld b,(hl)         ;BC now has the file size
inc hl            ;HL now points to the first byte which is also
                  ;the codes entry point
push de           ;push the start address onto the stack
push BC           ;save the file size
ld BC,&7fc        ;first sector is 4 bytes short because of the
header
LDIR              ;transfer the short sector, no file size check
                  ; as all entrys must be at least one sector
ld bc,&f804       ;-&7fC
.block_loop
pop hl            ;get the remaining file size
add hl,bc
ret nc            ;if Carry isn't set then there were less than
                  ;&800 (7FC) bytes left  so exit via run address
                  ;stacked above
push hl           ;re-stack the remaining file size
ld bc,&800        ;and set BC for a full sector transfer
ld a,(sector)
inc a
ld (sector),a
call find_sector
ldir              ;transfer 1 sector to DE
ld bc,&f800       ;set BC to -&800 for 2nd and later sectors
jr block_loop
```

A graphics mode 2 screen takes up 12k of the video memory, 6k for the bitmap and 6k for the colour map. Rather than take up one of the data ROM pages, the bitmap is compressed before being loaded into the ROM image. The compression ratio is around 2:1, nothing special but it does allow for simple and speedy decompression by the Z80 CPU and it's enough to fit everything into the 8k available.

```
;Screem character map for graphics mode is stored at &0000,
;the colour data is stored at &2000, both tables are &1800 bytes
;long bit 6 of 2nd transfer set to indicate writing to Vram
;the screen data is compressed, &FF isn't used anywhere in the
;screen builder so is the flag for compression
;&FF,byte,count, for a compressed chunk
;&FF,FF is the end of table marker, but the byte count should
; stop the transfer immediatly before them.

.welcome_page
push af
push de
push hl
```

```
LD HL,screendata
LD de,&1800
ld BC &4000
call decomp
LD HL,colourdata
LD de,&1800
ld BC &6000
call decomp
pop hl
pop de
pop bc
ret

.decomp
LD A,c
OUT (2),A
LD A,b
OUT (2),A
.decomp_loop
ld a,(hl)
inc hl
;check for compression marker
cmp &ff
jr z,compressed
out (1),a
dec de
ld a,d
or e
jr nz,decomp_loop
ret

.compressed
;get the data byte
ld c,(hl)
inc hl
;get the count
ld b,(hl)
inc hl
.comp_loop
ld a,c
out (1),a
dec de
;check to see if we're end of the bitmap
ld a,d
or e
ret z
DJNZ comp_loop
jr decomp_loop
```

Reading the keyboard is done using the MTX's built in keyboard routine, and unwanted keycodes are stripped out before returning to the main loop.

```
;MTX's main keyboard routine is at &0079, and returns Z set if no
key
;or the ASCII code in A

.readkey
call &0079          ;read keyboard,
jr z,readkey        ;wait for something to be pressed
cp +ASC"0"
jr c,readkey
cp +ASC":"
jr c,zero_to_nine
and +%11011111      ;take care of lower case
cp +ASC"@"
jr c,readkey
cp +ASC"\"
jr c,a_to_z_ish
jr readkey

.a_to_z_ish
sub 6
.zero_to_nine
sub +asc"0"
ld (key),a
RET
```

Each 16k rom page hold 8, 2k sectors, so a 3 bit shift or rotate will convert the sector number to a page number. The 3 low bit then also need shifting 3 places to form the 2k offset within the 16k page.

```
;"usefull" routines here

;return HL pointing to the start of a 2k secotor, with the
; relevant chunk paged in

.find_sector
push af
ld a,(sector)  ; sectors are 2k, banked pages 16k, so divide by 8
RRCA
RRCA
RRCA
and &1f
out (port),a    ;page the rom
ld a,(sector)  ;now work out where the load data is on the page
and &07        ;intra page offset, needs multiplying by 2k
add a,a
add a,a
add a,a
add a,&40     ;page offset in memory
ld h,a
```

```
ld l,0
pop af
ret
```

The CTC shut off routine pokes the software reset command to each of the 4 channels twice. Two reset commands are required just in case the CTC was part way through command on when the system was reset. On power up the duplicate command wouldn't be necessary.

```
;generic intrrupt cancelling routine.
.ctc_off
push bc
push af
ld b,2
.ctc_loop
ld a,3
out (&8),a
out (&9),a
out (&a),a
out (&b),a
djnz ctc_loop
pop af
pop bc
ret
```

The sound off routine uses NOPs to ensure the PSG read timing is satisfied as there's no compelling reason to do anything more efficient space wise.

```
; set all 4 sound channels to silence, using the data table that follows
.sound_off
PUSH AF
PUSH BC
push hl
Ld hl,sound_data
ld b,12
.sound_loop
ld a,(hl)
OUT (6),A
IN A,(3)
nop         ; at least 32 cycles needed  between accesses
nop         ; 8 nops used to ensure the requirement is met
nop
nop
nop
nop
nop
nop
inc hl
DJNZ sound_loop
pop hl
pop bc
pop af
```

```
ret

.sound_data
db &80,00    ;Channel 0, frequency 0
db &9f       ;channel 0, attenuation off
db &A0,00    ;Channel 1, frequency 0
db &Bf       ;channel 1, attenuation off
db &C0,00    ;Channel 2, frequency 0
db &Df       ;channel 2, attenuation off
db &Ff       ;channel 3, attenuation off
db &E0,00    ;Channel 3, periodic noise N/512
```

Straight forward delay of approx. 26 x HL cycles used by the start-up sound code.

```
.delay
PUSH HL
PUSH BC
.delay_loop
DEC HL
LD A,H
OR L
JR NZ,delay_loop
POP BC
POP HL
RET
```

The beeper code uses the HL value to sound a note on the tone 1 channel

```
.beep_hl
push AF
push BC
LD A,L
AND &0F
OR &80
OUT (6),A    ; SEND TONE 1 + 4 BITS OF FREQUENCY
NOP
IN A,(3)
LD A,L
SRL A
SRL A
SRL A
SRL A
LD C,A
LD A,H
SLA A
SLA A
SLA A
SLA A
OR C
AND &3F      ; REMAINING 10 BITS OF FREQUENCY
OUT (6),A
```

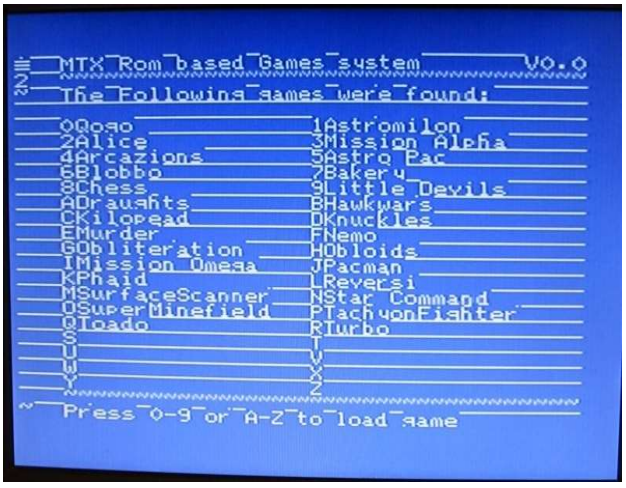
```
NOP
IN A,(3)
LD A,&90      ; ATTENUATION 0DB TONE 1
NOP
NOP
NOP
NOP
OUT (6),A
NOP
IN A,(3)
pop bc
pop AF
RET
```

The remaining date is poked into the ROM by an external program using addresses that were found by trial and error. This was due to limitations of the assembler, a later version would have allowed for binary data block to be inserted.

```
ORG &2280
.Screendata
; compressed screenimage placed here by the builder

ORG &3180
.colourdata

ORG &3FC0
.directory

;list of starting sector for each game
;&FF should be present in the first unused entry
```

The rom code itself has no indication of what it's loading, the screen image has the game titles "drawn" in at build time.

The early versions of the code use a text mode based interface, and that version did keep a directory of the game names. This screenshot of one of the very early development versions shows the pitfalls of not



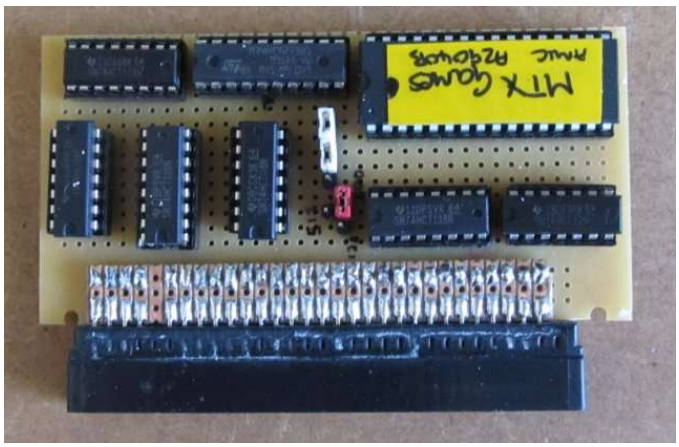
setting the VDP pointers correctly. Miss-setting bit 14 of the address resulted in an off by one error in both the character definitions and some of the text positioning.

The move to graphics mode 2 resulted in a much more “professional” display, as the screenshot of the last of the development roms shows. The background image was dumped from the emulator on my RiscPC and then loaded into a BBC basic program to insert the windows that Dave dubbed “Aero for the MTX”. Care had to be taken with the positioning to ensure that the resulting image could be rendered by the VDP. Conversion to VDP format and compression were also done with the RiscPC for inclusion in the rom code above.

9.1.2 The Magrom Hardware

There were 5 different versions of the Magrom design, 3 of which were built as prototypes, before the V1.0 board was finalised. Broadly they fit into 2 basic types, the early versions used logic gates to set address lines low when the rom was accessed through the #2000 in paged rom space.

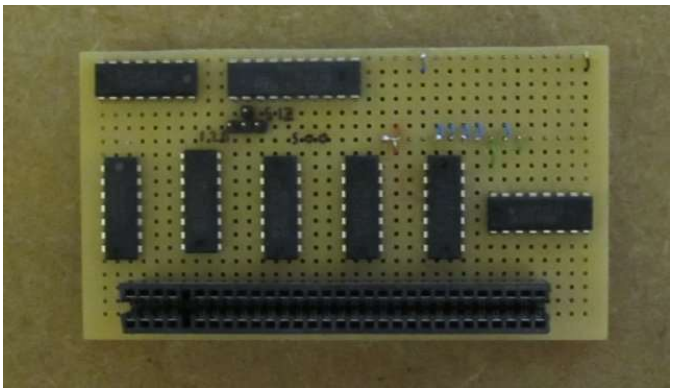
Following the discussion of the prototype on the Memorum forum, the last versions used pull down resistors.



Version 1 never got past the design stage. Version 2 was the first one to be built. As the photograph shows the board was built to fit horizontally on the MTX external connector. The edge connector was a 31 way vertical fit part intended for PC ISA slots and needed adaption to fit the board.

At this point there was still a possibility of using a 1024k EPROM instead of Flash so the white jumpers are included to allow for the larger device.

This prototype highlighted a design issue with paging that required a re-design



Prototype 2 also investigated the possibility of using a vertical fit. On proto board this caused issues, the "natural" position of the chips was on the inside, hover than left insufficient clearance for the ROM to be socketed, so that was fitted on the reverse. This version worked completely as intended, at the cost of requiring 8 logic devices in addition to the ROM. The EPROM option was removed

This version used a 74HCT138 decoder to detect the 3 possible positions for the data portion of the ROM, which fed to the T jumper that selected one of the 3 to be used as the data ROM chip select. A second 74HCT138 decoded the chip select for the program portion of the ROM, all 8 possible ROM slot were decoded. the ROM7 output was used, but any of the available ROM slots could have been used.

There were two 74HCT08 AND devices as there were 5 address lines that had to switch between zero in program mode, and the selected page in data mode. One of the 3 spare gates was used to combine the 2 different mode selects into one chip select with the final 2 gates unused with the inputs tied to 0v to avoid leaving them floating.

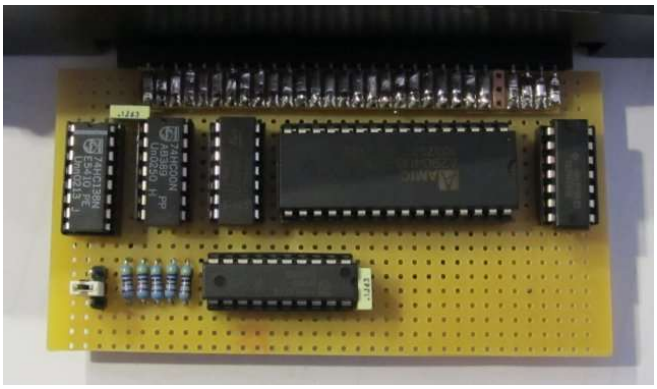
The paging data was held in a 74HCT273 exactly the same way that the MTX motherboard stores the main ROM/RAM paging register. All 8 inputs were connected to the data bus but only 5 of the 8 possible outputs were used, with the others left unconnected.

Because the Z80 has separate IO and Memory address spaces, a 74HCT32 OR was used to combine the I/O request output and write pins into a single I/O Write used for setting the paging register, and also memory request and read for reading he ROM. Again the 2 unused gates had their inputs tied to 0v.

The final section of the design used a 74HCT20 NAND to decode all bits high on A0 to A3 and A4 to A7. Those 2 signals along with the I/O Write went to a 3rd 74HCT138 to decode write for IO port #FF roe setting the page port.

The vertical fit proved to be impractical, with issues locating the rom select jumper and the front corner of the board being a hazard to the user's left hand.

Version 4 of the board was the final "active" design which replaced the dual 4 input NAND with an 8 input NAND which then eliminated the 3rd 74HCT138, as once of the spare OR gates could then be used to create the page port select. That design was never built, as the discussion on Memorum had identified further refinements.



The version 5 board used the GROM output from the edge connector instead of doing its own ROM decode, saving another 74HCT138. The page port was changed to a 74HCT374 which has 3 state outputs, these are only active in data mode. In ROM 7 mode the 5 resistors pull the address lines low. Making the all but one of the AND gates redundant. The data ROM page selection is still done with the 74HCT138. The 74HCT32 OR remains to combine chip selects and the Z80 signals. 2 Gates of a 74HCT00 NAND is used in place of the last AND gate, while the 3rd gate enables the page port output, the 4th gate is spare so had its inputs wired to 0v.

Using Port #FF could interfere with some of the other MTX hardware addons, most notably the Speculator. So, the board was modified to use the final NAND gate as another inverter so as to use port #FB instead, and it was this modified Version 5.1 that went on to become the V1.0 production board, seen here being assembled.

The production board has additional refinements. Provision is made for edge connectors on both sides, for internal or external fit. The positioning of the chips at the top of the board was required to ensure there was adequate clearance under the sloping MTX keyboard.

The right-angle edge connectors used raise the board sufficiently far above the main PCB that there is only a millimetre or so clearance between the Magrom and the keyboard PCB at the low point.

The final production board only bears a passing resemblance to the original, as the design process resulted in modifications for:

- Compatibility with other expansion
- Physical fit
- Component reduction

9.2 CFX

After the Magrom reached ”release” status I joined the MTXplus+ project. Towards the end of 2014 we began work on the I/O board for them MTXplus+ and I started writing the code to support the IDE interface. Early versions of the MTXplus+ ran BASIC and used a patched version of the MTX ROM where the tape loading code was replaced by some custom code to read FAT32 CF cards. Extending the read only system to read and write FAT32 was abandoned. Manipulating potentially 32k long clusters in 64k of memory proved to be impractical.

Andy Key had done a LOT of work to patch the FDX and SDX firmware to run on the REMEMOorizer and ReMemotech, and made the full source code available. Having studied the code there were just 3 routines in each system that dealt with the disc system:

Initialise
Read one sector
Write one sector

Replacing those with the equivalents I already had in the FAT32 code was straightforward and the MTXplus+ then had a fully working storage system. The MTXplus+ project then reverted to using the original CPM disc format as used by the FDX and SDX. Whist that made transferring software to the MTXplus+ a little more complicated, it did mean both CPM and the SDX extensions to Basic were available, and there was no requirement to patch into the tape code.

At that point I realised that the same hardware could be fitted to a standard MTX and so CFX was born.

9.2.1 CFX IDE Software

The original FDX systems ran CPM and used a boot ROM in paged ROM slot 4 to control the system. The SDX basic extensions used ROM slot 5 which enabled a single 16k EPROM to handle both. 16K isn’t a commonly available size for “modern” devices 32k EEPROMs are available, but 128k Flash is both cheaper, faster and available from more sources. For this reason, the CFX is actually fitted with 128k of rom, however only 16k is used to provide the 2 rom images for slots 4 and 5.

The CF support software is the same for both ROMs, for the FDX it actually gets built to run in RAM, as CPM doesn’t (normally) use the boot rom once the start-up sequence is completed.

The SDX on the other hand runs the disc support software in ROM. The different locations for the code, buffers etc requires different source files, but the essentials are the same. This is the one from the SDX side

```
IDE8255_LSB EQU &6c             ;108  Port A
IDE8255_MSB EQU &6d             ;109  Port B
IDE8255_CTL EQU &6e             ;110  Port C
IDE8255_CFG EQU &6f             ;111  direction control
```

The IDE interface is handled through a 82C55 PPI which needs just 4 I/O locations in the MTX memory map to fit in the 16 ports of the IDE interface itself.

```
;control port settings
IDE8255_READ  EQU %10010010     ;Port C is output A and B
                                ;inputs,using mode 0
IDE8255_WRITE EQU %10000000     ;all 3 ports are outputs
```

The PPI needs 2 control setups, 16 bits of input when reading and 16 bits of output when writing, the IDE control interface is always write only.

