

BBC B and Master I2C V3.1 Rom

© M.P.Barr 2018

```
22:25:08  Fri 23-11-18  19degC

BBC Computer

Acorn 1770 DFS

BASIC

>*HELP I2C

I2C 3.1B
  I2C
  I2CRESET
  I2CQUERY (Q)
  I2CTXB <addr> (<#nn>) <byte>(;)
  I2CTXD <addr> (<#nn>) <no.bytes>(;)
  I2CRXB <addr> (<#nn>) (A%-Z%)
  I2CRXD <addr> (<#nn>) <no.bytes>
  I2CSTOP
  TBRK
  TIME
  DATE
  TEMP
  NOW
  NOW$
  TSET <hh:mm:ss>
  DSET <day> <dd-mm-yy>

OS 1.20
>_
```

The I2C command set in the V3.1 rom shown as displayed by the **HELP I2C* Beeb OS command.

Introduction :

The Beeb I2C rom implements and exploits a Philips-NXP compliant I2C bus master capability in the Acorn Model B, B+, Master 128 and Electron computers. From version 3.0 onwards, for all computer models other than the Electron, the connection point for the I2C bus is the rear-facing Analogue Port connector through which the System 6522 at \$FE40 is utilised by the rom to implement the 2-wire I2C bus. For the Electron, the basic computer must be upgraded to provide a 6522 VIA and this is normally achieved through the addition of a Plus 1 expansion unit together with either an AP5 (legacy ACP or modern *Retro-Hardware*) or a modern *Stardot EUP* module. Since an expanded Electron still does not have the Analogue Port of its larger siblings, the Elk I2C bus is accessed through the User Port utilising Port B of the 6522 at \$FCB0. Refer to the *Supporting Information* section beginning on page 14 of this manual for detailed connection and cable details.

The primary focus of the version 3.1 release is to facilitate the easy addition of a properly integrated, real-time battery-backed clock and calendar to the BBC B, B+ and Electron range of computers. The I2C 3.1 rom command extensions implement bespoke support for readily available and inexpensive DS3231 I2C RTC modules (again, see *Supporting Information*) offering new time and date 'star' commands and dedicated OSWORD calls, together offering easy access to time and date information from both BASIC and assembler.

Significantly, the RTC user interface does not require the user to have any I2C knowledge (or interest!) whatsoever in order to take advantage of the clock and calendar functions. Indeed, other than seeing the I2C command set in the rom *HELP dialogue, the user will not be exposed to the I2C layer of the rom unless they specifically choose to use those advanced aspects for their own I2C projects.

General notes on the I2C functions :

1. To avoid confusion with the Acorn BBC Master series computer, unless explicitly stated otherwise, references to Master in these instructions are referring to an I2C Master bus controller, or often Microcontroller, as companion to I2C Slave devices.
2. A basic minimum knowledge of the I2C protocol is assumed as these instructions do not provide this information. There is of course a plethora of easy I2C tutorials available on the Internet.
3. I2CTXB and I2CRXB are commands processing a single 8-bit data byte only.
4. I2CTXD and I2CRXD are commands processing multiple 8-bit data bytes.

5. Command Parameter Definitions :

<addr> is the 7-bit, two-digit hex I2C bus address of the target Slave device. Valid range is 00 to 7F but for the I2CTXB command only, address FF is used to mean 'no address' – see TXB command later.

#<hh> is the optional, 8-bit, two-digit hex address of a specific register within the target Slave device. Not all Slave devices have multiple registers but many do. If specified, it must be preceded by the '#' symbol. Valid range 00 to FF.

<byte> is an 8-bit, two-digit hex data byte. Valid range 00 to FF.

5 (contd.)

<no. bytes> is a two-digit, hex number specified with the TXD and RXD commands to indicate the number of bytes to be transmitted or received. Valid range is 00 to FF but note that for both TXD and RXD, the number 00 is used to specify 256₁₀ or &100 bytes.

;
; is optionally specified as the trailing character for the TX commands only to indicate that an I2C **Stop** command is not to be issued after transmission of the specified data byte(s). It is conceptually similar to its meaning in the PRINT statement of many BASIC language implementations where it is used to indicate that a <CR/LF> is not to be issued allowing multiple print items (here data bytes) to be streamed consecutively.

6. All two-digit hex items in the above commands can be optionally replaced by one of BASIC's 26 system A%-Z% integer variables. For all the I2C rom commands except one, this will result in the integer variable being read-only accessed to provide the value of the address, register, byte value or number of bytes that the variable replaces in the command. The variable A-Z letter must be specified in upper case and two-characters must be given where the second must always be %. For example :

***I2CTXB 68 #02 A4** is functionally identical to ***I2CTXB A% #R% D%** assuming that A%=&68, R%=&02 and D%=&A4

The exception to an integer variable being read-only to provide a value to a function is the I2CRXB command where the data byte returned by the Slave device is written directly into the specified integer variable.

7. In all cases, where an integer variable is used in place of a two-digit hex value in a command string, only the least significant byte of the integer variable will be read to obtain the required eight-bit value. For example, if A%=47,574₁₀ or &B9D6, the I2C command will only use the least significant byte of the 4-byte integer variable (LS 8-bits) giving in the example an effective value of 214₁₀ or &D6. The individual commands do not range check a given integer variable for exceeding 255₁₀ or &FF. In the single case of the I2CRXB command where a returned value is written to the optionally specified integer variable, the upper three bytes of the integer variable are set to zero by the command and the least significant byte is set to the returned 8-bit value.
8. The I2C commands use the RS423 buffer at Page &0A00 as a transmit and receive data buffer. The single byte to be transmitted to the Slave in TXB and the single byte returned by the Slave in RXB, are always stored in address &0A00. In the case of RXB, the returned byte is additionally copied to a BASIC integer variable if specified. In the case of TXD and RXD, the multiple data bytes are pre-stored by the user from &0A00 to a possible maximum of &0AFF. For example :

***I2CRXD 68 00**

...would fill the whole of Page &0A with 256₁₀ bytes of data returned from the Slave device whose address is &68. Note that the example uses the special case of <no.bytes> = 00 to specify 256₁₀ bytes to be read from the Slave. Buffer management is entirely the responsibility of the user and will be reasonably straightforward as the number of bytes being transmitted or received is always known.

9. When a Master issues a command to a Slave, each 8-bit transfer will be acknowledged by the Slave with an ACK on the ninth clock bit. If this ACK is not received by the Master when it is expected (technically becoming a NACK response), this is, in nearly all situations, an error condition. After any of the four TX or RX commands, location &67 will be non-zero if such an error occurs and this can be used as required to validate I2C communications.

Typically, an error during a TX command will be flagged as &67=1 and an error during a RX command will be flagged as &67=2. The detail is probably unimportant in most cases and so in BASIC, we can simply use **IF ?&67<>0 THEN <error response>** or in assembler, **LDA &67:BNE <error response>**

Generally, once a given configuration has been tested, it is unlikely to be necessary for software to continually poll &67 but if some other error condition arises within a program, &67<>0 can be used to determine if the problem is related to Master-Slave I2C comms.

10. Whilst the basic premise of I2C is that only a Master can control the clock line (SCL), a Slave device is permitted to hold the clock line low after a transfer is complete if the device is busy processing the transmitted data – this is known as Clock Stretching. The Master will not issue any further commands on the bus whilst the latter is being reserved in this way. A good example of the use of Clock Stretching would be an eeprom with a slow write cycle completion time.

A potential consequence of implementing Clock Stretching tolerance in a Master is that under fault conditions, the Master could potentially loop indefinitely waiting for the clock line to be released. Therefore, to allow the user the ability to manually abort a process that has permanently hung in this way, the **<Escape>** key is polled under these circumstances and can be used to abort the current command.

11. The I2C rom makes use of zero page ram locations **&60 - &6F**. Locations **&6D - &6F** are used during *HELP I2C output and locations &60 - &6F are used transiently during command execution and processing. None of the locations are required for persistent data whilst the I2C rom commands are dormant.

I2C Rom Commands :

***I2C**

No parameters are passed with this command but two results are returned indirectly to (a), confirm the presence of the I2C rom and to (b), identify the sideways rom socket number (or slot) into which the rom is fitted. When the I2C rom claims and responds to this command, it will write the value &2C into location &66 and set location &65 to it's own sideways rom slot id, i.e. to a number in the range of zero to fifteen. The '&2C' transponder response can be usefully used from either BASIC or assembler and the rom slot id is particularly relevant to assembler programs allowing the I2C rom to be paged-in for direct (non-CLI) calls to the individual commands as detailed later in these instructions.

