SPI and I²C Interface for the TEC-1 and SC-1

Presenting the SPI²C, a simple I/O interface and supporting software for interfacing with modern peripherals.

By Craig Hart

Introduction

One of the limitations of the TEC platform is a lack of 'usable' I/O to communicate with the outside world. TE Issue 14 brought us the I/O board, however this design was very simple as each I/O pin was either a fixed input or output.

Many modern peripherals communicate using bidirectional signalling protocols – that is, one pin is be used both to transmit and receive data; depending on the state of the communication protocol, the direction of data flow varies in real time.

The SPI²C board overcomes these limitations, allowing a new world of cheap peripherals to be easily supported.

SPI²C implements two common interfaces found across a large range of modern peripherals; firstly the **SPI** or Serial Peripheral Interface bus, and secondly the **I²C** (pronounced I squared C or simply I to C) bus.

Hardware Design

In the spirit of TE learning, we have not simply dumped a clever custom chip onto the Z80 bus.

Instead, the SPI²C introduces a new design building block for the TEC – a true Bi-Directional IO pin.

We have built our interfaces using easy to understand 74xxx family TTL logic chips. In this way the entire operation of the design can be understood and followed.

Of course, any hardware is only as good as the software it comes with, and so we are providing supporting software that implements the I²C and SPI bus protocols at the most basic level.

Both protocols are simple, one-bit asynchronous busses, and can be easily implemented by software 'bit banging'.

This approach is far more educational than just 'talking to a smart controller chip' or calling a software library you don't have the source code to.

SPI - a simple bus

SPI is a very simple synchronous serial bus. It has a clock(SCLK) line, a chip select(SS) line, and two data lines read(MISO) and write(MOSI).

Don't be put off by the unfamiliar labels, just mentally substitute ('In' or 'read' for MISO, and 'Out' or 'write' for MOSI).

In the above image, the TEC plays the role of Master, and our peripheral device(s) are the Slave(s).

When we say the bus is synchronous, we simply mean it has a clock(SCLK) line that dictates bus timing.

Data is considered valid when (and only when) the state of the SCLK line changes. Our code can be as fast or slow as we please and it does not matter what frequency the Z80 is running at or if interrupts are occurring.

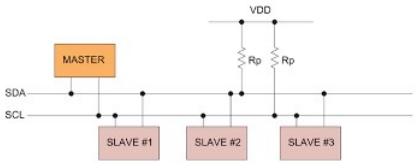
The SS line works much like it does on the Z80 data bus - multiple SPI slave devices are connected to the bus, but only one device can be active at a time, as selected by the SS pin of that device; each device has it's own, non-shared, SS pin. The MOSI, MISO and SCLK lines are wired in parallel to all devices.

The MISO pin is normally held in tri-state condition by all devices. Only when a slave device is actively sending data is the pin not tri-stated; the SPI standard says this pin is active LOW, so an SPI slave can only pull this pin low or tri-state. Slaves never output a logic 1; therefore a pull-up resistor is required to hold the pin in a logic 1 state whenever the bus is idle.

Theoretically, an unlimited number of SPI devices can share a single bus, however there are practical limits to the number of devices; our board offers up to 6 Chip Select lines; hence we can support up to 6 devices.

I²C – something new

The I²C bus is a much more interesting fellow. with I²C the whole bus is just two wires – clock(SCL) and data(SDA).



The bus is also synchronous in nature (SCL line), however a very clever signalling protocol and some smart hardware design is used to reduce the other pins down to one. Chip select is integrated within the

bus protocol (each I2C device has a unique 'address' on the I2C bus), hence the SDA pin performs 3 functions - data in, data out and device select.

More on this later.

Hardware Operation

The hardware design can be broken down into several sections, as follows:

I/O address decoder

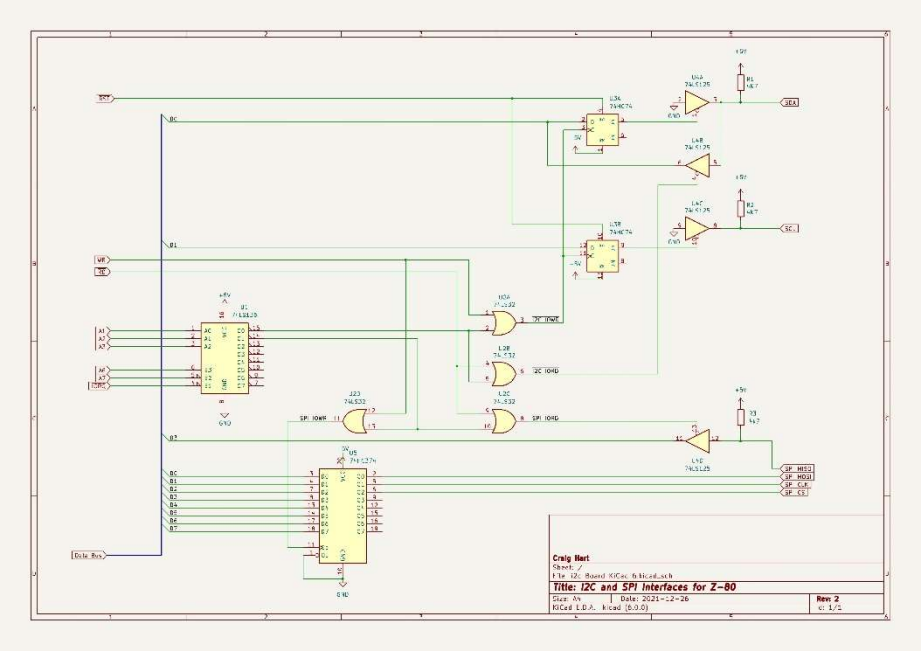
The 74HC138 address decoder is functionally similar to the TECs own 74LS138s – except it decodes the IO address range 40h to 7Fh. Note that we ignore address line A0, so every odd port is a duplicate of the preceding even-numbered port.

