pyLCI Documentation

Release 1.0

Arsenijs Picugins (CRImier)

Sep 21, 2017

Contents

pyLCI stands for Python-based Linux Control Interface. It's an external interface for configuration of your Linux devices in an easy and quick way.

It can be used on:

- Embedded devices (where dependency on Python isn't problematic to satisfy), such as OpenWRT-powered routers
- Single-board computers, including, but not limited to Raspberry Pi, BeagleBone and many others
- Tablets and laptops
- Servers
- Desktop PCs and HTPCs

CHAPTER 1

Guides:

- *Hardware guide*
- *Setup and configuration guide*
- *Debugging issues*
- *Managing and developing applications*

pyLCI system - the software part (pyLCI daemon) and the hardware part - typically consisting of a character LCD and a keypad of some sort.

The pyLCI daemon consists of 5 parts:

1. *Input system*
2. *Output system*
3. *UI elements*
4. *Applications*
5. Glue logic (mostly main.py launcher)

Development plans

FAQ&contacts

Hardware guide

Absolute necessities:

- A HD44780-compatible character display, from 16x2 to 20x4. They are cheap and available in most electronics shops in the world, as well as in most starter kits.
- At least 5 simple pushbuttons (very cheap and salvageable from just about anything), or a USB keyboard/numpad.

Some remarks:

- There are Raspberry Pi shields which have a character LCDs and some buttons. They're good, too - as long as they're in the "supported" list.

  Supported shields:

  - PiFaceCAD Raspberry Pi shield
  - Adafruit 16x2 Character LCD + Keypad for Raspberry Pi
  - Chinese "LCD RGB KEYPAD ForRPI" shield (black PCB, pin-to-pin copy of aforementioned Adafruit shield)

Ways to connect your hardware:

- Is it a shield pyLCI supports? Great, plug it on top of your Raspberry Pi and you're done!
- If all you have is the character display and some buttons, you can:
  - Connect them over GPIO (works for both screen and buttons) (only Raspberry Pi GPIO supported at the moment)
  - Connect them over I2C using a PCF8574 expander (works for both screen and buttons, 1$ on eBay)
- When assembling the hardware yourself, you can easily combine connection methods - for example, connect your LCD over I2C and buttons over GPIO, or use a shield for LCD and use a USB numpad.

Afterwards, follow to the *pyLCI setup* part.

Buying/choosing guide

- Want something cheap and minimum effort? Get a "LCD RGB KEYPAD ForRPI" shield. It's 6$, you can find it on eBay just by searching "Raspberry Pi LCD shield" and sorting the list by "Lowest price first". It'll take its time to arrive, but it's a great value for the price.
- Want something quickly and minimum effort? Get a PiFaceCAD shield, or an Adafruit one. They're sold by distributors in UK/USA, and will arrive quickly. Moreover, they're nicely made.
- Want something quickly and cheaply? You can assemble your own hardware from what you have. I2C expanders come in handy when you need to save pins, but connecting things through GPIO is a good alternative.
- genindex
- modindex
- search

Installing and updating pyLCI

```
git clone https://github.com/CRImier/pyLCI
cd pyLCI/
./setup.sh #Install main dependencies and create a install directory
./config.sh #Install dependencies for your input&output devices
nano config.json #Describe your input&outputdevices (if you have a supported shield,␣
↪previous step will edit this for you)
sudo python main.py #Start the system to test your configuration - do screen and␣
↪buttons work OK?
```

```
#Once configured:
./update.sh #Transfer the working system to your install directory
```

Note: Behind the scenes:

When you run `./setup.sh`, pyLCI is copied to `/opt/pylci`, this is done to make autorun code easier and allow experimentation while making it harder to lock you out of the system if pyLCI is your main control interface. `/opt/pylci` will be referred to as "install directory", while the directory you cloned the repository to will be referred to as "download directory". `./update.sh`, when run from download directory, will transfer the changes from the download directory (and GitHub) to the install directory.

Setup

`setup.sh` is the first script to be run after installation. It checks if you have python and python-pip installed and installs them if they aren't (using apt-get), then creates the install directory, copies all the files to it and installs a `systemd` unit file for system to run at boot. Perfect for Raspbian and Debian Jessie, TODO: add support for other systems.

Note: The system typically runs as root, and therefore is to be run as sudo/root user. Curious about the reasons? It's *explained in the FAQ*.

Installing dependencies for hardware

`config.sh` is the script that installs all the necessary packages and python libraries, depending on which hardware you're using. It will also set proper `config.json` contents if you're using a shield which has a pyLCI driver.

Configuring input and output devices

`config.json` is the file currently responsible for input and output hardware module configuration. It's JSON, so if you launch the system manually and see `JSONError` exceptions in the output, you know you have misspelled something.

Note: Generally, you won't need to edit `config.json` if you're using any shields recognised by `config.sh` because the configuration will be done automatically.

Its format is as follows:

```
{
  "input":
  [{
    "driver":"driver_filename",
    "args":[ "value1", "value2", "value3"...]
  }],
"output":
  [{
    "driver":"driver_filename",
    "kwargs":{ "key":"value", "key2":"value2"}
```

```
  }]
}
```

Documentation for *input* and *output* drivers have sample `config.json` for each driver. `"args"` and `"kwargs"` get passed directly to drivers' `__init__` method, so you can read the driver documentation/files to see if there are any options you could tweak.

Systemctl commands

- `systemctl start pylci.service`
- `systemctl stop pylci.service`

Note: This document refers to two pyLCI directories. First is "download directory", this is the directory which has been created by running `git clone` command. Second is "install directory", which is where pyLCI has been copied over by the `setup.sh` script.

Directory separation is good for being able to experiment with configuration options without breaking the current install, as well as for developing applications for the system while not cluttering your install version.

Launching the system manually

For testing configuration or development, you will want to launch the system directly so that you'll see system exception logs and will be able to stop it with a simple Ctrl^C. In that case, just run the system like `python main.py` from your download/install directory.

Tip: If system refuses to shut down (happens due to input subsystem threads not finishing sometimes), feel free to find its PID using `ps ax|grep "python main.py"` and do a `kill -KILL $PID` on it.

After you're done configuring/developing on the system, you can use `update.sh` to transfer your changes to the install directory.

Updating

`update.sh` is for updating your pyLCI install, pulling new commits from GitHub and copying all the new files from download directory to the install directory. This is useful to make your installed system up-to-date if there have been new commits or if you made some changes and want to transfer them to pyLCI install directory.

Note: `update.sh` automatically pulls all the GitHub commits - just comment the corresponding line out if you don't want it. It also runs `systemctl start pylci.service`.

Debugging issues

Debugging in general

Basic debugging steps:

- Launch system manually and see the error messages. Go to the directory you installed pyLCI from and launch `python main.py`. Alternatively, use `journalctl -u pylci.service` for a system that was running in daemon mode but crashed unexpectedly.
- Check your connections.

Hardware/driver issues:

- Check that the I2C/SPI/GPIO interfaces you're trying to use are available in `/dev/`. You might need to run `sudo raspi-config` and enable the interfaces you need (for Raspberry Pi boards) or do other system-specific changes (for other boards, see the manuals the manufacturer should provide).
- In case of I2C connection, check your I2C device connection with i2cdetect, you should see its address in i2cdetect's output when connected the right way.
- Check your connections in case you assembled things manually. In case of shields, there shouldn't be any problems.

