USING THE LEFT COAST LIBRARIES

Basic tools & packages

LC_baseTools

https://github.com/leftCoast/LC_baseTools

timeObj

https://github.com/leftCoast/LC_baseTools/blob/master/timeObj.h

timeObj is a timer object can be set to a time (in ms) and polled as to when the time has expired. It can take long times or times less than a ms, because it's input is a float. The timer will automatically run in mili seconds or micro seconds, whatever works best for the inputted time.

Internally the timer has three states. preStart, running, expired. You can change states with start(), stepTime() and reset().

Constructors

```
timeObj(float inMs=10, bool startNow=true);
```

Public methods

```
void setTime(float inMs, bool startNow=true);
```
Sets the time for the next running.

```
void start(void);
```
Starts or restarts the timer.

```
void stepTime(void);
```
Restarts the timer calculated from the last start. Eliminating error.

```
bool ding(void);
```
Returns true if the state of the timer is "expired". Does not change the state of the timer.

```
float getTime(void);
```
In case you forgot what you set the time to, this'll return the value to you.

```
float getFraction(void);
```

This returns a value 1..0 representing the fraction of time left. Like a gas gauge?

```
void reset(void);
```
Resets the timer back to preStart state.

mapper

https://github.com/leftCoast/LC_baseTools/blob/master/mapper.h

Mappers (linear mappers) Map one set of values to another. For example 0..5V maps to 3.6..1.2 psi kinda' thing. Arduino supplies an integer one. The Left coast one uses float values. Much more flexible. When asked to map a value out of range, say larger, it returns the max mappable value. If less than the mappable value it will return the least mappable value. In other words, it limits it's values for you.

Also, since we had the data. slope, min max & integration was added.

Constructors

```
mapper(void);
Default mapper constructor. Will map values 0..1 to 0..1.

mapper(double x1,double x2,double y1,double y2);
Constructor including the input end points, x1 & X2 and the mapped endpoints
y1 & y2
```

Public methods

```
double map(double inNum);
Maps the inputted value to its calculated mapped value.

void setValues(double x1,double x2,doubley1,double y2);
Same as the usual constructor.

double  getSlope(void);
Returns the slope of the resultant mapped line.

double  getMinX(void);
Returns the minimum value end of the mapped line

double  getMaxX(void);
Returns the maximum value of the mapped line.

Returns the value of x where the line crosses the y axis.
double  getIntercept(void);

double  integrate(double x1,double x2);
Returns the calculation of the area under the line segment from x1 to x2 to
the y axis. (Used internally. disregard?)


double  integrate(void);
```

Returns the calculation of the area under the line to the y axis. This one is for pubic consumption.

multiMap

https://github.com/leftCoast/LC_baseTools/blob/master/multiMap.h

Multi map is used like a mapper in than it has a map(value) method. But it can be a non-linear, curve fitting mapper. The way it's setup is a little different in that you put in a list of ordered pairs that follow the curve in question.

Constructors

```
multiMap(void);
Constructor returns an empty map.
```

Public methods

```
void addPoint(double x, double y);
Add point is how you add points to the mapper. It keeps track of the max & in
for you. Also sorts the points as well so you don't need to worry about order
of points.

void clearMap(void);
Dumps out all the added points and recycles their memory. Now you can add more
if you like.

double map(double inVal);
like Just like the linear mapper drop a value and it will return the mapped
value.

double integrate(double x1, double x2);
This returns the integration of the curve you plotted into this.
```

runningAvg

https://github.com/leftCoast/LC_baseTools/blob/master/runningAvg.h

runningAve sets up a running average function. You set it up with the number of datapoints you would like to use for your running average. Then, for each data point you enter, it discards the oldest data point and returns the average of this new point along with the rest of the saved datapoints.

Well, that's how it all started, and was for a long time. Then people got interested in it and suddenly it got a bunch more, features. Like standard deviation. And optionally setting upper and lower limits for input filtering of outliers.

NOTE : What if it doesn't have all the data points? It averages what it has.

Constructors

```
runningAvg(int inNumData);
Constructor, give it the number of data points it should keep an average of.
```

Public methods

```
float addData(float inData);
Drop in value, receive average.


float getAve(void);
Returns the current average of the data points it has.

float getMax(void);
Returns the maximum value data point.

float getMin(void);
Returns the minimum value data point.

float getDelta(void);
Returns the of all the data points.

float getEndpointDelta(void);
Returns the delta latest - oldest data points. Only those two points.

float getStdDev(void);
Returns the standard deviation of all the data points.

int getNumValues(void);
```

Returns the number of data values currently stored in the object.

`float getDataItem(int index);`
Returns the data value at that index. If there is no data value at that index, or the index is out of bounds, zero is returned.

`void setUpperLimit(float limit);`
Sets an upper limit for filtering outliers from input data.

`void clearUpperLimit(void);`
Stops filtering upper limits for input data upper limits.

`void setLowerLimit(float limit);`
Sets a lower limit for filtering outliers from input data.

`void clearLowerLimit(void);`
Stops filtering lower limits for input data lower limits.

`void setLimits(float lowerLimit, float upperLimit);`
Sets an upper & lower limits for filtering outliers from input data.

`void clearLimits(void);`
Stops all filtering of input data.

strTools

https://github.com/leftCoast/LC_baseTools/blob/master/strTools.h

strTools a grab bag of c string things I either got tired of typing or tired of looking for.

Function calls

```
void upCase(char* inStr);
Pass in a string and this makes all the letters uppercase.

void lwrCase(char* inStr);
Pass in a string and this makes all the letters lowercase.

bool heapStr(char** resultStr,const char* inStr);
Allocates and makes a copy of inStr.

void freeStr(char** resultStr);
Recycles resultStr.
```

NOTE : resultStr MUST be initialized to NULL to begin with. After that? It can be reallocated as many times as you like using heapStr().

tempStr

tempStr works like heapStr in that it will allocate and create a copy of it's inputted c string. In this case there is no al local string to start as NULL and recycle with freeStr() like in above. You just toss in strings with either the constructor or with setStr(). When this object goes out of scope it auto deletes the internal string for you.

Constructors

```
Can be constructed with an empty or non-empty string.
tempStr(const char* inStr=NULL);
```

Public methods

```
Recycles the current string and sets up this new string.
void setStr(const char* inStr);

Returns the number of chars contained in this string. Just in case you needed
to know.
int numChars(void);

Passed back the actual string being held in the object.
const char*  getStr(void);
```

colorObj

https://github.com/leftCoast/LC_baseTools/blob/master/colorObj.h

colorObj gives you a common class for passing colors about from screens to neoPixles to .bmp files, what have you. Included with this is the ability to blend colors. Map values to colors (color mappers, like value mappers). Ability to swap between 24 bit RGB colors to 16 bit based colors. (Used for most FTF & OLED displays. Common predefined colors are included to be used as base and blended in colors. There is also a compact 24 bit color struct used for memory based bitmaps and passing colors to and from .bmp files.

Public Definitions

```
Predefined colors for start points and mixing colors.

colorObj red;
colorObj blue;
colorObj white;
colorObj black;
colorObj green;
colorObj cyan;
colorObj magenta;
colorObj yellow;


Predefined settings for colors.

//                      Red,Grn,blu
#define LC_BLACK         0,  0,  0
#define LC_CHARCOAL      50, 50, 50
#define LC_DARK_GREY    140,140,140
#define LC_GREY         185,185,185
#define LC_LIGHT_GREY   250,250,250
#define LC_WHITE        255,255,255

#define LC_RED          255,  0,  0
#define LC_PINK         255,130,208

#define LC_GREEN          0,255,  0
#define LC_DARK_GREEN     0, 30,  0
#define LC_OLIVE         30, 30,  1

#define LC_BLUE           0,  0,255
#define LC_LIGHT_BLUE   164,205,255
#define LC_NAVY           0,  0, 30
```

```
#define LC_PURPLE       140,  0,255
#define LC_LAVENDER     218,151,255
#define LC_ORANGE       255,128,  0

#define LC_CYAN           0,255,255
#define LC_MAGENTA      255,  0,255
#define LC_YELLOW       255,255,  0
```

Constructors

```
colorObj(RGBpack* buff);
```
Create a colorObj from a RGBpack.

```
colorObj(byte inRed, byte inGreen, byte inBlue);
```
Create a colorObj from red green and blue values.

```
//colorObj(colorObj* inColor);
// Wanted this one, but the compiler mixes it up with color16.
```

```
colorObj(word color16);
```
Creates a best match colorObj from a 16 bit color value.

```
colorObj(void);
```
Creates a colorObj that is black;

Public methods

```
void setColor(RGBpack* buff);
```
Set this color from a RGBPack.

```
void setColor(byte inRed, byte inGreen, byte inBlue);
```
Set this color by RGB values.

```
void setColor(word color16);
```
Set this color by best match of a 16 bit color value.

```
void setColor(colorObj* inColor);
```
Set this color by copying another color.

```
word getColor16(void);
```
Return a 16 bit color value from a colorObj.

```
byte getGreyscale(void);
```
Returns a greyscale version of this colorObj.

```
byte getRed(void);
```
Returns the red value of this colorObj.

```
byte getGreen(void);
Returns the green value of this colorObj.

byte getBlue(void);
Returns the blue value of this colorObj.

RGBpack packColor(void);
Returns a color pack from this colorObj.

colorObj mixColors(colorObj* mixinColor,byte mixPercent);
Create a new color by mixing a new color with this colorObj. (I never use this
one.)

void blend(colorObj* mixinColor,byte mixPercent);
Blend 0% -> 100% of new color into this colorObj. (Used ALL the time.)
```

colorMapper

Then there are the color mappers. Fist is the linear mapper. It maps 0..100 percent. Returning a color blended from start color to end color.

Constructors

```
colorMapper(void);
Creates a mapper that just outputs black.

colorMapper(colorObj* inStart, colorObj* inEnd);
Creates a mapper that blends from start colorObj to end colorObj.

colorMapper(word startC16,word endC16);
Creates a mapper that blends from start 16 bit color value to end 16 bit color
value.
```

Public methods

```
void setColors(colorObj* inStart, colorObj* inEnd);
Sets up a new set of colors in this colorMapper.

colorObj map(float percent);
Retunes a mapped color for every value of 0..100 entered.
```

colorMultiMap

And then there is the non-linear color mapper. This one is so incredibly useful! Imagine a battery level barograph. Use this color mapper to map the power values to the color of the barograph. Or fuel gauge, or drawing a tachometer?

Constructors

```
colorMultiMap(void);
Returns a blank multi color mapper.
```

Public methods

```
void addColor(double inX, colorObj* color);
At this numeric value we resolve to this color.

void clearMap(void);
Remove and recycle all the color mapping values.

colorObj map(double inVal);
Returns the color that maps to this value.
```

NOTE: Between value, color pairs the color is blended. Starting color at the previous pair and ending at the next pair.

idler

https://github.com/leftCoast/LC_baseTools/blob/master/idlers.h

Now that the simple stuff has been dealt with, it's time to kick it up a notch.

Idler is a base class for objects that can run in the background. (Magic) And this changes ***EVERYTHING*** about how to code a project. The library comes with a idle() function, this is typically is placed as the first line of your global loop() routine. Each object that is to become an idler needs to have it's hookup() method called to activate it. Some do this automatically, some can't.

To use?

To use any predefined or user defined idlers all you need to do is add the function idle() called as the first line of your global loop() routine.

NOTE : Once using idlers, do NOT call delay()! But don't fear. Idlers give you a sleep() function that gives you the same pause effect of your loop(). But, doesn't stop the idlers from running in the background.

If you would like to create your own idler class, just inherit idler and override the idle method for your own action during "idle time".

```
#include <idlers'h>

Class myBackgroundClass : public idler {
      public:
            bla..
            bla..
       void idle(void);
            bla..
};
```

NOTE : If there is ANY chance that your idler class may be created as a global, IE before your call to setup(), Do NOT put the hookup call in your constructor. Why? Because before setup() is called, there is no control over

the order of globals being created. Idlers rely on their support code being complete before they are able to hook into it.

blinker

https://github.com/leftCoast/LC_baseTools/blob/master/blinker.h

You need to blink an LED without blocking? Here's your ticket. Blinker is an idler that runs in the background. All you need to do is turn it on. By default it will blink the Arduino's LED in the background. Heck you could setup 10 of them at different rates on different pins and they'd all blink, at those rates, in the background, without blocking. Magic!

```
#include <blinker.h>

blinker     myBlinker(pinNum,pulseMs,periodMs);

void setup() {
      myBlinker.setOnOff(true);
}

Void loop() {
      idle();
}
```

NOTE: Any program that uses an idler must have idle() called in loop().

Public Definitions

```
#define defPin        13
#define defOnMs       50
#define defPeriodMs   400
```

Constructors

```
blinker(int inPin=defPin,float inOnMs=defOnMs, float
inPeriodMs=defPeriodMs,bool inInverse=false);
```

Creates a blinker to your specs. Give it a pin number, pulse in ms, period in ms and whether it's wired backwards or not. (Backwards being the pulse is pulling to ground)

Public methods

```
void setOnOff(bool onOff);
```

Start or stop the blinking.

```
bool blinking();
```
Returns if we are blinking or not.

```
void pulseOn(void);
```
What to do when the pulse goes high. (Best to ignore this for blinking)

```
void pulseOff(void);
```
What to do when the pulse goes low. (Best to ignore this on as well)

NOTE: Now, you may notice that blinker inherits from squarewave. If you look at squarewave, all the public methods of squarewave are available to blinker. So it can actually do a lot more than what you see on the surface. This and a mapper, and you have a great little RC servo driver.

mechButton

https://github.com/leftCoast/LC_baseTools/blob/master/mechButton.h

Mechanical buttons can be a pain. They “bounce” giving multiple readings when you are expecting only one change. So you have to “debounce” them. You gotta’ check them all the time to make sure you don’t miss a click. Or go through setting up an interrupt to handle them. And, you may have a limited amount of interrupts to use. Wouldn’t it be nice to have a button that “just does something”? Introducing mechButton. Debounced, auto polling using no interrupts, can make a call into your code when it’s state changes. If that’s what you desire. Magic!

mechButton can be setup to use in three different ways. First ,you can use it just like any other button and during loop() just poll it for high or low, actually true or false. It is debounced so you just need to read it’s state. Granted, the way this is written, you will get multiple calls to your “doYourAction()” function whoever it gets pressed. Because it’ll return true until the user lets the button up.

```
#include <mechButton.h>

mechButton myButton(pinNum);

void setup() {  } // Nothing needed for mechButton for polling.

void loop() {
      if (myButto.getState()==false) {      //button has been pressed (grounded)
            doYourAction();
      }
      doUnrelatedStuff();
}
```

The second and more popular way to use mechButton is to create a function for it to call when it’s state changes. “Press button, do this.”

```
#include <mechButton.h>
```

```
mechButton myButton(pinNum);

void setup() {
	myButton.setCallback(btnClk);
}

void btnClk(void) {
	if (myButto.getState()==false) {	//button has been pressed (grounded)
		doYourAction();
	}
}

Void loop() {
	idle();
}
```

Here's a working example of setting a callback for a mechButton.

https://wokwi.com/projects/315187888992027202

This version using the callback only calls your callback when the button's state changes. You get called for a press, and a call for a release. This removes an entire layer of complexity not caring if the button has been held down or not. You only get the changes.

And the third way of using mechButton? You can inherit it and write whatever smart button you can dream up.

Constructors

```
mechButton(byte inPinNum);
Give it a pin number and it'll do the rest.
```

Public methods

```
bool getState(void);
Returns the button's current state.

void setCallback(void(*funct)(void));
Sets the user written function as the callback for this button.
```

NOTE: You need to write a void functionName(void) for the button to call when it's state changes;

```
void takeAction(void);
Something for the Pro's to inherit.

void idle(void);
It's an idler, this is where all the work get's done.
```

autoPOT

https://github.com/leftCoast/LC_baseTools/blob/master/autoPOT.h

autoPOT is a quick and easy way to get analog readings from the analog port. When the reading changes, your callback is called, giving you the new value. Create an autoPOT instance using the analog input pin of your choice. In your setup() function, attach this to your callback function.

```
#include <autoPOT.h>

autoPOT  myPOT(A0);
int      POTReading;

Void setup() {
    myPOT.setCallaback(gotChange);
}

void gotChange(int newReading) { POTReading = newReading; }

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

autoPOT's public interface..

Constructors

```
autoPOT(int inPin);
```

Set the analog pin number with the constructor.

Public methods

```
void setCallback(void(*funct)(int));
```

Write a void function the passes in an int. Call this with that function's name.

