REBL\_UI

The REBL\_UI offers a simple way to build a user interface with a **R**otary **E**ncoder, a **B**utton, and an **L**CD. Currently it is set up for 16x2 LCD, but could easily be changed to whatever. It includes a very simple and easy to set up menu system and several functions to facilitate using the encoder and button with minimal code on the user’s part. Debouncing and edge detection are handled by the class.

There are several headers in the REBL\_UI folder that handle different parts of the interface. REBLInterface has the code for the button and encoder. REBLDisplay has code to handle the display. The user code never touches the LCD, it only interacts with this code. So changing an LCD is as simple as rewriting a little code in this one class. There is a copy of my MenuClass and an implementation of it called REBLMenu. There is also a header for some useful functions in building a UI.

**REBLDisplay.h**

This header includes code to interact with the display. Currently it allows you to print to buffers that it will display to the screen, control the backlight, and control the cursor. The current implementation is for an LCD with an RGB backlight with a 595 shift register controlling the data pins and a library I wrote that controls the whole thing using the SPI interface. LiquidCrystal\_SPI\_8Bit.h (at my github page). The code will easily change to support the normal LiquidCrystal library by simply changing the include and the initLCD function to call the right begin method.

void **initLCD**(byte aRsPin, byte aEnablePin, byte aRedPin, byte aGreenPin, byte aBluePin);

Sets up the pins and calls begin for the LCD. Call this function from setup before trying to do anything with the display. See LiquidCrystal\_SPI\_8Bit library documentation for description of RS pin and Enable pin. The others are for the backlight. If you have a monochromatic LCD, then just set them all to the same pin you used for the backlight and use any color to turn the LCD on and BLACK to turn it off.

void **setColor**(color\_vars color);

Sets the color of the backlight to a selection from the color\_vars enum. The choices are: {BLACK, RED, GREEN, YELLOW, BLUE, PURPLE, CYAN, WHITE}.

void **doDisplay**();

Causes the contents of the display buffers to be written out to the LCD. There should be some delay between consecutive calls or the screen will flash. It is generally called from loop with a “Blink Without Delay” style timer.

void **displayLineLeft**(int row, const char\* message);

void **displayLineRight**(int row, const char\* message);

These functions put text into the display buffers to be displayed on the screen in left or right justified manner. These functions will fill any unused space in the buffer with spaces and the functions will truncate any strings too long to fit. The functions are overloaded to take \_\_FlashStringHelper\* so they’re F macro friendly. These are the functions you’ll mostly use from your code to write to the screen.

void **displayTextLeft**(int row, int startcol, int colwidth, const char\* message);

void **displayTextRight**(int row, int startcol, int colwidth, const char\* message);

These functions also put text into the display buffers, but only operate on a piece of the buffer instead of writing the entire line. They will always fill at least colwidth characters and will use spaces if the text isn’t long enough in left or right justified manner but will leave other text in the buffer unchanged. These functions are also overloaded with \_\_FlashStringHelper\* versions so they are F macro friendly.

void **setCursor**(byte row, byte column);

void **showCursor**(boolean cursorOn);

These allow you to control the cursor from your code without touching the LCD code.

Hidden Secret: There is a define in the REBLDisplay.h file that will allow you to send the LCD output to the serial monitor instead along with info about the cursor and screen color. It makes for easy debugging when the gizmo with the LCD is not close to the computer.

See the example code for some examples of using these functions to control the display.

**REBLInterface.h**

This header contains code to interact with the button and rotary encoder as well as some convenience functions for doing user interface type things with them. Both the encoder and button have an on/off setting and when turned off will not register presses or turns. This is handy to avoid the first update to a menu scrolling a thousand lines down and selecting something random because you turned the encoder a bunch before you entered the menu. When off, the button will always return unpressed and the encoder will always return zero change.

void **initInterface**(byte buttonPin, byte encoderInterruptPin, byte encoderBPin);

Sets up the pins and starts the interface working. Should be called once from setup before interacting with the button or encoder. Does NOT turn the button or encoder on, those functions must also be called before using. Currently, the encoderInterruptPin can be any pin on the Arduino that supports hardware interrupts (pins 2 and 3 on UNO). It will not work with pin change interrupts without some changes to the code.

void **buttonOn**(); void **buttonOff**(); void **encoderOn**(); void **encoderOff**();

Turn on and off the reading of the button and encoder.

boolean **isEncoderOn**(); boolean **isButtonOn**();

Return true is the device is active.

char **checkRotaryEncoder**();

Returns the change in position of the encoder since the last time it was read. Negative numbers one way and positive numbers the other.

void **useRotaryEncoder**(T& var, int multiplier = 1);

Updates the variable var using the rotary encoder. If the multiplier is set to 0, then it only increments or decrements var by one no matter how far the encoder turned. If the multiplier is set higher, then it does the obvious and multiplies the change before applying it. The variable is passed by reference, so the real variable from the calling code will be updated.

void **useRotaryEncoder**(T& var, T minimum, T maximum, int multiplier = 1);

Same as above but includes limit checking.

void **useRotaryEncoderASCII**(char& var);

Similar to useRotaryEncoder but gets values from a table of asci values instead of numbers. Handy for inputting strings.

boolean **checkButton**();

Return true if the button has been pressed and released. Clears the flag so subsequent calls will return false until the button has been pressed again. If you need to use it more than once, read it into a variable and use that.

boolean **checkButtonLongPress**();

Return true if the button has been pressed and held for a defined time. Similar caveat to checkButton.