Output issues

Basic debugging steps:

- Launch the output driver manually to display the test sequence. Go to the directory you installed pyLCI from and launch the output driver directly like `python output/drivers/your_driver.py`. You might need to adjust variables in `if __name__ == "__main__":` section.
- Is the driver you're using even the correct one? See the config.json and documentation for the driver you're using.

Currently, pyLCI uses HD44780-compatible screens as output devices. Minimum screen size is 16x2. There are some known issues when using those. Again, you're not likely to run into hardware problems when using shields.

Screen displaying garbage

- If using a breakout board, check if it's compatible with the driver. Some breakouts might use same ICs but have different pinouts, and any two pins interchanged can cause problems.
- Try and tie D0-D3 lines to GND. Those lines floating freely may cause instabilities, though it doesn't happen often.
- You can try to tie the R/W line to GND, too. It's even necessary in some cases, like Pi GPIO driver.
- Put a ~100pF capacitor between GND and EN. If screen starts quickly filling up with blocks after some time, pull the EN line down with a 10K resistor.

Screen characters being shifted incorrectly

- Try to set `"autoscroll":True` in `config.json` in output description (in `kwargs` section).

Only half of the screen is used

- Make sure you didn't set `"autoscroll":True` in `config.json` in output description (in `kwargs` section).

Nothing on the screen

- Is first row of blocks shown? If not, regulate the contrast with a potentiometer. You can also try to tie the contrast pin to GND.
- Does screen receive 5V (not 3.3V) as VCC? Unless it's a screen that's capable of doing 3.3V (must be stated in screen's description), that's a no-go.
- genindex
- modindex
- search

Input subsystem

These are the devices that receive key commands from some external source and route them to your applications. At the input system core, there's `InputListener`. It receives key events from drivers you use and routes them to currently active application.

Available input drivers:

- *HID input driver*
- *PCF8574 input driver*
- *PiFaceCAD input driver*
- *Adafruit CharLCD Plate&Chinese "LCD RGB KEYPAD" shield input driver*
- *Raspberry Pi GPIO input driver*

InputListener

The `i` variable you have supplied by `main.py load_app()` in your applications is an `InputListener` instance. It's operating on key names, such as "KEY_ENTER" or "KEY_UP". You can assign callback once a keypress with a matching keyname is received, which is as simple as `i.set_callback(key_name, callback)`. You can also set a dictionary of `"keyname":callback_function` mappings, this would be called a `keymap`.

class `input.input.`**`InputListener`**(*drivers*, *keymap=None*)
A class which listens for input device events and calls corresponding callbacks if set

`__init__`(*drivers*, *keymap=None*)
Init function for creating KeyListener object. Checks all the arguments and sets keymap if supplied.

`atexit`()
Exits driver (if necessary) if something wrong happened or pyLCI exits. Also, stops the listener

`check_special_callback`(*key_name*)
Raises exceptions upon setting of a special callback on a reserved/taken keyname

`clear_keymap`()
Removes all the callbacks set

event_loop(*index*)
Blocking event loop which just calls callbacks in the keymap once corresponding keys are received in the `self.queue`.

listen()
Start event_loop in a thread. Nonblocking.

receive_key(*key*)
This is the method that receives keypresses from drivers and puts them into `self.queue` for `self.event_loop` to receive

remove_callback(*key_name*)
Removes a single callback of the listener

remove_streaming()
Removes a callback for streaming key events, if previously set by any app/UI element.

replace_keymap_entries(*keymap*)
Sets all the callbacks supplied, not removing previously set but overwriting those with same keycodes

set_callback(*key_name*, *callback*)
Sets a single callback of the listener

set_keymap(*keymap*)
Sets all the callbacks supplied, removing the previously set keymap completely

set_maskable_callback(*key_name*, *callback*)
Sets a single maskable callback of the listener. Raises CallbackException if the callback is one of the reserved keys or already is in maskable/nonmaskable keymap.

A maskable callback is global (never cleared) and will be called upon a keypress unless a callback for the same keyname is already set in keymap.

set_nonmaskable_callback(*key_name*, *callback*)
Sets a single nonmaskable callback of the listener. Raises CallbackException if the callback is one of the reserved keys or already is in maskable/nonmaskable keymap.

A nonmaskable callback is global (never cleared) and will be called upon a keypress even if a callback for the same keyname is already set in `keymap` (callback from the `keymap` won't be called).

set_streaming(*callback*)
Sets a callback for streaming key events. The callback will be called with key_name as first argument but should support arbitrary number of positional arguments if compatibility with future versions is desired.

stop_listen()
This sets a flag for `event_loop` to stop. It also calls a `stop` method of the input driver `InputListener` is using.

Note: In v1.0 architecture, there's a single `InputListener` instance shared among all applications, so when you set some callbacks for your application and then exit it or execute your application's menu element, there's a very good chance your callbacks won't be there anymore once you return. You won't need to think about it unless you're setting `InputListener` yourself - mostly it's taken care of by UI objects, which set the keymaps themselves themselves (for example, `Menu` UI element sets the callbacks each time a menu is activated and each time a menu element callback execution is finished (because a `Menu` can't be sure whatever got called by the callback didn't set some of callbacks some other way, say, the element's callback was activating a nested menu.)

If you do set callbacks/keymap yourself (very useful for making your own UI elements, or for applications needing custom keybindings), it's important to remember that you need to stop `InputListener` before and start it again

afterwards, since the changes do not take place until that's done. For example, this is how you would set your own callback:

```
i.stop_listen()
i.clear_keymap() #Useful because there might be callbacks left from whatever your␣
↪function was called by
#... Set your callbacks
i.set_callback("KEY_ENTER", my_function)
i.listen()
```

Glue logic functions

Warning: Not for user interaction, are called by `main.py`, which is pyLCI launcher.

`input.input.`**`init`**`()`
This function is called by main.py to read the input configuration, pick the corresponding drivers and initialize InputListener.

It also sets `listener` globals of `input` module with driver and listener respectively, as well as registers `listener.stop()` function to be called when script exits since it's in a blocking non-daemon thread.

Drivers:

HID input driver

Sample config.json:

```
"input":
   [{
     "driver":"hid",
     "kwargs":
      {
       "name":"HID 04d9:1603"
      }
   }]
```

To get device names, you can just run `python input/driver/hid.py` while your device is connected. It will output available device names.

class `input.drivers.hid.`**`InputDevice`**(*path=None*, *name=None*, ***kwargs*)
A driver for HID devices. As for now, supports keyboards and numpads.

`__init__`(*path=None*, *name=None*, ***kwargs*)
Initialises the `InputDevice` object.

Kwargs:

- `path`: path to the input device. If not specified, you need to specify `name`.
- `name`: input device name

`runner``()`
Blocking event loop which just calls supplied callbacks in the keymap.

- genindex

- modindex
- search

PCF8574 input driver

It works with PCF8574 IO expanders. You can see an guide on modifying them and connecting them to buttons & I2C here.

```
"input":
   [{
     "driver":"pcf8574",
     "kwargs":
      {
       "addr":63,
       "int_pin":4
      }
   }]
```

class input.drivers.pcf8574.**InputDevice**(*addr=39*, *bus=1*, *int_pin=None*, ***kwargs*)

A driver for PCF8574-based I2C IO expanders. They have 8 IO pins available as well as an interrupt pin. This driver treats all 8 pins as button pins, which is often the case.

It supports both interrupt-driven mode (as fr now, RPi-only) and polling mode.

__init__(*addr=39*, *bus=1*, *int_pin=None*, ***kwargs*)

Initialises the InputDevice object.