```
void setWindow(int plusMinus);
```

Sets a +/- dead-zone window where changes in value from the analog port are ignored. Get a value beyond this "window" and the change is announced in the callback.

```
void idle(void);
```
The autoPOT’s idle method. This is the method that keeps track of the POT’s changes and when to call the user’s callback.

serialStr

https://github.com/leftCoast/LC_baseTools/blob/master/serialStr.h

serialStr watches a serial port for incoming characters. When it reads an end character, it calls your callback function passing in the string that it copied from the serial port. The default is to read from the serial monitor but the can be changed to any Stream input.

Here's a simple sudo-code example that will read strings from the serial monitor and let you do stuff with them.

```
#include <serialStr.h>


serialStr serialMgr;


void setup() {
      Serial.begin(9600);
      serialMgr.setCallback(gotCommand);
}


Void gotCommand(char* cmdStr) {

      // parse cmdStr and do stuff.
}


void loop() {
      idle();
      // Other stuff.
}
```

Or how about a working example?

https://wokwi.com/projects/316669619542688320

Your serialStr public interface..

Constructors

serialStr(Stream* inPort=&Serial,char endChar='\n',int numBytes=DEF_BUFF_BYTES);
Given a incoming stream, the end of string character to look for and a size of the incoming string buffer this creates the serialString object for you to use.

But, Just to make things simpler, all these inputted items have defaults that can be used. The incoming stream is defaulted to the Serial port. The end of string character is defaulted to newline, what the serial monitor uses. The size of the incoming string buffer has a usable number of bytes already preset. So for most purposes you can ignore all of them and just create on with no parameters at all.

Public methods

void setCallback(void(*funct)(char*));
This takes the name of the function you would like to use as a callback. It must be a void function that accepts a char* as an input parameter. When called that incoming parameter will be the string that came in the serial port.

void idle(void);
This is what runs everything in the background. Better to ignore this one.

bool hadOverrun(void);
You can query this method to see if there was a buffer overrun or not. Or "Was the incoming string too long?"

textBuff

https://github.com/leftCoast/LC_baseTools/blob/master/textBuff.h

textBuff was designed to deal with bursts of text coming in from a stream. In many cases, text can come in as bursts and you don't have time to process it. This gives a quick place to store big chunks until you have time to process it.

```
#include <textBuff.h>

textBuff aBuff(100);            // 100 char buffer.

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

void loop() {
      if(Serial.available()) {                  // Got a char?
            aBuff.addChar(Serial.read());  // Read & Save the char.
      }
      // Do your other stuff.
}
```

For an example of using textBuff:

https://wokwi.com/projects/446119909427638273

Now for textBuff's public interface..

Constructors

```
textBuff(int inNumBytes,bool inOverwrite=false);
Specify the number of bytes for the buffer. And, whether or not it should
overwrite when past full.
```

Public methods

```
bool addChar(char inChar);
Adds the inputted char.

char readChar(void);
```

Read out the next char. (removes it)

```
bool addStr(char* inCStr,bool andNULL=true);
```
Add c string. (With or without the null terminator.)

```
int rStrlen(void);
```
Returns the number of chars for the next string to read. Not counting the '\0'.

```
char*   readStr(void);
```
Reads out a c string. Returns a pointer to the string.
NOTE : Make a local copy of the result. Or, use it immediately. It's a temp.

```
int buffSize(void);
```
How many chars CAN we store?

```
int numChars(void);
```
How many chars ARE we storing?

```
bool empty(void);
```
Are we empty?

```
bool full(void);
```
Are we full?

```
int itemCount(void);
```
How many are we storing?

```
int maxItems(void);
```
How many items can we store?

```
void flushItems(void);
```
Dump all the items, reset to empty.

ringIndex

https://github.com/leftCoast/LC_baseTools/blob/master/ringIndex.h

ringIndex takes care of the math and calculations for running ring buffers. It doesn't actually run the buffer. It runs the accounting, given the number of items the buffer can hold. To use it, you would need to create a class that inherits ringIndex and allocates an array for storage of items. Then this class can use ringIndex for supplying all of the array indexes for reading and writing to the array.

textBuff is an excellent example of this.

But, lets say we wanted a ring buffer for integers?

```
//the .h would be something like this..

class intBuff : public ringIndex {
      public:
                  intBuff(int numInts);
                  ~intBuff(void);

                  bool  addInt(int newInt);  // Add an int.
                  int  readInt(void);          // Read out the next int. (removes it.
      private:
                  int*  buff;
};

// The .cpp would be something like this..

#include <intBuff.h>
#include <resizeBuff.h>

ntBuff::ntBuff(int numInts,bool inOverwrite)
      : ringIndex(numInts) {
      int    numBytes;
```

```
	overwrite = inOverwrite;		// Save off, if over writing or not.
	buff		= NULL;			// Always start pointers at NULL.
	numBytes = sizeof(int)*numInts;		// We'll need number of bytes..
	if (!resizeBuff(numBytes,&buff)) {	// If we can't allocate the array?
		numItems = 0;			// We reset the buffer size to zero.
	}					//
}

// Destructor, clean up what we allocated.
ntBuff::~ntBuff(void) { resizeBuff(0,&buff); }

// Add an int and update the state. If its full it can take two different actions.
// If overwrite is true the oldest int will be bumped off and the new int will be saved.
// If overwrite is false the no action is taken a false is returned. In all other cases
// true is returned.
bool intBuff::addInt(int inInt) {

	if (full() && overwrite) {	// If we're full, and overwrite is true..
		readItem();		// Make room by dumping the oldest int.
	}				//
	if (!full()) {			// If not full..
		buff[addItem()] = inInt;	// Add the int.
		return true;		// return success.
	}				//
	return false;			// If we end up here, we failed.
}


// If not empty, read the next int, update the state
// and return the read int. Otherwise return a zero?
int intBuff::readInt(void) {
```

```
    if (!empty()) {                 // If we have some ints..
        return buff[readItem()];    // Return the oldest int.
    }                               //
    return 0';                      // If not, pass back a 0?
}
```

squareWave

https://github.com/leftCoast/LC_baseTools/blob/master/squareWave.h

Square waves are very common in the digital world. Things that blink, blink square waves. Motor controllers and lamp dimmers run on square waves. So many things use this model that it seemed like a good idea to have a base, theoretical, square wave class to use as a foundation of all these things we'd like to use them for.

You have already met one use of the square wave. blinker. blinker is just a square wave packaged up with the ability to turn a digital pin on and off. So anything a square wave can do, blinker can do.

Here it the squareWave's public interface.

Constructors

```
squareWave(void);
Constructs a squareWave that is initialized, but set to all zero times.

squareWave(float periodMs,float pulseMs,bool blocking=false);
Constructs a squareWave with all the goodies.
```

Public methods

```
bool running(void);
Returns if the squareWave is running or not.

bool pulseHiLow(void);
Returns if the pulse state is high or low at this moment.

void setPeriod(float ms);
Sets the length in ms of the square wave period.

void setPulse(float ms);
Sets the length in ms of the pulse time. (Length of on time)

void setPercent(float perc);
Sets the percent time on of the current period of the square wave.

void setBlocking(bool onOff);
Sets permission to block while the pulse is high. Default is off.

void setOnOff(bool onOff);
Turns the pulseWave on or off.

void pulseOn(void);
```

Can be overwritten by the user to perform an action when the pulse comes on.

```
void pulseOff(void);
```
Can be overwritten by the user to perform an action when the pulse turns off.

```
void idle(void);
```
Runs the pulse wave. Can also be overwritten and changed.

NOTE: squareWave objects are constructed in a non-running state. You need to call setonOff(true) to fire them up.

NOTE: Remember squareWave is an idler so idle() must be called in your loop() function.

resizeBuff

https://github.com/leftCoast/LC_baseTools/blob/master/resizeBuff.h

resizeBuff

Now we drop down into the lower level tools. The bits that make up the tools you've already seen. To reduce typing, and force a uniform memory management, resizeBuff() was developed. resizeBuff() will delete a memory allocation and reallocate it to a new size. Returning true if successful. The only rule is, when declared.. All memory buffers MUST be set to NULL! This is why you will see, in these libraries, every time a pointer variable is declared, it's set to initial value of NULL. Because all of the memory allocation besides new() is done by resizeBuff().

Function calls

```
Resizes byte buffers.
bool resizeBuff(int numBytes,uint8_t** buff);

Resizes char* buffers.
bool resizeBuff(int numBytes,char** buff);

Resizes void pointers.
bool resizeBuff(int numBytes,void** buff);

#include <resizeBuff.h>


char* aCString;


setup() {
      aCString = NULL;
      if (resizeBuff(&aCString,50)) {
            Serial.println("Allocated 50 byte string.");
      } else {
            Serial.println("Failed to allocate the string.");
      }
      resizeBuff(&aCString,0);
}
```

The above sudo code declares a char*. Sets it to NULL. Has a go at allocating it. Reports success or failure. Then, regardless of success, recycles the string buffer.

NOTE : Always call resizeBuff() with byte count of zero to recycle your pointer's memory. Whether it successfully allocated it or not.

maxBuff

Lets say you need to write to something with, possibly, more data than you can allocate at one time? What to do? This is a common issue when moving one file to another. You can do it byte by byte, but thats' really slow. What you want is the biggest block of RAM that you can get.

This is where maxBuff comes in. You tell it the number of bytes you need to move and it creates a maxBuff obj with three variables. A pointer to the allocated buffer. the number of bytes in that buffer and the number of "passes" you will need to move your data.

Or, if it fails? The pointer to the buffer will be NULL.

For our example we'll sudo-code fcat(), a file concatenation function. Given destination and source files, Concatenate the source onto the end of destination.

```
void fcat(File dest,File src) {
      unsigned long  filePos;
      maxBuff        cpyBuff(src.size());
      unsigned long  numBytes;
      unsigned long  remaingBytes;

      dest.seek(dest.size());                  // End of dest file.
      filePos = src.position();                // Save the file pos.
      src.seek(0);                             // First byte of src file.
      remaingBytes = src.size();                    // Remaining bytes.
      for (int i=0;i<cpyBuff.numPasses;i++) {
            numBytes = min(cpyBuff.numBuffBytes,remaingBytes);
            src.read(cpyBuff.theBuff,numBytes);
            dest.write((char*)(cpyBuff.theBuff),numBytes);
            remaingBytes = remaingBytes - numBytes;
```

```
    }
    src.seek(filePos);                    //  Put it back.
}
```

Constructors

`maxBuff(unsigned long numBytes,unsigned long  minBytes=BYTE_CUTOFF);`
Pass in the total amount of bytes to move along with the minimum buffer size before calling it quits.

Public variables

`void* theBuff;`
The actual allocated buffer.

`unsigned long numBuffBytes;`
Indicates the size of this allocated buffer.

`int numPasses;`
The amount of passes through this buffer to complete the job. (See the for loop above).

lists

https://github.com/leftCoast/LC_baseTools/blob/master/lists.h

Lists is all about dynamic linked lists. There are two types. Single linked lists that have a manager class, and double linked lists that can be self managing. No manager is supplied for the double linked lists.

These are meant as a toolbox as base classes for all sorts of stuff. They are very handy, but you must be comfortable dealing with pointers and type casting to really be able to use them.

linkListObj

Single linked lists. Good for lists and stacks. This is the linkListObj class. They are like the beads that go on the necklace.

Constructors

```
linkListObj(void);
Constructs an empty, unconnected node.
```

Public methods

```
void linkAfter(linkListObj* anObj);
Given a pointer to a node, link yourself after it.

void linkToEnd(linkListObj* anObj);
Given a pointer to a node, link yourself after the last in the chain.

linkListObj* getNext(void);
Pass back the next pointer. (Each has one, it's the link.)

void setNext(linkListObj* ptr);
Point somewhere else. (Point to any other node.)

void deleteTail(void);
Call delete on everyone hooked to us. (Snips off the tail and recycles the
nodes.)

bool isGreaterThan(linkListObj* compObj);
Are we greater than the obj being passed in? Primary sorting function.

bool isLessThan(linkListObj* compObj);
Are we less than the obj being passed in? Primary sorting function.
```

linkList

Now as stated before single linked lists can't really manage themselves because they have no idea what is linked to them. So they need a little help from a link list manager. For this task we have the linkList class. (Strings the beads together and keeps track of them.)

Along with the usual adding and removing nodes, it has the ability to sort the list. If the nodes have their greaterThan() lessThan() methods filled out.

NOTE : Be wary of the two unlink methods because they do NOT dispose of the nodes they unlink. That is up to you. dumpList() does recycle everything. As does the destructor for the linkList itself.

Constructors

```
linkList(void);
Constructs a single linked list manager ready to use.
```

Public methods

```
void addToTop(linkListObj* newObj);
Give it a node and it'll add it to the top. (Fast)

void addToEnd(linkListObj* newObj);
give it a node and it'll addd it to the end  (Slower)

void unlinkTop(void);
Push off the first one. (Does NOT delete it.)

void unlinkObj(linkListObj* oldObj);
Find it and push this one off. (Does NOT delete it.)

void dumpList(void);
Unhooks all the nodes from the list and delete them all.

bool isEmpty(void);
Returns if the list is empty or not.

linkListObj* getFirst(void);
Hand back a link to the top node.

linkListObj* getLast(void);
Hand back a link to the list node on the list.

linkListObj* findMax(linkListObj* anObj);
If you filled out the isGreaterThan() & isLessThan() methods.
```

```
linkListObj* findMin(linkListObj* anObj);
These two will pass back links to the max & min of the list.

void sort(bool ascending);
And, if you did fill them out, this'll sort the list.

int getCount(void);
Count the number of nodes, hand back that number.

linkListObj* getByIndex(int index);
Same as a c type array. Hand back a link to the node at index.
int findIndex(linkListObj* anObj);
returns -1 if NOT found.

void looseList(void);
Someone has taken control of our list, let it go.
```

Stack & queue

Stacks have Push, Pop and Peek methods. Push adds a node to the stack. Pop pulls the latest item from the stack and hands it to you, Peek shows you the latest item on the stack without removing it.

Queues are exactly the same except, Peek and Pop deal with the earliest item on the list. The one that's been on the list longest.

Both stack & queue are supplied for you to use and/or inherit for other classes.

NOTE: The pop() methods do NOT recycle the nodes that they remove from their lists. They hand the pointer to those nodes to you and you have to recycle them. Calling delete on the stack or queue themselves DOES delete all the nodes.

First we have the stack..

Constructors

```
stack(void);
Creates an empty stack for you to use.
```

Public methods

```
void push(linkListObj* newObj);
Adds en item to the list.
```

```
linkListObj* pop(void);
Removes an item from the list.(Does NOT delete it.)

linkListObj* peek(void);
Hands back a pointer to the next item to be removed from the list.
```

And the queue, much the same.

Constructors

```
queue(void);
Creates an empty queue for you to use.
```

Public methods

```
void push(linkListObj* newObj);
Adds en item to the list.

linkListObj* pop(void);
Removes an item from the list.(Does NOT delete it.)

linkListObj* peek(void);
Hands back a pointer to the next item to be removed from the list.
```

dblLinkListObj

The dblLinkListObj is, like it says, a node that has two links previous and next. So, by keeping track of these, one can run up and down the string of nodes. Double link lists tend to be self managing, so no manager class has been supplied.

Anyway lets see the public interface to dblLinkListObj.

Constructors

```
dblLinkListObj(void);
Creates a ready to use dblLinkListObj  .
```

Public methods

```
void linkAfter(dblLinkListObj* anObj);
Given a pointer to a node, link yourself after it.

void linkBefore(dblLinkListObj* anObj);
Given a pointer to a node, link yourself before it.

dblLinkListObj* getFirst(void);
Runs up the list 'till dllPrev == NULL. Returns link to that node.

dblLinkListObj* getLast(void);
```

Runs up the list 'till dllNext == NULL. Returns link to that node.

```
void linkToEnd(dblLinkListObj* anObj);
```
Given a pointer to any node, link yourself after the last in the chain.

```
void linkToStart(dblLinkListObj* anObj);
```
Given a pointer to any node, link yourself before the first in the chain.

```
dblLinkListObj* getTailObj(int index);
```
Hand back the "nth" one of our tail. Starting at 0.

```
void unhook(void);
```
Unhook myself.

```
void dumpTail(void);
```
Delete entire tail.

```
void dumpHead(void);
```
Delete entire head section.

```
void dumpList(void);
```
Delete both head & tail.

```
int countTail(void);
```
Returns how many nodes in our tail.