This means port 40h and 41h are an equivalent pair as far as software is concerned (…and port pairs 41h & 42h, 43h and 44h, etc.) so code can address either port and read or write the same device.

Also, since we are ignoring address lines A4 and A5, the I/O ports 'wrap around' every 16 address, so port 40h, 50h, 60h and 70h all access the same device. The Z80 has plenty of IO port space so we don't really care about wasting some; its easier and cheaper than fully decoding a single IO port, which would require more logic.

The 74HC138 chip can also be omitted entirely if you wish to use the TEC's built in IO ports e.g. ports 6 and 7. Just leave out the 74HC138 and connect the I2C_SEL and SPI_SEL pins to the TEC's IO ports instead.

The TEC also exhibits a 'wrap around' effect with its IO address decoding. See the article '[TEC I/O Decoding Improvements]' for more information as a TEC mod needs to be done if you don't want to use the TECs built in IO ports.



I/O read and write cycle decoder

The purpose of the 74HC32 is to take our port select signals from the 74HC138 (or TEC) and combine them with the Z80's RD and WR signals, to create separate IORead and IOWrite signals.

This is important because different chips are used for the read and write functions, so they need to be selected individually based on a Z80 IN or OUT instruction (IOread or IOwrite cycle).

SPI controller

The 74HC374 is responsible for the write side of the SPI bus - in fact, if you have a 'write only' type SPI device (e.g. an LED display), this chip is essentially all you need.

The SPI_IOWR signal from the 74HC32 enables this chip only when data is written to the SPI port – and all 8 bits are latched by this chip.

Reading from SPI devices is performed by Gate 4 of the 74HC125, and simply gates the MISO pin onto the Z80 data bus during a read cycle. The SPI bus's MISO pin is held in tri-state (high impedance) mode whenever data is not actively being sent by a slave device, hence this pin is shared by all SPI slave devices.

The 4k7 pull up resistor holds the MISO line high when idle, hence our software will read a '1' whenever the bus is idle. SPI devices only ever output a logic 0 to the MISO pin (*never* a logic 1!!)

Reads are gated onto the Z80 data bus to bit D3 by the 74HC125 in response to a decoded IORead cycle.

I²C controller

The I²C controller is a little more complex than SPI. I²C pins are active low; therefore held high by pull-up resistors (devices tr-state) whenever idle.

We need to create an SDA pin that supports the following abilities:

a) Be at logic 1 when idle
b) Output a logic 0 or 1 during our writes
c) Read an input 0 or 1 state from a slave device (and not our own previous output) during reads
d) Be independent of the SCL pin

We can't simply use a 74HC374 here as it is not bidirectional and other 74HC24x family chips won't suit either as we can't address pins individually (e.g. have one pin as an

output at the same time another pin is set as an input).

To overcome these limitations, we have built two independent one-bit controllers using a 74HC74 flip flop (output latch) and 2 gates from a 74HC125 (tri-state buffer).

SDA pin
Firstly, the 74HC74 latches our writes of bit D0.

If our write is a logic 1, the I^2C bus is placed into tri-state, since the slave devices are also idle/tri-state, and the 4k7 resistor pulls the bus up to logic 1.

If we write a logic 0, this is latched by the 74HC74 and then a 0 is gated onto the bus by the 74HC125.

Note the input of the 74HC125 is wired to ground – hence we can only ever output a 0 to the I2C bus, never a 1.

Input is the same as for SPI – the other 74HC125 gate simply gates the SDA line directly onto the Z80 data bus during an IORead.

The observant amongst you would have noted – to read valid data from the SDA pin, we need to first output a '1' to the SDA pin, in order to set it to tri-state mode; otherwise a bus short could occur.

The Z80 reset line sets the 74HC74 flipflop to a logic 1 so as to ensure the I2C bus is tri-stated at power up.

Our software should also ensure a bus clash never occurs.

SCL pin
The SCL pin works the same way as the SDA pin, except it is only ever driven as an output, so we don't need a '125 gate on this pin, and any data read of this bit will be undefined.