Kwargs:

- bus: I2C bus number.
- addr: I2C address of the expander.
- int_pin: GPIO pin to which INT pin of the expander is connected. If supplied, interrupt-driven mode is used, otherwise, library reverts to polling mode.

loop_interrupts()

Interrupt-driven loop. Currently can only use RPi.GPIO library. Stops when stop_flag is set to True.

loop_polling()

Polling loop. Stops when stop_flag is set to True.

process_data(*data*)

Checks data received from IO expander and classifies changes as either "button up" or "button down" events. On "button up", calls send_key with the corresponding button name from self.mapping.

runner()

Starts listening on the input device. Initialises the IO expander and runs either interrupt-driven or polling loop.

- genindex
- modindex
- search

PiFaceCAD input driver

This driver works with PiFace Control and Display Raspberry Pi shields.

Sample config.json section:

```
"input":
   [{
     "driver":"pfcad"
   }]
```

Note: Generally, you won't need to edit `config.json` if you're using this shield because it'll be done automatically by `config.sh`.

class `input.drivers.pfcad.`**InputDevice**

A driver for PiFace Control and Display Raspberry Pi shields. It has 5 buttons, one single-axis joystick with a pushbutton, a 16x2 HD44780 screen and an IR receiver (not used yet).

__init__()

Initialises the `InputDevice` object and starts `pifacecad.SwitchEventListener`. Also, registers callbacks to `press_key` method.

activate()

Starts a thread with `start` function as target.

deactivate()

Starts a thread with `start` function as target.

loop_polling()

Polling loop. Stops when `stop_flag` is set to True.

process_data(*data*)

Checks data received from IO expander and classifies changes as either "button up" or "button down" events. On "button up", calls send_key with the corresponding button name from `self.mapping`.

send_key(*keycode*)

A hook to be overridden by `InputListener`. Otherwise, prints out key names as soon as they're pressed so is useful for debugging.

start()

Starts listening on the input device. Initialises the IO expander and runs either interrupt-driven or polling loop.

stop()

Sets the `stop_flag` for loop function.

- genindex
- modindex
- search

Adafruit CharLCD Plate&Chinese "LCD RGB KEYPAD" shield input driver

This driver works with Adafruit Raspberry Pi character LCD&button shields, as well as with Chinese clones following the schematic (can be bought for 5$ on eBay, typically have "LCD RGB KEYPAD ForRPi" written on them).

Sample `config.json` section:

```
"input":
   [{
     "driver":"adafruit_plate"
   }]
```

Note: Generally, you won't need to edit `config.json` if you're using this shield because it'll be done automatically by `config.sh`.

class `input.drivers.adafruit_plate.`**InputDevice**(*addr=32*, *bus=1*, ***kwargs*)

A driver for Adafruit-developed Raspberry Pi character LCD&button shields based on MCP23017, either Adafruit-made or Chinese-made.

Tested on hardware compatible with Adafruit schematic and working with Adafruit libraries, but not on genuine Adafruit hardware.

__init__(*addr=32*, *bus=1*, ***kwargs*)

Initialises the `InputDevice` object.

Kwargs:

- `bus`: I2C bus number.
- `addr`: I2C address of the expander.

init_expander()

Initialises the IO expander.

process_data(*data*)

Checks data received from IO expander and classifies changes as either "button up" or "button down" events. On "button up", calls send_key with the corresponding button name from `self.mapping`.

readMCPreg(*reg*)

Reads the MCP23017 register.

runner()

Polling loop (only one there can be on this shield, since interrupt pin is not connected).

setMCPreg(*reg*, *val*)

Sets the MCP23017 register.

- genindex
- modindex
- search

Raspberry Pi GPIO input driver

Driver for buttons connected to GPIO. Up to 8 button are supported now. Sample config.json:

```
"input":
   [{
     "driver":"pi_gpio",
     "kwargs":
      {
       "button_pins":[25, 24, 23, 18, 22, 27, 17, 4]
      }
   }]
```

class `input.drivers.pi_gpio.`**InputDevice**(*button_pins=[]*, ***kwargs*)

A driver for pushbuttons attached to Raspberry Pi GPIO. It uses RPi.GPIO library. Button's first pin has to be attached to ground, second pin has to be attached to the GPIO pin and pulled up to 3.3V with a 1-10K resistor.

__init__(*button_pins=[]*, ***kwargs*)
Initialises the `InputDevice` object.

Kwargs:

- `button_pins`: GPIO mubers which to treat as buttons (GPIO.BCM numbering)
- `debug`: enables printing button press and release events when set to True

runner()
Polling loop. Stops when `stop_flag` is set to True.

- genindex
- modindex
- search
- genindex
- modindex
- search

Output subsystem

Currently pyLCI uses HD44780-compatible screens as output devices. Minimum screen size is 16x2, 20x4 screens are tested and working. Available output drivers:

- *PCF8574 I2C LCD backpack driver*
- *PiFaceCAD output driver*
- *Adafruit CharLCD Plate&Chinese "LCD RGB KEYPAD" shield output driver*
- *Raspberry Pi GPIO output driver*
- *MCP23008 I2C LCD backpack driver*

Screen object

The `o` variable you have supplied by `main.py` `load_app()` in your applications is a `Screen` instance. It provides you with a set of functions available to HD44780 displays. Most of drivers just provide low-level functions for `HD44780` object, which, in turn, provides `Screen` object users with high-level functions described below:

class `output.drivers.hd44780.`**HD44780**(*cols=16*, *rows=2*, *do_init=True*, *debug=False*, *buffering=True*, ***kwargs*)
An object that provides high-level functions for interaction with display. It contains all the high-level logic and exposes an interface for system and applications to use.

__init__(*cols=16*, *rows=2*, *do_init=True*, *debug=False*, *buffering=True*, ***kwargs*)
Sets variables for high-level functions.

Kwargs:

- `rows` (default=2): rows of the connected display
- `cols` (default=16): columns of the connected display
- `debug` (default=False): debug mode which prints out the commands sent to display
- `**kwargs`: all the other arguments, get passed further to HD44780.init_display() function

autoscroll()
This will 'right justify' text from the cursor

blink()
Turn the blinking cursor on

clear()
Clears the display.

createChar(*char_num*, *char_contents*)
Stores a character in the LCD memory so that it can be used later. char_num has to be between 0 and 7 (including) char_contents is a list of 8 bytes (only 5 LSBs are used)

cursor()
Turns the underline cursor on

display()
Turn the display on (quickly)

display_data(**args*)
Displays data on display. This function checks if the display contents can be redrawn faster by buffering them and checking the output, then either changes characters one-by-one or redraws the screen completely.

`*args` is a list of strings, where each string corresponds to a row of the display, starting with 0.

home()
Returns cursor to home position. If the display is being scrolled, reverts scrolled data to initial position..

init_display(*autoscroll=False*, ***kwargs*)
Initializes HD44780 controller.

Kwargs:

- `autoscroll`: Controls whether autoscroll-on-char-print is enabled upon initialization.

leftToRight()
This is for text that flows Left to Right

noAutoscroll()
This will 'left justify' text from the cursor

noBlink()
Turn the blinking cursor off

noCursor()
Turns the underline cursor off

noDisplay()
Turn the display off (quickly)

println(*line*)
Prints a line on the screen (assumes position is set as intended)

rightToLeft()
This is for text that flows Right to Left

scrollDisplayLeft()
These commands scroll the display without changing the RAM

scrollDisplayRight()
These commands scroll the display without changing the RAM

setCursor(*row*, *col*)
Set current input cursor to `row` and `column` specified

Glue logic functions

Warning: Not for user interaction, are called by `main.py`, which is pyLCI launcher.

