Q68 I²C Driver

This driver is a patched version of the Minerva I²C driver. Like the Minerva implementation, it is restricted to a single bus master, the Q68 itself. It does not support bus arbitration, or clock stretching. As the Q68's SCL line is write only, clock stretching is not possible.

Loading the driver

Just **LRESPR** the **I2C_Q68_BIN** file. If you want to call the I2C driver routine directly you will need to use **LBYTES** and **CALL** as you will need to know the start address of the driver.

The I2C interface can be accessed from SuperBASIC or machine code. The address of the Minerva **II_DRIVE** system vector routine can be accessed from 4 bytes after the start of the Q68 I2C driver.

II_DRIVE=PEEK_L(base+4)

where 'base' is the address that the driver was loaded to.

Using the Q68 with 5 volt I²C devices

The Q68 runs on a 3 volt supply, which will interface with 3 volt target I²C devices. But If you wish to use 5 volt target devices, The target I²C device should not have any pull up resistors on the SDA, or SCL lines to the 5 volt supply. However the Q68 does have clamp diodes to protect it's SDA and SCL lines from 5 volts. **Do not** try to connect the Q68's 3 volt supply on the Q68 I²C connector to the target device.
Better safe, than a blown up Q68.

The 3 volt signals from the Q68 should be large enough to trigger a '1' on the 5 volt Target device.

Note

This version (00.4) of the driver is the first working version. It has only been tested with the built in Real Time Clock on the Q68.
The Serial Clock currently runs at about 35KHz.
The **I2C_IO** function enters supervisor mode, but does not yet disable interrupts. So an interrupt may affect an I²C transmission.

The following is an edited copy of the I²C description from the Minerva manual.

II_DRIVE

Registers:

	Entry	**Exit**
D0		error code
D1		register result
D2	device parameter	smashed
Al	pointer to data	buffer updated
A3	pointer to command	buffer updated

Entry should be in supervisor mode, and preferably with interrupts disabled.

The I^2C driver is controlled by a byte stream contained in the (read-only) command buffer. Data to be written may come from either the command or data buffer. Results may be returned into D1 or the data buffer.

Four error codes are returned at present:-

ERR.FF	Implies that the hardware is not functioning.
ERR.NF	The addressed device is not present.
ERR.TE	An acknowledge was not received when it was expected.
ERR.BP	A bad command byte has been encountered.

During the interpretation of the command stream, D2 holds the number of the addressed device in its more significant word, and a *parameter* word in its less significant word.

Command stream bytes are as follows:

Parameter build byte:

7	6	5	4	3	2	1	0
0	seven parameter data bits						

The contents of the parameter word are shifted left seven bits, and this byte is *OR*ed into it. A contiguous sequence of three of these can be used to set up a full 16 bits of parameter. Only two uses of this are currently made. A single byte is used before a special command which is to copy it to the device group register, so we can change devices during a sequence. The other usage is to set up the byte count for a normal I/O command. This will make use of a 16-bit count, and may need anything from zero to three of these parameter build bytes. The parameter register is always cleared to zero after each of the normal i/o and special byte types has been processed.

Normal input/output byte:

7	6	5	4	3	2	1	0
1	0	S	R	B	P	A	0

The bits of this byte are essentially handled from left to right, to allow the most typical i/o sequence to be handled in its entirety.

S = 0	no START required (assumed SDA high and SCL low)
S = 1	send START and device (SDA/SCL assumed. high)
R = 0	write mode, or R = 1: read mode
B = 0	if R=0, write from control, or R=1 read to register
B = 1	write/read uses data buffer
P = 1	send STOP sequence
A = 1	send acknowledge on last read (R=1) byte

R=0 and A=1 is invalid, as is R=1, P=1 and A=1. Also bit 0 must be clear. If these conditions are not met, an err.bp is reported after processing all but the P bit.

The parameter value specifies the exact byte count for a write sequence, but on a read (R=1) sequence, it counts only those bytes to be acknowledged. If R=1 and A=0, the final byte with standard non-acknowledge is extra.

Write sequence data byte:

7	6	5	4	3	2	1	0
Data byte							

If a normal i/o byte requests writes from this control buffer, it will be immediately followed by the appropriate number of data bytes to be written.

Special i/o and control byte:

7	6	5	4	3	2	1	0
1	1	G	V	D	C	1	Q

Once again, the bits are handled from left to right, and these control all the exceptional cases we wish to cope with. Note that the SDA and SCL setting will occur simultaneously, hence to be valid, only one should differ from its currently known state. If V=O, the state will always be both ones before they are applied, so the combination of V, D and S all zero is always invalid.

G = 0	set device group addresses as 2 current parameter value
G = 1	assume device group is already in its register
V = 0	kill bus (assume NOTHING about bus, ensure in standard free state)
V = 1	assume the bus is valid, whatever state it is in
D = d	set SDA
C = c	set SCL
Q = 1	quit

Note that bit 1 is reserved and must be set, or an ERR.BP is reported after processing the G and V bits, but before setting the D/C combination.

The control buffer must finish up with a special command that has its quit (lsb) set. Normally this will be all ones, but where the bus is not being released between calls a value of $F3, keeping SDA and SOL low, will be typical.

The general rules for the bus go as follows:

Before a START+device, SDA should be high. After a START+device, SDA will be high and SCL will be low. For a read/write, SDA high and SCL low are required and are left the same. Before a STOP, SCL low is expected and both SDA and SCL will be left high. Before an initialise, SDA and SCL are irrelevant. When using the special" command, only one of SDA and SCL should be changed at one time. When it includes an initialise, that will preset them high.

I2C_IO

This SuperBASIC function uses the above routine almost directly:

res$=I2C_IO(cmnd$,res len[device[,parameter]])

The command string cmnd$ is as described above. The data buffer is effectively "write only", being the result of the function, and is thus not available as a data source; in addition, the anticipated length of the result must be supplied as the second parameter so that space can be allocated to store it.

So:

```
x$=CHR$ (164) &CHR$ (0)&CHR$(4)&CHR$(188)&CHR$ (255)
PRINT I2C_IO(x$,4,104,1)
```

will read four bytes from location 0 of the Q68's RTC.

CHR$(164) is a normal I/O byte, saying *write <parameter> bytes to the <device>;* the DS1374's device number (104) and the parameter (1) are set by the last two function parameters.

The one byte written is taken from the command stream, thus absorbing the CHR$(0) and setting the DS1374's address register to 0 for the TOD counter byte 0.

The CHR$(4) then sets the parameter to 4; this will be used as the count of bytes read and acknowledged, so the estimated result length Is four (the second function parameter).

CHR$(188) then says *read <parameter> bytes and send a <stop>*; the four bytes from address locations 0..3 are thus read.

CHR$(255) terminates the command stream, and the four bytes read are returned as a string.