I2C and SPI Foundation

Revision

- 30 September 2010 Release
- 17 March 2018 changed ref: command 'f' to 'x'

Introduction

I2C (I squared C) and SPI (Serial peripheral Interface) are two main ways that microcontrollers use to communicate with the outside world. Both are reasonably high speed and use just a few wires. Typical I2C devices are A to D converters, port expanders, temperature measurement etc. Typical SPI devices are the same but SPI being much faster is also used on memory devices and displays. The MMC standard (SD Cards) for example have an SPI interface.

This text serves two purposes, the first being an introduction to both busses but in a practical way and the second is to help familiarise with the BV4221-V2 which is a versatile tool for using these busses.

This is not 'plug and play'; some basic programming and electronics experience is necessary in order to carry out the projects.

> **IMPORTANT:** Please read or do Project 1 (1C or 1P) first as this contains basic information that is not in any other project. In other words this project is the introductory project and all other projects assume that this one has been at least read.

Resources

To complete the projects listed in this text the following resources will be required:

Just Basic

This is a free programming language that is easy to install and use. One big advantage is that it handles the PC comm. port quite well. It can be obtained from http://www.justbasic.com/ and is only a small download. The projects can be done without this but it would mean a lot of typing into a terminal (BV-COMM) for some projects.

Zip File

The Foundation ZIP file contains all of the examples and other software resources, including data sheets, that may be needed. This should be unzipped into a suitable directory. The zip file contains a series of project directories.

BV4221-V2

This is the USB to I2C/SPI convertor and can be obtained from http://www.byvac.com

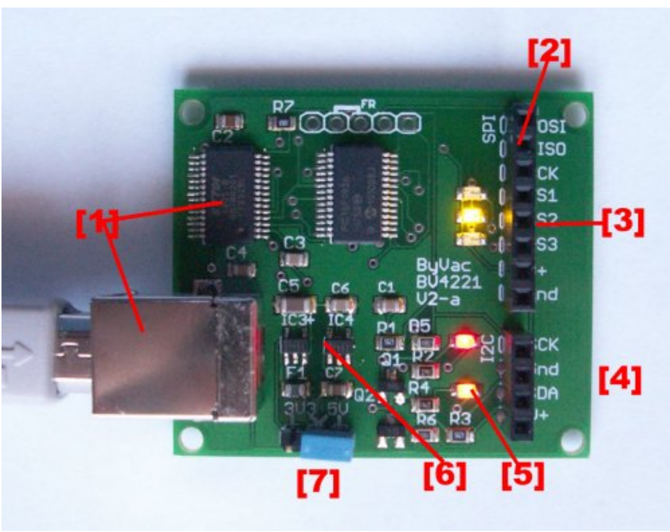


Figure 1 BV4221-V2

1. USB interface is Virtual COM port
2. SPI Interface
3. Three Chip select lines to drive up to 3 devices
4. I2C interface
5. SCK & SDA LED indicators, go out when bus is stalled
6. On Board voltage regulators
7. Voltage Selector 3.3V or 5V to cater for differing logic

Getting Started

Connect the BV4221-V2 to a suitable PC running Windows. Install the FTDI software if required following the instructions form the FTDI web site. The BV4221-V2 uses the VCOM driver. If you have the BV4221-V1 then there will be no need to install this driver. A driver is provided in the resources zip file but it may not be the latest.

When this is installed it will present itself as a COM port, using BV-COMM (supplied on the ZIP file) set the correct port number and press the red icon on the left which will turn green, now press <enter> and you will be presented with an 0x42> as Figure 2.

Pressing <enter> again will produce another prompt. If this does not work check the following items:

1) Is this green
2) Is the COM port set correctly, on this system COM 9 is used it WILL be different on your system. To see what it may be use Settings>Refresh Ports and then pick the correct COM port.
3) Try resetting the BV4221-V2 using

