

Phi -menu Software Documentation

1. Number and enumerate inputs

Enter date:
 DEC 05 2010 SUN

Numbers are bound:
2000-2100 for year and 1-31 for day

Alarm clock:
 08:10 Weekday

Enumerates are bound:
MON-SUN, JAN-DEC, YI

Erase EEPROM?
 YES

Enumerates are custom
Weekday, Weekend, Eve

3. Classic menu with index

Active menu item

Parameters
 123 56789

Index
^ Indicator symbol

1. Classic menu
One item at a time

Menu:
 Live GPS Display

↑
Active menu item

2. Two by two menu

Active menu item

• Update • Test PW
 • Credit • Game 1

Menu items 1,2 (top) and 3,4 (bottom)
> Indicator symbol

2. Number and arbitrary text inputs

Enter password:
 487259

Range of characters is customizable
Use **number only** for password

Enter word:
 HELLO WORLD! 25

Use **full range** for **arbitrary** text input
Other ranges can be defined as well.

*Last reviewed on 3/13/2011
John Liu*

1.	Introduction.....	3
I.	What is phi-menu.....	3
II.	How phi-menu helps your project.....	3
III.	Logic layers.....	4
IV.	Minimal effort to run phi-menu.....	5
2.	Program details.....	5
I.	Phi_menu_XXX.pde.....	5
•	Credits.....	6
•	Includes.....	6
•	Core variable definitions and object instantiations.....	7
•	setup() function.....	9
•	loop() function.....	9
•	Show_credit() function.....	9
II.	FUNCTIONS.pde.....	10
•	Menu PROGMEM definitions.....	11
•	User PROGMEM definitions.....	11
•	User variable definitions and object instantiations.....	11
•	Menu renderers.....	14
•	The menu.....	12
•	Menu items.....	12
III.	ALIUDRuser_interface.pde.....	15
•	Rendering functions.....	17
•	Interaction functions.....	17
IV.	buttons.cpp and buttons.h.....	19
V.	defs.h.....	20
•	Shield definitions.....	20
VI.	progmem.h.....	21
•	LCD custom character PROGMEM definitions.....	21
•	Other common keywords PROGMEM definitions.....	22
3.	An example.....	22
4.	Future improvements.....	25
5.	The legal stuff.....	25

1. Introduction

I. What is phi-menu

The phi-menu is the software layer that supports user interactions on the popular Phi-1 and Phi-2 shields. Using the onboard LCD and keypad, a user can easily browse a menu, enter YES/NO, numbers, date, time, select from custom list, or enter passwords and arbitrary texts. It provides the ultimate front end of human machine interaction (HMI) for your projects that is light in weight, easy to add, modify and looks professional. You can concentrate on your project and let phi-menu handle the rest.

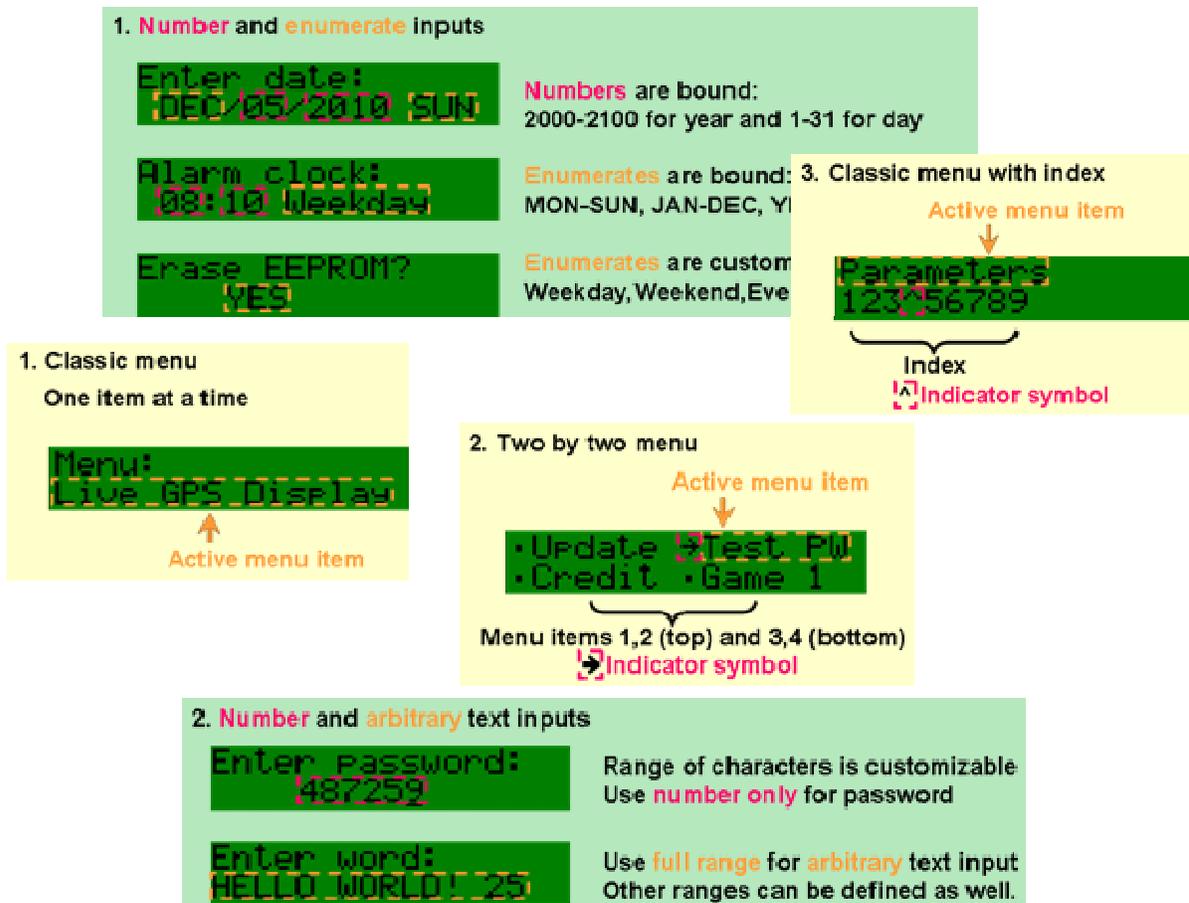


Figure 1-1: Collage of phi-menu features.