`output.output.`**`init`**`()`
This function is called by main.py to read the output configuration, pick the corresponding drivers and initialize a Screen object.

It also sets `screen` global of `output` module with created `Screen` object.

Drivers:

MCP23008 I2C LCD backpack driver

This driver was written for wide.hk I2C LCD backpacks (picture). They are very small and slim and don't have any means to configure their I2C address.

If you have another backpack and the driver doesn't work with this one, please open an issue on GitHub with a link to the backpack and its drivers for Arduino/Raspberry Pi

Sample config.json:

```
"input":
   [{
     "driver":"mcp23008",
     "kwargs":
      {
       "addr":"0x3f"
      }
   }]
```

Note: If you provide backpack's I2C address as a kwarg, you should pass it as a string (as shown above).

To test your screen, you can just run `python output/driver/mcp23008.py` while your screen is connected to I2C bus (you might want to adjust parameters in driver's `if __name__ == "__main__"` section). It will initialize the screen and show some text on it.

class `output.drivers.mcp23008.`**`Screen`**(*bus=1*, *addr=39*, *debug=False*, ***kwargs*)
A driver for MCP23008-based I2C LCD backpacks. The one tested had "WIDE.HK" written on it.

`__init__`(*bus=1*, *addr=39*, *debug=False*, ***kwargs*)
Initialises the `Screen` object.

Kwargs:

- `bus`: I2C bus number.
- `addr`: I2C address of the board.
- `debug`: enables printing out LCD commands.
- `**kwargs`: all the other arguments, get passed further to HD44780 constructor

`i2c_init``()`
Inits the MCP23017 IC for desired operation.

setMCPreg(*reg*, *val*)
Sets the MCP23017 register.

write4bits(*data*, *char_mode=False*)
Writes a nibble to the display. If `char_mode` is set, holds the RS line high.

write_byte(*byte*, *char_mode=False*)
Takes a byte and sends the high nibble, then the low nibble (as per HD44780 doc). Passes `char_mode` to `self.write4bits`.

- genindex
- modindex
- search

PCF8574 I2C LCD backpack driver

This driver works with PCF8574 IO expanders. You can see an guide on modifying them and connecting them to LCD screens & I2C here.

```
"output":
   [{
     "driver":"pcf8574",
     "kwargs":
      {
       "addr":"0x3f"
      }
   }]
```

Note: If you provide backpack's I2C address as a kwarg, you should pass it as a string (as shown above).

To test your screen, you can just run `python output/driver/pcf8574.py` while your screen is connected to I2C bus (you might want to adjust parameters in driver's `if __name__ == "__main__"` section). It will initialize the screen and show some text on it.

class `output.drivers.pcf8574.`**Screen**(*bus=1*, *addr=39*, *debug=False*, ***kwargs*)
A driver for PCF8574-based I2C LCD backpacks.

__init__(*bus=1*, *addr=39*, *debug=False*, ***kwargs*)
Initialises the `Screen` object.

Kwargs:

- `bus`: I2C bus number.
- `addr`: I2C address of the board.
- `debug`: enables printing out LCD commands.
- `**kwargs`: all the other arguments, get passed further to HD44780 constructor

expanderWrite(*data*)
Sends data to PCF8574.

write4bits(*value*, *char_mode=False*)
Writes a nibble to the display. If `char_mode` is set, holds the RS line high.

write_byte(*data*, *char_mode=False*)
Takes a byte and sends the high nibble, then the low nibble (as per HD44780 doc). Passes `char_mode` to `self.write4bits`.

- genindex
- modindex
- search

PiFaceCAD output driver

This driver works with PiFace Control and Display Raspberry Pi shields.

Sample config.json section:

```
"output":
   [{
     "driver":"pfcad"
   }]
```

Note: Generally, you won't need to edit `config.json` if you're using this shield because it'll be done automatically by `config.sh`.

class `output.drivers.pfcad.`**Screen**(*rows=2*, *cols=16*)
A driver for PiFace Control and Display Raspberry Pi shields. It has a simple 16x2 LCD on it, controlled by a MCP23S17 over SPI.

Doesn't yet conform to HD44780 library specs, many functions are not transferred from the `pifacecad` library.

TODO: rewrite it to remove dependency on PiFaceCAD library.

__init__(*rows=2*, *cols=16*)
Initialises the `Screen` object.

Kwargs:

- `rows` (default=2): rows of the connected display
- `cols` (default=16): columns of the connected display

clear()
Clears the display.

disable_backlight()
Disables backlight.

display_data(**args*)
Displays data on display.

`*args` is a list of strings, where each string fills each row of the display, starting with 0.

enable_backlight()
Enables backlight. Doesn't do it instantly, you'll have to wait until data is sent to the display.

- genindex
- modindex
- search

Adafruit CharLCD Plate&Chinese "LCD RGB KEYPAD" shield output driver

This driver works with Adafruit Raspberry Pi character LCD&button shields, as well as with Chinese clones following the schematic (can be bought for 5$ on eBay, typically have "LCD RGB KEYPAD ForRPi" written on them).

If you have a genuine Adafruit board, pass `"chinese":false` keyword argument to the driver in config.json so that the backlight works right.

Sample `config.json` section for Adafruit board:

```
"output":
   [{
     "driver":"adafruit_plate",
     "kwargs":
     {
      "chinese":false
     }
   }]
```

Sample `config.json` section for Chinese clone:

```
"output":
   [{
     "driver":"adafruit_plate"
   }]
```

Note: Generally, you won't need to edit `config.json` if you're using this shield because it'll be done automatically by `config.sh`.

---

class `output.drivers.adafruit_plate.`**Screen**(*bus=1*, *addr=32*, *debug=False*, *chinese=True*, ***kwargs*)

A driver for Adafruit-developed Raspberry Pi character LCD&button shields based on MCP23017, either Adafruit-made or Chinese-made. Has workarounds for Chinese plates with LED instead of RGB backlight and LCD backlight on a separate I2C GPIO expander pin.

Tested on hardware compatible with Adafruit schematic and working with Adafruit libraries, but not on genuine Adafruit hardware. Thus, you may have issues with backlight, as that's the 'gray area'.

__init__(*bus=1*, *addr=32*, *debug=False*, *chinese=True*, ***kwargs*)

Initialises the `Screen` object.

Kwargs:

- `bus`: I2C bus number.
- `addr`: I2C address of the board.
- `debug`: enalbes printing out LCD commands.
- `chinese`: flag enabling workarounds necessary for Chinese boards to enable LCD backlight.

i2c_init()

Inits the MCP23017 expander.

setMCPreg(*reg*, *val*)

Sets the MCP23017 register.

write4bits(*data*, *char_mode=False*)

Writes a nibble to the display. If `char_mode` is set, holds the RS line high.

- genindex
- modindex
- search

Raspberry Pi GPIO output driver

This driver works with HD44780-screens connected to Raspberry Pi GPIO. The screen connected has to have its RW pin tied to ground.

Sample config.json:

```
"output":
   [{
     "driver":"pi_gpio",
     "kwargs":
      {
       "pins":[25, 24, 23, 18],
       "en_pin":4,
       "en_pin":17
      }
   }]
```

class `output.drivers.pi_gpio.`**`Screen`**(*pins=[]*, *rs_pin=None*, *en_pin=None*, *debug=False*, ***kwargs*)

Driver for using HD44780 displays connected to Raspberry Pi GPIO. Presumes the R/W line is tied to ground. Also, doesn't yet control backlight.