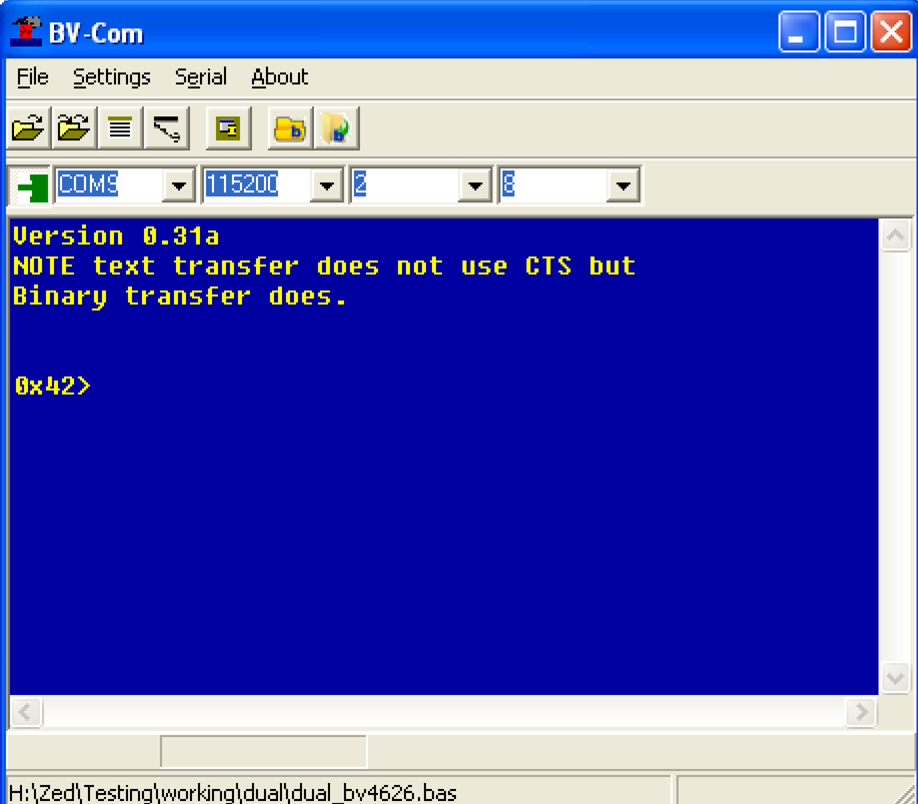


Figure 2 Initialising the BV4221-V2

The above will verify that the BV4221-V2 is working with your system. All of the projects could be carried out using this terminal but it would involve a great deal of typing on some projects, that is the reason for using JB (Just Basic)

I2C

For in depth technical details about I2C read http://en.wikipedia.org/wiki/I%C2%B2C this is quite comprehensive. The **practical** side is dealt with here, actually using the bus to interface with devices.

This is a two wire bus but in practice 4 wires are needed, the two extra being to carry the power supply. It is slightly more complicated than the SPI system and a bit slower but the design lends itself to very many applications.

This is also an **addressable** protocol which means that several devices can share the same bus because each device has its own address.

Project C1 24C16 EEPROM

The 24C16 EEPROM and its derivatives is one of the more complicated I2C devices but it does have all the ingredients required for using the bus. If this is mastered then all of the other I2C devices should be easily understood.

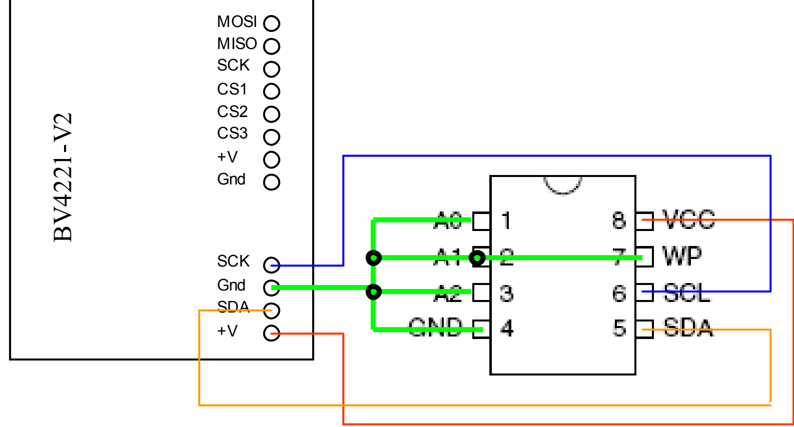


Figure 3 I2C EEPROM ** 24C01 Device Leave A0-A2 disconnected for 24C16

The I2C side of the BV4221-V2 is used for wiring the EEPROM. The easiest way to do this is some ribbon cable and a pin head connector going to a breadboard, see Figure 10.

Most I2C devices have a few pins dedicated to address lines so that the address of the chip can be changes, thus allowing several of the same type of device to be used on the same bus. **The 24C16 is unusual in that the A0 to A2 lines are not used and are left disconnected. The base address of 0xA0 (see reading & addressing) is used to access the first page and the three lower bits of the address are used for accessing the pages.