***I2CRESET**

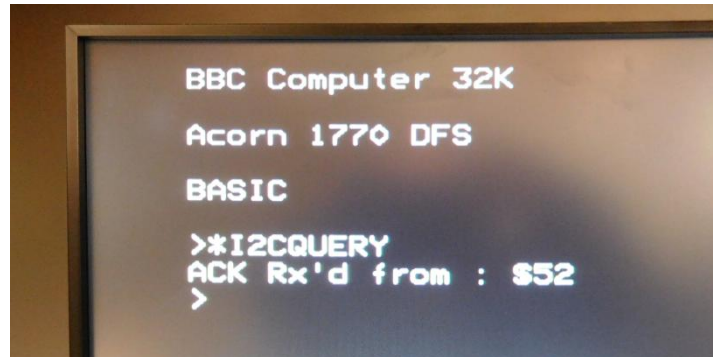
No parameters passed or returned, this command is used to resynchronise any devices on the bus that may be 'hung' if a transmit or receive has been interrupted mid-transfer. The term '*reset*' has been coined by I2C device manufacturers but it is not a hardware reset as such, (we only have a data and a clock line), it is accomplished by the Master issuing nine consecutive clock pulses followed by an I2C Stop to 'flush' any outstanding transfers from devices that may be currently expecting to complete a transaction. In the Acorn environment, you could theoretically get a slave device hang if <Break> were pressed mid-transfer but I haven't seen many, if any, occurrences of this. It certainly wouldn't hurt to include this command at the beginning of any program for robustness but it doesn't need to be used routinely thereafter within a given session.

***I2CQUERY (Q)**

Every I2C Slave device you obtain will have a preset hard-wired address (<addr> in the rom commands) and many additionally have the lower two or three address bits (A0-A3) broken out onto links, pads and/or bias resistors such that some conflict-control is possible. You should know the given device address from its documentation or you can consult the Philips allocation table for clues but in the event that you are unsure or if you have multiple devices connected in a daisy-chain (which is another great feature of I2C), the I2C rom provides this query command to report the address of any devices that can be detected on the I2C bus. Essentially, the command attempts to consecutively wake-up every legal device address and looks for an acknowledge (ACK) from that address. If a given device is present at the current address, it should respond and in which case, the wake-up is terminated, the address of the responding device is output to the display and the next address is interrogated. In addition, the byte address of any responders will be written to the I2C buffer at &A00 onwards, the list being terminated with &FF. Thus, if no devices are detected, on return from I2CQUERY, location &A00 will contain &FF. The 'legal' slave device address range iaw accepted I2C industry standard protocols is &08 to &77 inclusive.

(contd.)

By default, the command will output a comma-delimited list of all the hex addresses received from responding devices. In the example output shown below, a single I2C slave device is attached to the I2C bus, here a Wii Nunchuck :



There is a single optional switch available for use with the I2CQUERY command, 'Q', which will suppress the screen output where this is not wanted. When this switch is used, the responding device list at &A00 onwards will still be populated.

Note that some devices will occasionally report two or three consecutive 'phantom' addresses and in which case this will usually be obvious and the first report will likely be the genuine address. This is not a limitation of the Beeb I2C software but of certain I2C device implementations.

***I2CTXB <addr> (#<hh>) <byte>(:)**

This command is used to transmit a single 8-bit byte to a Slave device. A target register within the device can optionally be specified. The command will, by default, finish the transmission by issuing an I2C Stop command but if this is not wanted, a trailing ';' can be used immediately after the data byte two-hex digits or equivalent integer variable. In this case, just the eight bits are written and the command is left 'open' for further data transmission.

For example, when addressing an I2C eeprom device, the required write sequence may be a contiguous three bytes of <address-hi><address-lo><byte> and in which case you could use three TXB commands specifying a trailing ';' after the first two commands so as not to prematurely terminate the command sequence with a Stop.

For TXB only, the special value of FF can be specified for <addr> to allow an 8-bit data byte only to be transmitted on the bus to a slave that is already mid-transaction and listening. Thus, in the command :

***I2CTXB FF <byte>**

...the out-of-range I2C device address will be ignored and the command will only transmit the eight bits of <byte> on the bus without an I2C Start command or a device address.

***I2CTXD <addr> (#<hh>) <no.bytes>(:)**

This command is used to transmit a contiguous sequence of data bytes to a Slave device where the data bytes to be transmitted are first written by the user to the I2C buffer beginning at &0A00. A target start register within the device can be optionally specified (although this is probably unusual for a multi-byte write) and the device would then need to auto-increment the register number or write addresses within the target register. The command will, by default, finish the transmission by issuing an I2C Stop command but if this is not wanted, a trailing ‘;’ can be used immediately after the <no.bytes> two-hex digits or equivalent integer variable. In this case, just the eight bits are written and the command is left ‘open’ for further data transmission.

In the previous TXB example of writing a byte to an eeprom with a sequence of three single byte transmissions, the same sequence of <address-hi><address-lo><byte> could be achieved using TXD by pre-storing the three bytes in &A00-&A02 and issuing a single TXD command specifying three bytes to transmit.

***I2CRXB <addr> (#<hh>) (A%-Z%)**

This command is used to read a single byte from a Slave device optionally specifying a specific source register within the device. A single RXB byte will always be written to the I2C buffer address of &0A00 and additionally to a BASIC integer variable if specified.

***I2CRXD <addr> (#<hh>) <no.bytes>**

This command is used to read a contiguous sequence of bytes from a Slave device optionally specifying a start register and the device would then auto-increment the register number or read addresses within the target register. The received bytes are written to the I2C buffer page from &0A00 to &0AFF.

***I2CSTOP**

By default, all the TX and RX commands issue an I2C Stop on completion but in the case of the TX commands, we can override this Stop using the trailing ‘;’ switch. This option can be exploited for use with contiguous writes when for certain device command streams, there is a valid I2C condition known as a RE-START where a new Start can be sent in lieu of a Stop/Start sequence.

For certain devices however, or if no further new commands are to be issued, it is useful for the I2C system designer (the Beeb programmer, you!) to be able to explicitly issue an I2C Stop and this can be achieved with this parameter-free command.

***TBRK**

Toggles an internal non-volatile '*Time-On-Break*' flag which determines, at power-on or any type of *<Break>* reset, whether current time, date and temperature information is printed at the top of the screen before any system dialogue. This is a toggle (flip-flop) command (alternates on/off) and the current state is reported after each use. The flag is physically maintained in the clock module NVM and is therefore persistent through power cycles as long as the RTC module battery is good.

The TBRK active display and use of the command are shown in the example screenshot below :

```

17:28:45  Fri 09-11-18  23degC

BBC Computer

Acorn 1770 DFS

BASIC

>*TBRK
Off
>
>*TBRK
On
>

```

The following four commands, *TIME, *DATE, *TEMP and *NOW, are all principally intended to be used in immediate mode (from the command prompt) to individually display the primary three RTC data items of time, date and temperature or, in the case of *NOW, to display all three together. The commands are not intended to allow BASIC or assembler programming access to the RTC data, the latter being achieved through the use of the new OSWORD 14 (&0E) commands and *NOW\$ as detailed further on in the manual.

***TIME**

Displays the current time in the form of ***hh:mm:ss***

***DATE**

Displays the current date in the form of ***<day> dd-mm-yy***

***TEMP**

Displays the current temperature (as measured at the RTC module) in the form of ***TTdegC***

***NOW**

Displays the current time, date and temperature in the form of the example below :

21:14:35 Fri 19-10-18 21degC

***NOW\$**

This is a 'silent' version of *NOW and is intended for use within BASIC programs enabling easy time, date and temperature manipulation via standard string handling functions. An ASCII text RTC data string, similar to that produced by *NOW, is generated at the I2C buffer address of &0A00 where it can be quickly accessed using the BASIC string indirection operator, *<string name>\$ = \$<address>*

The 25-character ASCII string is created at &0A00-&0A18 in the form of the example below :

21:14:35<spc>Fri<spc>19-10-18<spc>21<cr> (where <spc>=32₁₀ and <cr>=13₁₀)

The code below shows how the RTC data string can be simply accessed from BASIC :

```
*NOW$  
TD$ = $&A00  
TIME$ = LEFT$(TD$,8)  
DATE$ = MID$(TD$,14,8)
```

...where TIME\$ would become "21:14:35" and DATE\$ would become "19-10-18" (quote marks shown for clarity only.) The individual string data items can be converted to true numeric values using the BASIC *<num>=VAL(<string name>\$)* function.