```
int countHead(void);
```
Returns how many nodes in our head.

Variables

```
dblLinkListObj*  dllPrev;
```
This one points a up the list to the guy before us.

```
dblLinkListObj*  dllNext;
```
This one points down the list to the guy after us.

LC_lilParser

https://github.com/leftCoast/LC_lilParser

There are times when you really want to test stuff in your code and you start adding Serial read commands and it gets out of hand.. (Sigh..) What one rally needs is an easy to setup back door. lilParser is your ticket!

LilParser takes in string commands in the form of Command word followed by an optional string of param words followed by an EOL character.ow to set this up?

First you need to write and enum command list starting with, noCommand.

```
enum myComs {
      noCommand,
      firstCom,
      secondCom
};
```

Then in your setup() function. You need to link what you plan on typing to each command. A command can have more than one string linked to it. Typed commands or parameters can have no white space in them.

```
myParser.addCmd(firstCom,"doFirst");
myParser.addCmd(secondCom,"doSecond");
```

Now in loop() you need to read the Serial port and feed the chars into the parser.

```
void loop(void) {

      char  inChar;
      int   command;

       if (Serial.available()) {                  // If serial has some data..
            inChar = Serial.read();               // Read out a charactor.
            Serial.print(inChar);                 // Echo the charactor.
```

```
            command = ourParser.addChar(inChar);  // Try parsing what we have.
            switch (command) {                                  // Check the results.
                  case noCommand :                  break;      // Nothing to report, move
along.
                  case firstCom : handleFirst();    break;      // Call handler.
                  case secondCom : handleSecond();  break;      // Call handler.
                        default :  Serial.println("What?");     break;      // No idea. Try again?
            }
      }
}

void handleFirst(void) {
      // do whatever first should do.
}

void handleSecond(void) {
      // do whatever second should do.
}
```

Dealing with command parameters. For this there is supplied numParams() method that will give you the number of command parameters that came along with the command. Also there is the getNextParam() method. Here's an example on how to use these.

```
void handleFirst(void) {

      char* paramStr;

      if (myParser.numParams()) {                   // If there is a param.
            paramStr = myParser.getNextParam();     // grab it.
            if (!strcmp(paramStr,"on") {            // If it's "on".
                  turnLED(true);                    // Turn on the LED.
            } else {                                // In all other cases?
                  turnLED(false);                         // Turn it off.
```

```
        }                                   //
    } else {                                // No param?
        Serial.print("LED is ");            // Then output..
        Serial.println(getLEDState());      // The LED state.
    }
}
```

So let's take a look at lilParser's public interface.

Constructors

```
lilParser(int inBufSize=DEF_BUFF_SIZE);
Returns a parser ready to go with at least a default pram buffer.
```

Public methods

```
void addCmd(int inCmdNum,const char* inCmd);
Adds a link from a command number to a typed command string.

int addChar(char inChar);
Pass a new character read from the incoming stream.

int numParams(void);
Returns the number of parameters packaged along with the incoming command.

char* getNextParam(void);
Passes back a pointer to the next param string available. Don't recycle it.
It's owned by the parser and the parser will reuse it.

char* getParamBuff(void);
Passes back a string containing all the params. Again, don't recycle it. It's
owned by the parser and the parser will reuse it.
```

LC_slowServo

https://github.com/leftCoast/LC_slowServo

So, you want to control your RC servo speed, but trying to do that with delay() is making everything else in your code stop? Here's your solution. slowServo will run your RC servos in the background and you can adjust their speed before or during a move.

How about a running example?

https://wokwi.com/projects/358434260375219201

Lets have a look at slowServo's public interface.

Constructors

```
slowServo(int inPin,int startDeg=0);
Contructs a servo object, saves some intial data. Can have initial position
set if desired. So your servo doesn't do a random wipe on startup.
```

Public methods

```
void  begin(void);
Puts everything together for a working slowServo. Including hooking it into
the idle loop.

void  setMsPerDeg(int inMs);
Sets rate of movement. Ms/degree.

void  setDeg(int inDeg);
Sets desired position in degrees.

bool  moving(void);
Returns if the software thinks the servo is still moving or not.

void  stop(void);
Tells the servo that it really wants to be where it is currently at.

void  idle(void);
Runs the servo in the background.
```

NOTE: The servoMoving() call works only if you are telling the servo t move slower than it can. This call follows the commanded angle over time not the actual angle.

LC_neoPixel

https://github.com/leftCoast/LC_neoPixel

This library main task is to tie the the Left Coast colorObj into Adafruit neoPixels. It contains two different classes. The first being neoPixel.

neoPixel

https://github.com/leftCoast/LC_neoPixel/blob/master/src/neoPixel.h

This class extends Adafruit's Adafruit_NeoPixel class allowing colorObj info to be passed in and out. This class also adds some nifty functions like, setAll() which sets all the pixels on the strip to the inputed colorObj's color. Also, there is shiftPixels(). This moves all the pixels on your strip over by one returning the last one to you, so you could, if you want, pop it back in the other end. Then there's roll() that does pretty close to shiftPixels() but is setup for rolling color patterns around neoPixel rings.

Lets have a look at the neoPixel public interface.

Constructors

```
neoPixel(uint16_t n, uint8_t p, uint8_t t=NEO_GRB + NEO_KHZ800);
Constructor, for most neoPixels you will run across all you will need is
number of pixels and pin number.
```

Remember to call begin() on the strip before trying to make it go. Since it's an inherited call, I sometimes forget.

Public methods

```
bool isRGBType(void);
returns true if this is a 3 byte RGB pixel, False for4 byte including the soft
white LED.

void setPixelColor(uint16_t n,RGBpack* inColor);
This uses an RGBpack to set the pixel color. Very handy for copying .bmp image
files.

void setPixelColor(uint16_t n,colorObj* inColor);
This uses a standard colorObj for setting a pixel color. Allows blending of
colors and things.

colorObj getPixelColor(uint16_t n);
Returns a colorObj set to the same color as a pixel.
```

```
void setAll(colorObj* color);
Sets all the colors on a strip to the same colorObj's color. Added 'Cause its
so handy.

colorObj shiftPixels(bool toEnd=true);
Shifts all the color on a strip one space over, either to the end or to the
beginning of the strip. Passes back a colorObj of the last color that was
"pushed off". Very very very handy!

void roll(bool clockwise=true);
For rolling pixelRings. See the wokwi example in the next section.
```

chainPixels

https://github.com/leftCoast/LC_neoPixel/blob/master/src/chainPixels.h

This next set of classes are all about chainPixels. Imagine you have a project that includes a bunch of pixel groups. Each needs to "play" a different pattern. Maybe some patters will be repeated in other places? Wouldn't it be nice if you could just daisy chain all these groups together on one data line? With chainPixels, this is suddenly very easy. Just program each pattern independently as if it had it own private set of pixels. Then, when you have all your patterns working as you would like, just add them all to a chain pixel queue in the order that the groups are wired. You can even add them multiple times. ChainPixels combines and manages the entire string for you.

The usual example for this would be a quadcopter. You have your taillight showing roll, yaw and power settings. Your floodlights for night flying, and many have pixel rings on the motor pods showing other cools stuff. Ok, Write a taillight object, a floodlight object, a motor ring object. String the data wire from your processor pin, to the tail light, each motor pod light, and through the flood lights, In any order that makes wiring easy. Now, just take instances of your light objects and stuff them in the queue in the same order that they acre connected to the data line. Done!

Here's an example of chain pixels :

https://wokwi.com/projects/449595175359016961

Let's have a look at chainPixels & pixelGroup's public interfaces. For most projects, chainPixels can typically be used just as it is.

chainPixels

NOTE : Pixel numbers in chainPixels are global to the string of pixels. In pixelGroup they are local to that actual group itself.

Constructors

```
chainPixels(byte inPin);
```
Constructs a chainPixels objet given a pin number to associate it with.

Public methods

```
void resetChain(void);
```
Used internally if a group is added or removed.

```
void push(linkListObj* item);
```
Also used internally for adding groups.

```
linkListObj* pop(void);
```
Again, internal method for removing groups.

```
void addGroup(pixelGroup* inGroup);
```
Used by the user to add a pixelGroup.

```
void idle(void);
```
Runs everything in the background.

```
colorObj getPixelColor(word index);
```
Used by the pixelGroups to return the color of a pixel.

```
void setPixelColor(word index,colorObj* inColor);
```
Used by the pixelGroups to set the color of a pixel.

NOTE: really the only call you need from chainPixels is the addGroup() method. Just ignore the rest for now.

pixelGroup

Now for the pixelGroup class. This is the one you will need to inherit and modify for your purposes.

Constructors

```
pixelGroup(word inNumPixels);
```
Constructs a pixelGroup given the number of pixels it contains.

Public methods

`void setIndex(word inIndex);`
Used internally to set our pixel index starting point. (offset)

`void setChain(chainPixels* inChain);`
Used internally to seta link to our owner.

`word getNumPixels(void);`
Returns the number of pixels in this group.

`colorObj getPixelColor(word pixelNum);`
Returns the color of that pixel. Relative to our zero pixel.

`void setPixelColor(word pixelNum, colorObj* color);`
Set THIS pixel this color. Relative to our zero pixel.

`void setPixels(colorObj* color);`
Set ALL our pixels this color. Very handy!

`colorObj shiftPixels(bool toEnd=true);`
Shift all colors over by one pixel. return the displaced end color. Even handier!

`void roll(bool clockwise=true);`
Used for pixel rings. ROlls a pattern one click clockwise or counter clockwise.

`void draw(void);`
This will be called repeatedly. Override and fill with your drawing code.

NOTE: As of this writing the roll() method is pretty screwed up. Goes backwards and messes up some of the pixels. When used in real life, I didn't notice. Slowed it down in Wokwi and it was obvious. Needs some help.

LC_cardIndex

https://github.com/leftCoast/LC_cardIndex/tree/main

If you would like to deal out playing cards, or anything like that where you need to choose items out of a set, one at a time. This will do the trick for you. You need to know how many items (cards) are in your set. And you will need to come up with mapping from numbers to what each card would be.

For example playing cards number 52 cards. You have 52 cards. Figure out a numbering for them 1..52. Meaning? I give you a number say.. 26. You will need to be able to tell me what card that is. However you like to order them is up to you. Your "cards" will be numbered from 1. So playing cards would be 1..52. Returning a zero means there are no more cards.

cardIndex.h

https://github.com/leftCoast/LC_cardIndex/blob/main/src/cardIndex.h

A look at the user interface :

Constructors

```
cardIndex(int inNumCards);
```
Create a card index by telling it how many cards in it's deck.

Public methods

```
void shuffle(int inNumCards=0);
```
This is used to reset the deck. Pick up all the cards and.. Shuffle them.

NOTE: Only pass in a value if you want the amount of cards to be changed.

```
void omitCard(int value);
```
Used for removing cards from the deck.
NOTE: Only for this current deal. Once shuffle() is called, it will be a full deck of cards again.

```
int dealCard(void);
```
Randomly selects a card from the list. Removes it from the list, and returns the value to you.

```
int cardsLeft(void);
```
Returns the number of cards left to deal.

NOTE : The cardIndex class is what was shown above. There is also the indexObj class, but that's used internally. You should just ignore that class.

LC_piezoTunes

Need link here

This is a toolkit for playing "simple" tunes from a piezo "speaker". The original idea for this was to have something to play little tunes for games. And for that, it actually works pretty well. Then it was thought that, if you could read decode MIDI files? Well, making the tunes would be so much easier. You could just play them on your favorite MIDI device, capture the information, and use that to build your tune.

Well, that part sort of works. For simple, one voice, staccato tunes, it works fine. Start getting fancy? It starts to have a really rough time. Hopefully, with time and updates, it'll get better. That being said. I use it now and it needs to be documented.

How this all works, we'll start with the building blocks and work up.

toneObj

need link

First there is a list of frequencies and their note names. This is replicated in different places, but the idea was that since we are building on it, it would be nice to have a copy where we could have control over it. I'm not going to list them here, you can see them in the source code. There are some common note types and default time/quarter note. Those should probably be listed here.

Definitions

```
#define REST          0         // Freq. of zero means "rest".

#define Q_NOTE        250.0     // milliseconds

#define H_NOTE        2 * Q_NOTE
#define DH_NOTE       3 * Q_NOTE
#define W_NOTE        4 * Q_NOTE
#define E_NOTE        Q_NOTE/2.0
#define S_NOTE        Q_NOTE/4.0
```

toneObj

The tone class is basically a pin number where you can crate a output frequency, or a tone, for a certain amount of time. And be able to query whether it is playing a tone currently.

Constructors

```
toneObj(int inPin);
Give it a pin number and it returns a toneObj. Currently, the pin you choose
needs to be able to generate an output square wave. Some do, some don't.
Choose wisely.

void play(int inHz,float inMs);
This will play a tone, of the inputted frequency for the inputted
milliseconds.

bool isPlaying(void);
Returns whether this is playing a tone or not.
```

note

This is basically a chunk of time. Either a note, or a rest. This class is setup as a linked list to be strung together in a list by the ***tune*** class. These end up being the notes for your tune.

Constructors

```
note(int inHz,float inMs);
```
Frequency and duration in milliseconds gives you a note. There are common note types defined to make this easy to do. If doing this by hand.

```
note* getNext(void);
```
Used by tune to grab the next note on the list.

```
void playSelf(toneObj* inTone,float timeMult);
```
Also used by tune to make this note play its sound or rest for it's set time.

tune

This manages the list of notes that make up your tune. You add notes to it to build your tune then when you want to play your tune you hand it a toneObj to play it with.

Constructors

```
tune();
```
No parameters, just create one.

```
void addNote(int inHz,float inMs);
```
This is where you add notes to the list. Do remember to add them in order.

```
void adjustSpeed(float inTimeMult=1);
```
The default speed is 250 ms/quarter note. The time multiplier can make that time longer, or shorter.

NOTE : A time multiplier of 0.5 makes the tune run twice as fast.

```
void startTune(toneObj* inTone, int inIndex=0);
```
Had this a toneObj and it'll start playing it's tune through that pin. It will start at the beginning, unless you specify a later index value. Then it'll start there.

```
int stopTune(void);
```
Turn it off.

```
bool playing(void);
```
Rsdturns whether we are playing a tune now.

```
void idle(void);
```
The engine that runs everything in the background. Basically says, if we are playing a tune, and we have a valid tone object. See if the toneObj is currently playing a note. If not? Load the next note into it, and kick it back on.

MIDItune

MIDItune opens, reads and decodes a MIDI file from the SD card. Then it uses this information to load a list of notes into it's base tuneObj to be played.

Constructors

```
MIDItune(void);
No pramaters, just create one.

void createTune(const char* filePath);
Given a file path to a MIDI file. Decode this file and build a tune with it.
```

And that's about it.

MIDINotes

need link

The MIDI decoder doesn't actually have any classes defined in it. It's just a bunch of constant definitions and functions calls. A few of these functions are just for debugging and seeing how these files are laid out internally. But, this is the code that make the MIDITune class possible.

Definitions

```
#define META_SEQUENCE_NUM 0x00
#define META_TEXT         0x01
#define COPYWRITE         0x02
#define TRACK_NAME        0x03
#define INST_NAME         0x04
#define LYRIC             0x05
#define MRKER             0x06
#define CUE_POINT         0x07
#define CHANNEL_PREFIX    0X20
#define END_OF_TRACK      0x2F
#define SET_TEMPO         0x51
#define SMPTE_OFFSET      0x54
#define TIME_SIG          0x58
#define KEY_SIG           0x59


enum MIDINums {
    MIDI_B0 = 11,
    ..
    ..
    MIDI_C1,
    MIDI_DS8
};

#define MIDI_KEY_DN       0x9
#define MIDI_KEY_UP       0x8

struct MIDIHeader {
    char     chunkID[4];
    uint32_t chunkSize;
    uint16_t formatType;
    uint16_t numTracks;
    uint16_t timeDiv;
};
```

```
struct trackHeader {
   char     chunkID[4];
   uint32_t chunkSize;
};

struct eventHeader {
    uint32_t deltaTime;
    byte     eventType;
    byte     channel;
    byte     param1;
    byte     param2;
};

struct metaEvent {

    byte     metaType;
    uint32_t numBytes;
    uint32_t location;
};
```

Functions

```
bool readMIDIHeader(MIDIHeader* theHeader, File MIDIFile);
```
First thing found in the MIDI files was a header. This reads and decodes it.

