# BV4618 User Guide

| Rev | Change |
| --- | --- |
| Nov 2010 | Preliminary |
| Nov 2010 | Version 1.1 I2C updated to remove timing issues. |
| Jan 2011 | Added support for Arduino |

Further resources at www.byvac.com

# Introduction

The BV4618 is a LCD character display controller and will interface with any display that uses the HD44780 or similar device. Typical displays are 16x2, 20x4 etc. The controller enables the display to be used with a serial or I2C interface without taking up and resource from the host.

The device will also interface to a 4x4 keypad thus it can be used as a complete user interface.
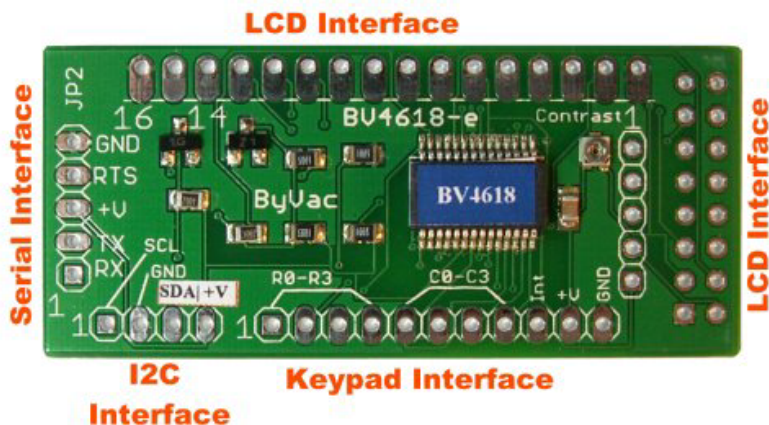


# Physical Interfacing

This device has no less than three interface options:

1) Serial via Microcontroller or conversion devise (MAX232)
2) Serial via RS232 (PC Com Port)
3) I2C

Only one interface at a time can be used and this is detected on reset. The default is serial via microcontroller. When pin 4 is connected this is detected as serial via RS232 and when the I2C is connected this is detected as I2C. The detection process is automatic.

## *Serial Via Microcontroller*

This option uses pins 1,2,3 and 5 of the serial interface. This interface it to the left hand side of the controller and pin 1 is marked on the PCB.
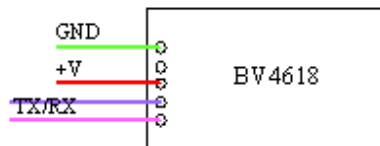
When connecting to a Microcontroller the TX pin on the microcontroller UART goes to the RX pin on the BV4618 and the RX pin on the microcontroller goes to the TX pin on the BV4618.

The voltage can range between 3V and 5V, however if 3.3V logic is being used then make sure that the LCD display will work on this low voltage. At the time of writing 3.3V LCD displays are rare and expensive compared to the 5V types.

## Serial Via RS232

The BV4618 can accept plus and minus 12V on pin 4 of the interface, it will also invert that signal so as to be compatible with the microcontroller interface. The device does NOT put out plus and minus 12V but most RS232 interfaces will tolerate this. Again the output is inverted because the RS232 describes an inverted signal.
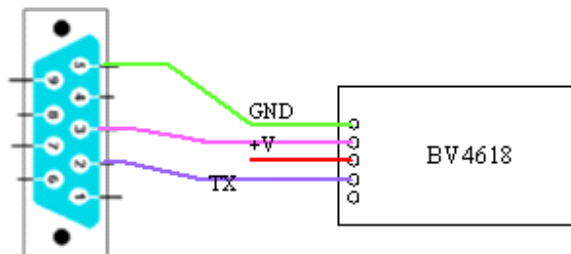


NOTE that pin 4 is marked RTS, in this mode it is actually RX that can accept plus and minus 12V. When using this interface power will need to be supplied to pin 3. This must not exceed 5V and preferably be regulated. The RS232 COM port interface cannot provide this power.

