**Working draft, mostly as I remember things so apologies for the order..**

**contact will-i-was@verginmedia.com - for any help/enquiries or via FB.**

**EJC80 MKI Z80 based single board computer.**

**First and most important, warnings..**

1) The greatest care MUST be taken if using this board in conjunction with a 'RASPBERRY PI' If the connecting cable is misplaced at either end it is almost certain that either the PI or the SBC will be fatally wounded. PLEASE double check prior to applying power. The Pi supplies 5V and 3V to the SBC (3v for the serial interface, 5v for the rest of the circuits.

2) When used with a PI all power for the card is taken from the PI, please ensure your PI can supply around 350mA for a fully configured SBC (this figure includes a cfcard adaptor loaded with a 16GB cfcard. IF using a mechanical hard drive or SSD drive you may need to provide separate power to these device(s)

3) The Raspberry PI and all software therein is not part of this package, however a sample set of [modified] PIGFX software is provided in order to get things going and demonstrate using the PI as a graphics adaptor. The design however, allows for terminal operation if a 3v serial to RS232 adaptor is wired in place of the PI - Note however in this case the SBC can only provide a 5v connection to any such adaptor and, the SBC should then be powered from the 5V SBC Power Jack.

4) When powering the SBC via the Power Jack (J)K1, please note the CENTRE PIN IS NEGATIVE. This is opposite to normal practice and due to a senior moment...

5) Any SBC software/Firmware or other programs supplied are examples only, no warranty for these is expressed or implied, from time to time updates may be available but again these are free and without warranty.

6) Please observe the usual antistatic precautions if building this yourself! - And during operation, storage and transport! - Please note other warnings may in place further in this document Please read all!

**A little history.**

The designer of this system is a former IBM design engineer now using his free time to play with electronics and software, fast approaching 70 years old if things go quiet..... Looking for something to do I have finally got around to designing my own microcomputer for fun, as a hobby, pastime etc. After some six months the project has resulted in a final PCB and design. Getting the card etched was a little expensive than I had hoped so a deal was negotiated where 12 cards would be supplied (bulk of cost is always the setup). First design incorporated two 8255s (for fun) but these were dropped and the layout altered to allow greater flexibility, all of the SBC firmware was developed and extensively tested on the 'A Model' then updated to work on the final etch. Much head scratching (of what little hair is left) for graphics, various alternatives were looked at but eventually a PI was chosen for cost reasons, a short while after starting to write the code I was told of PIGFX freeware, after evaluating this it was decided to re-write some of its features to suit better this project. I grabbed a copy of Tiny Basic which had been transcribed and Commented by Douglas Gabbard III and adapted it to run on the prototype to test the PI Graphics. - It is not intended to modify this to run on the final SBC. IN order to keep the size small and flexibility high a simple serial comms was implemented in software.

Enough, now the nitty-gritty!

**Hardware Brief Description**

**Features:**

Z80 Based SBC -Clock 8-16.257MHz.

64Kx8BIt rom

128Kx8bit RAM

Simple 'Beeper Circuit'

Serial Comms

CFCARD / PATA -IDE Interface

LED (indicates RAM/ROM mode)

Single Bit Data out User definaable.

**Connectors:**

JK1 - 5 Volt Power Supplies power to the board where a Raspberry pi is not being used. CAUTION!! centerpin is NEGATIVE.

J1 - External reset SW connector - Make to reset, release to run.

J2 - Comms interrupt, when made incoming comms is applied to the Z80 INT pin, in use this means an interrupt routine should imediately Disable INterrupts then get the character from the data port.

J3 - ROM Page select, the upper 4 address bits of the eprom may be set high (no link) or grounded (link in place) to select 1 of 16 4k EPROM pages. - N.B Card may be supplied with page 15 used for the Boot ROM, (no links fitted) or page 0 (4 links fitted).

J4 - IDE Connector, 16bit IDE interface for CFCARD or SSD/HARD DRIVE.

J5 - Audio out - simple speaker or buzzer connector. Circuit is short circuit protected.

J6 - When made this connects 5 volts to PIn 20 of the IDE/CFcard Connector (Historically this was the IDE keyway, ensure the link is ONLY fitted for CF cards that adopt this method of powering. For other types of CF adaptor power can be taken from J6 pin 1 (found next to C3).

J7 - CPU Parallel connector, 40 pin connector connected directly to it's (physically) corresponding CPU PIN .

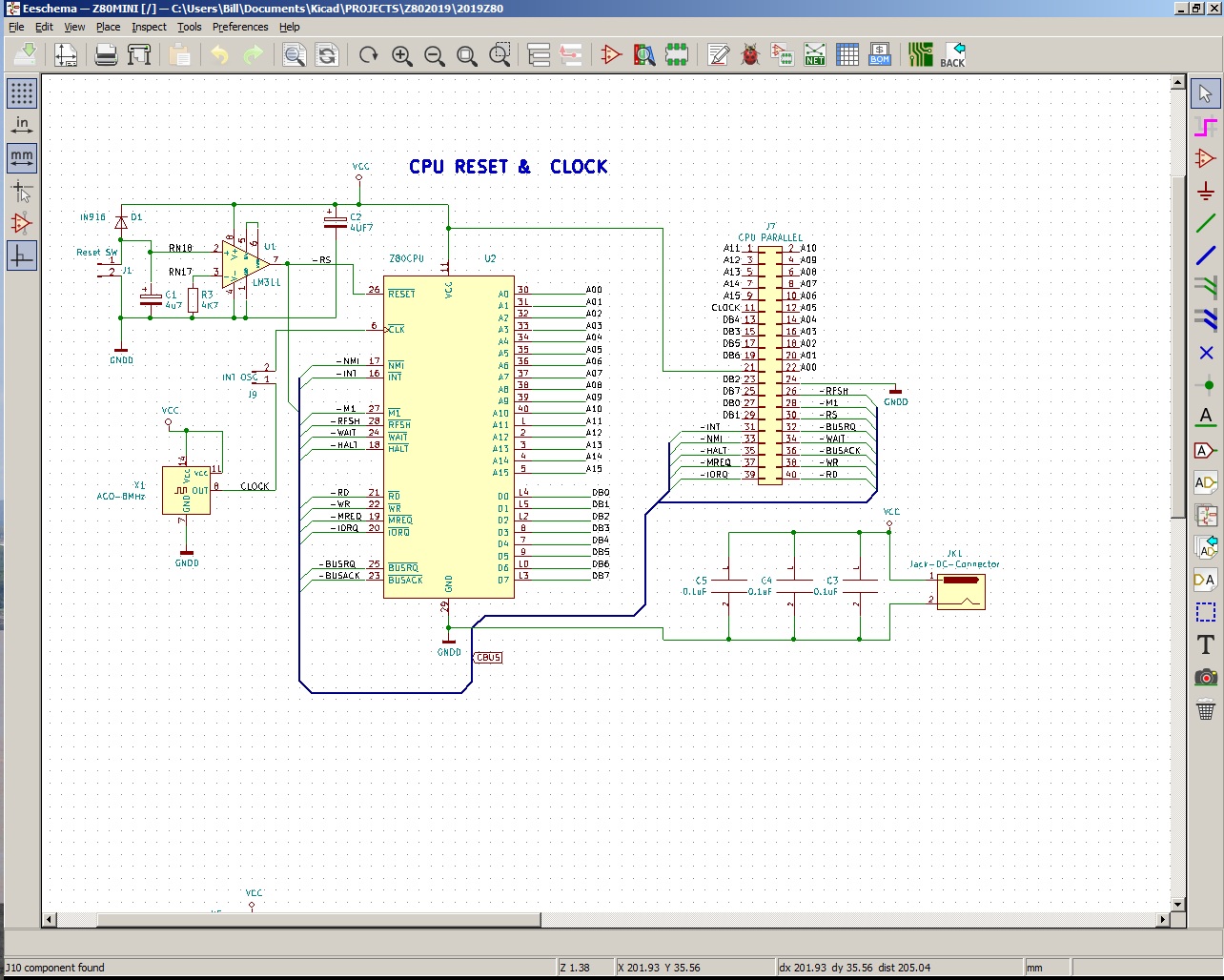
J8 - 'PI connector, pin for pin same as [pins 1-10] on the PI, thus a simple IDC 10 pin connector wired '1 for 1' is required. (two pins on the pi board may need to be removed to allow the connector to be fitted.