II. How phi-menu helps your project

Phi-menu provides three different styles of customizable LCD menus (see Fig. 1-1 yellow shaded blocks or details in the menu renderer section) together with rendering and

interaction functions to collect user inputs such as strings, numbers, Y/N, month, day of the week, time, even Morse code input (see Fig. 1-1 green shaded blocks or details in the rendering and interaction function sections).

To develop this level of user interaction with flexibility and sophistication on your project, you need months. You could also attempt to hodgepodge codes from separate online authors and see them fight since those codes weren't written to work together. You can also choose the open-source phi-menu to easily develop your own interactive projects without having to deal with the hassles. From the very start, your project looks like a million dollars.

III. Logic layers

The phi-menu contains a number of program codes. They are all related in their simple hierarchy. To help understand these codes and use them, here is a diagram:

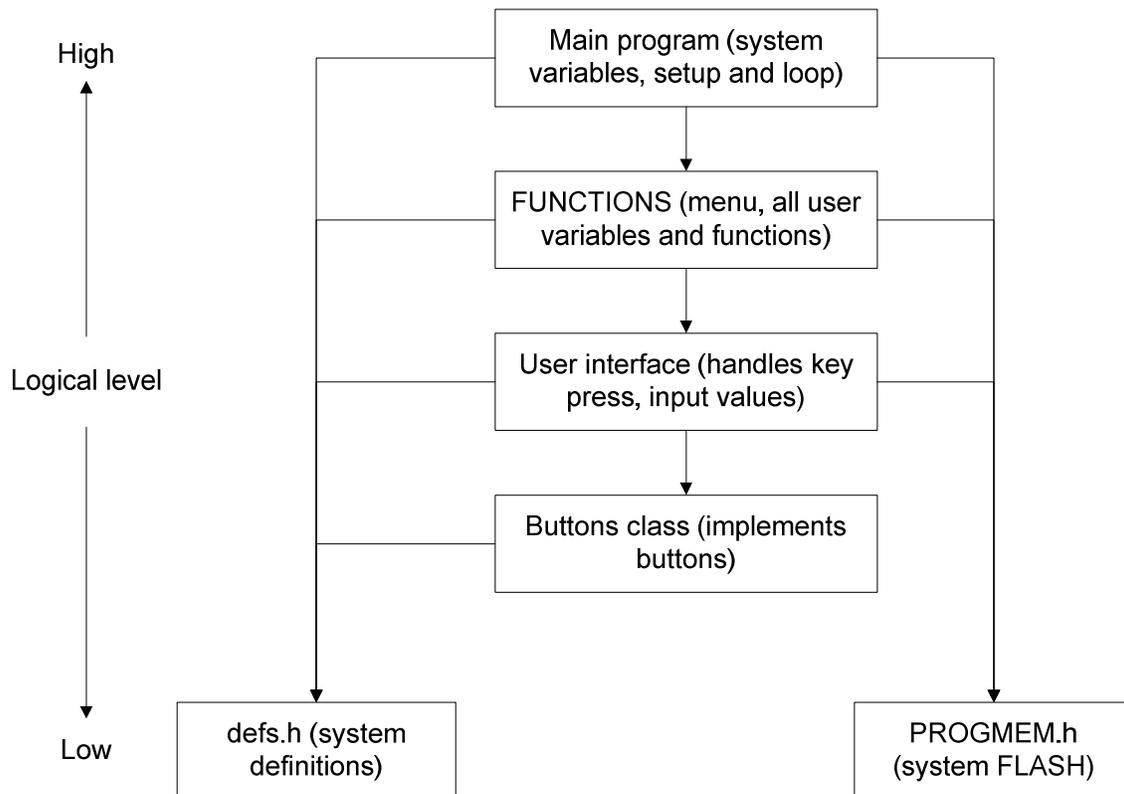


Figure 1-2: Structure of the phi-menu

- The main program contains global variables and objects, setup() and loop().
- The “FUNCTIONS.pde” contains all project-related contents, including variables, menu texts, the menu handler do_menu, and menu functions that are called if the corresponding menu item is selected.

- The “ALIUDRuser_interface.pde” contains functions that handle user inputs such as numbers, strings, month, dates, and support functions to display the menu.
- The buttons.h and buttons.cpp define buttons class. You can use physical buttons, button-like digital sensors such as Hall Effect switch and photogate, or virtual buttons such as null button or auto button. An auto button can automatically cycle through some input values (say MON-SUN) so the user can just confirm when the desired value comes up. This saves you one button and one Arduino input pin.
- The defs.h contains important definitions in order to run all codes, such as which pins are connected to the LCD and which pins are connected to which button.
- The progmem.h contains system static string messages used on the LCD, such as months, days of the week. Please define your own in the FUNCTIONS.pde.

In order to isolate system code and user code for debugging and easy to read, you are only allowed to write your codes in the FUNCTIONS.pde and the main pde file. Please DONNOT change the content of the other files.

