RPLCD Documentation

Release 1.2.0

Danilo Bargen

Nov 27, 2018

Contents

CHAPTER 1

About

RPLCD is a Python 2/3 Raspberry PI Character LCD library for the Hitachi HD44780 controller. It supports both GPIO (parallel) mode as well as boards with an I^2C port expander (e.g. the PCF8574 or the MCP23008). Furthermore it can use the pigpio library to control the (remote) LCD.

This library is inspired by Adafruit Industries' CharLCD library as well as by Arduino's LiquidCrystal library.

For GPIO mode, no external dependencies (except the `RPi.GPIO` library, which comes preinstalled on Raspbian) are needed to use this library. If you want to control LCDs via I^2C, then you also need the `python-smbus` library. If you want to control the LCD with `pigpio`, you have to install the pigpio library.

CHAPTER 2

Features

Already implemented

- Simple to use API
- Support for both 4 bit and 8 bit modes
- Support for parallel (GPIO), I^{2}C and `pigpio` connections
- Support for custom characters
- Support for backlight control circuits (including PWM dimming when using the `pigpio` backend)
- Support for contrast control (when using the `pigpio` backend)
- Built-in support for `A00` and `A02` character tables
- Python 2/3 compatible
- Caching: Only write characters if they changed
- No external dependencies (except `RPi.GPIO`, and `python-smbus` if you need I^{2}C support)

Wishlist

These things may get implemented in the future, depending on my free time and motivation:

- MicroPython port

Supported I^{2}C Port Expanders

- PCF8574 (used by a lot of I^{2}C LCD adapters on Ali Express)
- MCP23008 (used in Adafruit I^{2}C LCD backpack)
- MCP23017

CHAPTER 3

Contents

Installation

From PyPI

You can install RPLCD directly from PyPI using pip:

```
$ sudo pip install RPLCD
```

If you want to use I^2C, you also need smbus:

```
$ sudo apt-get install python-smbus
```

If you want to use pigpio, the easiest way is to install the library via your packet manager (select the Python version you need):

```
$ sudo apt-get install pigpio python-pigpio python3-pigpio
```

Manual Installation

You can also install the library manually without pip. Either just copy the scripts to your working directory and import them, or download the repository and run `python setup.py install` to install it into your Python package directory.

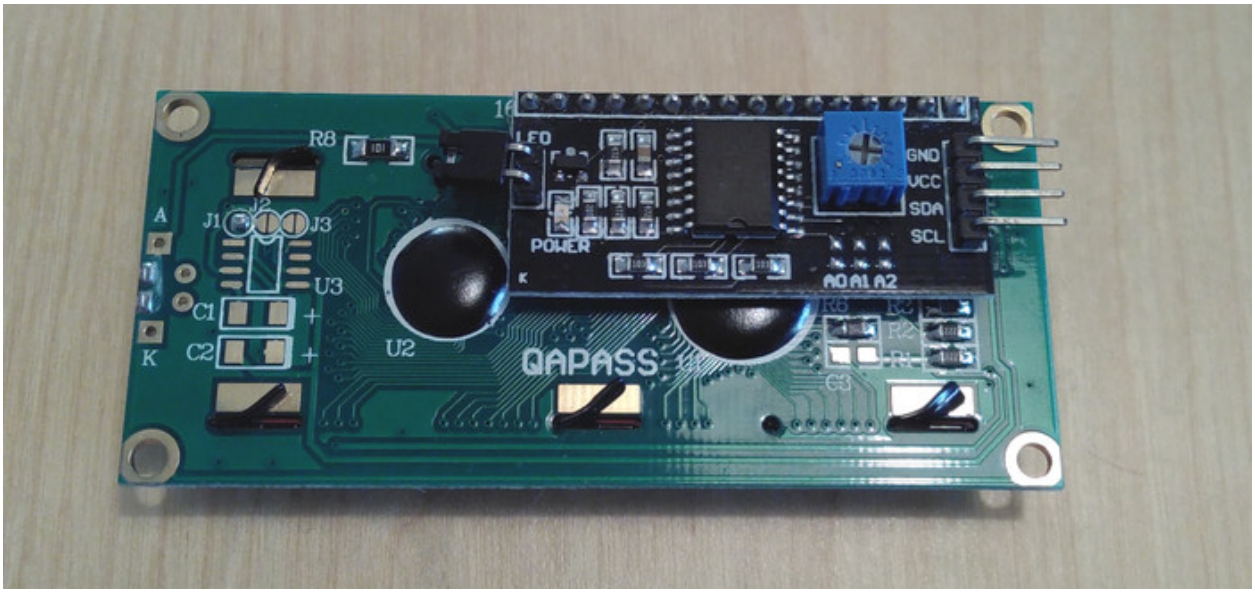
Getting Started

After you've *installed* RPLCD, you need two more steps to get started: Correct wiring and importing the library.

