
# Domipheus Labs

### Stuff that interests Colin ‘Domipheus’ Riley





# Teensy Z80 – Part 1 – Intro, Memory, Serial I/O and Display

Posted Jan 9, 2015, Reading time: 14 minutes.

# My Teensy Z80 Homebrew Computer

A few months ago, I bid on several ‘box of surplus electronic components’ listings on ebay. My lab needed some more components and I saw some of the things I needed in the listing pictures, so thought I’d go for it. I won all of them, at pretty much my lowest bid price, and when I got the boxes was really happy (I paid ~£20 for >£200 of components, most sealed new). At the bottom of one box was a Zilog Z80 CPU, in 40-pin DIP. It’s a Z84C0008PEC, designed to run at 8MHz. It looked pristine, but was not sealed, and it sat in my junk box for quite some time.


Last month I won a contest at Pimoroni (check them out, they are awesome), where I received £100 in gift vouchers. In the box of delight which followed, were two Teensy 3.1 boards. I didn't know what I wanted to do with them, I just knew they packed some punch and had a plugin for the Arduino IDE. Soon after unpacking them, and seeing just how many I/O pins it had, I wondered if it was enough to connect a Z80 for a 'working' computer.

*It was!*

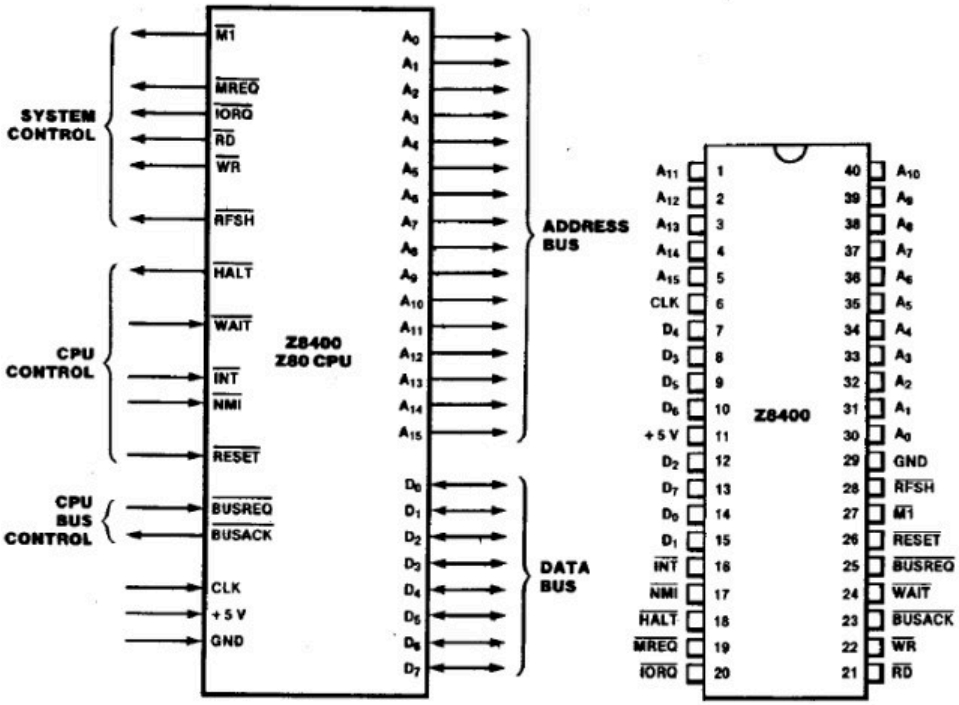
So my project over the holiday season was decided: TeensyZ80. I wanted to have a usable Z80 running its own code, with the teensy supporting it providing the RAM, I/O peripherals, and clock.

Now, some homebrew/single board computer enthusiasts may be groaning at this point "another Z80 board, and it's not even using real ICs" but I was more interested in the timing, what speed could you actually achieve, and the Z80 in general. My first computer was a ZX Spectrum, and I'm feeling quite sentimental. I never programmed in Z80 assembly and thought that should change.