IV. Minimal effort to run phi-menu

The phi-menu distribution package comes complete with a sample menu and a few sample examples. Here are the things to do in order to run phi-menu on your code:

1. Type in your menu texts such as “Display speed” in FUNCTIONS.pde
2. Code each menu functions such as “menu_function0” in FUNCTIONS.pde
3. Hit compile and upload

At the end of this document, in section 3, an example is presented. You can digest the example and go back to section 2 for details of each program. If you prefer, you can try to understand everything in section 2 and examine the example code later. The last section is legal stuff. Always, more samples and stuff can be found on my website:

<http://liudr.wordpress.com/phi-menu/>

2. Program details

The following are details of each program file in the phi-menu distribution package:

1. Phi_menu_XXX.pde

This is the main program with setup() and loop(). You should change it to your preferred project name. Its sections are described below:

• Credits

This part contains the project description, my contact info, a summary, and a list of functions. You can add your credits before mine but please don't remove this part. If you like this code, pass it on, with its summary and credits. Think what you would do if others pass your code around with missing pieces.

```
/*
```

```
Phi-1 shield and phi-menu for Arduino
```

```
Test program: Phi-menu car backing/reverse obstacle sensor with sonic ranger v1
```

```
This program is fairly long. For a nutshell version of this program to study with, visit
```

```
http://liudr.wordpress.com/phi-1-shield/
```

```
Programmed by Dr. John Liu
```

```
Revision: 03/04/2011
```

```
Commercial use without authorization is prohibited.
```

```
Find details of the Phi-1 shield, Phi-2 shield or contact Dr. Liu at http://liudr.wordpress.com/phi-1-shield/
```

```
All rights reserved.
```

```
Summary:
```

```
This software alerts a driver the distance to an obstacle behind the car for parking or backing safety with both audio and LCD.
```

```
Distance sensing: 0m to 6m
```

```
List of functions:
```

```
* Menu gives you several choices:
```

```
* Display distance is the main function. Any key escapes to main menu.
```

```
* Parameters menu allows you to adjust parameters on frequency of audio beeps and turn audio on and off.
```

```
*/
```

• Shield selector

This part selects which Phi shield you use. Note that Phi-2 shield uses slightly different pins than Phi-1 shield but that all gets taken care of if you select the shield below:

```
#define phi_1_shield
```

```
//#define phi_2_shield
```

To use Phi-2 shield, comment the first line and uncomment the second line. This is the last release that defaults into phi-1 shield. The next release will default into phi-2 shield. With phi-2 shield, you also need to jump the buzzer pin to analog pin 2.

• Includes

This part contains all the necessary includes. Please don't remove these commands. You may add your include in the end.

```

#include <LiquidCrystal.h>
#include <WProgram.h>
#include <Wire.h>
#include <stdio.h>
#include <avr/pgmspace.h>
#include <EEPROM.h>
#include "defs.h"
#include "progmem.h"
#include "phi-buttons.h"

```

• Core variable definitions and object instantiations

This part contains all global variables needed to run the programs and all instantiations of objects such as **lcd** and **all buttons**. You can choose buttons scenario to use in your project. Each scenario uses less buttons thus saves more Arduino pins for your project than the last. You can go back and forth between scenarios. Unless you want to remove the LCD or redefine certain buttons you should leave this part alone.

User modification: you should not modify this.

```
byte menu_pointer=0;
```

```
LiquidCrystal lcd(LCD_RS, LCD_EN, LCD_D4, LCD_D5, LCD_D6, LCD_D7); // Create the lcd object
```

These are initializations of the buttons.

There are many scenarios you can use. A user can operate the menu and input with as many as 6 buttons for convenience or as few as one button to save Arduino I/O pins. Just enable the respective define. Right now the 6-button scenario is used.

use_6_btn: all 6 buttons are defined for maximal convenience to the operator. Operator may jog in text input.

use_5_btn: all but the “left” are defined for convenience to the operator but there is no way to jog on text input.

use_4_btn: all but “left” and “right” are defined for medium convenience to the operator. This works best when all you do is entering numbers or choosing from a list and don't use text input.

use_3_btn: only “up”, “right” and “confirm” are defined to save Arduino pins offering less convenience to the operator. This works alright if input options are limited such as month or password numbers.

use_2_btn: only “up” and “right” are defined to save Arduino pins. The operator will still be able to operate everything but entering text will be a pain.

use_1_btn: This enables auto button on “up” and only defines “right”. This way saves most Arduino pins since only one button is used but the operator is only expected to enter limited information.

```

#define use_6_btn
//#define use_5_btn
//#define use_4_btn
//#define use_3_btn
//#define use_2_btn
//#define use_2_btn

```

```
//#define use_1_btn
```

```
#ifdef use_1_btn
```

```
phi_buttons btn_1(btn_auto, LOW);  
phi_buttons btn_2(btn_null, LOW);  
phi_buttons btn_3(btn_null, LOW);  
phi_buttons btn_4(btn_r, LOW);  
phi_buttons btn_5(btn_null, LOW);  
phi_buttons btn_6(btn_null, LOW);  
#endif
```

```
#ifdef use_2_btn
```

```
phi_buttons btn_1(btn_u, LOW);  
phi_buttons btn_2(btn_null, LOW);  
phi_buttons btn_3(btn_null, LOW);  
phi_buttons btn_4(btn_r, LOW);  
phi_buttons btn_5(btn_null, LOW);  
phi_buttons btn_6(btn_null, LOW);  
#endif
```

```
#ifdef use_3_btn
```

```
phi_buttons btn_1(btn_u, LOW);  
phi_buttons btn_2(btn_null, LOW);  
phi_buttons btn_3(btn_null, LOW);  
phi_buttons btn_4(btn_r, LOW);  
phi_buttons btn_5(btn_b, LOW);  
phi_buttons btn_6(btn_null, LOW);  
#endif
```

```
#ifdef use_4_btn
```

```
phi_buttons btn_1(btn_u, LOW);  
phi_buttons btn_2(btn_d, LOW);  
phi_buttons btn_3(btn_null, LOW);  
phi_buttons btn_4(btn_null, LOW);  
phi_buttons btn_5(btn_b, LOW);  
phi_buttons btn_6(btn_a, LOW);  
#endif
```

```
#ifdef use_5_btn
```

```
phi_buttons btn_1(btn_u, LOW);  
phi_buttons btn_2(btn_d, LOW);  
phi_buttons btn_3(btn_null, LOW);  
phi_buttons btn_4(btn_r, LOW);  
phi_buttons btn_5(btn_b, LOW);  
phi_buttons btn_6(btn_a, LOW);  
#endif
```

```
#ifdef use_6_btn
```

```
phi_buttons btn_1(btn_u, LOW);  
phi_buttons btn_2(btn_d, LOW);  
phi_buttons btn_3(btn_l, LOW);  
phi_buttons btn_4(btn_r, LOW);  
phi_buttons btn_5(btn_b, LOW);  
phi_buttons btn_6(btn_a, LOW);  
#endif
```

- **setup() function**

The first few lines **set up the LCD** and **loads custom characters**. Next, **buzzer and LED** pins are set up. Unless you want to use the buzzer and LED pins other than driving the buzzer and LED, you should not change this part.