J9 - INTERNAL oscillator, when the link is made the CPU clock is direct from the TTL CLOCK OSC [also fed to J7 for external use. When not connected, an external clock may be applied to the appropriate pin on J7

J10 - This is a single IO bit [out only] plus 5V and Gnd for user defined use.

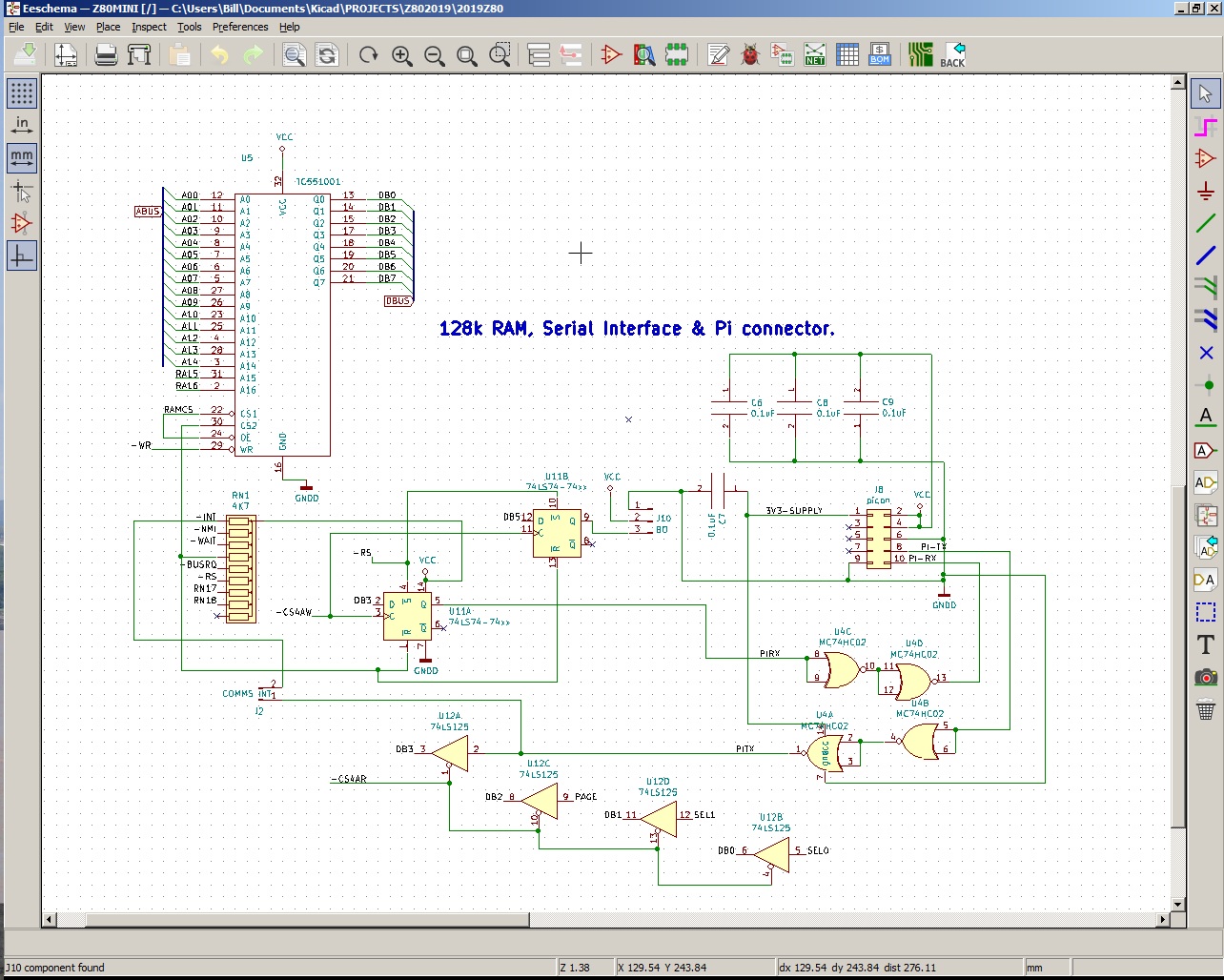
**Partial Tech Description:**

**RESET:**

****

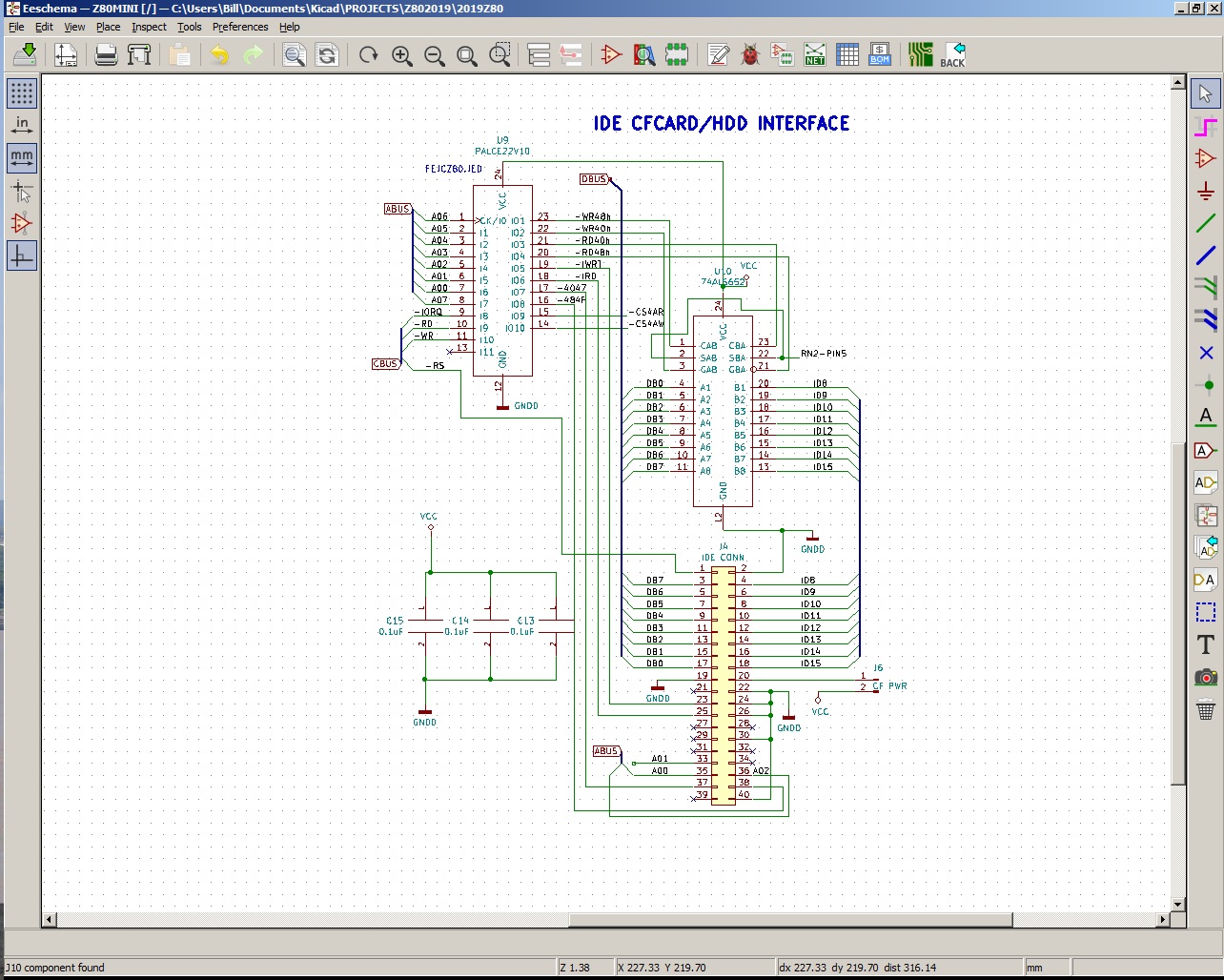
At power up the reset line is initially held low as C1 charges via a section of RN1 (resistor network) - it's voltage is sensed by the comparator U1 pin 2. As this voltage passes the 1/2 VCC supplied to U1 Pin 3 the LM311 output switches from logic 0 to 1. The -reset pulse is supplied to the CPU, the CFCARD/IDE interface and ALL 7474 sections to set the SBC into a fixed state prior to CPU reset release. The initial state is as follows: Serial output LOW (this will immediately be set high by the software), Lowest 4k of memory is set to ROM (via the 'PAGE' line to the ZCTRL GAL chip) . Speaker/ buzzer Line Set HIGH, RAM Segment set to 0 via lines 'SEL0 & SEL1' The lower RAM segment address 0000 08000h is always set to the same address and is neither switchable [other than the lowest 4k] or moveable. the remaining 96K of ram is split into three 32K pages, any of which my be used as the upper 32K of system memory.

**Serial Comms:**

****

The output is latched in U11A then fed to a 2 stage buffer U4C which is supplied by 3.3v from the PI. IF a pi is not used be sure to connect the supply of this chip to 5Volt U11 is sufficiently tolerant to operate at 5v input signals whilst it's supply is only 3.3v - N.B this is outside the manufacturers spec yet, after persistent testing (running for several days) no adverse effects were noted, You connect this circuit to a PI at your own risk. Serial input is again buffered and then passed to U12A, the output of which can be directly read by the CPU. The input of U12A is also routed to J2 where it may be used to raise an interrupt upon being driven low.