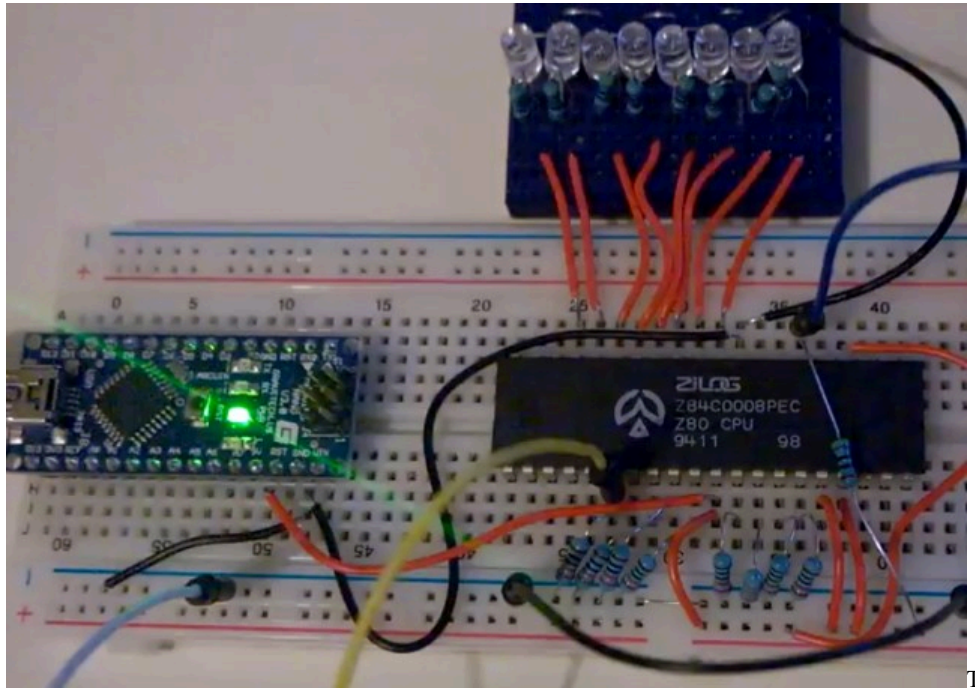
It seemed after reading the datasheets that it would be quite simple to achieve, but the first thing I needed to do was check if the Z80 actually worked!

# A Z80 test circuit

The Z80 pinout shows just how simple this is to hook up and at least test PC increment on NOP instructions. Hooking up some LEDS (with resistors!) to some low address bus lines, powering it with USB 5v and also fixing some of the other status inputs to 5v – they are active low inputs – means we should see the address bus change as the CPU requests from memory.




The Z80 NOP instruction is represented by the 1-byte sequence 0x00, so if we pull all the data lines low to ground we should see the address bus increment as it executes NOPS at every data location.