```
bool readTrackHeader(trackHeader* theHeader, File MIDIFile);
```
Next in line are tracks. The files we can deal with have (1) track. (So far)

```
bool readEventHeader(eventHeader* header, File MIDIFile);
```
Tracks are made up of a list of events. This decodes an event.

```
bool readMetaEvent(eventHeader* theHeader, metaEvent* theEvent, File
MIDIFile);
```
If it IS a meta event, this'll decode the meta header for you. Once you read the Meta header, you need to either go through the data, or jump past it. See next two functions for that choice.

```
bool jumpMetaEvent(metaEvent* theEvent, File MIDIFile);
```
Nine times out of ten you just want to jump past Meta events. They typically have trailing data blocks. This calculates the hyperspace jump to land past the entire event from the point of just finishing reading the header (above).

```
bool readAndShowMetaData(metaEvent* theEvent, File MIDIFile);
```
This basically drags you through the meta event's data. Some have data, some of it's readable text. Some is binary. This will show you what it found. Byte by grueling byte.

```
bool showMIDIHeader(MIDIHeader* theMIDIHeader);
```
Just read the MIDI header and wonder what it has in it? This'll print it out.

```
bool showTrackHeader(trackHeader* theHeader);
```
Just read the track header and wonder what it has in it? This'll print it out.

```
bool showEventHeader(eventHeader* theHeader);
```
Just like the others. Just read the event header and wonder what it has in it? This'll print it out.

```
bool isMetaTag(eventHeader* theHeader);
```
There are basically note events and Meta events. And a third I've not seen. This looks at an event Header, and says whether its Meta or not.

```
int MIDI2Freq(int MIDINote);
```
And this guy.. This maps a MIDI note to a sound freq. Kinda' glue code for the Piezo buzzer.

```
bool decodeFile(const char* filePath);
```
This'll read through a MIDI file. And, using the functions above, show you what it thinks is in there. Basically a debugging tool

Graphics

Oh boy.. The Left Coast **G**raphical **U**ser **I**nterface. (**GUI**)

For drawing to a display, the Left Coast GUI keeps a list of objects to be drawn. Think of them like playing cards spread on a table. When looking at the cards, they are drawn from the bottom up. Each card can partially or completely cover the cards below it. When you reach down to touch a card, you figure out what card you touched by checking from the top down. This is the bases for drawing and clicking. Draw up from the bottom up. Check for clicks from the top down.

This list of drawable and possibly clickable objects are based on the drawObj class. The manager of the list of drawable objects is accessed through global variable viewList.

But, what about the display itself? All displays that you can draw on are based on displayObj. The nice thing about that is, they all work the same. Some will have touch screens, some don't. They can all have different X & Y sizes. But they all use the same drawing commands. And those commands are called, using the same global variable, *screen*.

One other thing. When you draw, you set the color that you draw with using? Our old friend colorObj. This same colorObj that you use to set the color of neoPixles. Meaning? You can freely pass colors between neoPixles, displays and .bmp files.

There is also an event manager working in the background for touch screens. The event manager ties all this interactive stuff together.

Anyway, let's see how to draw something to a display using this stuff shall we?

We need a display. Currently we have drivers for..

LC_Adafruit_684

LC_Adafruit_1431

LC_Adafruit_1947 - My favorite. And? Available in Wokwi!

LC_Adafruit_2050

LC_DFRobot_0995

We'll get to those later.

We'll pretend we have an Adafruit 1947. We hook up the SPI bus (For the screen), the I2C bus (For the capacitive touch), and the Display and the SD card chip select wires. If you are hooking to an UNO, Mega, rPi pico or Teensy. The system should automatically choose the correct pins for your display's hookups.

Now drawing, clicking, dragging, etc.? This is not something **you** as user of this stuff deals with. You don't call the draw command directly or look for clicks etc. You add your objects to the list, and they, with the help of the viewMgr. Run everything in the background by themselves. The logic is in the objects.

Let's start out with a simple program using the toolset we've been reading about.

We'll have a rectangle that fades from red to the screen background by turning a dial.

We'll have a label on the screen. This label will watch the serial port and when you type something in the serial monitor, it'll show up as that label.

```
#include <adafruit_1947.h>
#include <label.h>
#include <colorRect.h>
#include <autoPOT.h>
#include <mapper.h>
#include <serialStr.h>

#define DSP_CS      10                          // Display chip select

label*      HelloText;                          // Pointer to a label object.
serialStr   serialMgr;                          // Manager for the serial port.
colorRect*  ourRect;                            // A colored rectangle object.
colorObj    backColor;                          // Background color.
colorObj    rectColor;                          // Rectangle color.
autoPOT     rectControl(A0);                    // Analog pin manager.
mapper      percentMaper(0,1023,0,100);         // Map raw POT values to a percent.
colorMapper rectColorMap;                       // Map colors as a percent.


// setup() here is mostly for hardware setup. If that all works,
// then we go on to setup the screen object and controls.
void setup() {

    Serial.begin(57600);                                        // Fire up serial.
    screen = (displayObj*) new adafruit_1947(DSP_CS,-1);        // Create screen.
    if (screen) {                                               // We got one?
```

```
        if (screen->begin()) {                          // begin() ok?
            screen->setRotation(PORTRAIT);              // For handhelds.
            setupScreen();                              // Build screen.
            return;                                     // All good, go.
        }                                               //
    }                                                   //
    Serial.println("NO SCREEN!");                       // Send an error.
    while(true)delay(10);                               // Lock processor.
}


// When the serialMgr sees a message come in. It arrives here.
void newMsg(char* inStr) { HelloText->setValue(inStr); }


// When the POT sees a change in value. The new value arrives here.
void rectColorChg(int newValue) {

    float     percent;
    colorObj  newColor;

    percent = percentMaper.map(newValue);  // Map raw value to a percent.
    newColor = rectColorMap.map(percent);  // Map percent to color.
    ourRect->setColor(&newColor);          // The rect becomes that color.
}


// Puts all the screen bits and things together.
void setupScreen(void) {

    int height;

    // Setup background display color.
    backColor.setColor(&blue);                      // Set this color to blue.
    backColor.blend(&black,70);                     // Mix in some black.
    screen->fillScreen(&backColor);                 // Set background color.

    // Setup a label we can interact with.
    HelloText = new label("Hello world!");          // Create a label.
    if (HelloText) {                                // If we got one..
        HelloText->setColors(&yellow,&backColor);   // Set text color to yellow
        HelloText->setLocation(10,10);              // Set the top left corner.
        height = HelloText->height;                 // Save off the height.
        HelloText->setSize(300,height*2);           // Reset it's size.
        HelloText->setTextSize(2);                  // Make it bigger.
        viewList.addObj(HelloText);                 // Add to view list
    } else {                                        // Didn't get a label?
        Serial.println("No label!");                // Tell Miss user.
```

```
        while(true)delay(10);                           // Lock processor.
    }                                                   //
    serialMgr.setCallback(newMsg);                      // Set the callback.
    Serial.println("Type a message for the label");// Give Miss user a hint.

    // Setup a rectangle we can change the color of.
    rectColor.setColor(&red);                               // Set the rect color.
    ourRect = new colorRect(20,40,50,50,&rectColor);    // Our colorRect.
    if (ourRect) {                                          // If we got one..
        viewList.addObj(ourRect);                           // Add to view list
    } else {                                                // No rect?
        Serial.println("No color rect!");                   // Tell Miss user.
        while(true)delay(10);                               // Lock processor.
    }                                                       //
    rectColorMap.setColors(&rectColor,&backColor);      // Setup color map.
    rectControl.setCallback(rectColorChg);              // Setup callback.
}


// In loop, to run all of this, all you need is a call to idle.
void loop() { idle(); }
```

To see all of this in action, there is a working Wokwi simulation.

https://wokwi.com/projects/446482597378580481

So to recap. You basically have a double link list of drawObjects. During idle time, the viewMgr will run up the list from the bottom checking to see if any of these objects need to be redrawn. If so, they will have their draw() method called. This in turn will end up calling their drawSelf() method. The draw() method basically sets up the drawing environment for the object. the draw() method is typically left alone and not rewritten. drawSelf() is typically inherited and customized to do this object's specific drawing. You want something different to be drawn? Change your drawSelf() method. **NOT** your draw() method.

There is a lot more included in this stuff, but this should at least get you going for putting things on the screen.

Lets have a look at the basic set of things we have available to draw.

drawObj (overview)

https://github.com/leftCoast/LC_GUITools/blob/main/drawObj.h

This is the base class of all drawing items. It's a rectangle that can be told that it needs to be redrawn. It can be clicked on and programmed to respond to clicks and drags. It's a double linked list node, so it can link with other drawObj objects in a chain. It will draw itself when necessary. But as it stands, it draws itself in a very boring manner. Black rectangle with white border.

Later on you will discover event sets for interaction, focus() to show what drawObj the user is currently interested in, different flavors of doAction() that can be inherited, setCallback() for those that like that approach.

We'll go into all that later. For now lets just focus on drawing stuff. Just wanted you to know that this is the base of everything that will be on the display.

LC_GUITools (overview)

https://github.com/leftCoast/LC_GUITools/tree/main

So, here lies all the **basic** bits that make up stuff to draw on the screen. Like we said above, this is kinda' an overview to get you going. To recap. The base drawing class is drawObj. It's a rectangle that does all the tricky bits that allows it to "live" on a display.

baseGraphics holds the fundamentals of things like points & rects. You may need to drop down to this level on occasion. So we'll leave a link to it here.

https://github.com/leftCoast/LC_GUITools/blob/main/baseGraphics.h

But for now, we'll focus on things we can draw today.

That list follows..

colorRect

https://github.com/leftCoast/LC_GUITools/blob/main/colorRect.h

This is a rectangle that is also a colorObj. So you can put it on the display and use all of the colorObj methods on it as well as color mappers etc.

The colorRect public interface:

Constructors

```
colorRect(void);
This gives you a default rect. If you look in baseGraphoics.h you will see
that it's location is 16,16 and it's size in pixels is 16x16. And the default
color can be found in colorObj.cpp as black.

colorRect(rect* inRect,colorObj* inColor,int inset=0);
This one copies the size and location from the passed in rect. The color from
the passed in color and can be inset or not. Inset is a shadow and highlights
that make the rectangle look inset. A negative value of inset makes the rect
look puffed out, like a button.

colorRect(int inLocX,int inLocY,int inWidth,int inHeight,colorObj* inColor,int
inset=0);
This one works the same as the one above, only it's x,y location and width,
height size are sepereate values as opposed to be loaded into a rect.

colorRect(int inLocX,int inLocY,int inWidth,int inHeight,int inset=0);
This one works the same as the one above, only it uses the default, black
color.
```

Public methods

```
void setInset(int inset);
Inset will darken the top and left edges by 50%, lighten the bottom and right
edges by 50% for "inset" pixels. Making the rect look like it's inset. If
inset is passed in as a negative the effect is reversed making it look like
the rect is "proud" of the background. Like a button.

void drawSelf(void);
Where all the math and colors come together to make this look like a color
rect.

void setColor(byte r,byte g,byte b);
Given RGN values this will change the color of the rect and force a redraw so
you can see it.

void setColor(word color16);
```

Same as above but a two byte color16 is passed in. Most small color displays seem to use these two byte colors for their preferred color data type.

```
void setColor(colorObj* inColor);
```
As above again but using a colorObj as the inputted color.

```
void setLocation(int inX, int inY);
```
This one moves the rect and forces a redraw. All if these remaining calls do nearly the same thing. Change location, size or shape. Forcing a redraw but NOT erasing the current rect. How the original is cleared, if you want to cleared at all, is up to you.

```
void setSize(int inWidth,int inHeight);
```
Changes the shape of the rect and forces a redraw.

```
void setRect(rect* inRect);
```
Copies inRect to change the shape and/or location of the rect. Then forces a redraw.

```
void setRect(point* inPt1,point* inPt2);
```
Using two points this sets the size and location of the rect. And again forcing a redraw.

```
void setRect(int inX, int inY, int inWidth,int inHeight);
```
Same as above using values.

```
void insetRect(int inset);
```
This one is different! This does not change how it's drawn. This changes the the size of the rect by inset amount of pixels. Positive values make it smaller, negative values make it bigger. Then forces a redraw.

```
void addRect(rect* inRect);
Stretches this rect to fit inRect inside. Making a bigger rect. Then redraw..
```

That should get you going, To see more things you can do with rects see the base rect class in baseGraphics. Lot more stuff there.

flasher

https://github.com/leftCoast/LC_GUITools/blob/main/flasher.h

Flasher is like a colorRect except it has two colors on and off color wrapped in with a square wave. When the pulse is high, you get the onColor. When the pulse is low, you get the offColor. Basically giving you a blinking rectangle.

But, you don't have to draw a rectangle. You could make a class based on this, rewrite drawSelf() and draw anything you like, in two states. Also, all the pubic methods from square wave are available to you to mess about with. The possibilities are endless!

Once again let's take a look at eh public interface.

Constructors

```
flasher(rect* inRect,colorObj* offColor=&black,colorObj* onColor=&red);
Given a rect for size and location along with an onColor and an offColor this
gives us your basic black to red flasher.

flasher(int inLocX,int inLocY,int inWidth,int inHeight,colorObj* offColor=
&black,colorObj* onColor=&red);
Same as above except using values for size and location.
```

Public methods

```
void setColors(colorObj* onColor,colorObj* offColor);
Used for resetting different colors.

void drawSelf(void);
The inheritable drawSelf method. Does the actual drawing to the screen.

void pulseOn(void);
From the squareWave class that is built into flasher. Called when the
squareWave goes high. We use it for switching to onColor and forcing a redraw.

void pulseOff(void);
Also from the squareWave class that is built into flasher. Called when the
squareWave goes low. We use it for switching to offColor and forcing a redraw.
```

NOTE: Remember this is run by the squareWave class. Meaning? You must turn it on before it'll run, using the : setOnOff(true); squareWave method.

lineObj

https://github.com/leftCoast/LC_GUITools/blob/main/lineObj.h

Lines. Not a lot to say about lines. Two points, draw a line between them. There is the caveat that everything drawn is actually a rectangle. So there are actually four points to choose from. This is resolved by the enum slopeType that specifies what points to use.

Let's just get into the public interface now, shall we?

Definitions

```
enum slopeType {
    vertical,
    positiveSlope,
    negativeSlope,
    horizontal
};
This will set what two points in a rectangle define the actual line.
```

Constructors

```
lineObj(void);
Creates a default line. Positive slope across 16x16 rectangle drawn in black.

lineObj(int x1,int y1,int x2,int y2,colorObj* inColor);
Creaes a line from x1,y1 to x2,y2 drawn in inColor.
```

Public methods

```
void setColor(colorObj* inColor);
Sets a new color for the line to be drawn in. Causes redraw.

void setSize(byte inSize);
Sets the line width size in pixels. Sadly, never was supported. Maybe later?

void  setEnds(int x1,int y1,int x2,int y2);
Another method to change4 the location of the line endpoints. Causes redraw.

void setEnds(point* startPt,point* endPt);
Same as above using a pair of points. Again causes redraw.

void setEnds(rect* inRect,slopeType inSlope);
And as before, this is the same as above using a rect. In fact all the methods
above just end up calling this one to do the change.

void drawSelf();
The method used to do the actual line drawing. Vertical and horizontal lines
are anchored from their rectangle's top left corner. (location point)
```

label

https://github.com/leftCoast/LC_GUITools/blob/main/label.h

Putting text on the display. The label class started out as a wrapper around the Adafruit label drawing code written for a calculator display. So it resembles their functionality and uses their actual bitmapped font. But this allows right, center & left justification. You can move it and click on it. assign numeric values to it etc. It also is used as a base for other more interesting label things.

We'll have a look at it's public interface.

Definitions

```
enum {
  TEXT_RIGHT,
  TEXT_LEFT,
  TEXT_CENTER
};
Our justification constants.
```

Constructors

```
label(void);
Default label constructor. Gives rectangle location 16,16, size 16 by 16, text
size of 1, left justified, black text on white background, numeric precision
of 2.

label(const char* inText);
Same as above but stretches the rectangle out to fit the passed in string.

label(const char* inText, int inSize);
Same as above but sets the text size and stretches the rectangle out to fit
the passed in text of that size.

label(int inLocX, int inLocY, int inWidth,int inHeight);
Same as the initial default constructor but changes the rectangle to fit the
inputted value. No text.

label(int inLocX, int inLocY, int inWidth,int inHeight,const char* inText);
Same as above but incuding text.

label(int inLocX, int inLocY, int inWidth,int inHeight,const char* inText,int
textSize);
Same as above but the text size is set to the incoming value.

label(rect* inRect,const char* inText,int textSize=1);
```

So many flavors, the rectangle is set by the incoming rect. Text is set and the text size is set by the incoming value.

label(label* aLabel);
Create a label that's a copy of this passed in one.

Public methods

void setTextSize(int size);
Set the text size, 1,2,3.. - Ends up as multiples of 8 pixals.

void setJustify(int inJustify);
Set left rishgt or center justification.

void setColors(colorObj* tColor);
Set only the forground color of the text allowing it to be display over any background pattern.

void setColors(colorObj* tColor, colorObj* bColor);
Set the foreground and background colors allowing auto updating of changing text.

void setPrecision(int inPrec);
Set how many digits after the decimal point we will display.

void setValue(int val);
Given a signed integer value, display it.

void setValue(unsigned long val);
Given a unsigned integer value, display it.

void setValue(float val);
Given a signed real number value, display it.

void setValue(double val);
Given a higher precision signed real number value, display it.

void setValue(const char* str);
Given a string, copy it and draw it on the screen.

int getNumChars(void);
We want to know how long the string is.. (MINUS THE '\0')

int getViewChars(void);
We want to know how many chars can we display?

void getText(char* inBuff);
We asked above how much you have. Hand it over.