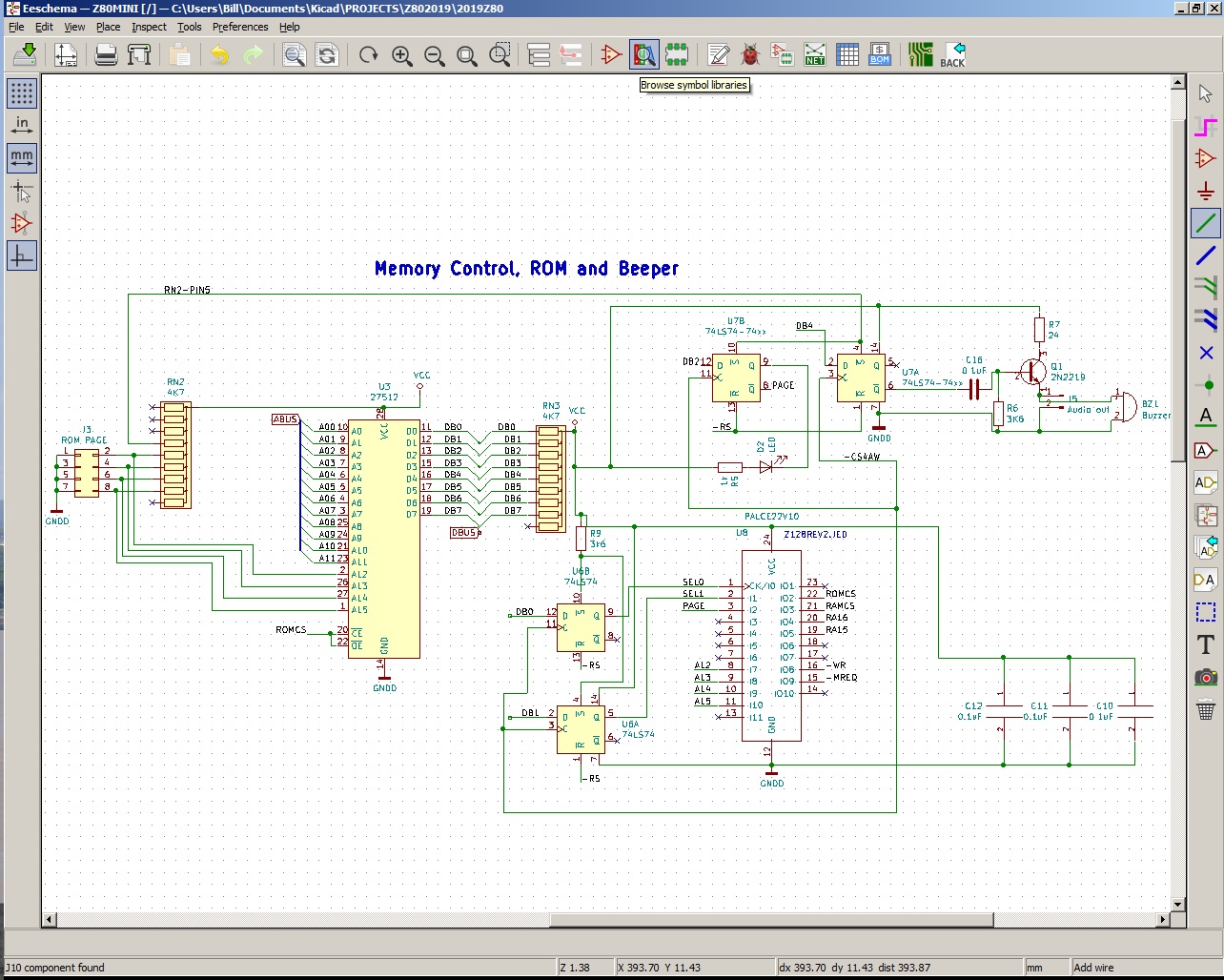
**CFCARD IDE Interface.**

****

After much design refinement this interface has been optimised by using only two narrow 24 pin chips, a Programmable address & function controller and a single 74ALS652 Bidirectional Buffer. This circuit requires no setting up, or special commands, it appears as a standard IDE register for the primary register set (IBM PCAT 170h-177H) moved to 40-47h, 48H is used for the upper 8 databits (D8-D15) and the original Alt register set (originally 376-377h?) moved to 4eh-4fH . Essentially to send the upper 8 bits to the IDE one writes the data to address 48h, the lower data bits are then written to 40h. The action of writing to 40H also clocks any latched data in the 652 onto the upper data lines, thus the sequence is Write upper byte, write lower byte. Read is essentially similar but here we first read the LOWER byte, thus also clocks the high data into the 'reverse' latches within the 652. After reading the lower byte, simply perform a read to 48h to recover the high byte.

Since the 'ALT' Register set uses only addresses 4e & 4f, plus the high data byte using 48H it was decided to use IO address 4Ah for memory control, speaker, LED & Comms.

**Memory Control**



Memory control is achieved though the software. 3 bits of data (representing 8 possible, though only 6 useable memory states) are fed to the address decoder chip U8, these are SEL0&1 and PAGE.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| SEL0 | SEL1 | PAGE | 0000-0FFFh | 1000-7FFFH | Ram A15 | Ram A16 |
| 0 | 0 | 0 | RAM | RAM | 1 | 0 |
| 1 | 0 | 0 | RAM | RAM | 0 | 1 |
| 0 | 1 | 0 | RAM | RAM | 1 | 1 |
| 1 | 1 | 0 | RAM | RAM | -- | -- |
| 0 | 0 | 1 | EPROM | RAM | 1 | 0 |
| 1 | 0 | 1 | EPROM | RAM | 0 | 1 |
| 0 | 1 | 1 | EPROM | RAM | 1 | 1 |
| 1 | 1 | 1 | EPROM | RAM | -- | -- |

It can be seen from the above that the 128K ram chip is used as 4x32k segments (0-3) Segment 0 is always addressed from 0000 (or 1000h) to 7fffH, with the upper three 32k blocks used from 8000h to ffffh as addressed by SEL0 & SEL1. Writing 1's to both select bits has no reliable or documented function. The ability to erplace the 4K rom with ram via a simple software command affords the ability to use CP/M

**TESTFIRMWARE:**

After reset is released the test firmware behaves as follows.

Interrupt mode 1, disable interrupts.

System writes 18h to the Memory and sundry control IO address (4Ah) This Sets the serial port output to '1' (idle) and speaker bit LOW, ram slice 1 at 8000h-ffffh. And keeps ROM selected as the low 4k.

The software then checks a small portion of ram (located at 1000h for use as a temporary stack, A memory check of all other ram from 1020H to 1FFFFH (ie the full 128K) is performed, If it fails execution stops after writing E8h to the MEMCON register.

If memory testing is passed then a copy of the BOOT ROM is loaded to address 0F000h, execution then continues at $+f000h where a memory switch is performed leaving RAM at address 0-0fffh, the rom code is then copied back to 0 with execution continuing at $-f000h. After this a BOOT is performed, system is tested for a hard drive/cfcard, if found the string FAT32 is searched for, and from than the root directory is located and it's parameters loaded to RAM. IF no disk (or no FAT32 signature is found) the systems drops to Z:> prompt, otherwise R:> (system is in ROM mode) or C:>(system is running in RAM) is displayed.