User modification: you should not modify this.

Next part has credit. You're welcome to change this part. Please do your necessary set up after the credits so the code stays well-organized.

```
void setup()
{
  byte ch_buffer[10]; // This buffer is required for custom characters on the LCD.
  lcd.begin(16, 2);

  //Programming custom characters. The custom characters will be lost when power is cut.
  #ifndef prog_char_now
  for (int i=0;i<8;i++)
  {
    strcpy_P((char*)ch_buffer,(char*)pgm_read_word(&(ch_item[i])));
    lcd.createChar(i, ch_buffer);
  }
  #endif

  // Set the two channels to output to drive the buzzer and LED.
  pinMode(buzzer,OUTPUT);
  digitalWrite(buzzer,LOW);
  pinMode(led,OUTPUT);
  digitalWrite(led,LOW);

  // Display credits
  show_credit();
}
```

- **loop() function**

This function is very simple. It only has one line, to run the menu:

```
void loop()
{
  do_menu();
}
```

Since `do_menu()` doesn't trap execution, the `loop()` is running the menu non-stop. If you add anything in the loop, it gets executed with the menu. If that something you add traps execution, say waiting for serial communication, the menu will not be responsive.

- **Show_credit() function**

This function shows my contact information. You can remove it.

```
void show_credit()
{
  // Display credits
  lcd.clear();
  lcd.print("Phi-menu PASSWD4");
  lcd.setCursor(0,1);
  lcd.print("Dr.Liu 03/10/11");
  wait_on_escape(2000);
  lcd.clear();
  lcd.print("http://liudr.");
  lcd.setCursor(0,1);
  lcd.print("wordpress.com");
  wait_on_escape(2000);
  lcd.clear();
}
```

II. FUNCTIONS.pde

This file contains all your project contents, including variables, objects, PROGMEM, #define commands, menu texts, the menu, and all menu functions. Here is a diagram depicting the flow of the code in the file:

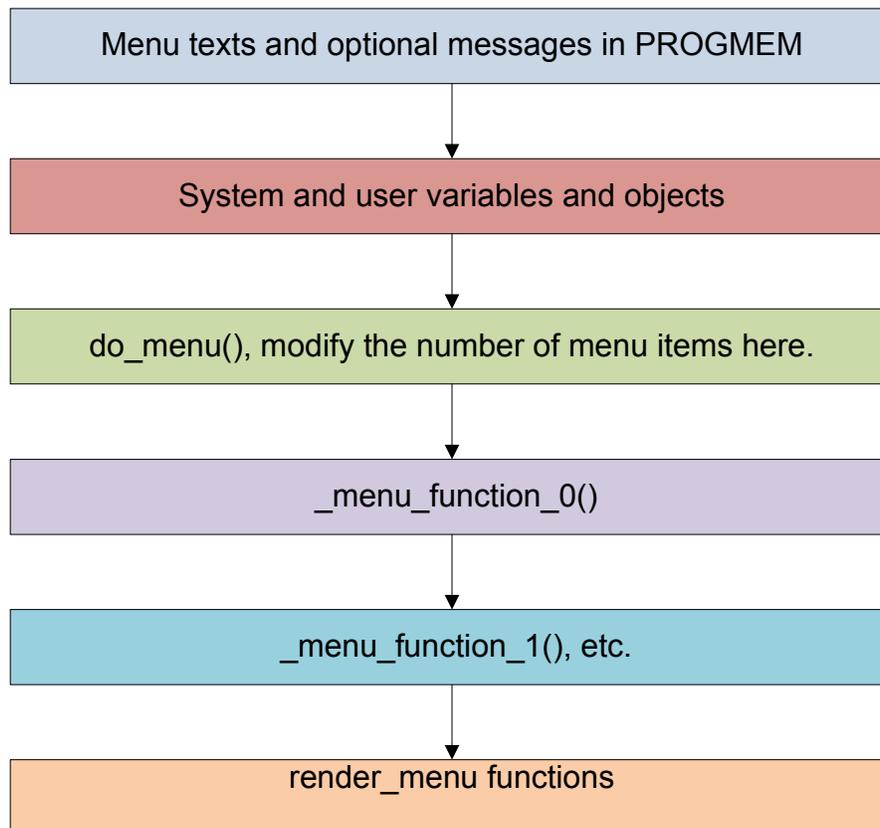


Figure 2-1: Structure of the FUNCTIONS.pde

The details of each part are as the following:

- **Menu PROGMEM definitions**

These are defined by you. You write your LCD menu texts here

Example:

This program has two menu items, display distance and parameters. You define menu_item00[] and menu_item01[], where you put the menu texts in the arrays. Then you list all the menu_item00, menu_item01 etc in the menu_item array so the items will be displayed in the menu and their corresponding functions will be called if a user confirms on the menu item.

```
PROGMEM prog_char menu_item00[]="Parameters";  
PROGMEM prog_char menu_item01[]="Display distance";  
PROGMEM const char *menu_item[] = {menu_item00, menu_item01};
```

- **User PROGMEM definitions**

These are optional definitions. You can store all your LCD messages here to save lots of room in SRAM. I have articles on my blog regarding PROGMEM:

<http://liudr.wordpress.com>

- **System variable definitions and object instantiations**

System variables and objects go here in the FUNCTIONS.pde. You may change some of their values but shouldn't remove any definitions.

```
//The following are related to the menu  
int menu_pointer=0;  
int phi_menu_style=phi_menu_style_classic;
```

Choices: phi_menu_style_classic, phi_menu_style_2X2, and phi_menu_style_index

- **User variable definitions and object instantiations**

You define your variables and instantiate your objects here in the FUNCTIONS.pde. This way you can debug more easily by not mixing your variables with the system code and system variables and your program stays organized.