***TSET <hh:mm:ss>**

Sets the time to the supplied values. The fields and colon delimiters must be exactly as shown using 24hr format with all leading zeroes being required. Any errors in syntax will result in the required format being displayed as per *HELP I2C. Correct entry will result in the time being echoed as per *TIME command.

***DSET <day> <dd-mm-yy>**

Sets the date to the supplied values. The fields and hyphen delimiters must be exactly as shown with all leading zeroes being required. Any errors will result in the required format being displayed as per *HELP I2C. Correct entry will result in the date being echoed as per *DATE command.

(Note that *<day>* is the first three characters of the day of the week. e.g. **Fri** and can be entered in upper or lower case.)

Calling the RTC commands from Assembler

For assembler-programming, the rom implements OSWORD 14₁₀ (&0E) Type 1 exactly as Acorn-documented and as appears on the Master (RTC-equipped) range of computers. Additionally, a new OSWORD 14₁₀ sub-type, number 4, is implemented by the rom and this performs the equivalent of *NOW\$ (as detailed above) but the 25-character string is placed at the address pointed to by the OSWORD entry XY (little-endian).

The screenshot below shows example usage of OSWORD 14₁₀ Type 1 and Type 4 as implemented by this rom and in the presence of a recommended DS3231 RTC module. (Although shown as BASIC immediate mode commands, assembler programmers will readily recognise the equivalence in assembler/machine code.

```

17:28:45  Thu 09-11-18  23degC

BBC Computer

Acorn 1770 DFS

BASIC

>A%=14:X%=0:Y%=&09: ?&0900=1
>CALL&FFF1
>P.÷!&900
      6091118
>P.÷!&0904
      452817
>
>A%=14:X%=0:Y%=&09: ?&0900=4
>CALL&FFF1
>P.$&0900
17:28:45  Fri 09-11-18  23
>_

```

Note that the ASCII/BCD character string produced by *NOW\$ and by OSWORD 14₁₀ Type 4, includes the current temperature as the trailing two-character number (representing Degrees Centigrade with a resolution of 1 degree) but it *does not* include the “degC” as displayed by the *TEMP and *NOW commands. This is intentional as the choice of temperature units annotation is regarded as the remit of the particular user application software accessing the string data. (e.g. as provided by the *TEMP and *NOW commands.)

Calling the I2C commands from Assembler

The I2C rom implements new *commands and these can be called from within assembler programs using the standard OSCLI method. Longer term, in later versions of the I2C rom, the commands may be implemented as extended OSBYTE calls. During development, all release versions of the rom have a fixed call table implemented at the top of the rom and this allows any of the I2C rom star commands to be called via a vector address whenever the I2C rom is paged-in at &8000. There is also an I2C rom-present star command, ***I2C**, which, when called (via OSCLI in assembly), will set location &66 to &2C and location &65 to the zero-to-fifteen rom slot number in which the I2C rom is fitted or loaded. The latter slot id can thus be saved and used by assembler software to page-in the I2C rom after which the required command can be called using the fixed table addresses after setting up any required pre-call parameters as detailed below. These call addresses will not change throughout the development life-cycle of the I2C rom and therefore any software exploiting the I2C rom is immune to rom version iterations.

The following information details the I2C rom call address for each command together with any preset data entry criteria. The full command descriptions are given in the preceding section.

***I2C**

It is not sensible to call this command directly because in order to do so, the I2C rom needs to be first paged-in and in which case, the slot id must already be known! Therefore, from assembler, this command should be called using the standard OSCLI method and the returned slot id can then be subsequently used to page-in the I2C rom as required to allow direct calling of the other I2C commands.

To absolutely guarantee the integrity of the returned rom-present indications, it is advisable (though not essential) to set locations &65 and &66 to zero before calling.

i2c rom slot number	&65	Set to 0
i2c rom present	&66	Set to 0

Although direct calling of this command is of limited value to the end-user as discussed above, for consistency purposes the *I2C command does have an entry in the direct call address table :

JSR &BFF8

***I2CRESET**

No preset data

JSR &BF80

***I2CQUERY**

suppress output	&6E	Set to non-zero to prevent device list being displayed
-----------------	-----	--

JSR &BF88

***I2CTXB <addr> (#<hh>) <byte>(:)**

<addr>	&68	&00-&7F, target device address - &FF = none, Tx data byte only
<hh>	&69	&00-&FF, register number - see also &6C for validity flag
<byte>	&6A	&00-&FF, the byte to be transmitted
register valid	&6C	Set to zero if register not valid else non-zero if &69 valid
;	&6D	Set non-zero to inhibit post-Tx Stop else set to default zero

JSR &BF90

On return, check bit 0 clear in &67 else a Slave ACK response error occurred

***I2CTXD <addr> (#<hh>) <no.bytes>(:)**

<addr>	&68	&00-&7F, target device address
<hh>	&69	&00-&FF, register number - see also &6C for validity flag
<no.bytes>	&6A	&00-&FF, number of bytes to be transmitted, zero = 256 ₁₀
register valid	&6C	Set to zero if register not valid else non-zero if &69 valid
;	&6D	Set non-zero to inhibit post-Tx Stop else set to default zero

JSR &BF98

On return, check bit 0 clear in &67 else a Slave ACK response error occurred

***I2CRXB <addr> (#<hh>) (A%-Z%)**

<addr>	&68	&00-&7F, target device address
<hh>	&69	&00-&FF, register number - see also &6C for validity flag
register valid	&6C	Set to zero if register not valid else non-zero if &69 valid
BASIC int var (1)	&6A	Set to zero if int var% not required else set to 1 and...
BASIC int var (2)	&6E	...set &6E to hold the &0400 offset to the target int var% i.e. A%=4, B%=8, C%=12 etc.

JSR &BFA0

On return, check bit 1 clear in &67 else a Slave ACK response error occurred

***I2CRXD <addr> (#<hh>) <no.bytes>**

<addr>	&68	&00-&7F, target device address
<hh>	&69	&00-&FF, register number - see also &6C for validity flag
<no.bytes>	&6A	&00-&FF, number of bytes to receive, zero = 256 ₁₀
register valid	&6C	Set to zero if register not valid else non-zero if &69 valid