Wiring

Via I²C

The wiring is much simpler if you have a LCD module with I²C support. These boards usually have a "backpack board" and look similar to this:

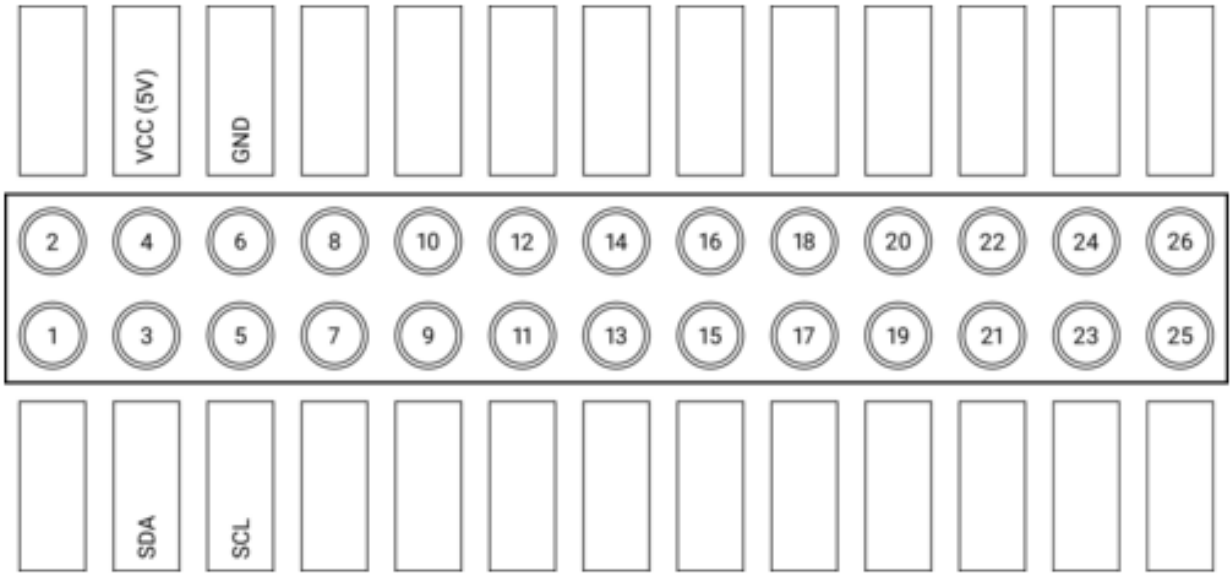


The board on this photo has a PCF8574 port expander chip on it. There are also boards with other chips, e.g. the Adafruit I²C/SPI LCD Backpack which uses an MCP23008 port expander.

First, connect the pins on the right with the Raspberry Pi:

- GND: Pin 6 (GND)
- VCC: Pin 4 (5V)
- SDA: Pin 3 (SDA)
- SCL: Pin 5 (SCL)

To make things clearer, here's a little visualization:



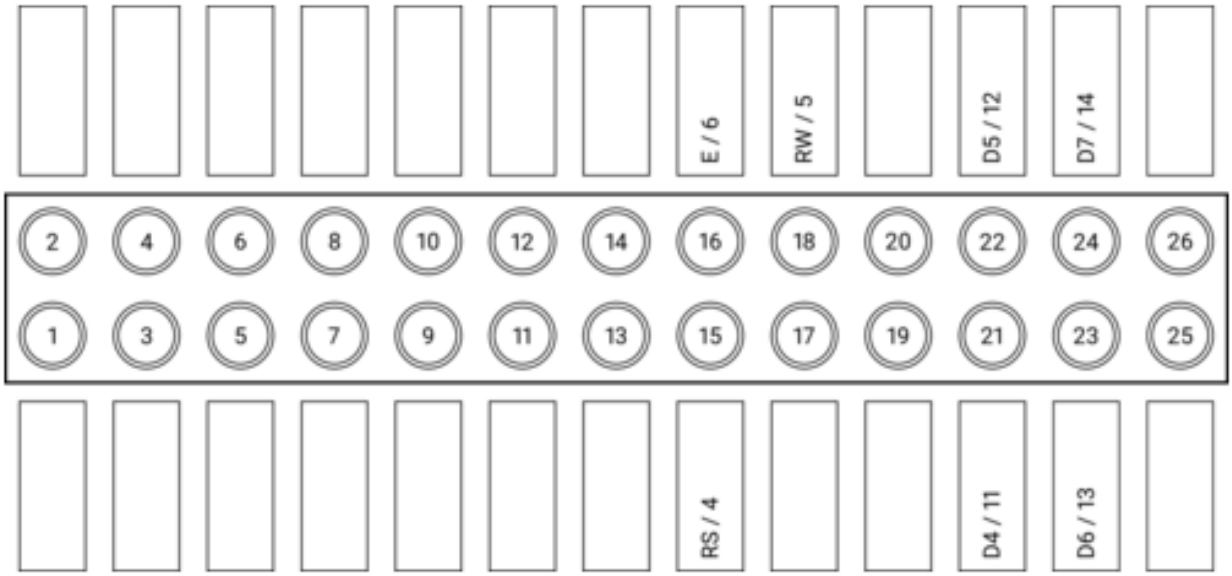
Via GPIO

If you don't have an I^2C version of the board, you can also connect the LCD Pins directly to the GPIO header of the Raspberry Pi.

If you don't know how to wire up the LCD to the Raspberry Pi, you could use this example wiring configuration in 4 bit mode (BOARD numbering scheme):

- RS: 15
- RW: 18
- E: 16
- Data 4-7: 21, 22, 23, 24

To make things clearer, here's a little visualization:



After wiring up the data pins, you have to connect the voltage input for controller and backlight, and set up the contrast circuit. As there are some differences regarding the hardware between different modules, please refer to the Adafruit tutorial to learn how to wire up these circuits.

Via pigpio

If you decide to use the `pigpio` library to control the LCD, follow the instructions set out above. Please keep in mind that the `pigpio` can only use the BCM numbering scheme.

The advantage of using the `pigpio` library is that you could control the backlight and contrast via PWM. You could also run the program on one computer (there is no need for this computer to be a Raspberry Pi) and control a LCD on any Raspberry Pi because `pigpio` follows a server-client approach. The disadvantage is, that it might be a bit slower when updating compared to using the GPIO library.

Initializing the LCD

Setup: I²C

First, import the RPLCD library from your Python script.

```
from RPLCD.i2c import CharLCD
```

Then create a new instance of the *CharLCD* class. For that, you need to know the address of your LCD. You can find it on the command line using the `sudo i2cdetect 1` command (or `sudo i2cdetect 0` on the original Raspberry Pi). In my case the address of the display was `0x27`. You also need to provide the name of the I²C port expander that your board uses. It should be written on the microchip that's soldered on to your board. Supported port expanders are the `PCF8574`, the `MCP23008` and the `MCP23017`.

```
lcd = CharLCD('PCF8574', 0x27)
```

If you want to customize the way the LCD is instantiated (e.g. by changing the number of columns and rows on your display or the I²C port), you can change the corresponding parameters. Example:

```
lcd = CharLCD(i2c_expander='PCF8574', address=0x27, port=1,
              cols=20, rows=4, dotsize=8,
              charmap='A02',
              auto_linebreaks=True,
              backlight_enabled=True)
```

Setup: GPIO

First, import the RPLCD library from your Python script.

```
from RPLCD.gpio import CharLCD
```

Then create a new instance of the *CharLCD* class. If you have a 20x4 LCD, you must at least specify the numbering mode and the pins you used:

```
lcd = CharLCD(pin_rs=15, pin_rw=18, pin_e=16, pins_data=[21, 22, 23, 24],
              numbering_mode=GPIO.BOARD)
```

If you want to customize the way the LCD is instantiated (e.g. by changing the pin configuration or the number of columns and rows on your display), you can change the corresponding parameters. Here's a full example:

```
from RPi import GPIO

lcd = CharLCD(pin_rs=15, pin_rw=18, pin_e=16, pins_data=[21, 22, 23, 24],
              numbering_mode=GPIO.BOARD,
              cols=20, rows=4, dotsize=8,
              charmap='A02',
              auto_linebreaks=True)
```

Setup: pigpio

First, import the the pigpio and RPLCD libraries from your Python script.

```
import pigpio
from RPLCD.pigpio import CharLCD
```

Then create a connection to the pigpio daemon

```
pi = pigpio.pi()
```

and create a new instance of the *CharLCD* class. If you have a 20x4 LCD, you must at least specify the previously initiated pigpio connection and the pins you used:

```
lcd = CharLCD(pi,
              pin_rs=15, pin_rw=18, pin_e=16, pins_data=[21, 22, 23, 24])
```