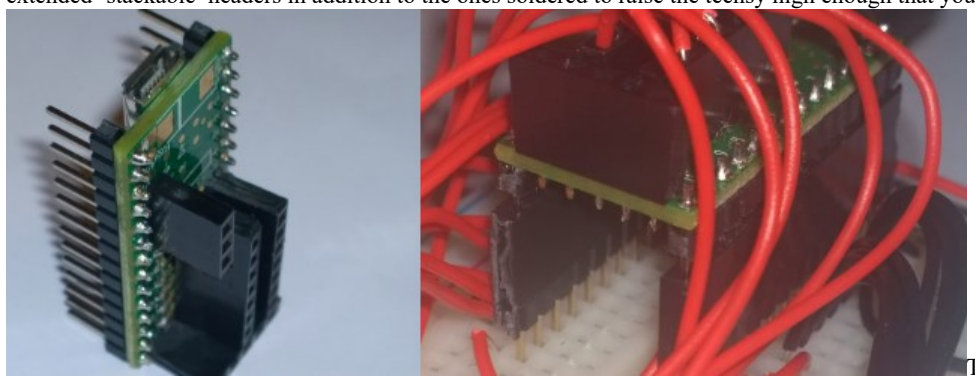



The Z80 uses static registers, so this means there is no minimum clock frequency to keep the CPU running. Because of this, technically you can run this with a clock sourced from a push switch and some other circuitry to clean up the signal, but I just used an Arduino and got it to output a slow square wave. With this all put together on a breadboard, the Z80 booted, but was quite erratic – I was failing to reset the chip. To reset, you pull the reset line low, for a minimum of 4 complete clock cycles. Once I did this, the address bus started at 0 and worked up, counting. One thing to note is that it takes several cycles to fetch and execute instructions on the z80, so the bus does not increment every cycle, but usually every 4 cycles. More information about Z80 test circuits can be found here.

A bonus video below, showing the need for clean clocks. this is what happens when I just touch the clock pin with my fingers!

# On to the Teensy

At this point I was confident the z80 worked, and so then started soldering headers to the teensy I got from Pimoroni. I wanted this for breadboard use, so put the headers on the opposite side from what most would expect, and then soldered 90 degree female headers to the underside pad I/O pins. This meant I had all the Teensy 3.1 pins available to me, on the breadboard. One issue with this is that you will need to use the extended ‘stackable’ headers in addition to the ones soldered to raise the teensy high enough that you can press the program button on the unit to flash it.



The first thing to do was attempt to run the test circuit and the z80 at 3.3 volts. The Teensy is 5v tolerant, but some

of the analog pins are 3v3 input only, so I wanted to make sure. Once this was confirmed working, I got on to making the test circuit do additional things. I added an SPI 2.2″ TFT to easily display debugging information and started connecting the data bus to the Teensy.

The first job was to create a clock signal for the Z80 to work with. To make things easier, this was going to be designed in such a way that the clock is completely synchronous with all other events, in that the loop() function of the Teensy simply sets the clock high, samples inputs, provides outputs, sets the clock low, return. Doing this allows for every operation to stall the CPU by way of delaying it's lower clock edge, meaning we do not need to use the WAIT line to stall the Z80. It makes implementation easier for this project, but isn't how things are done normally – although clock manipulation is a useful tool. One thing to note is that if you stall the clock like this, it should be done while it is high. Extended periods of the clock being low may lead to unexpected behavior.

# Memory Requests

The next thing to implement is memory reading and writing. To do that, you need to know how all of the signals from the z80 cooperate to form a memory request. The Z80 manual has timing diagrams and you can see the one regarding memory requests below.