JSR &BFA8

On return, check bit 1 clear in &67 else a Slave ACK response error occurred

***I2CSTOP**

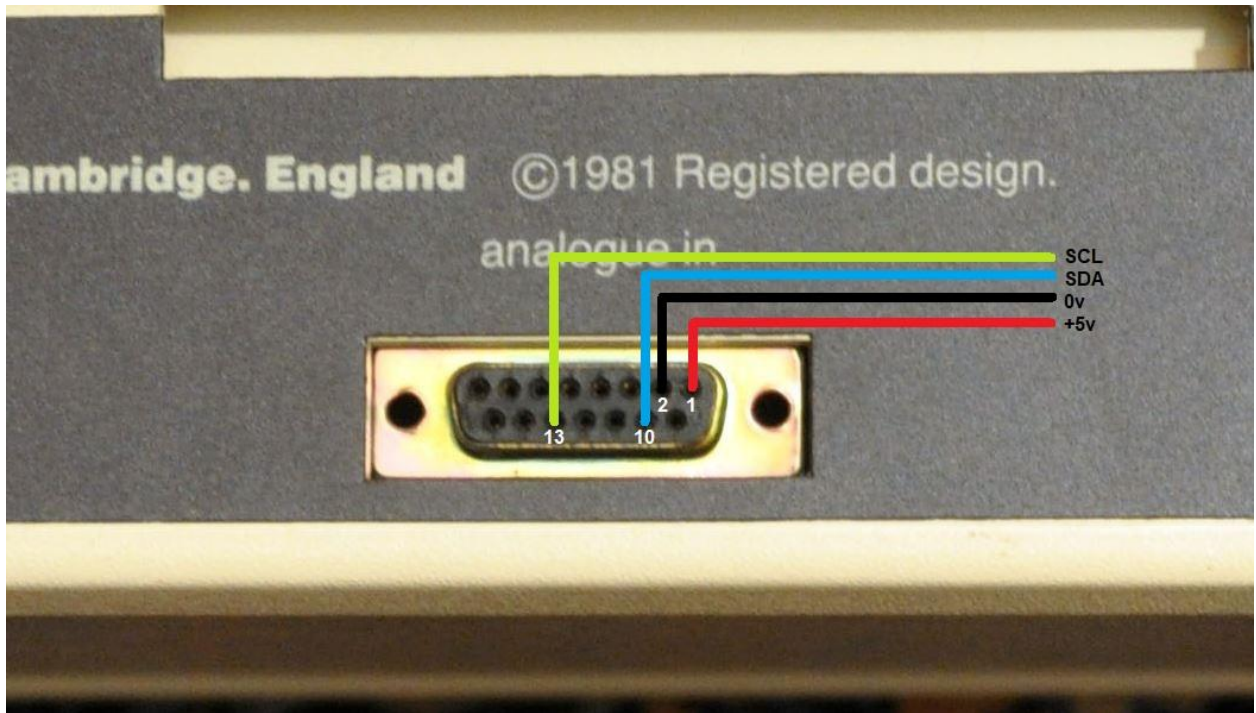
No preset data

JSR &BFB0

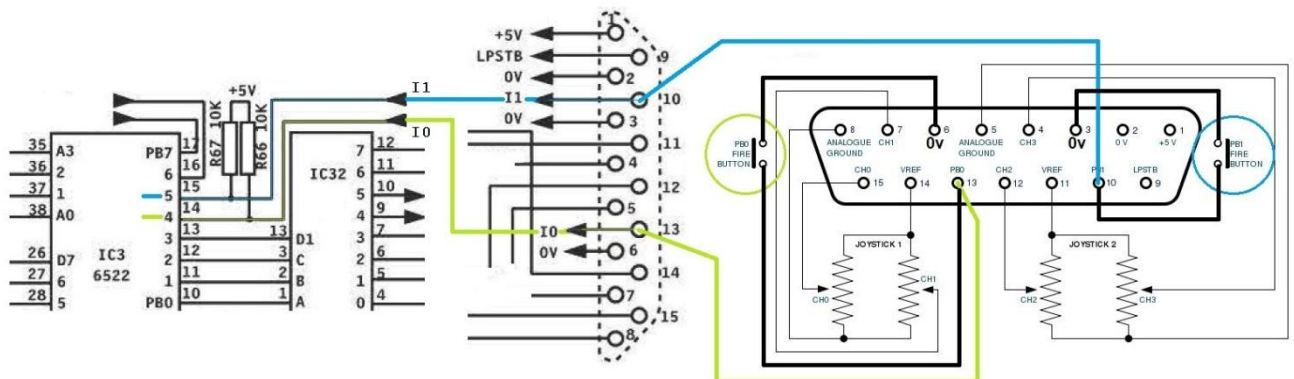
Supporting information :

Specific to the BBC B, B+ and Master 128 computers

a) I2C Bus Connection / Cable Details



b) Schematic of Analogue Port I2C Bus



c) Example non-pass-through Analogue Port I2C cable



This cable has been recycled from a desktop PC chassis interconnect loom since the small headers are typically the same pitch as most widely available I2C modules. Here, the header (white) is a five-pin item but only four ways are electrically connected and the Beeb Analogue Port connector (blue) is a 15-way IDC D-Type plug.

d) Example pass-through Analogue Port I2C cable

In order to allow joysticks to be connected simultaneously with I2C devices, an Analogue Port pass-through cable can be assembled using a 15-way IDC plug and socket connected by a short length of 15-way ribbon cable. The latter cable would be extended out beyond one of the D-Type connectors to the desired length forming the I2C connection point noting that only 4 cores of the extended section are required and as shown in the pictures at (a) and (b) above, these cores are 1 (+5v), 2 (0v), 10 (SDA) and 13 (SCL).



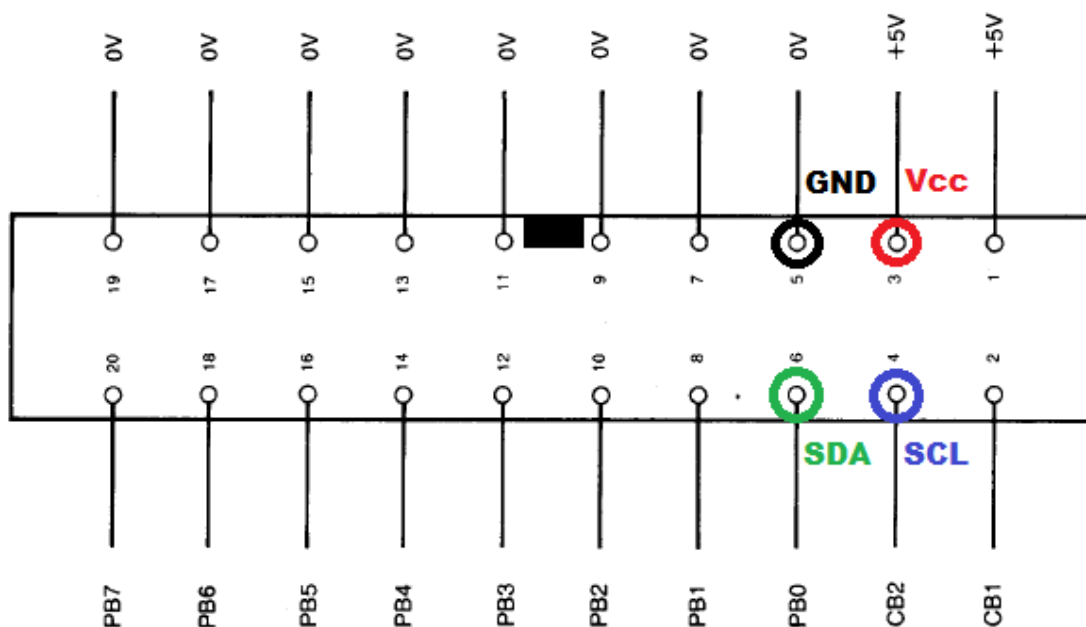
e) Joysticks

V3 of the rom represents a significant change in the Beeb I2C implementation due to the physical bus migrating from the User Port, as utilised in versions 1 & 2 of the system, to the Beeb and Master 128 Analogue Port. Specifically, the I2C active bus lines, SCL and SDA, directly piggy-back the joystick fire-button inputs, PB0 and PB1, at pins 13 and 10 respectively of the D-Type socket Analogue Port connector. Due to the electrical characteristics of I2C, the dual-purposing of these lines in this way presents no problems but it is recommended that the joystick fire-buttons are not operated whilst running programs that access I2C devices as this *could* cause transient data corruption of the I2C bus.

Specific to the Acorn Electron computer

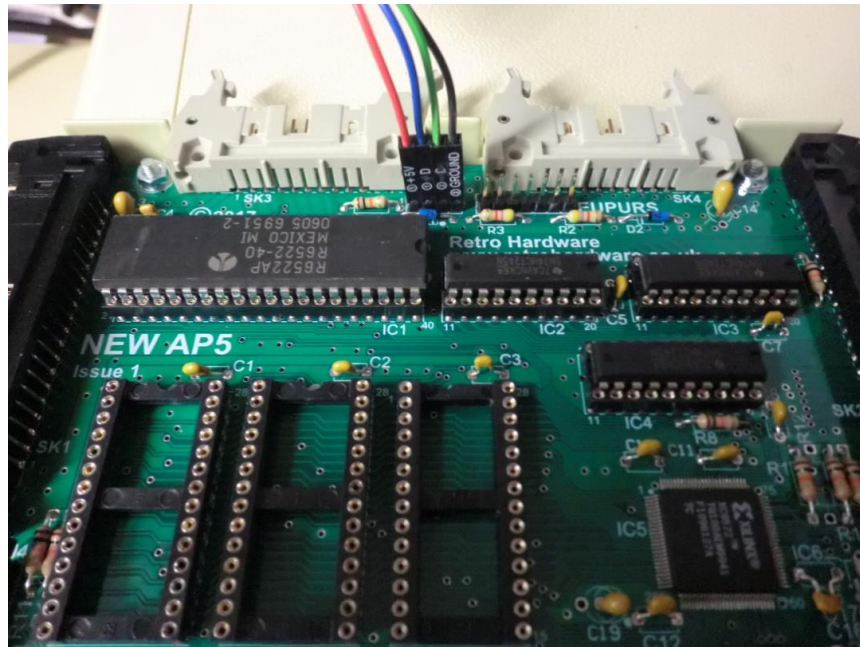
As discussed in the introduction, an I2C-capable Electron will necessarily have been expanded with a Plus 1 and either a Stardot EUP or a Retro-Hardware AP5 where in both cases, the User Port physical interface is a 20-way IDC connector as used on the BBC Micro range of computers. On these Electron expansions, the relevant connector idents are PL2 on an EUP and SK3 on an AP5.