If you want to customize the way the LCD is instantiated (e.g. by changing the pin configuration or the number of columns and rows on your display), you can change the corresponding parameters. Here's a full example:

```
import pigpio
from RPLCD.pigpio import CharLCD

pi = pigpio.pi()
lcd = CharLCD(pi,
              pin_rs=15, pin_rw=18, pin_e=16, pins_data=[21, 22, 23, 24],
              cols=20, rows=4, dotsize=8,
              charmap='A02',
              auto_linebreaks=True)
```

Writing Data

Now you can write a string to the LCD:

```
lcd.write_string('Hello world')
```

To clean the display, use the `clear()` method:

```
lcd.clear()
```

You can control line breaks with the newline (`\n`, moves down 1 line) and carriage return (`\r`, moves to beginning of line) characters.

```
lcd.write_string('Hello\r\n  World!')
```

And you can also set the cursor position directly:

```
lcd.cursor_pos = (2, 0)
```

Usage

Make sure to read the *Getting Started* section if you haven't done so yet.

Writing To Display

Regular text can be written to the *CharLCD* instance using the *write_string()* method. It accepts unicode strings (`str` in Python 3, `unicode` in Python 2).

The cursor position can be set by assigning a `(row, col)` tuple to *cursor_pos*. It can be reset to the starting position with *home()*.

Line feed characters (`\n`) move down one line and carriage returns (`\r`) move to the beginning of the current line.

```
lcd.write_string('Raspberry Pi HD44780')
lcd.cursor_pos = (2, 0)
lcd.write_string('https://github.com/\n\rdbrgn/RPLCD')
```

You can also use the convenience functions `cr()`, `lf()` and `crlf()` to write line feed (`\n`) or carriage return (`\r`) characters to the display.

```
lcd.write_string('Hello')
lcd.crlf()
lcd.write_string('world!')
```

After your script has finished, you may want to close the connection and optionally clear the screen with the *close()* method.

```
lcd.close(clear=True)
```

When using a GPIO based LCD, this will reset the GPIO configuration. Note that doing this without clearing can lead to undesired effects on the LCD, because the GPIO pins are floating (not configured as input or output anymore).

Clearing the Display

You can clear the display by using the *clear()* method. It will overwrite the data with blank characters and reset the cursor position.

Alternatively, if you want to hide all characters but keep the data in the LCD memory, set the *display_enabled* property to `False`.

Character Maps

RPLCD supports the two most commonly used character maps for HD44780 style displays: A00 and A02. You can find them on pages 17 and 18 of the datasheet.

The default character map is `A02`. If you find that some of the characters you are writing to the display turn out wrong, then try using the `A00` character map:

```
lcd = CharLCD(..., charmap='A00')
```

As a rule of thumb, if your display can show Japanese characters, it uses `A00`, otherwise `A02`. To show the entire character map on your LCD, you can use the `show_charmap` target of the `rplcd-tests` script.

Should you run into the situation that your character map does not seem to match either the `A00` or the `A02` tables, please open an issue on Github.

The same thing counts if you have a character that should be supported by your character map, but which doesn't get written correctly to the display. Let me know by opening an issue!

In case you need a character that is not included in the default device character map, there is a possibility to create custom characters and write them into the HD44780 CGRAM. For more information, see the *Creating Custom Characters* section.

Creating Custom Characters

The HD44780 supports up to 8 user created characters. A character is defined by a 8x5 bitmap. The bitmap should be a tuple of 8 numbers, each representing a 5 pixel row. Each character is written to a specific location in CGRAM (numbers 0-7).

```
>>> lcd = CharLCD(...)
>>> smiley = (
...     0b00000,
...     0b01010,
...     0b01010,
...     0b00000,
```

```
...     0b10001,
...     0b10001,
...     0b01110,
...     0b00000,
... )
>>> lcd.create_char(0, smiley)
```

To actually show a stored character on the display, you can use hex escape codes with the location number you specified previously. For example, to write the character at location 3:

```
>>> lcd.write_string('\x03')
```

The escape code can also be embedded in a longer string:

```
>>> lcd.write_string('Hello there \x03')
```

The following tool can help you to create your custom characters: https://omerk.github.io/lcdchargen/

Changing the Cursor Appearance

The cursor appearance can be changed by setting the *cursor_mode* property to one of the following three values:

- `hide` – No cursor will be displayed
- `line` – The cursor will be indicated with an underline
- `blink` – The cursor will be indicated with a blinking square

Backlight Control

I²C

If you're using an LCD connected through the I²C bus, you can directly turn on the backlight using the boolean *backlight_enabled* property.