## I2C

The I2C interface is at the bottom of the controller and can be connected to any I2C bus. The bus must of course have pull up resistors, these are not provided on the BV4618
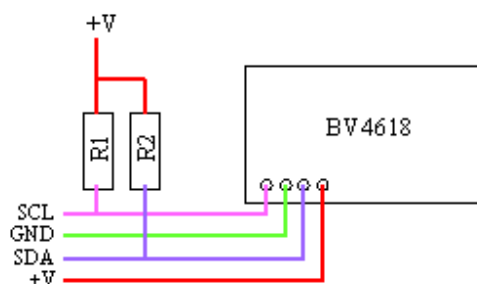


NOTE on version e of the PCB the SDA an V+ lines have been marked up wrongly, they are covered with a sticker. The above diagram shows the correct wiring.

## Keypad Interface

The keypad has interface has 5 outputs C0-C3 & nInt and 4 inputs R0-R3. The Column outputs (C0-C3) are constantly being changed, this is detected by the software when a key is pressed and the signal goes to one of the inputs. After a suitable denounce period the key is registered and the scan code placed in a buffer.

When there is a value in the key buffer the nInt line will go low and remain low until all of the keys have been read out.

The data sheet has full details on how this works.

# Software

The device is intended to be used with automated systems and software as it can provide a complete user interface. It is however advisable to understand how the device behaves by manually inputting commands. This will save time later and also help with better software functions later on. The first part of this section is therefore using the device manually from a terminal

## *Manual Use Serial*

For this we will use BV-COMM, the best place to download this is form here: http://www.asi.byvac.com/da_data.php it is free and does not need any installing, simply run the exe.

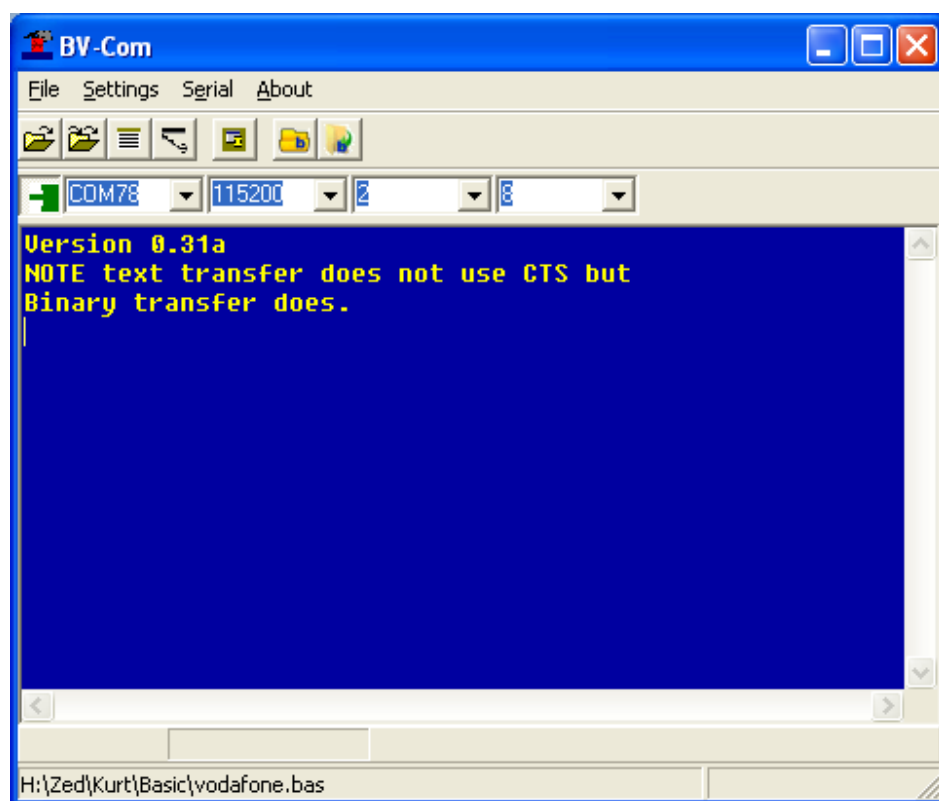You can connect via RS232 or via a BV101
(http://www.byvac.com/bv3/index.php?route=product/product&path=48&product_id=84 )

> The serial using RS232 works in exactly the same way, it just comes from a different source.