The diagram below shows the required PL2/SK3 User Port connector pin numbers and their associated I2C signal names :

I2C User Port Connections

I2C Rom Software for Acorn Computers – Issue 3.1

In the case of the RH AP5, in addition to SK3 there is also a dedicated row of four header pins allocated to I2C as shown in the photograph below. This header plug arrangement is annotated **PL2** & **IIC** and the pins can be used to connect any I2C bus devices including the RTC modules for which the I2C v3.1 rom provides support. (Note that the PL2 header pins are electrically connected to PB0 and CB2 as per User Port connector PL2.)




The photograph below shows a DS3231 I2C RTC module connected to PL2 of an EUP which is itself plugged into an Electron fitted with an Acorn Plus 1.



RTC modules

Ideally, you should obtain one of the module types shown first below as these also have a 32k EEPROM on-board and this storage is to be exploited in future versions of the I2C rom. It is necessary to remove one SMD resistor (easily done, no particular expertise required) to disable the charging circuit and to allow the module to be used with a non-rechargeable Lithium coin cell. Details of this small modification are available in the Stardot forum “**I2C 4 U**” thread and the relevant post(s) can be linked to here :

<https://stardot.org.uk/forums/viewtopic.php?f=3&t=10966&start=210#p215427>



Free P&P

Click to view larger image and other views

DS3231 ZS042 AT24C32 IIC Precision Real Time Clock RTC Memory Module for Arduino

★★★★★ Be the first to write a review.

Condition: **New**

Quantity: 4 available

£2.53

Buy it now

Add to basket

[Add to Watch list](#)

Free postage 30-day returns Posts from United Kingdom

Postage: Economy Delivery | [See details](#)
Item location: London, United Kingdom
Posts to: [See exclusions](#)

Delivery: Varies

Payments: **PayPal** Processed by PayPal | [See payment information](#)

Returns: 30 days refund, buyer pays return postage | [See details](#)

Shop with confidence

eBay Money Back Guarantee
Get the item you ordered or your money back. [Learn more](#)

Seller information

97.6% Positive Feedback


[Save this seller](#)

[Contact seller](#)

[Visit Shop](#)

Registered as a business seller

Other types of I2C DS3231 module are also fine for our purposes, such as the one shown below, with the only caveat being that if a given module doesn't have the on-board EEPROM, you might miss out on further I2C rom feature developments.



Free P&P

Click to view larger image and other views

HW-022-1DS3231 Raspberry ds3231 Clock Module DS3231

Condition: **New**

£1.95

Buy it now

Add to basket

[Add to Watch list](#)

A seller you've bought from Free postage 30-day returns

Postage: **Free** Economy Delivery | [See details](#)
Item location: Manchester, United Kingdom
Posts to: United Kingdom | [See exclusions](#)

Delivery: Estimated by Wed. 31 Oct. 🌐

Payments: **PayPal** Processed by PayPal | [See payment information](#)

Returns: 30 days refund, buyer pays return postage | [See details](#)

Shop with confidence

eBay Money Back Guarantee
Get the item you ordered or your money back. [Learn more](#)

Seller information

97.3% Positive Feedback

[Save this seller](#)

[Contact seller](#)

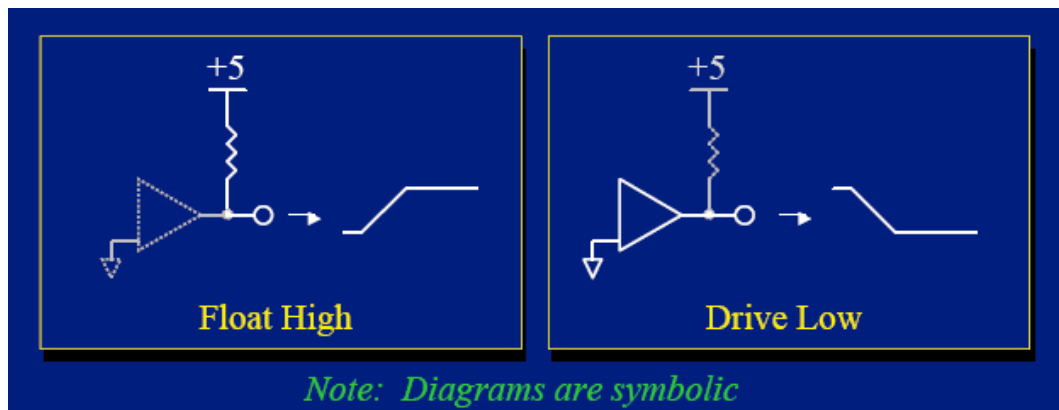
[Visit Shop](#)

Registered as a business seller

General I2C information for all computers

a) Pull-Up resistors

The I2C bus is unusual in that connected devices must only have the capability to drive the bus clock or data lines to a logic low (0v) level. In order for the lines to achieve a logic high (+5v) level, all connected devices must adopt a high impedance state allowing the mandatory Vcc pull-up resistors to float the lines high. This concept of operation is shown diagrammatically below :



The I2C specification recommends that the externally connected resistors from the SCL (clock) and SDA (data) lines to +5v should have values ranging from approximately 1K ohms to 4.7K ohms depending on the typical bus speed of a given application. For our BBC Micro I2C implementation, we are using a 6522 VIA chip whose I/O driver stages are capable of adopting logic levels of high (~5v), low (~0v) or a high impedance state when listening or when disabled. The same driver stages also have built-in pull-up resistors of approximately 1K ohm and therefore, with these collective electrical characteristics, the 6522 device is suitable for use as an I2C Master or Slave device.

Given the above 6522 characteristics and in relation to this Beeb I2C implementation, the key point to be aware of is that no external resistors are required and none should be used. Most commercially available I2C slave modules do not have any pull-up resistors fitted but if a particular I2C slave module does have optional link-selectable pull-up resistors then these should always be disabled and if possible, any hard-wired resistors should ideally be removed.

b) I2C bus Speed

Version 1 of the rom elicited a bus speed of approximately 13 KHz but versions 2 and 3 of the rom have been significantly optimised and elicit a bus speed of around 20-23 KHz depending on the operation being performed.

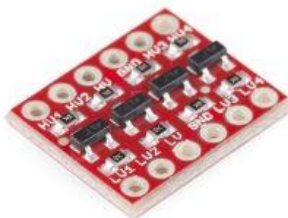
c) 5v to 3.3v level shifting

There are an ever-increasing number of I2C-based devices, modules and gadgets available online through eBay and bespoke electronics sellers and in most cases, these items are designed to operate with a 5v power supply. The Beeb Analogue Port I2C cable specified in section (a) above picks up +5v and 0v on pins 1 and 2 of the port connector allowing 5v I2C modules to be connected directly to the Beeb using only four wires. However, there are also some modules and devices available that are designed to operate in a lower 3.3v environment and in this case, we cannot connect these units directly to the Beeb.

There are two aspects to correctly using 3.3v devices, the first being the provision of a power supply for the device or module and the second being to bi-directionally condition the I2C clock (SCL) and data (SDA) lines such that they are 0-5v at the Beeb and 0-3.3v at the I2C module. Fortunately, there are cheap, ready-made modules that provide this voltage conditioning and these are generically known as level shifters. These modules are placed in series with the Beeb and I2C device and are connected to both 5v, which is available in the Beeb cable, and importantly, also to a 3.3v supply which can in most cases be simply obtained by fitting a low-power solid-state regulator onto the level shifter board.