GPIO

By setting the `pin_backlight` parameter in the *CharLCD* constructor, you can control a backlight circuit.

First of all, you need to build an external circuit to control the backlight, most LCD modules don't support it directly. You could do this for example by using a transistor and a pull-up resistor. Then connect the transistor to a GPIO pin and configure that pin using the `pin_backlight` parameter in the constructor. If you use an active high circuit instead of active low, you can change that behavior by setting the `backlight_mode` to either `active_high` or `active_low`. Now you can toggle the *backlight_enabled* property to turn the backlight on and off.

pigpio

When using the `pigpio` library, it is also possible to control the backlight with PWM.

The API is compatible to the backlight control of I²C and GPIO explained above, but the *backlight_enabled* property (and parameter) now also accepts a value between `0` and `1` as a backlight level (`0` or `False` turns the backlight off, `1` or `True` turns it on). The perceived brightness of the backlight should roughly correspond to the given value.

The PWM dimming of the backlight has to be enabled explicitly by setting the `backlight_pwm` parameter to `True` during initialization of *`CharLCD`*. If this parameter is `False` (the default value), the interface only switches the backlight on and off. If this parameter is a number, dimming of the backlight is enabled and the value is interpreted as the PWM frequency in Hertz.

Contrast Control

This is currently only possible with the pigpio backend.

pigpio

The API is similar to that controlling the backlight. The `pin_contrast` specifies the pin connected to the LCDs contrast input. The `contrast_mode` can be `active_high` or `active_low` and the `contrast_pwm` sets the PWM frequency.

The *`contrast`* property sets the contrast level. It should be a value between `0` and `1`. It is also recognized as a parameter to *`CharLCD`* to set the initial contrast level.

If you don't set the `pin_contrast` parameter, the contrast control stays disabled.

Automatic Line Breaks

By default, RPLCD tries to automatically insert line breaks where appropriate to achieve (hopefully) intuitive line wrapping.

Part of these rules is that manual linebreaks (either `\r\n` or `\n\r`) that immediately follow an automatically issued line break are ignored.

If you want more control over line breaks, you can disable the automatic system by setting the `auto_linebreaks` parameter of the `CharLCD` constructor to `False`.

```
lcd = CharLCD(..., auto_linebreaks=False)
```

Scrolling Text

I wrote a blogpost on how to implement scrolling text: https://blog.dbrgn.ch/2014/4/20/scrolling-text-with-rplcd/

To see the result, go to https://www.youtube.com/watch?v=49RkQeiVTGU.

Raw Commands

You can send raw commands to the LCD with *`command()`* and write a raw byte to the LCD with *`write()`*. For more information, please refer to the Hitachi HD44780 datasheet.

API

CharLCD (I²C)

The main class for controlling I²C connected LCDs.

class RPLCD.i2c.**CharLCD**(*i2c_expander*, *address*, *expander_params=None*, *port=1*, *cols=20*, *rows=4*, *dotsize=8*, *charmap='A02'*, *auto_linebreaks=True*, *backlight_enabled=True*)

CharLCD via PCF8574 I2C port expander:

Pin mapping:

7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 D7 | D6 | D5 | D4 | BL | EN | RW | RS

CharLCD via MCP23008 and MCP23017 I2C port expanders:

Adafruit I2C/SPI LCD Backback is supported.

Warning: You might need a level shifter (that supports i2c) between the SCL/SDA connections on the MCP chip / backpack and the Raspberry Pi. Or you might damage the Pi and possibly any other 3.3V i2c devices connected on the i2c bus. Or cause reliability issues. The SCL/SDA are rated 0.7*VDD on the MCP23008, so it needs 3.5V on the SCL/SDA when 5V is applied to drive the LCD.

The MCP23008 and MCP23017 needs to be connected exactly the same way as the backpack.

For complete schematics see the adafruit page at: https://learn.adafruit.com/i2c-spi-lcd-backpack/

4-bit operation. I2C only supported.

