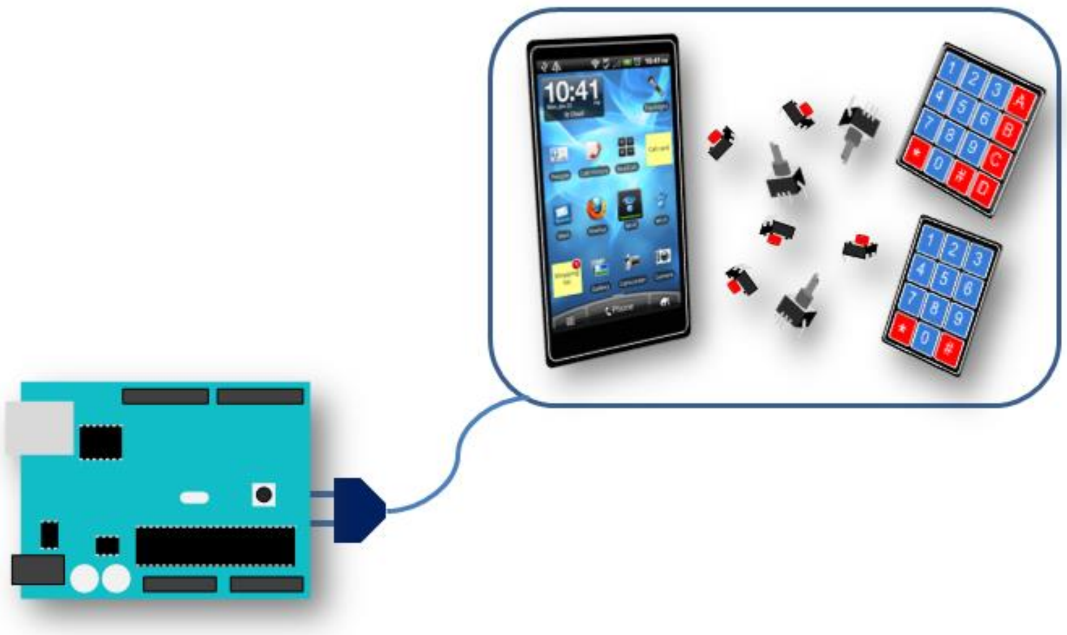
Phi_interfaces Arduino input hardware library

- Connecting inputs to Arduino



Last reviewed on 11/4/2015
Dr. John Liu
St. Cloud, MN USA

1. Introduction

Phi_interfaces is the very first and only open-source Arduino library that handles all input devices for Arduino such as push buttons, matrix keypads, rotary encoders, analog buttons, even Bluetooth-enabled smartphones, all under a common framework. See figure 1-1 for a concept. At the moment it contains classes for buttons, matrix keypads, analog buttons, rotary encoders, phi-panel serial LCD keypads (or smartphone input or simulated keypads over serial) and liudr keypad, plus very easy ways to expand the interface to include things like capacitive keypads, IR remote controls, PS/2 keyboards, touch screen keyboards, Ethernet shield, etc. With this library, you no longer have to find and learn many libraries, one for each type of inputs. You simply include this one library and start sensing all your input devices.

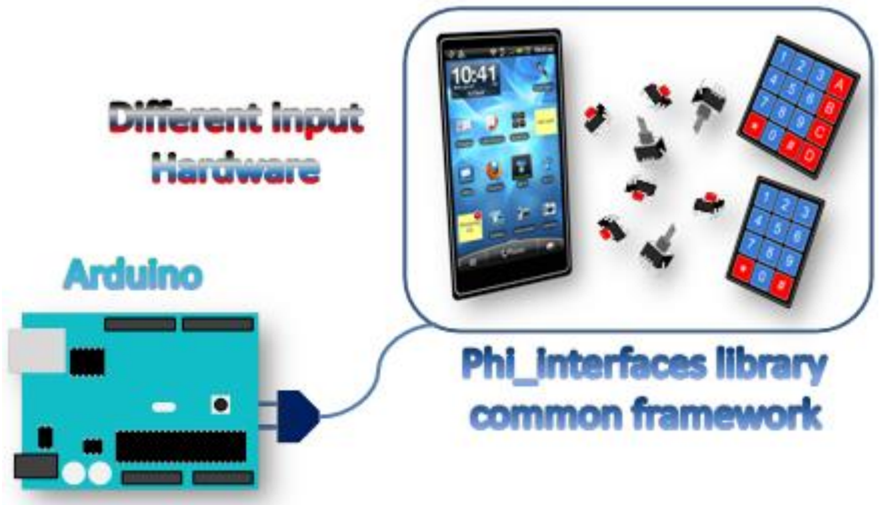


Fig. 1-1. All different input hardware are united under the phi_interfaces library's common framework and to Arduino programming they are handled in the same way.

Each object class in the library corresponds to one type of input hardware, such as ***phi_matrix_keypad*** is the object corresponding to matrix keypads. All classes share the this aspect: you call getKey() function in a loop to see what key is pressed, regardless what hardware is generating the key press.

Since all different hardware are sensed by getKey(), you can easily modify what type of input you use without changing your code. See figure 1-2.