WP has also been connected to ground for this project as pulling it high (connecting to V+) will prevent any writes to the device.

Address Discovery

With just this device connected and the BV4221-V2 in the I2C mode use the 'x' command to discover the address of the device:

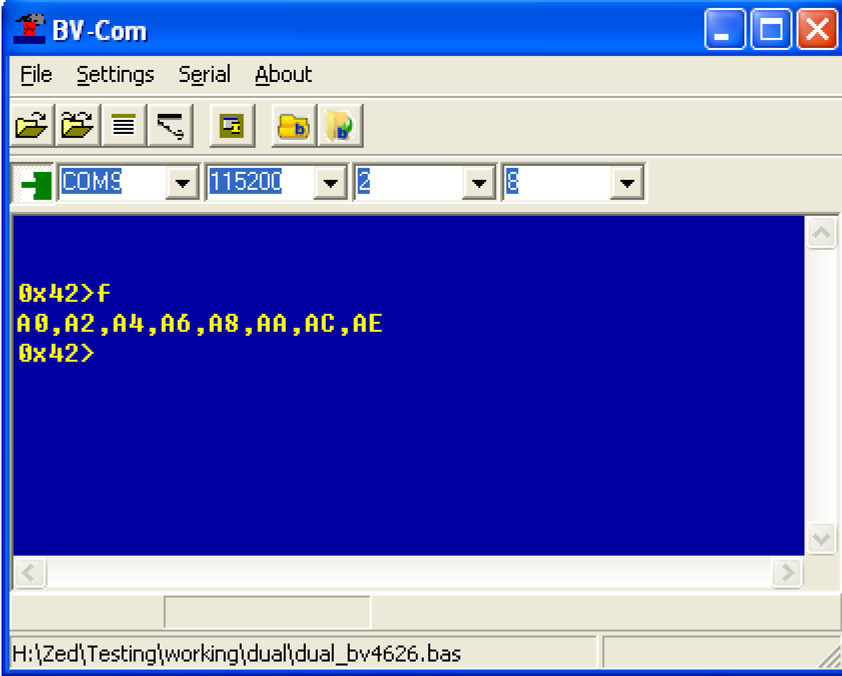


Figure 4 24C16 Addresses

As previously mentioned for the 24C16 device this will appear as 8 addresses. The reason for this is that the device address is used as part of the EEPROM address. To address the contents of EEPROM a single byte is used, therefore this is only capable of addressing 256 bytes, adding the extra three bits of the DEVICE address extends the 8 bits to 11 bits.

One of the most confusing parts of using an I2C memory device is separation out the device address from the contents of the EEPROM address but using this scheme just adds to the confusion – ah well.

The Address Counter

As far as I know all EEPROM and Flash memory devices have an address counter, the purpose is to save having to set the address each time a byte is read or written. When reading for example the address counter will increment to the next location after read completes. This is highly convenient but comes with a price. The counter is usually a lot smaller[1] than the total array size, in the case of this EEPROM it is 16 (8 for smaller devices) so when it reaches 16 it rolls over back to 0. To further complicate matters this counter is set at fixed locations or boundaries in the example of this chip the boundaries are at 0,0x10,0x10,0x30 etc.

This must be taken into account for all reading and writing operations.

[1] The reason it is smaller is that it usually points to some temporary ram otherwise if it were just a counter then there would be very little restriction to its size.

Reading & Addressing

A read can be performed at any time and it will give the contents of the EEPROM at the address counter location, the address counter will be incremented for the next read. On an I2C bus the first bye is always the address of the device. I2C uses a 7 bit address (there are 10 bit versions not discussed here) with the least significant bit set to 1 for reading and 0 for writing.

In practice this give an 8 bit EVEN address for writing and an 8 bit ODD address for reading. So using this chip the address is 0xA1 for read and 0xA0 for write for the first part of the EEPROM. For more information about addressing problems see this page under addressing http://www.i2c.byvac.com/ar_trouble.php.