Pin mapping:

7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 BL | D7 | D6 | D5 | D4 | E | RS | -

Parameters

- **address** (*int*) – The I2C address of your LCD.
- **i2c_expander** (*string*) – Set your I²C chip type. Supported: "PCF8574", "MCP23008", "MCP23017".
- **expander_params** (*dictionary*) – Parameters for expanders, in a dictionary. Only needed for MCP23017 gpio_bank - This must be either `A` or `B`

  If you have a HAT, A is usually marked 1 and B is 2

  Example: expander_params={'gpio_bank': 'A'}
- **port** (*int*) – The I2C port number. Default: `1`.
- **cols** (*int*) – Number of columns per row (usually 16 or 20). Default: `20`.
- **rows** (*int*) – Number of display rows (usually 1, 2 or 4). Default: `4`.
- **dotsize** (*int*) – Some 1 line displays allow a font height of 10px. Allowed: 8 or 10. Default: `8`.
- **charmap** (*str*) – The character map used. Depends on your LCD. This must be either `A00` or `A02`. Default: `A02`.
- **auto_linebreaks** (*bool*) – Whether or not to automatically insert line breaks. Default: `True`.
- **backlight_enabled** (*bool*) – Whether the backlight is enabled initially. Default: `True`.

backlight_enabled

Whether or not to enable the backlight. Either `True` or `False`.

clear()

Overwrite display with blank characters and reset cursor position.

close(*clear=False*)

command(*value*)
Send a raw command to the LCD.

cr()
Write a carriage return (`\r`) character to the LCD.

create_char(*location*, *bitmap*)
Create a new character.

The HD44780 supports up to 8 custom characters (location 0-7).

Parameters

- **location** (*int*) – The place in memory where the character is stored. Values need to be integers between 0 and 7.
- **bitmap** (*tuple of int*) – The bitmap containing the character. This should be a tuple of 8 numbers, each representing a 5 pixel row.

Raises **AssertionError** – Raised when an invalid location is passed in or when bitmap has an incorrect size.

Example:

```
>>> smiley = (
...     0b00000,
...     0b01010,
...     0b01010,
...     0b00000,
...     0b10001,
...     0b10001,
...     0b01110,
...     0b00000,
... )
>>> lcd.create_char(0, smiley)
```

crlf()
Write a line feed and a carriage return (`\r\n`) character to the LCD.

cursor_mode
How the cursor should behave (`hide`, `line` or `blink`).

cursor_pos
The cursor position as a 2-tuple (row, col).

display_enabled
Whether or not to display any characters.

home()
Set cursor to initial position and reset any shifting.

lf()
Write a line feed (`\n`) character to the LCD.

shift_display(*amount*)
Shift the display. Use negative amounts to shift left and positive amounts to shift right.

text_align_mode
The text alignment (`left` or `right`).

write(*value*)
Write a raw byte to the LCD.

write_shift_mode
The shift mode when writing (`cursor` or `display`).

write_string(*value*)
Write the specified unicode string to the display.

To control multiline behavior, use newline (`\n`) and carriage return (`\r`) characters.

Lines that are too long automatically continue on next line, as long as `auto_linebreaks` has not been disabled.

Make sure that you're only passing unicode objects to this function. The unicode string is then converted to the correct LCD encoding by using the charmap specified at instantiation time.

If you're dealing with bytestrings (the default string type in Python 2), convert it to a unicode object using the `.decode(encoding)` method and the appropriate encoding. Example for UTF-8 encoded strings:

```
>>> bstring = 'Temperature: 30°C'
>>> bstring
'Temperature: 30Â°C'
>>> bstring.decode('utf-8')
u'Temperature: 30°C'
```

CharLCD (GPIO)

The main class for controlling GPIO (parallel) connected LCDs.

class RPLCD.gpio.**CharLCD**(*numbering_mode=None*, *pin_rs=None*, *pin_rw=None*, *pin_e=None*, *pins_data=None*, *pin_backlight=None*, *backlight_mode='active_low'*, *backlight_enabled=True*, *cols=20*, *rows=4*, *dotsize=8*, *charmap='A02'*, *auto_linebreaks=True*)
Character LCD controller.

The default pin numbers are based on the BOARD numbering scheme (1-26).

You can save 1 pin by not using RW. Set `pin_rw` to `None` if you want this.

Parameters

- **pin_rs** (*int*) – Pin for register select (RS). Default: `15`.
- **pin_rw** (*int*) – Pin for selecting read or write mode (R/W). Set this to `None` for read only mode. Default: `18`.
- **pin_e** (*int*) – Pin to start data read or write (E). Default: `16`.
- **pins_data** (*list of int*) – List of data bus pins in 8 bit mode (DB0-DB7) or in 4 bit mode (DB4-DB7) in ascending order. Default: `[21, 22, 23, 24]`.
- **pin_backlight** (*int*) – Pin for controlling backlight on/off. Set this to `None` for no backlight control. Default: `None`.
- **backlight_mode** (*str*) – Set this to either `active_high` or `active_low` to configure the operating control for the backlight. Has no effect if pin_backlight is `None`
- **backlight_enabled** (*bool*) – Whether the backlight is enabled initially. Default: `True`. Has no effect if pin_backlight is `None`
- **numbering_mode** (*int*) – Which scheme to use for numbering of the GPIO pins, either `GPIO.BOARD` or `GPIO.BCM`. Default: `GPIO.BOARD` (1-26).
- **rows** (*int*) – Number of display rows (usually 1, 2 or 4). Default: `4`.