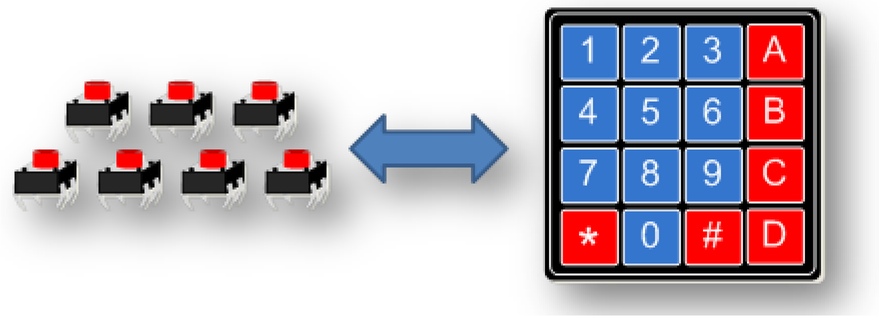


Fig. 1-2 You can start with several buttons and end up using a matrix keypad. You don't need to change your Arduino code even you changed your hardware.

So you can start with a few buttons, then decide to use a matrix keypad, and later add a couple rotary encoders, then eventually decided to spread a few buttons over a wooden box top (just for the look). Maybe later you decide to use an android phone as the control panel for your Arduino project, yeah, you can make the change without rewriting any of your code! Your program code won't need any change. That is to say the library makes what hardware you use transparent (invisible) to your code so any hardware can be used as input keys.

This library will also be serving as the new lower layer for my well-known phi_prompt user interface library to render interactive lists, messages, menus, etc. with any input device and an LCD. Read more about this library on my blog.

2. Main features

Here are the main features of the phi_interfaces library:

- ☆ Compatible with Arduino IDE 1.6.0
- ☆ Group of push buttons, hold-and-repeat, and multi-tap if there are enough keys.
- ☆ Matrix keypads of any dimension, hold-and-repeat, and multi-tap.
- ☆ Rotary encoders, the library makes them act just like regular keys.
- ☆ Analog buttons of any amount and matrix of keys with several analog buttons, hold-and-repeat, and multi-tap if there are enough keys.
- ☆ Phi-panel serial LCD keypad as input device
- ☆ Using android phone over Bluetooth serial port
- ☆ Simulate keypad inputs over serial port with Arduino serial monitor or program.
- ☆ Easily expandable to any other button or keypad types with already-included state machine for single buttons and keypads.
- ☆ One common interface so you can replace your type of interface without changing any of your codes, say go from analog keys on a shield to matrix keypad on the final project without changing code.

The following diagrams depict the hierarchy of the classes:

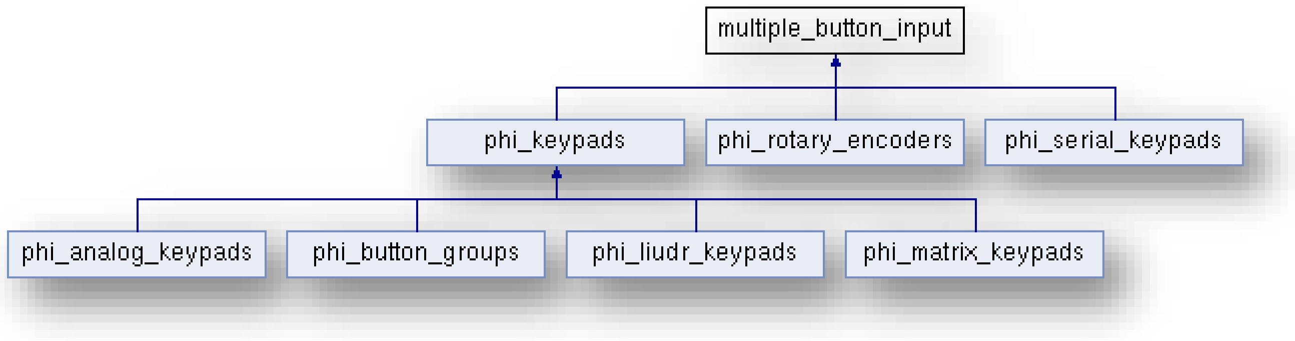


Figure 2-1. The multiple_button_input class hierarchy.

Since all input devices inherit from base class multiple_button_input, which has a method getKey, you can read any of these inputs with the getKey().

There are a few key words I need to explain here:

<u>Scan code</u>: a number between zero and the total number of keys on a particular input device. It is generated each time a confirmed key press occurs or a NO_KEYs is generated if there is no confirmed key press. It is never exposed to the outside of the class so please don't poke it out.

<u>Key names</u>: each key press (scan code) on an input device is translated into a key name, a single character such as 'A' or '1'. You call getKey() on any input device and you will end up with such a character or NO_KEY.

<u>Mapping</u>: the correspondence between the scan code and the key names in a char array. Say you have 3 buttons. You want them to generate 'A', 'B', and 'C' when you press them, you create a mapping: *char mapping[]={'A', 'B', 'C'};*

<u>The implications of the above concepts:</u>

- **You can have more than one button generate the same key name, thus performing the same function.**

  Say keys 1 and 2 both generate 'A'. Then these two keys will induce identical results. If a key 'A' in your program toggles a light on and off, then keys 1 and 2, located at two opposite ends of a hallway will automatically become two toggle switches for the hall light. You can have any number of switches generating same key names to have multiple toggle switches to control the same hall light.

