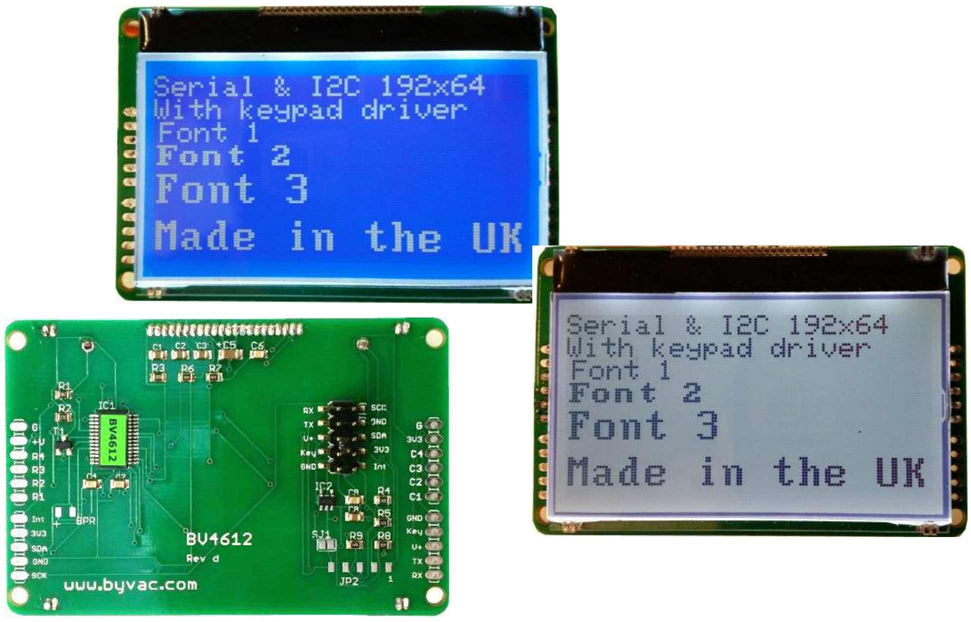


BV4612B

Serial/I2C Graphics display with Keypad control

Product specification July 2015

Contents

Serial/I2C User Interface BV4612B

Rev	Change
July 2015	Preliminary

1. Introduction

This is a serial / I2C user interface for use with microcontrollers, for example the Arduino or Raspberry Pi.

The output is in the form of an LCD display and the input is a user-configured touch keypad with 16 keys.

Full I/O control can be realised with only 2 wires. The keypad control has a 32 byte buffer relieving the host microcontroller of a considerable burden.

2. Description

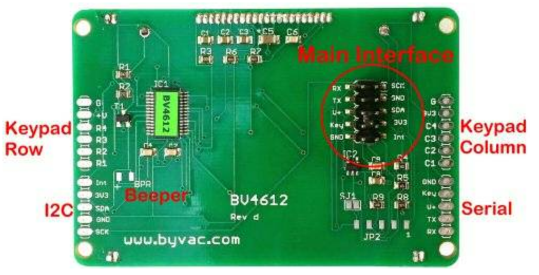
The device consists of an LCD display on a PCB back panel which has the control microcontroller.

A cross-point switch type keypad can be fitted to the PCB connections provided. The controller will scan the keypad and store the key values. No intervention of the host controller is needed, simply read out the keys from time to time.

3. Features

- Display 128x64 Graphic
- 3 Fonts
- User selectable Serial/I2C address
- Software adjustable contrast
- Software switchable back light
- 16 Pad keypad interface
- 32 key buffer
- Interrupt pins
- Pads fully configurable
- Dual Voltage 3.3V & 5V
- 16mA @ 3V3 BL full on, 13mA off
- Sleep Mode 8.4mA
- Only 2 wires for full I/O control
- Beeper output

4. Physical Description



There are 5 interface points but some of these are duplicated.

Pin	Description
RX	Serial receive pin
TX	Serial Transmit pin
V+	* See text
KEY	Normally high, will go low if there are any keys in the buffer.
INT	Goes low only when a key is being touched otherwise it is high
SCL	I2C Clock
SDA	I2C Data
GND	Ground
3V3	* see text

Main Interface

NOTE These pins are also repeated in the pads the left and right of the PCB to facilitate different mounting methods.