- **cols** (*int*) – Number of columns per row (usually 16 or 20). Default `20`.
- **dotsize** (*int*) – Some 1 line displays allow a font height of 10px. Allowed: `8` or `10`. Default: `8`.
- **charmap** (*str*) – The character map used. Depends on your LCD. This must be either `A00` or `A02`. Default: `A02`.
- **auto_linebreaks** (*bool*) – Whether or not to automatically insert line breaks. Default: `True`.

backlight_enabled
Whether or not to turn on the backlight.

clear()
Overwrite display with blank characters and reset cursor position.

close(*clear=False*)

command(*value*)
Send a raw command to the LCD.

cr()
Write a carriage return (`\r`) character to the LCD.

create_char(*location*, *bitmap*)
Create a new character.

The HD44780 supports up to 8 custom characters (location 0-7).

Parameters

- **location** (*int*) – The place in memory where the character is stored. Values need to be integers between 0 and 7.
- **bitmap** (*tuple of int*) – The bitmap containing the character. This should be a tuple of 8 numbers, each representing a 5 pixel row.

Raises **AssertionError** – Raised when an invalid location is passed in or when bitmap has an incorrect size.

Example:

```
>>> smiley = (
...     0b00000,
...     0b01010,
...     0b01010,
...     0b00000,
...     0b10001,
...     0b10001,
...     0b01110,
...     0b00000,
... )
>>> lcd.create_char(0, smiley)
```

crlf()
Write a line feed and a carriage return (`\r\n`) character to the LCD.

cursor_mode
How the cursor should behave (`hide`, `line` or `blink`).

cursor_pos
The cursor position as a 2-tuple (row, col).

display_enabled
Whether or not to display any characters.

home()
Set cursor to initial position and reset any shifting.

lf()
Write a line feed (`\n`) character to the LCD.

shift_display(*amount*)
Shift the display. Use negative amounts to shift left and positive amounts to shift right.

text_align_mode
The text alignment (`left` or `right`).

write(*value*)
Write a raw byte to the LCD.

write_shift_mode
The shift mode when writing (`cursor` or `display`).

write_string(*value*)
Write the specified unicode string to the display.

To control multiline behavior, use newline (`\n`) and carriage return (`\r`) characters.

Lines that are too long automatically continue on next line, as long as `auto_linebreaks` has not been disabled.

Make sure that you're only passing unicode objects to this function. The unicode string is then converted to the correct LCD encoding by using the charmap specified at instantiation time.

If you're dealing with bytestrings (the default string type in Python 2), convert it to a unicode object using the `.decode(encoding)` method and the appropriate encoding. Example for UTF-8 encoded strings:

```
>>> bstring = 'Temperature: 30°C'
>>> bstring
'Temperature: 30Â°C'
>>> bstring.decode('utf-8')
u'Temperature: 30°C'
```

CharLCD (pigpio)

The main class for controlling LCDs through pigpio.

class RPLCD.pigpio.**CharLCD**(*pi*, *pin_rs=None*, *pin_rw=None*, *pin_e=None*, *pin_e2=None*, *pins_data=None*, *pin_backlight=None*, *backlight_mode='active_low'*, *backlight_pwm=False*, *backlight_enabled=True*, *pin_contrast=None*, *contrast_mode='active_high'*, *contrast_pwm=None*, *contrast=0.5*, *cols=20*, *rows=4*, *dotsize=8*, *charmap='A02'*, *auto_linebreaks=True*)
Character LCD controller.

The pin numbers are based on the BCM numbering scheme!

You can save 1 pin by not using RW. Set `pin_rw` to `None` if you want this.

Parameters

- **pi** (*pigpio.pi object*) – A pigpio.pi object to access the GPIOs.
- **pin_rs** (*int*) – Pin for register select (RS). Default: `15`.