- **You can establish alternate interfaces, one with hardware keys on Arduino, and more over software on smartphones and computers over the internet.**

  They all operate the same way. It's like you can control your project locally with a keypad, on your PC, or online with a PC or on a smartphone a world away. Say if you also use a serial port keypad, then if your computer sends 'A' down the serial port, Arduino toggles that hall light, not to mention whether your computer does that because you tell your smartphone to tell your computer to do it from thousands miles away or your computer does that according to a program you write to randomly turn on and off lights to deter burglary while you are away ☺

  You can certainly send key presses from your smartphone to your Arduino via Bluetooth wireless link too, if you are in the room with the Arduino, only that the Arduino is hidden and you miraculously turn your hall light on and off with your phone.

  See the following figure for a concept:

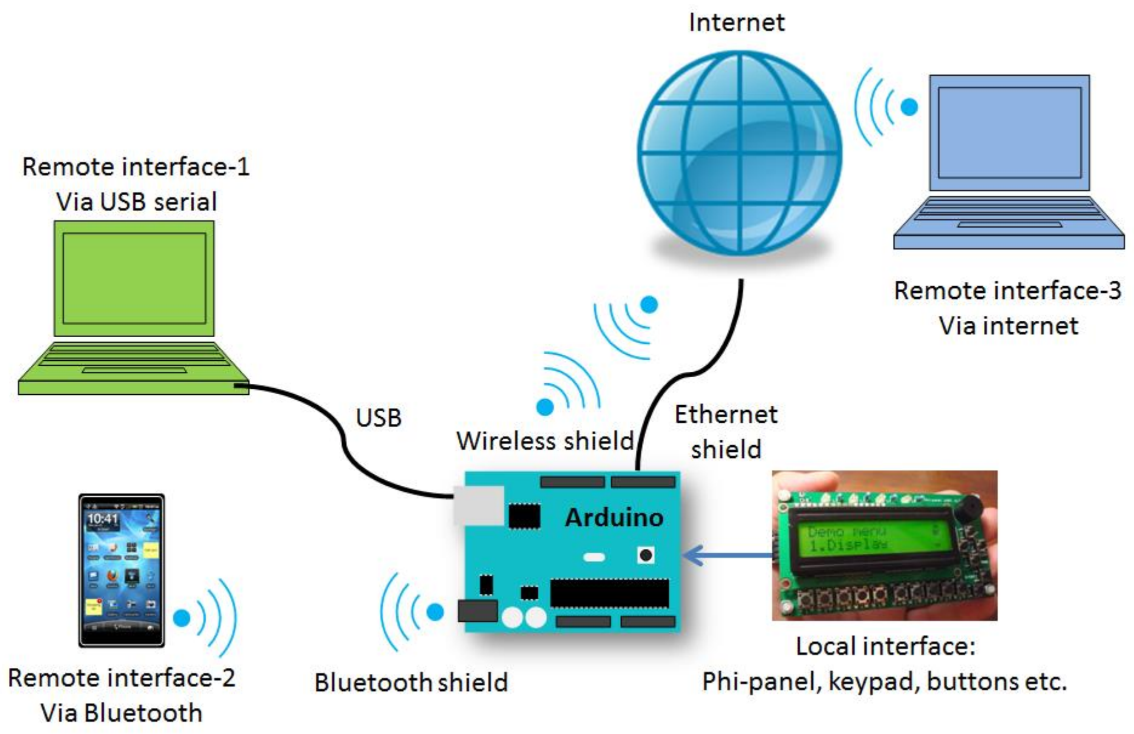


Fig. 2-2. With phi_interfaces library, you can easily establish a local interface on your Arduino project, then go ahead and duplicate the same interface remotely, however you like it, on a smartphone, a PC or another machine over the internet.

Just imagine what else you can do! (Imagine pigeon, leaves, globe, etc.)

3. Updates

The current version 1.0 (for Arduino IDE 1.0). More updates will be released over the year of 2012, possibly including but not limited to the addition of support to capacitive keypads, IR remote controls, PS/2 keyboards, touch screen keyboards, Ethernet shield as input stream etc.

4. Short examples

The current release of the library comes with a number of sample codes. The following are their introductions.

1) Push buttons

Push buttons are just tactile switches. You press it down, it makes the connection. Typically you need one digital input pin to sense one button. This limits how many buttons you can connect to your Arduino UNO, since there are only 20 pins on it. On the other hand, push buttons come in all different shapes, colors, and sizes, you can put them anywhere. You can make a nice panel with just a few old-school circular arcade game buttons located in a few places, like under pictures of animals or on pedestals of art pieces. Instead, a keypad can only be located at a central location, which may work better for more technical projects.

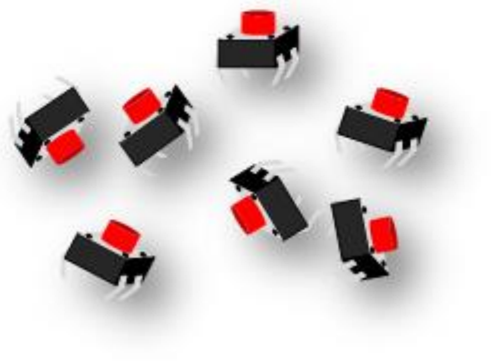


Fig. 4-1 Tactile push buttons.