So reading is a matter of sending the device address with the read bit set and then reading the data, thus:

```
0x42>s-a1 g-1 p
```

To explain, 0x42> is the prompt from the BV4221-V2.

s-a1

This will send a start condition followed by address 0xA1, because we are in hex mode. If 's' was used on its own then it would send the start condition followed by 0x42 which is the default address, this can be changed by using the 'A' command.

g-1

Gets a byte from the I2C bus and returns it to the console, this will probably be 0xFF if it is a blank EEPROM.

p

Is the stop condition that must be on the end of all BV4221-V2 command, it does not need to be on the same line but it is this 'p' command that will initiate the sending of the complete line of commands, i.e. "s-a1 g-1" in this case.

As we do not know the value of the address counter then we have no Idea where this value has come from. What is required is to set the EEPROM address before reading, this is called a random read in the datasheet and uses a very common technique of:

<write><restart><read>

The first write is to tell the EERPOM to set the address counter, a restart is required after this in order to read the next byte. After that of course the program counter is known and just a read can be used.

```
0xA0>s 0 r g-1 p
```

In this sequence the default address has been set to 0xA0 by using the 'A' command, this as can be seen, changes the prompt.

s 0

This sends the start condition and the device address 0xA0 (write) to set the program counter to 0

r

Is the restart command which is basically a stop command followed by a start command and the device address+1 (0xA1 in this case). The rest is as before, this time the result will be the contents of address 0.

Writing

This is a bit simpler than read as it consist of <device address><eeprom address><data....>

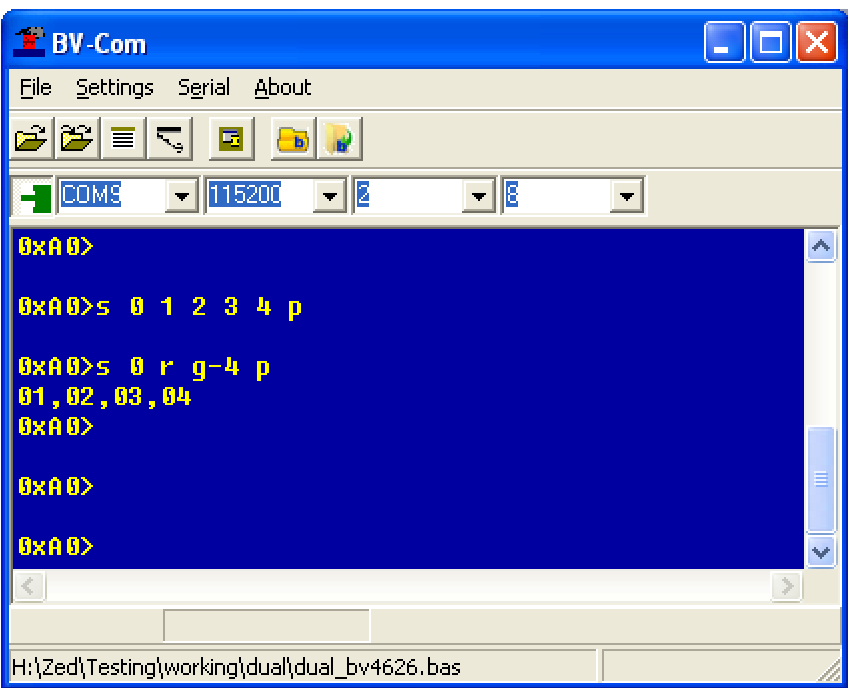


Figure 5 Writing followed by read

The above Figure 5 should work for all devices, it write 1,2,3,4 to location 0 and then reads it back.

Summary

The EEPROM device uses the I2C bus in a typical way with a mixture of reading and writing and the use of the restart command. Simple devices such as the port expander will use the R/W bit to determine if data should come from the device or go to the device, as can be seen from the above most devices are not that simple.

Be careful with memory devices and their page boundaries, quite a bit of extra code is needed to cope efficiently with this.