Start the terminal, select a suitable COM port and don't forget to press the connect

icon to turn it green.



If connecting Via a BV101 then the display will say "Press CR", if connecting via RS232 then the display will say "232 Press CR". This is automatically detected at reset and the display will operate in that particular mode until reset again. This text (Press CR) is designed to fit into 16 characters and so if using a smaller display some of the text will be missing.

The CR means carriage return or the enter key on the keyboard. Pressing <enter> will clear the display and produce a '*' on the terminal. This is how the Baud rate is determined and should always be the first byte sent*. The device does not echo any character and so to see what you are typing switch this on via the settings menu.
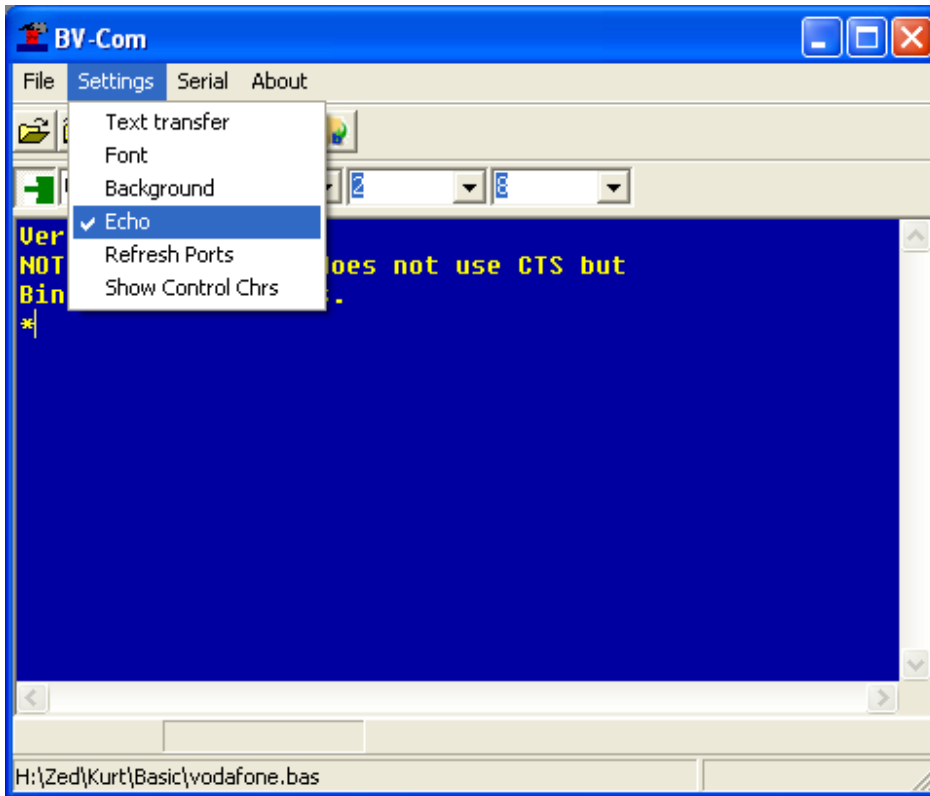
Any thing you type from now on will go to the display. To see the device ID type:

**esc[?31d**

The esc means the escape key on the keyboard, this is usually top right and is one keypress so the above command is a total of 6 key presses which will send 6 bytes to the device. (0x1b 0x5b 0x3f 0x33 0x31 0x64). The command is carried out as soon as the last character (d in this case) is received. This is how all commands work in serial mode.