My phi-2 shield has 6 push buttons. Each button is connected to one Arduino pin. See the following picture:

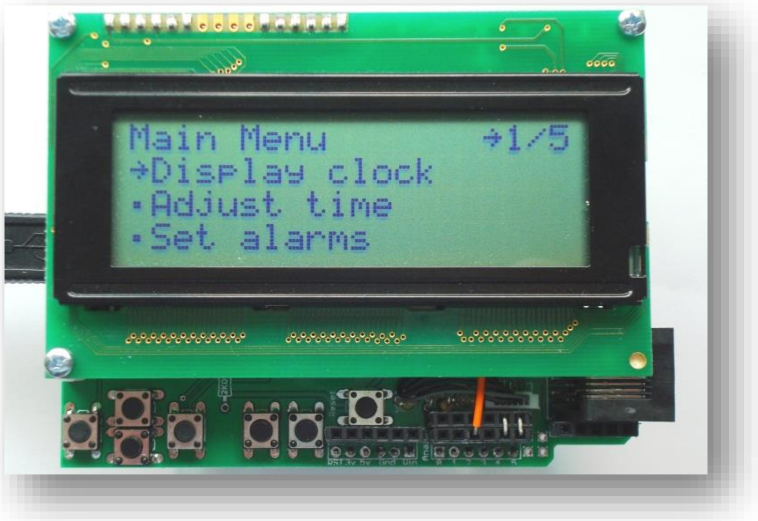


Fig. 4-2 Phi-2 2004 shield with 6 push buttons and a reset button.

To connect a push button to Arduino, connect one side of it to ground and the other side to an Arduino pin.

To control a group of push button, you can use the ***phi_button_groups*** class. The following is example code for 6 push buttons on my phi-2 shield with You may use this for your own set up if you change the pin definitions.

```
#include <phi_interfaces.h>

#define btn_u 5
```

```
#define btn_d 10
#define btn_l 11
#define btn_r 4
#define btn_b 14
#define btn_a 15

#define total_buttons 6

char mapping[]={'U','D','L','R','B','A'}; // This is a list of names for each button.
byte pins[]={btn_u,btn_d,btn_l,btn_r,btn_b,btn_a}; // The pins connected to the 6
buttons.
phi_button_groups my_btns(mapping, pins, total_buttons);

void setup(){
  Serial.begin(9600);
}

void loop(){
  char temp;
  temp=my_btns.getKey(); // Use phi_button_groups object to access the group of buttons
  if (temp!=NO_KEY) Serial.write(temp);
}
```

The above code includes the library .h file in line 1 and then creates a ***phi_button_groups*** object with 6 buttons in the group and senses them in loop() to print out any button presses. The key presses are translated according to the char array mapping. It's very simple to understand. You call getKey() in a loop to see if a key is pressed or no keys (NO_KEY is returned).

2) Matrix keypad 4X4

A matrix keypad solves the problem of single push buttons costing too many pins. A keypad as the one in the following figure has 16 keys and only costs 8 pins instead of 16. The secret is the interconnection of the keys. The keys are arranged in 4 columns and 4 rows. The connections behind the keys are also arranged in 4 columns and 4 rows. Each key sits on top of the crossing of a column wire and a row wire. If you press the key down, the column is connected to the row. This arrangement needs not to force keys in rigid columns and rows as in the figure. Since column and row wires can be routed anywhere, so can the keys be located. Such custom membrane keypad other than the square layout will likely cost hundreds of dollars to prototype so you only see them on commercial products instead of on hobby projects. You can wire your own keypad but it gets a big confusing if you have too many keys.

The matrix keypad in the figure is made with membrane and has adhesive backing for easy installing. You will need a clean and flat surface, cut a slot at the bottom of the space where the keypad will go, so its cable will pass through the surface. To connect this matrix keypad to Arduino, connect its connectors to 8 Arduino pins, the first 4 are rows and the next 4 are columns. See how I connected my phi-panel serial LCD keypad backpack to a nice enclosure on my blog. A picture is attached. Very good looking, right?

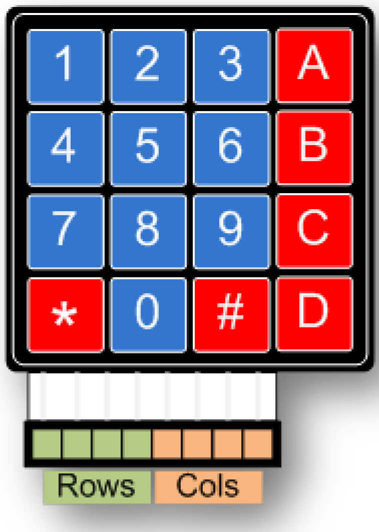


Fig. 4-3 A 4*4 membrane matrix keypad. A typical keypad has a connector at the bottom of the pad and row pins are followed by column pins. Adhesive backing is provided too.