```
;IDE address lines A0-A2 direct to Interface
;IDE Control lines are inverted, so need to be set to trigger an
; active LOW pulse
;B3 = CS0, select main registers
;B4 = CS1, select aux registers
;B5 = IDE Write line
;B6 = IDE Read line
;B7 = IDE Reset line
IDE_WR_LINE EQU %00100000
IDE_RD_LINE EQU %01000000
IDE_RESET   EQU %10000000

;IDE Command registers
IDE_DATA         EQU %00001000  ;CS0 register 0 16 bit wide data
IDE_DATA_rlow    EQU %01001000  ;CS0 register 0 16 bit wide
                                ;data read line low
IDE_DATA_wlow    EQU %00101000  ;CS0 register 0 16 bit wide
                                ;data write line low
IDE_ERR          EQU %00001001  ;CS0 register 1
IDE_COUNT        EQU %00001010  ;CS0 register 2 sector count
                                ;always use 1 to transfer to
                                ;the buffer
IDE_LBAlow       EQU %00001011  ;CS0 register 3 low 8 bits of LBA
                                ;address
IDE_LBAmid       EQU %00001100  ;CS0 register 4 mid 8 bits of LBA
                                ;address
IDE_LBAhigh      EQU %00001101  ;CS0 register 5 upper 8 bits of
                                ;LBA address
IDE_LBAtop       EQU %00001110  ;CS0 register 6 top 4 bits of LBA
```

```
                                    ;address
IDE_COMMAND      EQU %00001111      ;CS0 register 7 write command
IDE_STATUS       EQU %00001111      ;CS0 register 7 read status
IDE_CONTROL      EQU %00010110      ;CS1 register 6
IDE_ASTATUS      EQU %00010111      ;CS1 register 7 read other status
                                    ;register
```

The hardware dedicates 5 bits to register selection, and 3 bits to handle read, write and reset. The logic for those 3 pins is inverted in the hardware so a "1" sets the signal low (or active) and a zero sets it high. The register definitions for setting the data direction register have extra equates that also include the read and write line, that was done as part of the optimisation to increase the interface transfer rate.

```
; IDE Status Register:
; B7 = Busy      1=busy, 0=not busy
; B6 = Ready     1=ready for command, 0=not ready yet
; B5 = DF        1=fault occured inside drive
; B4 = DSC       1=seek complete
; B3 = DRQ       1=data request ready, 0=not ready to xfer yet
; B2 = CORR      1=correctable error occured
; B1 = IDX       vendor specific
; B0 = ERR       1=error occured

;the actual IDE commands used
CMD_RECAL   EQU &10
CMD_READ    EQU &20
CMD_WRITE   EQU &30
CMD_INIT    EQU &91
CMD_ID      EQU &EC
CMD_SPINUP  EQU &E0
CMD_SPINDN  EQU &E1
```

CFX only uses a small subset of the available commands, in the end it didn't even need all of the ones listed. CF obviously doesn't need to spin up or spin down. And it turned out running a "real" IDE drive on the interface didn't require them either.

The SDX system actually loads in the CPM BDOS off of the drive, so requires equates for the CPM functions used. Andy's re-built source code includes a file for that.

```
; BDOS.INC  Basic BDOS function numbers etc.
;
;  BDOS    EQU     &05
;  TRUE    EQU     &FFFF
;  FALSE   EQU     &0000
;  K       EQU     1024
;  CONIN   EQU     1
;  CONOUT  EQU     2
;  RDRIN   EQU     3
;  PUNOUT  EQU     4
;  LSTOUT  EQU     5
;  DCONIO  EQU     6
```

```
;  GETIOB  EQU     7
;  SETIOB  EQU     8
;  PRINTF  EQU     9
;  RCBUFF  EQU     10
;  CONST   EQU     11
;  RETVER  EQU     12
   RESDSC  EQU     13
   SELDSC  EQU     14
   OPENF   EQU     15
   CLOSEF  EQU     16
   SFF     EQU     17
   SFN     EQU     18
   DELFIL  EQU     19
;  RSEQ    EQU     20
;  WSEQ    EQU     21
   MAKFIL  EQU     22
   RENFIL  EQU     23
;  RETLV   EQU     24
   RCDSC   EQU     25
   SETDMA  EQU     26
   GETAA   EQU     27
;  WPDSC   EQU     28
;  GETROV  EQU     29
   SFATTR  EQU     30
   GETDPB  EQU     31
;  GSUSR   EQU     32
   RRAN    EQU     33
   WRAN    EQU     34
   CFSIZE  EQU     35
;  SETRR   EQU     36
;  RESDRV  EQU     37
;  WRANF   EQU     40
;  FCB1    EQU     &5C
;  FCB2    EQU     &6C
;  R0      EQU     33
;  R1      EQU     34
;  R2      EQU     35
```

None of the BIOS calls (functions 1 to 12) are used as BASIC already has its own routines for screen and keyboard handling. Memory usage is also different so the file control blocks aren't in low memory like they would be running CPM.

```
; SDX ROM use of high memory
;

LSTLOW  EQU    &D3FF
BDOSADDR EQU   &D700
LSTHIGH EQU    &D700
JP59K   EQU    &D706
CPMLOC  EQU    &D708
DSKMSG  EQU    &D7BA
```

```
DSKERR  EQU    &D7C6
ERRFLG  EQU    &D7E5
PRINT0  EQU    &D804
CRLF0   EQU    &D80E
CHNL1   EQU    &D840
CHNL2   EQU    &D86A
CHNL3   EQU    &D894
CHNL4   EQU    &D8BE
CHNL5   EQU    &D8E8
LINPUT  EQU    &D8F4
IXTEMP  EQU    &D8F6
STORE   EQU    &D8F8
USERSAV EQU    &D912
CURDSK  EQU    &DA42

BIOSADDR EQU   &E500
DPH_A   EQU    &E533
DPH_B   EQU    &E543
DPH_C   EQU    &E553
DPH_D   EQU    &E563
DPH_E   EQU    &E573
DPH_F   EQU    &E583
DPH_G   EQU    &E593
DPH_H   EQU    &E5A3
DPH_I   EQU    &E5B3
DMA_DEF EQU    &E680
DISCASS EQU    &E880
DIRBUF  EQU    &E8C0
VECST   EQU    &E940
AVEC_AB EQU    &E940
CVEC_AB EQU    &E954
AVEC_C  EQU    &E964
CVEC_C  EQU    &E978
VECEND  EQU    &E9C0

DBUF    EQU    &E9C0
TDBUF   EQU    &EAC0
PTRKP   EQU    &EB80
LCA     EQU    &EB82
EFLAG   EQU    &EB83
LSTOUTX EQU    &EB84
SWUF    EQU    &EB85
CURDRV  EQU    &EB86
TRACKS  EQU    &EB87
BFID    EQU    &EB8B
CFGTAB  EQU    &EB93
TRUST   EQU    &EB9B
DRVRQ   EQU    &EB9C
CFGBYT  EQU    &EB9D
TRKRQ   EQU    &EB9E
SECRQ   EQU    &EBA0
DMARQ   EQU    &EBA2
```

```
CDDRV   EQU     &EBA4
BPNT    EQU     &EBA5
RETRY   EQU     &EBA6
TOAM    EQU     &EBA7
JPLINK  EQU     &EBA9
ULINK1  EQU     &EBAC
SKEW    EQU     &EBAF
```

Because CPM is basically a pre-built unit the various routines and storage locations are fixed and additional equates for these are also included.

```
DBUF2    EQU    &ED00      ;256 bytes allocs
TDBUF2   EQU    &EF00      ;128 byte checks
DBUF3    EQU    &F000      ;256 bytes allocs
TDBUF3   EQU    &EF80      ;128 byte checks
DBUF4    EQU    &F100      ;256 bytes allocs
TDBUF4   EQU    &F200      ;128 byte checks
```

The original SDX rom allowed access to a single drive. 3 additional drive buffers are defined here to allow for 4 drives or partitions.

```
; F280 to F2FF unused by SDX rom
WORKSPACE     EQU &F280    ;string workspace 32 bytes
math_a        EQU &F2A0    ;2x32 bit working registers for
                           ;the math calculator
math_b        EQU &F2A4
```

FAT32 support needed buffers for 32 math plus filenames etc these locations were retained for use by the support/start-up portion of the ROM.

```
; The SD Card support is incompatible with the NODE ROM,
; because we overwrite part of its memory space.
SECADR  EQU     &F300      ;word offset
CBLKBN  EQU     &F302      ;SECADR+2       4 byte block number
CBLKBF  EQU     &F306      ;CBLKBN+4       512 byte buffer
SDREGV  EQU     &F506      ;CBLKBF+&0200   16 byte area
SDLBA   EQU     &F516      ;SDREGV+16      4 byte LBA

SDXBDOS EQU    &F5B0
USRJMP  EQU    &F5B3
DSCFLG  EQU    &F5D3
KEYJP   EQU    &F5D4

Var_base      EQU &F8F2    ;standard saved base of system
variables
```

The CF deals with 512 byte sectors, CPM deal with 128 byte sectors so a larger sector buffer is required and because of the ROMs in the memory map cannot be located at CPM's usual #0080. For the same

reason the default BDOS/BIOS entry using CALL #0005 won't work and so an alternate entry point is provided.

The CFX SDX rom is based on the original SDX code, however the start-up sequence is considerably different.

The SDX boot ROM seems to have been designed to cause the minimum of interference with tape based software and to prove the best start-up speed, so no setup is done until the first disc access.

CFX on the other hand has the MUCH more responsive CF for storage and doesn't have to take "hot swapping" into account so does initialise the drive on start-up. Since there is more than enough room in the ROM it also displays the boot screen.

;

```
;TITLE   SDXMAIN SDX main
; From Andy Key's SDX code:
; This is a reconstruction of the source for the SDX ROM.
; A SDX ROM was disassembled using BE and BEZ80.
; Symbols were taken from a PDF of a printed listing of the ROM.
; This was post-processed using a custom script, to make it
; more M80-like.
; Finally, salient text in the PDF was transcribed to this
; source.

; MTXplus+ CF init and load/save code replaces Andy's SD card
; routines

; signature
ORG &C000
Offset &2000
DB      8,7,6,5,4,3,2,1

DW      ROMCODE
DW      0
;
; ROM command calls here (=0200CH)
;
.ROM 5
JP      CPMLDR
db  0
;
; power-up code calls here (=02010H)
;
.POC
LD      A,&7F
OUT     (5),A
IN      A,(5)
AND     &8
RET     Z                   ;if "M" pressed, return to BASIC
IN      A,(6)
AND     1
```

```
ret     z                   ;if "space" pressed, return to BASIC
call signon
call initdsc
call wfs
RET

.CPMLDR
CALL    INITDSC
jr nz   no_cf
CALL    u_loader  ;only call the loader if the CF is present
JP      RETBASIC
.no_cf
jp basic2
```

The code starts with a standard MTX Autoboot signature, the “ROM 5” entry point is retained as that provides an alternative to CPM’s CTRL C to reset CPM in the event of an error etc.

The power up code then checks for 2 possible keypresses, “M” and space and will return immediately id either is pressed. “M mode” is provided to allow the MTX to bypass the CF system if the intention is to use the MTX in “tape mode”. If a Magrom were also fitted, that needs space pressed in order to start it. However, Magrom is in paged ROM 7 which is the last paged ROM to be accessed.

Loading the BASIC extensions in itself doesn’t affect Magrom, but the time taken is wasted, since nothing on Magrom knows anything about the disc sub system (or even Basic!).

Having checked the keys, the first half of the boot screen is displayed, followed by the drive setup code that prints the CF status. That’s followed by the code that prints the 2nd half of the boot screen and then waits for the user to press return. On the MTXplus+ where this code originated, the system waits for the space bar, but that was changed for compatibility with Magrom.

The CPM loader as called on “ROM 5” is extended to only try to load the system tracks if a CF is found, there isn’t one, it exits without an extra error message straight into the “Ready” prompt.

The rest of the code in SDXMAIN is as per Andy’s re-build of the original source, with the addition of drive parameter blocks to support the extra 3 partitions.

The CPM to SD drive code in Andy’s SDXSD file is unchanged, with the exception that the calls to SD hardware handlers are replaced with calls to CF hardware handlers.

The CF low level driver provides the 3 driver entry points expected by CPM to SD translation routines plus all the PPI access code.

```
; CF specific low level routines

;
; Initialise CF card
; after:
;   if ok, Z, CF initialised
;   if not ok, NZ, CF not initialised
;
.CFInit
```

```
;do a hard reset on the drive, by pulsing its reset pin.
PUSH BC
push DE
LD A,IDE8255_READ          ;set the 8255 to read mode
OUT (IDE8255_CFG),A
LD A,IDE_RESET
OUT (IDE8255_CTL),A        ;Pull the reset pin LOW
LD B,8                     ;needs to stay low for 25usec
                           ;according to the 1992 spec
.RESET_LOOP                ;100 cycles at 4mhz
DJNZ RESET_LOOP
XOR A
OUT (IDE8255_CTL),         ;Pull the reset pin back high
LD C,IDE_LBAtop
LD E,%11100000             ;select the master device. LBA mode
call IDE_WR_8
;
;LD B,0                    ;for the timeout counter
LD BC,IDE_STATUS           ;implied LD B,0 in addition to setting
                           the IDE register
jr IDE_init_pass1          ;jump past the timeout checks on the
first pass
.IDE_INIT_loop
dec B                      ;exit if we've tried 256 times and
                           ;drive still not ready
JR Z m_timeout
call CF_delay              ;CF doesn't need to spin up, but
                           ;allow some extra time
                           ;just in case. The 255 calls
                           ;of the delay routine take
                           ;approx 2.5 sec at 4mhz
.IDE_INIT_pass1
CALL IDE_RD_8              ;read status register to E
RL E                       ;bit 7 to carry
JR C IDE_INIT_LOOP         ;wait for BSY to be clear
RL E                       ;bit 6 to carry
JR NC IDE_INIT_LOOP        ;wait for RDY to be set
POP DE
POP BC
```

The init code doesn't actually need to save registers, but some of the later code using the time-out does save registers, so to maintain stack integrity every routine that could time out saves DE and BC.

```
rst &10
db &93
db 13,10
ds " CF initialised"
db 13,10
XOR A                ; A zero'd and ZF set to indicate all OK
RET
```

If the drive initialises this is reported on screen. This is used for both the boot up code and “ROM 5” software reset to confirm success.

```
.m_timeout
rst &10
db &94
db 13,10
ds " Drive not found"
db 13,10
.timeout
POP DE
POP BC
xor A
DEC A                           ;return A no zero and Z reset
RET
```

If the drive fails to respond within time at any stage a message is printed. No BASIC error message isn’t generated, it’s left to the calling code to decide what to do with the error. In the CPM version of the driver code there are no status messages. The RST 10 interface being BASIC only.

```
;waste approx 3339 * 12 is approx 40,000 cycles delay 0.01 sec at
4mhz.
.CF_delay
push bc
ld BC,12
.CF_delay_loop
DJNZ CF_delay_loop    ;3323 cycles on inner loop
dec c                    ;4
JR nz CF_delay_loop   ;12
pop bc
ret
```

The code uses a generic 1/100 sec delay routine, that doesn’t affect any registers. From basic, interrupts could have been used to deal with detection a timeout. However, the CTC isn’t setup in CPM and really it’s not worth the extra work, it’s simpler to just waste cycles.

```
;
; Read block from CF card
;   before:
;     SDLBA is 32 bit block 512 byte block number, HL is buffer
;   after:
;     if ok, Z
;     if not ok, NZ
;
.CFREAD
PUSH BC
PUSH DE
CALL IDE_WAIT_NOT_BUSY
CALL    CFSetLBA       ;send LBA number to the drive
```

```
LD E,CMD_READ
LD C,IDE_COMMAND
CALL IDE_WR_8
CALL IDE_WAIT_DRQ
AND 1                ;isolate bit 0
JR NZ,GET_ERR        ;error bit set, find out why
CALL CFREAD_DATA
POP DE
POP BC
xor A   ;exit with A zero'd and Z set
RET
```

CFread is the entry point for reading one sector, it waits for any running command to finish, sets up the 24 bit LBA sector number, and then issues the read command. Once the device reports ready the actual read is done. For readability reasons this is a separate routine, but performance would be marginally improved is the code was included directly.

```
.CFRead_Data
PUSH    BC
PUSH    DE
PUSH    HL
; ld B,0              ;256 2 byte transfers needed implied
ld BC,IDE8255_CTL     ;preset C to PIO port C
LD e,IDE_DATA         ;all access through the 16 bit wide data
                      ;register
ld d,IDE_DATA_RLOW
.READ_DATA_LOOP
OUT (C),e             ;(12) set the register & chip select bits
                      ;drive with read high
OUT (C),d             ;(12) drive the read line low
IN A,(IDE8255_LSB)    ;(11)
LD (HL),A             ;( 7)
INC HL                ;( 6)
IN A,(IDE8255_MSB)    ;(11)
LD (HL),A             ;( 7)
INC HL                ;( 6)
DJNZ READ_DATA_LOOP   ;(13)
xor A
out (C),A             ;deselect everything also drives
                      ;read high a final time
POP     HL
POP     DE
POP     BC
RET
```

The low level code is optimised to minimise the transfer time. Because the IDE interface is using 16 bit wide date, 2 bytes are read on each pass through the inner loop, so 256 passes are required to read the whole sector. Registers C D and E are pre-set to save cycles in the inner loop. OUT (C),reg is a cycle slower than OUT (port),A but not having to set A saves more than that. As the cycle timings indicate it takes 85 cycles to complete the inner loop, meaning a 512 byte sector is loaded in just under 22,000 cycles giving a peak transfer rate of 180k a sec. CPM and BASIC overheads mean the overall transfer rate is a LOT lower. D and E are set separately for code clarity at the cost of a few cycles.