Time for a coffee....

Ah that's better, now where were we? Oh yes, Firmware description...

Available functions after powerup:

N.B. Most commands can be edited thus if writing RESDM instead of READM it can be corrected PRIOR to pressing CR. For addresses and data however if you make a mistake it is actioned thus if you write 34h to address 2000 but intended to write CBh (or whatever) you will need to repeat the whole operation.

DIR - If a formatted FAT32 disk is present the system should display the contents of the root directory (or at least the first part) So far in order to ease programming such disks are formatted to 64K cluster sizes with a single cluster allocated for each file regardless of actual file size. With today's high capacity drives, this is more than enough for our humble Z80! A 1Gig drive would allow around 16000 files and so on. This firmware is evolving, although it works it has limitations (many yet to be discovered where disk operations are performed! Each file is given a Hex index number. IF there are deleted files then these are NOT displayed but still numbered.

LOAD - Load a file, input the HEX number and the file will be loaded from address 11f0h - upwards.

SAVE - THis will save a file provided it was loaded from the disk and the stored DISK data has NOT been corrupted, it will put the file back to the same location it loaded it from. IF the disk data in ram has been corrupted the system will almost certainly write it 'somewhere' on the disk. Remember this is an experimental system it's for the Geek, the Curious, the Eccentric and NOT a Christmas present for little Johnny. A level of Common sense and care are key requisites!.

Best advice back up your IDE files Elsewhere!

WRITEM (CR) - this will prompt for a write address, give the address then start typing the data, NO spaces, and NO CRs - thus you would type WRITEM (CR) then [e.g] 30001122334455[ESC] which when read back will show:

3000 11 22 33 44 55

READM - READM(CR) will prompt for an address and upon the 4th digit entered it will go fetch the data - it will display 16 lines of 16 bytes (+ ascii) then prompt for more when you should respond with + or - to read next/previous 256 byte block. Any other key [should] return you to the previous system prompt.

RDIOB - Read a single byte of IO After CR it will prompt for Address, and then immediately display the read data.

RDIOM - Read IO MAP - this reads the IO map, and places the data at location FF00-FFFF (so watch out) it then displays (by auto performing a read operation) the results. 256 reads are performed.

WRIOB (CR) prompts for IO address (2 byte) and data.

RUN Prompts for a 16bit address and immediately transfers control to that address, to run a loaded file 1200 is chosen although files are loaded from 11f0h up, the first 16 bytes are reserved for some reason which I've no doubt will return at some future date.

FILLRAM - Prompts for from, to and with (single byte only) to fill a block of ram.

CLRRAM - Clears RAM with from to addresses prompted for.