`__init__`(*pins=[]*, *rs_pin=None*, *en_pin=None*, *debug=False*, ***kwargs*)

Initializes the GPIO-driven HD44780 display

All GPIOs passed as arguments will be used with BCM mapping.

Kwargs:

- `pins`: list of GPIO pins for driving display data bits in format [DB4, DB5, DB6, DB7]
- `en_pin`: EN pin GPIO number. Please, make sure it's pulled down to GND (10K is OK). Otherwise, block might start filling up the screen unexpectedly.
- `rs_pin`: RS pin GPIO number,
- `debug`: enables printing out LCD commands.
- `**kwargs`: all the other arguments, get passed further to HD44780 constructor

`write4bits`(*bits*, *char_mode=False*)

Writes a nibble to the display. If `char_mode` is set, holds the RS line high.

`write_byte`(*byte*, *char_mode=False*)

Takes a byte and sends the high nibble, then the low nibble (as per HD44780 doc). Passes `char_mode` to `self.write4bits`.

- genindex
- modindex
- search
- genindex
- modindex

- search

UI element reference

UI elements are used in applications and some core system functions to interace with the user. For example, the Menu element is used for making menus, and can as well be used to show lists of items.

Using UI elements in your applications is as easy as doing:

```
from ui import ElementName
```

and initialising them, passing your UI element contents and parameters, as well as input and output device objects as initialisation arguments.

UI elements:

Menu UI element

```
from ui import Menu
...
menu_contents = [
["Do this", do_this],
["Do this with 20", lambda: do_this(x=20)],
["Do nothing"],
["My submenu", submenu.activate]
]
Menu(menu_contents, i, o, "My menu").activate()
```

class `ui.menu.`**`Menu`**(*contents*, *i*, *o*, *name='Menu'*, *entry_height=1*, *append_exit=True*, *catch_exit=True*, *exitable=True*, *contents_hook=None*, *scrolling=True*)

Implements a menu which can be used to navigate through your application, output a list of values or select actions to perform. Is one of the most used elements, used both in system core and in most of the applications.

Attributes:

- `contents`: list of menu elements which was passed either to `Menu` constructor or to `menu.set_contents()`.

  Menu element structure is a list, where:

  - `element[0]` (element's representation) is either a string, which simply has the element's value as it'll be displayed, such as "Menu element 1", or, in case of entry_height > 1, can be a list of strings, each of which represents a corresponding display row occupied by the element.
  - `element[1]` (element's callback) is a function which is called when menu's element is activated (such as pressing ENTER button when menu's element is selected). * Can be omitted if you don't need to have any actions taken upon activation of the element. * Can be specified as 'exit' if you want a menu element that exits the menu upon activation.

If you want to set contents after the initalisation, please, use set_contents() method.

- `_contents`: "Working copy" of menu contents, basically, a `contents` attribute which has been processed by `self.process_contents`.
- `pointer`: currently selected menu element's number in `self._contents`.
- `in_background`: a flag which indicates if menu is currently active, either if being displayed or being in background (for example, if a sub-menu of this menu is currently active)
- `in_foreground` : a flag which indicates if menu is currently displayed. If it's not active, inhibits any of menu's actions which can interfere with other menu or UI element being displayed.
- `first_displayed_entry` : Internal pointer which points to the number of `self._contents` element which is at the topmost position of the menu as it's currently displayed on the screen
- `last_displayed_entry` : Internal pointer which points to the number of `self._contents` element which is at the lowest position of the menu as it's currently displayed on the screen
- `no_entry_message` : The entry displayed in case menu has no elements

__init__(*contents*, *i*, *o*, *name='Menu'*, *entry_height=1*, *append_exit=True*, *catch_exit=True*, *exitable=True*, *contents_hook=None*, *scrolling=True*)

Initialises the Menu object.

Args:

- `contents`: a list of values, which can be constructed as described in the Menu object's docstring.
- `i`, `o`: input&output device objects

Kwargs:

- `name`: Menu name which can be used internally and for debugging.
- `entry_height`: number of display rows one menu element occupies.
- `append_exit`: Appends an "Exit" alement to menu elements. Doesn't do it if any of elements has callback set as 'exit'.
- `catch_exit`: If `MenuExitException` is received and catch_exit is False, it passes `MenuExitException` to the parent menu so that it exits, too. If catch_exit is True, MenuExitException is not passed along.
- `exitable`: Decides if menu can exit at all by pressing `KEY_LEFT`. Set by default and disables `KEY_LEFT` callback if unset. Is used for pyLCI main menu, not advised to be used in other settings.

activate()

A method which is called when menu needs to start operating. Is blocking, sets up input&output devices, renders the menu and waits until self.in_background is False, while menu callbacks are executed from the input device thread. This method also raises MenuExitException if menu exited due to it and `catch_exit` is set to False.

deactivate()

Deactivates the menu completely, exiting it. As for now, pointer state is preserved through menu activations/deactivations

generate_keymap()

Sets the keymap. In future, will allow per-system keycode-to-callback tweaking using a config file.

print_contents()

A debug method. Useful for hooking up to an input event so that you can see the representation of menu's contents.

print_name()
A debug method. Useful for hooking up to an input event so that you can see which menu is currently processing input events.

set_contents(*contents*)
Sets the menu contents, as well as additionally re-sets `last` & `first_displayed_entry` pointers and calculates the value for `last_displayed_entry` pointer.

class ui.menu.**MenuExitException**
An exception that you can throw from a menu callback to exit the menu that callback was called from

- genindex
- modindex
- search

Printer UI element

```
from ui import Printer
Printer(["Line 1", "Line 2"], i, o, 3, exitable=True)
Printer("Long lines will be autosplit", i, o, 1)
```

ui.printer.**Printer**(*message*, *i*, *o*, *sleep_time=1*, *skippable=False*)
Outputs string data on display as soon as it's called.

Args:

- `message`: A string or list of strings to display. A string will be split into a list, a list will not be modified. The resulting list is then displayed string-by-string.
- `i`, `o`: input&output device objects. If you're not using skippable=True and don't need exit on KEY_LEFT, feel free to pass None as i.

Kwargs:

- `sleep_time`: Time to display each the message (for each of resulting screens).
- `skippable`: If set, allows skipping message screens by presing ENTER.

Refresher UI element

```
from ui import Refresher
counter = 0
def get_data():
    counter += 1
    return [str(counter), str(1000-counter)] #Return value will be sent directly to␣
↪output.display_data
Refresher(get_data, i, o, 1, name="Counter view").activate()
```

class ui.refresher.**Refresher**(*refresh_function*, *i*, *o*, *refresh_interval=1*, *keymap=None*, *name='Refresher'*)
Implements a state where display is refreshed from time to time, updating the screen with information from a function.

__init__(*refresh_function*, *i*, *o*, *refresh_interval=1*, *keymap=None*, *name='Refresher'*)
Initialises the Refresher object.

Args:

- `refresh_function`: a function which returns data to be displayed on the screen upon being called, in the format accepted by `screen.display_data()`
- `i`, `o`: input&output device objects

Kwargs:

- `refresh_interval`: Time between display refreshes (and, accordingly, `refresh_function` calls)
- `keymap`: Keymap entries you want to set while Refresher is active
- `name`: Refresher name which can be used internally and for debugging.

`activate``()`
A method which is called when refresher needs to start operating. Is blocking, sets up input&output devices, renders the refresher, periodically calls the refresh function&refreshes the screen while self.in_foreground is True, while refresher callbacks are executed from the input device thread.