Example:

```
//These are related to the specific program  
char pw[5]="4872";  
int _trials=3;
```

- **The menu**

void do_menu()

This program is the main menu. It gets called repeatedly by loop(). It handles user inputs and executes a certain menu function when the user selects it.

User modification: only a part of this function should be modified to handle your menu items (see below). The rest should stay unchanged.

You may change the content inside this switch. As many as ten menu functions are included in the phi-menu distribution package. You may just leave them there or add or remove *_menu_function_x* as you understand more about phi-menu.

```
switch (temp1)
{
  case 0:
    _menu_function_0();
    break;

  case 1:
    _menu_function_1();
    break;
  //There are 8 more of them.
}
```

- **Menu items**

_menu_function_0()

_menu_function_1()

...

Menu item functions go here. As many as ten menu functions are included in the phi-menu distribution package. If you don't need these many, leave them empty. Each function is called when its menu item is selected. They are called only once. This works pretty well if you just to do a series of things in a menu item such as adjusting a series of parameters:

```
void _menu_function_0()
{
  int temp1, temp2;

  delay(100);
  lcd.clear(); // Input on_divider
  lcd.print("On divider:");
  temp1=on_divider;
  temp2=hmi_with_update_3(&temp1, 10, 100, 1, 4, 1, 4, render_number_in_place);
  on_divider=(temp2==-1)?on_divider:temp1;
```

```

delay(100);
lcd.clear(); // Input off_divider
lcd.print("Off divider:");
temp1=off_divider;
temp2=hmi_with_update_3(&temp1, 4, 50, 1, 4, 1, 4, render_number_in_place);
off_divider=(temp2==1)?off_divider:temp1;

delay(100);
lcd.clear(); // Input audio_alert
lcd.print("Audio alert?");
temp1=audio_alert;
temp2=hmi_with_update_3(&temp1, 0, 1, 1, 1, 1, 3, render_YN_in_place);
audio_alert=(temp2==1)?audio_alert:temp1;
}

```

The content of the code was purposely obscured to show you that three things happen in sequence, each with delay of 0.1 seconds. There is no need to repeat these three things, such as setting values of three parameters.

If you want to do something repeatedly, say displaying some analog readings repeatedly, make sure you have a while loop:

```

void _menu_function_1()
{
  While(1)
  {
    //Do display stuff
  }
}

```

You also need a means of quitting these functions so the program returns to the menu. This is done by calling the following function:

```

void _menu_function_1()
{
  While(1)
  {
    //Do display stuff
    if (wait_on_escape(50)>0) return; // Return to menu if any key is pressed
  }
}

```

The `wait_on_escape()` is another utility function defined in `ALIUDRuser_interface.pde`

The function has the same effect as `delay()` except that it keeps checking the buttons while waiting and returns any buttons pressed or 0 for no buttons pressed during the waiting. See `wait_on_escape()` under `ALIUDRuser_interface.pde` for details.

Almost all sensor measurement functions need some sort of delay so replace your `delay()` with `wait_on_escape()` and you can control the flow of the control and returns to menu if

a specific key is pressed (`wait_on_escape(50)=1` for up button) or if any key is pressed (`wait_on_escape(50)>0`).

- **Menu renderers**

They display the menu in several styles.

void render_menu(int me)

Display the LCD menu item according to the argument [me]. This renders the classic menu as shown below:

User modification: you should not modify this.

1. Classic menu

One Item at a time



Figure 2-2: classic menu

To display the classic menu, have this line inside `do_menu()`. Notice the *render_menu*:
`temp1=hmi_with_update_3(&menu_pointer,0,sizeof(menu_item)/sizeof(char*)-1,1,16,1,0,render_menu);`

void render_menu_2X2(int me)

Display the LCD menu item according to the argument [me]. This renders the two by two menu as shown below. The right arrow indicates the item being highlighted. All menu items are being truncated to fit within 7 characters. You may have multiple pages and the program handles flawlessly:

User modification: you should not modify this.

2. Two by two menu

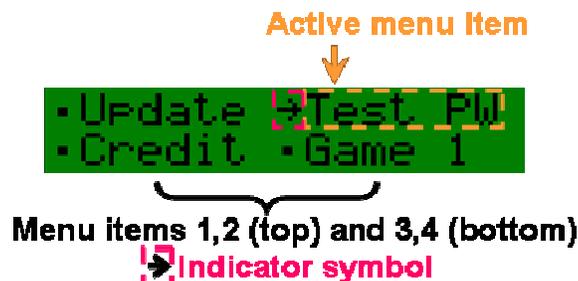


Figure 2-3: two by two menu

To display the two by two menu, have this in `do_menu()`. Notice the `render_menu_2X2`:

```
temp1=hmi_with_update_3(&menu_pointer,0,sizeof(menu_item)/sizeof(char*)-  
1,1,16,1,0,render_menu_2X2);
```

```
void render_menu_index(int me)
```

Display the LCD menu item according to the argument [me]. This renders the classic menu with a menu item index below the highlighted menu item, as shown below. The up symbol indicates the index number of the highlighted item:

User modification: you should not modify this.

3. Classic menu with index

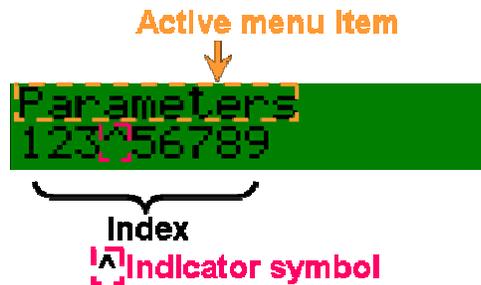


Figure 2-4: classic menu with index

To display the menu with index, have this in `do_menu()`. Notice the `render_menu_index`:

```
temp1=hmi_with_update_3(&menu_pointer,0,sizeof(menu_item)/sizeof(char*)-  
1,1,16,1,0,render_menu_index);
```

III. ALIUDRuser_interface.pde

This file stores all interface functions that support menu and other functions that need access to button status and render messages on the LCD. Please don't alter the content of this file.