```
int getTextWidth(void);
```
How wide in pixels is our text?

```
int getTextHeight(void);
```
How tall in pixels are the characters?

```
void drawSelf(void);
```
The method that does the custom drawing to the display.

liveText

https://github.com/leftCoast/LC_GUITools/blob/main/liveText.h

Live text is text that can change over time in the background. This could be blinking? Or color changes or fade out. You pretty much have complete control over the line of text as one line. You don't have a char by char control here. Just color over time on a string.

You have your standard x,y, length, height parameters. Then you have framerateMs.

framerateMs sets how long between changes it waits. This is not a perfect clock. Things can effect it like the program doing something else somewhere else. But it does the best it can and really, all this is for is just looks & sizzle. So it should be fine.

How it works?

It's a colorMultiMap run over time. Each point is a time in ms and a color. Then once it's running it changes the text's color each framerateMs as calculated by the color map. Undisturbed ot will run to the end of the color map and stop. If the loop parameter is set to true, it will run over and over.

There is actually more info in the source files.

NOTE : Best to use this over solid colors. Trying to animate on a .bmp background is a flashy mess. Too slow.

Lets have a look at it's pubic interface.

Constructors

```
liveText(int x,int y, int width, int height,int framerateMs=100,bool
loop=false);
Given the values for the rectangle, updates/ms and if you would like it to
loop or not. This creates your liveText object.
```

Public methods

```
void addAColor(int timeMs,colorObj* color);
Add a color by timestamp to the list of colors for this object.

void hold(void);
Stop the animation.

bool isHolding(void);
Returns if this is holding or not.
```

```
void release(bool reset=true);
restart the animation, either from where it stopped or, if reset is true from
the beginning.

void idle(void);
The method that keeps track of time and updates the color when needed.
```

Also remember this class is based on the label class as well as idler, timeObj and colorMultiMap classes. So you have these public methods to use as well.

How about a working example so you can see it running.

https://wokwi.com/projects/447095231921680385

fontLabel

https://github.com/leftCoast/LC_GUITools/blob/main/fontLabel.h

A better looking font you ask? When the default bitmapped font is no longer cutting it for you, fontLabel is available. This is, again, a wrapper for an Adafruit drawing library. The issue with this was that the default font and these smoother fonts don't really line up very well. So, much of this code is concerned with offsets and such forth to take care of these issues. Now, at the time of this writing, only a small subset of the available fonts have been "tweaked". Should be enough to get you going. More should be added as time is available.

NOTE: Also as of this writing. This class is still "young" and I'm thinking it will probably be modified quite a bit from what it is now, in the future.

Lets have a look at the pubic interface.

Definitions

```
#define AFF_SANS_BOLD_24_OB   &FreeSansBoldOblique24pt7b,45,-12
#define AFF_SANS_BOLD_12_OB   &FreeSansBoldOblique12pt7b,24,-6
#define AFF_SANS_BOLD_9_OB        &FreeSansBoldOblique9pt7b,18,-5
#define AFF_SANS_9_OB         &FreeSansOblique9pt7b,18,-5
#define AFF_MONO_12               &FreeMono12pt7b,20,-6
#define AFF_MONO_9                &FreeMono9pt7b,17,-3
```

These #defines group the three variable needed for setFont() method below. They hold the font names, drawObj height and yOffset for lining them up to match the default text from the label class.

Constructors

```
fontLabel(void);
```

Returns a fontLabel with no font set and all the defaults of the original label class.

```
fontLabel(rect* inRect);
```

Same as above, but sets the size and location of the drawing rectangle from the inputted rect.

```
fontLabel(int inX,int inY,int inW,int inH);
```

Same as above, but using values for setting the size and location of the drawing rectangle.

Public methods

```
void setFont(const GFXfont* font,int yOffset);
```

Sets up using a font using the inputted yOffset. Mostly used internally. Probably should just ignore it.

```
void setFont(const GFXfont* font,int inHeight,int yOffset);
```
This is the main method one would use for setting fonts. The idea is to pass in one of the font #defines (Above) to get the font with it's favorite height and y offset.

```
void drawSelf(void);
```
And this is where it all comes together to be drawn on the display. You probably won't interact with this.

textView

https://github.com/leftCoast/LC_GUITools/blob/main/textView.h

The idea behind textView is to have an editable, rectangle of flowing text. It was designed after the classic Macintosh textEdit library. This is still the direction I'd like it to go in.

As of this writing, it works pretty well for scrolling text. Like a terminal. In fact, that's what it's been mainly used for. Wrapped into a handheld and used, with the keyboard library, as a terminal for programming devices. (Or text based games.) It is supposed to be able to deal with selection of text and editing etc. But that has never really been tested. So.. Beware.

Also, as of this writing. It only works with the default Adafruit text. Like the label class. Perhaps at some later date it can be extended for different fonts. we'll see.

So, as always, let's check out the public interface.

NOTE : There is a bunch of other classes and things you will. find in the .h file. I'm not going to list them all here because by and large, you won't interact with them. Later I may add technical descriptions of them.

We'll see..

Definitions

```
enum scrollCom  {
     topOfText,
     endOfText,
     lineAtBottom,
     lineAtTop,
     indexAtBottom,
     indexAtTop,
     upOneLine,
     downOneLine
};
```

Used for calling the scroll function. You'll need these. Mostly it'll be automatic though.

NOTE: I've forgotten what these are supposed to do. I see endOfText used for adding text to the end. That seems to work well. Other than that, nothing else seems to be used.

Constructors

```
textView(int inLocX, int inLocY, int inWidth,int inHeight,eventSet
inEventSet=noEvents);
```

Public methods

```
void calculate(void);
```
Something changed, Repaginate. Why didn't I just call this Repaginate in the first place?

```
void setTextSize(int size);
```
Set the size of the text we are dealing with. Adafruit sizes 1, 2, 3 gigantic.

```
void setTextColor(colorObj* tColor);
```
This sets the color for the text. This sets transparency of the text to true.

```
void setTextColors(colorObj* tColor,colorObj* bColor);
```
This sets the color for the text and the background. And, sets transparency to false.

```
void setScroll(scrollCom choice,int inNum=0);
```
Better look at the scrolling command list.

```
void setText(const char* text);
```
Replace our text buff with a copy of this.

```
void appendText(const char* text);
```
Add this to the end of our text.

```
void appendText(int inVal);
```
I know, its an integer. Make it text. Add it to the end.

```
void appendText(char inChar);
```
Add just a char to the end.

```
void insertText(int index,const char* text);
```
Stick a NULL terminated substring in at this index.

```
void deleteText(int startIndex,int numChars);
```
Remove some text from middle, beginning? End?

```
char* seeText(void);
```
Hand back a pointer to the actual text buffer. DON'T MESS WITH IT!

```
void drawSelf(void);
```
Its draw time!

bargraph

https://github.com/leftCoast/LC_GUITools/blob/main/bargraph.h

bargraph

bargraph contains two classes, the first is bargraph. bargraph is a base class drawObj that does the math to calculate the two rectangles necessary to draw a bar graph. The foreRect, being the bar itself. And backRect, the background where the bar is not currently drawn. This class adds setValue() to a drawObj. setValue() takes a value along with limiting min & max values. From these it calculates the two rectangles.

Definitions

```
enum orientation {
     bottomUp,
     leftRight,
     topDown,
     rightLeft
};
```

Constructors

```
bargraph(rect* inRect,orientation inOrientation);
Given a rect to set location and size along with an orientation, this will
create a bargraph object.
```

Public methods

```
void setValue(float inValue,float min,float max);
Given any type of value along with min& max limits, this will calculate the
foreRect and backRect rectangles in local screen coordinates for drawing.
Local meaning that they work even if this is a sib object of an drawObj group.
```

Public variables

```
rect         backRect;
rect         foreRect;
```

colorBargraph

colorBargraph extends bargraph and adds a colorMultiMap to hold value,color data pairs. This allows the bar to be drawn in different colors as a function of value. Also makes for a really good example on how to extend bargraph for your own purposes.

Ans example of color colorBargraphs :

https://wokwi.com/projects/447194632033400833

Lets have a look at colorBargraph's public user interface.

Constructors

```
colorBargraph(rect* inRect,colorObj* inBackColor=&black,orientation
inOrientation=bottomUp);
```

Given a rectangle for size and location. A back color for filling in where the bar has not been drawn. And nn orientation for direction. This creates a bar graph for you with an empty multiColorMap built in.

Public methods

```
void addColor(double inX, colorObj* color);
```

Adds a value, color point to your drawing. You can add as many value,color points as you like.

```
void clearMap(void);
```

Like it says, dump out all the value,color points in the map. Leaving you with an empty color mp.

```
void setValue(float inVal);
```

Write a value to the bar graph for it to show on your display. This would be in the same units as the points added to the color map.

```
void drawSelf(void);
```

Grabs the two rectangles calculated by the inherited bargraph, and draws them to the display.

slider

https://github.com/leftCoast/LC_GUITools/blob/main/slider.h

Ability to draw an analog slider, either vertically or horizontally, on a touchscreen. Interact with it, read values from it, write values to it. Those things sliders do.

There are two classes in the file. knob & slider. To create a slider, decide if you want it horizontal or vertical? How many pixels will you be able to slide it and how wide will the knob be? The knob is defaulted at 8 pixels thick. After all this verbiage what you are really looking for is the swath that the slider knob will sweep through. Height & width. This gives the rectangle for your slider. Well, and orientation. Horizontal is the default.

You can ignore the knob class for now. You create what you want for the display by creating a instance of slider, with the rectangle it should slides through. Then call the setColors() method to set your knob color and you background color. There will be a hardcoded black outline around the knob's path.

Here's an example for you to play with :

https://wokwi.com/projects/447391581409518593

knob

The knob is a color rect. So you can do any of the color rect calls on it. The slider is a drawGroup but you've not run across those yet. Deal with that later.

And here's the public interface of knob.

Constructors

```
knobObj(int length,int thickness,int inRange,bool vertical);
Create's a knob of these dimensions, range and orientation.
```

Public methods

```
void setColors(colorObj* inForeColor,colorObj* inBackColor);
Sets the colors for the knob and the background color for erasing the knob.

void eraseSelf(void);
Yes this erases the knob by covering it with the background color.

void setPos(point* inPt);
Internal call to locate the knob. This basically is how it follows your
finger.
```

```
float getValue(void);
Returns the mapped value of the position of the knob. Default is 0..100.

void setValue(float value);
If you want to set the position of the knob without touching it? Use this. The
default input range is 0..100.
```

slider

Now, with that all being said, to start off you can ignore knob and focus on slider. This is the one that creates everything for you and also the one you will interact with. Lay out the recangle you want to slide in and use that to create your slider.

We'll have a look at slider's public interface.

Constructors

```
slider(int x,int y,int width,int height,bool inVertical=false);
Given the rectangle that the knob will slide though along with if it should
slide vertically or horizontally. This creates your slider object.
```

Public methods

```
void setColors(colorObj* inForeColor,colorObj* inBackColor);
Sets the knob & background colors.

float getValue(void);
Just in case you were wondering. returns the mapped 0..100 value of the
location of the slider knob.

void  setValue(float value);
Great for pre-loading your slider value.

void  drawSelf(void);
This manages and draws all the bits hat make up the knob & slider.

void  doAction(event* inEvent,point* localPt);
Used internally for dealing with sliding and finger drags etc.
```

LC_GUITools (detailed)

https://github.com/leftCoast/LC_GUITools/tree/main

Well, if you got this far I guess your serious. We'll start this off with the glue code for display hardware. the displayObj class.

displayObj

https://github.com/leftCoast/LC_GUITools/blob/main/displayObj.h

displayObj holds the list of command a display needs to be able to perform to be used by the Left Coast GUI code. It was originally modeled after AdafruitGFX so it will be similar to that.

You choose a display to use. Create a class that inherits displayObj. Fill in the methods that displayObj needs filled out to function and Viola! You can use your hardware as a display for the all of the LC_GUI code. Well, at least the methods that you can fill out. The more the better.

Example of creating a display to use.

```
screen = (displayObj*) new adafruit_1947(DSP_CS,-1);    // Create screen.
if (screen) {                                           // We got one?
    if (screen->begin()) {                              // begin() ok?
        screen->setRotation(PORTRAIT);                  // For handhelds.
        setupScreen();                                  // Build screen.
    }                                                   //
}                                                       //
```

So lets see the public interface.

Definitions

```
#define PORTRAIT      0  // Default narrow.
#define LANDSCAPE     1  // Default wide, clockwise.
#define INV_PORTRAIT  2  // Upside down narrow, clockwise.
#define INV_LANDSCAPE 3  // Upside down wide, clockwise.
```

Constructors

```
displayObj(bool inGraphic,bool inColor,bool inTouch,bool inSD,bool
inReadable);
Given the input parameters this will create your display object. As stated
above, this as a base class for each model of display we use. When used, the
```

derived constructor will need to assign the address to your display object to the variable "screen". (See above)

Public methods

```
void pushOffset(int x, int y);
```
When draw objects are in group their coordinates x,y are local to that group. Not not global to the display. This is the call that sets up that translation.

```
void popOffset(int x, int y);
```
When we are no longer using a group, could be a sub group of a group. This brings our coordinated up one level.

```
int gX(int lX);
```
When drawing to a display we need global coordinates, Here's where they come from.

```
int gY(int lY);
```
Same as above except for y.

```
int lX(int gX);
```
When you get a touch they come in on global coordinates. Each drawObj only knows it's local coordinates so this is used for mapping touches locally.

```
int lY(int gY);
```
Same as above except for y.

```
point lP(point* gP);
```
global to Local mapping but reading both x,y as a global point.

```
point gP(point* lP);
```
local to global mapping but reading x,y as a local point.

```
int width(void);
```
Returns the width of the display in pixels.

```
int height(void);
```
Returns the height of the display in pixels.

```
bool begin(void);
```
Filled out by the derived display driver to do initialization of the hardware etc. returns success or failure of the operation.

```
void startWrite(void);
```
If the hardware has a startWrite() you can call, it goes here.

```
void endWrite(void);
```
Same as above for endWrite().

```
void setRotation(byte inRotation);
```
If the display can rotate, using the definitions above this is where the code goes to do that.

```
void setTextColor(colorObj* tColor);
```
Sets the drawing color of the text. This would be for the default Adafruit text. This also sets text with no background drawing.

```
void setTextColor(colorObj* tColor,colorObj* bColor);
```
Sets the drawing color of the text. And the background color for the text. This would be for the default Adafruit text. This sets text with background drawing.

```
void setTextSize(byte inSize);
```
This sets the text size for the default Adafruit font. Multiples of 8 pixels?

```
void setTextWrap(bool inWrap);
```
You know what? I never use this except to turn it off. This will wrap your default Adafruit text the full width of your screen and overwrite anything that's on there. Useless!

```
void setFont(const GFXfont* font);
```
For setting the more advanced Adafruit fonts that come hardcoded with their size.

```
void setCursor(int inX,int inY);
```
Sets the location of where text writing will start.

```
int getCursorX(void);
```
Get the x location of where the text writing will start. These two are not always used.

```
int getCursorY(void);
```
Get the y location of where the text writing will start.

```
rect getTextRect(const char* inText);
```
When using the advanced fonts this passes back a best guess rectangle of a c string. Not always perfectly accurate.

```
void drawText(const char* inText);
```
After setting up all your text parameters, this draws it to the display.