MOVMEM - ( Should be copy, But I reserved the word copy foe possible Disk ops) Prompts for Source, Destination and Size, then copies as directed.

HDDPARMS - Reads the HDD Parameter block (512 bytes) and writes a copy to EE00H

BOOT - Outwardly this simply reloads (or attempts to reload) root directory data and information from any IDE attached device.

DISK - performs a low level disk operation. Prompts to fill the IDE register set, and then executes. USe with exceptional Caution!

RAMSW - intended to switch between RAM and ROM for the first 4K of Memory Caution required! - Somewhere down the line it's action has been corrupted! - use a direct write to the relevant bits at IO 4Ah

RSTHDD - Intended to issue a software reset to the HDD. - Use with caution!

**PI executed Commands:**

FORCOL - set text Color, Upper 4 bits Foreground, Lower 4 Backgrnd. I.E 'B4' = High intensity FG, with BLUE BG. Works ONLY if connected to PI with correct level Software installed.

HOME - Send CSR to home position

CLRSCR - Clear screen only.

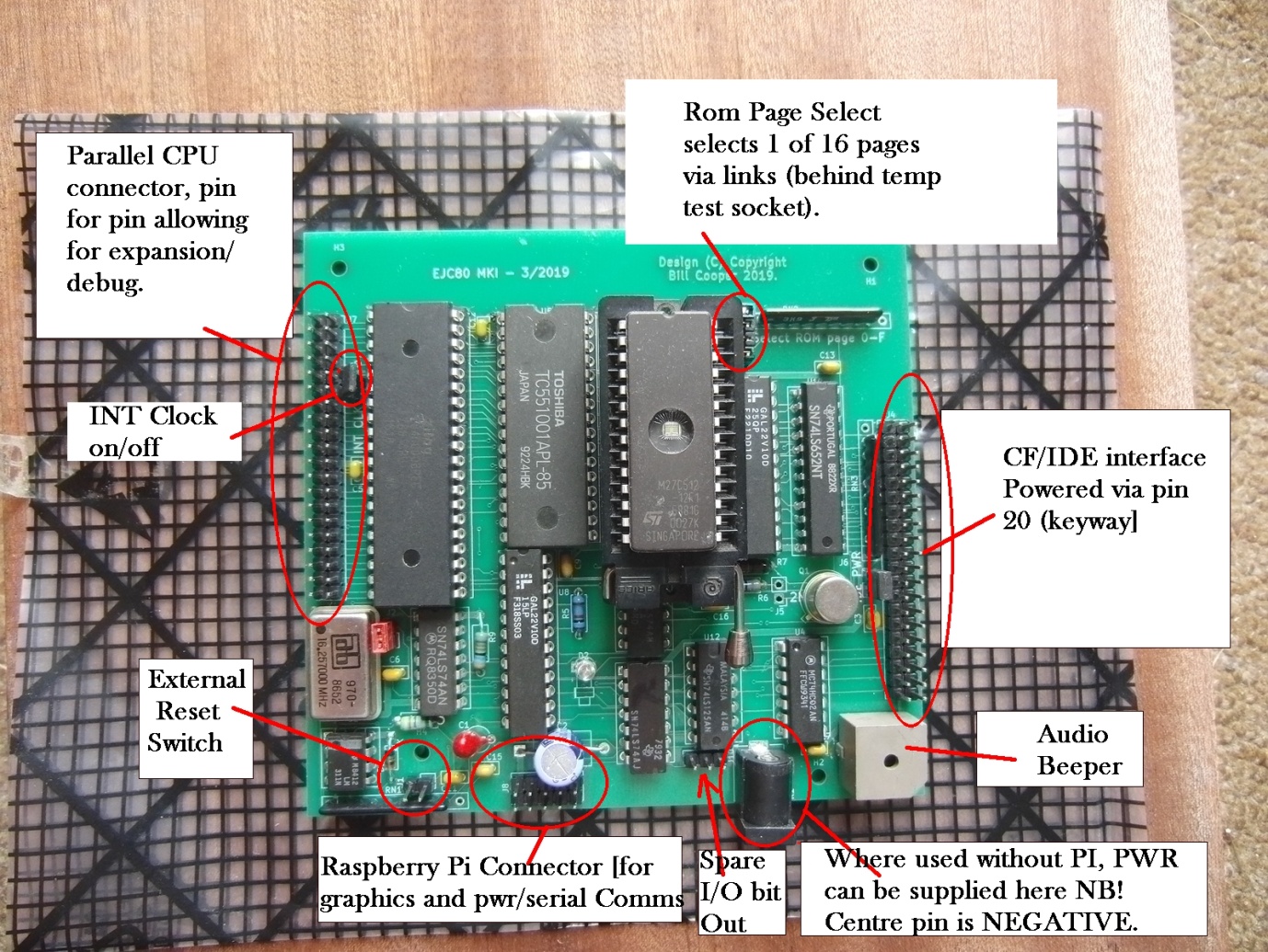
CLRHOM - Clear screen and place cursor at home position.

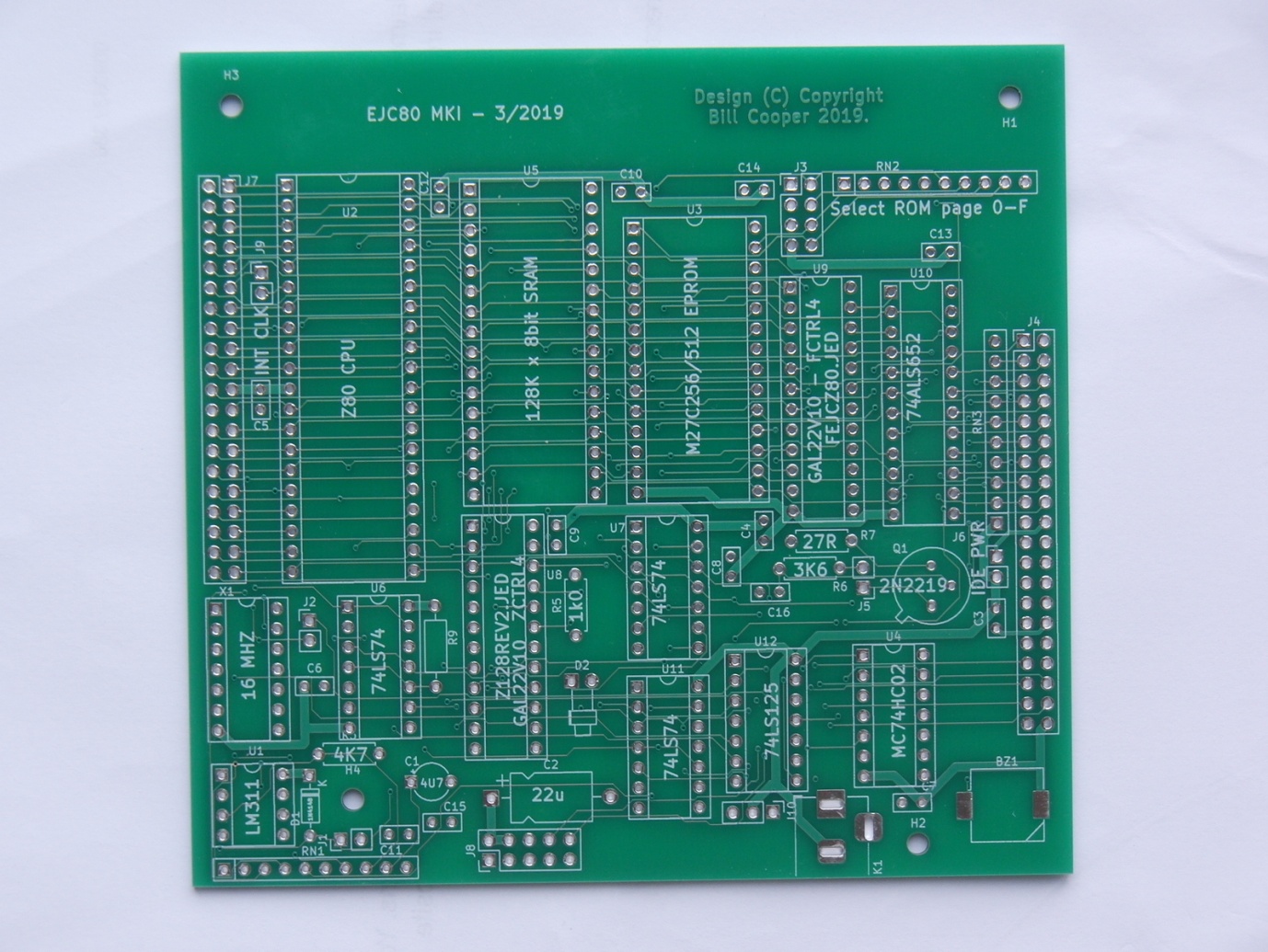
SETBP - Set a breakpoint, this places a breakpoint at selected location then stores the original byte for later replacement