Fig. 4-4 A mounted matrix keypad on the outside of a box enclosure with an LCD.

To use a matrix keypad in your Arduino project, simply create an instance of the ***phi_matrix_keypads*** class. Make sure you supply an array with names of each key and an array with the row and column pins. With a large enough keypad, typically 4X4, you can even run my multi-tap code to enter all characters (upper and lower cases) and symbols.

The following is example code for a 4X4 matrix keypad with ***phi_matrix_keypads*** class:

```
#include <phi_interfaces.h>
```

```
#define btns_per_column 4
#define btns_per_row 4
char mapping[]={'1','2','3','A','4','5','6','B','7','8','9','C','*','0','#','D'};
byte pins[]={17, 16, 15, 13, 12, 11, 9, 8}; // Row pins then column pins
phi_matrix_keypads panel_keypad(mapping, pins, btns_per_row, btns_per_column);

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  char temp;
  temp=panel_keypad.getKey(); // Use phi_keypads object to access the keypad
  if (temp!=NO_KEY) Serial.write(temp);
}
```

The above code creates a ***phi_matrix_keypads*** object to sense a 4X4 matrix keypad. The row and column pins, in that order, are stored in pins[]. The names of the keys mapped to all the keys, one row at a time, are stored in mapping[]. You call getKey() in a loop to see if a key is pressed or no keys (NO_KEY is returned).

3) Rotary encoder

A rotary encoder looks like and act like a potentiometer, or a knob. The difference is that a rotary encoder usually has no mechanical limit so you can make any number of turns in either direction. A potentiometer is a variable resistor and is mechanically limited with a single turn, or a few turns. A rotary encoder works well in situations a potentiometer are suitable for and more. You can use a rotary encoder to perform up/down actions to select a menu, or volume up/down or position left/right adjustment etc. You can easily reset the position of the rotary encoder since there is no limit. Some rotary encoder even has a clickable shaft to add more function to it.

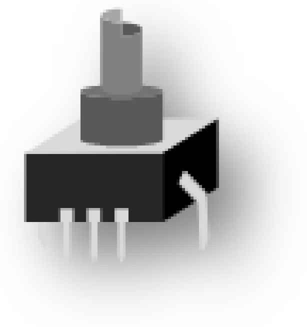


Fig. 4-5. A rotary encoder. The three pins in the front are channels A, common, and B. Some rotary encoders have detents and some even have clickable shafts. You may add a knob of any fancy shape to the shaft to add to its look.

A rotary encoder has two channels, A and B, essentially like two buttons, and one common pin. Clickable shaft has two pins just like a push button. Some encoders have detents, which are positions around the rotation that arrests the shaft. You feel a click while you spin the shaft. Some encoders don't have detents. It is highly recommended to purchase encoders with detents, at least 12 per rotation.

To connect a rotary encoder, connect the channels A and B to two Arduino pins, connect the common (usually the center pin) to ground. If your encoder has a clickable shaft, connect one pin to ground and the other pin to an Arduino pin.

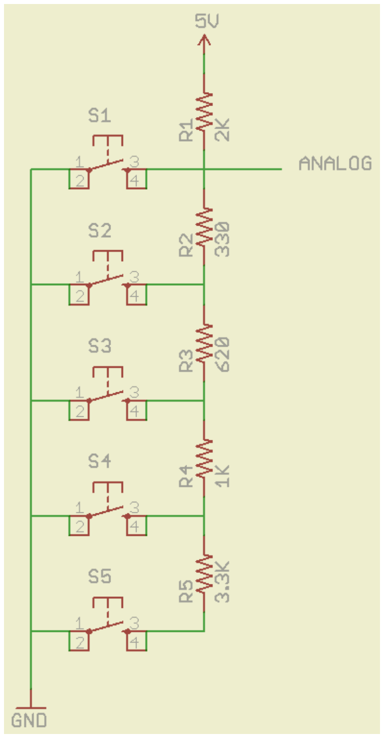
To use a rotary encoder in your Arduino project, simply create an instance of the ***phi_rotary_encoders*** class. Provide which pins are connected to the encoder channels and how many detents. Also provide a char array with two characters to represent the return value of the encoder when you rotate it forward and backward one detent. The following is example code for a rotary encoder:

```
#include <phi_interfaces.h>

#define Encoder1ChnA 2
#define Encoder1ChnB 3
#define Detent 12

char key_mapping_matrix1[]={'U','D'}; // Encoder 1 returns U for up and D for down.
phi_rotary_encoders my_encoder1(mapping1, Encoder1ChnA, Encoder1ChnB, Detent);

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  char temp;
  temp=my_encoder1.getKey(); // Use phi_keypads object to access the keypad
  if (temp!=NO_KEY) Serial.println(temp);
}
```

The above code uses the ***phi_rotary_encoders*** class to sense a rotary encoder. The encoder returns 'U' and 'D' if it is dialed up or down one notch. You need to call getKey() in a loop to see if the encoder is turned or not (NO_KEY is returned). If it is turned forward, a 'U' is returned. If backward, a 'D' is returned.

If you have a clickable shaft, treat it like a push button and create an instance of the ***phi_button_groups*** class to sense all push buttons and all rotary encoder shaft clicks.