4.1. Power Supply

The device works on 3.3V but there is an on board 3.3V regulator and so it can operate from 3.3V or 5V as follows:

3.3V

Use the 3v3 to power device, I2C should use pull up resistors to 3.3V.

5V

The input regulator can have an input of up to 6.5V. Connect to the V+ input. The serial RX pin is 5V tolerant but the TX pin will only output 0 and 3.3V. This will be good enough for nearly all 5V serial devices.

I2C pull up resistors should be connected to a 3.3V supply.

5. Beeper

There is an output that goes to 3.3v momentarily when a key is pressed. This can be attached to a standard beeper or buzzer to indicate that a key has been pressed.

6. Keypad

The keypad interface expects a cross point switch.

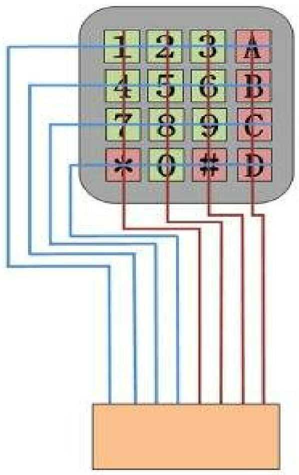
Below is a typical keypad, there are 4 row pins and 4 column pins. It does not really matter if they are the other way round.

There are also extra power and ground pins that are not needed for the this type of keypad.

6.1. Key Buffer

There is a 32 key, key buffer to store pressed keys. It is a circular buffer for maximum flexibility. It is up to the user to ensure that the buffer does not become full as this will overwrite previous keys.

There is an indicator bit (see Device Parameters section) that will send a message to the display if the buffer becomes full.



Typical Keypad layout

7. I2C Interface

The device has a standard I2C interface and will act as a slave device.

Address: 0x6a (0x35 7 bit)

All commands go through the single I2C address that can be changed if required by the user.

NOTE: The address is stored in EEPROM in three places and a check is made at each reset to verify the value. At leas two address location values have to agree, if this is the case the third is set to that. If no addresses agree then the default address is used.

This is a robust method of storing addresses in a semi-volatile memory and in nearly all cases the address set by the user is maintained for ever. However if it is critical that the address cannot change under any circumstances then the part can be ordered with a fixed address.

8. Serial Interface

The serial interface is via the TX and RX pins, By default the Baud rate is set at 9600. This can be changed by altering a value in the EEPROM via the EEPROM write command.

The protocol follows the standard 1 start bit, 8 data bits and 1 or 2 stop bits. All data (with the odd exception) is ASCII coded so that is the number 75 is sent via the serial interface then this will be TWO bytes '7' and '5', the actual value of the bytes will be 55 and 53, that being the ASCII codes for 7 and 5.

The exception to this is when sending image data that requires a faster throughput.

8.1. Hand Shaking

This has been avoided by the use of ACK. A serial command consist of a packet <address><command and data><EOL>

All packets are less then the buffer size and so the device will not respond until a full packet is received. When the device receives a packet it carries out the command and THEN sends the ACK back to the host. The host should not send any commands until the ACK is received. This method of communication avoids the need for a hardware handshake that is the cause of so many serial problems.

9. Sleep Mode

The device can be set to sleep mode via an I2C / Serial command. In this mode the keypad is inactive however the device can be awakened by an I2C / serial read or write.

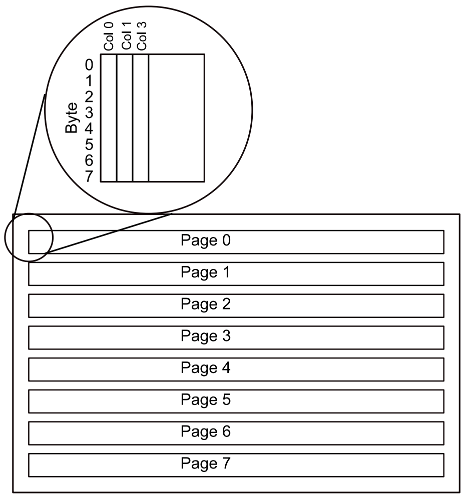
10. LCD

The LCD is a chip on glass (COG) type.