SPI

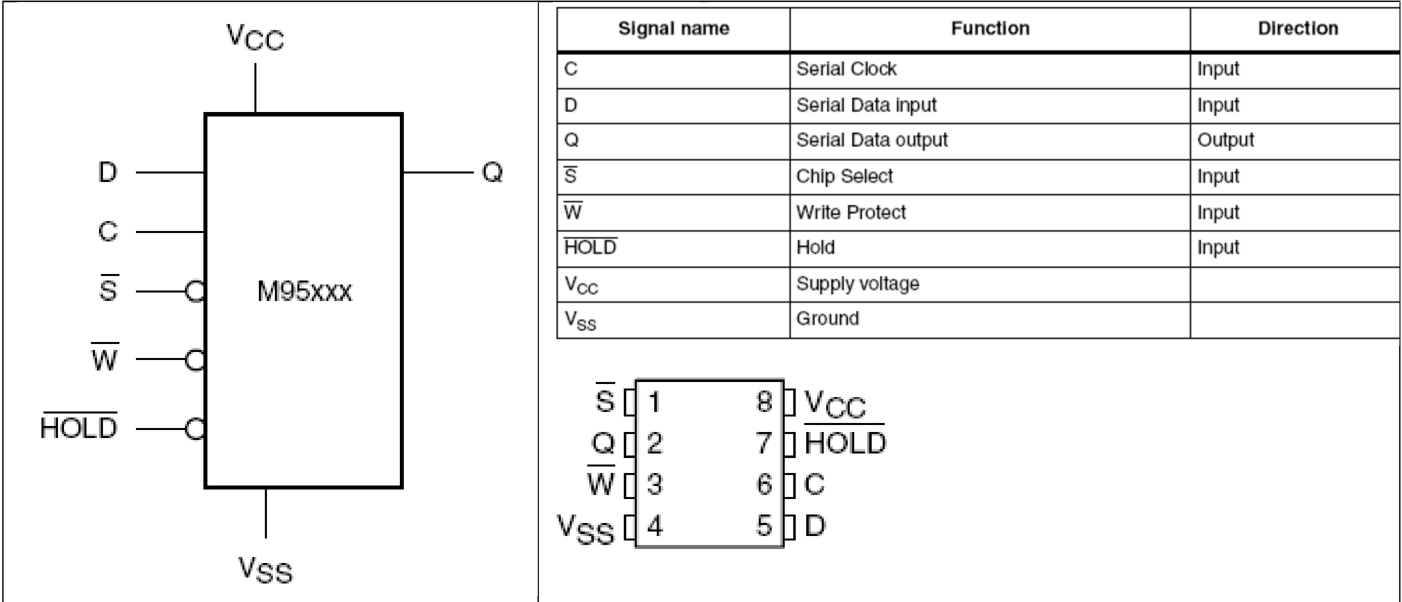
For in depth information about the SPI bus see http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus this is a 4 wire bus plus power supply and so in practice 6 wires will be needed.

Project P1Reading & Writing EEPROM

The first project uses a '95160' EEPROM device. This can store and retrieve information on and EEPROM (Electrically Erasable Programmable Read Only Memory). Its main use is for storing small amounts of data that needs to be retained after a device has been switched off. For example the volume setting of a TV could be remembered by one of these, it would be most annoying having to adjust the volume every time the TV was switched on.

The 95160 has 2kB (2048 x 8bit) memory and can be read, written or erased from any location. It can also operate ate 10MHz and so is considered reasonably fast.

Figure 6 shows the logic diagram and the meaning of the pins along with the physical layout. For this project hold and W are tied high.



Signal name	Function	Direction
C	Serial Clock	Input
D	Serial Data input	Input
Q	Serial Data output	Output
$\overline{S}$	Chip Select	Input
$\overline{W}$	Write Protect	Input
$\overline{HOLD}$	Hold	Input
V_{CC}	Supply voltage	
V_{SS}	Ground	

Figure 6

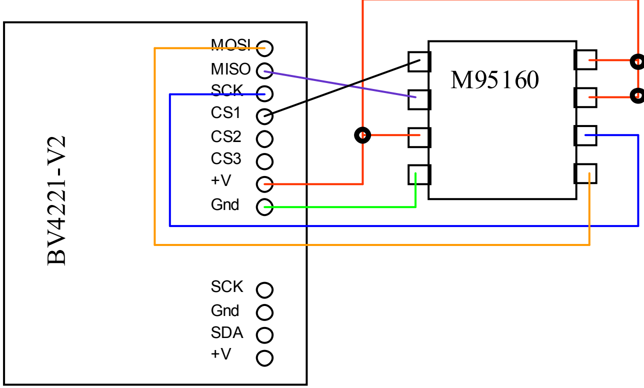


Figure 7 BV4221-V2 Wiring

The 95160 should be wired as shown in Figure 7.

Method

Wire the EEPROM as shown in the diagram. The data sheet can be found in the 95160 folder for further reference. The associated JB file is 95160.bas. MOSI stands for Master Out Slave In and is thus connected to the salve input. MISO stands for Master In Slave Out and is therefore connected to the slave output.