4) Analog buttons



If you are running out of digital pins in a complex project, you may consider to sense several buttons with one analog pin. You connect a number of resistors in series between 5V and ground. Between every two resistors, you connect a button. You connect the other side of the buttons to ground. Sensing analog pins takes much longer time than sensing digital pins but not much more than a fraction of a millisecond. There is nothing bad about analog buttons except that you need several resistors, meaning more parts for the project.

To construct a typical analog button keypad, read the attached figure. Notice the analog pin is connected between the first and second resistor next to 5V. There are many ways to connect them.

To use analog button keypads in Arduino project, you simply create an instance of ***phi_analog_keypads*** class. If you need more than 5 buttons, you can duplicate the same setup on a second analog pin or even more. So if there are 5 buttons connected to each analog pins and there are 4 analog pins with identical setups (buttons, resistors and how they interconnect should all be the same on all analog pins), you get 20 buttons. Just provide the analog pins (0,1,2 etc.) to the constructor, along with analog values when each button is pressed down, number of buttons on each analog pin, total number of analog pins, and a char array for the key names. Please look at the following code:

```
#include <phi_interfaces.h>

#define buttons_per_column 5 // Each analog pin has five buttons with resistors.
#define buttons_per_row 1 // There are two analog pins in use.

byte keypad_type=Analog_keypad;
char key_mapping_analog[]={'1','2','3','4','5'}; // This is an analog keypad.
byte row_pins_analog[]={0}; // The pin numbers are analog pin numbers.
int values[]={0, 146, 342, 513, 744}; //These numbers need to increase monotonically.
phi_analog_keypads pad(mapping, pins, values, buttons_per_row, buttons_per_column);
void setup()
{
  Serial.begin(9600);
}

void loop()
```

```
{
  byte temp=pad.getKey(); // Use phi_keypads object to access the keypad
  if (temp!=NO_KEY) Serial.write(temp);
}
```

The above uses ***phi_analog_keypads*** class to sense 5 buttons connected to analog pin 0. The values[] contains analog values when each button is pressed down. You may find this value with analogRead() and print it out on the serial port and copy it to the values[] array. The match needs not to be exact. If the analog value is within the value ± 10, the button is considered pressed down. One thing to make sure is that the values stored in values[] should increase monotonically.

5) Serial keypads

A serial keypad is an actual keypad or a virtual keypad that is connected to an Arduino's hardware or software serial port. Software serial keypad is only supported for Arduino 1.0 while hardware serial keypad is supported for Arduino 1.0 and 0022.

An actual keypad is a keypad, usually integrated with a display that communicates with Arduino with serial port. The most functional serial keypad that I am aware of is my own product, the family of phi-panels. The following figure gives you an overview of the phi-panels:

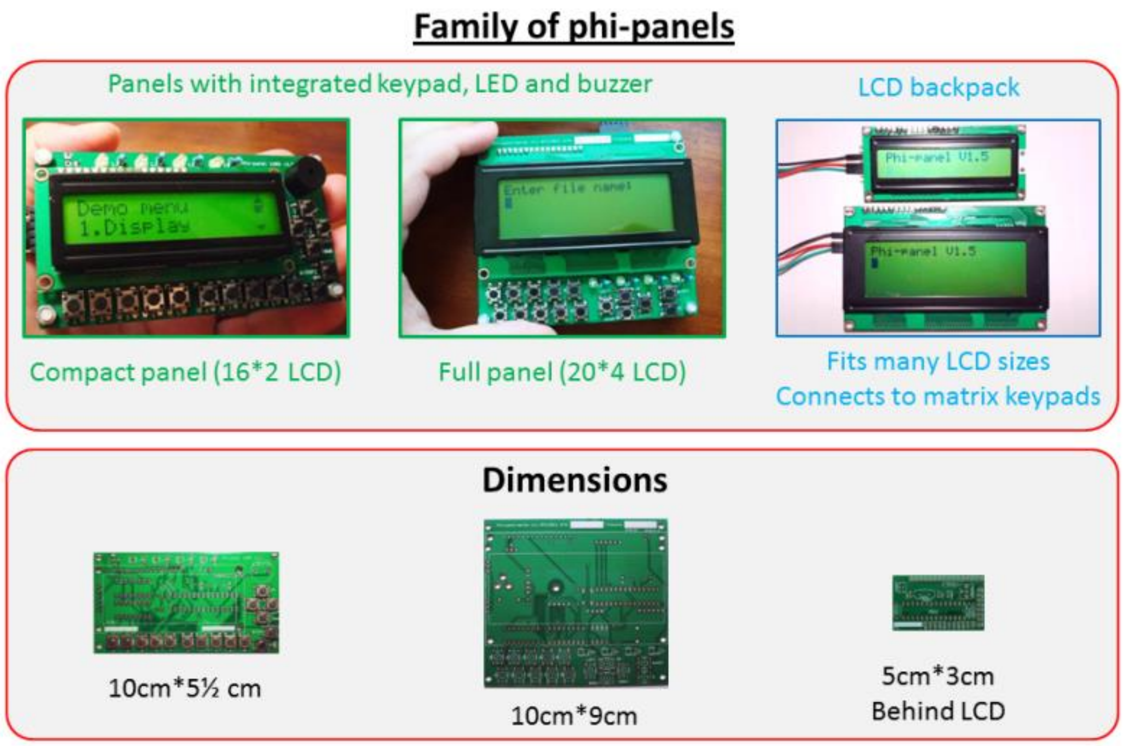


Fig. 4.7 Family of phi-panels, my line of serial LCD keypad products. The compact and full panels have integrated keypads. The backpack has a connector for up to 4*4 matrix keypad.