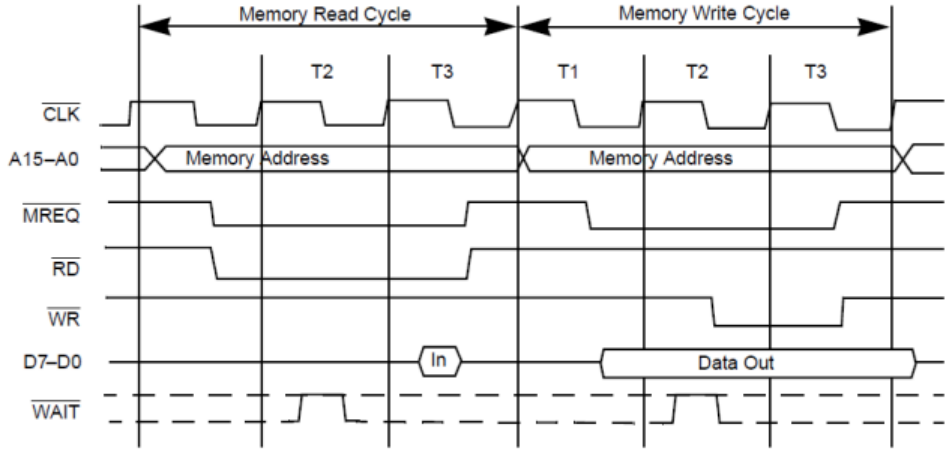



**Figure 6. Memory Read or Write Cycle**

As I mentioned before, many cycles are required before a single operation is completed. They are grouped into 'T-states'. We are interested first with reading from RAM, so we look at the MREQ signal and the RD signal, when both active means a memory read is requested. From the diagram above we can see that simply checking for those two signals, then sampling the address bus, and then writing the data bus, should be enough. So the Teensy sketch looks now like a very large amount of pin definitions, Address lines input, and data lines output:

```
void loop() {
    digitalWrite(Z_CLK, HIGH);
    delay(300);
    updateZ80Control();
    if (MEMREQ && RD) {
        readAddressBus();
        if (addressBus < ROM_LENGTH) {
            dataBus = TEST_ROM[addressBus];
        } else {
            dataBus = 0x0;
        }
        writeDataBus();
    }
    digitalWrite(Z_CLK, LOW);
}
```

This should be enough to satisfy reads from the Teensy. The readAddressBus() and writeDataBus() functions simply build a short value from i/o pins, or write a byte value to i/o from a byte value.

```
void readAddressBus() {
    addressBus = 0;
```

```
    addressBus |= ((digitalRead(AD0)==HIGH)?1:0)<<0;
    addressBus |= ((digitalRead(AD1)==HIGH)?1:0)<<1;
    addressBus |= ((digitalRead(AD2)==HIGH)?1:0)<<2;
    addressBus |= ((digitalRead(AD3)==HIGH)?1:0)<<3;
snip
```

… and so on

Now the Z80 should be able to read from memory, but I had to now write some Z80 code to test that the CPU correctly interpreted this data. So I downloaded ZMAC.

Despite the ZX Spectrum being my first computer, I never did any coding on it. Nothing. So this is my first look at Z80 assembly. Really, though, it's pretty basic stuff as far as an ISA goes. I coded a small example which is assembled at location 0 – all it does is execute several nops, before entering an infinite loop. I assembled with zmac, and then used bin2h on the outpu .cim file so I could simply paste the instruction stream into my Teensy Sketch source as RAM. I did this, built it all, and….

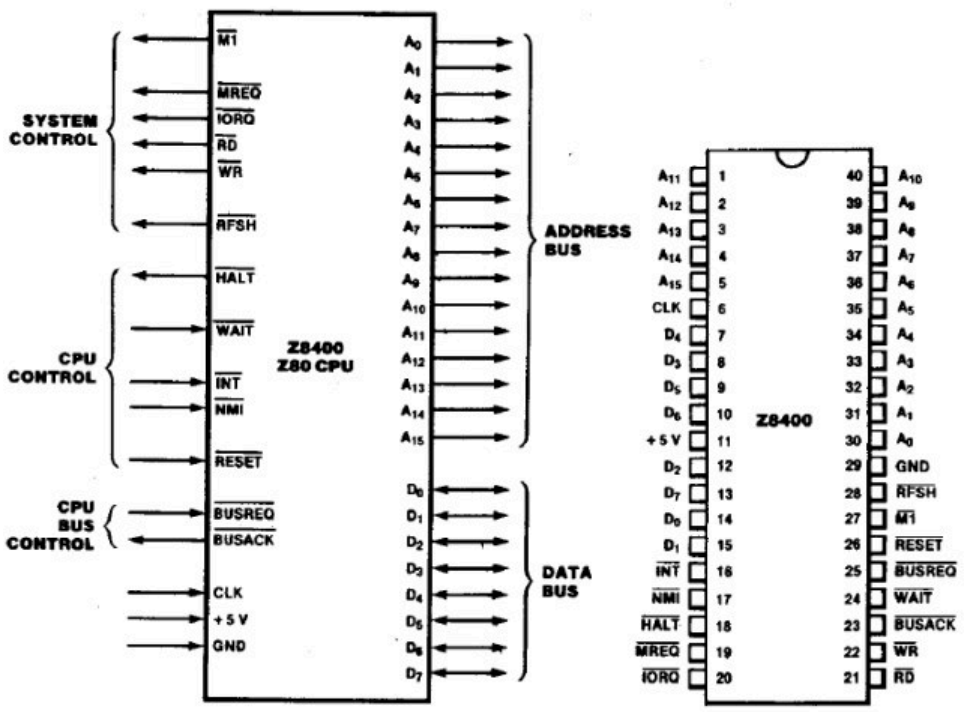
The address bus counted up as expected, and then become incredibly erratic.

```
;  To assemble: zmac asm.z
;  To get data for the sketch source:
;     bin2h zout/asm.cim > asm_binary.h

    .org 0000h
start:
    nop
    nop
    nop
infloop:
    jr infloop
```