```
; CF routines only require A=0/Z=1 good or A<>0/Z=0 error
; however error read on return for future expansion
.GET_ERR
LD C,IDE_ERR
CALL IDE_RD_8
POP DE
POP BC
AND A
RET NZ
DEC A   ;make sure A isn't zero and the flag isn't set
RET
```

The error register is checked on failure, none of the current code uses it as the driver is only expected to provide a yes/no response. But the error data is there for any future use. As with timeout the stack needs to be tidied on exit.

```
;
; Write block to CF card
;   before:
;     SDLBA is 32 bit block 512 byte block number, HL is buffer
;   after:
;     if ok, Z
;     if not ok, NZ

.CFwrite
PUSH BC
PUSH DE
CALL IDE_WAIT_NOT_BUSY
CALL CFSetLBA
LD E,CMD_WRITE
LD C,IDE_COMMAND
CALL IDE_WR_8
CALL IDE_WAIT_DRQ
AND 1
JR NZ,GET_ERR
CALL CFWRITE_DATA
CALL IDE_WAIT_NOT_BUSY
AND 1
JR NZ,GET_ERR
POP DE
POP BC
xor A
RET
```

Writing to the CF is a done in exactly the same way as the read, except the data travels in the other direction.

```
;
.CFWrite_data
PUSH    BC
PUSH    DE
PUSH    HL
```

```
;ld B,0                 ;256 2 byte transfers needed implied
ld BC,IDE8255_CTL       ;preset C to PIO port C
LD e,IDE_DATA           ;all access through the 16 bit wide data
                        ;register
ld d,IDE_DATA_WLOW
LD A,IDE8255_WRITE      ;set the 8255 to output mode
OUT (IDE8255_CFG),A
.WRITE_DATA_LOOP
OUT (C),E               ;set the register & chip select bits,
                        ;Write is high
LD A,(HL)
INC HL
OUT (IDE8255_LSB),A     ;put the low 8 bits onto the bus
LD A,(HL)
INC HL
OUT (IDE8255_MSB),A     ;put the high 8 bits onto the bus
OUT (C),D               ;drive the write line low
DJNZ WRITE_DATA_LOOP
xor A
out (C),A               ;deselect everything also drives write high
LD A,IDE8255_READ
OUT (IDE8255_CFG),A     ;set the 8255 back to read mode
POP     HL
POP     DE
POP     BC
xor A
RET
```

Because the PPI code defaults to setting the IDE interface to reading, writing is a tiny bit slower reducing peak transfer rate by a fraction. in practice however the overheads from the need to deal with the CPM code writing 128 byte sectors, means writing to the drive from CPM or BASIC will take at least 4 times as long as reading.

```
;
; Set CF LBA
;
.CFSetLBA
push hl
push de
push bc
push af
LD HL,(SDLBA)
LD A,(SDLBA+2)
LD E,&e0                ;Top 4 bits of address zero, &E is for LBA
master
LD C,IDE_LBAtop
CALL IDE_WR_8           ;write LBA mode to LBA top byte register
ld E,A
ld c,IDE_LBAhigh
CALL IDE_WR_8           ;write to LBA bits 16-23
```

```
ld E,H
ld c,IDE_LBAmid
CALL IDE_WR_8        ;write to LBA bits 8-15
ld E,L
ld c,IDE_LBAlow
CALL IDE_WR_8        ;write to LBA bits&-7
LD E,1
LD C,IDE_COUNT
CALL IDE_WR_8        ;set the sector count to 1
pop af
pop bc
pop de
pop hl
RET
```

Setting the LBA also sets the drive to master, and the sector count to 1, This is done every time. Just in case something has updated the registers between calls. Only 24 bits, of the possible 28 bits or sector address are used, so only 1/16 of the maximum address range is available. In practice this isn't an issue The MTX implementation of CPM uses a maximum partition size of 8 megabytes and 8 partitions. 64 megabytes of storage would need 131072 of the CF's 512 byte sectors, which only requires a 17 bit address.

```
.IDE_WAIT_NOT_BUSY
PUSH BC
PUSH DE
.IDE_WAIT_NOT_B_LOOP
LD C,IDE_STATUS
call IDE_RD_8            ;read status register to E
LD A,E                   ;return status in A
RL E                     ;bit 7 to carry
JR C IDE_WAIT_NOT_B_LOOP ;wait for BSY to be clear
pop DE
pop BC
RET
```

3 different status checks are performed, drive busy, drive ready, and data ready. The released versions of the CFX code don't actually have any timeout code in these routines, as there were no failures of this type in testing. Not having the code speeds up the access and shortens the code.

```
.IDE_WAIT_READY
PUSH BC
PUSH DE
.IDE_WAIT_R_LOOP
LD C,IDE_STATUS
CALL IDE_RD_8          ;read status register to E
LD A,E                 ;return status in A
RL E                   ;bit 7 to carry
JR C IDE_WAIT_R_LOOP   ;wait for BSY to be clear
RL E                   ;bit 6 to carry
JR NC IDE_WAIT_R_LOOP  ;wait for RDY to be set
```

```
POP DE
POP BC
RET
```

Wait for data and wait for ready both check the busy flag first, as the other flags aren't guaranteed to be accurate if the device is busy.

```
        ;Wait for the drive to be ready to transfer data.
        ;Returns the drive's status in A
.IDE_WAIT_DRQ
PUSH BC
PUSH DE
.IDE_WAIT_D_LOOP
LD C,IDE_STATUS
CALL IDE_RD_8           ;read status register to E
LD A,E                  ;return status in A
RL E                    ;bit 7 to carry
JR C IDE_WAIT_D_LOOP    ;wait for BSY to be clear
RL E                    ;bit 6 to carry
RL E                    ;bit 5 to carry
RL E                    ;bit 4 to carry
RL E                    ;bit 3 to carry
JR NC IDE_WAIT_D_LOOP   ;wait for DRQ to be set
POP DE
POP BC
RET
```

The final 2 routines read and write the IDE registers, there are 8 bit transfers in the low byte, the upper byte is meaningless. Being an 8 bit transfer, it's much simpler than the data transfer.

```
;8 bit read of IDE register
;Data in E, (D unused in 8 bit transfer)
;register number in C
;8255 in input mode by default

.IDE_RD_8
PUSH AF
LD A,C
OUT (IDE8255_CTL),A        ;set the register & chip select bits
OR IDE_RD_LINE
OUT (IDE8255_CTL),A        ;drive the read line low
IN A,(IDE8255_LSB)
LD E,A
LD A,C
OUT (IDE8255_CTL),A        ;drive the read line high again,
                           ;completing the write cycle
xor A
out (IDE8255_CTL),A        ;deselect everything
POP AF
RET
```

```
;8 bit write to IDE register
;Data in E, (D unused in 8 bit transfer)
;register number in C

.IDE_WR_8
PUSH AF
LD A,IDE8255_WRITE          ;set the 8255 to output mode
OUT (IDE8255_CFG),A
LD A,E
OUT (IDE8255_LSB),A         ;put the low 8 bits onto the bus
LD A,C
OUT (IDE8255_CTL),A         ;set the register & chip select bits
OR IDE_WR_LINE
OUT (IDE8255_CTL),A         ;drive the write line low
LD A,C
OUT (IDE8255_CTL),A         ;drive the write line high again,
                            ;completing the write cycle
xor A
out (IDE8255_CTL),A         ;deselect everything
LD A,IDE8255_READ
OUT (IDE8255_CFG),A         ;set the 8255 back to read mode
POP AF
RET
```

As with the data transfer, register writing is factionally slower the reading because of the default to reading the PPI

9.2.2 CFX Support Software

In addition to providing the extensions to BASIC via the USER command in the same way that the original SDX did, CFX also has a start-up screen. Unlike Magrom that screen uses text mode, however there are enough unused character definitions that I was able to include a small logo.

```
;TITLE CFX Extras
;Code specific to the CFX Expansion
;let the world know the expansion is working
.Signon
RST  &28       ; Utility routine
DB   &42       ; call VDINIT   (&2E85)
```

As with Magrom, the rom is called before the screen is set up so the utility routine that sets things up has to be called early

```
XOR  A
LD   (SCRN5+3),A
LD   A,&28
LD   (SCRN5+5),A
```

The screen has to be set to the full 40 column width, BASIC usually only uses 39 because of the positioning issues of the TMS9929. If these values are left unchanged there's a possibility of garbage not being cleared from the screen on reset.

```
RST  &10       ; output routine
DB   &4D      ; select VS5 and clear
```

```
call killsnd  ;rom routine to turn off all sound channels
```

Again, as with Magrom the sound chip needs muting.

```
RST &10
DB &84
DB 6,15
DB 4,4
```

The RST 10 call sets the screen colours to the darker blue background with white text. The logo is then printed before the rest of the screen display is generated.

```
call logo
RST &10
DB &92
DB 3,1,0
DS "Memotech MTX"
DB 3,1,2
call print_mem_size
RST &10
db &84
DS " RAM"
RST &10
DB &9C
DB 3,1,4
DS "Compact Flash File System"
RST &10
db &8e          ;3 for the cursor positioning + 11 byte date
                ;string inserted = 14 &0E
DB 3,29,4
DATE
RST &10
db &8e          ;3 for the cursor positioning 6 for the text + 5
                ;for the string input by BUILD = 14 &0E
DB 3,29,3
DS "Build "
BUILD
call beep
RST &10
DB &9C
DB 3,1,6
DS "Searching for boot drive:"
ret
```

At this point control passes back to the boot code where the CF is checked and set up

WFS which used to be “Wait for Space” is called after the CF status message is printed, and build the rest of the screen.

```
.wfs
push af
RST &10
DB &B6
DB 3,1,10
```

```
DS "Other boot options:"

DB &B9
DB 3,1,12
DS "Reset C     - CPM mode"

DB &B9
DB 3,1,13
DS "Reset M     - MTX mode"

DB &B0
DB 3,1,14
DS "Reset Space -"
DB &B7
DS " start Magrom if fitted"

DB &AF
DB 3,6,18
DS "Press <RET> "
DB &B2
DS "to enter SDX BASIC"

DB &AD
DB 3,6,20
DS "Press <I> "
DB &94
DS "for more information"

.wfs_loop
ld a,&FB
OUT (5),A
IN A,(5)
AND &10
call Z info_screen

LD A,&DF
OUT (5),A
IN A,(5)
AND &40
jr nZ wfs_loop
pop af
ret
```

The wait loop repeatedly tests the keyboard hardware directly for “I” and return. If “I” is pressed the info screen displays, return from either the info screen, or the boot screen will pass control back to the boot code, and then the rest of the BASIC start-up sequence

```
.info_screen
ld HL,screen_data
ld a,12
.screen_loop
```

```
RST &28
DB &AC
ld a,(hl)
INC HL
and &7f
jr nz screen_loop
ret
```

The info screen is just pushed to the VDU drive 1 byte at a time until the whole page is displayed. It exits when the zero byte at the end is found.

```
.screen_data
DS "SDX Basic additional commands:"
DB 13,10
DS "USER DIR"
DB 13,10
DS "USER LOAD ¢FILENAME.EXT¢"
DB 13,10
DS "USER SAVE ¢FILENAME.EXT¢"
DB 13,10
DS "USER READ ¢FILENAME.EXT¢,start"
DB 13,10
DS "USER WRITE ¢FILENAME.EXT¢,start,length "
DB 13,10
DS "USER MTX ¢FILENAME.MTX¢"
DB 13,10
DS "USER RUN ¢FILENAME.RUN¢"
DB 13,10
DS "USER ERA ¢FILENAME.EXT¢"
DB 13,10
DS "USER REN ¢NEWNAME.EXT¢=¢OLDNAME.EXT¢"
DB 13,10
DS "USER COPY ¢NEWFILE.EXT¢=¢OLDFILE.EXT¢"
DB 13,10
DS "USER STAT ¢<drive>:¢ or ¢FILENAME.EXT¢"
DB 13,10
DS "USER OPEN#channel,¢FILENAME.EXT¢,type"
DB 13,10
DS "USER CLOSE#channel"
DB 13,10
DS "USER KILL#channel"
DB 13,10
DS "USER PRINT#channel,variable list"
DB 13,10
DS "USER INPUT#channel,variable list"
DB 13,10
DS "USER LINE INPUT#channel,variable"
DB 13,10
DS "USER REC#channel,record number"
DB 13,10
DS "USER EOF#channel,line number"
DB 13,10
DS "USER TYPE ¢FILENAME.EXT¢"
```

```
DB 13,10
DS "USER QUIT  (perform a NEW)"
DB 13,10
DS "ROM 5      (reset the CF)"
DB 13,10
DS "Press <RET> to continue into SDX BASIC"
DB 0
```

The 32 bit math routine used to calculate and print the memory size do the job, but are horribly inefficient and wasteful. They're a remnant of the original MTXplus+ FAT32 code that really should have been replaced with a short look up table. There are after all only 16 possible memory sizes.

```
.print_mem_size
push hl
push af
ld a,(lstpg)        ;number of RAM pages found on startup
inc a               ;add 1 for the fixed page
ld (math_b),a       ;extend to 32 bit number in math register
xor a
ld (math_b+1),a
ld (math_b+2),a
ld (math_b+3),a
ld a,5
ld hl,math_b
call shift_32       ;multiply by 32, as 32k per page
call bin_to_dec
call print_decimal
LD a,+ASC"K"
RST &28
DB &AC
pop af
pop hl
RET

.print_decimal
push hl
push bc
push af
ld hl,workspace+-1
;print the number in the workspace, skipping leading zero's
;HL is pre-incremented so that the pointer is in the correct
;place on the fall through into printing, hence the -1 above
ld B,10
.p_dec1
inc hl
ld a,(hl)
cp +ASC"1"
JR HS p_dec2
djnz p_dec1
;if we get to here all 10 digits were Zero's and B is 0
```

```
.last_digit
inc B           ;so set B back to one (inc is shorter &
                ;faster) so that at least one digit is printed
.p_dec2
RST &28         ;A holds the first digit on entry so start
DB &AC          ;with the printing
inc HL
ld a,(hl)       ;load next digit wasted on final pass but makes
                ;for short code
djnz p_dec2
pop af
pop bc
pop hl
RET


; 32 bit math constants
.one
EQUD 1
.ten
EQUD 10
.hundred
EQUD 100
.thousand
EQUD 1000
.tenK
EQUD 10000
.hundredK
EQUD 100000
.oneM
EQUD 1000000
.tenM
EQUD 10000000
.hundredM
EQUD 100000000
.oneG
EQUD 1000000000

.two
EQUD 2

;Binary to decimal, of Math_b to workspace
;limited to 2^31 or smaller
.bin_to_dec
PUSH HL
PUSH BC
PUSH AF
;fill workspace with 10 ascii zero's
ld hl,workspace+10
ld b,10
.bin_dec1
dec HL              ;fill backwards so that HL finishes pointing
```

```
                        ;to the start of the workspace
ld (hl),+ASC"0"
DJNZ bin_dec1
;setup the alternate registers for the math pointers
exx
ld DE,oneG
ld HL,math_b
ld BC,math_B
exx
;LD HL, workspace     ;not needed now the fill counts down.
ld B,9                ;loop down as far as 10's last digit can be
                      ;done with an add
.bin_dec2
exx
call sub_32
exx
ld a,(math_b+3)
add A,A             ;if the high byte is negative it's overflowed
jr C add_back       ;shift sign to carry as no JR on minus
inc (hl)            ;no overflow so increment the binary digit
jr bin_dec2

.add_back
exx
call add_32
dec de              ;move DE' to the next lower digit before
returning to
dec de              ;the main register set
dec de
dec de
exx
inc HL              ;move on to the next ASCII digit
DJNZ bin_dec2
ld a,(math_b)       ;get the last digit
add a,+ASC"0"       ;convert to ASCII
ld (HL),A           ;and store, pointer already incremented
pop AF
POP BC
POP HL
RET

; ##### Integer math routines #####
; all use pointers in one or more register pairs, to 4 byte
little endian values.

;(BC)=(HL)+(DE)
.add_32
push HL
push DE
PUSH BC
PUSH AF
ld a,(DE)
```

```
add a,(HL)
ld (BC),A
inc HL
INC DE
INC BC
ld a,(DE)
adc a,(HL)
ld (BC),A
inc HL
INC DE
INC BC
ld a,(DE)
adc a,(HL)
ld (BC),A
inc HL
INC DE
INC BC
ld a,(DE)
adc a,(HL)
ld (BC),A
pop af
pop bc
pop de
pop hl
ret


; (BC)=(HL)-(DE)
.sub_32
push HL
push DE
PUSH BC
PUSH AF
EX DE,HL        ;swap the pointers as there's no "SUB A,(DE)" type
commands
ld a,(DE)
sub (HL)
ld (BC),A
inc HL
INC DE
INC BC
ld a,(DE)
sbc a,(HL)
ld (BC),A
inc HL
INC DE
INC BC
ld a,(DE)
sbc a,(HL)
ld (BC),A
inc HL
INC DE
```

```
INC BC
ld a,(DE)
sbc a,(HL)
ld (BC),A
pop af
pop bc
pop de
pop hl
ret

;"A" bit shift left of (HL) used in cluster to sector calculatons
.shift_32
push AF
.shift32_loop
push HL
SLA (HL)
inc hl
RL (HL)
inc hl
RL (HL)
inc hl
RL (HL)
pop hl
dec a
jr nz shift32_loop
pop AF
ret
```

The Magrom start up with a simple 2 tone beep. CFX is much more orchestral, it plays 5 notes on start-up! Anyone that hasn't had a "close encounter" with a CFX will have to guess what they might be (Dreadful pun I know)

```
.beep
push hl
push IX
push BC
LD IX,TONES
LD B,5
.Beep_note
ld L,(IX+0)
LD H,(IX+1)
ld c,(IX+2)
call beep_hl
.beep_loop
call CF_delay
dec C
jr nz beep_loop
INC IX
INC IX
INC IX
```

```
call killsound
call CF_delay
DJNZ beep_note
POP BC
pop IX
pop hl
ret

.TONES
DW 319
DB 30
DW 284
DB 30
DW 358
DB 30
DW 716
DB 30
DW 478
DB 45

;unlike the rom routine this one only silences channel 0
.killsound
PUSH AF
LD A,&9f          ; ATTENUATION OFF TONE 1
OUT (6),A
IN A,(3)
POP AF
RET

;"beep" code from the Magrom
.beep_hl
push AF
push BC
LD A,L
AND &0F
OR &80
call safe_sound          ; SEND TONE 1 + 4 BITS OF FREQUENCY
LD A,L
SRL A
SRL A
SRL A
SRL A
LD C,A
LD A,H
SLA A
SLA A
SLA A
SLA A
OR C
AND &3F                  ; REMAINING 6 OF THE 10 BITS OF
FREQUENCY
call safe_sound
```

```
LD A,&90                    ; ATTENUATION 0DB TONE 1
call safe_sound
pop bc
pop AF
RET


.Safe_sound
OUT (6),A
IN A,(3)
ex (SP),HL
ex (SP),HL
RET
```

There are 40 free character definitions that could be used to create the logo, code 0 to 31 which aren't used in the displayable part of the ASCII character set and 248 to 255 which is the last 64 bytes of the memory where the text screen and character definitions overlap but is beyond the end of the 24th row.

The logo is 11 characters wide by 2 high, giving a 66 by 16 resolution, as in text mode each character is 6 by 8 pixels.