Phi-panels and other serial keypad panels do is they process the key presses on their on-board processors and stream key presses via serial port, all without the Arduino involved. For phi-panels, there are 16 keys, number keys return characters between '0' and '9'. For convenience of editing functions, up/down/left/right/enter returns '-'/space/back space/period/new line. Escape calls up the on-board interactive menu to adjust panel settings and is not relayed back to Arduino via serial.

The panel also has special functions to help render long messages, lists, menus, Y/N dialogs etc. You are encouraged to find its documentation on my blog.

Besides, the phi-panel also has an option to enable multi-tap. Once engaged, the panel only returns confirmed keys are passed to Arduino via serial port. Intermediate key presses are processed and displayed on the panel, so the Arduino needs not know the multi-tap is engaged and needs not do anything special to accommodate the multi-tap. The following is a multi-tap key diagram:

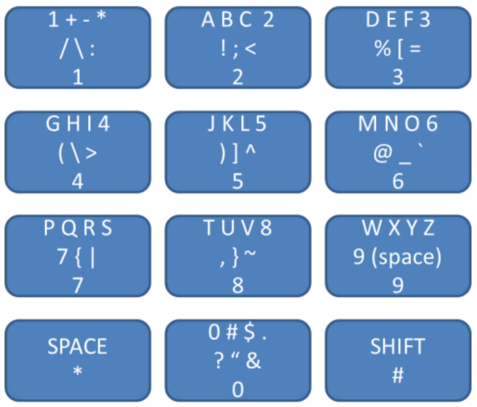


Fig. 4-8 Phi-panel multi-tap keypad diagram. Pressing '#' once will toggle shift between upper and lower cases. You don't need to hold the '#' key.

Although this class was originally designed for my phi-panel serial LCD keypad and other serial keypads, you can also use it for Bluetooth-enabled smartphones to "wirelessly press a key on Arduino", or simulate key presses with Arduino serial monitor, or generate a stream of characters from a PC program to generate key presses for Arduino.

Devices that can act as serial keypads are: actual serial keypads, smartphones via Bluetooth serial port with Arduino and a Bluetooth link, another Arduino or a PC via Xbee, or other devices that relay information to Arduino serial port. See figure 4-9.

The purpose of a serial keypad is simple: you want to focus on your project codes and completely ignore how a keypad is implemented, hardware and software. All you want is a stream of key presses sent to Arduino via serial port. This way, a few benefits are obvious:

- The key presses are buffered against lost even if you don't sense the keys frequently enough, since serial ports are always buffered.

- It frees you from having to know how to sense some special keypad and keep reading the keypad for a press.
- It also makes anything that streams characters into a keypad device.