It should be obvious from the above that the character case is important, for example esc[?31D is a different command from esc[?31d.

With the above setup it is possible to try out all of the commands and understand how they work before implementing in software. This is also a good debugging tool as you can type the commands that the software would do and see if there are any errors.

## Display Size

The default display size is 16x2, that is 2 lines by 16 characters. For this device the size of the display is important as it will determine when the text has reached the end of the line and when scrolling needs to begin. The first operation the software should carry out after determining the Baud rate is to set the display size to whatever is connected to the device. For example if a 20x4 display is connected the following commands are needed:

esc[4L // sets number of lines to 4

esc[20c // sets number of characters or columns

The device will revert back to its 16x2 default on reset so these commands must form part of the initialisation sequence. The commands should really be entered thus:

esc[4Lesc[20c

as a continuous stream because everything (including CR and LF) other than a command will be sent to the display, so having a CR between the two commands will cause the display to perform that action, i.e. go to the beginning of the line and down one.

## *Pseudo Code*

Using the display as part of a system requires sending and receiving serial data accurately. It is easy to miss a character or put an extra one in like CR and this will throw the whole function out. To demonstrate a typical application pseudo code will be used, it can then be translated to any particular language.

There are only a few simple functions needed:

```
com_putc(character)      // outputs a character to the com port
com_puts("string")       // outputs a string to the com port
com_getcq()              // returns 1 if there is a character waiting otherwise
returns 0
com_getc()               // gets a character from the com port
```

```
function init_display
      // put here code needed to initialise the com port
      com_putc(13);      // send CR to establish Baud rate
      wait 500ms         // **** needed to establish Baud rate ****
      // set up display for 20x4
      com_puts(0x1b);com_putc("[4L");  // 4 lines
      com_puts(0x1b);com_putc("[20c"); // 20 chars
      // set ACK to '*'
      com_puts(0x1b);com_putc("[42k"); // 42 is '*'
end function
```

In the above part of the initialisation is to use the ACK mechanism, this will prove useful when obtaining values from the keypad.

```
function send_text(string)
      com_puts(string);
end function
```

```
function position(row, col)
      com_putc(0x1b);com_puts("[");  // command is esc[r;c;H
      r$=convert_to_string(r);
      c$=convert_to_string(c);
      com_puts(r$+";"+c$+"H");
end function
```

The above two functions simply use the commands to send text and move the cursor.

```
function get_value(command$)
      val$="";
      com_puts(command$);     // send command
      while com_getcq() <> 0
            v = com_getc()
            if v = '*' break;  // finish when ACK received
            val$=val$+v;
      wend
return val$
```

```
end function
```

The above is a bit more complex but shows the principle of obtaining information from the device. Suppose we need to obtain the device ID which is command esc[?31d, the function may be used thus:

```
print get_value(0x1b+"[?31d");
```

To explain; the command will be sent to the device and the device will respond with "4618*". The get_value function will collect all of the bytes form the device until it encounters the ACK (*) byte which was set to this during initialization. ACK can of course be set to any value between 1 and 255, 42 just happens to be the ASCII code for '*'.

All values returned form the device are in text so these need to be converted to values to use in a program. For example suppose the key scan code contains 221 and this is returned using esc[k, e.g.

```
a$=get_value(0x1b+"[k");
```

The a$ will contain 3 bytes as a string "221". This can be a common pitfall by wrongly expecting a single byte value of 221.

# Arduino Serial Library

The Arduino serial now uses BSerail which can be found here: http://www.asi.byvac.com/da_data.php it is in the ASI link, ASI is not needed for this display but BSerial is.