```
void  drawChar(int x,int y,char inChar,colorObj* fColor,colorObj* bColor,int size);
```
This goes to this location and draws a character of this color right there.

```
void fillScreen(colorObj* inColor);
```
Fill the entire screen with this color.

```
void fillRect(int locX,int locY,int width,int height, colorObj* inColor);
```
Fills a rectangle with a solid color.

```
void fillRect(rect* inRect,colorObj* inColor);
```
Same as above.

```
void drawRect(rect* inRect,colorObj* inColor);
```
Outlines a rectangle in this passed in color.

```
void drawRect(int locX,int locY,int width,int height, colorObj* inColor);
```
Same as above.

```
void fillRoundRect(int locX,int locY,int width,int height,int radius,colorObj*
inColor);
```
Fills a rounded corner rectangle with the passed in cormner radius and color.

```
void drawRoundRect(int locX,int locY,int width,int height,int radius,colorObj*
inColor);
```
Draws and out line around a rounded rectangle using this passed in color.

```
void fillRoundRect(rect* inRect,int radius,colorObj* inColor);
```
Same as above. But this is just a call to the above.

```
void drawRoundRect(rect* inRect,int radius,colorObj* inColor);
```
Same as above. But this is just a call to the above.

```
void drawCircle(int locX,int locY,int inDiam, colorObj* inColor);
```
Of course this draws a circle at loc x,y of diameter inDiam.

```
void drawCircleHelper(int locX,int locY,int inRad,byte corner,colorObj*
inColor);
```
I couldn't tell you what this does. It's an adafruit thing, so I left it in. I wonder if I should pull it out?

```
void fillCircle(int locX,int locY,int inDiam, colorObj* inColor);
```
Like above except draws the circle as opposed to filling it in.

```
void drawTriangle(point* pt0,point* pt1,point* pt2,colorObj* inColor);
```
Draws the triangle around the three inputted points of the inputted color.

```
void fillTriangle(point* pt0,point* pt1,point* pt2,colorObj* inColor);
```
Drwas a solid triangle between the inputted points of the inputted color. Very handy for doing 3D modeling.

```
void drawVLine(int locX,int locY,int height,colorObj* inColor);
```
Draws a vertical single pixel line. Supposedly very fast? The line is drawn DOWN from it's x,y location. One pixel short of height.

```
void drawHLine(int locX,int locY,int width,colorObj* inColor);
```
Draws a horizontal single pixel line. Also Supposedly very fast. The line is drawn RIGHT from it's x,y location. One pixel short of width.

```
void drawLine(int locX,int locY,int locX2,int locY2,colorObj* inColor);
```
Draw one pixel line between these two points of this color.

```
void drawLine(point* startPt,point* endPt,colorObj* inColor);
```
Same as above but passing in points. Also this is just a call to the method above.

```
void drawPixel(int locX,int locY,colorObj* pColor);
```
Make the pixel at x,y the color passed in.

```
void blit(int locX,int locY,bitmap* inBitmap);
```
Draw this bitmap at the passed in location.

```
void drawPixelInvert(int x,int y);
```
If you have a readable screen, you can read a pixel then draw it's inverted self there.

NOTE : OLED displays will burn out of run 100% all the time. I've cooked a few. These next three methods allow you to set up a screensaver by dimming the screen to black as a percentage.

```
void setPercBlack(float percent);
```
Set the percentage of black to mix in with all of the drawing to the display.

```
float getPercBlack(void);
```
Return the percentage of black currently being mixed into the display's drawing.

```
word dim16(colorObj* inColor);
```
Used in place of the standard "inColor->getColor16()" calls used to translate from color objects to the two byte color16 values.

```
void frameRectInvert(int x,int y,int width,int height);
```
If you have a readable screen, you can read a rectangle of pixels then draw their inverted self there.

```
void fillRectGradient(int inX,int inY,int width,int height,colorObj*
startColor,colorObj* endColor,bool rising=true,bool vertical=true);
```
These next three are all filed out in this object as calls to other methods. This one fills a rectangle with a color gradient either vertical or horizontal.

```
void fillRectGradient(rect* inRect,colorObj* startColor,colorObj*
endColor,bool rising=true,bool vertical=true);
```
Same as above, calls the one above.

```
void fillScreenGradient(colorObj* startColor,colorObj* endColor,bool
rising=true,bool vertical=true);
```
This is the same as the ones above but fills the entire screen with a gradient.

```
point getPoint(void);
```
If the display is a touch screen, this returns the last point touched on the display.

```
bool touched(void);
```
If the display is a touch screen, this returns if it is being touched.

```
 bool isGraphic(void);
```
Returns if the display is graphic. Seems sort of a silly call now. But whatever.

```
bool isColor(void);
```
Returns if the display is a color display.

```
bool hasTouchScreen(void);
```
Returns if the display has a touch screen.

```
bool hadSDDrive(void);
```
Returns if the display has an SD card. Also somewhat pointless seeing as it doesn't seem to effect anything.
Except, using SD cards on display that have NOT been initalized can lead to the SD cards to get kinda' wacky.

```
bool canRead(void);
```
Returns whether a display can return what it's displaying.

So all these calls are what the display offers to the drawing environment. And what uses these calls, drawObject.

But before we delve into drawObj let's have a look at baseGraphics where much of this stuff originates from.

baseGraphics

https://github.com/leftCoast/LC_GUITools/blob/main/baseGraphics.h

baseGraphics is the foundation of all of this drawing stuff. This defines two dimensional integer points. Good for graphics. Some public point functions. Direction, distance etc. And the rect class. Used nearly everywhere.

Public user interface..

point

Definitions

```
struct point {
    int x;
    int y;
};
```

Public functions

```
int xDistance(point ptA, point ptB);
Return difference in the x direction between two points.

int yDistance(point ptA, point ptB);
Return difference in the y direction between two points.

float distance(point ptA, point ptB);
Calculate and return the distance between two points. We're talking Trig here.

float angle(point ptA, point ptB);
Radians, ptA is center point. east would be zero radians.

point rotate(point ptA,float angle);
Rotate the inputted x,y location around the x,y axis (0,0) angle, radians from
East. Return the resulting point.
```

NOTE: This one is all integer based. So it's not terribly accurate.

rect

Definitions

```
enum rectPt {
    topLeftPt,
    topRightPt,
    bottomLeftPt,
    bottomRightPt
```

```
};
```

Constructors

```
rect(void);
rect(rect* inRect);
rect(int inX, int inY, int inWidth,int inHeight);
```

Public methods

```
void  setLocation(int inX, int inY);
```
Set the x,y location to these values.

```
void setSize(int inWidth,int inHeight);
```
Set the size to these values.

```
void setRect(rect* inRect);
```
Got a rect? Make this one the same.

```
void setRect(point* inPt1,point* inPt2);
```
Or, set size and location to span these two points..

```
void setRect(int inX, int inY, int inWidth,int inHeight);
```
Or, set size and location to these values.

```
void insetRect(int inset);
```
Inset all sides by this much. Or expand, if negative.

```
void addRect(rect* inRect);
```
Become the rect that spans ourself and this incoming rect.

```
int maxX(void);
```
Where's our last pixel?

```
int maxY(void);
```
Same as above, but in the Y direction.

```
int minX(void);
```
Where's our first pixel?

```
int minY(void);
```
Same as above, but in the Y direction.

```
bool inRect(int inX, int inY);
```
Is this point in us?

```
bool inRect(point* inPoint);
```
Is this point in us?

```
point getCorner(rectPt corner);
Pass back the corner point.

bool overlap(rect* checkRect);
Is that rect touching us?

bool isSubRectOf(rect* checkRect);
Are we contained in that rect?

bool isSameAs(rect* checkRect);
Are we identical to this rect?

void startBoundsRec(void);
Clear out the rect and start recording points to it.

void addBoundsPt(point* inPt);
Add in data as a point.

void addBoundsPt(int x,int y);
Add in data as x,y values.
```

drawObj (detailed)

https://github.com/leftCoast/LC_GUITools/blob/main/drawObj.h

As stated before draw object is the base of all things that can be drawn on the screen. It's a rectangle area with a x,y top left corner location. The draw objects, view manager and event manager work together along with the displayObj to run the GUI. The drawObj keeps track of locations and drawing commands, the displayObj does the actual drawing and keeps track of coordinates. The event manager gathers and interprets the events. And the view manager runs the show. All in idle time. So basically, the entire thing is transparent to the user.

The drawObj file has a few classes in it. well start at the top with drawObj.

drawObj

So much has already been said about drawObj. So let's start with the public interface.

Definitions

```
enum eventSet {
    noEvents,    // No events accepted.
    touchLift,   // Touch and lift events accepted.
    fullClick,   // Just full click events accepted.
    dragEvents,  // Drag dragBegin, dragOn and liftEvent accepted.
    touchNDrag   // Same as above but adding touchEvent accepted.
};
```

These are the different type of event sets that drawObjects can respond to.

Constructors

```
drawObj();
```

Default object created with location of 16,16 and size of 16 by 16. will be drawn as a black rectangle with white outline. No events accepted.

```
drawObj(rect* inRect,eventSet inEventSet=noEvents);
```

Size, location and event set can be selected.

```
drawObj(int inLocX, int inLocY, int inWidth,int inHeight,eventSet
inEventSet=noEvents);
```

Same as above but using values for size and location.

Public methods

```
bool wantRefresh(void);
```

Called by manager to see if I need to draw now.

```
void setNeedRefresh(bool=true);
```

```
Called to cause a redraw. (Used a LOT)
void aTouchAbove(void);
I've changed. Going to draw. Everyone above, draw too.

void setLocation(int x,int y);
Changed our mind, move it over here..

void draw(void);
Call this one. Don't inherit this one.

void eraseSelf(void);
Mostly you can ignore this one. Used for animation.

void drawSelf(void);
Inherit this one and make it yours.

void setThisFocus(bool setLoose);
We are either getting or loosing focus.
```

NOTE: Focus is offered just in case you need to track what thing on the display is currently active. All it is, is a pointer to a draw object that you can query. When drawing, you could show that it has the focus. You could maybe read or write information to or from the focused object. It's up to you. It's just a pointer, that you control, that can only point to one item at a time, or to NULL. That's all it is.

```
bool haveFocus(void);
Do we have focus or not?

void setEventSet(eventSet inEventSet);
Want to change our event set on the fly?

eventSet getEventSet(void);
Want to see what this guys even set is?

bool acceptEvent(event* inEvent,point* localPt);
Is this event for us?

void doAction(void);
Override me for action!

void doAction(event* inEvent,point* localPt);
Special for them that drag around.

void setCallback(void(*funct)(void));
```

Or use a callback.

viewMgr

Here we have the management of the screen. Get the clicks Find drawObjs to act on the clicks. And tell the objects when to redraw themselves. "viewList" is the global that does all these things during idle time.

Lets have a look at the viewMgr's public interface. (And globals)

Constructors

```
viewMgr(void);
```
No need to deal with this. It will construct itself on boot up.

Public methods

```
void addObj(drawObj* newObj);
```
This call is used mostly during setup() when assembling drawObjs onto the display.

```
void dumpDrawObjList(void);
```
If for some reason you need to clear out and recycle all your draw objects? This'll do the trick for you.

```
bool checkEvents(event* theEvent);
```
Returns if there is an even waiting to handle. IE. See if a drawObj will accept it.

```
void checkRefresh(void);
```
Runs up the list from the bottom seeing if anyone needs to be redrawn.

```
int numObjects(void);
```
Returns the number of drawObjs the list is made of.

```
drawObj* getObj(int index);
```
Finds the Nth starting at zero object on this list. Returns a pointer to it. If not found, returns NULL.

```
bool viewMgr::findObj(drawObj* theObj)
```
Is this drawObj one of ours? Returns true if we can fid this drawObj on our list.

```
drawObj* theList(void);
```
Returns the first item on the list. NULL if none.

```
void touchAllAbove(drawObj* fromMe);
```

Starts at the top of the list and touches every drawObj 'till it finds "fromMe". Causing them all to redraw on the next call to checkrefresh. Used to erase modal dialog and alert boxes.

```
void idle(void);
```
If there's an event, it deals with the even. But in all cases it checks to see if anyone needs to be redrawn.

Public globals

```
viewMgr      viewList;
```
Our global GUI manager.

```
drawObj*     theTouched;
```
Who's accepted a finger touch on the screen?

```
drawObj*     currentFocus;
```
Focus goes hand in hand with view management.

```
bool         drawing;
```
This is ALWAYS true. WTF?!

Public functions

```
void         setFocusPtr(drawObj* newFocus);
```
Anyone can set focus by calling this function.

Ok, that gives a pretty good overview of the viewMgr. There is more information in the source files .h & .cpp. Next we move on to..

drawGroup

Now things start getting a bit recursive..

Draw group is a drawObj that is **also** a viewMgr. This is the base of all groups, lists, popup windows etc. Or like the comments say. "At least that's the hope."

Now the global viewList is still the base of all the drawing. Just now it can pass drawing commands an everything else to a drawGroup and that is now a sub list of the viewlist. This becomes very important later as the bases of , not only groups and lists, but actual code swapping of different entire "applications". But we'll go into all that when we get to that.

NOTE : Before taking any action with a sub object, the drawGroup pushes its location onto the displayObj. This allows all sub objects to work on local coordinates.

Here's the drawGroup public interface, Similar to the viewMgr one.

Constructors

```
drawGroup(rect* inRect,eventSet inEventSet=noEvents);
Create a drawGroup at this location and size with these event sets accepted.

drawGroup(int x,int y,int width,int height,eventSet inEventSet=noEvents);
Sme as above only using values as opposed to a rect.
```

NOTE : These constructor event sets are just for the drawGroup objects themselves. When an event comes in, regardless of these constructor settings, the child objects get to check the events first. Then, if the group accepts events, it gets a shot at handling it.

Public methods

```
bool checkGroupRefresh(void);
Return if anyone in our group needs to be redraw.

void setLocation(int x,int y);
Move the entire group over to here.

void setGroupRefresh(bool refresh=true);
Either set or clear the need to redraw everyone in our group. But not
ourselves.

bool wantRefresh(void);
Return if anyone of the "kids" need a redraw or possible we ourselves do?

void setNeedRefresh(bool refresh=true);
Sets or clears the need to redraw for ourselves, and all the "kids".

bool acceptEvent(event* inEvent,point* localPt);
Returns weather ourselves or anyone on our list will accept this event.

void addObj(drawObj* newObj);
Add a drawObj to our list.

void draw(void);
The Drawing command you typically don't inherit. Causes the list to draw all
the "kids" that need to be redrawn.

void idle(void);
We use this block idling. Odd huh? We get all our idle time from the viewMgr.
Having our own idle() method messes things up. So we inherit it, and do
nothing in it. Other things that extend us could use it. Or that's what the
notes say..
```

That kinda' wraps up drawGroup for now. Does a lot but it mostly all o automatic. Next we extend drawGroup to..

drawList

drawList extends drawGroup in that is lines up all it's items in a row. Either vertically or horizontally. drawList makes the assumption that all the items added to it are of the same height. Not that adding different height items would break anything. It's just that the math for aligning and visibility calculations wouldn't work out anymore.

NOTE : Speaking of calculations. As of this writing, it seems that setPositions(), lastY(), isVisible() and showItem() are all written assuming that all lists are vertical. I'm betting they were developed for some project(s) that only had vertical lists and never got fully filled out. This needs to be looked into.

So as always, let's have a look at the public interface.

Constructors

drawList(rect* inRect,eventSet inEventSet=noEvents,bool vertical=true);
Same as the drawGroup contructors above with the addition of the vertical or not vertical selection.

drawList(int x, int y, int width,int height,eventSet inEventSet=noEvents,bool vertical=true);
Same as above except using values for location and size.

NOTE: As with the drawGroup class. The passed in event sets are just for this class itself. The child items have their own event sets, independent from this. So they all get to see and respond to the events coming in.

Public methods

void addObj(drawObj* newObj);
As always, the method for adding a new object to the list. In addition, this method aligns the new object with the rest of the list.

void setPositions(int offset=0);
If you have a vertical list, sigh.. , and, say delete an item. This will relocate all the items to fill the hole. Offset is the amount of pixels to leave as a gap at the top of the list.

```
int lastY(void);
```
Again, for vertical lists. Calculate and return the last possible y position an item can have to be viewable.

```
bool isVisible(drawObj* theItem);
```
Pass in any item on a vertical list, and this returns if it is visible or not. (Falls inside the lists's rectangle.)
```
void showItem(drawObj* theItem);
```
If this item is not showing, scroll the list so that it is visible. Assuming of course it is a vertical list.