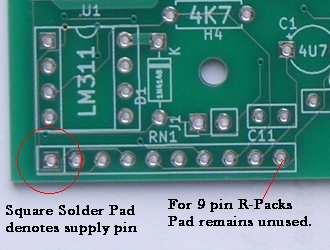
RESBP - replaces BP code with original

Upon execution of a breakpoint data is displayed onscreen of various registers,

These are brief descriptions, you will need to play and experiment if you wish to use them or, as is intended write your own EPROM program. (maybe move this one to a safe EPROM page?.





Bareboard, The three Resistor network positions are designed to take either 9 or 10 pin SIL networks where the supply pin is at one end, (usually marked with a dot) 

A very rough Bill Of Materials, Use in conjunction with the Circuit and Board Photos.

BZ1 A buzzer, or better still use a small speaker. The prototype was fitted with a reclaimed unmarked buzzer, hence the inclusion of a two pin speaker connector.

C1 & C2 4uF7 16Volt

C3 - C16 0.1uF decoupling, This design only really needs about half of these BUT make sure C16 is one of them - it is in the 'audio' path.

D1 - IN4148 signal, designed to drain the reset cap when the system is switched off.

D2 - common or garden Indicator LED any colour except black.

J1 - J10 are standard 0.1pitch male connector Unshelled pins.

JK1 - RCA Power jack - I think it's called.

Q1 2N2219 - or any similar GP NPN Silicon 250mA IC max should be fine.

R3 4K7

R5 1K0

R6 3K9

R7 24 ohms

R9 3K6

N.B apart from R7, the above resistors may all be 3k3-4k7, these will all work fine.

RN1/2/3 Resistor networks, (8 or 9 section 0.1" SIL) - can any value from 2k7 to 4k7.

U1 LM311

U2 Z80 (16Mhz Clock capable) or 8Mhz with software change.

U3 27C256/512 Eprom with Firmware loaded.

U4 MC74HC02

U5 TC551001 128K x 8Bit RAM or equivalent.

U6/U7/U11 74LS74

U8 & U9 GAL22V10 Programmable logic arrays with programms.

U10 74ALS652 BI-Directional Latch.

U12 74LS125

X1 Xtal Oscillator, 16Mhz/16.257 - Or any other with suitable software mod.

; See FCTRL02.PLD for PAL source.

; I/O Map - revised 19/05/19 -

; shifted 00, 08, 10-1f functions to 4A

; 40-4F Cfcard/IDE interface Control

; 40h IDE Controller DATA bits 0-7

; 41h " ERROR/FEATURES registers.

; 42h " SECTOR COUNT register.

; 43h " SECTOR NUMBER register.

; 44h " CYLINDER LOW register.

; 45h " CYLINDER HIGH register.

; 46h " DRIVE HEAD register.

; 47h " STATUS/COMMAND register.

; 48h " DATA bits 8-15.

; 49h " unused

; 4Ah " SEE BELOW.

; 4Bh " N/U

; 4Ch " "

; 4Dh " "

; 4Eh " ALTERNATE/Drive CTRL register.

; 4Fh " DRIVE ADDRESS register.

; 4AH BEEPER,SERIAL & MEMCONTROL REG.

; bit R/W function.

; 0 r/w [SEL0] Select ram page

; 1 r/w [SEL1] Select ram page

; 2 r/w [PAGE] select rom(1) / ram (0) ;NB When reading this location the data is inverted thus to set the SBC to ROM, bit 2 is set to 0 setting the PAGE line to a Logic 1 when reading back, the PAGE line is read directly thus upon readback the result be a logic '1'.

; 3 r/w [Serial comms]

; 4 wo [Beeper]

; 5 wo [j10 bit] GP OUT bit.

; 6 unused

; 7 unused

The PI. Using the PI as a graphics card is still under development.

Appetite wetter.

Example of PI graphics receiving commands from the Z80.



The above photo is of a simple basic program running under Tiny Basic, - purely for the purpose of assessing speed (Graphics mode 640 x 480). The interpreter is loaded, the above list is typed in and the program run. The tiny Basic used was modified to allow testing of the PI graphics only the basic commands necessary to run the program were modified to run on the SBC. The basic interpreter used is supplied with source code such that others may undertake improvements. (E.G print "Hello World is valid but print "Hello World" will error (due to the second set of quotes) - A printout of the basic assembly language listing will easily show the extra commands in use. Since this software was only intended to test the graphics interface it is supplied as a sample without instructions. Itis known that by 'butchering' 'TIny' in order to prove the SBC graphics, many other TIny commands may have been adversely affected. It was never the intention of the author to produce a working Basic compiler for this project.

Brief list of Tiny basic commands added to check the SBC commands;

USR,PEEK,POKE,PIXEL,CLRSCR,SIN,RECT,LINE,COLOR.

USR nnnnn Transfer control to address nnnnn, where nnnnn is Decimal address thus USR 4608 will transfer control to Hex address 1200h.