The methods inherited from BDerail are:

**Methods inherited from BSerial:**
```
baud(rate)
handshake(uint8_t rtsPin, uint8_t ctsPin)
flush()
putch(char c)
unsigned char puts(char *s)
unsigned char buffer()
char getch()
```

A description of what they do and how they work are in the "ASI_library.zip" which is at the link indicated above under ASI.

Writing to the display, including setting up is simply sending text and so a single method is used for all of this and advantage is taken of the built in compiler escape codes.

To write to the display use puts(<string>) (from BSerial).

**Some Examples** (assuming bv is the instance of the class):

```
bv.puts("\e[2j");        // clear the screen
bv.puts("\e[4L\e[20c"); // set display to 4 lines by 20
bv.puts("\e[?26I"); // turn off the back light
vb.puts("\e[3;5HFred"); // print Fred on the third line, 5 column
```

Basically look at the command table in the datasheet and substitute esc for \e and that it.

**Input or Read**

This is always more difficult and to help some methods have been created to get information from the device. The best way to see this is to have a look at the example.

**Setup**

There are a few options for setting up how the device will work. The class itself has two or three options:

**BV4618_S bv(rxPin, txPin, int_pin);**

The first two parameters are obvious. The last one is a pin to specify for the interrupt pin on the device. This can then be used by the keyint() method and return the value on the pin. Specifying the pin value is optional.

When the class has been instantiated, it must be followed by begin:

**begin(<Baud rate>,<delay>,<ack>);**

Example begin(9600,0,'*');

When receiving data from a serial port, how do you know that you have received the last character? For example retrieving the number of keys in the buffer could return 5 or 12, how do you know to wait for the 2 of 12 and return 12 and not 1?

There are two methods used in this device: 1) is to simply wait until you are sure no more characters are forthcoming (about 100mS). 2) to have a special character at the end of every output from the device, this is called and ACK. In the example given and using an ACK character of '*' 12 would be 12*. So it is a simple matter to wait for the ACK and anything before it is the correct output. ACK can be any value from 1 to 255 (not 0) it does not have to be a printable character.

The advantage of this is speed as there is no unnecessary waiting, however in a noisy environment the ACK may be missed and this will spoil the output so just a delay may be advisable in that situation.

Using this with the begin(..) method, set delay to a value in mS, delay can also be used with ACK although I cant see there will be a reason to do so. If ACK is not required then set it to 0 and it will be ignored.

**setkeycodes(const char *codes)**

> The keybuffer can be interrogated with keyscan() or keys(). There is not need to set this if using keyscan as that will always return the scan code. It may be more convenient to map the keys to the values on the keytops. In this case supply this with a constant array of the scan key codes that represent the keytops. For example if the scan code is 0xdd when 0 is pressed then put 0xdd as the first byte in the constant array.

**char keyint()**

> Returns 0 if there is a key in the key buffer otherwise 1. The interrupt pin used must be set in the constructor.

**char keyscan()**

> This returns the scan code from the key buffer.

**char keys()**

> Gets the number of keys in the key buffer

**char key()**

> Gets 1 byte from the key buffer and looks it up in the constant character array provided to return a key value. If the value can't be found then it will return 0xff.

**clskeybuf()**

> Clears the key buffer. This is included but can be also set using puts(..)

**keydebounce(char db)**

> This is an arbitrary value (default 50) that will be applied as a delay to the keypad input. A 'noisy' keypad will require a greater value but it will cause a delay when entering values. This is included but can be also set using puts(..)

**int cmd(string)**

> This is the workhorse for getting information out of the display. It uses the delay and ACK as set in the begin(..) method. See the example of how to obtain the device ID using this method.

# I2C

The I2C interface uses a different part of the firmware and works differently from the serial firmware. The basic strategy is to display any byte received via I2C on the display unless it is a 0x1b, in which case a command will be expected.

To make things cleared the following notation will be used:

**s**     This is a start condition followed by byte 0x62 which is the 8 bit default write address of the device. In some systems this is a 2 function operation thus:

i2c_start();

i2c_write(0x62)

In practice these two operations can always be combined because the device address always follows the start condition.