```
drawObj* findItem(point* where);
```
Given a point return the item on the list that this falls in. NULL if none are found.

eventMgr

https://github.com/leftCoast/LC_GUITools/blob/main/eventMgr.h

eventMgr

The event manager takes over and manages all of the touch screen events. For those displayObjs that have touch screens. You should never have to interact with it at all. Just turn it on and let it get on with doing it's thing. Granted, the displayObj you would like to use, needs to have it's touch event methods correctly filled in and functioning.

What the event manager does is gather up the touches, lifts and drags on the display. Grab as much data about them as possible. Where the touch went down, when, is it moving? How fast? What direction? Then it packages all this up into an event object and passes it along to the viewMgr to broadcast to the drawObjs.

Let's have a look at the evenMgr's public interface.

Definitions

```
#define  DRAG_TIME    150
How many milliseconds before it goes from a click to a drag?
```

```
#define  DRAG_DIST    10
If the finger moves further than this, we'll call it a drag.
```

```
enum eventType {
     nullEvent,
     touchEvent,
     liftEvent,
     dragBegin,
     dragOn,
     clickEvent
};
Different types of events we can supply.
```

```
struct event {
     eventType         mType;
     unsigned long     mTouchMs;
     unsigned long     mLastMs;
     unsigned long     mNumMs;
     point             mTouchPos;   // Where the touch began.
     point             mLastPos;    // Where the touch is NOW.
     int               mXDist;
     int               mYDist;
```

```
    float           mDist;
    float           mXPixlePerSec;
    float           mYPixlePerSec;
    float           mPixalPerSec;
    float           mAngle;
};
```
The event strut holds all the information we can glean out of finger touches and drags.

Constructors

eventMgr(void);
Creates our event manager.

Public methods

void begin(void);
Basically just calls hookup() so it can start idling.

void flushEvents(void);
Empties our list of pending events.

bool haveEvent(void);
Returns if we have any pending events to hand off or not.

event getEvent(void);
Returns the next waiting event from our list.

bool active(void);
Returns if we have pending events or we are currently touched.

void addEvent(eventType inType);
Add an event to our queue of pending events.

void push(eventObj* newEventObj);
Used internally for doing the actual adding of an event to the queue. Used by the addEvent() method.

eventObj* pop(void);
Also used internally but for removing an event from the queue.

void idle(void);
As usual, runs the machine in the background.

That should cover the event manager. The last bits are the different displays that have been assimilated. Starting with..

LC_Adafruit_684

https://github.com/leftCoast/LC_Adafruit_684

The Adafruit #684 is a 96x64 color pixel OLED display. It comes complete with a built in SD drive. Both the display and SD drive are controlled by the SPI bus. Both have chip select connections. There is no touch ability with this display.

Pretty much the code is just a filled in version of the deviceObj base class. The shared SPI bus pin numbers are sourced from the LC_SPI library. Make sure the processor you are using is setup in there.

Being an OLED display, the colors for drawing are all passed through the dim16() method. This is so the end user can implement a screen saver feature if desired. See displayObj for more info on the dim16() method and its use.

NOTE : As stated before. You really don't want to run an OLED full time, as they have a limited life span. The have an issue with "burn in" cooking the pixels. Dimming them down to black, when not in use, will stop the burn in. Even if they are still technically "drawing".

The important part for the user is how to set up the display for use. You will need your choosen processor setup in LC_SPI. Chip select pins for at least the display. Most likely both display and the SD card reader. And a reset pin number.

Constructor

```
adafruit_684_Obj(byte inCS,byte inRST=-1);
Your consructor. It'll want chip select & reset pin numbers for the display.
The SD card is setup separately by your SD setup code.
```

Typical setup procedure for the display.

```
bool success;
success = false;
screen = (displayObj*) new adafruit_684_Obj(displayCS,displayReset);
if (screen) {
      success = screen->begin();
}
```

LC_Adafruit_1431

https://github.com/leftCoast/LC_Adafruit_1431

The Adafruit #1431 is a 128x128 color pixel OLED display. It comes complete with a built in SD drive. Both the display and SD drive are controlled by the SPI bus. Both have chip select connections. There is no touch ability with this display.

Pretty much the code is just a filled in version of the deviceObj base class. The shared SPI bus pin numbers are sourced from the LC_SPI library. Make sure the processor you are using is setup in there.

Being an OLED display, the colors for drawing are all passed through the dim16() method. This is so the end user can implement a screen saver feature if desired. See displayObj for more info on the dim16() method and its use.

NOTE : As stated before. You really don't want to run an OLED full time, as they have a limited life span. The have an issue with "burn in" cooking the pixels. Dimming them down to black, when not in use, will stop the burn in. Even if they are still technically "drawing".

The important part for the user is how to set up the display for use. You will need your chosen processor setup in LC_SPI. Chip select pins for at least the display. Most likely both display and the SD card reader. And a reset pin number.

Constructor

```
adafruit_1431_Obj(byte inCS,byte inRST=-1);
Your constructor. It'll want chip select & reset pin numbers for the display.
The SD card is setup separately by your SD setup code.
```

Typical setup procedure for the display.

```
bool success;
success = false;
screen = (displayObj*) new adafruit_1431_Obj(displayCS,displayReset);
if (screen) {
      success = screen->begin();
}
```

LC_Adafruit_1947

https://github.com/leftCoast/LC_Adafruit_1947

The Adafruit #1947 is a 240x320 color capacitive touchscreen. This display was originally designed to be a "shield" plugin for the Arduino UNO. For this reason it has two constructors one hardcoding the default pin set to match the UNO when pugged in. The other lets you choose your chip select and your reset pin numbers. This display takes care of selecting the "correct" pins for the shared SPI bus for each processor. With the exception of the D/C pin and that one comes from the LC_SPI library.

Being a TFT display, the screen is illuminated by a backlight. This can be controlled by you by modifying a jumper and hooking a wire from it to a PWM output pin. Adafruit has instructions on setting this up on their web sight.

https://www.adafruit.com/product/1947

The important part for the user is how to set up the display for use. You will need your chosen processor setup in LC_SPI. (At least for the D/C pin. Chip select pins for at least the display. Most likely both display and the SD card reader. And a reset pin number.

The big difference in the code for this display is that it has the ability to mask drawing added to it. There is a global object gMask that controls this. You will can look at mask.h to see how this works.

https://github.com/leftCoast/LC_GUITools/blob/main/mask.h

The constructor for adafruit_1947.

Constructors

```
adafruit_1947(void);
If you are running it wired up like a shield? This'll do the trick.

adafruit_1947(byte inCS,byte inRST=-1);
Not wired like a shield? This one is the one you will need to use. It'll want
chip select & reset pin numbers for the display. The SD card is setup
separately by your SD setup code.
```

Typical setup procedure for the display.

```
bool success;
success = false;
```

```
// Choose constructor Default or Shield..
// screen = (displayObj*) new adafruit_1947(displayCS,displayReset); // Default.
// screen = (displayObj*) new adafruit_1947(); // Only if wired like a shield.

if (screen) {
      success = screen->begin();
}
```

LC_DFRobot_0995

https://github.com/leftCoast/LC_DFRobot_0995

The DFRobot0995 is a 172x320 color TFT. I wasn't happy with the original driver code from DF Robot. Just seemed like they wasted the fine resolution of this display. I was able to get it working with the Adafruit ST7789 drivers. And that worked way better! Also very easy to interface to, seeing as it was based on Adafruit GFX.

The Adafruit-ST7735-Library

https://github.com/adafruit/Adafruit-ST7735-Library

As of this writing, the Left Coast code for this display is still kinda' in development. The big change for this display is that it has been the testbed for using Adafruit's nicer font drawing library. This library has the getTextBounds() method in it. Works ok. It does not yet have the image mask code in it. This is also a development thing.

The fontLabel class has the ability to use this new font code. For information on how all that works, one should have a look there.

https://github.com/leftCoast/LC_GUITools/blob/main/fontLabel.h

Also this display has a backlight wire pad on the board. But I've not had any luck finding how it should be used to control the backlight.

So, in summery there is two methods to know about for this display the constructor and and the new getTextRect() method.

Constructors

```
DFRobot_0995_Obj(byte inCS,byte inRST);
This uses the hardware SPI bus. You willl need to supply the chip select and
reset pin numbers.
```

Public Methods

```
rectgetTextRect(const char* inText);
Returns a rectangle that this c string will fit in. Pinned at 0,0. Used for
drawing calculations.

For setting up this display to use.

bool success;
success = false;
```

```
screen = (displayObj*) new DFRobot_0995_Obj(displayCS,displayReset);
if (screen) {
      success = screen->begin();
}
```

LC_Adafruit_2050

https://github.com/leftCoast/LC_Adafruit_2050

The Adafruit 2050 is a 320x480 resistive touch color TFT. As of this writing, the Left Coast code for this display is still kinda' in development. The big change for this display is that it has been one of the testbeds for using Adafruit's nicer font drawing library. This library has the getTextBounds() method in it. Works ok. It **does** have the image mask code in it. This is also a development thing. Here is a link for more info. on the mask.

https://github.com/leftCoast/LC_GUITools/blob/main/mask.h

The fontLabel class has the ability to use this new font code. For information on how all that works, one should have a look there.

https://github.com/leftCoast/LC_GUITools/blob/main/fontLabel.h

Also at this time of writing, the touch code has not been written. Not even wired up yet. Been busy. So, unless there's change don't expect clicks and drags from this one for awhile.

In summery, there are two methods to know about for this display the constructor and and the new getTextRect() method.

Constructors

```
adafruit_2050(byte inCS,byte inRST);
This uses the hardware SPI bus. You will need to supply the chip select and
reset pin numbers.
```

Public Methods

```
rect getTextRect(const char* inText);
Returns a rectangle that this c string will fit in. Pinned at 0,0. Used for
drawing calculations.

For setting up this display to use.
```

```
bool success;
success = false;
screen = (displayObj*) new adafruit_2050(displayCS,displayReset);
if (screen) {
      success = screen->begin();
}
```

Offscreen drawing

https://github.com/leftCoast/LC_GUITools/blob/main/offscreen.h

That's right, we setup a virtual display that you can use to write to offscreen bitmaps in your processor's RAM. You'll want a lot or RAM for this. Bitmaps get really large quickly.

offscreen

offscreen is the display you will interact with. You will have to setup a bitmap in RAM for it to draw to. Then there are beginDraw() and endDraw() methods that switch your "screen" global to draw to your RAM display and then back to you "real" display.

This display is derived from displayObj just like all the others. we'll have a look at its public interface that is unique to it.

Constructors

```
offscreen(void);
Creates your offscreen drawing environment.
```

Public methods

```
bool begin(void);
Does nothing at the moment besides return true

void beginDraw(bitmap* inMap,int inOffsetX=0,int inOffsetY=0);
Given an allocated bitmap and x,y offsets this will cause the global object
"screen" to now draw into this bitmap.
```

NOTE: When drawing offscreen, imagine you place a rectangle on your "real" display around the part you want to capture. The x,y location of this rectangle is passed in as the offsets.

```
void endDraw(void);
This ends the drawing session and resets your global object "screen" to your
original display.
```

mapDisplay

There is also a hardware emulator based on AdafruitGFX called mapDisplay. (Short for Bitmap Display). The mapDisplay is used internally, so we'll not bother with it much here. Just know it's there in the background making drawings for you.

bitmap
https://github.com/leftCoast/LC_GUITools/blob/main/bitmap.h

bitmap is the thing in RAM we draw to. It's also the thing in RAM that we can blit to a "real" display. The speed increase for a image from RAM as opposed an Sd card? Is astonishing! Also for doing complex drawing, it's invaluable. Hence why all this stuff to make it possible.

Lets have a look at bitmap's public interface.

Constructors
```
bitmap(void);
Give back an initialzed but empty bitmap.

bitmap(int width,int height,bool alpha=false);
Give back a fully functonal bitmap. If it can get the RAM that is.

bitmap(bitmap* aBitmap);
Create ourselves as a clone of another bitmap.
```

Public methods
```
bool setSize(int width,int height,bool alpha=false);
Copy down the bitmap size then, if possible, allocate the buffer.

void clearMap(void);
Dumps everything. object is still valid, just empty.

bool getHasMap(void);
Returns whether we have an allocated map.

int getWidth(void);
Returns width of map.

int getHeight(void);
Returns height of map.

void setColor(int x,int y,colorObj* aColor);
Set this location to a color. From a colorObj.

void setColor(int x,int y,RGBpack* aColor);
Set this location to a color. From a RGBpack.

void setAlpha(int x,int y,byte alpha);
Set this location's alpha channel.

colorObj getColor(int x,int y);
Grab a color object out of RAM. We need this fast so, no sanity checking.
```

```
RGBpack getColorPack(int x,int y);
Grab a color pack out of RAM. Again, we need this fast so, no sanity checking.

byte getAlpha(int x,int y);
Grab alpha channel from this location in RAM.

RGBpack* getBitmapAddr(void);
Returns the address of our bitmap in RAM. Scary..

byte* getAlphaAddr(void);
Returns the address of our alpha channel map in RAM. Also scary!
```

NOTE: You don't need to use the offscreen drawing to effect your bitmap. You can change whatever you like with their get & set color methods. Using offscreen is just one way to generate patters in them.

mask

https://github.com/leftCoast/LC_GUITools/blob/main/mask.h

mask

mask is the base class for setting up masking of display (Or offscreen) drawing. This is used a lot for drawing icons. The standard Left Coast icon button is a rounded rectangle shape. So there is a standard mask file that goes along with them to mask off drawing of the corners. Ta da! Now they blend much better to the background.

Mask, the base class giving this public interface.

Definitions

```
enum masktype {
    unMasked,         // Draw it all.
    totalMasked,      // Draw nothing of this!
    partialMasked     // Better check pixel by pixel.
};
```

Global variables

```
mask* gMask;
```

This is the global that many of the Left Coast displayObj displays will use for masking drawing.

Constructors

```
mask(void);
```

Creates a mask object with default settings.

Pubic methods

```
void setInverse(bool inInverse);
```

How the mask behaves is really up to you. However you decide this can flip it to the opposite.

```
masktype checkRect(rect* inRect);
```

Check an entire rectangle to see if its completely masked, not masked at all or some of it may be masked? Uses inverse to flip value if true.

```
masktype checkRect(int x,int y,int w,int h);
```

Same as above except using values as inputs.

```
bool checkPixel(int x,int y);
```

Is this pixel masked or not. Also uses inverse to flip value if true.

maskRect

A rectangular mask object. It uses the built in code from it's base rect class to tell if points or entire rectangles lie inside or outside it's rectangle. Makes for a good example of extending the mask class.

The public interface.

Constructors

```
maskRect(rect* inRect);
maskRect(int inX, int inY, int inWidth,int inHeight);
```

Public methods

```
masktype checkRect(rect* inRect);
Is this incoming rectangle contained in us? Separate from us? Or overlapping
with us?

masktype checkRect(int x,int y,int w,int h);
Same as above, but using values.

bool checkPixel(int x,int y);
```

scrollingList

https://github.com/leftCoast/LC_GUITools/blob/main/scrollingList.h

Base class for our scrollable collection of.. drawObj(s) living on drawList(s). You fill up a drawList with objects and this is the interface used to control scrolling it. But.. This is one of the ones that gets added to as the need comes up. Eternally under development. SO some stuff works, some is, sos so.

Now, there are a few different ways to do this.

First is the pushPotGUI. The pushPotGUI is a simple rotating potentiometer that you can push to click. A very simple GUI that is only really suited to really simple applications.

Second is the button interface. But I've not written that one yet.

Third is single touch, touchscreen. This one is currently working. I have some ideas for improvements, but it's doing fine as it is.

Lets have a look.

Definitions

```
enum scrollType {
     touchScroll, // Touchscreen.
     dialScroll,  // Push pot interface, includes the next three.
     dSOpenTop,   // This allow scrolling into a header above the list.
     dSOpenBtm,   // This allow scrolling into a footer below the list.
     dSOpenBoth,  // This one allows both ways.
     buttonScroll // "Coming soon." :D
};

enum locType {
     onList
     aboveList
     belowList
};
```

Used internally to tell where an item ended up.

Constructors

```
scrollingList(int x, int y, int width,int height,scrollType sType,eventSet
inEventSet=noEvents,bool vertical=true);

void setScrollValue(float percent);
```

Used for pushPot scrolling. Map your POT values to a percent to drop in here.