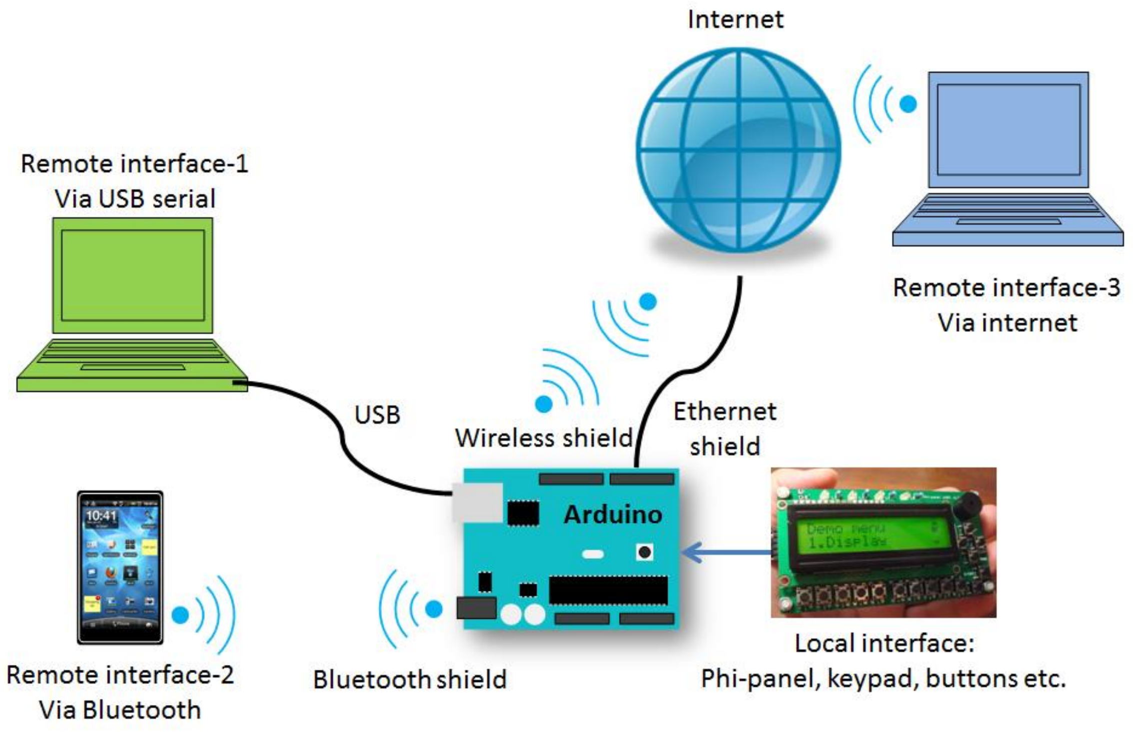


Fig. 4-9 (a.k.a.2-2) As long as you connect an Arduino serial port to some device that streams characters over the serial port, that device is considered as a serial keypad.

Here is the detail to the last point, a very nice point. You can use a Bluetooth shield on Arduino and connect it to your Bluetooth-enabled smartphone. Then you write some app on your phone with a few buttons. Say each button will send a character to Arduino via the Bluetooth serial port. You then push the buttons on your phone and Arduino receives characters representing each button press you made on your phone touch screen. Arduino doesn't care if the key presses are from actual keys or from a smartphone touch screen, it will do what you want it to do when a certain key is pressed. You can also write a program to automatically send characters to Arduino via serial port. These characters will be interpreted by Arduino as key presses and Arduino will do what it is told to do when these keys are pressed. So you get an interface that you can operate with actual keys on Arduino project, or over serial port on your smartphone, or even over the internet (use internet to relay a key press from afar and send it to Arduino via serial (or Bluetooth). This is truly powerful.

The following code is simple enough to need no explanations:

```
#include <phi_interfaces.h>
phi_serial_keypads panel_keypad(&Serial, 19200);

void setup()
```

```
{
  Serial.begin(19200); // Default baud rate for phi-panels
}
void loop()
{
  char temp;
  temp=panel_keypad.getKey(); // Use phi_keypads object to access the keypad
  if (temp!=NO_KEY) Serial.write(temp);
}
```

Please note the ampersand ('&') in front of the hardware serial port keyword Serial is needed for proper transfer of the hardware (or software) serial port object address to the ***phi_serial_keypads*** object. The baud rate is used in the constructor but is not utilized. A future version may make use of it.

If you are using a phi-panel, then the key presses are relayed back to the panel's LCD via serial port so you see your key press echoes back on the LCD. You will notice the left key, which is the back space does what it is supposed to do on a computer so does the enter key. If you use the same code with an Arduino Bluetooth shield and a Bluetooth-enabled PC or smartphone, you can use the free app SENA BTerm on android phones to send key presses to Arduino and the result is relayed back to your phone.

Of course you can program Arduino to do other things, such as sense temperature and relay back to your phone, if you send a 'T' character or maybe send an 'L' to turn on hall light ;)

6) Liudr keypads

This is a keypad LED indicator combination that I use in my phi-panel serial LCD keypad design. I have not published its design yet so it’s of internal use to my phi-panel firmware only.

5. Useful functions

There are many useful functions that you may want to know. They include how to adjust keypad behavior and special functions. They are listed below.

1) getkey()

Details:

This is the public method to get a key press from any input hardware class defined in this library. The key press is translated into one-character-long names according to key_names[] or a NO_KEY is returned if there are no key presses.
This function code is inherited from multiple_button_inputs. All subclasses share it. You want to sense your matrix keypad, you call this function. You want to sense your rotary encoder, you call this function too.

Since all ***multiple_button_inputs*** devices have this method, you can treat all of them as ***multiple_button_inputs*** and call this method to get a key press from any type of input.

This is the function you should call to sense key presses.
Return: It returns the name of the key that is pressed down, such as 'A', '1', or 'U'.

2) *get_angle()*

Details:
Get the angle or orientation of a rotary encoder between 0 and detent-1. This function calls getKey to update the angle.
If you call getKey BEFORE get_angle, you get the dial up/down from getKey and the correct angle from get_angle. If you call get_angle BEFORM getKey, the dial up/down is read and lost but you get the correct angle.
So make your decision. Do you want just dial up/down actions? Then only call getKey. Do you want just angle? Then only call get_angle. Chances of you need them both is very slim but as mentioned you should call getKey first.

To properly update the angle, you need to call this function inside of a loop.
Return: It returns a value between 0 and detent-1. You can calculate angle with it return.

3) *set_hold(unsigned int ht)*

This sets how long the button needs to be held before it is considered held down.

4) *set_debounce(unsigned int dt)*

This sets how long the button needs to be held before it is considered pressed.

5) *set_repeat(unsigned int rt)*

This sets how often the button press repeats after being held.

6) *Other functionss*

This document is meant for a quick start for anyone trying to use the library. All functions are documented in the detailed documentation refman.pdf.

6. Future improvement

To get involved in future improvements, please visit my blog and leave some feedback: http://liudr.wordpress.com and leave your feedback under phi_interfaces. The future improvement to include all possible input devices rely on you. If you use this library and like it, please consider donating $5 on my blog (right side has a button). If you want to use this in a commercial product and make money off the product, please read the next section.

7. The legal stuff