The LCD uses a UC1701 (equiv. KS0108B) controller via an internal interface. The interface is write only which limits the display to addressing columns and pages only.

Even with this limitation is possible to produce images using a free utility.

Special note: Most of the time the layout will not concern the user as the processor will take care of fonts and some graphics.



The layout of the display is in 8 pages, each page is 128 columns wide and 8 bits high. When writing to the display a byte is written at a time. This produced 8 pixels in a vertical line.

The x co-ordinate can be 0 to 127, however the y co-ordinate must specify a page 0 to 7. It is possible to specify an exact (0-63) y co-ordinate by specifying the correct page and then writing a byte that corresponds to that pixel. This is how images and fonts are produced.

The user need not worry too much about this as the device has a built in font generator.

10.1.1. Fonts

Fonts can be specified to begin on any x pixel but must start on a page (0-7).

There are three fonts, font 1 is 8 bits high and 6 bits wide and thus will fit into a page line, font 2 is the same as font 1 but 8 bits wide and so looks 'bold'. Font 3 is double height occupying 2 pages.

10.2. Images

Images are sent to the device as binary for both I2C and Serial. The format is:

<number of pages> <number of columns> <data>

The data is in a particular format which suits the page layout of the display. There is a utility written in Python that will convert a 32bit colour BMP image into a monochrome data block suitable for incorporating into either a C for ByPic file.

Images must not exceed 128x64 otherwise distortion will occur.

10.3. I2C

Pseudo code for sending an I2C image. NOTE: The serial function requires a time out, this is a time out for getting each character

```
i2c_start(106)
i2c_putc(34) // command
pages = img[0]
bpp = img[1]
i2c_putc(pages) // pages
i2c_putc(bpp) // bytes per page
for j = 0 to (pages*bpp)-1
    i2c_putc(img[j+2]) // binary
next
```

10.4. Serial

Sending serial data requires a timeout as there is no built in handshaking as there is for I2C. The time out should be sufficient to allow the device to laydown a byte of data.

Pseudo code for sending a serial image.

```
puts(”jp5000\r") // 5000 is timeout
pages = img[0]
bpp = img[1]
putc(pages) // pages
putc(bpp) // bytes per page
for j = 0 to (pages*bpp)-1
    putc(img[j+2]) // binary
next
```

11. Device Parameters

The EEPROM contains important values that control the way the device behaves. All of the values can be changed by the user using the i2c interface.

The EEPROM consists of 255 bytes and in general the first 16 bytes are used by the system

Adr	Default Value	Description
0	0	System Use
1	106	Device address ‘j’
2	25	Default contrast
3	4	Indicator flag
4	6	ACK [1]
5	21	NACK [1]
6	1	Beeper
7	4	Baud rate Code [1]
8	13	End Of Line (EOL)
14	106	Device address copy
16	0	Reserved
17	0	Mode (keep to 0)
18		Not used
19		Not used
20		Not used
21	30	Key table pointer (KP)
22	16	Key table size
23	20	Debounce
24	10	Repeat H
25	0xc4	Repeat L
26		Not used
27	1	Back light
28	60	Sign on message location
250	106	Device address copy

Table 1 System EEPROM use NOTE [1] applies to serial only

Key Code Table		
Location	**Name**	**Content**
KP+0	K1	0x37
KP+1	K2	0x3b

KP+2	K3	0x3d
KP+3	K4	0x3e
KP+4	K5	0x27
KP+5	K6	0x2b
KP+6	K7	0x2d
KP+7	K8	0x2e
KP+8	K9	0x17
KP+9	K10	0x1b
KP+0	K11	0x1d
KP+1	K12	0x1e
KP+2	K13	0x07
KP+3	K14	0x0b
KP+4	K15	0x0d
KP+5	K16	0x0e

Table 2 Step tables

The user is free to use any locations that are not occupied by the system but for future use it is best to avoid locations below 32.

Most EEPROM values are only read on start up so when changing values they may not take effect until the device is reset.

11.1. Address

These EEPROM locations contains the device address. By convention the address is set to values between the values 97 to 122, no checking is made by the device so setting values outside this range may or may not work.