- **pin_rw** (*int*) – Pin for selecting read or write mode (R/W). Set this to `None` for read only mode. Default: `18`.
- **pin_e** (*int*) – Pin to start data read or write (E). Default: `16`.
- **pins_data** (*list of int*) – List of data bus pins in 8 bit mode (DB0-DB7) or in 4 bit mode (DB4-DB7) in ascending order. Default: `[21, 22, 23, 24]`.
- **pin_backlight** (*int*) – Pin for controlling backlight on/off. Set this to `None` for no backlight control. Default: `None`.
- **backlight_mode** (*str*) – Set this to either `active_high` or `active_low` to configure the operating control for the backlight. Has no effect if pin_backlight is `None`.
- **backlight_pwm** (*bool or int*) – Set this to `True`, if you want to enable PWM for the backlight with the default PWM frequency. Set this to the frequency (in Hz) of the PWM for the backlight or to `False` to disable PWM for the backlight. Default: `False`. Has no effect if pin_backlight is `None`.
- **backlight_enabled** (*bool or float*) – Whether the backlight is enabled initially. If backlight_pwm is `True`, this can be a value between 0 and 1, specifying the initial backlight level. Default: `True`. Has no effect if pin_backlight is `None`.
- **pin_contrast** (*int*) – Pin for controlling LCD contrast. Set this to `None` for no contrast control. Default: `None`.
- **contrast_mode** (*str*) – Set this to either `active_high` or `active_low` to configure the operating control for the LCD contrast. Has no effect if pin_contrast is `None`.
- **contrast_pwm** (*int*) – Set this to the frequency (in Hz) of the PWM for the LCD contrast if you want to change the default value. Has no effect if pin_contrast is `None`.
- **contrast** (*float*) – A value between 0 and 1, specifying the initial LCD contrast. Default: 0.5. Has no effect if pin_contrast is `None`
- **rows** (*int*) – Number of display rows (usually 1, 2 or 4). Default: `4`.
- **cols** (*int*) – Number of columns per row (usually 16 or 20). Default `20`.
- **dotsize** (*int*) – Some 1 line displays allow a font height of 10px. Allowed: `8` or `10`. Default: `8`.
- **charmap** (*str*) – The character map used. Depends on your LCD. This must be either `A00` or `A02`. Default: `A02`.
- **auto_linebreaks** (*bool*) – Whether or not to automatically insert line breaks. Default: `True`.

backlight_enabled
Turn on/off or set the brightness of the backlight.

clear()
Overwrite display with blank characters and reset cursor position.

close(*clear=False*)

command(*value*)
Send a raw command to the LCD.

contrast
Set the LCD contrast.

cr()
Write a carriage return (`\r`) character to the LCD.

create_char(*location*, *bitmap*)
Create a new character.

The HD44780 supports up to 8 custom characters (location 0-7).

Parameters

- **location** (*int*) – The place in memory where the character is stored. Values need to be integers between 0 and 7.
- **bitmap** (*tuple of int*) – The bitmap containing the character. This should be a tuple of 8 numbers, each representing a 5 pixel row.

Raises **AssertionError** – Raised when an invalid location is passed in or when bitmap has an incorrect size.

Example:

```
>>> smiley = (
...     0b00000,
...     0b01010,
...     0b01010,
...     0b00000,
...     0b10001,
...     0b10001,
...     0b01110,
...     0b00000,
... )
>>> lcd.create_char(0, smiley)
```

crlf()
Write a line feed and a carriage return (`\r\n`) character to the LCD.

cursor_mode
How the cursor should behave (`hide`, `line` or `blink`).

cursor_pos
The cursor position as a 2-tuple (row, col).

display_enabled
Whether or not to display any characters.

home()
Set cursor to initial position and reset any shifting.

lf()
Write a line feed (`\n`) character to the LCD.

shift_display(*amount*)
Shift the display. Use negative amounts to shift left and positive amounts to shift right.

text_align_mode
The text alignment (`left` or `right`).

write(*value*)
Write a raw byte to the LCD.

write_shift_mode
The shift mode when writing (`cursor` or `display`).

write_string(*value*)
Write the specified unicode string to the display.

To control multiline behavior, use newline (`\n`) and carriage return (`\r`) characters.

Lines that are too long automatically continue on next line, as long as `auto_linebreaks` has not been disabled.

Make sure that you're only passing unicode objects to this function. The unicode string is then converted to the correct LCD encoding by using the charmap specified at instantiation time.

If you're dealing with bytestrings (the default string type in Python 2), convert it to a unicode object using the `.decode(encoding)` method and the appropriate encoding. Example for UTF-8 encoded strings:

```
>>> bstring = 'Temperature: 30°C'
>>> bstring
'Temperature: 30Â°C'
>>> bstring.decode('utf-8')
u'Temperature: 30°C'
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index