INTERFACING TO THE I2C BUS

INTRODUCTION

If you are not yet familiar with the I2C bus but you do know SPI bus then the best way to describe it is to say that it is a multidrop SPI bus that only needs two I/O. An SPI bus employs a clock line, a data out, a data in, and a chip select so that it may select a chip and start clocking serial data in or out, in a shift register fashion. When you want to add another SPI bus chip you can reuse the same clock, data in, and data out but you still need another I/O for selecting the second chip. This can quickly get out of hand with many chips but the I2C bus "cheats" by using a special start condition that can be sensed easily by I2C devices so that it can send an address out serially of the chip it wants "selected".The address is 7-bits plus a R/W bit so that it can address up to 118 devices with another 10 addresses reserved for special cases and address expansion. The protocol is fully explained in the I2C specifications document but suffice to say that this is the bare basics of the I2C bus. One important point to note is that since the bus is shared it uses pullup resistors on both lines and devices simply switch to ground to sink current when driving or leave the line floating high.

HARDWARE

The P2D2 module is already equipped with three I2C devices that use P57 and P56 for the I2C bus. The pullup is provided by the P2 since the drive strength is programmable but TAQOZ I2C drivers set this to 1k5 for high bus speeds.

To list all devices connected to the selected I2C bus use the **`lsi2c`** word which scan the i2c bus and list the devices that it discovers and also provide some details if available.

```
TAQOZ# lsi2c --- I2C DEVICES
 $36           P2D2 UB USB+SUPPORT  UUID:FF9BAD5834DEE811A8D742B1A51F80DA
 $A4           RV-3028 RTC  2021/01/17 SUN 17:35:09
 $C4           Si5351A CLOCK GEN ok
```

HOW TO

Accessing the I2C bus as a master is fairly straightforward since all you need to do is to issue the START condition, send out the address+rw, then read or write data. Most of the time you don't even have to worry about a STOP although that is recommended and necessary for some transactions.

Ok, to try this out let's read the seconds register of the RTC chip that was discovered at $A4. Most RTC chips store time in archaic BCD formats despite this not being a very friendly format for software. The RTC chip has a bank of addressable registers and to read a specific register it is necessary to issue a START+ADDRESS+W then the register address but because the chip is in write mode for the register address we need to restart the I2C transaction and put it in read mode to read the data. So we then issue a START+ADDRESS+R and then simply read the data and because the register address autoincrements with each access we can continue to read consecutive data bytes. There is one caveat to allow for and that is when that last data byte is read that we issue a negative acknowledgment otherwise the chip will have data queued that will interfere with the next time we talk to it.

In this example we will read the seconds register which is register 0 and this is followed by the minutes and hours. Only low level access words from TABLE 1 will be used so we can understand what it happening.

```
TAQOZ# I2C.START $A4 I2C! 0 I2C! I2C.START $A5 I2C! nakI2C@ I2C.STOP .BYTE --- 32 ok
```

The result was 32 in hex but this displays 2 BCD digits that mean 32 in decimal, not 48. BCD is a way fo limit 4-bits to encode a decimal digit only with the next 4-bits holding the more significant decimal digit.
Let's analyze what happened here though.

`I2C.START $A4 I2C!`	The RTC write address is transmitted after a START which will select the RTC chip
`0 I2C!`	The first byte after a write address is the register address (followed by write data)
`I2C.START $A5 I2C!`	But we want to read from the chip so we start again with the read bit set ($A4+1)
`nakI2C@ I2C.STOP`	This will read a byte from the chip while issuing a NAK before a stop
`.BYTE`	The byte is on the stack so to view the digits we print it as if it were a hex number.

Now that you are armed with the basic tools you can talk to any I2C device at all. All you need to know is the device address, the method of addressing its internal registers, what they are, and any gotchas.

EXAMPLE

A further example is given here of interfacing to a common PCF8574 8-bit I/O expander chip. There are many variations of this chip but this is the grand-daddy of them all.

Using the tools (the words etc) that we have we can construct a word for reading the 8-bit port, and another for writing the port.

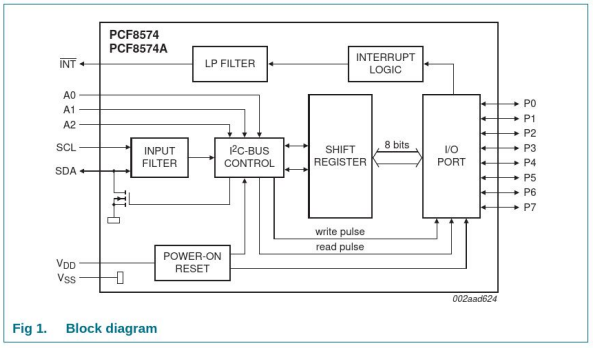

Fig 1. Block diagram

PCF8574 style I2C I/O

```
pub IO@ ( device -- data )        I2C.START 1+ I2C! I2C@ I2C.STOP ;

pub IO! ( data device -- )        I2C.START I2C! I2C! I2C.STOP ;
```

Yes, it really is that simple and that is all you need to talk to the chip. No need for a "library" or "driver".

FURTHER THOUGHTS

WHAT'S THIS ABOUT ACK and NAK?

TAQOZ COMPILES WHAT YOU TYPE

Looking at the IO! example above it is interesting to see how TAQOZ compiled this. Did it optimize anything? We can see inside the code like this:

```
TAQOZ# SEE IO!
1DBAE: pub IO!
07558: 4566  I2C.START
0755A: 4590  I2C!
0755C: 4590  I2C!
0755E: 1540  I2C>
07560: 005D  ;
       ( 10  bytes )
```

But that is what was typed in, there is no optimization, but then again, it is only 10 bytes of code. Isn't that optimal? :)

TABLE 1: I2C APPLICATION SPECIFIC WORDS

I2C.START		Generate a start condition (or a restart)
I2C.STOP		Generate a stop condition
I2C.RD?	(adr -- ack)	Generate a start and send the address as a read, return with ack
I2C.RD	(adr --)	Same as I2C.RD? but discard the ack
I2C.WR	(adr --)	Generate a start and send the address as a write
I2C!	(data --)	Write the data to the bus and count the ack in variable acks
I2C@	(-- data)	Read and ack the data from the bus
nakI2C@	(-- data)	Read and nak the data from the bus (last byte)
I2CPINS	(scl sda --)	Select the I/O to be used as the I2C bus (and initialize to a stop)
I2C.KHZ	(khz --)	Set the bus frequency
!I2C		Initialize the I2C bus pins with pullups and default to 400kHz