```
.logo
LD a,&00
out (2),a
ld a,&58
out (2),a
ld HL,character_data
ld B,176
.logo_loop
ld a,(hl)
inc HL
out (1),A
djnz logo_loop

Ld a,69
out (2),a
ld a,&5c
out (2),A
ld a,1
call do_logo_row

Ld a,29
out (2),a
ld a,&5c
out (2),A
ld a,0
.do_logo_row
ld b,11
.logo_loop2
out (1),a
inc a
inc a
```

```
djnz logo_loop2
ret
```

Character data is declared in binary as that allowed the pixels to “drawn” without having to convert to Hex or decimal. During testing the logo was adjusted up one row, and rather than re-format everything, the first byte was simply moved from the start to the end.

```
.character_data
DB %11111100
DB %00000000
DB %00111100
DB %01111100
DB %11100000
DB %11000000
DB %11000000
DB %11000000
DB %11000000
DB %11000000
DB %11100000
DB %01111100
DB %00111100
DB %00000000
DB %11111100


DB %00000000
DB %11111100
DB %00000000
DB %10011100
DB %11011100
DB %11011000
DB %00011000
DB %00011000
DB %00011100
DB %00011100
DB %00011000
DB %11011000
DB %11011000
DB %10011000
DB %00000000
DB %11111100

DB %00000000
DB %11111100
DB %00000000
DB %11111000
DB %11111000
DB %00000000
DB %00000000
DB %00000000
DB %11100000
```

```
DB %11100000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %11111100


DB %00000000
DB %11111100
DB %00000000
DB %11000000
DB %11100000
DB %01110000
DB %00111000
DB %00011100
DB %00001100
DB %00011100
DB %00111000
DB %01110000
DB %11100000
DB %11000000
DB %00000000
DB %11111100


DB %00000000
DB %11111100
DB %00000000
DB %00011000
DB %00111000
DB %01110000
DB %11100000
DB %11000000
DB %10000000
DB %11000000
DB %11100000
DB %01110000
DB %00111000
DB %00011000
DB %00000000
DB %11111100


DB %00000000
DB %11111100
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
```

```
DB %01110000
DB %10001000
DB %10000000
DB %01110000
DB %00001000
DB %10001000
DB %01110000
DB %00000000
DB %11111100

DB %00000000
DB %11111100
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %10001000
DB %10001000
DB %10001000
DB %01110000
DB %00100000
DB %00100000
DB %00100000
DB %00000000
DB %11111100

DB %00000000
DB %11111100
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %01110000
DB %10001000
DB %10000000
DB %01110000
DB %00001000
DB %10001000
DB %01110000
DB %00000000
DB %11111100


DB %00000000
DB %11111100
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
```

```
DB %11111000
DB %00100000
DB %00100000
DB %00100000
DB %00100000
DB %00100000
DB %00100000
DB %00000000
DB %11111100

DB %00000000
DB %11111100
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %11111000
DB %10000000
DB %10000000
DB %11100000
DB %10000000
DB %10000000
DB %11111000
DB %00000000
DB %11111100


DB %00000000
DB %11111100
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %00000000
DB %10000100
DB %11001100
DB %10110100
DB %10000100
DB %10000100
DB %10000100
DB %10000100
DB %00000000
DB %11111100

DB %00000000
```

9.2.3 CFX CPM Screen Driver

CPM traditionally used an 80 column by 24 or 25 row display. The VDP in the MTX has only 2 display options 40 by 24 or 32 by 24, and because of screen positioning issues on a typical TV of the era even those ranges weren't fully visible.

For CPM mode the CFX there were 2 basic options, use the text display, and ignore the first column for a 39 x 24 display that would be visible on most screens. Or use the 256 pixel wide graphics display and reduced size character matrix to squeeze in more columns. The graphics modes have the same positioning issues that text mode has so the first few pixels are ignored with 250 or so visible.

To fully form all the characters, a minimum of 5 pixels are required ("m" or "w"), plus a pixel for spacing between characters gives you the same spacing that text mode uses, so there is nothing to be gained switching to graphics mode. To make room for more characters some of the clarity has to go.

Programming wise, using half a character is a lot easier than any of the other options so I designed a 3 pixel wide set of characters, readability isn't great, but it is good enough to use short term.

On my test screen 31 of the 32 columns were usable, so I settled for 62 columns. For the maximum performance I decided not to update the colours, the resulting display being mono.

All the CPM hardware drivers (Video, keyboard and Storage) have to fit into 4k at the top of the memory map. Space is sufficiently tight that there isn't room to store 256 character definitions within the driver so the driver switches the ROM back in every time a character is drawn. With there be less restrictions on rom space where 8k is available, characters are stored twice in each 8 byte definition, the driver then picks the left or right half when printing.

The driver is a very heavily modified version of the original FDX screen driver. The original programmer used self-modifying to handle multi byte commands and selectable fonts. That code is all still present, though some of it is redundant given the reduced capabilities of the display compared to the FDX.

```
; ####### VDP OUTPUT ENTRY POINT ######
.VDU_OUTPUT
PUSH AF
PUSH BC
PUSH DE
PUSH HL
push ix
push iy
LD   HL,v_exit
PUSH HL
CALL killcur
.jp_vec
JP   initl
```

The main entry point is modified after the first call, so that instead of jumping to "initl" it jumps to "Crtgo" instead. Basically, the first time the driver is called, it resets/clears the screen without being explicitly told to do that.

For the CFX I had to add in the additional code to save the index registers, the original driver didn't need to do that as it only used the 8080 compatible registers. That required corresponding changes to the exit routine. Because the driver is run from ram, the local storage is embedded into the code.

```
.ascr
DB &20
.v_attr
DB &02
.scrflg
DB &01
.pratr
DB &02
.npatr
DB &02
.wrmsk
DB &E0
.xloc
DB &00
.yloc
DB &00
.csr_flag
DB &FF

; jump table for printing control codes
.ctab
DW      DUMMY   ; ^@
DW      DOTDO   ; ^A
DW      VCTDO   ; ^B
DW      CXYDO   ; ^C
DW      BKGSET  ; ^D
DW      EOLDO   ; ^E
DW      ATRSET  ; ^F
DW      BELDO   ; ^G
DW      BSDO    ; ^H
DW      TABDO   ; ^I
DW      LFDO    ; ^J
DW      UPDO    ; ^K
DW      CLRDO   ; ^L
DW      CRDO    ; ^M
DW      BLSET   ; ^N
DW      BLOFF   ; ^O
DW      COLSET  ; ^P
DW      COLSET  ; ^Q
DW      COLSET  ; ^R
DW      COLSET  ; ^S
DW      COLSET  ; ^T
DW      COLSET  ; ^U
DW      COLSET  ; ^V
DW      COLSET  ; ^W
DW      INITLZ_CRT  ; ^X
DW      FWDDO   ; ^Y
```

```
DW      HMEDO   ; ^Z
DW      ESCDO   ; ^[
DW      SCRSET  ; ^\
DW      PGESET  ; ^]
DW      CSSET   ; ^^
DW      CSOFF   ; ^_
```

First call entry point, save the character code then clear the screen, before restoring the character and changing the jump vector, which is all original code.

```
.initl
PUSH BC
CALL clrdo
POP  BC
LD   HL,crtgo
LD   (jp_vec+1),HL
.crtgo
LD   A,C
AND  &E0
JR   Z,ctrl_code
```

Once the original code has detected a control character, the modified printing routine takes over. The 62 column driver does not have hardware scroll or a hardware cursor like the FDX so jumps direct to print the character, and move the cursor forward before exiting through the pre-stacked routine.

```
;       PRINTABLE CHARACTER in C
;.frig1
;CALL dummy        ;code only needed with selectable fonts
call do_print
CALL FWDDO
RET             ;jump to exit via stacked value above

.v_exit
;CALL xycalc
CALL setcur
pop iy
pop ix
POP  HL
POP  DE
POP  BC
POP  AF
RET
```

The characters are stored in rom as 8 consecutive bytes, with identical left and right halves. Once the location of the character is calculated, the rom is temporarily paged in, and all 8 bytes copied to a temporary buffer.

The screen address is then calculated, and the 8 screen bytes are also copied into the buffer. Because of the VDP's auto incrementing pointer, copying all 8 bytes this way, then writing all 8 modified bytes back

saves having to keep re-setting the VDP pointer, It only has to be done twice, instead of the 16 times that a byte by byte update would require.

```
;print the character in the "c" register
.do_print
;calculate character position character map starts at &2800 in
rom 4
ld H,5
ld L,C          ;HL =&0500 + "C"
add hl,hl       ;    &0A00 + C*2
add hl,hl       ;    &1400 + C*4
add hl,hl       ;    &2800 + C*8
ld de, scroll_buff
ld BC,8
;move 8 bytes of character data into temporary storage
;no interrupts for this bit as the rom has to be paged in
DI
ld a,&40
out (0),A     ;page in ROM 4
LDIR
ld a,&80
out (0),A     ;back to CPM mode
EI
```

Notice that interrupts are disabled while the memory map is changed, just in case. In theory nothing in the standard CPM system is using interrupts, but that doesn't mean user software can't.

```
;next grab the 8 bytes off the screen into the next 8 bytes
push DE
call xycalc
call set_read
pop DE          ;push pop is shorter than re-loading DE with
                ; buffer+8
ld b,8
.char_data_loop
in a,(1)
ld (de),A
inc DE
djnz char_data_loop
call set_write  ;re-set the VDP pointer before moving on to
                ;the next bit of code
```

To do the actual printing D and E are used as masks for speed they only need to be set up once per character.

```
; now work out if we want the left or right hand side of the
character
ld a,(xloc)
and 1
jr z plot_even
;plot_odd
ld de,&F00F          ;left side of new character and right side
                     ;of onscreen character
jr plot
```

```
.plot_even
ld de,&0FF0           ;right side of new character and left side
                      ;of onscreen character
```

With the mask data set up the character data is read in from the buffer and one half blanked. The screen data is read, and has the other side blanked. The 2 halves are then re-combined and sent to the VDP

```
.plot
ld ix,scroll_buff
ld b,8
.plot_loop
ld a,(IX+0)
and E
ld C,A
ld a,(IX+8)
and D
or C
out (1),a
inc IX
djnz plot_loop
RET
```

The original code to jump to the control code vector is used unchanged. There are 32 possible control codes, so HL is set to point to the start of the table, DE is then formed as control code value x 2 and added to HL to get a pointer to the start of the vector, that address is then read and transferred to HL when it's entered via the indirect jump. The secondary control code table linked to the escape character (#1B) is accessed the same way.

```
;       CONTROL CODE
.ctrl_code
LD   HL,ctab
LD   D,&00
LD   A,C
ADD  A,C
LD   E,A
ADD  HL,DE
LD   A,(HL)
INC  HL
LD   H,(HL)
LD   L,A
JP   (HL)

.escdo
CALL frigit          ;redirect VDU stream
LD   A,C
CP   &20
JR   C,v_normal
AND  &1F
ADD  A,A
LD   HL,esctab
LD   E,A
LD   D,&00
```

```
ADD  HL,DE
LD   A,(HL)
INC  HL
LD   H,(HL)
LD   L,A
JP   (HL)
```

The cursor XY code is also as per the FDX, so each the XY position data is offset by 32. Invalid cursor positions are silently filtered out by the checks against the screen dimensions. The don't generate any errors.

```
;cursor X,Y invalid values ignored
.cxydo
CALL frigit          ;redirect VDU stream
LD   A,C
SUB  &20
AND  &7F
CP   display_width
JR   NC,cxskip
LD   (xloc),A
.cxskip
CALL frigit          ;redirect VDU stream
LD   A,C
SUB  &20
AND  &7F
CP   screen_rows
JR   NC,cyskip
LD   (yloc),A
.cyskip
```

The self-modifying code uses 2 main routines, one to move the call pointer into the required subroutine, the other to put it back to the main entry point. The cursor XY code falls through onto the return to normal routine on exit.

```
;this section returnes the VDU stream to normal after assembling
a multi byte command
.v_normal
LD   HL,crtgo
LD   (jp_vec+1),HL
RET

;set the VDU stream to code following the call to frigit, for
assembling multi byte commands
.frigit
POP  HL
LD   (jp_vec+1),HL
RET
```

To maintain as much compatibility with the FDX display as possible, the modified drive will accept all the original control codes, it just ignores the codes it doesn't support. The attribute code is unchanged from the FDX, the internal values etc are updated, but nothing in the driver makes use of the values saved.

```
;character background not setable on V9958, so code does nothing
.bkgset
CALL frigit          ;redirect VDU stream
LD   A,C
AND  &07
RLCA
RLCA
RLCA
LD   C,A
LD   A,(pratr)
AND  &C7
OR   C
LD   (pratr),A
LD   A,(npatr)
AND  &C7
OR   C
LD   (npatr),A
JP   v_normal

;no attributes on 9928 code does nothing
.atrset
CALL frigit          ;redirect VDU stream
LD   A,C
LD   (pratr),A
LD   (npatr),A
JP   v_normal
```

The FDX 80 column board has a 2nd character rom that defines a set of 2 x 4 block characters for low resolution plotting and line drawing. The original code to assemble the plot or draw command is retained, but no output is produced, as there is no equivalent of the block graphics on CFX.

```
; no bitmap character set on the 9928 setup so plot and draw
disabled
; swallow the character codes to maintain compatibility
.dotdo
CALL frigit          ;redirect VDU stream
LD   A,C
SUB  &20             ;&20 byte offset in plot commands
LD   (initl),A        ;save first perameter
CALL frigit          ;redirect VDU stream again
LD   A,C
;SUB  &20
LD   H,A             ;2nd parameter
LD   A,(initl)
LD   L,A
;CALL plotd
JP   v_normal          ;reset stream

.vctdo
CALL frigit
LD   A,C
SUB  &20
```

```
LD   (initl),A
CALL frigit
LD   A,C
SUB  &20
LD   (initl+1),A
CALL frigit
LD   A,C
SUB  &20
LD   (initl+2),A
CALL frigit
LD   A,C
SUB  &20
LD   C,A
;CALL plotv
JP   v_normal
```

Although a lot of the display commands aren't supported, the screen init code and blink on/off are included more or less unchanged.

```
.initlz_crt
LD   A,&FF
LD   HL,scrflg
LD   (HL),&FF
INC  HL
LD   (HL),&02
INC  HL
LD   (HL),&02
INC  HL
LD   (HL),&E0
CALL    CSSET
CALL    CRDO
CALL    LFDO
ret
;JP      ESTD

;blink on and off not used, as no character level attributes

.blset
LD   A,(pratr)
OR   &40
LD   (pratr),A
RET

.bloff
LD   A,(pratr)
AND  &BF
LD   (pratr),A
RET
```

Cursor on/off is supported, the character printing routine updates the cursor on exit, and will use the value stored here to decide whether to make it visible or not.

```
;no hardware cursor on 9928 in text mode
;turn cursor on value will be used as bit mask
```

```
.csset
ld a,&FF
ld (csr_flag),a
RET

.csoff
xor a
ld (csr_flag),a
RET
```

Scroll mode code, is a simple on/off switch that gets checked on row 24 of the display. The screen will with scroll and stay on row 24, or reset to zero.

```
;scroll flag is 0 for page mode (no scroll)
.scrset
LD   A,&FF
LD   (scrflg),A
RET

.pgeset
XOR  A
LD   (scrflg),A
RET

; 9928 does not support individual charater attributes in text
; mode, so this code does nothing
.colset
LD   A,C
AND  &07
LD   C,A
LD   A,(pratr)
AND  &F8
OR   C
LD   (pratr),A
RET
```

The control codes for positioning the cursor have been tweaked a little from the original Memotech code, greater use is made of falling through from one routine to the next where possible to improve performance.