There are two types of interactive inputs, constructed from the interaction and rendering functions described below.

Number and enumerate rendering functions render numbers and enumerates with texts. With `hmi` function, you can obtain input from users.

1. Number and enumerate Inputs



Numbers are bound:
2000-2100 for year and 1-31 for day



Enumerates are bound:
MON-SUN, JAN-DEC, YES-NO



Enumerates are customizable:
Weekday, Weekend, Everyday, Once, Off

Figure 3-1: Number and enumerate inputs

To receive a number input from users, use the following example code:

```
hmi_with_update_3(&para_val, 2000, 2100, 1, 5, 1, 4, render_number_in_place);
```

The above code asks the user to input a number between 2000 and 2100. The value is stored in para_val. The step size is 1. The function displays the number at (column,row) (5,1) and will erase 4 spaces in order to keep the number clean.

If you store the return value of the above function, you will know what button was pressed (left, right, confirm, or escape) and you can act accordingly.

To receive YES/NO response from users, use the following example code:

```
hmi_with_update_3(&para_val, 0, 1, 1, 5, 1, 3, render_YN_in_place);
```

The above code asks the user to input NO (0) or YES (1). The value is stored in para_val. The step size is 1, to go from NO to YES. The function displays the choice at (column,row) (5,1) and will erase 3 spaces in order to keep the choice clean.

With character rendering, input panel, and hmi functions, you can obtain texts from users.

2. Number and arbitrary text Inputs



Range of characters is customizable
Use **number only** for password



Use **full range** for **arbitrary** text input
Other ranges can be defined as well.

Figure 3-2: Input panel

To receive a number password (as string) from users, use the following example code:

```
input_panel3(msg, 6, 4, 1, '0', '9');
```

This function asks user input and stores it in string msg. The input is up to 6 characters, displayed at (column,row) (4,1). Valid inputs are between '0' and '9'.

To receive an arbitrary string input from users, use the following example code:

```
input_panel3(msg, 10, 0, 1, ' ', 'z');
```

This function asks user input and stores it in string msg. The input is up to 10 characters, displayed at (column,row) (0,1). Valid inputs are between ' ', space and 'z' lower case z.

- **Rendering functions**

They help render information on the LCD. They are used to support interaction functions.

```
void msg_lcd(char* msg_line)
```

Displays strings stored in PROGMEM. Provide the string name [msg_line] stored in PROGMEM to be displayed on the LCD's current cursor position.

User modification: you should not modify this.

```
void render_number_in_place(int number)
```

Displays a number on the LCD

User modification: you should not modify this.

```
void render_00number_in_place(int number)
```

Displays a two-digit number on the LCD

User modification: you should not modify this.

```
void render_char_in_place(int number)
```

Displays a character on the LCD

User modification: you should not modify this.

```
void render_YN_in_place(int number)
```

Displays "Yes" or "No" on the LCD

User modification: you should not modify this.

```
void render_xxx()
```

Displays information on the LCD

User modification: you should not modify this.

- **Interaction functions**

They facilitate button interactions.

int wait_on_escape(int ref_time)

Returns key pressed or 0 if time expires before any key was pressed. Use this function to wait for users to press a key. You may replace delay() with this function to regulate program flow while checking user input at the same time.

User modification: you should not modify this.

1	Up	2	Down	3	Left
4	Right	5	B	6	A
		0	No key		

Table 1. wait_on_escape() return values.

int hmi_with_update_3(int current, int lower, int upper, byte inc, byte column, byte row, byte space, void (*update_function)(int))*

Human machine interaction (HMI) with one update function and input value wrap-around capability.

This function prints the initial value first so the caller doesn't need to.

Function traps until the update is finalized by the left, right, enter button or escape button.

Value is updated through the *current pointer instead of return value so the return value is used to carry which functional button was last pushed. Also negative input values are allowed including -1.

Returns buttons pushed so the caller can determine what to do:

Go back to the last slot with left (-3)

Go forward to the next slot with right (-4)

Enter(1)

Escape(-1).

int input_panel3(char msg, byte length, byte column, byte row, char _start, char _end)*

Alphanumerical input panel for texts up to 16 characters.

The option starts with the character _start and ends with character _end

The total length will not exceed length.

The first letter appears at column, row

The function prints the content of the buffer msg before polling for input

The function returns number of actual characters

The function returns -1 if the input is cancelled.

The function fills the buffer after the last character with \0 only if the buffer is not filled.

The caller is responsible to fill the character beyond the end of the buffer with \0.

User modification: you should not modify this.

*[RETIRED] int hmi_with_update_2(int current, int lower, int upper, byte inc, byte column, byte row, byte space, void (*update_function)(int))*

HMI function with one update function and input value wrap-around capability. This version is older. If you want your user to input a value for a parameter, you use this function and it returns the user input. It handles user button pushes and intermediate values and only returns the final value the user decides on.

Function traps until the update is finalized by the confirm button (B) or forfeited by the esc button (A).

Returns updated value (zero or more) when enter is pressed. Returns -1 if escape is pressed.

You need to first display the current value on the lcd and then call this function to solicit for user input. If -1 is returned, you don't change the variable.

User modification: you should not modify this.

[RETIRED] int input_panel(char msg, byte length, byte column, byte row)*

Alphanumerical input panel for texts up to 16 characters.

The total length will not exceed length.

The first letter appears at column, row

The function prints the content of the buffer *msg before polling for input

The function returns number of actual characters

The function returns -1 if the input is cancelled.

The function fills the buffer after the last character with \0 only if the buffer is not filled.

The caller is responsible to fill the character beyond the end of the buffer with \0.

User modification: you should not modify this.

IV. phi_buttons.cpp and phi_buttons.h

This is the phi_buttons class. The various status of a button is modeled as the buttons class. Its details are covered elsewhere.

User modification: you should not modify this.

Besides real buttons, physical buttons and button-like inputs such as photogates and Hall switches, there is another type of buttons, virtual buttons. Currently, buttons class support two kinds of virtual buttons.