The EEPROM works, as many SPI devices, by sending a command followed by specific data. In most cases the command will be followed by an address of where to store or retrieve the data from. An added complication for this device is that any writing to the EEPROM requires an 'UNLOCK' command first. This is to prevent accidental writes. As reading is simpler we will take that one first.

The Address Counter

As far as I know all EEPROM and Flash memory devices have an address counter, the purpose is to save having to set the address each time a byte is read or written. When reading for example the address counter will increment to the next location after read completes. This is highly convenient but comes with a price. The counter is usually a lot smaller[1] than the total array size, in the case of this EEPROM it is 32 so when it reaches 32 it rolls over back to 0. To further complicate matters this counter is set at fixed locations or boundaries in the example of this chip the boundaries are at 0,0x20,0x40,0x60 etc.

This must be taken into account for all reading and writing operations.

> *[1] The reason it is smaller is that it usually points to some temporary ram otherwise if it were just a counter then there would be very little restriction to its size.*

Reading

The read command for this device is 3, see table 4 on the 95160 datasheet. Reading is quite straightforward and involves the following commands:

1) set CS low
2) send 3
3) send the address
4) read the data
5) set CS high

On ALL SPI devices the CS is set first, this is usually low to activate but not always. As this is the first project we will do this by 'hand' using BV-COMM first before using JB.

1) Set up the BV4221-V2 use commands "S" <enter> "mo" <enter> "D" <enter>
2) Issue the command "l1 3 0 0 g-10 h1"

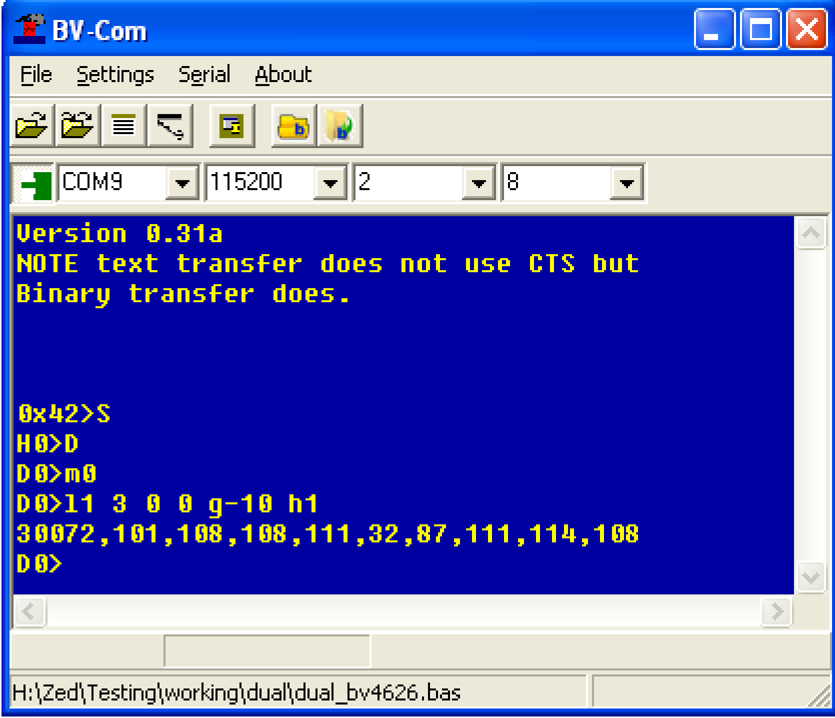


Figure 8 Simple Read

At 1) the BV4221-V2 has been set to DPI mode, Decimal and Mode 0. This only needs to be done once after reset, if this is going to be used all of the time then it can be 'fixed' with the F command so that the initialisation does not need to take place each time. I would NOT recommend doing this on an automated system as setting the device each time has some advantages.

At 2) 'l1', that is lower case L and one will set the CS1 on the BV4221-V2 to low. The '3' is the read command and '0 0' is the address[1], g-10 will get 10 bytes from the SPI bus and h1 resets CS1 to high.

> ***[1]*** *The datasheet for the 95160 requires a 16 bit address and as the default transfer length (see command n) is 8 bits then two bytes are required.*

The values returned are shown in Figure 8. On a blank EEPROM these will be all 255 values. It may be worth explaining what is happening here in terms of the SPI bus. The SPI bus is full duplex which means that as bits go in so bits come out, this means that when you put something in you always get something out and to get something out you have to put something in. This can cause confusion but it is simply a matter of ignoring what you don't want.