**REBLMenu.h**

REBLMenu is an implementation of my MenuClass which is specialized for working with the REBL\_UI. Menu items are created as structs containing display text and a function pointer. The functions called by the menu should take no arguments and return boolean. The boolean is to allow for the code in these functions to be non-blocking and allows these functions to be called repeatedly from loop via the doMenu() method. The functions should return false if they need to be called again and true if they are ready to return control to the menu. Small functions or intentionally blocking functions should simply return true. Please see the example code for a better idea.

Menu items are put into MenuList objects which represent the individual menus. The currently active menu is selected through the setMenu method. MenuLists should be placed in PROGMEM. There is a #define in the source for MenuClass that will allow you to have the items in SRAM, but this is best avoided except in test cases.

void **doMenu**();

This method should be called frequently from a tight loop while the menu is active. This method advances the menu, handles selection, handles display, handles calling any functions that are selected from the menu.

boolean **isCancelled**();

In the REBL\_UI, a long press (greater than 2 sec) will register the cancel flag and allow you to write code to cancel out of processes or functions.

void **setCurrentMenu**(MenuList\* aList);

This method sets the currently active menu.

**Rebl\_Functions:**

This header contains some pre-canned useful functions for doing things like entering strings and setting the time. Expect additions as I find need for functions in my code. Please see the source and example code for their use.

Please see the following code for an example using this interface. This example creates a simple menu that shows off each function of the library. This also shows an example of how to write non-blocking functions for use with the menu. Blocking functions could simply return true at the end.

#include "REBL\_UI.h"

#include <LiquidCrystal\_SPI\_8Bit.h>

#include <Time.h>

#define ENCODER\_INTERRUPT\_PIN 2

#define ENCODER\_B\_PIN 4

#define BUTTON\_PIN 8

#define LCD\_ENABLE\_PIN A1

#define LCD\_RS\_PIN A3

#define HEART\_BEAT\_PIN A5

#define LCD\_BLUE A0

#define LCD\_GREEN A2

#define LCD\_RED A4

MenuItem PROGMEM menuItems[] = {

{ "display", fun1 } ,

{"press", pressToExit } ,

{"enter number", pickNumber } ,

{"enter string", fourLetterWord } ,

{"set time", enterTime } ,

{"show time", showTime }

};

MenuList menuList(menuItems, menuListSize(menuItems));

void setup() {

initLCD(LCD\_RS\_PIN, LCD\_ENABLE\_PIN, LCD\_RED, LCD\_GREEN, LCD\_BLUE);

initInterface(BUTTON\_PIN, ENCODER\_INTERRUPT\_PIN, ENCODER\_B\_PIN);

reblMenu.setCurrentMenu(&menuList);

}

void loop() {

pollButton();

static unsigned long preMil = millis();

unsigned long curMil = millis();

if (curMil - preMil >= DISPLAY\_DELAY) {

preMil = curMil;

reblMenu.doMenu();

doDisplay();

}

}

boolean fun1() {

displayLineLeft(0, F("Display This"));

displayLineLeft(1, F(" ")); // to clear the line

return displayDelay(3);

}

boolean pressToExit() {

displayLineLeft(0, F("Press Button"));

displayLineLeft(1, F("to Exit"));

if(!isButtonOn()) {

buttonOn();

}

else if(checkButton()) {

buttonOff();

return true;

}

return false;

}

boolean pickNumber() {

static int state = 0;

static int val = 0;

switch (state) {

case 0:

{

val = 0;

state++;

}

case 1:

{

if (enterNumber(val, F("Enter Number"), 0, 50, 1)) {

state++;

}

break;

}

case 2:

{

if (displayDelay(3)) {

state = 0;

return true;

}

displayLineLeft(0, "You Entered:");

char buf[6];

sprintf\_P(buf, PSTR("%d"), val);

displayLineLeft(1, buf);

break;

}

} // end switch (state)

if(reblMenu.isCancelled()){

state = 0;

val = 0;

return true;

}

return false;

}

boolean fourLetterWord() {

static char theStr[5] = "four";

static int state = 0;

switch (state) {

case 0:

{ // enter the string

if (enterString(theStr, 4)) {

state++;

break;

}

theStr[4] = 0; // just in case.

displayLineLeft(0, F("4 Letter Word"));

displayLineLeft(1, theStr);

break;

}

case 1:

{ // display the entered string

displayLineLeft(0, F("You Entered"));

displayLineLeft(1, theStr);

if(displayDelay(3)) {

state = 0;

encoderOff();

buttonOff();

return true;

}

break;

}

} // end switch (state)

if(reblMenu.isCancelled()){

state = 0;

return true;

}

return false;

}

boolean enterTime() {

static time\_t timeSetTime = 0;

static int state = 0;

switch (state) {

case 0:

{ // turn things on and setup variables

timeSetTime = now();

state++;

}

/\* no break \*/

case 1:

{ // input the time

if (inputTime(timeSetTime)) {

state++;

setTime(timeSetTime);

}

break;

}

case 2:

{ // display the running time for 10 seconds

displayTime(now());

if (displayDelay(10)) {

state = 0;

return true;

}

break;

}

} // end switch (state)

if(reblMenu.isCancelled()){

state = 0;

timeSetTime = 0;

return true;

}

return false;

}

boolean showTime() {

displayTime(now()); // display the running time

if(reblMenu.isCancelled()){

return true;

}

return displayDelay(10); // do it for ten seconds

}