The internet is packed with level shifting how-to information so I won't go into more detail here but the key message is, always be careful to check the voltage requirements of a given I2C module.

An example level shifter is shown below...



Logic Level Converter - Bi-Directional

Description: If you've ever tried to connect a 3.3V device to a 5V system, you know what a challenge it can be. The SparkFun bi-directional logic level converter is a small device that safely steps down 5V signals to 3.3V AND steps up 3.3V to 5V at the same time. This level converter also works with 2.8V and 1.8V devices. What really separates this Logic level converter from our previous versions is that you can successfully set your high and low voltages and step up and down between them safely on the same channel. Each level converter has the capability of converting 4 pins on the high side to 4 pins on the low side with two inputs and two outputs provided for each side.

The level converter is very easy to use. The board needs to be powered from the two voltages sources (high voltage and low voltage) that your system is using. High voltage (5V for example) to the 'HV' pin, low voltage (3.3V for example) to 'LV', and ground from the system to the 'GND' pin.

Dimensions: 0.63 x 0.52" (16.05 x 13.33mm)

d) I2C rom usage Examples

The following usage examples are taken from the Stardot I2C thread.

DS3231 RTC module with 24C32 eeprom

Generally, always begin by obtaining and reading the datasheet paying particular attention to the address map and register function descriptions.

DS3231

Extremely Accurate I²C-Integrated
RTC/TCXO/Crystal

ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/24	AM/PM 20 Hour	10 Hour	Hour				Hours	1–12 + AM/PM 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/24	AM/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + AM/PM 00–23
0Ah	A1M4	DY/DT	10 Date		Day				Alarm 1 Day	1–7
					Date				Alarm 1 Date	1–31
0Bh	A2M2	10 Minutes			Minutes				Alarm 2 Minutes	00–59
0Ch	A2M3	12/24	AM/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + AM/PM 00–23
0Dh	A2M4	DY/DT	10 Date		Day				Alarm 2 Day	1–7
					Date				Alarm 2 Date	1–31
0Eh	EOSC	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

Figure 1. Timekeeping Registers

Note: Unless otherwise specified, the registers' state is not defined when power is first applied.

The following short program puts HH:MM:SS clock in the middle of the computer display :

```

10 CLS
20 REPEAT
30 *I2CRXB 68 #00 S%
40 *I2CRXB 68 #01 M%
50 *I2CRXB 68 #02 H%
60 H%=H% AND &3F
70 PRINT TAB(15,12);~H%;":";~M%;":";~S%
80 UNTIL FALSE

```

Note that this RTC module as supplied has an I2C device address of &68

The only I2C Rom command used in the short RTC time display program is *I2CRXB which, as described earlier in the manual, requests a single 8-bit byte from a slave device with the option of specifying a particular source register within that device. The DS3231 datasheet tells us that we can read seconds from register 00, minutes from register 01 and hours from register 02 where all are pre-scaled in hex-coded decimal and are pre-ranged to 00-59 for the seconds and minutes and 00-23 for the hours (the latter when the default 24hr format is selected.)

So, the three commands...

```
30 *I2CRXB 68 #00 S%
40 *I2CRXB 68 #01 M%
50 *I2CRXB 68 #02 H%
```

...are all addressing device &68 (the module default as supplied) and each of the three commands accesses a different register as specified by the two-digit hex number preceded by the # symbol. (I changed to a # after the original 'R' in the v0.1 rom to designate a register number because after I added the BASIC integer %variable facility, it was misleading to have say RS% when #S% looks more intuitive.)

The only additional 'processing' carried out on the data is to apply a bit mask to the hours value H% because the datasheet tells us that in 24hr format (which the device defaults to), the hours are contained in register 02 bits 0-5 only so the H%=H% AND &3F of line 60 zeroes the top two bits of H% (&3F = binary 0011 1111).

Now, I have tried to design a lot of flexibility into the I2C rom commands and to highlight this, here are some possible variations of the program exploiting some of this flexibility...

```
10 CLS
15 T%=&68
20 REPEAT
30 *I2CRXB T% #00 S%
40 *I2CRXB T% #01 M%
50 *I2CRXB T% #02 H%
60 H%=H% AND &3F
70 PRINT TAB(15,12);~H%;":";~M%;":";~S%
80 UNTIL FALSE
```

The above is an example of how we can replace any of the I2C command two-digit hex parameters with a BASIC integer %variable. Here, line 15 sets T% to the device address of &68 and T% is then used in the command instead of an explicit hex number.

```
10 CLS
20 REPEAT
30 *I2CRXB 68 #00
35 S%=?&0A00
40 *I2CRXB 68 #01
45 M%=?&0A00
50 *I2CRXB 68 #02
55 H%=?&0A00
60 H%=H% AND &3F
70 PRINT TAB(15,12);~H%;":";~M%;":";~S%
80 UNTIL FALSE
```

The above shows that the I2C RX commands do not need to pass returned device bytes directly to a BASIC variable and that the optional destination parameter can therefore be omitted. The alternative method exploits the fact that an RXB byte is always stored at address &0A00 from where the value can be read and assigned as required post-command.

```
10 CLS
20 REPEAT
30  *I2CRXD 68 #00 03
40  S%=?&A00:M%=?&A01:H%=?&A02
60  H%=H% AND &3F
70  PRINT TAB(15,12);~H%;": ";~M%;": ";~S%
80 UNTIL FALSE
```

The above shows how the RXD (note D) command (receive multiple data bytes) can be used to replace a sequence of individual RXB commands provided the source registers of interest are sequential. Thus, in our example, the three RXB commands have been replaced by a single RXD command with a start register of 00 and a total of 3 bytes to receive. In the case of RXD commands, the received bytes are stored sequentially in memory page &0A, the designated I2C buffer, and in line 40 the bytes are read and assigned to appropriate BASIC integer %variables.

Note that when using the RXD command, the multiple byte transfer is achieved by the rom command transparently (to the user) compiling and issuing a series of contiguous single byte requests to the device. In receiving such a series of requests, the device will auto-increment either the source register number or the memory address in the case of storage devices. Here, we specify register 00 as the start register and this is sent to the device prior to the first read and the three subsequent read requests then cause the device to automatically provide the contents of registers 00, 01 and 02. If a register or memory address is not specified, most devices will respond to a read request by sending a byte from the most recently read register or address value incremented by one.

Moving on to the 24C32 eeprom which is also resident on the subject module...

In the thread video, there was a simple test program writing to and reading back from an I2C serial eeprom where the device in question is a 24C32 32k x 8 bit serial eeprom chip. The eeprom is resident on the RTC module we saw in the video to presumably provide bonus functionality and the device is in fact very similar to the AT28C256 we've already put hard to work in our Beebs. The AT28C256 differs in that it is a 28-pin parallel device and to access it, we have some high order address bus decoding which condenses down to a unique chip select for the specific device we want to talk to, some low order address lines to select a particular location within the chip and finally, some variation of a RnW line to indicate a read or write operation. In the case of a serial random-access memory device, we have to effectively do all these same operations but now sequentially using the I2C bus. This typically involves sending a unique device identifier to wake up the target chip, an indication of Read or Write, an address Hi byte, an address Lo byte and finally, we have to send data to the chip when writing or receive data from the chip when reading.

Pictorially, the write process for the 24C32 looks like this...