For security the address is stored in three places and to change the address of the device at least two of the locations need to be set otherwise the device will detect the anomaly at start up and revert to the majority value.

Normally to change the address of a device locations 1 and 14 are both changed. The device will detect this at start up and change the address in location 250 to match.

11.2. Contrast

This is the default contrast setting for the LCD display and the default value will give good results in normal conditions.

The contrast can be set at any time so this value does not need changing it is simply the value that is used for initialisation.

11.3. Indicator Flag

NOTE: This flag is intended for debugging mainly. The 'features' will more than likely get in the way of a user program and so should probably be switched off (set to 0). There is one exception and that is the buffer full flag. As the buffer should never get full it will indicate programming errors.

This is a byte that has three bit value, when set to 1 the indicator is on, when set to 0 it is off:

0b0000ABCD

bit A is set by default, this will place text on the top right to indicate I2C or serial mode

If bit B is set (**key buffer full**)

When this flag is set and the key buffer becomes full, a message is printed on the bottom line of the LCD display.

If bit C is set (**keys in buffer**)

If this bit is set the number of keys in the buffer will be displayed top left.

If bit D is set (**BL key flash**)

If set then when a valid key is detected the back light will flash off and then on.

The default value of the flag is 12, i.e. A+B

11.4. ACK character

By default this is 6 but can be changed using the EERPOM Write command. The effect will not be implemented until the device is reset.

11.5. NACK character

By default this is 21 but can be changed using the EERPOM Write command. The effect will not be implemented until the device is reset.

11.6. Baud Rate

The Baud rate has the following values:

0. no valid
1. Baud rate is fixed at 2400
2. Baud rate is fixed at 4800
3. Baud rate is fixed at 9600 (default*)
4. Baud rate is fixed at 14400
5. Baud rate is fixed at 19200
6. Baud rate is fixed at 38400
7. Baud rate is fixed at 57600
8. Baud rate is fixed at 115200

11.7. CR Character

By default this is 13 which is the standard ASCII CR and the whole serial protocol relies on this being at the end of every command. It may be that this is unsuitable in some systems and so this can be changed.

11.8. Mode

This is used for testing purposes and should always be 0

11.9. Key Table Pointer

This holds the address of where the key table is. It would of course be possible to have other key tables stored by adjusting this pointer

11.10. Key table size

As it says

11.11. Debounce

This is mainly a delay before the keypad is read, it will not effect most keypads.

11.12. Repeat

This is a 16 bit number stored high and low. The actual value is found by trial and error. When a pad is touched the value is immediately recorded, if the finger is held there another, same value is recorded until the pad is untouched.

The time delay between each key record is determined by this value. The default value of 1256 (0x4e8) gives about 1/2 second.

11.13. Back Light

This is the back light condition at start up 0 is off 1 is on.

11.14. Key Table

When a key (made up of 2 or more channels) is touched it produces a unique scan code depending on which channels have been touched.

The key table is searched for the scan code and if it is found then the POSITION of the code is stored in the key buffer.

The actual code will depend on how the keypad is wired. By default it will read from top left to bottom right.

11.15. Sign On

The start up message is stored in EEPROM and so can be changed using the write to EEPROM command. The start of the message location is given in the table above.

The EEPROM is read from that location and will send any byte as data to the display. The font and position can also be sent by specifying a sequence <font><column><page>.

The sequence should be terminated with 0xff.

Example: "Hello" on first line "World" on second line, indented by 5 using font 2.

2,5,1,"Hello",2,5,1,"World",0xff

The command must be in the form of < font><column><page> and in the above example this follows the Hello and World.

12. Keypad Commands

12.1. I2C

Key pad commands **I2C address 0x6a (0x35 7 bit address)**

All I2C transactions start with a command for example:

8 bit pseudo code get value from buffer	**7 bit pseudo code get value from buffer**

i2c_start(0x7a) // write
i2c_putc(3)
i2c_stop()
i2c_start(0x7b) // read
value = i2c_getc()
i2c_stop()
	i2c_start()
	i2c_write(0x3d,3)
	i2c_stop()
	i2c_start()
	value = i2c_read(0x3d)
	i2c_stop()

The examples given in this table user notSMB (http://www.pichips.co.uk/index.php/RPi_Not_smBUS) that has three parameters:

optional value = bus.i2c(<i2c 7 bit address>[write to i2c],read from i2c), example :

value = bus.i2c(0x3d,[3,7],2)

This will address a device 0x3d, send bytes 3 and 7 and then read two bytes. In Python 'value' will be a list that can handle multiple bytes.