PEEK nnnnn The value at decimal nnnnn memory location is placed into variable 'Y' thus PEEK 0 will place the decimal value 24 into Y, the next line may then say PRINT Y which will show 24. (signed address)

POKE nnnnn ddd Poke decimal address nnnnn with decimal data ddd Thus POKE 32767 192 will place 0C0h into location 7FFFH N.B Address is 'signed' thus address -1 will POKE/PEEK 0FFFFh

PIXEL xxx yyy where xxx is 0-639 & yyy is 0-479 will set the addressed pixel to the current forecolor.

CLRSCR will clear the screen and place the cursor at the home position.

SIN nnnnn where nnnnn is -32767 to +32767 (degrees NOT radians) will place a sin value into variable 'Z' N.B. this is acheived by a lookup table of values from 0-90 degrees, >90 degrees and <0 degrees are calculated from these values, the values placed in Var 'Z' vary from 0(0 degrees) to 32000 (90 degrees).

RECT x1 y1 x2 y2 THIS will draw a FILLED rectangle of the current forcolor, where x is 0-639 and y is 0-479.

LINE x1 y1 x2 y2 This will draw a line from x1 y1 to x2 y2

COLOR nnn (0-255) N.B Upper 4 bits (4-7, i.e. multiples of 16) are used for the foreground color, with the lower 4 bits (0-3) used for the background color. BIts 0&4(RED), 1&5(GREEN), 2&6(BLUE), 3&7(HIGH)

**LOW LEVEL PI COMMANDS:**

PI Graphics commands, NOTE these are modified from the original PIGFIX files. (Search GITHUB) for original files. The original PIGFIX was intended (as I understand) to purely simulate a Terminal Application, the graphics offered required ascii strings with ascii digits in order to draw a line or a rectangle. For the Z80 it was decided that this was impractical (it can still be used though by downloading the original PIGFIX suite) reason being if one wanted to draw a series of lines, each one had to be converted to ascii and sent, sending in Binary form however proved faster as the Z80 did not need to create the ASCII and the PI does not need to convert it back. So it's binary to binary, rather than Binary to ASCII - ASCII to Binary! THus to make the Z80 programming easier and faster things were changed. The modified software is offered free as a sample program to demonstrate the use of the SBC and PI working together.

notes (fill out and check for accuracy. later).

Command List [No of Parameter bytes, correct number of bytes MUST be sent).

'B0' [2] Writes a byte to the PI gpio bits 0-7 (Unfinished code).

'C0' [2] Set text Fore/Back Parm(0) 20h + Forecolor(0-0fh), Parm(1) 20h + BKCOL(0-f)

'C1' [2] Set foreground Color (0-255) Parm(0) 20H+ High Nibble/16, Parm(1) 20H+Low Nibble

'C2' [2] Set Background Colour (0-255) as for 'C1'

N.B 'C0' fully written, tested - and in use. C1&C2 written but UNTESTED. TODO!!!.

'D0' [6] Set Pixel X,Y X=12 bits, Y=12 bits thus in total 8 bytes

'D1' [12] Draw line From x,y to x,y each parm again as 3 bytes Total string = 14 bytes.

'D2' [12] Draw rectangle Top left x,y to bottom Right x,y again 12 bytes.

'E0' [0] Clear screen (no parms)

'E1' [0] Cursor Home (No parms)

'E2' [0] Clear screen and place cursor home'

'E3' [??] set cursor position - **UNWRITTEN**

All commands are sent in a 'frame' of various length.

The File gfx.c contains most of the modified code and all of the graphics commands.

Commands are sent in the form 0FFH, xx,pp,pp,pp,pp........ Where 0FFh informs the PI a graphics command follows. 'xx' is the command byte and pp,pp,pp... any parameters

For some quirkey reason the codes 00 3f and 7f and bf are diverted by the pi or used for some other purpose, to overcome this (without stripping all the PI code) It was decided to send each byte (of parameters) as two separate nibbles but with bit 5 in each byte set. thus to send 56h one has to send 26h,25h. Note Low nibble is sent first.

To write a byte to GPIO bits 07 (eg A7h)

Send 0FFH, 0B0h, 027h, 02Ah

See file gfx note.txt

**MEMORY SEGMENT SWITCHING.**

Since the PAGE bit is the inverse of what was written care should be taken when addressing IO 4Ah - irrespective of which of the 48H functions is to be 'written to' Below is a simple example which will switch from the current memory segment to an alternative one.

|  |  |
| --- | --- |
| ASM | Comment |
| IN A,(MEMCON) | Read IO address 4Ah (Memory control & sundry register. |
| XOR 04 | Invert 'PAGE' bit - as it will always read the inverse of the last write. |
| SET 1,A | Set SEL1 high |
| RES 0,A | SET SEL0 low |
| OUT (MEMCON),A | Write to MEMCON IO address. |
|  |  |
|  |  |

The above is recommended for any and all intended write(s) to IO address 4Ah

Sequence, READ MEMCON, INVERT BIT 2, MODIFY other bits as required, WRITE to 4Ah.

**Some brief ROM functions**.

JR INITIALISE ;SOFT RESET/INITIALISE routine.

jp WAIT5X ;wait till HDD status = 5x -For ext user progs

jp GETBYTE ;Programs using KBD interrupt get data from here

ORG 08H

jp SEND\_STRING ;send string (Hl) over rs232 till 0

JP SEND\_BYTE ;send byte in A

ORG 10H

jp DUMP\_REGS ;Breakpoint Interrupt.

jp BEEP

ORG 18H

JP GETCHAR ; NON Interrupt get keyboard byte ;get byte in A

JP GETLINE ;Get a line of KBD CHARs 79+null store at 11B0h

ORG 20H

JP IDE\_CMDS ;Reserved IDE File functions.

JP DISK\_EXE ;load regs from WorkPad and run!

N.B all reg values entered here must be 2 digit hex, i.e. 00 77 ff etc. For the DTA value four hex digits must be supplied. The responses, 20,00,00,00,40,20,A000 Will read the first 32 sectors of the disk and place the data into memory at location A000h. Neat eh?

ORG 28H

JP SET\_TEXT\_COLOUR

call GRAPH\_REQ ;was jump??

ORG 30H ;CAUTION! STACK IS RESET!!!

jp MAINLOOP ;CAUTION! STACK IS RESET!!! ROM code mainloop (usually running in RAM.

**POWER\_UP**

Please ensure power is removed from the SBC when inserting/removing any connections, or storage devices from the system. No testing has been (or will be) undertaken to assess the effects or possible damage of the system if any connectors are 'HOT PLUGGED' - It is also recommended that power is removed whilst any links are removed/adjusted/installed. - Please observe ESD rules!.