1 - Null buttons

Null buttons are used where you want to disable say the left button. You can declare it as null in the button initializing. The use of this is to avoid having to remove or revise phi-menu functions in the event that several buttons are not needed and their pins are used for other purposes. Say you don't need left or right buttons and use their Arduino pins for LEDs, then without the null buttons, you will remove the object instantiation lines, remove sensing these buttons inside the phi-menu system. It's a mess. Then you realize that you need them back so you go and add them back everywhere you removed them. Now with null buttons, you just assign btn_null as their pins and that button is treated as null button and sensing it always gives buttons_up status.

2 - Auto buttons

This button is automatically pressed after every period of time, say 1.25 seconds. The button is not associated with any physical Arduino pin so you can free up a pin. The use of auto button can be that you are in a crunch for more pins and have to sacrifice buttons to save pins. What can you do with less buttons? Let them be pressed automatically. Say,

in the following video, I only needed ONE button to operate the whole password input panel that I used to use all 6 buttons to operate!!!

Well it's a waste of time if you want to input something very long but for prescribed things like day of the week or else, you can certainly waste some time to save Arduino pins. You still need to sense the buttons to make the auto buttons work.

Video: ONE button operates the entire password input panel:

V. defs.h

This contains all the system definitions such as “#define buzzer 16”. The main .pde file includes this file by adding #include “defs.h” to the beginning of the program. Please don't modify the content of this file. Your #define goes in the FUNCTIONS.pde

There is only one type of definitions in this version:

- **Shield definitions**

These definitions lay the ground work of what shield function is using which Arduino pins so later versions of Phi shields will only need to change this section without the need to change user-written codes.

User modification: you should not modify this.

The following applies to the Phi-1 shield or Phi-2 shield automatically. Refer to shield selector section under Phi_menu_XXX.pde on how to select your shield:

```
ifndef phi_1_shield
//Phi-1 shield definitions

//Phi-1 shield buttons and channel pin assignments
#define buzzer 16
#define ch1 16
#define ch1_analog 2
#define led 17
#define ch2 17
#define ch2_analog 3
#define btn_u 5
#define btn_d 10
#define btn_l 11
#define btn_r 3
#define btn_b 14
#define btn_a 15

//Phi-1 shield LCD pin setting
#define LCD_RS 8
#define LCD_EN 9
#define LCD_D4 7
#define LCD_D5 6
#define LCD_D6 2
```

```

#define LCD_D7 4

#endif

#ifdef phi_2_shield
//Phi-2 shield definitions

//Phi-2 shield buttons and channel pin assignments
#define buzzer 16
#define btn_u 5
#define btn_d 10
#define btn_l 11
#define btn_r 4
#define btn_b 14
#define btn_a 15

//Phi-2 shield LCD pin setting
#define LCD_RS 8
#define LCD_EN 9
#define LCD_D4 7
#define LCD_D5 6
#define LCD_D6 2
#define LCD_D7 3

#endif

//Maximal bytes on your Phi-1 shield eeprom, depending on the chip you use. I'm using 24LC256, with
256KiloBits so 32KiloBytes=32768 bytes.
#define EEPROM_size 32768UL

//#define setRTC // Uncomment this to set the clock time.

```

VI. progmem.h

This file contains all the PROGMEM definitions. Using PROGMEM can save significant SRAM. A different article discusses the details of using PROGMEM.

- **LCD custom character PROGMEM definitions**

These are defined by users like you. If you want an hour glass symbol or a “/s” symbol in one character to save space, you can define as many as 8 characters. Their ASCII codes are 0-7. You will need to use lcd.write() to put the zeroth character on the display. A different article discusses in details how to construct custom characters.

User modification: you may modify this.

```

//Programming custom characters. The custom characters will be lost when power is cut.
//In the character definition, you have to use B100000 instead of B0 since you will need to do strcpy_P,
which terminates at 0 or B0.
PROGMEM prog_char lcd_ch0[]={B110,B100,B100,B110,B100000,B1010,B10101,B10101,0};
PROGMEM prog_char lcd_ch1[]={B1110,B11011,B1010,B11011,B1010,B11011,B1010,B1110,0};

```

```

PROGMEM prog_char lcd_ch2[]={B10001,B1110,B11110,B11101,B11011,B10111,B100000,B11111,0};
PROGMEM prog_char lcd_ch3[]={B100000,B11101,B11011,B11101,B11110,B1110,B10001,B11111,0};
PROGMEM prog_char lcd_ch4[]={B11101,B11001,B10101,B1101,B100000,B11101,B11101,B11111,0};
PROGMEM prog_char lcd_ch5[]={B10,B100,B1000,B10011,B100,B10,B1,B1110,0};
PROGMEM prog_char lcd_ch6[]={B100000,B100000,B100,B1110,B11111,B100,B100,B100000,0};
PROGMEM prog_char lcd_ch7[]={B100000,B100000,B100,B100,B11111,B1110,B100,B100000,0};
PROGMEM const char *ch_item[] = {lcd_ch0,lcd_ch1,lcd_ch2,lcd_ch3,lcd_ch4,lcd_ch5,lcd_ch6,lcd_ch7};

```

- **Other common keywords PROGMEM definitions**

These are commonly used keywords for RTC and other devices and can be commented out to save space.

User modification: you should not modify this.

```

PROGMEM prog_char dow_00[]="SUN";
PROGMEM prog_char dow_01[]="MON";
PROGMEM prog_char dow_02[]="TUE";
PROGMEM prog_char dow_03[]="WED";
PROGMEM prog_char dow_04[]="THU";
PROGMEM prog_char dow_05[]="FRI";
PROGMEM prog_char dow_06[]="SAT";
PROGMEM prog_char dow_07[]="M-F";
PROGMEM prog_char dow_08[]="Day of the week";

```

3. An example

The following is an example code for a simple sonic ranger for distance measurement and obstacle avoidance when you back your car. It demonstrates how easy it is to construct a project with the phi-menu. This is a featured project on instructables.com

We want to have an audio alert of the distance of the object so the driver doesn't have to look at the LCD all the time. Then we also want the option to adjust how often the buzzer beeps and to turn the audio alert on and off.