I couldn't figure out what was wrong. I wired up an LED to the HALT output of the CPU and assembled a program that should just immediately halt the CPU, thus make the LED go out, but it didn't.

```
.org 0000h
start:
    halt
```

After a length of time I'm far too embarrassed to state, I realised the data bus was wired to the Teensy in the wrong order. The address bus pins are all in order from A0 to A15, but the data bus is not. Oops. This meant the data was being fed to the Z80 from the teensy wrongly, which is why the address bus was erratic: it was executing completely different opcodes to what was intended. Here is the pinout of the Z80, again.




After fixing that little mishap, we had a Z80 which counted up to the address where the infinite loop was, and it stayed there. Note, you may still see the address bus count for some cycles. That's the refresh cycles at work – which are used to refresh dynamic ram. We can just ignore these cycles. If you connect an LED to the RFSH line you can tell the cycles to ignore. I attached this line to the Teensy and if it's active on a clock I simply skip everything. I also connected an LED to the HALT line – the led is on when the z80 is active, and off when in a halt state.

Next, implement writing to RAM, of course! (Yes, I know the RAM is in a variable called TEST_ROM, stupidly 🙂 ).

```
if (MEMREQ_val) {
    if (RD_val) {
        if (addressBus < ROM_LENGTH) {
            dataBus = TEST_ROM[addressBus];
        } else {
            dataBus = 0x0;
        }
        writeDataBus();
    } else if (WR_val) {
        readDataBus();
        if (addressBus < ROM_LENGTH) {
            TEST_ROM[addressBus] = dataBus;
        }
    }
}
```

The implementations of readDataBus() and writeDataBus() must remember to set the input/output mode of the pin. This can be done at the start of the function. For example:

```
void writeDataBus() {
    pinMode(D0, OUTPUT);
    pinMode(D1, OUTPUT);
    pinMode(D2, OUTPUT);
    pinMode(D3, OUTPUT);
    pinMode(D4, OUTPUT);
```

```
    pinMode(D5, OUTPUT);
    pinMode(D6, OUTPUT);
    pinMode(D7, OUTPUT);

    digitalWrite(D0, (dataBus&(1<<0))?HIGH:LOW);
    digitalWrite(D1, (dataBus&(1<<1))?HIGH:LOW);
    digitalWrite(D2, (dataBus&(1<<2))?HIGH:LOW);
    digitalWrite(D3, (dataBus&(1<<3))?HIGH:LOW);
    digitalWrite(D4, (dataBus&(1<<4))?HIGH:LOW);
    digitalWrite(D5, (dataBus&(1<<5))?HIGH:LOW);
    digitalWrite(D6, (dataBus&(1<<6))?HIGH:LOW);
    digitalWrite(D7, (dataBus&(1<<7))?HIGH:LOW);
}
```

That's it. This is actually good enough, now, to get the z80 doing some real computation. I picked a section of RAM as a 'frame buffer' and just wrote ASCII characters to there from another area of Z80 memory, and got the Teensy every loop() to draw the contents of that frame buffer to the SPI TFT screen. It all worked. Slowly, mind – redrawing the TFT is very slow given how quickly we want the clock to tick.

At the moment, you cannot get data into the z80 that isn't already flashed to the Teensy. We can solve this by using the Serial feature of the Teensy, and expose this functionality to the z80 via it's I/O ports. It's what I did next.