**p**     This is the stop condition

**g-n**   This will receive bytes from the device where n is the number of bytes to receive for example g-3

Important: when a receiving a byte from the device the host will send the clock to clock out the byte from the device, on the 9th clock the host will either send an ACK or a NACK depending on whether it is the last byte the host wants. For example when receiving 3 bytes the host will send ACK for the first two bytes and NACK for the last byte. The device needs to know which is the last byte being sent. A properly specified host will do this.

**number**     Any number on its own will be sent to the I2C bus. Hex values.

**r**     Is a restart command always used after sending a command that will return some values. This may be implemented in some systems as:

i2c_stop()

i2c_start()

i2c_send(0x62+1)

Note that the 8bit address is now 0x63 because an odd numbered address will indicate to the device that data is being requested.


To send Hello to the display requires the following:
s 48 65 6c 6c 6f p


To clear the display would be:
s 1b 50 p


To receive the device ID:
s 1b 40 r g-2 p


When reading from the device, take for example the ID bytes, a command is first sent to request the bytes and then they are read out by the master. There is a very small delay between the request and the device presenting this data on the I2C bus. During this time clock stretching is used and if the host implements this then no further consideration is necessary as this will be correctly read by the host.

If the host does not implement clock stretching then a small delay between the requesting command and receiving command is recommended.


Initialization of a 20x4 the display would be:
s 1b 30 4 p        // set number of lines to 4
s 1b 31 20 p       // set number of characters to 20

# Arduino I2C Library

Support has been added by way of an Arduino Library

## *I2C*

Class BV4618_I

**BV4618(char i2adr, char int_pin)**

>    i2adr is the I2C address of the device

>    int_pin an input pin that is used for detecting when there are keys in the buffer.

**BV4618(char i2adr)**

>    Alternative is interrupt pin is not used. It is possible to pole the keyboard buffer with keys() to see if there is any keys in the buffer and so the interrupt will not be required. This does load the I2C bus and processor more though.

**setdisplay(char cols, char rows);**

>    Sets type of display or rather the number of lines and characters it has.

**putch(char c)**

>    Sends a single character to the display.

**puts(char *s)**

>    Sends a string to the i2c bus. The string must be a string, i.e. null terminated.

**crup()**

>    Moves cursor up one line.

**crdown()**

>    Moves cursor down one line.

**crright()**

>    Moves cursor right one space.

**crleft()**

>    Moves cursor left one space.

**rowcol(char line, char col)**

>    Moves the cursor to a specified row and column position.

**lineposition(char line, char pos)**

>    To move to the start of a particular line on a display an address is sent as a command. This address is normally 0x80,0xc0,0x94 and 0xd0 for lines 1 to 4 respectively. However on a 2 line x 40 display or some other combination this may not be the case. This command will set the starting line address for the line specifed.

**backlight(char blon)**

>    Turns the back light on and off, 1 is on.

**crhome()**

>    Returns the cursor to the home position – without clearing the screen

**int deviceid()**

>    Obtains the device ID which will be 4618, this is useful if multiple devices are on the same bus

**version(char *ver)**

>    Gets the firmware version as a string.

**setaddress(char newaddress)**

>    Sets a new I2C address for the device. The new address will not take effect until the device has been reset.

**reset()**

>    Resets the device.

**resetEEPROM()**

> Resets EEPROM back to the factory settings, this will also set the I2C address back to its default setting.

**delayms(char del)**

> Causes a delay of 'del' milliseconds, the display will stop responding for that time. As there is no ACK mechanism for I2C there is no way to tell if the delay has finished so it limits the usefulness of this command when using I2C

**delays(char del)**

> Causes a delay of 'del' seconds, the display will stop responding for that time. As there is no ACK mechanism for I2C there is no way to tell if the delay has finished so it limits the usefulness of this command when using I2C

**cls()**

> Clears the screen and sets the cursor in the home position

**clright()**

> Clears a line from the current cursor position to the end of the line.