`deactivate``()`
Deactivates the refresher completely, exiting it.

`print_name``()`
A debug method. Useful for hooking up to an input event so that you can see which refresher is currently active.

Checkbox UI element

```
from ui import Checkbox
contents = [
["Apples", 'apples'],
["Oranges", 'oranges'],
["Bananas", 'bananas']]
selected_fruits = Checkbox(checkbox_contents, i, o).activate()
```

class `ui.checkbox.`**`Checkbox`**(*contents*, *i*, *o*, *name='Menu'*, *entry_height=1*, *default_state=False*, *append_exit=True*)
Implements a checkbox which can be used to enable or disable some functions in your application.

Attributes:

- `contents`: list of checkbox elements which was passed either to `Checkbox` constructor or to `checkbox.set_contents()`.

  Checkbox element structure is a list, where:

  - `element[0]` (element's representation) is either a string, which simply has the element's value as it'll be displayed, such as "Menu element 1", or, in case of entry_height > 1, can be a list of strings, each of which represents a corresponding display row occupied by the element.
  - `element[1]` (element's name) is a name returned by the checkbox upon its exit in a dictionary along with its boolean value.
  - `element[2]` (element's state) is the default state assumed by the checkbox. If not present, assumed to be default_state.

  If you want to set contents after the initalisation, please, use set_contents() method.

- `_contents`: "Working copy" of checkbox contents, basically, a `contents` attribute which has been processed by `self.process_contents`.
- `pointer`: currently selected menu element's number in `self._contents`.

- `in_foreground` : a flag which indicates if checkbox is currently displayed. If it's not active, inhibits any of menu's actions which can interfere with other menu or UI element being displayed.
- `first_displayed_entry` : Internal flag which points to the number of `self._contents` element which is at the topmost position of the checkbox menu as it's currently displayed on the screen
- `last_displayed_entry` : Internal flag which points to the number of `self._contents` element which is at the lowest position of the checkbox menu as it's currently displayed on the screen

__init__(*contents*, *i*, *o*, *name='Menu'*, *entry_height=1*, *default_state=False*, *append_exit=True*)
Initialises the Checkbox object.

Args:

- `contents`: a list of element descriptions, which can be constructed as described in the Checkbox object's docstring.
- `i`, `o`: input&output device objects

Kwargs:

- `name`: Checkbox name which can be used internally and for debugging.
- `entry_height`: number of display rows one checkbox element occupies.
- `default_state`: default state of the element if not supplied.

activate()
A method which is called when checkbox needs to start operating. Is blocking, sets up input&output devices, renders the checkbox and waits until self.in_background is False, while checkbox callbacks are executed from the input device thread.

print_contents()
A debug method. Useful for hooking up to an input event so that you can see the representation of checkbox's contents.

print_name()
A debug method. Useful for hooking up to an input event so that you can see which UI element is currently processing input events.

set_contents(*contents*)
Sets the checkbox contents, as well as additionally re-sets `last` & `first_displayed_entry` pointers and calculates the value for `last_displayed_entry` pointer.

Numeric input UI elements

```
from ui import IntegerAdjustInput
start_from = 0
number = IntegerAdjustInput(start_from, i, o).activate()
if number is None: #Input cancelled
    return
#process the number
```

class `ui.number_input.`**IntegerAdjustInput**(*number*, *i*, *o*, *message='Pick a number:'*, *interval=1*, *name='IntegerAdjustInput'*)
Implements a simple number input dialog which allows you to increment/decrement a number using which can be used to navigate through your application, output a list of values or select actions to perform. Is one of the most used elements, used both in system core and in most of the applications.

Attributes:

- `number`: The number being changed.
- `initial_number`: The number sent to the constructor. Used by reset() method.
- `selected_number`: A flag variable to be returned by activate().
- `in_foreground` : a flag which indicates if UI element is currently displayed. If it's not active, inhibits any of element's actions which can interfere with other UI element being displayed.

__init__ (*number*, *i*, *o*, *message='Pick a number:'*, *interval=1*, *name='IntegerAdjustInput'*)
Initialises the IntegerAdjustInput object.

Args:

- `number`: number to be operated on
- `i`, `o`: input&output device objects

Kwargs:

- `message`: Message to be shown on the first line of the screen when UI element is active.
- `interval`: Value by which the number is incremented and decremented.
- `name`: UI element name which can be used internally and for debugging.

activate()
A method which is called when input element needs to start operating. Is blocking, sets up input&output devices, renders the UI element and waits until self.in_background is False, while callbacks are executed from the input device thread. This method returns the selected number if KEY_ENTER was pressed, thus accepting the selection. This method returns None when the UI element was exited by KEY_LEFT and thus it's assumed changes to the number were not accepted.

deactivate()
Deactivates the UI element, exiting it and thus making activate() return.

print_name()
A debug method. Useful for hooking up to an input event so that you can see which UI element is currently processing input events.

print_number()
A debug method. Useful for hooking up to an input event so that you can see current number value.

Character input UI elements

```
from ui import CharArrowKeysInput
password = CharArrowKeysInput(i, o, message="Password:", name="My password dialog").
↪activate()
if password is None: #UI element exited
    return False #Cancelling
#processing the input you received...
```

class ui.char_input.**CharArrowKeysInput**(*i*, *o*, *initial_value=''*, *message='Value:'*, *allowed_chars=[' '][S'*, *' '][c'*, *' '][C'*, *' '][s'*, *' '][n']*, *name='CharArrowKeysInput'*)
Implements a character input dialog which allows to input a character string using arrow keys to scroll through characters

Attributes:

- `value`: A list of characters of the currently displayed value

- `position`: Position of the currently edited character.
- `cancel_flag`: A flag that is set when editing is cancelled.
- `in_foreground`: A flag which indicates if UI element is currently displayed. If it's not active, inhibits any of element's actions which can interfere with other UI element being displayed.
- `charmap`: Internal string that contains all of the possible character values
- `char_indices`: A list containing char's index in `charmap` for every char in `value` list
- `first_displayed_char`: An integer pointer to the first character currently displayed (for the occasions where part of value is off-screen)
- `last_displayed_char`: An integer pointer to the last character currently displayed

__init__(*i, o, initial_value='', message='Value:', allowed_chars=[']['S', '][c', '][C', '][s', '][n'], name='CharArrowKeysInput'*)

Initialises the CharArrowKeysInput object.

Args:

- `i`, `o`: input&output device objects

Kwargs:

- `initial_value`: Value to be edited. If not set, will start with an empty string.
- `allowed_chars`: Characters to be used during input. Is a list of strings designating ranges which can be the following: * '][c' for lowercase ASCII characters * '][C' for uppercase ASCII characters * '][s' for special characters from those supported by HD44780 character maps * '][S' for space * '][n' for numbers * '][h' for hexadecimal characters (0-F) If a string does not designate a range of characters, it'll be added to character map as-is.
- `message`: Message to be shown in the first row of the display
- `name`: UI element name which can be used internally and for debugging.

activate()

A method which is called when input element needs to start operating. Is blocking, sets up input&output devices, renders the element and waits until self.in_background is False, while menu callbacks are executed from the input device thread. This method returns the selected value if KEY_ENTER was pressed, thus accepting the selection. This method returns None when the UI element was exited by KEY_LEFT and thus the value was not accepted.

deactivate()

Deactivates the UI element, exiting it and thus making activate() return.

print_name()

A debug method. Useful for hooking up to an input event so that you can see which UI element is currently processing input events.

print_value()

A debug method. Useful for hooking up to an input event so that you can see current value.