```
; carriage return just sets the X location back to zero
; return used for dummy entry point for the nul codes
.crdo
XOR  A
LD   (xloc),A
.dummy
RET


; tab forward by setting the low bits of the x location then drop
into cursor right
```

```
.tabdo
LD   A,(xloc)
OR   &07
LD   (xloc),A
.fwddo
LD   A,(xloc)
CP   display_width+-1
JR   Z,fwdl
INC  A
LD   (xloc),A
RET

; need to go down a line so reset x to zero and drop through into
cursor down
.fwdl
XOR  A
LD   (xloc),A
.lfdo
LD   A,(yloc)
CP   &17           ; are we on the 24th row (rows are 0-23)
JR   Z,LFS         ;jump forward to test the scroll flag
INC  A
.pagem
LD   (yloc),A
RET

.lfs
LD   A,(scrflg)
OR   A
JR   Z,pagem      ;set Y back to the top of the screen if it's
                  ;page mode
CALL scrup        ;otherwise scroll up, leaving Y as 23
RET
```

The FDX uses hardware so sound an on-board buzzer, for the CFX the PSG has to be used instead, and so a short tone is played on channel 1

```
; "DING"
;"beep" code from the Magrom
.beldo
push AF
push BC
push hl
ld HL, 284
LD A,L
AND &0F
OR &80
call s_sound          ; SEND TONE 1 + 4 BITS OF FREQUENCY
LD A,L
SRL A
SRL A
SRL A
SRL A
```

```
LD C,A
LD A,H
SLA A
SLA A
SLA A
SLA A
OR C
AND &3F                  ; REMAINING 6 OF THE 10 BITS OF
FREQUENCY
call s_sound
LD A,&90                 ; ATTENUATION 0DB TONE 1
call s_sound
ld bc,00A0
.delay_loop
djnz delay_loop
dec c
jr nz delay_loop
;kill sound
LD A,&9f          ; ATTENUATION OFF TONE 1
call s_sound
pop hl
pop bc
pop AF
RET

;send data to sound port with delay to guarantee sound chip time
to load it.
.S_sound
OUT (6),A
IN A,(3)
ex (SP),HL
ex (SP),HL
RET


;non clearing backspace
.bsdo
LD   A,(xloc)
OR   A
JR   Z,bsu
DEC  A
LD   (xloc),A
RET

;backspace needs to go up a line
.bsu
LD   A,display_width+-1
LD   (xloc),A
LD   A,(yloc)
OR   A
JR   Z,bss
DEC  A
```

```
LD   (yloc),A
RET

;we're already at 0,0 so need to put the x position back to 0
.bss
;   XOR  A    ; not needed A was 0 or we wouldn't be here?
LD   (xloc),A
RET

;cursor up
.updo
LD   A,(yloc)
OR   A
ret  Z             ;can't go up from top row
DEC  A
LD   (yloc),A
RET

.clrdo
CALL clrscn     ;clear the screen and fall through into home to
                ;set the cursor
.hmedo
XOR  A
LD   (xloc),A
LD   (yloc),A
RET
```

Erase line, temporarily sets the start of the line to zero, then calls erase to the end of the line routine, before restoring the cursor position withing the line – all original code.

```
.erln
LD   A,(xloc)
PUSH AF
XOR  A
LD   (xloc),A
CALL eoldo
POP  AF
LD   (xloc),A
RET

.eoldo
CALL xycalc
LD   A,(xloc)
LD   B,A
JP   erase_eol
```

Most of the "utilities" from the original code deal with the alternate character mops, which aren't supported.

```
;       UTILITIES

.grpmap
LD   A,C
```

```
AND  &7F
LD   C,A
AND  &40
RET  Z
LD   A,C
AND  &20
RLCA
RLCA
OR   C
AND  &9F
LD   C,A
RET

.altmap
LD   A,C
OR   &80
LD   C,A
RET

.getmsk
LD   A,C
CP   &30
JR   NZ,getbit
XOR  A
RET

.getbit
DEC  A
AND  &07
LD   C,A
CALL ncalc
OR   A
RET
```

To meet the different VDP set-up requirements for reading and writing to the VRAM, there is one routine for each type of VRAM access

```
; set the VRAM pointer to HL for reading and writing respectively
.set_read
push AF
ld a,L          ;setup VDP address
out (2),a
and &3f
ld a,h          ;bit 6 and 7 clear for VRAM read
out (2),a
pop AF
ret

.set_write
push AF
ld a,L          ;setup VDP address
out (2),a
```

```
ld a,h          ;set bit 6, bit 7 clear for VRAM write
or &40
out (2),A
pop AF
ret
```

The 6845 on the FDX 80 column board has a built-in support for a hardware cursor. To provide something similar on the VDP means using a sprite. The cursor on code will always set the sprite to the cursor position. If the cursor is off its colour is set to transparent. If it's on, the colour is set to white. The code is capable of sending data faster than the VDP can accept it, so a delay is added to every access to make sure the timing requirements are met.

```
;cursor on/off needs to preserve registers as is called before
any VDU output
;uses sprite 0, pattern 0,
.setcur
push hl
push DE
push af
ld HL,&3f00
call set_write
ld a,(yloc)
add a,a
add a,a
add a,a
sub 2
CALL OUTA           ;send sprite Y
ld a,(xloc)
add a,2             ;not using first column
add a,a
add a,a
CALL OUTA           ;send Sprite X
ld a,0
CALL OUTA           ;send sprite number
ld a,(csr_flag)     ;get the cuersor flag
and &0f             ;and with the colour, resulting in a
                    ;transparent sprite if the flag is zero
CALL OUTA           ;send sprite colour
pop af
pop DE
pop hl
RET

;USE CALL RETURN TO DELAY SCREEN ACCESSES
.OUTA
OUT (1),A
RET
```

The cursor off routine disables sprite processing by setting the sprite vertical position to 208, which is a "magic number" that in addition to being off screen since there are only 192 displayed rows, is also used as an off switch by the VDP for the sprite processing for that frame.

```
.killcur
push hl
push DE
push af
ld HL,&3f00
call set_write
ld a,208          ; set vertical position to "stop sprite
                  ; processing"
out (1),a
pop af
pop DE
pop hl
RET
```

The screen x,y position calculator has 2 entry points, the main one, and a secondary one where the DE pair already holds the cursor position. The routine is remarkably simple. Because the screen is 32 rows, of 8 bytes, there are 256 bytes per row, meaning there is no need for any row calculation, the value can be used directly.

The position within the row, has 2 characters sharing each block of 8 bytes, requiring the lowest bit to be discarded. The modified value has to be multiplied by 4 (as there are 8 bytes per pair of character positions). If the display were to be tuned to use 60 characters, ignoring the first 2 columns, then the offset would increase to 16.

```
; calculate the cursor position
.xycalc
LD   A,(xloc)
LD   D,A
LD   A,(yloc)
LD   E,A
.calc1
; enter here if DE already set
; no calculation required to find the high byte,
; low byte needs the low bit removing and the remainder
multiplied
; by 8
LD   H,E
LD   a,D
and &FE
add a,a
add a,a
add a,8          ; add in the offset as the first column isn't
                 ; being used
ld  l,a
RET
```

Ncalc is a redundant routine used to convert a number in (0-7) C to a bit position in A.

```
.ncalc
INC  C
LD   A,&01
.ncalcl
DEC  C
```

```
RET  Z
RLCA
JR   ncalcl
```

Some of the more “exotic” escape codes require more than a few lines of code to implement. However is seems that NewWord (and possibly other CPM software) uses the codes extensively so they’re all implemented.

```
;delete line at cursor and scroll up
; set HL to point to the start of the line below
; set C to the number of lines remaining
; then call the scroll up code at it's looping point
.edcsln
ld a,(yloc)
cp &17           ;are we already on the last line ?
jp z blank_last
inc a
ld e,a
ld d,0
call calc1
ld a,e
CPL
ADD a,24
ld c,a
inc c
call scroll_loop     ;does the scroll and blanks the last line
jp v_normal
```

To get the best performance from the scrolling, a whole line is copied at a time from the VDP to main memory, and then copied back in the new position.

```
.scrup
ld HL,&100       ;start with line 1
ld c,23          ;23 rows to scroll
.scroll_loop
call set_read
ld de,scroll_buff
ld b,0
.scroll_read
in a,(1)
ld (de),a
inc DE
djnz scroll_read
ld de,&FF00      ;subtract 256 from HL to find the address to
                 ;write
add hl,de
call set_write
ld de,scroll_buff
ld b,0
.scroll_write
ld a,(de)
```

```
out (1),a
inc DE
djnz scroll_write
ld de,&200      ;add 2x line length
add hl,de       ;move HL on to the next line to be read in
dec c
jr nz,scroll_loop
jp blank_last
```

Although the screen driver is mono, and none of the character updates will change the colour attributes, early versions of the CLS code did set the background colour. The “final” build however disabled that for speed, and just cleared the character area.

```
.clrscn
ld hl, &0000
call set_write
ld BC,6144
.CLRSCLP
xor A
out (1),a
dec bc
ld a,b
or c
JR NZ,clrsclp
;ld hl,&2000
;call set_write
;ld BC,6144
;.CLRLP2
;ld a,screen_colour
;out (1),a
;dec bc
;ld a,b
;or c
;JR NZ,clrlp2
RET
```

Erase to the end of the line by poking 0’s into VRAM for each pair of characters left on the line. It does nothing special for the half character. In practice it doesn’t seem to affect the output, though I can foresee some circumstances where the display could be corrupted if cursoring backwards before issuing this command.

```
; erase to the end of the line from character B
;xycalc has been called so HL holds the screen position
.erase_eol
;print a space if we're in the "odd" character position ???
LD A,64
sub b
add a,a
add a,a   ;multiply by 4 if we've sorted the half character
properly.
```

```
LD   B,A
call set_write
xor A
.ereol_lp
call outa
djnz  ereol_lp
ret
```

The escape sequence table has only a few unique commands, the majority of the command available are duplicates of control codes.

```
;       ESCAPE SEQUENCE LOOK-UP TABLE

.ESCTAB
DW      v_normal  ; @
DW      v_normal  ; DW      EALT    ; A set alternate character
font
DW      EBOTH   ; B set both attribute bytes
DW      ESCRL   ; C set scroll mode
DW      EPGE    ; D set page mode
DW      ECSON   ; E cursor on
DW      ECSOFF  ; F cursor off
DW      v_normal  ; DW      EGRPH   ; G set graphic font
DW      v_normal  ; H
DW      EIBLLN  ; I insert blank line at cursor move the other
lines down
DW      EDCSLN  ; J delete line at cursor move the other lines up
DW      v_normal  ; K
DW      v_normal  ; L
DW      v_normal  ; M
DW      ENPATR  ; N set non printing attribute
DW      v_normal  ; O
DW      EPRATR  ; P set printing attribute
DW      v_normal  ; Q      ; in SCPM ROM, disable color, cls
DW      EREAD   ; R      ; in SCPM ROM, enable color, cls
DW      v_normal  ; DW      ESTD    ; S set standard font
DW      ESIPR   ; T
DW      ESINP   ; U
DW      ESIBT   ; V
DW      EWRMS   ; W
DW      CNTSIM  ; X
DW      v_normal  ; Y
DW      v_normal  ; Z
DW      v_normal  ; [
DW      v_normal  ; \
DW      v_normal  ; ]
DW      v_normal  ; ^
DW      v_normal  ; _
```

These control codes were removed from the table – they now just exit.

```
;       ESCAPE SEQUENCE HANDLERS
;only one character set provided in text 2 mode
```

```
;.ESTD
;LD   HL,DUMMY
;LD   (frig1+1),HL
;JP   v_normal

;.ealt
;LD   HL,altmap
;LD   (frig1+1),HL
;JP   v_normal

;.egrph
;LD   HL,grpmap
;LD   (frig1+1),HL
;JP   v_normal
;
;simulate control character
.cntsim
CALL frigit
LD   A,C
AND  &1F
LD   C,A
CALL v_normal
JP   ctrl_code

.escrl
CALL scrset
JP   v_normal

.epge
CALL pgeset
JP   v_normal

.ecson
CALL csset
JP   v_normal

.ecsoff
CALL csoff
JP   v_normal

; set attributes directly, rather than by bit
.esipr
CALL frigit
LD   A,C
LD   (pratr),A
JP   v_normal

.esinp
CALL frigit
LD   A,C
LD   (npatr),A
JP   v_normal
```

```
.esibt
CALL frigit
LD   A,C
LD   (npatr),A
LD   (pratr),A
JP   v_normal
```

Clearing the last line is used in scrolling and elsewhere so is implemented as a routine of its own. Note the NOP instructions to ensure VDP timing requirements are satisfied.

```
.blank_last
ld HL,&1700        ; = the start of the last screen row
.blank_current
call set_write
xor A
ld b,A
.blanks_loop
out (1),a
nop
nop
djnz blanks_loop
jp v_normal
```

Another one of the “pesky” escape commands that needed extensive coding.

```
;insert line at cursor and scroll down
.eiblln
ld a,(yloc)
cp &17
jr z blank_last     ;on the last row, just blank it
CPL                 ;ones complement so A=-(row)-1
add a,24            ;add 24, as we want one less row - to allow
                    ;for inserting
ld c,a              ;rows to scroll
ld HL,&1600         ;working fom the bottom up, row 22 (to row
                     ;23) is always the first to be moved
.scroll_d_loop
call set_read
ld de,scroll_buff
ld b,0
.scroll_d_read
in a,(1)
ld (de),a
inc de
DJNZ scroll_d_read
ld de,&100            ;move down to the start of the next row
add hl,DE
call set_write
ld de,scroll_buff
```

```
ld b,0
.scroll_d_write
ld a,(de)
out (1),a
inc DE
djnz scroll_d_write
ld de,&FE00       ;-&200 to move up 2 rows
add HL,DE
dec C
jr nz scroll_d_loop
ld de,&100         ;move back to the "current" line
add hl,de
; now insert the blank line,
JP  blank_current
```

The FDX display being character based, it was possible to read the character at the cursor, the VDP can't do that, but "something" has to be returned to the top line of the character display is read, the VRAM pointer is set up, so the remaining 7 bytes could be read, and the character shape detected. But that probably won't ever be done!

```
;read character at cursor ??
.eread
CALL xycalc
call set_read
ld   a,2             ;set an attribute just in case
ld   (v_attr),A
IN   A,(1)
LD   (ascr),A
JP   v_normal
```

More redundant code retained for compatibility.

```
; setup write mask
.ewrms
CALL frigit
LD   A,C
LD   C,&E0
CP   &30
JR   Z,swrm
LD   C,&C0
CP   &31
JR   Z,swrm
LD   C,&A0
CP   &32
JP   NZ,v_normal
.swrm
LD   A,C
LD   (wrmsk),A
JP   v_normal

;set printing attribute
```

```
.epratr
CALL frigit
CALL getmsk
JR   Z,setpr
LD   C,A
LD   A,(pratr)
OR   C
.setpr
LD   (pratr),A
JP   v_normal

; set non printing attribute
.enpatr
CALL frigit
CALL getmsk
JR   Z,setnp
LD   C,A
.enpa1
LD   A,(npatr)
OR   C
.setnp
LD   (npatr),A
JP   v_normal

;set both printing and non prining attributes. Does nothing on
V9958.
.eboth
CALL frigit
CALL getmsk
JR   NZ,v_setb
LD   (pratr),A
JR   setnp

.v_setb
LD   C,A
LD   A,(pratr)
OR   C
LD   (pratr),A
JR   enpa1
```

The Original driver ends with a local stack. In the original drivers each hardware sub system maintained its own stack, which was rather wasteful of space. A single “driver” stack could have done the job and save memory.