12.2. Serial (address 'j')

All serial commands start with the address, for convenience only the command values have been chosen to be in the printable rage. This makes debugging and experimentation easier.

A serial transaction is a packet that has the following elements:

<address><command and data><EOT>

The address for this device is the same as the I2C address by default 106 ('j'). All devices connected to the bus listen out for their address as the fist byte of a packet.

<command and data> The next byte will be a command as indicated in the table below followed by any necessary data. There is no separator for the fist byte after a command but subsequent data items should be separated by a comma or space unless the command says otherwise. As an example to get a particular key in the buffer the command is 'd'. If we look dor say key 3 then the complete command would be "jd3".

For commands that reqire more then one byte for example write to EEPROM then a comma is used, in this example: jW5,21 the value of 21 is written to address 5.

Serial	I2C	range	Default Value	EEPROM Location	Description
a	1	n/a			**Clears keypad buffer** **bus.i2c(0x3d[1],0)**
b	2	0-79			**Gets number of keys in buffer** Returns 0 if no keys are in the buffer **value = bus.i2c(0x3d[2],1)**
c	3	0-16			**Get key value from buffer** Key values are from 1 to 16 but this can be reduced or increased by configuring the key pad table in the EEPROM 0 is returned if no keys are in the buffer **value = bus.i2c(0x3d[3],3)**
d	4	0-16			**Key in Buffer** Checks to see if a particular key is in the buffer. It returns 0 if the key has not been

					found or a number representing the position of the key in the buffer. jd3 – returns n if 3 is in the buffer **value = bus.i2c(0x3d[4,keyToFind],1)**
e	5	0-255			**Get scan code** See 'tuning' section in the text for an explanation of what a scan code is. This will return a scan code if a pad is being touched and 0 if not. The command will produce unreliable results for a particular key however it may useful for something like a volume control. Any valid key in the key table will still be stored in the buffer so this should be cleared from time t time. **value = bus.i2c(0x3d[5],1)**
j	6	0-255			**Beep** Turns on beeper for number of mS, Example short beep 50mS **value = bus.i2c(0x3d[6,50],0)**
f	10	0-65535			**Returns 8 average values representing channels 2 through 9** This will in fact return **sixteen** values as I2C can only return 8 bits at a time. The value is sent as high low. To get the actual value requires something like: value = i2c_get() << 8 value = value + i2c_get() The value will now contain a 16 bit number. **value = bus.i2c(0x3d[10],16)**
g	11	0-65535			**Delta values for all channels** This returns 16 values (8 16 bit channels see command 10) The value returned represents channels 2 through 9, 2 is the first 9 is the last. The actual value returned is the difference between the average value and the touched value. If no pads are touched then the return value will be 0. The trigger value is important in that the touched value has to be lower then the average minus the trigger for this value to change from 0. If ny zeros are being returned when the pad is touched then the trigger is set too high. See also the 'tuning' section of this text. **value = bus.i2c(0x3d[11],16)**
i	21				**Sleep** This will put the device into sleep mode. Once in this mode the only way to wake is either by reset or an I2C read/write. The keypad will not work in sleep mode.