To relate this to the above example, 3 and 0 0 have been put onto the SPI bus and thus into the EEPROM and 3 and 0 0 have been returned by the EEPROM which we can choose to ignore. The 'g-10' command actually sends 10 x 0xff values to the EEPROM, the EEPROM responds by returning the correct values. This is a common technique, when the SPI device does not need any information but has information to give then 0xff is sent. By the same token when the slave requires information but does not have any to give then the output from the slave is simply ignored.

When using numbers on their own, the result is always returned back. To send a value to the SPI buts but to ignore any returned values then use the 's' command. This sequence "l1 s 3 0 0 g-10 h1" for example will only return the bytes we are interested in.

Writing

This is a bit more complex, there are two factors to consider: 1) before writing the EEPROM needs write enable setting and 2) because write is a relatively slow process monitoring needs to take place to determine when the write has finished before writing anything else.

To initiate a write, the write enable command must be sent thus:

"l1 s 6 h1"

The write enable command is '6' and this must be bounded by chip select as above, this fact usually catches people out.

Once this is set then the EEPROM can be written to, for example:

"l1 s 2 0 0 97 98 99 h1"

will write "abc" to address 0. As previously mentioned the write is a relatively slow process and so the status register needs checking to see that the write has finished. In the case of this particular EEPROM the bit in the status register to check is bit 0.

```
[doWriteLoop]
    call doSend "l1 s 5 g-1 h1" ' get status
    call doWait 62, 1000 ' ends in prompt
    if val(rc$) > 0 then goto [doWriteLoop]
end sub
```

This shows a snippet of JB code where the register is checked, command '5' will return the register status and this under normal circumstances will return 1 if it is still writing or 0 if finished. This is a bit simplistic but will do for the illustration. In practice it would be better to 'and' the result with 1 and see if it is 0.

JB Code

The Just basic code is in the file 95160.bas and has a simple user interface for initialising, reading and writing to the EERPOM. At the top of the code set the COM port before using.

Sending Data

When sending data to the BV4221-V2 it expects it to be terminated with CR and only CR (#13), the routine doSend will ensure this. Also in this routine the first line will clear the input buffer of any characters that may be left over from the previous commands. This was found necessary in practice.

Receiving Data

The tried and tested technique for receiving data is to read in characters from the COM port until a known terminating character has been received. You may have noticed that the BV4221-V2 always ends with a prompt, at the end of the prompt is '>' which is #62. The routine doWait, will wait for a character or will time out. The gathered characters are placed in the global variable rc$.

Summary

The EEPROM M95160 is typical of all of the EERPOM and Flash devices and so understanding this one will help with all of the others. This particular EEPROM does not need erasing before writing which makes it very simple to use.

In the code given no consideration is given to the page boundary and so if 10 bytes are written to address 30 you will end up with the first two bytes going to address 30 and 31 and the rest of the bytes going to 0 to 7 as the page wraps round. To simplify the code any page checking has been left out.

Project P2 MCP23S08 Port Expander

This is an 8bit serial to parallel port expander. In other words it will give an 8 bit input/output port from a serial interface. This type has an SPI interface there are also available MCP23008 that have an I2C interface.

This device is slightly over complex for a simple expander but with the complexity comes flexibility. Using the device basically amounts to writing or reading from 11 internal registers. To access those registers a read or write op-code is sent followed by the address of the register followed by the data.

One added complication which is unusual for an SPI device is that the Op Code is an address and can be changed in hardware. This is unusual because SPI devices are not addressed but normally selected by the CS line.