```
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
```

```
.vstk

END
```

9.2.4 CFX Hardware

The 82C55 PPI based IDE interface isn't my design. It's based on Paul Stroffregen's 8051 interface and code from pjrc.com, he in turn credits the hardware design to Peter Faasse whose design connected an IDE hard drive to a 63B03 using an 8255. The only other device required for the interface is a 74HCT04 which is a hex inverter.

The 82C55 and its earlier cousin the 8255 have 24 bi-directional I/O pins forming 3 x 8 bit I/O ports. This makes it better suited to building an IDE interface than the Z80 PIO Which only has 2 x 8 bit I/O ports but additional handshake lines.

The ports on the 82C55 are labelled A,B and C. Ports A and B connect to the 16 bit IDE data bus. Pins PA0 to PA7 are connected to the IDE header data bus pins D0 to D7 with Pins PB0 to PB7 connected to D8 to D15.

Port C pins PC0 to PC2 connect to the IDE line A0 to A2 for selecting which register is being accessed. PC3 to PC7 are all connected to 5 of the 6 inverters in the 74HCT04. The IDE interface uses active low signals, Peter's original notes indicate his 8255 was briefly pulsing the output signals low when changing the 8255 I/O mode. Which was apparently enough to issue a reset to the IDE device. Using inverters means a low at the PPI holds reset (and the other signals) in the inactive high state. There is then no mode change glitch.

PC3 connects to the IDE CS1 line which is used to select the primary bank of 8 IDE registers. PC4 connects to CS2 and the alternate set. In practice, none of the current software needs any of the alternate registers and CS2 is unused.

Of the final 3 pins, PC5 connects to IDE Write, PC6 to IDE read and PC7 to the IDE interface reset pin.

The 82C55 isn't a Z80 family device, its origins are the Intel 8080 family, and the reset signal is active high. This is the inverse of the Z80 requirements where reset is active low. The 6th gate of the 74HC0T4 is user to invert the main reset signal so that the 82C55 reset at the same time the Z80 does.

In addition to the 82C55 and 74HCT04 the CFX has the 128k flash chip mentioned in the software section. It also needs to be able to provide a chip select for ROM and a chip select for the 82C55. Like the Magrom the chip selects are created with 74 series logic and not any form or programmable device.

The ROM chip select needs 8 signals.

RELCPMH	Must be low, as ROM mode is required
R1	Is low for both ROM 4 and ROM 5
R2	Is high for both ROM 4 and ROM 5
A15	Is low, as the ROM lives at #2000 to #3FFF
A14	Is low, as the ROM lives at #2000 to #3FFF
A13	Is high, as the ROM lives at #2000 to #3FFF

MREQ	Is low on a memory access
RD	The ROM is read only, RD must be low

The R0 signal the determines which image ROM 4 or ROM 5 is selected.

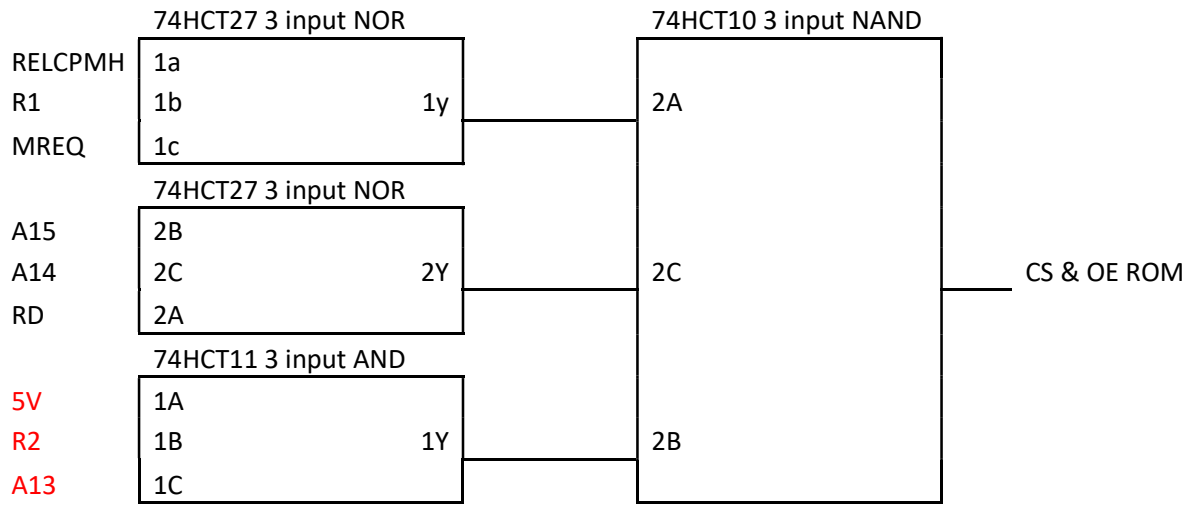
The 82C55 chip also needs 8 chip selects

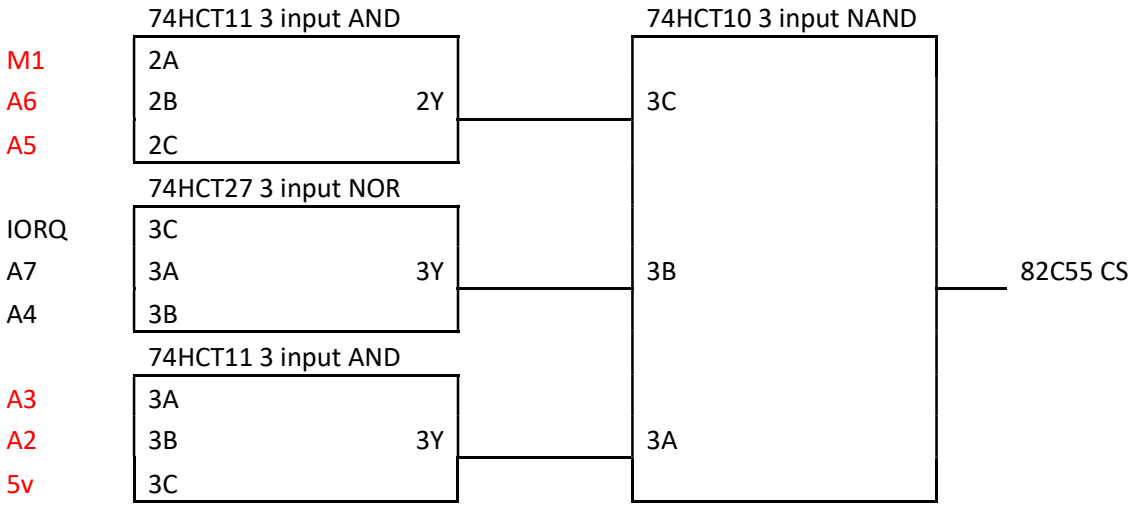
M1	Must be high to eliminate the IRQ acknowledge
IORQ	Is low, for an I/O access not memory
A7	Is low in for I/O range #C6 to #6F
A6	Is high in for I/O range #C6 to #6F
A5	Is high in for I/O range #C6 to #6F
A4	Is low in for I/O range #C6 to #6F
A3	Is high in for I/O range #C6 to #6F
A2	Is high in for I/O range #C6 to #6F

Since there are no signals in common that's a total 16 signals, 7 high and 9 low. I could have used an 8 input NAND (74HC30) for each select, but that would have required inverting the 9 low signals, and that would have needed 2 more 74HCT04s for a 4 chip solution.

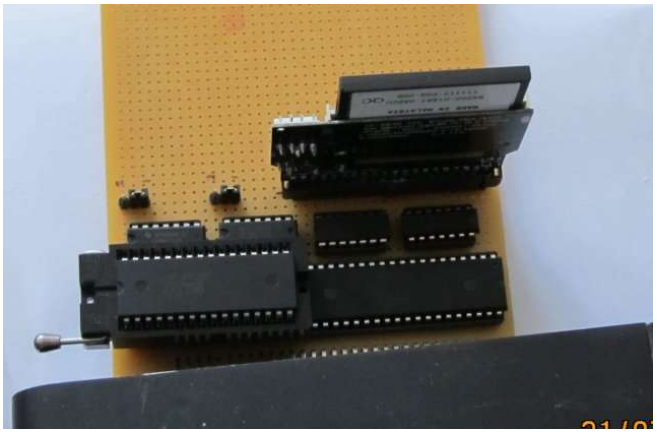
However, I realised that is was possible to use 3, 3 input devices to achieved the same result. The ROM select needs 6 low signals, the 82C55 needs 3. Using a 74HCT27, which is a 3 input NOR device combines 3 low signals an produces a high if all 3 are low. I could at that point have used the pair of 74HCT30 's to produce the chip select outputs, by using the red signals from the table above, plus the relevant outputs from the 74HCT27 and connecting the unused inputs to 5v. The untidy part of doing things that way is that signal paths are of different lengths. With the black signals from the tables above experiencing 2 delays while the red ones only have one. I therefore used a 74HCT11 which is a 3 input AND device to combine the high signals, as that will only output high if all 3 inputs are high. Both selects needed 1 input pin connected to the 5V supply as there are only 2 and 5 high signals respectively.

That then means each chip select is distilled down to 3 signals all of which are high when selected, A 3 input NAND gate is then required, to generate the final active low output, which meant using 2 of the 3 gates of a 74HCT10. This is the drawing I did at the time to map out the logic.



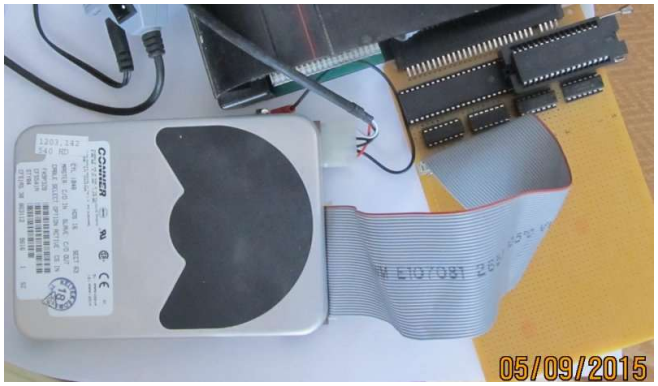


The prototype board therefor had 6 IC's and one 40 pin header. The prototype also had options for 1 ROM image or 2 but that was quickly shelved in favour of using 2 images all the time.

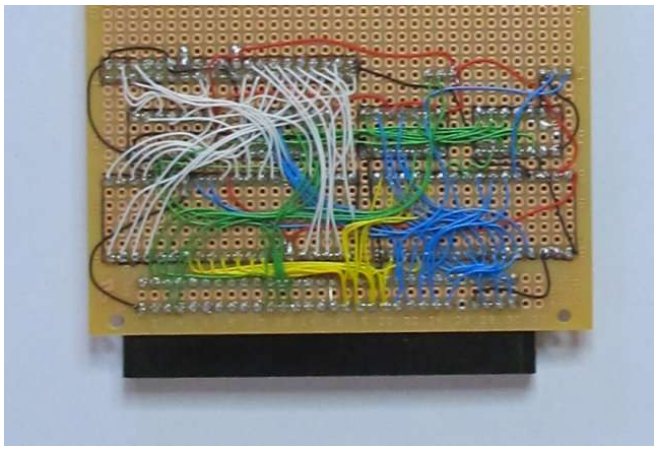


Seen here connected to the testbed MTX, the prototype made a very compact package, the upright CF card was above the level of the keyboard, so not ideal.

Test were also done with a real IDE drive, although it worked, the requirement for an external power supple for the drive and the cabling were less practical than using the self-contained CF. Also note, that by the time this picture was taken, the jumpers have been removed and the prototype was permanently configured for 2 rom images



The wiring side, connections to the 82C55 I/O ports are in white, the data bus is yellow, address bus in blue and other control signals in green. Since the flash chip has 17 address lines, and only 14 are used, other are tied low. Which shows up in the photograph as apparently empty pins on the right side of the socket. Other than power very few wires needed to connect to more than 2 points.



Buried in the wiring are surface mount smoothing capacitors for all chips and one pull up and one pull down 4k7 resistor on the ID interface. They're not required for CF use, but the ATA spec suggested they were needed for the real IDE drive.

Having sorted out the basic design, I moved on to working with the software while Dave took my notes and wiring plans and created a proper schematic and designed a proper PCB. As with the Magrom design the lower part of the board had to be kept empty to fit under the keyboard. The upright IDE header was also replaced with right angle version to bring the CF down below the keyboard keeping it out of harm's way.

That schematic and copies of the binaries etc are available on Dave's primrosebank.net website.

A first run of "Production" boards was completed in time to be demonstrated at Memofest in October 2015.

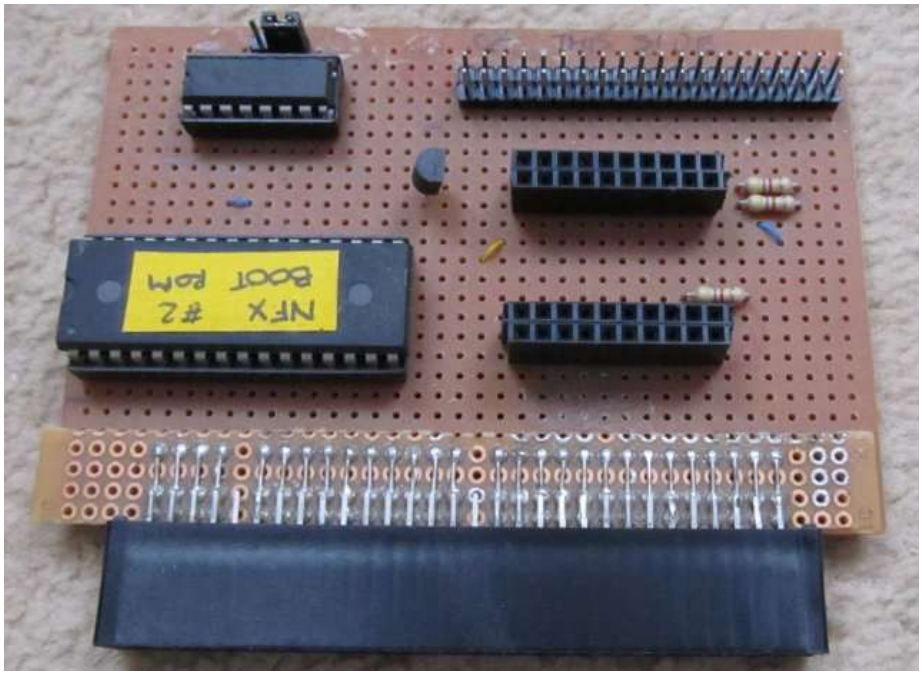
9.3 NFX

The NFX is a proof of concept board, aimed at giving the MTX an Ethernet connection using the Wiznet 5100 all in one ethernet controller. The W5100 has hardware support for TCP/IP and other network protocols. There are a multitude of Arduino "shields" that us it. I thought that if an ATmega microcontroller could talk to it then there was no reason why a Z80 shouldn't be able to do the same.

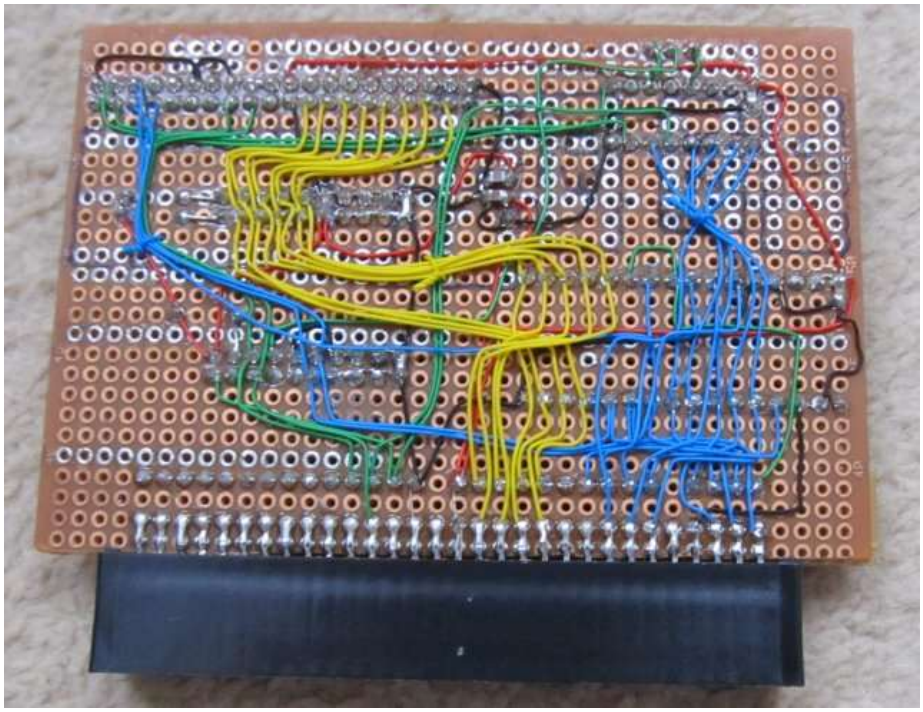
9.3.1 NFX Hardware

The minimalist NFX design uses the 8 bit IDE interface originally developed for the CFX-II, 8 K of ROM mapped as a game rom in slot 7 and a WIZ811 ethernet module.

With the CF and ethernet module removed, the minimal nature of the design is readily apparent. The only logic device on the board is a 74HC138. The WIZ5100 is a 3.3 volt device, so there is a 3 pin LDO to supply the 3v to the module, plus pull up resistors for the module SPI pins.



What isn't visible from the top is the two 0.1nf capacitors across the power supply of the 2 chips. The WIZ811 module doesn't need one, as that has multiple smoothing capacitors already fitted. 2 Further capacitors are fitted to the LDO voltage regulator



The spacing of the header pins on each header of the WIZ811 module 2.54mm, perfect for the matrix board, but with one issue. The 2 banks are 0.4mm closer together than would be ideal. There is just about enough wiggle room to "force" the fit for the prototype though the headers aren't quite square.

The WIZ5100 has 2 modes of operation, it can either present as 32k of RAM, or as 4 I/O ports. The MTX's memory map, with its 32k pages would have allowed either type of connection, however I went with the 4 I/O port option, as using one of the RAM pages complicates the driver software. The I/O port option also simplifies the hardware, as it enables the WIZ811 module to be decoded by the same device that decodes the IDE interface.

The 74HC138 is a 3 to 8 decoder and provides the I/O decode. (As an aside, the 8 bit IDE interface really needs a CMOS Z80, which is fitted to the test system, so the HC version of the decoder is fine, for an NMOS CPU the HCT version of the decoder should be used.)

The although it's a 3 to 8 decoder, the 138 actually has 6 inputs. 3 lines that it decodes into one of 8 outputs, plus 3 chip enables, one active high, and 2 active low. The address decode attaches the Z80's IORQ line to one of the active low selects, the A7 line goes to the active high input and A6 to the other active low. Which means in order for the 138 to be selected, the MTX must be doing an I/O request in the #80 to #BF range it won't respond to a memory request. The 3 decode inputs are connected to A3 to A5, each of the 8 outputs therefore responds to 8 I/O ports.

The 8 bit IDE interface is connected to the Y6 and Y7 outputs, giving it 16 I/O addresses from #B0 to #BF. The WIZ811 select used Y4 giving an 8 port address range of #A0 to #A7. The jumper gives the option of using Y2 instead of Y4, re-mapping the WIZ811 to #90 to #97. Since the WIZ5100 chip is only using 4 ports, the other 4 port will just duplicate the access to the first 4.

As with most of my designs, there's no circuit diagram for the NFX, the reverse view wiring "map" I drew up as a guide when soldering.

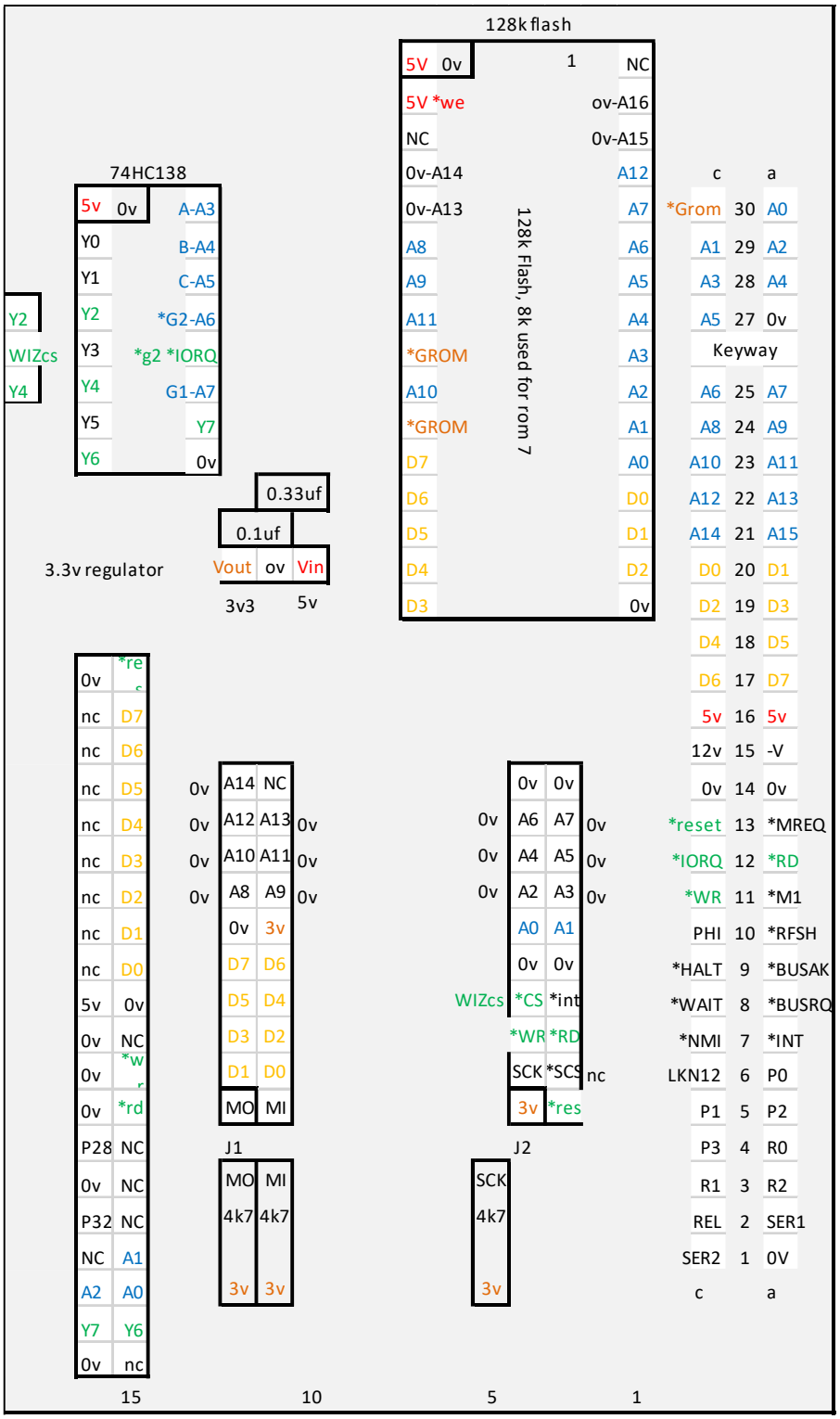
The WIZ811 module has 15 address pins to accommodate the 32k mode, 13 of those are wired to ground, only A0 and A1 are connected to the Z80 address bus. To isolate the SPI bus in the WIZ5100 MOSI, MISO and SCK pins are all pulled high (to 3v) via resistors. Other than power and the data bus the only other connections are read, write and reset from the Z80 and the module select from the 74HC138.

The IDE interface connections are similar, but that needs 3 address lines, so A2 is also needed.

The 128k flash device also has the extra address lines connected to 0v so that only 8k is seen. Because there is only 8k rom available I chose to use a modified CPM boot rom instead of the other option that would have been SDX style extensions to BASIC.

CPM being a far better environment to develop the software than BASIC would be.

NFX wiring “map” created in Excel



9.3.2 NFX IDE support

The NFX uses and 8 bit IDE interface, which is specifically intended for use by CF cards. Since it's interfacing to CPM in exactly the same way that the CFX does, the IDE driver has to supply the same 3 basic routines, initialise, read 1 sector and write 1 sector.

; CF specific low-level routines