Power up sequence (with PI) , - Connect all cables and supplies. Power on your display and wait until it's ready (usually this will show a 'No signal' or similar message). Next power on the PI, after a few seconds some messages (PI generated should appear). Finally an SBC prompt should appear, this will either be 'Z:>' or 'C:>'. If no disk is found a series of '-' characters will have been displayed, this denotes the expected disk status code of 05xh was not returned during or after several attempts and will result in the 'Z:>' prompt. In this situation the system can still be used for experiments or debugging. If 'No FAT' message is displayed the SBC was unable to find a valid FAT32 formatted device. (Formatting for all the development work was carried out on a PC running either WIN7 or XP, the disks were formatted as FAT32 with a cluster size of 64K. IF no messages appear after the 'Terminal Ready' message, reset the SBC which should solve the problem. If the C:> prompt appears it means a storage device has been found with a FAT32 formatted partition that is compatible with the SBC (it was found that disks formatted under LINUX were not always successful). So here we are with at either the C:> or the Z:> Monitor prompt. The image below shows the intended configuration of the system, which also works with the Pi Zero (non-WIFI version). No testing with a Model 3 has been undertaken as it uses a different core to the Model Zero.

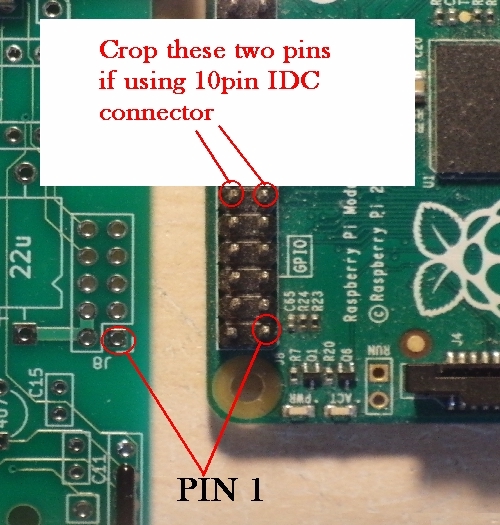
The picture below shows the system wired as intended. The USB connector at the top of the picture is connected to the Keyboard, The Large black connect to the right of the PI is the HDMI Monitor cable and below that is the (white) power connector to the PI. The interconnecting cable shown is a 16Pin cable BUT only 10 pins are used thus a 10PIN connector is all that is required.



Below showing use with a pi 'zero' Top to bottom connectors: PWR,KBD,Mini HDMI. Note only the original (non-wifi) Pi zero has been tested with the system and therefore proven.

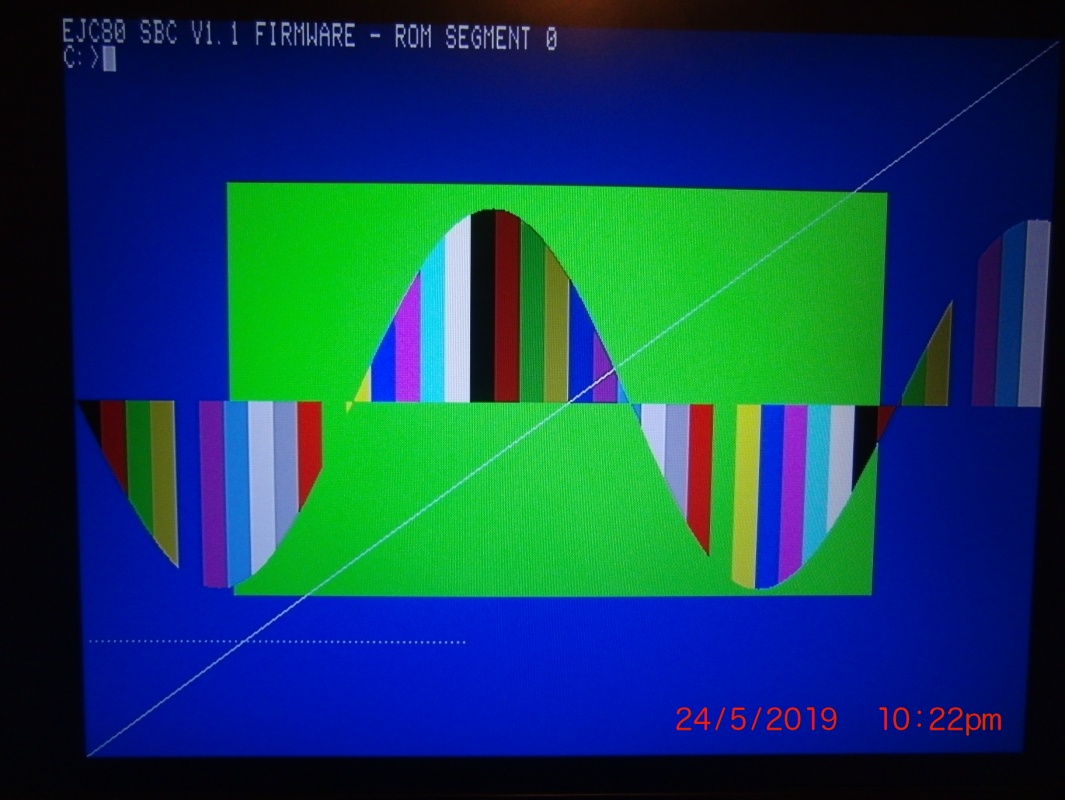


PI to SBC detail First 10 pins of the PI to be connected to the SBC pin for pin.



**GRAPP**

At either of the monitor prompts Typing:- GRAPP <CR> will display the following screen which is generated by sending commands to the PI to produce a Rectangle, set colour(s), and draw lines. (the sine function is achieved by the z80 issuing command to produce a set of vertical lines of X sin(y) height). If the display shows what looks like random text, then the PI may not have the correct level of software installed or, where the PI software has been altered/manually installed or not supplied with the SBC. The image shows the Set Pixel, Draw Rectangle, Draw Line, Set Colour and Clear\_Home features available to the SBC for graphics.



**Pi - Z80 Comms set up.**

The file CONFIG.TXT file found on the PI root directory contains the line: ***init\_uart\_clock=3000000***  This is set as the primary clock frequency, which is then divided down to give a baud rate [currently] set at 38400. In order to match this the Z80 is set up with a 16MHz xtal oscillator. You will need to ensure your own CPU functions reliably at this clock rate. For the Z80 the baud rate is set approximately to CLK / 417 thus at 16MHZ this is 38370 Baud which is well within 1/2 bit tolerance. The Pi source file which contains the divider settings is in the file uart.s but to adjust this would require a complete re-compile of the Pi core and is thus not within the scope of this document. Where a 16Mhz clock is not used it is either necessary to change Eprom values and re-program the Z80 Eprom or, (and this is the easiest and recommended solution) the value in CONFIG.TXT file should be changed.

New config.txt value = z80 clock (hz) / 5.333r

Thus 16Mhz/5.333r = 3Mhz

for an 8Mhz z80 Clock 8Mhz/5.333r = 1.5Mhz //init\_uart\_clock=1500000

for a 12Mhz osc 12Mhz/5.3333 = 2.25Mhz //init\_uart\_clock=2250000

WARNING! if you clock frequency is below 8Mhz you may need to reprogram the Z80 rom as the program divider values may need tweaking no testing has been carried out below a z80 clk of 8Mhz/.