FIGURE 4-1: BYTE WRITE

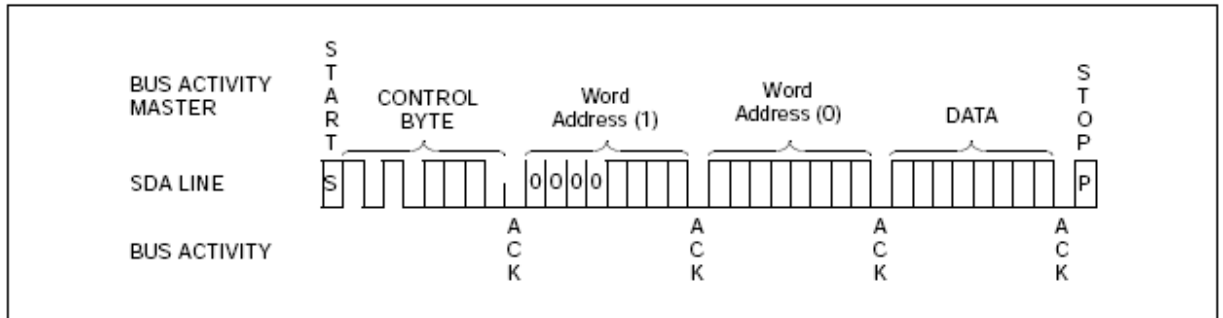
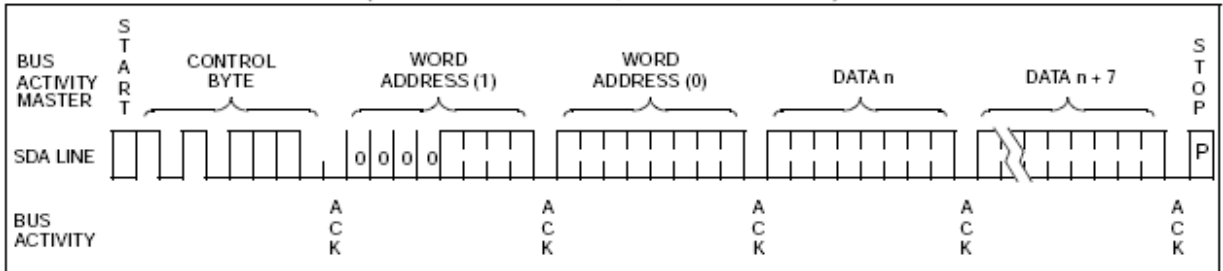


FIGURE 4-2: PAGE WRITE (FOR CACHE WRITE, SEE FIGURE 8-1)



In the thread video, you saw addresses and data scrolling up the screen and this was the output of a simple eeprom write/read program that nicely demonstrates the Byte Write process shown in the graphic above, with each write being additionally followed by a read-back of the same address.

```

10 FORA%=0TO&7FFF
20 H%=(A%/256)AND&FF
30 L%=(A%AND&FF)
40 *I2CTXB 57 H%;
50 *I2CTXB FF L%;
60 *I2CTXB FF A%
65 :
70 *I2CTXB 57 H%;
80 *I2CTXB FF L%;
90 *I2CRXB 57 D%
100 PRINT~A%,~D%
110 IFD%<>(A%AND&FF)THEN PRINT "Error":STOP
120 NEXT

```

Lines 10 and 120 loop the test through all 32k of the eeprom locations, i.e. &0000 to &7FFF. Lines 20 and 30 set H% and L% to the upper and lower bytes of the 16-bit eeprom address such that, for example, at an address of &1234, H% is set to &12 and L% is set to &34.

Referring now to the Byte Write graphic, the Start, Control Byte and RW are constructed by the I2C rom in response to the *I2CTXB 57 component of line 40 where our eeprom module has a device address of &57 and the H% component of line 40 results in the Word Address H of the graphic. (Note that all ACK in the graphic are transparently handled for you by the rom.)

You can see that there is no Stop specified after the Word Address H and no Start preceding the Word Address L. The Stop is suppressed by the trailing ; in line 40 and to ensure that we only send the eight bits of L% byte in line 50, the device address is set to the special value of &FF (out of range for I2C devices) which directs the rom to transmit only a single naked byte. Again, from the graphic, there is no Stop specified after the Word Address L and no Start preceding the Data byte so as before, line 50 is terminated with a ; and in line 60, the device address is specified as &FF. Line 60 specifies A% as the data byte to send and because the rom will only use the lowest eight bits of A%, each location of the eeprom has a value written to it which equates to the low byte of the target location address. For example, the eeprom memory location at address &1234 will have data &34 written to it. The graphic shows that a byte write is completed with a Stop and therefore, unlike the preceding two lines, line 60 is not terminated with a ; and hence a Stop is sent by default.

After each write, the program reads back the just-written data and the procedure for this is slightly unusual when compared to parallel devices because we have to begin the read with a couple of writes! Read on...

Most I2C devices maintain an internal latched counter that is used to keep track of either the last-accessed register in the case of bespoke-function chips such as our RTC or, the last-accessed memory location in the case of memory-based devices such as eeproms. If sequential reads or writes are made to a device with such a counter, the latter is simply auto-incremented from its last accessed value. However, if we want to read a specific (or random) location, we need to first set this counter to point to the target address and this explains the need for the writes prior to a random read. (In the more usual parallel system, the process just described is the equivalent of the CPU setting up the low order address bus lines to configure the target memory device prior to the read.)

Pictorially, the read process for the 24C32 looks like this...

FIGURE 6-2: RANDOM READ

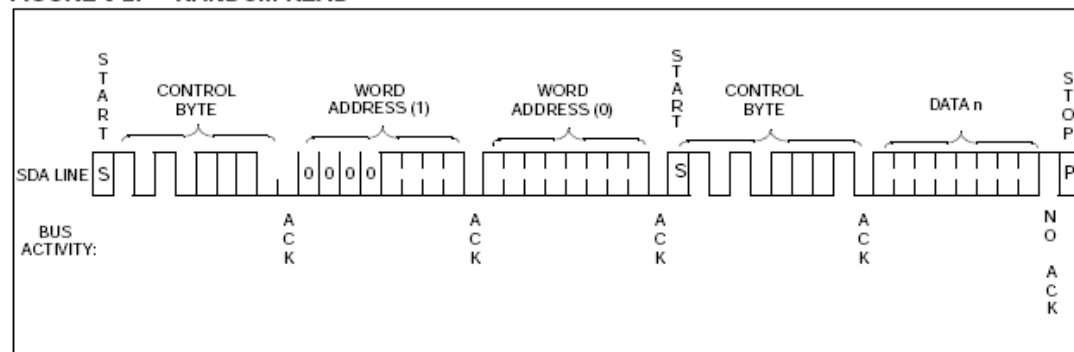
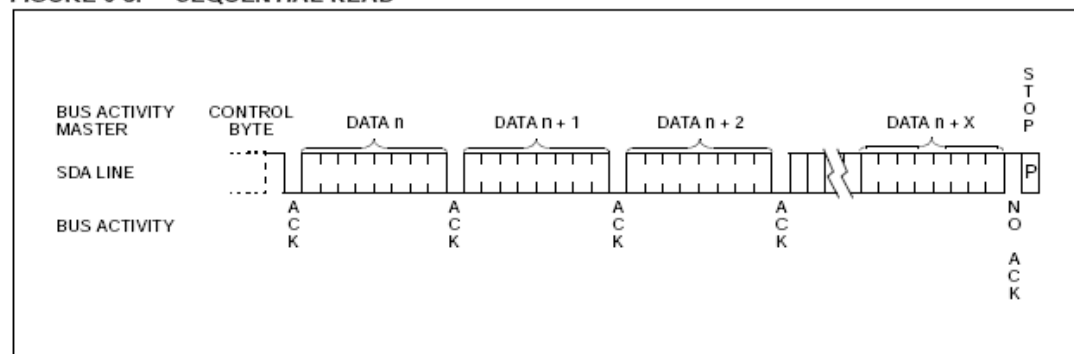


FIGURE 6-3: SEQUENTIAL READ



So, in the demo program, you can see that although we now want to read the eeprom, lines 70 and 80 are identical to earlier lines 40 and 50 where we were preparing to write data to the eeprom because in both cases, we first need to set the address counter to point to our target location. The difference now is that in line 90, we perform a Re-Start (a Start without a previous Stop) and request a read of the data byte at the current address (that we have just set) into D%. Finally, there is no trailing ; in line 90 and so a Stop is sent on completion.

Line 100 hex-prints the current address (A%) followed by the read-back data (D%) and the data is validated in line 110 since we know it should be equal to the lower byte of the eeprom subject address.

Now, I described how eeproms can auto-increment their address counter and during sequential contiguous writes, this is known as a Page Write. However, because eeprom devices require a finite time to update a memory location, during a Page Write, an eeprom has to internally buffer sequentially-written data prior to its been physically written to memory. This means that eeproms will have an upper limit to the size of a Page Write and in the case of the specific 24C32 I'm using (the sub-types vary), the page size is 32 bytes – generally specified as one Byte Write (or Random Write) plus 31 further page byte writes.