```
;write registers
CFdata     EQU &B0
CFfeature  EQU &B1
CFcount    EQU &B2
CFLBAlow   EQU &B3
CFLBAmid   EQU &B4
CFLBAhigh  EQU &B5
CFLBAtop   EQU &B6
CFcommand  EQU &B7
;
;read registers
CFerr      EQU &B1
CFstatus   EQU &B7
;
;the actual IDE commands used
CMD_RECAL   EQU &10
CMD_READ    EQU &20
CMD_WRITE   EQU &30
CMD_INIT    EQU &91
CMD_ID      EQU &EC
CMD_SPINUP  EQU &E0
CMD_SPINDN  EQU &E1
```

The current IDE software only uses the primary bank of I/O ports, ports #B8 to #BF are decoded, but not actually used. The direct connection of the 8bit IDE makes for faster transfers and simpler software, as the Z80 can handle all of the transfers on its own, without needing to go via an 82C55 like CFX does.

```
; Initialise CF card
; after:
;   if ok, Z, CF initialised
;   if not ok, NZ, CF not initialised
;
.CPMCFInit
PUSH BC
LD a,CMD_init
OUT (CFcommand),A
;
LD B,0                          ;for the timeout counter
jr init_pass1                  ;jump past the timeout checks on
                                ;the first pass
.INIT_loop
dec B                           ;exit if we've tried 256 times
                                ;and drive still not ready
```

```
JR Z cpm_timeout
call cpm_delay                    ;CF doesn't need to spin up, but
                                  ;allow some extra time
                                  ;just in case. The 255 calls of
                                  ;the delay routine take
                                  ;approx 2.5 sec at 4mhz
.INIT_pass1
in a,(CFstatus)                   ;read status register
bit 7,a                           ;bit 7 is bsy
JR nz INIT_LOOP                  ;wait for BSY to be clear
bit 6,a                           ;bit 6 is rdy
JR z INIT_LOOP                   ;wait for RDY to be set
```

The Init command is the only one that will time out, the rest of the software assumes that the CF that was present on boot hasn't been removed.

```
ld a,1                    ;set CF 8 bit mode
out (CFfeature),A
ld a,&EF
out (CFcommand),A
call CPMwait_ready

ld a,&82                  ;set no write cache
out (CFfeature),A
ld a,&EF
out (CFcommand),A
call CPMwait_ready

;LD A,%11100000          ;select the master device. LBA mode
;out (CFLBAtop),A


POP BC
XOR A                     ;A zero'd and ZF set to indicate all OK
RET
```

Device setup requires the extra step of letting the CF card know that 8 bit mode should be used.

```
.cpm_timeout
POP BC
xor A
DEC A                     ;return A no zero and Z reset
RET
```

Not having the CF present will prevent CPM from booting, unlike CFX where running without a "disc" is a valid option. There are no error messages, as CFX used facilities from BASIC to print that, and this isn't present in CPM.

```
;waste approx 3339 * 12 is approx 40,000 cycles delay 0.01 sec
;at 4mhz.
.CPM_delay
```

```
push bc
ld BC,12
.Cpm_delay_loop
DJNZ Cpm_delay_loop    ;3323 cycles on inner loop
dec c                    ;4
JR nz Cpm_delay_loop   ;12
pop bc
ret


;
; Read block from CF card
;   before:
;     SDLBA is 32 bit block 512 byte block number, HL is buffer
;   after:
;     if ok, Z
;     if not ok, NZ
;
.CPMCFREAD
PUSH BC
PUSH hl
CALL CpmSetLBA        ;send LBA number to the drive
LD a,CMD_READ
out (CFcommand),A
CALL CpmWAIT_DRQ
AND 1                ;isolate bit 0
JR NZ,CpmGET_ERR      ;error bit set, find out why
LD BC,CFdata         ;C set to the data register INIR count
                      ;set to zero for 2 x 256
DI
INIR
INIR
EI
pop hl
POP BC
xor A               ;exit with A zero'd and Z set
RET
```

Reading and writing the CF is about twice as fast over the 8 bit interface, as although less data is transferred on each access, the Z80's block I/O commands can be used

```
; CF routines only requre A=0/Z=1 good or A<>0/Z=0 error
; however error read on return for future expansion
.CpmGET_ERR
pop hl
POP BC
in a,(CFerr)
AND A
RET NZ
DEC A   ;make sure A isn't zero and the flag isn't set
RET
;
```

```
; Write block to CF card
;   before:
;     SDLBA is 32 bit block 512 byte block number, HL is buffer
;   after:
;     if ok, Z
;     if not ok, NZ

.CPMCFwrite
PUSH BC
PUSH HL
CALL CpmSetLBA
LD A,CMD_WRITE
out (CFcommand),A
CALL CpmWAIT_DRQ
AND 1
JR NZ,CpmGET_ERR
ld BC,CFdata           ;C set to the data register OTIR count set
                        ; to zero for 2 x 256
DI
otir
otir
EI
CALL CpmWAIT_ready
AND 1
JR NZ,CpmGET_ERR
POP HL
POP BC
xor A
RET
```

The low-level support routines are pretty much the same as the CFX, but they act directly and so are shorter. Direct access means there is no need to use read/write register subroutines.

```
;
; Set CF LBA
; corrupts A
.CpmsetLBA
CALL CpmWAIT_ready
LD A,%11100000         ;Top 4 bits of address zero, b4 clear
                       ;is for LBA master
out (CFLBAtop),A       ;write LBA mode to LBA top byte register
ld A,(CPMsdlba+2)
out (CFLBAhigh),A      ;write to LBA bits 16-23
ld A,(CPMSDLBA+1)
out (CFLBAmid),A       ;write to LBA bits  8-15
ld A,(CPMSDLBA)
out (CFLBAlow),A       ;write to LBA bits  0- 7
LD A,1
out (CFcount),A        ;set the sector count to 1
RET
```

```
; wait for device to be both not busy and ready
;Returns the drive's status in A
.CpmWAIT_READY
in A,(CFstatus)          ;read status register
bit 7,a                  ;bit 7 is BSY
JR nz CpmWAIT_READY       ;wait for BSY to be clear so we can
                           ;get a good read on ready
bit 6,a                  ;bit 6 is ready
JR z CpmWAIT_READY        ;wait for RDY to be set
RET


;Wait for the drive to be ready to transfer data.
;Returns the drive's status in A
.CpmWAIT_DRQ
in A,(CFstatus)          ;read status register
bit 7,a                  ;bit 7 is BSY
JR nz CpmWAIT_DRQ         ;wait for BSY to be clear so we can
                           ;get a good read on DRQ
bit 3,a                  ;bit 3 is DRQ
JR z CpmWAIT_DRQ          ;wait for DRQ to be set
RET

;   End of 8 bit CF low level routines

END
```

9.3.3 NFX CPM screen driver.

The NFX uses the 40 column text mode for the CPM screen. There isn't any legacy CPM software that can use the WIZ5100, so there was no need to compromise the clarity of the display to maintain a wider screen. The output from some of the CPM utilities can be a little unclear, but that's acceptable given that the main focus of the NFX is transferring data over the Ethernet and that required brand new software. The screen drive, like the 62 column version for CFX is derived from the original FDX 80 column driver code.

```
; ####### VDP OUTPUT ENTRY POINT ######
.VDU_OUTPUT
PUSH AF
PUSH BC
PUSH DE
PUSH HL
LD   HL,v_exit
PUSH HL
CALL killcur
.jp_vec
JP   initl
```

Th initial setup code is identical to the CFX as is the inclusion of the variables in-lined into the code which by this point has been copied into ram.

```
.ascr
DB &20
.v_attr
DB &02
.scrflg
DB &01
.pratr
DB &02
.npatr
DB &02
.wrmsk
DB &E0
.xloc
DB &00
.yloc
DB &00
.csr_flag
DB &FF
; jump table for printing control codes
.ctab
DW      DUMMY   ; ^@
DW      DOTDO   ; ^A
DW      VCTDO   ; ^B
DW      CXYDO   ; ^C
DW      BKGSET  ; ^D
DW      EOLDO   ; ^E
DW      ATRSET  ; ^F
DW      BELDO   ; ^G
DW      BSDO    ; ^H
DW      TABDO   ; ^I
DW      LFDO    ; ^J
DW      UPDO    ; ^K
DW      CLRDO   ; ^L
DW      CRDO    ; ^M
DW      BLSET   ; ^N
DW      BLOFF   ; ^O
DW      COLSET  ; ^P
DW      COLSET  ; ^Q
DW      COLSET  ; ^R
DW      COLSET  ; ^S
DW      COLSET  ; ^T
DW      COLSET  ; ^U
DW      COLSET  ; ^V
DW      COLSET  ; ^W
DW      INITLZ_CRT  ; ^X
DW      FWDDO   ; ^Y
DW      HMEDO   ; ^Z
DW      ESCDO   ; ^[
DW      SCRSET  ; ^\
DW      PGESET  ; ^]
DW      CSSET   ; ^^
DW      CSOFF   ; ^_
```

There are no changes to the control code jump table, the target routines maintain the same names with the code updated as required.

```
.initl
PUSH BC
CALL clrdo
POP  BC
LD   HL,crtgo
LD   (jp_vec+1),HL
.crtgo
LD   A,C
AND  &E0
JR   Z,ctrl_code
;       PRINTABLE CHARACTER
;.frig1
;CALL dummy        ;code only needed with selectable fonts
CALL XYCALC
call set_write
ld a,c
out (1),a
CALL FWDDO
RET              ;jump to exit via stacked value above
```

The first major change, text mode on the VDP is character based, so there is no need for a print character routine, once the screen address is calculated and sent to the address port, sending the character code to the data port is all that is needed to output the character.

Exiting via cursor forward and the previously stacked exit routine address is exactly as in CFX

```
.v_exit
;CALL xycalc
CALL setcur
POP  HL
POP  DE
POP  BC
POP  AF
RET
```

The NFX exit routine is the same as CFX, the original X,Y position calculation call isn't required on exit as the cursor routine will deal with that itself.

```
;       CONTROL CODE
.ctrl_code
LD   HL,ctab
LD   D,&00
LD   A,C
ADD  A,C
LD   E,A
ADD  HL,DE
LD   A,(HL)
```

```
INC  HL
LD   H,(HL)
LD   L,A
JP   (HL)
```

No changed to the control or escape code jump table access routines

```
.escdo
CALL frigit          ;redirect VDU stream
LD   A,C
CP   &20
JR   C,v_normal
AND  &1F
ADD  A,A
LD   HL,esctab
LD   E,A
LD   D,&00
ADD  HL,DE
LD   A,(HL)
INC  HL
LD   H,(HL)
LD   L,A
JP   (HL)

;cursor X,Y invalid values ignored
.cxydo
CALL frigit          ;redirect VDU stream
LD   A,C
SUB  &20
AND  &7F
CP   40              ;screen width
JR   NC,cxskip
LD   (xloc),A
.cxskip
CALL frigit          ;redirect VDU stream
LD   A,C
SUB  &20
AND  &7F
CP   24              ;screen rows
JR   NC,cyskip
LD   (yloc),A
.cyskip
;this section returnes the VDU stream to normal after
;assembling a multi byte command
.v_normal
LD   HL,crtgo
LD   (jp_vec+1),HL
RET
```

The cursor positioning code retains the 32 character offset in both the X and Y axis, but checks the Y position against 40 because of the narrower screen.

```
;set the VDU stream to code following the call to frigit, for
assembling multi byte commands
.frigit
POP  HL
LD   (jp_vec+1),HL
RET
```

The original rom used self-modifying code to build multibyte control codes, and that's retained unchanged.

```
;character background not setable, so code does nothing
.bkgset
CALL frigit          ;redirect VDU stream
LD   A,C
AND  &07
RLCA
RLCA
RLCA
LD   C,A
LD   A,(pratr)
AND  &C7
OR   C
LD   (pratr),A
LD   A,(npatr)
AND  &C7
OR   C
LD   (npatr),A
JP   v_normal

;no attributes on 9928 code does nothing
.atrset
CALL frigit          ;redirect VDU stream
LD   A,C
LD   (pratr),A
LD   (npatr),A
JP   v_normal
```

The attribute code runs un-altered but the attributes that it sets aren't actually used, as the VDP doesn't have the ability to do per character attributes.

```
; no bitmap character set on the 9928 setup so plot and draw
disabled
; swallow the character codes to maintain compatibility
.dotdo
CALL frigit          ;redirect VDU stream
LD   A,C
SUB  &20             ;&20 byte offset in plot commands
LD   (initl),A        ;save first perameter
CALL frigit          ;redirect VDU stream again
LD   A,C
;SUB  &20
LD   H,A             ;2nd parameter
LD   A,(initl)
```

```
LD   L,A
;CALL plotd
JP   v_normal          ;reset stream

.vctdo
CALL frigit
LD   A,C
SUB  &20
LD   (initl),A
CALL frigit
LD   A,C
SUB  &20
LD   (initl+1),A
CALL frigit
LD   A,C
SUB  &20
LD   (initl+2),A
CALL frigit
LD   A,C
SUB  &20
LD   C,A
;CALL plotv
JP   v_normal
```

The dot and line code builds the command bytes exactly as the 80 column driver did, however the final routine to do "something" with that data isn't implemented as the VDP doesn't support a 2nd character set in the way the 80 column board did.

```
.initlz_crt
LD   A,&FF
LD   HL,scrflg
LD   (HL),&FF
INC  HL
LD   (HL),&02
INC  HL
LD   (HL),&02
INC  HL
LD   (HL),&E0
CALL    CSSET
CALL    CRDO
CALL    LFDO
ret
;JP      ESTD
```

The screen setup routine is the same as CFX, compared to the FDX original it exits early without setting up the alternate character set, as that's not implemented.

```
;blink on and off not used, as no character level attributes

.blset
LD   A,(pratr)
```

```
OR   &40
LD   (pratr),A
RET

.bloff
LD   A,(pratr)
AND  &BF
LD   (pratr),A
RET
```

As with CF, the blink on/off, cursor on/off and scroll on/off routines are all implemented. However, the blink attribute isn't implemented as there is no hardware to support it. Nor is the cursor on/off supported by the software.