- genindex
- modindex
- search

Applications

Applications are layer between user's goals and same goals accomplished. pyLCI applications are similar to desktop applications, each of them is mean to perform one function/set of similar functions, and perform it well. There can be an application for any task you want to use pyLCI for - in the worst case, you can write one ;-)

Applications bundled with the default install are:

- *Clock app*
- *I2C toolkit application*
- *USB device info app*
- *Music player control app*
- *Network interface info app*
- *Partition unmount&info app*
- *Script execution app*
- *Shutdown&reboot app*
- *System information app*
- *Service control app*
- *Raspberry Pi video settings app*
- *pyLCI update app*
- *Volume control application*
- *Wireless connections app*

Some information on maintaining and writing applications:

- *Developing and managing applications*
- *Skeleton application*

Clock app

This application gives you a simple clock that refreshes once a second. Time shown is the system time.

I2C toolkit application

As for now, this is a fairly simple application which just scans the I2C bus and lists all the devices that have responded. Plans are to include I2C read and I2C write functionality in it.

USB device info app

This application gives you information about connected USB devices.

It lists them in format `["{bus}{dev},{vid_pid}", "name"]`, you can click on an entry to see full name of the device.

Music player control app

This is proof-of-concept application for controlling a music player - in this case, MOCP. It can switch track to next/previous, as well as toggle play/pause.

Network interface info app

This application shows you network connection information. Under the hood, it uses "ip" command.

It shows you:

- Interface state (up/down)
- IP and IPv6 addresses
- MAC addresses

Partition unmount&info app

This application lets you see the mounted partitions on your system, as well as unmount and eject them.

It's capable of:

- Listing mounted drives
- Unmounting them
- Ejecting them
- Unmounting them lazily

Note: Lazy unmounting means the filesystem is unmounted as soon as it stops being busy

Script execution app

This application lets you run various pre-defined scripts and commands.

Note: It isn't yet capable of stopping application's execution or displaying application's output.

Defining applications is done in `config.json` file which is located in the application's directory (currently `apps/scripts`). Its format is as follows:

```
[
 {"path":"./s/login.sh", #Defining a script which's located relative to application␣
↪directory (``apps/scripts``)
  "name":"Hotspot login"}, #Defining a pretty name which'll be displayed by pyLCI in␣
↪the application menu
 {"path":"/root/backup.sh", #Defining a script by absolute path
  "name":"Backup things",
  "args":["--everything", "--now"]}, #Giving command-line arguments to a script
 {"path":"mount", #Calling an external command available from $PATH
  "name":"'mount' with -a",
  "args":["-a"]} #Again, command-line arguments
]
```

Note: #-starting comments aren't accepted in JSON and are provided solely for explanation purposes

It also gets all the scripts in `s/` folder in application's directory and adds them to the script menu, if they're not available in `config.json`. If "name" parameter is not provided or is not available, it falls back to using script's filename.

Shutdown&reboot app

This application lets you shutdown and reboot your system cleanly.

Skeleton app

This is an example application. It shows basics of initializing your application, some conventions you need to follow and basics of working with UI elements.

```
menu_name = "Skeleton app" #App name as seen in main menu while using the␣
↪system

from subprocess import call
from time import sleep

from ui import Menu, Printer

def call_internal():
    Printer(["Calling internal", "command"], i, o, 1)
    print("Success")

def call_external():
    Printer(["Calling external", "command"], i, o, 1)
    call(['echo', 'Success'])

#Callback global for pyLCI. It gets called when application is activated in␣
↪the main menu
callback = None

#Some globals for us
i = None #Input device
o = None #Output device

def init_app(input, output):
    global callback, i, o
    i = input; o = output #Getting references to output and input device␣
↪objects and saving them as globals
    main_menu_contents = [
    ["Internal command", call_internal],
    ["External command", call_external],
    ["Exit", 'exit']]
    main_menu = Menu(main_menu_contents, i, o, "Skeleton app menu")
    callback = main_menu.activate
```

System information app

This application gives you information about various system parameters.

It can list:

- Total, used and free memory amounts - same figures `free` command gives you
- Uptime and load average ratings
- System information - hostname, kernel version, acrhitecture and distribution information

Service control app

This application lists all systemd units available and lets you manage them.

It's capable of:

- Starting/stopping/restarting/reloading units
- Enabling and disabling units
- Filtering units by their type (service/target/mount/etc.)

Raspberry Pi video settings app

This application lets you change the HDMI/TV display parameters on your Raspberry Pi. Useful when you, for example, want to hot-plug it to a monitor and make RPi recognise it.

It uses a 'tvservice.py' wrapper library to provide a layer between command-line calls and UI (library is included in the application and resides in the application folder).

It's capable of:

- Turning HDMI display on (with preferred settings, see `tvservice -p`) and off, as well as calling appropriate `fbset` triggers afterwards.
- Choosing resolution from those the display supports
- Viewing TVService status
- Parsing and showing TVService flags

TVService is installed by default on Raspbian.

pyLCI update app

This application updates your pyLCI install by pulling the latest commits straight from pyLCI GitHub.

Note: Do remember this updates only the pyLCI install currently running, effectively, doing a `git pull` in the current directory. So, if it's launched (it is unless you're launching it manually at the moment) from the install directory (most likely), it'll "git pull" inside the download directory (/opt/pylci by default), and vice-versa.

Volume control application

This is a simple application for controlling volume. As for now, it supports turning volume up/down or muting it for a single mixer channel. Under the hood, it uses 'amixer'.

Wireless connections app

This application lets you connect to wireless networks and manage connections. Under the hood, it uses `wpa_cli` to connect to a running `wpa_supplicant` instance.

Note: Seriously, wpa_supplicant as wireless management daemon is awesome. Minimalistic and really easy to interface. Also, it's included and running in latest Raspbian versions (from 02.16).

It uses a 'wpa_cli.py' wrapper library to provide a layer between command-line calls and UI (library is included in the application and resides in the application folder).

It's capable of:

- Scanning wireless networks and displaying scan results
- Connecting to known and open wireless networks
- Viewing wireless connection status
- Managing multiple wireless interfaces
- Saving configuration changes to wpa_supplicant.conf file

If you're not running wpa_supplicant as a daemon and you want to do it, you should follow *this guide <https://learn.sparkfun.com/tutorials/using-pcduinos-wifi-dongle-with-the-pi/edit-interfaces>* for adjusting your /etc/network/interfaces and *this guide<https://learn.sparkfun.com/tutorials/using-pcduinos-wifi-dongle-with-the-pi/edit-wpasupplicantconf>* for creating contents of your /etc/wpa_supplicant/wpa_supplicant.conf.

- genindex
- modindex
- search

Managing and developing applications

General information

- Applications are simply folders which are made importable by Python by adding an `__init__.py` file. pyLCI loads `main.py` file residing in that folder. It needs an `init_app()` function inside the main.py file. It also expects a variable called `callback` which is called when the application is activated by launching it from the menu, and a variable named `menu_name` which contains a name that'll be shown in the main menu.
- You can combine UI elements in many different ways, including making nested menus, which makes apps less cluttered.
- pyLCI main menu can have submenus. Submenu is just a folder which has `__init__.py` file in it, but doesn't have a `main.py` file. It can store both application folders and child submenu folders.
  - To set a main menu name for your submenu, you need to add `_menu_name = "Pretty name"` in `__init__.py` file of a submenu.
  - Submenus can be nested - just create another folder inside a submenu folder. However, submenu inside an application folder won't be detected.
- All application modules are loading when pyLCI loads. When choosing an application in the main menu/submenu, its global `callback` is called. It's usually set as the `activate()` method of application's main UI element, such as a menu.