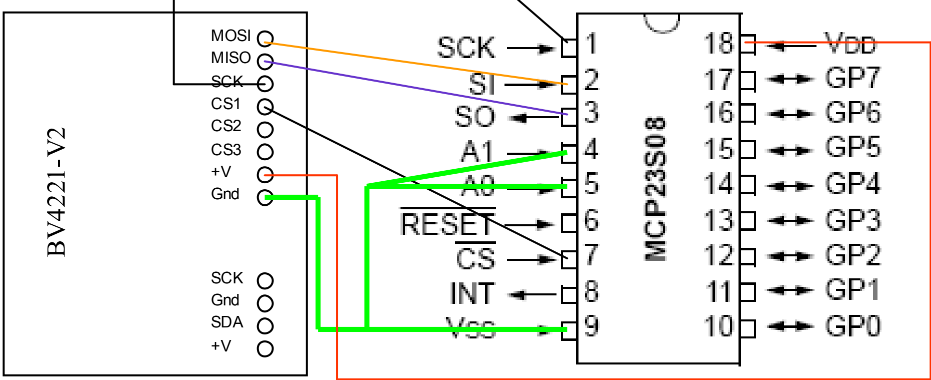


Figure 9 28S08 Wiring

The reset should be tied to +V but this is not shown on the diagram. There is a power on reset and so this pin will not be needed for this example. Tying A0 and A1 to ground will give an Op Code value of 0x40 for write and 0x41 for read.

Method

The easiest way to connect this up is by placing the IC in a breadboard and making up a lead consisting of an 8 way pin head and some ribbon cable. Attaching stiff wire to the other end of the cable makes it possible for the end of the wire to be inserted into the breadboard (Figure 10).

Figure 10 Connecting Cable

When testing the o/p simply place an LED on one of the i/o lines, for input a resistor to +V was used and then shorted to ground to see that the input could be detected.

As this text is to illustrate the SPI bus and the BV4221-V2 the finer details of this device will be ignored

Writing

It may be easier to start off with this. At reset all of the i/o is set to input so the first thing to do is set all of the ports to output as follows:

1) Set the BV4221-V2 to SPI, mode 0 and Hex "S" <enter> "m0" <enter> "H" <enter>
2) Set device to o/p "l1 s 40 0 0 h1"
3) Set GP0 to high "l1 s 40 9 1 h1"

This can be done using BV-COMM and observing the results at pin 10 with an LED or meter. The I/O register is 0 and so this was set to all 0 in step 2. At step 3 1 was sent to the GPIO register 9, this made GP0 high.

Reading

Here we need to get some results back from the device and so we can use the g-n command:

1) Set all of the port to input "l1 s 40 0 ff h1"
2) Read from the GPIO register "l1 s 41 9 g-1 h1"

At stage 2 the Op Code changes to 0x41 because this is a read operation, 9 is the register to read from and the data sent is 0xff from the g-n command. Using this command will return the value.

JB Code

The code is in a file called "23S08.bas" and the reading and writing the device part is very simple, almost mirroring what you would do manually using BV-COMM. The complex part is in the reading for the serial port and waiting for the prompt characters, this is fully explained in the first project.

Project P3 Digital Pot

Digital Potentiometers can also be used as Digital to Analogue devices but these are a bit more flexible in that a virtual wiper can be moved across a resistance. The IC used for this project is the MCP41100. This is the single channel device in a 8 pin DIP package.

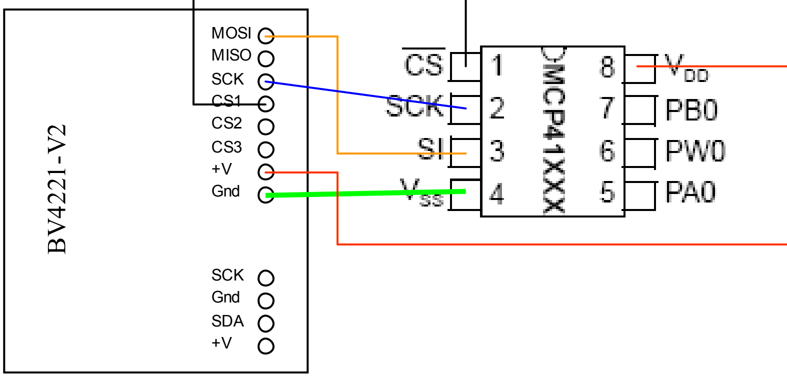


Figure 11 MCP41100 Digital Pot

As this is an output only device and so there is no connection for MISO (input to the master). For the purposes of this text connect PB0 [7] to ground and PA0 [5] to +V. A voltage meter can then be connected across ground and PW0 [6], this will reflect the digital value.

Writing

This device in keeping with most other SPI devices has some sort of command byte that precedes the data. The command byte in this case selects the mode and channel. The mode is selected by the high nibble and the channel by the low nibble. As this has only one channel and we will be writing to the device the command will always be 0x11. To write to the device therefore uses two bytes 0x11 and the value, thus:

"l1 s 11 <value> h1"

Try this using BV-COMM. It is probably not worth using JB for this as there is so little to do.