```
;no hardware cursor on 9928 in text mode
;turn cursor on value will be used as bit mask
.csset
ld a,&FF
ld (csr_flag),a
RET

.csoff
xor a
ld (csr_flag),a
RET

;scroll flag is 0 for page mode (no scroll)
.scrset
LD   A,&FF
LD   (scrflg),A
RET

.pgeset
XOR  A
LD   (scrflg),A
RET

; 9928 does not support individual charater attributes in text
mode
; so this code does nothing
.colset
LD   A,C
AND  &07
LD   C,A
LD   A,(pratr)
AND  &F8
OR   C
LD   (pratr),A
RET
```

Most of the character positioning control codes are as per the FDX original code, with the necessary changes to deal with the narrower screen. The order has been tweaked slightly to allow for "fall through" from one command to another where they're related.

```
; carriage return just sets the X location back to zero
; also has the dummy entry point for the nul codes
.crdo
XOR  A
LD   (xloc),A
.dummy
RET


; tab forward by setting the low bits of the x location then
; drop into cursor right
.tabdo
LD   A,(xloc)
OR   &07
LD   (xloc),A
.fwddo
LD   A,(xloc)
CP   39             ;screen width less 1
JR   Z,fwdl
INC  A
LD   (xloc),A
RET

; need to go down a line so reset x to zero and drop through
; into cursor down
.fwdl
XOR  A
LD   (xloc),A
.lfdo
LD   A,(yloc)
CP   &17           ; are we on the 24th row (rows are 0-23)
JR   Z,LFS         ;jump forward to test the scroll flag
INC  A
.pagem
LD   (yloc),A
RET

.lfs
LD   A,(scrflg)
OR   A
JR   Z,pagem      ;set Y back to the top of the screen if it's
                ; page mode
CALL scrup        ;otherwise scroll up, leaving Y as 23
RET
```

The 80 column board had hardware and its own speaker to create a buzzer type noise in response to the "bell" character code. CFX and NFX create their "bell" using the build in PSG, the rather nice bell from MTX basic isn't available, instead a short 440hz or "A4" tone is played over sound channel 1.

The 284 value for the note is calculated as the 4mhz CPU clock divide by 32 which is the PSG pre-scaler value. Which gives the theoretical maximum frequency of 125,000hz, dividing that by 440hz of "international A" is 284.1, since the PSG doesn't do fractional values, 284 is used.

```
; "DING"
;"beep" code from the Magrom
.beldo
push AF
push BC
push hl
ld HL, 284
LD A,L
AND &0F
OR &80
call s_sound        ; SEND TONE 1 + 4 BITS OF FREQUENCY
LD A,L
SRL A
SRL A
SRL A
SRL A
LD C,A
LD A,H
SLA A
SLA A
SLA A
SLA A
OR C
AND &3F             ; REMAINING 6 OF THE 10 BITS OF FREQUENCY
call s_sound
LD A,&90            ; ATTENUATION 0DB TONE 1
call s_sound
ld bc,00A0
.delay_loop
djnz delay_loop
dec c
jr nz delay_loop
;kill sound
LD A,&9f            ; ATTENUATION OFF TONE 1
call s_sound
pop hl
pop bc
pop AF
RET

;send data to sound port with delay to guarantee sound chip
;time to load it.
.S_sound
OUT (6),A
IN A,(3)
ld B,2
.ss_loop
djnz ss_loop
RET
```

The sound chip delay isn't the same as CFX, the looped version takes a few more bytes of code, but uses closer to the theoretical 32 cycles then the CFX version.

```
;non clearing backspace
.bsdo
LD   A,(xloc)
OR   A
JR   Z,bsu
DEC  A
LD   (xloc),A
RET

;backspace needs to go up a line
.bsu
LD   A,39          ;screen with less 1
LD   (xloc),A
LD   A,(yloc)
OR   A
JR   Z,bss
DEC  A
LD   (yloc),A
RET

;were already at 0,0 so need to put the x position back to 0
.bss
;   XOR  A    ; not needed A was 0 or we wouldn't be here?
LD   (xloc),A
RET
```

The NFX 40 column driver can uses the same cursor back and up code as CFX, with the one change required for the narrower screen width.

```
;cursor up
.updo
LD   A,(yloc)
OR   A
ret  Z           ;can't go up from top row
DEC  A
LD   (yloc),A
RET

.clrdo
CALL clrscn      ;clear the screen and fall through into home
                 ; to set the cursor
.hmedo
XOR  A
LD   (xloc),A
LD   (yloc),A
RET
```

Clear screen falls through into the cursor home routine in the same way that CFX does.

Erase line, by setting the x position to zero and calling erase to the end of the line is also unchanged.

```
.erln
LD   A,(xloc)
PUSH AF
XOR  A
LD   (xloc),A
CALL eoldo
POP  AF
LD   (xloc),A
RET

.eoldo
CALL xycalc
LD   A,(xloc)
LD   B,A
JP   erase_eol
```

Most of the “utilities” relate to unimplemented features, but the code is included for compatibility, just in case.

```
;       UTILITIES

.grpmap
LD   A,C
AND  &7F
LD   C,A
AND  &40
RET  Z
LD   A,C
AND  &20
RLCA
RLCA
OR   C
AND  &9F
LD   C,A
RET

.altmap
LD   A,C
OR   &80
LD   C,A
RET

.getmsk
LD   A,C
CP   &30
JR   NZ,getbit
XOR  A
RET

.getbit
DEC  A
```

```
AND  &07
LD   C,A
CALL ncalc
OR   A
RET
```

As with the CFX code, 2 separate routines need to be coded for positioning the VDP memory pointer to allow for the wat the VDP differentiates between read and write accesses.

```
; set the VRAM pointer to HL for reading and writing respectively
.set_read
push AF
ld a,L          ;setup VDP address
out (2),a
and &3f
ld a,h          ;bit 6 and 7 clear for VRAM read
out (2),a
pop AF
ret

.set_write
push AF
ld a,L          ;setup VDP address
out (2),a
ld a,h          ;set bit 6, bit 7 clear for VRAM write
add a,&40
out (2),A
pop AF
ret
```

The cursor on code, is identical to the cursor off code, so one routine is used for both the character at the “current” cursor position is read and the top bit inverted. The character map set up on boot has the first 128 characters inverted as the 2nd 128 characters, so that flipping the top bit inverts the character.

```
;cursor on/off needs to preserve registers as is called before
any VDU output
;uses inverted version of screen character,
.setcur
.killcur
push hl
push DE
push af
call xycalc
call set_read
in a,(1)
call set_write
xor &80         ; need to qualify this with the cursor
                ; on/off flag
out (1),a
pop af
pop DE
pop hl
RET
```

The cursor code could have used the cursor on/off flag, however for NFX's intended use that wasn't required and would only have slowed down the character output code.

```
; calculate the cursor position
.xycalc
LD   A,(xloc)
LD   D,A
LD   A,(yloc)
LD   E,A
.calc1
; enter here if DE already set
LD   A,E
ADD  A,A      ;y x 2
ADD  A,A      ;y x 4
ADD  A,E      ;y x 5
ADD  A,A      ;y x10
              ;max Y value 24, now move to 16 bit addition
LD   L,A
LD   H,&00
ADD  HL,HL    ;y x20
ADD  HL,HL    ;y x40
LD   E,D
LD   D,&00
ADD  HL,DE    ;y x40 + x
LD   A,H
AND  &07
LD   H,A
RET
```

Because NFX need the full 256 characters available to implement the cursor, it can't use the same screen mapping as MTX basic, as the screen map there overlaps where the top half of the character definitions would be. So, instead, the screen map is located at in the lowest part of the video ram. Which has a benefit of speeding up the cursor position calculations, as there is no offset required.

```
.ncalc
INC  C
LD   A,&01
.ncalcl
DEC  C
RET  Z
RLCA
JR   ncalcl
```

The code to convert a bit position value in C to a bit mask in A is as per the FDX original.

NewWord can be configured to work with the 40 column display, and using it in non-document mode would allow source code to be developed on NFX. So the escape codes that NewWord uses for scrolling the display have to be implemented.

The basic idea of the code is the same as CFX, however only 40 bytes need to be read for each line scrolled up or down instead of 256, which increases the scrolling speed considerably.

```
;delete line at cursor and scroll up
; set HL to point to the start of the line below
; set C to the number of lines remaining
; then call the scroll up code at it's looping point
.edcsln
ld a,(yloc)
cp &17           ;are we already on the last line ?
jp z blank_last
inc a
ld e,a
ld d,0
call calc1
ld a,e
CPL
ADD a,23
ld c,a
call scroll_loop     ;does the scroll and blanks the last line
jp v_normal
```

The scroll up code is the same as CFX, with the difference of the shorter lines. Both displays start at the beginning of the VRAM so there are no offsets to calculate

```
.scrup
ld HL,40         ;start with line 1
ld c,23          ;23 rows to scroll
.scroll_loop
call set_read
ld de,scroll_buff
ld b,40
.scroll_read
in a,(1)
ld (de),a
inc DE
djnz scroll_read
ld de,&ffD8      ;subtract 40 from HL to find the address to write
add hl,de
call set_write
ld de,scroll_buff
ld b,40
.scroll_write
ld a,(de)
out (1),a
inc DE
djnz scroll_write
ld de,80         ;add 2x line length
add hl,de        ;move HL on to the next line to be read in
dec c
jr nz,scroll_loop
jp blank_last
```

The clear screen code uses 2 loops with B and C as loop counters, the inner "B" loop inserts 192 spaces on the first pass, and 256 on the other 3 for a total of 960

```
.clrscn
ld hl, &0000
call set_write
ld BC,&c004        ;write 960 spaces (&3c0)
ld a,32
.CLRscLP
out (1),a
djnz clrsclp
dec c
JR NZ,clrsclp
RET
```

Erase to the end of the line in test mode is simpler, calculate the number of characters from position "B" to the end of the line, and then output that many spaces. In text mode the VDP can accept data as fast as the Z80 can send it so there is no need for any software delays, a simple loop with a space being sent to the data port is all that's needed.

```
; erase to the end of the line from character B
;xycalc has been called so HL holds the screen position
.erase_eol
LD   A,40
sub b
LD   B,A
call set_write
ld a, 32
.ereol_lp
out (1),a
djnz  ereol_lp
ret
```

The escape code table is the same as for the CFX driver, it's basically the same as the original, but all the alternate character set codes are converted to null and exit without doing anything.

```
;       ESCAPE SEQUENCE LOOK-UP TABLE

.ESCTAB
DW      v_normal  ; @
DW      v_normal  ; DW  EALT    ; A set alternate character font
DW      EBOTH   ; B set both attribute bytes
DW      ESCRL   ; C set scroll mode
DW      EPGE    ; D set page mode
DW      ECSON   ; E cursor on
DW      ECSOFF  ; F cursor off
DW      v_normal  ; DW  EGRPH   ; G set graphic font
DW      v_normal  ; H
DW      EIBLLN  ; I insert blank line at cursor move the other
                ; lines down
DW      EDCSLN  ; J delete line at cursor move the other lines up
DW      v_normal  ; K
```

```
DW      v_normal  ; L
DW      v_normal  ; M
DW      ENPATR  ; N set non printing attribute
DW      v_normal  ; O
DW      EPRATR  ; P set printing attribute
DW      v_normal  ; Q      ; in SCPM ROM, disable color, cls
DW      EREAD   ; R      ; in SCPM ROM, enable color, cls
DW      v_normal  ; DW  ESTD    ; S set standard font
DW      ESIPR   ; T
DW      ESINP   ; U
DW      ESIBT   ; V
DW      EWRMS   ; W
DW      CNTSIM  ; X
DW      v_normal  ; Y
DW      v_normal  ; Z
DW      v_normal  ; [
DW      v_normal  ; \
DW      v_normal  ; ]
DW      v_normal  ; ^
DW      v_normal  ; _
```

Alternate character set isn't supported, so with the vectors removed in the table above, the code is commented out.

```
;       ESCAPE SEQUENCE HANDLERS
;only one character set provided in text 2 mode
;.ESTD
;LD   HL,DUMMY
;LD   (frig1+1),HL
;JP   v_normal

;.ealt
;LD   HL,altmap
;LD   (frig1+1),HL
;JP   v_normal

;.egrph
;LD   HL,grpmap
;LD   (frig1+1),HL
;JP   v_normal
;
```

The code to simulate control codes with escape codes is unchanged from the FDX.

```
;simulate control character
.cntsim
CALL frigit
LD   A,C
AND  &1F
LD   C,A
CALL v_normal
JP   ctrl_code

.escrl
```

```
CALL scrset
JP   v_normal

.epge
CALL pgeset
JP   v_normal

.ecson
CALL csset
JP   v_normal

.ecsoff
CALL csoff
JP   v_normal

; set attributes directly, rather than by bit
.esipr
CALL frigit
LD   A,C
LD   (pratr),A
JP   v_normal

.esinp
CALL frigit
LD   A,C
LD   (npatr),A
JP   v_normal

.esibt
CALL frigit
LD   A,C
LD   (npatr),A
LD   (pratr),A
JP   v_normal
```

Blanking the last line of the display is use by some of the other routines so is coded separately, as with the erase to the end of the line code, there is no need for any delays, 40 spaces can be sent in a simple loop.

```
.blank_last
ld HL,920        ; = the start of the last screen row
.blank_current
call set_write
ld a,32
ld b,40
.blanks_loop
out (1),a
djnz blanks_loop
jp v_normal
```

This is another scroll routine needed by NewWord, other than the changes required for the 40 character lines, it's the same code as CFX uses.

```
;insert line at cursor and scroll down
.eiblln
ld a,(yloc)
cp &17
jr z blank_last    ;on the last row, just blank it
CPL                ;ones complement so A=-(row)-1
add a,24           ;add 24, as we want one less row - to
                     ;allow for inserting
ld c,a             ;rows to scroll
ld HL,880          ;working fom the bottom up, row 22 (to row 23)
                   ;is always the first to be moved
.scroll_d_loop
call set_read
ld de,scroll_buff
ld b,40
.scroll_d_read
in a,(1)
ld (de),a
inc de
DJNZ scroll_d_read
ld de,40            ;move down to the start of the next row
add hl,DE
call set_write
ld de,scroll_buff
ld b,40
.scroll_d_write
ld a,(de)
out (1),a
inc DE
djnz scroll_d_write
ld de,&FFB0       ;-80 to move up 2 rows
add HL,DE
dec C
jr nz scroll_d_loop
ld de,40         ;move back to the "current" line
add hl,de
; now insert the blank line,
JP  blank_current
```

Unlike CFX's graphics mode display, the character at the cursor can be read in text mode. A dummy attribute is returned, but the character code is "real". Since this code would be called while the cursor itself is off, the true character is returned, and not the inverted version.

```
;read character at cursor ??
.eread
CALL xycalc
call set_read
ld   a,2            ;set an attribute just in case
```

```
ld   (v_attr),A
IN   A,(1)
LD   (ascr),A
JP   v_normal
```

The various write mask and attribute routines from the 80 column display are fully implemented, but do nothing as the attributes aren't implemented.

```
; setup write mask
.ewrms
CALL frigit
LD   A,C
LD   C,&E0
CP   &30
JR   Z,swrm
LD   C,&C0
CP   &31
JR   Z,swrm
LD   C,&A0
CP   &32
JP   NZ,v_normal
.swrm
LD   A,C
LD   (wrmsk),A
JP   v_normal

;set printing attribute
.epratr
CALL frigit
CALL getmsk
JR   Z,setpr
LD   C,A
LD   A,(pratr)
OR   C
.setpr
LD   (pratr),A
JP   v_normal

; set non printing attribute
.enpatr
CALL frigit
CALL getmsk
JR   Z,setnp
LD   C,A
.enpa1
LD   A,(npatr)
OR   C
.setnp
LD   (npatr),A
JP   v_normal

;set both printing and non prining attributes. Does nothing.
```

```
.eboth
CALL frigit
CALL getmsk
JR   NZ,v_setb
LD   (pratr),A
JR   setnp

.v_setb
LD   C,A
LD   A,(pratr)
OR   C
LD   (pratr),A
JR   enpa1
```

Finally, like CFX, the VDP driver's stack is implemented as per the FDX original.

```
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
DB 0,0
.vstk


END
```

The initial setup for the VDU driver is actually done in the boot rom and that code isn't copied to high memory, as it's only needed once. However, it isn't run until after the code is copied high, so all the routines in high memory can be called. That saves some assembler gymnastics that would be required to call the code in the rom instead.

The code is reasonably well commented so doesn't need massive explanation. The first step is to move the character map to the beginning of the video memory, that's done by setting VDP register 2 to zero.

The character data that was setup during boot is copied out into a convenient location, in this case that's #C000, but could have been anywhere between #4000 and #DFFF. On the way from video memory to main ram the data is inverted using the CPL instruction.

The VDP memory pointer is then set up to point to character code 160, and the inverted data placed into VDP memory to act as the inverted characters for the cursor.

The final step is personal choice, I didn't like using CPM with the default white on mid blue MTX colours. So register 7 is set to dark yellow on black, which is as close to a classic "amber" mono display as I could get using the colours available in the VDP.

```
;VDU setup step 1, move the name table (register 2) to offset 0000
ld a,0
out (2),A
ld a,&82
out (2),a
;step 2 extend the patern table from 1900-1BFF to 1FFF
;because the rom is copied we can call high memory routines
ld hl,&1900
call set_read
ld hl,&c000
ld e,3
ld BC,1          ; b=0  c=1
.read_loop
in a,(c)
cpl              ;invert the data, top bit characters are inverted and
                 ; used for the cursor
ld (hl),A
inc hl
djnz read_loop
dec E
jr nz,read_loop

ld hl,&1D00
call set_write
ld hl,&c000
ld e,3
ld BC,1          ; b=0  c=1
.write_loop
ld a,(hl)
nop
out (c),A
inc hl
djnz write_loop
dec E
jr nz,write_loop

;step 3 change the colours by updating register 7 - CPM looks really
;odd white on blue
ld a,&A1
out (2),A
ld a,&87
out (2),a

JP   setup
```