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, software controlled 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.

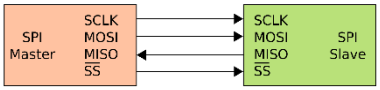
Of course, any hardware is only as good as the software it comes with, and so we are providing supporting software that implements the I²C and SPI bus protocols at the most basic level via

Both protocols are simple, one bit at a time asynchronous busses, which 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 is the Slave.

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 can be 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.

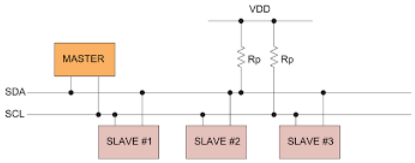
Theoretically, an unlimited number of SPI devices can share a single bus, however there are practical limits to the number of devices; our board offers up to 6 Chip Select lines; hence we can support up to 6 devices in the basic design.

Some non-standard SPI bus devices also have extra control pins for various tasks such as Reset, Backlight enable, mode selection or other custom functions. We can easily support these types of devices by simply changing our software and re-purposing the extra Chip Select lines as control signals.

Duinotech XC-3714: an SPI bus based 8 Digit 7-seg display is available for around $10 from Jaycar etc.

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.

Hardware Operation

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

I/O address decoder

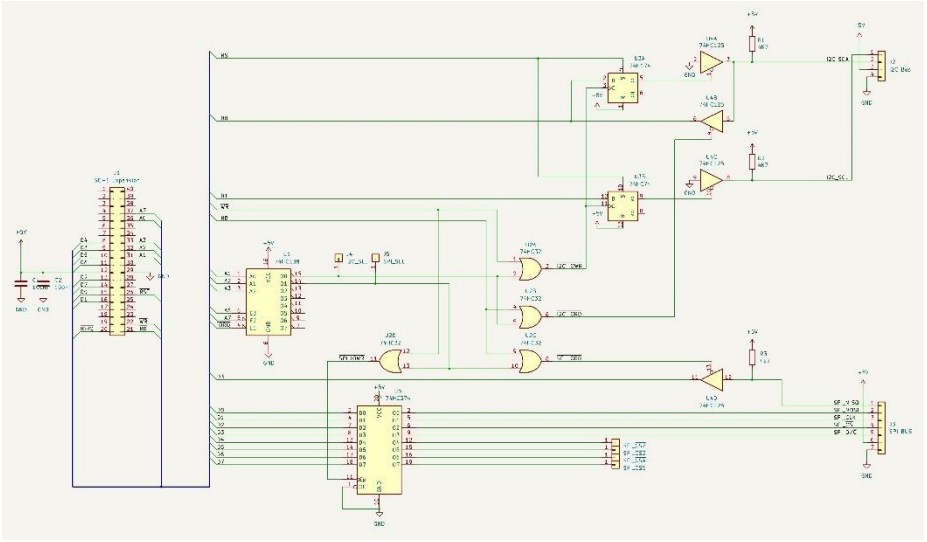
The 74HC138 address decoder is functionally similar to the TECs own 74LS138s – except it decodes the IO address range 40h to 7Fh. Note that we ignore address line A0, so every odd port is a duplicate of the preceding even-numbered port.

This means port 40h and 41h are an equivalent pair as far as software is concerned (...and port pairs 42h & 43h, 44h and 45h, 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 ports 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 a lot 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 port select pins instead.

This design will 'drop in' to the SC1's



Expansion connector with no further modifications.

The TEC however, exhibits a 'wrap around' effect with its onboard IO address decoding that occupies the whole 256 IO port range. See the article '[TEC I/O Decoding Improvements]' for the TEC mod which fixes this issue..

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 I²C 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.

Assembly

Any seasoned TEC builder should find assembly straight forward; simply load the parts onto the PCB as shown, the order isn't critical.

However, first decide if you will be building the SC1 version or the TEC version.

SC-1

The SC1 version is easy – fill all components as shown. The boar is designed to plug directly to the SC-1;'s expansion header.

TEC-1

First, decide how you wish to interface to the TEC – either omit the 74HC138 and use the TEC's own IO decoder, or, modify your TEC to fix the IO decoder wrap around issue, and build as per the SC-1.

As the board uses a 40 pin Z80 bus header you will also need to pick up various signals from the TEC Expansion port and the Z80 CPU. Some soldering to the TEC PCB will be required.

Alternatively, use Craig Jones' TEC CPU riser board to create an SC-1 style expansion bus connector.

Software

The key to this project is the really the software; the hardware itself simply handles the electrical signals, and all the smarts is in the software alone.

The latest versions of all the software as well as schematic, PCB design, supporting datasheets etc. are available at my GitHub site https://github.com/1971Merlin/SPI2C

A detailed line by line description of the software is beyond this article, however a quick review as follows:

spi_7segs

This program tests basic SPI bus functionality, using a MAX7219 8 digit 7-segment display module such as the Duinotech XC-3714 which is readily available from Jaycar for around $10.

The program simply scrolls a display buffer in RAM across the displays, but it demonstrates several code blocks in doing so – initializing and writing to the SPI bus, as well as managing a simple display buffer.

As a side note, it's mazing that you can buy EIGHT 7-seg displays, on a PCB, assembled, with a controller chip, pin headers etc. for just ten dollars. To build this yourself from discrete parts would be many times more expensive – hence the value in using these cheap modules and leveraging the wealth of products already out there flooding the Arduino market.

i2c test

This program builds on spi_7segs, adding support for an I²C bus based DS1307 Real Time Clock chip – again we are using the Duniotech XC-4450.

Obviously this program introduces the I2C bus control code, however it is a much more 'complete' program that includes several utility routines such as display scanning,, keyboard polling and also conditional assembly to build for either the SC-1 or TEC-1 hardware platforms.

Compiling the code

The code is written and assembled into the final binary using Telemark TASM – the table based assembler, which is available from

https://www.ticalc.org/archives/files/fileinfo/250/25051.html

I personally use TASM to generate a .obj file in Intel HEX format. I then use the the SC-1's serial upload feature (Fn 1) to upload the code from a PC.

Of course you can type the HEX bytes in by hand and write your code with pen, paper and opcode lookup tables the way TE introduced TEC computing, but I strongly suggest and encourage use of a modern development environment. More on this in another article.

To compile the code, simply run tasm -80 -g0 filename.asm and it will output a .obj file in intel hex, and a .lst file which is a text file showing the commented listing and the hex bytes to be keyed in.

Where to From Here?

Having introduced some basic routines and the 'raw' code, well the sky is the limit in terms of possibilities.

Just about any SPI or I^2C bus device can be used – and my goal is to build over time a library of subroutines to support numerous devices.

I am looking at support for the Duinotech XC-4616 84x48 LCD display next. This is a dot matrix display that can display graphics and text in a very flexible way. This display is also known as the "Nokia 5110" display, and is well supported in the community.

In time just about any code for this display could be ported to the TEC platform.

Any code will of course be published on Github as it is developed. And I encourage readers to become familiar with the various TEC Github resources available in the community.

Beyond this, I encourage and welcome you, dear reader, to start experimenting. Feel free to provide feedback, bug fixes and new features via Github and the TEC-1 Facebook group, and lets see what we can grow from here.

I look forward to receiving your input.