**cleft()**

> Clears a line from the current cursor position to the beginning of the line.

**clall()**

> Clears the whole line that the cursor is presently on.

**setkeycodes(const char *codes)**

> The keybuffer can be interrogated with keyscan() or keys(). There is not need to set this if using keyscan as that will always return the scan code. It may be more convenient to map the keys to the values on the keytops. In this case supply this with a constant array of the scan key codes that represent the keytops. For example if the scan code is 0xdd when 0 is pressed then put 0xdd as the first byte in the constant array.

**char keyint()**

> Returns 0 if there is a key in the key buffer otherwise 1. The interrupt pin used must be set in the constructor.

**char keyscan()**

> This returns the scan code from the key buffer.

**char keys()**

> Gets the number of keys in the key buffer

**char key()**

> Gets 1 byte from the key buffer and looks it up in the constant character array provided to return a key value. If the value can't be found then it will return 0xff.

**clskeybuf()**

> Clears the key buffer.

**keydebounce(char db)**

> This is an arbitrary value (default 50) that will be applied as a delay to the keypad input. A 'noisy' keypad will require a greater value but it will cause a delay when entering values.

## *Example*

The software and example have been tested on the Arduino Nano with an ATmega328 fitted.

Wiring the display is straightforward. Connect the BV4618 to the Arduino using the +5V and GND lines. SDA goes to A4 and SCL goes to A5 on the Arduino. Pin 9 (D9) was used as the interrupt pin and this goes to the Int pin on the BV4618.

A 4x4 keybaord was also connected to the BV4618.

```
 1 BV4618_I di(0x31,9); // 4 x 20 display
 2
 3 void loop()
 4 {
 5 const char kb[]={0x7d,0xee,0xed,0xeb,0xde,0xdd,0xdb,0xbe,0xbd,0xbb,
 6 char tmp;
 7   // set up display geometry
 8   di.setdisplay(4,20);
 9   // set up keyboard scan codes, alter the above constant array
10   // this will depend on how the keypad has been wired
11   di.setkeycodes(kb);
12   // clear screen
13   di.cls();
14   di.puts("Display Test");
15   di.rowcol(2,1);
16   while(1) {
17     if(!di.keyint()) {
18       tmp=di.key();
19       if(tmp > 9) tmp+='A'-10;
20       else tmp+='0';
21       di.putch(tmp);
22       di.putch(' ');
23     }
24   }
25   while(1);
26 }
```

The above is an extract from the example. Line 1 initialises the class and passes the I2c address (7 bit) and the pin that is used for the interrupt. Line 5 is the constant character array that is used to translate scan codes to key numbers (there is a few values missing from this picture). When pressing the 0 key the scan code 0x7d was returned, when pressing 1 0xee was returned etc. The codes are set using di.setkeycodes();

Line 8 sets the display to a 4 line by 20 character. If this is omitted the display controller will default to 2x16.

Line 17 gets the state of the interrupt pin. Low means there are keys in the keypad buffer that can be read out. A simple translation to text is performed and printed out to the LCD display.

## Notes

### *Reset Interface Detection*

The following table will illustrate how the detection works:

| Priority | Mode | Pin 4 on Serial Connector | Pin 3 on I2C Connector |
|----------|------|---------------------------|------------------------|
| 1 | I2C | n/a | High by external pull up resistor [1] |
| 2 | Serial RS232 | low [2] | low (disconnected) |
| 3 | Serial NOT RS232 | high (disconnected) | low (disconnected) |

[1] The I2C bus specification requires pull up resistors and so simply connecting to a valid I2C bus will cause the device to use the I2C mode.

[2] The TX pin mark or idle should be at a negative or zero voltage, thus when connected to RS232 this is detected.