					bus.i2c(0x3d,[21],0)
	LCD	**Range**	**Time**		
k	30	0-255	500mS		**Reset LCD** Resets LCD. This is just the LCD and will not print the sign on message NOTE: This command requires 500mS to complete before sending the next I2C command. **bus.i2c(0x3d,[30],0)**
m	31	0-255			**LCD Command** Sends a command to the LCD controller; a command usually effects the way the display behaves. Example: to clear the screen: **bus.i2c(0x3d,[31,1],0)** Example to put cursor on the second line: **bus.i2c(0x3d,[31,0xc0],0)**
n	32	0-255			**LCD Data** Writes a **byte** to the display at the current cursor position. Example, writes '66' **bus.i2c(0x3d,[32,66],0)** **NOTE:** This will not write a character but a byte. See the section in the text on the LCD layout.
o	33	string of characters followed by EOL			**LCD String** Writes a string of characters to the display at the current cursor position. Example, writes 'abcd' **bus.i2c(0x3d,[32,61,62,62,64,13],0)** **WARNING:** Leaving the terminating EOL off may cause indeterminate results for the next command and even cause the I2C bus to lock up. EOL is defined in the EEPROM and the default is 13
p,timeout	34				**LCD Data Image** Send data string as binary data Serial requires a time out so the command to send an image would be something like jp,5000 For more information see text and **[1] Note**
q	35		10mS		**LCD Sign on** Displays the current sign on string stored in EEPROM, this is useful for testing **bus.i2c(0x3d,[35],0)**
r	36	0-1			**Sets Back light on/off** 1 is on, 0 is off

					bus.i2c(0x3d[1],0)
s	37	0-63			**Sets Contrast level** The level required depends on the supply voltage and there is a considerable difference between 3.3V and 5V settings. Normally 25 is okay for 5V and 45 is okay for 3.3V. Example, set contrast to 25 **bus.i2c(0x3d,[37,25],0)** This is a hybrid command and can also be derived from using the lcd command (31) above.
n/a	38				**Send data Bytes** This is instead of the image command. Some masters (Arduino) cannot handle sending large amounts of I2C data. **[1] Note**
t	40	1 to 3			**Sets Font** There are three built in fonts: 1 is 8x8 normal 2 is 8x8 bold 3 is 16x8 Example, to set font 2 **bus.i2c(0x3d,[40,2],0)**
u	41				**Homes cursor and clears screen** Example, **bus.i2c(0x3d,[41],0)**
v	42	0-127			**Sets column address** The column address is the X direction, after setting this the next item to be printed will start at the position specified Example, set column to 25 **bus.i2c(0x3d,[42,25],0)**
w	43	0-8			**Sets Page** See the text, there are 8 pages which represent the Y direction. Example, set page to 5 **bus.i2c(0x3d,[43,5],0)**
x	44	0-63			**Sets Initial Scroll line** This can be used to scroll the display, it can adjust the display in a vertical direction. The default is 0 Example, set to 5 **bus.i2c(0x3d,[44,5],0)**
y	45	Valid Char			**Writes a character** Writes a character at the current location with the selected font.

					Example, To write A **bus.i2c(0x3d,[45,65],0)**
					System
W	0x91	n=0-255 m=0-255			**Write to EEPROM** This will write a single byte to an EEPROM location **I2C Example write 23 to location 7** **s 0x91 7 23 p** or **bus.i2c(0x34,[0x91,7,23],0)** See www.pichips.co.uk and 'notsmb' for an explanation of the above nomenclature.
R	0x90	n=0-255 m=0-255			**Read from EEPROM** Reads a singe EEPROM values from a given address. **Example** To read from location 3: **s 0x90 3 r g-1 p** or **bus.i2c(0x34,[0x90,3]1)** See www.pichips.co.uk and 'notsmb' for an explanation of the above nomenclature.
D	0xa1				**Device ID** Returns two bytes representing a 16 bit number, high byte first **s 0xa1 r g-2 p** or **bus.i2c(0x34,[0xa1],2)** See www.pichips.co.uk and 'notsmb' for an explanation of the above nomenclature.
C	0x95				**Reset** Resets an individual device. This is a soft reset. A soft reset will normally be the same as a reset at start-up but this may not always be the case. Example **s 0x95 p** or **bus.i2c(0x34,[0x95],0)** See www.pichips.co.uk and 'notsmb' for an explanation of the above nomenclature.
E	0xA2				**EEPROM reset** Will reset the EEPROM back to the default values. **bus.i2c(0x3d,[20],0)**

V	0xa0				**Version** Returns the firmware version as two bytes **value = bus.i2c(0x3d[0xa0],2)**

[1] Note

The best (fastest) way to send an image is to stream the data. This can more easily be achieved using a serial interface. This is the purpose of the image command. For an I2C interface however some masters cannot handle a constant stream of data, internal buffers limit the number of bytes that can be sent. In this case the Send Data Bytes command is used to get the data out block by block.