The program below demonstrates a single, full 32-byte Page Write beginning at eeprom address &1234. For homework, I'll leave it to you to figure out how it's done...

```
10 *I2CTXB 57 12;  
20 *I2CTXB FF 34;  
30 FORA%=0TO&1F  
40 *I2CTXB FF A%;  
50 NEXT  
60 *I2CSTOP  
70 *I2CTXB 57 12;  
80 *I2CTXB FF 34;  
90 *I2CRXD 57 20  
100 FORA%=0TO&1F  
110 PRINT~? (&A00+A%) ;  
120 NEXT
```

The DS3231 also contains an embedded temperature sensor for which the datasheet control and status register details are assembled below :

Control Register (0Eh)

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
NAME:	$\overline{\text{EOSC}}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE
POR:	0	0	0	1	1	1	0	0

Bit 5: Convert Temperature (CONV). Setting this bit to 1 forces the temperature sensor to convert the temperature into digital code and execute the TCXO algorithm to update the capacitance array to the oscillator. This can only happen when a conversion is not already in progress. The user should check the status bit BSY before forcing the controller to start a new TCXO execution. A user-initiated temperature conversion does not affect the internal 64-second update cycle.

A user-initiated temperature conversion does not affect the BSY bit for approximately 2ms. The CONV bit remains at a 1 from the time it is written until the conversion is finished, at which time both CONV and BSY go to 0. The CONV bit should be used when monitoring the status of a user-initiated conversion.

Temperature Register (Upper Byte) (11h)

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
NAME:	Sign	Data	Data	Data	Data	Data	Data	Data
POR:	0	0	0	0	0	0	0	0

Temperature Register (Lower Byte) (12h)

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
NAME:	Data	Data	0	0	0	0	0	0
POR:	0	0	0	0	0	0	0	0

Temperature Registers (11h–12h)

Temperature is represented as a 10-bit code with a resolution of 0.25°C and is accessible at location 11h and 12h. The temperature is encoded in two's complement format. The upper 8 bits, the integer portion, are at location 11h and the lower 2 bits, the fractional portion, are in the upper nibble at location 12h. For example, 00011001 01b = +25.25°C. Upon power reset, the registers are set to a default temperature of 0°C and the controller starts a temperature conversion. The temperature is read on initial application of V_{CC} or I2C access on V_{BAT} and once every 64 seconds afterwards. The temperature registers are updated after each user-initiated conversion and on every 64-second conversion. The temperature registers are read-only.

I2C Serial Data Bus

line while the clock line is high are interpreted as control signals.

Accordingly, the following bus conditions have been defined:

Bus not busy: Both data and clock lines remain high.

START data transfer: A change in the state of the data line from high to low, while the clock line is high, defines a START condition.

STOP data transfer: A change in the state of the data line from low to high, while the clock line is high, defines a STOP condition.

Data valid: The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the high period of the clock signal. The data on the line must be changed during the low period of the clock signal. There is one clock

The following program was used in a video posted in the Stardot I2C thread to display the temperature of the Beeb 6502 CPU chip over a period of minutes :

```

10 REM * I2C Temperature *
20 :
30 MODE0
40 PROCINIT
50 PROCTEMP
60 MOVE 100,130+(H%-10)*10)
70 REPEAT
80   N%=0:D%=-1
90   REPEAT
100    *I2CRXB 68 #00 S%
110    SU%=S% AND &F:ST%=INT(S%/16)*10:S%=ST%+SU%
120    IFS%<>D% THEN D%=S%:N%=N%+1
130  UNTIL N%=15
140  PROCTEMP
150  M=M+0.25
160  DRAW 100+(M*110),130+(H%-10)*10)
170 UNTIL FALSE
180 :
190 DEFPROCINIT
200 VDU23;8202;0;0;
210 MOVE0,0
220 MOVE100,130:DRAW100,930
230 MOVE100,130:DRAW1100,130
240 FORX=1TO9
250   PRINTTAB(3,3*X);100-(X*10)
260 NEXT
270 FORX=1TO9
280   PRINT TAB(5+(X*7),29);X;
290 NEXT
300 PRINT TAB(6,29);"0"
310 PRINTTAB(2,1);"Deg.C"
320 PRINTTAB(36,31);"Time Mins";
330 M=0
340 ENDPROC
350 :
360 DEFPROCTEMP
370 *I2CRXB 68 #0E C%
380 C%=C% OR &20
390 *I2CTXB 68 #0E C%
400 *I2CRXB 68 #11 H%
410 *I2CRXB 68 #12 L%
420 L=INT(L%/64)
430 L=L*0.25
440 T=H%+L
450 PRINTTAB(30,10);T;" Deg C    "
460 ENDPROC

```



Remembering that the RTC module has an I2C bus address of \$68, lines 90-130 are using the seconds register 00 in the RTC module to produce an n-second clicker which here gives a 15 second interval period for our temperature measurements. So, as we used previously in the time demo, line 100 uses `*I2CRXB` to read RTC seconds directly into `S%`. You will see from the data sheet that these arrive as a hex-coded decimal modulo 60 number - i.e. if you were to simply to use `PRINT ~S% ('print S% in hex')`, you would see an auto-wrap-round 0 to 59 being displayed. The other lines therefore pull `S%` apart to create a simple `N%` incremental seconds counter.

The temperature routine does several things - in lines 370-390, it reads the DS3231 RTC module Control Register with `*I2CRXB` (see datasheet description above), sets Bit 5 using logical OR to force a conversion and then writes the command back to the Control Register with `*I2CTXB`

Then, the temperature value is read back using two `*I2CRXB` commands in lines 400 and 410 putting the integer part (degrees C LSB of 1) directly into `H%` and the two-bit fractional part (degrees C, 0.5 in Bit 7 and 0.25 in Bit 6) directly into `L%`. The remaining lines just mess with the two parts to give a numeric display output and then the measured value is plotted on the graph elsewhere.

Note that the datasheet talks about the finite time needed for a temperature conversion and the corresponding conversion and busy status flags but in BASIC, with a long (seconds) sample period, we don't need to worry about using any such flags. If you were doing something much faster from assembler, you will probably need to read and respect the conversion status.

The remainder of the program is just standard Beeb BASIC code to draw the graph.

The final example, also demonstrated in the I2C forum thread, showed how a Wii Nunchuck, a modern off-the-shelf I2C device, can be used with a Beeb. This brought together many of the concepts discussed including the use of a 3.3v level shifter.



The Nunchuck is initialised in lines 90 & 100 and is actively read in lines 320 & 330.

```
10 REM * NUNCHUCK CROSSHAIR TEST *
20 MODE0
30 GCOL3,1
40 VDU23;8202;0;0;
50 MOVE0,0
60 X%=512:Y%=512
70 PROCCROSS
80 :
90 *I2CTXB 52 #F0 55
100 *I2CTXB 52 #FB 00
110 :
120 REPEAT
130 PROCNUNCH
140 DX%=(NX%-128)/4:DY%=(NY%-128)/4
150 IF ABS(DX%)>14 THEN DX%=DX%+(DX%/2)
160 IF ABS(DY%)>14 THEN DY%=DY%+(DY%/2)
170 IF DX%<>0 OR DY%<>0 THEN PROCXMOVE
180 IFC%=0 THEN PLOT69,X%,Y%
190 IFZ%=0 THEN PLOT69,X%,Y%
200 PRINTTAB(10,10);C$
210 PRINTTAB(50,10);Z$
220 UNTILFALSE
230 :
240 DEFPROCXMOVE
250 PROCCROSS
260 X%=X%+DX%:Y%=Y%+DY%
270 PROCCROSS
280 ENDPROC
290 :
```

(contd.)

I2C Rom Software for Acorn Computers – Issue 3.1

```
300 DEFPROC NUNCH
310 :
320 *I2CTXB 52 00
330 *I2CRXD 52 06
340 :
350 NX%=?&A00:NY%=?&A01
360 C%=?&A05 AND &02
370 Z%=?&A05 AND &01
380 C$="          ":IFC%<>0 THEN C$="FIRE-C"
390 Z$="          ":IFZ%<>0 THEN Z$="FIRE-Z"
400 ENDPROC
410 :
420 DEFPROC CROSS
430 MOVEX%-16,Y%:DRAWX%-64,Y%
440 MOVEX%,Y%-16:DRAWX%,Y%-64
450 MOVEX%+16,Y%:DRAWX%+64,Y%
460 MOVEX%,Y%+16:DRAWX%,Y%+64
470 ENDPROC
```