First we select the correct shield in the main program at line 20 or 21. We assume phi-1 shield is used:

```

#define phi_1_shield
//#define phi_2_shield

```

We need to modify menu texts in the PROGMEM area after the highlighted comment:

```

// Messages used in program.
PROGMEM prog_char menu_item00[]="Parameters ";
PROGMEM prog_char menu_item01[]="Display distance";
PROGMEM const char *menu_item[] = {menu_item00, menu_item01};

```

Next, we create variables to store our parameters after the highlighted comment line:

```
//These are related to the specific program  
boolean audio_alert=true; // This indicates whether audio beeps are on or off.  
int on_divider=40; // The larger this value, the shorter each beep is.  
int off_divider=10; // The larger this value, the less frequent the beeps occur.
```

Next, we modify the do_menu to make sure there are **two menu items**:
The grayed areas are not supposed to be changed.

```
void do_menu()  
{  
  //Do not change this  
  int temp1;  
  temp1=hmi_with_update_2(menu_pointer,0,sizeof(menu_item)/sizeof(char*)-1,1,16,1,0,render_menu);  
  menu_pointer=(temp1!=-1)?menu_pointer:temp1; // In case escape was triggered, value doesn't change.  
  switch (temp1)  
  {  
    case 0:  
      _menu_function_0();  
      break;  
  
    case 1:  
      _menu_function_1();  
      break;  
  
    default:  
      break;  
  }  
}
```

Next, we write the “parameters” function, which asks users to change parameters we just defined in the previous step. From the name of the function we know it is the first item in the menu:

```
void _menu_function_0 ()// Parameters  
{  
  int para_val, btn_pressed;  
  
  lcd.clear(); // Input on_divider  
  lcd.print("On divider:");  
  para_val =on_divider;  
  btn_pressed =hmi_with_update_3(&para_val, 10, 100, 1, 4, 1, 4, render_number_in_place);  
  on_divider=( btn_pressed ==-1)?on_divider: para_val; // In case of escape,, value doesn't change.  
  
  delay(100);  
  lcd.clear(); // Input off_divider  
  lcd.print("Off divider:");  
  para_val=off_divider;  
  btn_pressed=hmi_with_update_3(&para_val, 4, 50, 1, 4, 1, 4, render_number_in_place);  
  off_divider=(btn_pressed== -1)?off_divider:para_val; // In case of escape, value doesn't change.
```

```

delay(100);
lcd.clear(); // Input audio_alert
lcd.print("Audio alert?");
para_val=audio_alert;
btn_pressed=hmi_with_update_3(&para_val, 0, 1, 1, 1, 1, 3, render_YN_in_place);
audio_alert=(btn_pressed!=-1)?audio_alert:para_val; // In case of escape, value doesn't change.
}

```

The above function has three blocks of code, each one updating one parameter. The sequence of code used in each block is the following:

- Clear LCD
- Print message
- Assign the parameter's current value to para_val
- hmi_with_update_3 asks the user to update the para_val. The function returns button pressed in btn_pressed.
- If btn_pressed is a cancel button (A), the parameter value is not changed. Otherwise it's changed to the new value stored in para_val.

Then we write the code to sense and display distance. This should be called repeatedly inside of another menu function: menu_function_1(). Here is its content:

```

void _menu_function_1() //Displays sonic ranger result
{
  while (!sonic_ranger(ch2))
  {}; // Keeps running sonic ranger until user presses a key to escape
}

```

The sonic_ranger() function is the work horse. Inside the function, it reads distance by calling sonic_ranger_read() and then calls ranger_display_distance() to display it. Neither called functions are included below. They won't help you unless you also want a sonic ranger project. But some contents in the sonic_ranger function are common features you need in your project:

- Parameters that you defined in the first steps are used in this program to affect how the program runs, such as audio_alert affects whether buzzer turns on.
- You also need delay using wait_on_escape. This prevents Arduino from busy working on its measurement and forgetting to check user inputs. The **wait_on_escape** handles any key pressed while waiting and returns the key pressed. This also gives us a way to terminate the function and return to the menu, in case a key is pressed while waiting.

```

long sonic_ranger_read(int gate_pin) // Reads and returns the time pulse.
void ranger_display_distance(long duration)
int sonic_ranger(int gate_pin) // Reads and displays distance with audio alert. It returns 0 normally and -1 when user presses a key.
{
  long duration, time;
  float mm;
  time=millis();
  if (audio_alert) tone(buzzer,440);
}

```

```

duration=sonic_ranger_read(gate_pin);
mm = (float)(duration) / 5.80;
ranger_display_distance(duration);
if (wait_on_escape(int(mm/on_divider))>0) // Wait and then turn off buzzer and return to main menu with
key press
{
  noTone(buzzer);
  return -1;
}

noTone(buzzer); // After wait, turn off buzzer

if (wait_on_escape(int(mm/off_divider))>0) // Delay after buzzer is off and before next iteration.
{
  return -1;
}

return 0;
}

```

Now compile and upload! You can see more of this project and program on my blog:
<http://liudr.wordpress.com/2011/02/12/arduino-parking-sensor/>

You can download the code and give it a try!

4. Future improvements

I plan to improve the phi-menu in the following areas:

- Turn the codes into class and encapsulate all variables and functions
- Add multiline message stroll display function support to display longer messages on a 16*2 or larger character display.
- Port the code to work on KS0108-compatible 128*64 dot matrix displays

5. The legal stuff

Let's keep this short. The software is defined as any part of the complete collection of the code of the phi-menu system. There is no explicit or implicit warrantee that the software will work. You are free to modify and redistribute the software as long as the credit section is kept. You can add your own credit and modify the summary of the functions.

You are not allowed to use the software for commercial purposes without a license from the author, which include but not limited to selling the software or its modified versions for money directly or selling hardware with the software attached for "free". The software is meant to be shared among interested individuals and educational institutions.

If you like to use the software with your products, commercial licenses can be obtained by contacting the author at <http://liudr.wordpress.com>