# I/O Requests

The Z80 has another status pin, IOREQ, which if active signifies the low half of the Address Bus holds a port number, and the a read or Write should be applied to it. I would implement a serial status/command port, and a data port as follows:

- A read from the Status Port will return with the number of bytes available in the buffer for reading.
- A write to the Status port will be interpreted as a command to be carried out by the 'serial device'
- A read from the Data port will pop a byte from the buffer and put it on the data bus.
- A write to the Data port will write the byte on the data bus to the serial device.

With this functionality, I could use my computer as input keyboard or output terminal. I would allow the Z80 to configure the serial rate and other options, and then initialize the connection. I could then implement getchar and putchar with stdin/out defaulting to the serial connection.

The serial data port will be very easy to implement with the teensy Serial object. If a write is made to the port, we do Serial.write(dataBus), and if a read is made we do dataBus = Serial.read().

We want the z80 to configure the connection via a command port. We may as well have the Z80 do something, given we've really cheated here by making the Teensy do all heavy lifting. The first thing we want is the serial rate, which will be a 16 bit value. For this, we want an 8-bit 'command' followed by the 2 halfs of the 16-bit rate. We'll call this a command packet. As this uses multiple i/o writes to achieve a full packet, we need to store some state information on the Teensy over multiple cycles. It's done very simply, the serial device has a current command state, and we work out from that what any I/O status write should do. So for setting the rate, we have the following:

```
if (portAddress == PORT_SERIAL_CMD)
{
    if (ioSerialCurrentMode == SERIAL_CMD_READY)
    {
        //ready for commands, so put us in the right mode
        ioSerialCurrentMode = dataBus;
    }
    else
    {
        //take data for the given mode
        if (ioSerialCurrentMode == SERIAL_CMD_SET_RATE)
        {
            ioSerialCurrentMode = SERIAL_CMD_SET_RATE_2;
            ioSerialRate = dataBus;
        }
        else if (ioSerialCurrentMode == SERIAL_CMD_SET_RATE_2)
        {
```

```
            ioSerialCurrentMode = SERIAL_CMD_READY;
            ioSerialRate |= (((unsigned short)dataBus)<<8U);
        }
    }
}
```

When we run this with the following Z80 ASM:

```
ld a, SERIAL_CMD_SET_RATE
    out ($01), a       ; set the serial rate (9600)
    ld a, 80h
    out ($01), a
    ld a, 25h
    out ($01), a
```

we discover things don't really work. This is due to the I/O operation taking many cycles, and our code assumes one cycle, or execution of loop() between runs. So we need to debounce the I/O. Looking at the timing diagram from the Z80 guide we see that we must wait 4 cycles.