- You can prevent any application from autoloading (but still have an option to load it manually) by placing a `do_not_load` file (with any contents) in application's folder (for example, see skeleton application folder).

Development tips

- For starters, take a look at the *skeleton app*
- You can launch pyLCI in a "single application mode" using `main.py -a apps/app_folder_path`. There'll be no main menu constructed, and exiting the application exits pyLCI.
- You should not set input callbacks or output to screen while your application is not the one active. It'll cause screen contents set from another application to be overwritten, which is bad user experience. Make sure your application is the one currently active before outputting things and setting callbacks.

Future plans

A TODO document, if you will

Note: This list is not by any means complete. What's listed here is bound to appear sooner or later. What's not listed is either not yet considered or not going to be implemented - feel free to ask me at GitHub!

Global system changes

- Make hotplug of input/output devices possible
- Include a notification system

Hardware

- Make a simple Arduino setup with screen&buttons and firmware+drivers for it to act as pyLCI I/O device over serial
- Make a wireless (ESP8266-based) setup
- Make a fully working PiFaceCAD driver

Input devices

- Make a "passthrough" driver for HID so that a single keyboard can both be used for X and pyLCI
- Make an input emulator for development tasks
- Add key remapping to HID driver
- Pressed/released/held button states

Output devices

- Make an output emulator for development tasks
- Add backlight control layer to all displays

Supporting graphical displays

- Get/make fonts
- Include compatibility layers

Applications

- Bluetooth app (delayed, involves a lot of DBus work)
- MPD/Mopidy app
- Camera app
- Stopwatch/timer app
- UCI management app
- Counter app
- Calculator app
- Mount partitions app
- OpenHAB console
- Twitter reader
- NMap app
- SMS and call app - interfacing to mobile phones and GSM/3G modems

UI elements

Input UI elements

- Date/time picker
- "Quick reading" UI element (word-by-word)
- Simple number input
- Character input using keypads and keyboards
- Wraparound for Menu UI element
- Page up/down for Menu UI element

Development

- More example apps & examples for UI elements
- Guide about input callbacks and 5 main keys
- An app development course
- Make a release system
- More links to UI element usage examples in existing apps

Integration into projects

- Examples for RPC API wrapper (for integration in any projects running in separate threads)

Maintenance

- Refactor UI elements to use common classes
- Refactor main.py launcher
- Clean up comments in UI elements, decide what functions to expose in the docs
- Make an app for configuring pyLCI on the fly

Publicity

- Some videos
- A website
- Collection of projects developed with pyLCI

FAQ&contacts

Here are some answers to questions that arise. Don't forget to look through the "Future plans"! Got a question that isn't answered here? Try to look through GitHub issues. If not found, create a new one! If you have another questions, e-mail me .

FAQ

Does pyLCI support screen connected via 595/this particular Pi shield/some other input/output device I have?

Short answer - it may not, but it's likely easy to add support for it.

First of all, look through the drivers supported. If you don't understand something, feel free to ask (GitHub issue/e-mail)! I'll be happy to help you, as well as update the docs.

Second thing is - drivers for input/output devices are hella easy to implement. HD44780 screens use a common library, so that only the "sending actual commands/characters" to the character screen has to be implemented, and input devices just have to send "KEY_something" strings to `InputListener` when there's a keypress, optionally, do their best to shutdown cleanly (bane of the HID driver for now). You have a shield with a Python library available? Chances are, it's easy to write a driver for it by hooking it up to pyLCI driver structure - look at `pfcad` driver, that's exactly the approach used. Or look at the `output/driver/pi_gpio` driver, it's a nice example of leveraging the HD44780 abstraction. In short - you can do it yourself, and if you can't, then open an issue on GitHub and you can help develop and test the driver to whatever input device you have so that you can enjoy all the benefits of pyLCI.

Note: Well, I have to admit things are still better not using some additional libraries, but they work, and that's the main thing. You need a driver, quick? Great, just take a look at current version of `output/drivers/pfcad`, it's an example of both how to connect an external library and on various workarounds you might need to use.

Does pyLCI support graphical/color OLEDs/TFTs, or other non-character non-HD44780 displays?

Short answer - it yet doesn't, but I'm developing everything so that it will.

There's a significant amount of work to be put into it. You need to make fonts for applications/UI elements relying on character output, facilitate display re-draws so that it's not painfully slow because it's redrawing the whole display every time, provide abstraction layers for fallback & other screen types, oh, and document it well enough so that it's usable. And yet, this is something to be included.

The reasons it's not included now is to be able to focus on applications that need to be developed, and because HD44780 screens are the most popular ones - excluding, maybe, the HDMI-, VGA- and RCA-connected ones, but they're partly the reason pyLCI is developed =) If you lack on-screen place, 20x4 screens are popular and cheap.

Does pyLCI support multiple output devices, such as 2 or more screens?

Short answer - it yet doesn't, but I'll be happy to work on it once there's a user for it and there's a use case.

It's not hard to include this, but there are multiple ways to do it, and each one seems right. For now, many users say they'd just pass different screens to different applications, or use a separate screen for monitoring. This is possible, but would require close collaboration with end users of such a setup so that it's spot on for their applications and adjustable for others - in other words, not a dirty hack for the sake of adding a feature. So - contact me, we can work on it if you need it!

Also, I'd like to remind about `LCDproc` project, which is all about displaying relatively static information, such as music player/CPU load info and similar things. It's a well-developed project and pyLCI is not yet claiming its place because they have different use cases, each has their own strengths and weaknesses. It's not hard to imagine using one screen for pyLCI and another - for LCDproc. That said, it's also not hard to use full pyLCI configuration on one screen and pyLCI in single-app mode in another ;-)

Does the system need to run as root?

It does not *need* to, but it doesn't make much sense otherwise. pyLCI drivers&apps need all kinds of different privileges for various tasks, and it's run as a single application, so it either needs to be run as root or to be run as a user with enough privileges to do management tasks, which is not that far away from root in terms of danger.

However, from some point there will be a split between pyLCI core and applications, where only core will need to run as a user privileged enough to access input/output devices, and applications will be able to run under separate users. .. _openwrt_possible:

Is it possible to run pyLCI under OpenWRT?

Yes, as OpenWRT is a Linux distribution. It doesn't even really need `pip` if you take care of all dependencies. However, it's not tested. Also, you're likely to need extroot because Python takes a lot of space.

Note: UCI interface for now is lacking, but shouldn't be difficult to implement.

Why does it grab all the HID events from a device given to the HID driver?

Unfortunately, now there's no 'passthrough' driver that'd take only part of all the keypresses and pass all the other further. This driver is to appear soon.

Which hardware can you use for running pyLCI on desktop computer/server/HTPC?

- First of all, there are plans for making a firmware&driver for Arduino devices with commonly encountered button&16x2 LCD shields. The result will be connectable over USB as a USB-Serial device.
- Second thing is that most video cards have I2C lines on video ports accessible from Linux, and there's no problem with connecting I2C GPIO expanders to it, except that there's no GPIO to take advantage of button interrupt function.
- Third thing is that you can easily use HID keyboards and numpads as input devices.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

o

u

Index

Symbols

A

B

C

D

E

G

H

U

W