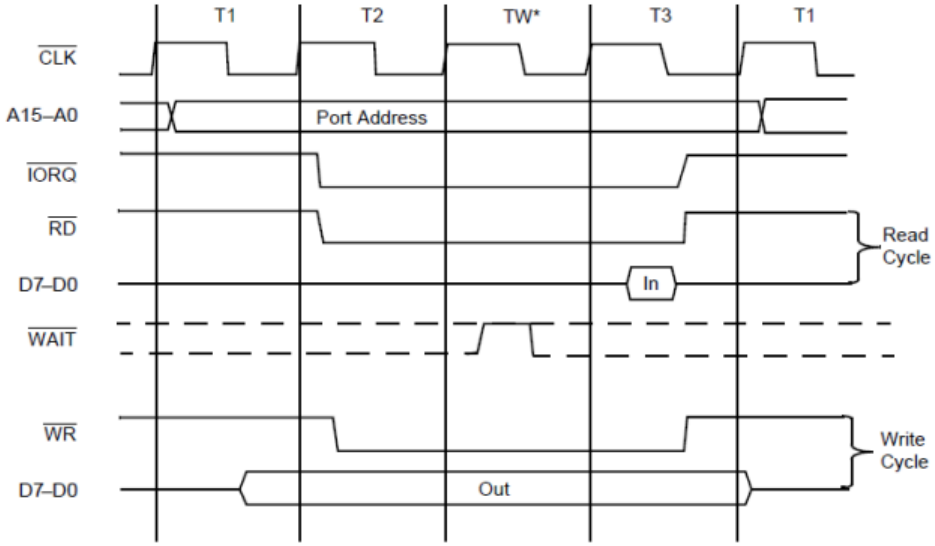



**Figure 7. Input or Output Cycles**

**Note:** *In Figure 7, TW is an automatically-inserted WAIT state.

Once this is changed so we always wait 4 cycles, we can implement the INIT process. I've got it set up to wait for a connection from the teensy before returning from the INIT command, but this could easily be put into the status 'bytes available' read port.

```
if (dataBus == SERIAL_CMD_INIT)
{
    Serial.begin(ioSerialRate);
    while (!Serial); // wait for a connection
    ioSerialInitialized = 1;
}
```

So, we can read and write serial data, cool. But I really want the z80 to have its 'own' screen. Not the hack we did before. We can do it two ways – set up an area of memory as some video ram and populate it, getting the teensy every frame to draw what is in that memory, be it characters or pixels, or we can create a set of ports to manipulate a virtual console. I've done both, but I'll only look into the virtual console using ports here.

We need several functions to get a console:

- put character
- get/set column
- get/set row
- optionally, set colour.

The console will be 32 columns by 24 rows. To get things up and running quickly, I made the decision to simply have the teensy deal with the set column/row edge cases, and have put character increment along the console each time it's used. For set colour, I used simple state to allow 16-but 5:6:5 colour input via two port writes. The code looks as follows:

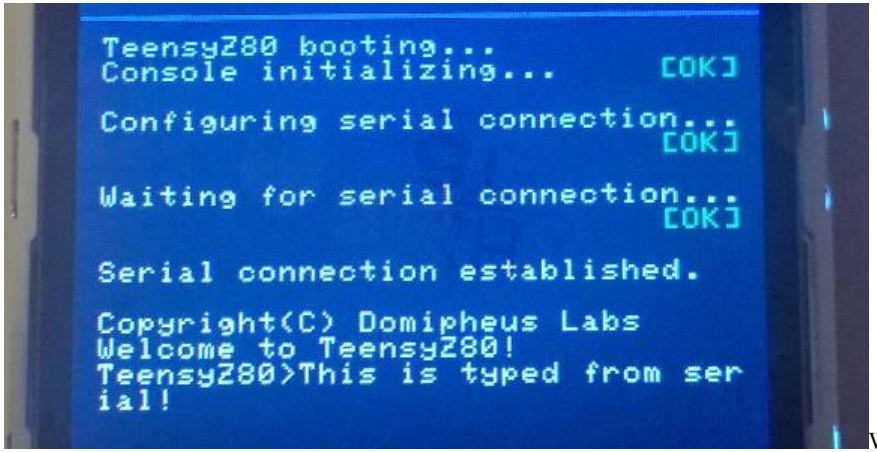
```
else if (portAddress == PORT_DISP_SETCOLOUR)
{
    if ((console_current_color_state&0x1)==0x1)
    {
        console_current_color |= dataBus<<8U;
    }
    else
    {
        console_current_color = dataBus;
    }
    console_current_color_state++;
}
```

The putchar port code is very simple again:

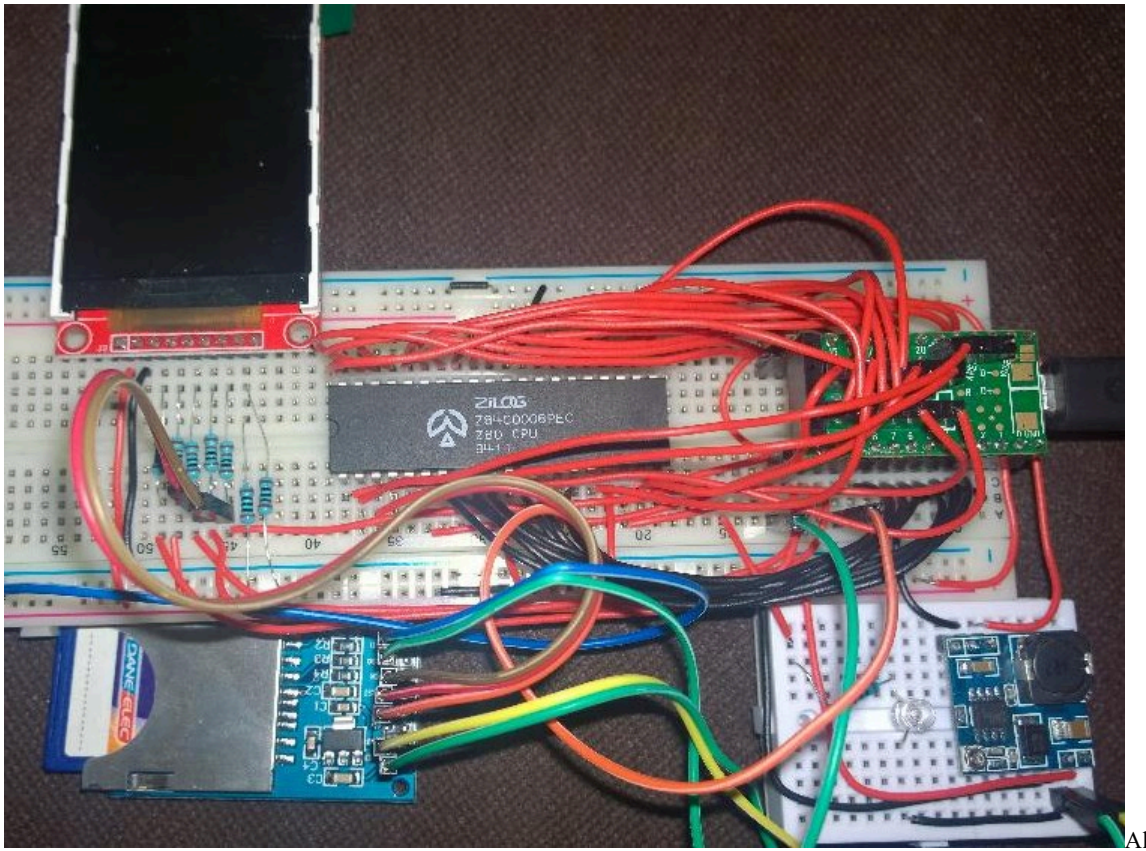
```
else if (portAddress == PORT_DISP_PUTCHAR)
{
    char c = dataBus;
    tft.setTextColor( console_current_color, ILI9341_BLACK);
    tft.setCursor (CONSOLE_START_X + (console_current_col*CONSOLE_FONTX), CONSOLE_START_Y + (console_current_row*CONSOLE_FONTY));
    tft.print(fmtstring("%c", c));

    console_current_col++;
    if (console_current_col >= CONSOLE_COLUMNS) {
        console_current_col = 0;
        console_current_row++;
    }
    if (console_current_row >= CONSOLE_ROWS) {
        console_current_row = 0;
    }
}
```

An optimization to the above is to only set the text colour on a second textcolour I/O write, but at the time of writing I had some debug draw stuff going on in the sketch, so wanted to ensure the console always used the correct colours. Hence the setTextColor call each putchar.

# Wrapping up




With this, we have a display, serial in/out, and can now try writing some more z80 ASM! But for now I think this is enough for this part. I've already got Mode 2 interrupts working, and I'm interfacing an SD card interface. I'll be cheating heavily with that, getting the Teensy to do all of the FAT heavy lifting. But it's a fun exercise.



All of the code for this is up my github https://github.com/Domipheus. Note it may not line up exactly with this post, as it's being edited fairly often.

If you enjoyed this, please let me know via twitter @domipheus.

The next part in this series of posts is available here.

- Arduino
- Electronics
- Projects
- Teensy
- TeensyZ80
- Z80