```
void offList(void);
We off list? If so, here is where you find out. Actually no. Not written yet.

void dragVertical(event* inEvent);
Finger drag events for vertical lists. Does the math and logic for moving the
list.

void dragHorizontal(event* inEvent);
Should do the same for horizontal, but not written yet.

void doAction(event* inEvent,point* locaPt);
This is the method where the events come in. Does some checking and setup then
passes the event to dragVertical() or dragHorizontal().
```

Setting up your first scrolling list can be a little, complex. You will need two things. Something to scroll, and a scrolling list. Typically you inherit both, make a custom listItem class and a custom scrollingList class. To see the changes that are typically done, there is a simple touch screen draggable list example on Wokwi for this.

https://wokwi.com/projects/447730542276428801

That should help a lot in getting you up and running.

LC_bmpTools

https://github.com/leftCoast/LC_bmpTools

Ok, lets bring images into the mix. LC_bmpTools is all about displaying .bmp files. To do this you will need a working SD drive to hold said files to be displayed. The base object that can be draw to a display is bmpObj. Give it a file path, it grabs the data, and draws it to the display, easy peasy.

But bmpObj relies on bmpImage.h. This draws in all the SD file handling stuff allowing you to easily grab the image off the SD card. But also, allows you to edit the .bmp file and save to back to the SD card. Meaning? Not only can you display bitmap files, you have all the tools necessary to write your own image editor.

bmpObj

This is the BASE CLASS for bitmap file drawing objects. Its a rect location on the display that can spat a bitmap to itself. Want to do a picture, a button or an icon? Start with this. (Or, maybe, just draw a bitmap)

Basically this is the glue that hooks a .bmp file path to a drawObj. Let's have a look at what we have to work with.

Constructors

```
bmpObj(int inX,int inY,int inWidth,int inHeight,const char* bmpPath);
Create a bitmap object given it's drawing location and size on the display.
Along with the full file path to the bitmap on your SD card.

bmpObj(rect* inRect,const char* bmpPath);
Same as above except using a rectangle for loaction and size on the display.
```

Public methods

```
void setSourceOffset(int offstX,int offstY);
The constructor sets up the size and location for where the drawing will go on
the display. This method sets the location of where you will read the data
from the bitmap file.
```

NOTE: In the constructor we are saying.. "I want a rectangle from this image drawn here." Then, with the offset we are saying.. "Copy the rectangle of the image from this location."

```
void setMask(mask* aMaskPtr);
If  there is a mask that goes with this image? Attach it here.

void setGreyedOut(bool trueFalse);
Sets if the image draw has the color removed or not.

colorObj greyscale(colorObj* inColor);
Returns a colorless version of the input color.

void drawSelf(void);
Combines everything together to give you the image on the display.
```

bmpMask

This reads a .bmp file from the SD card and converts to a one bit alpha channel. Dark colors are mask. Light colors are not. This is saved as a one bit per pixel bitmap. Meaning, a 32x32 pixel icon can have a complete bitmap mask for 128 bytes.

Let's have a look at the public interface.

Constructors

```
bmpMask(void);
Creates the bmpMask obj and does initial setup.
```

Public methods

```
void readFromBMP(const char* filePath);
reads the inputted bmp file and creates the mask.

bool checkPixel(int x,int y);
Using the internal mask this returns if we should draw this pixel or not.
```

iconButton

https://github.com/leftCoast/LC_bmpTools/blob/main/iconButton.h

This is a class defaulted to a 32x32 pixel .bmp file based icon that is clickable. You want cool looking clickable buttons? This is your starting point! This was designed to be easy to use so there's not a lot to it from the user interface point of view.

Let's have a look.

Constructors

```
iconButton(int xLoc,int yLoc,const char* path,int pix=32);
Constructs a clickable iconButton at this location, from this file, defaulted
to be 32x32 pixels.

void setEventSet(eventSet inEventSet);
Used for setting the button's event set. If you set it to noEvents, it
automatically grays out the the button to show inactive. Pretty slick huh?
```

NOTE: When choosing event sets for these clickable buttons, the usual choice is "fullClick". That will give the most expected result.

bmpFlasher

https://github.com/leftCoast/LC_bmpTools/blob/main/bmpFlasher.h

This is derived from the original flasher class. But, instead of sapping between two colors, this swaps between to .bmp images.

The public interface is an extension of flasher, check there for more public methods.

https://github.com/leftCoast/LC_GUITools/blob/main/flasher.h

Constructors

```
bmpFlasher(int inX,int inY, int width,int height,const char* onBmpPath,const
char* offBmpPath);
Constructs a bmpFlasher given location, size, with on and off .bmp files.

bmpFlasher(rect* inRect,const char* onBmp,const char* offBmp);
Same as above except using a rectangle for location and size.

void setup(const char* onBmpPath,const char* offBmpPath);
Used internally but could be called by user. Sets the two .bmp files.
```

```
void drawSelf(void);
Does all the drawing.
```

bmpLabel

https://github.com/leftCoast/LC_bmpTools/blob/main/bmpLabel.h

This allows one to use text over a .bmp image with, hopefully, no flickering. To do this it uses offscreen drawing. It sets up an offscreen buffer and draws the background to it. The it draws the text over the background. Once this is all complete it "blits" it to the display.

Most of the public UI is from the label class.

https://github.com/leftCoast/LC_GUITools/blob/main/label.h

All this class has for public user interface is it's constructor.

Constructors

```
bmpLabel(int inX,int inY,int width,int height,char* inText,char* bmpPath);
Given location, size, text and a .bmp file. This will construct a bmpLabel.
```

NOTE: This only works for the default Adafruit font. There needs to be one setup for the nicer fonts. But not today.

LC_keyboard

https://github.com/leftCoast/LC_keyboard/tree/main

LCPKeyboard is an entire package for editing text with the Left Coast GUI. Although it can be used stand alone. Typically it's used on a "panel" running under LC_lilOS. There *is* a base keyboard class that *may* work on it's own. But by the time you're thinking about editing text with keyboards, you should probably graduate to LC_lilOS anyway.

scrKeyboard

https://github.com/leftCoast/LC_keyboard/blob/main/scrKeyboard.h

scrKeyboard is the base class for keyboards. It's a drawGroup class holds a pointer to the current editable object. The keyboard sets the mode of the keys be it normal, shifted, symbols.

All of the keys are idlers. This means that they pretty much take care of themselves. When they get clicks, they notify the keyboards object. Some add data like letters & numbers, some are control like shift & enter. The keyboard picks up these messages and pass them on to the current editable object.

keyboard

Starting with keyboard we'll have a look at what we have for the interface.

Definitions and constants

```
enum  keyStates { chars, shifted, numbers, symbols };

struct keyColors {
    colorObj    inputKeyText;
    colorObj    inputKeyBase;
    colorObj    inputKeyHText;
    colorObj    inputKeyHBase;

    colorObj    contolKeyText;
    colorObj    contolKeyBase;
    colorObj    contolKeyHText;
    colorObj    contolKeyHBase;

    colorObj    deleteKeyText;
    colorObj    deleteKeyBase;
```

```
    colorObj    deleteKeyHText;
    colorObj    deleteKeyHBase;
};
```

Constructers

```
keyboard(editable* inEditObj=NULL);
```

Public methods

```
void loadKeys(void);
```
This creates and loads all the keys for the keyboard.

```
void keyClicked(keyboardKey* aKey);
```
Kind'a a general callback so you can do some universal thing when a key is clicked. Maybe beep?

```
void handleKey(char inChar);
```
Handler for input keys like letters and numbers. Packs up a keystroke with the info. and passes it on to the current editable object.

```
void handleKey(keyCommands inEditCom);
```
Much like above. In this case the message packed up is a command message. could be a delete key, could be enter.

```
void handleKey(keyStates inState);
```
This sets the internal state varaible that the keys watch to know what state they should be drawn in and how to respond to clicks.

```
void setEditField(editable* inLabel);
```
Tyically when a drawObj gets focus, if it's editable it will connect using this method.

```
editable* getEditField(void);
```
returns the current editable object.

```
keyStates getState(void);
```
Returns the current saved state for the keys to watch.

```
void drawSelf(void);
```
This draws the background for the keys.

keyboardKey

Base class for all keyboard keys. It's an idler with no idle() method. It let's is children have that one for themselves.

Very simple interface.

Constructors
```
keyboardKey(keyboard* inKeyboard);
The keyboard constructs these so it just passes in it's pointer.
```

NOTE : Seeing that this will never be created as a global. This calls hookup() for the keys. So they don't have to.

Public methods
```
void beenClicked(void)
Passes the click onto the keyboard. Can be overridden to "something" when any
key is clicked.
```

inputKey
A key that adds characters to the text. Derived from keyboardKey and label. So it can easily show it's character.

It's interface..

Constructors
```
inputKey(const char* inLabel,const char* inNum,const char* inSym,word locX,
word locY,byte width,byte height,keyboard* inKeyboard);
Fill in all the bits and it'll give you the key you're looking for.
```

Public methods
```
void idle();
Used to keep updated.

void drawSelf(void);
Draws the key in all it's different possible states.

void doAction(void);
Passes on it's character to the keyboard. If the shift key has been pressed it
causes that to be cleared.
```

controlKey
This is the kind of key that does everything else. Shift, delete, arrow, ok etc.

It's interface..

Constructors
```
controlKey(const char* inLabel,keyCommands inCom,word locX, word locY,byte
width,byte height,keyboard* inKeyboard);
Fill in all the bits and it'll give you the key you're looking for.
```

```
void drawSelf(void);
```
Draws the key in all it's different possible states.

```
void handleShift(void);
```
This actually tells the keyboard what to do when shift key is clicked.

```
void handleNumber(void);
```
Same as above except for number key.

```
void doAction(void);
```
This passes it's command up to the keyboard except for shft and number. In those two cases the above handle Shift() or handleNumber() methods are called.

NOTE: About the legacy drawing commands at the top of these files. Back in the day, the GUI was quite different and there was the idea of a stampObj that could be used to "stamp" common images on the screen. It was just a bad idea. But the base keyboard class relied on it. So the remnants are here for keyboard to use.

keystroke

https://github.com/leftCoast/LC_keyboard/blob/main/keystroke.h

This is what is passed between the keyboard and the editable object. Its an enum & struct defined in a .h file.

Definitions

```
enum  keyCommands {
     input,
     shift,
     number,
     symbol,
     backspace,
     arrowFWD,
     arrowBack,
     enter,
     cancel,
     ok
};
```

The list of possible commands from the keyboard.

```
struct keystroke {
  keyCommands editCommand;  // Edit command or inputted character?
  char        theChar;      // If inputted character, here it is!
};
```

The keyboard is responsible for assembling these and sending them on to the current editable object.

editable

https://github.com/leftCoast/LC_keyboard/blob/main/editable.h

The base class for all things editable. It contains all the logic an editable thing needs to interact with a keyboard. Like beginEditing() when activated, handleInputKey() when the user is typing, handleArrowKey() for moving around text. Stuff like that.

We'll have a look at it's public interface.

Constructors

```
editable(void);
Creates one, everything defaulted. Not editing, not a success, no exit on
return..
```

Public methods

```
void setExitOnEnter(bool trueFalse);
Sets up that this editable will end editing with a success when it sees a
return keystroke.

void handleKeystroke(keystroke* inKey);
This is the input point for all keystrokes from the keyboard. From here they
branch out to the individual keystroke handlers.

void beginEditing(void);
This is called to bring an editable from dormant to an active editing state.

void handleInputKey(void);
If handleKeystroke() receives and input keystroke, it's sent here. In this
case nothing happens. It's passed on to the "offspring" to deal with.

void handleBackspaceKey(void);
If handleKeystroke() receives a backspace keystroke, it's sent here. In this
case nothing happens. It's passed on to the "offspring" to deal with.

void handleArrowFWDKey(void);
If handleKeystroke() receives a forward arrow keystroke, it's sent here. In
this case nothing happens. It's passed on to the "offspring" to deal with.

void handleArrowBackKey(void);
If handleKeystroke() receives a back arrow keystroke, it's sent here. In this
case nothing happens. It's passed on to the "offspring" to deal with.

void handleEnterKey(void);
```

If handleKeystroke() receives an enter keystroke, it's sent here. It may stop the editing it may just pass it on to handleInputKey(). Depending on how exit on return key is set.

```
void handleCancelKey(void);
```
If handleKeystroke() receives a cancel keystroke, it's sent here.

```
void handleOkKey(void);
```
If handleKeystroke() receives an Ok keystroke, it's sent here.

```
void endEditing(void);
```
Shut down editing and return to a dormant state.

```
char getCurrentChar(void);
```
Returns the last current keystroke character.

```
bool getEditing(void);
```
Returns if in editing mode or not.

```
bool getSuccess(void);
```
Returns if this edit session was a success or not.

```
bool getExitOnEnter(void);
```
Returns if this editable is set to exit on return keystroke or not.

editLabel

https://github.com/leftCoast/LC_keyboard/blob/main/editLabel.h

This is your bread and butter editable line of text. Responds to mouse clicks. Keystrokes. Blinks a cursor. Can be auto reset if editing is canceled. All the usual edit field kind of things.

Public Ui.

Definitions

```
#define  CURSOR_BLINK 500
```

Constructors

```
editLabel();
Sets up the default editLabel. As for the rest? The label class has a lot of
different flavors of constructor. So, they were all added here just in case
someone liked one over another.

editLabel(const char* inText);
editLabel(const char* inText, int inSize);
editLabel(int inLocX, int inLocY, int inWidth,int inHeight);
editLabel(int inLocX, int inLocY, int inWidth,int inHeight,const char*
inText);
editLabel(int inLocX, int inLocY, int inWidth,int inHeight,const char*
inText,int textSize);
editLabel(rect* inRect,const char* inText,int textSize=1);
These all take different sets of parameters to set different sets of data.

editLabel(label* aLabel);
Or just copy another one with the settings you like?
```

Public methods

```
void init(void);
All the constructors call this to setup defaualt values.

void setIndex(int newIndex);
Set the cursor to this index.

int getIndex(void);
Where's the cursor now?

void beginEditing(void);
Overrides the original beginEditing(), takes care of setting the mEditing
variable and calls hookup() because this will never be created as a global.
```

```
void handleInputKey(void);
```
Does all the steps to add this keystroke to the editing string.

```
void handleBackspaceKey(void);
```
Removes the character just in front of the cursor.

```
void handleArrowFWDKey(void);
```
Bumps the cursor forward through the string.

```
void handleArrowBackKey(void);
```
Moves the cursor backward through the string.

```
void endEditing(void);
```
If this is a successful edit it saves the string. If not? It replaces it with the saved string. Reversing the changes.

```
void doAction(event* inEvent,point* locaPt);
```
A click was registered in the edit label field. Set the cursor at that location.

```
void drawSelf(void);
```
Draws the label. This has all the state variables available so drawing can be a function of editing state. If desired.

```
void setInitalPointers(void);
```
Just starting to edit. set up initial conditions.

```
void showText(void);
```
Before any drawing, this is called to set up all the parameters for drawing the text taking account of it's state etc.

```
void idle(void);
```
Pretty much, this just blinks the cursor.

datafield

https://github.com/leftCoast/LC_keyboard/blob/main/datafield.h

Now, editLabel can get you going. But very soon one wants multiple editLabels on the same screen (Or panel) And it would be nice to be able to control how they look when drawn. Also how to tell if it's a click to start editing or to place the cursor.. There is a lot of management here. dataField is here to help.

The idea of a datafield is an editing field that can..

A) Have a background drawObject and..

B) Optionally live on a panel/screen with multiple other datafields sharing a single keyboard. And of course..

C) Doing all the correct editing things.

Sharing the keyboard with other dataFields use the focus functionality given to us by drawObj. The idea is that when we gain or loose focus, we can do the "right thing" managing our given dataField.

Behind the scenes dataField manages this by extending drawGroup to group together two drawing objects. An editField & any other object type you like to use as background for the editField. This ends up with three things that can be clicked on basically in a stack.

A) Your background drawObj.

B) Your editLabel.

C) The dataField that is the group drawObj that contains the first two.

Remember, since this extends drawGroup, which in turn extends viewMgr, which in turn extends drawObj.. There is a ***lot*** of public, and protected, methods to draw on. Let's have a look at the public interface that dataField adds to the list.

Constructors

```
datafield(int x,int y,int w,int h);
Given the size and location, returns a dataField.
```

Public methods

```
void begin(keyboard* inKeyboard,editLabel* inEditLabel,drawObj*
background=NULL);
```

This sets up the necessary items to link up. Background is optional.

```
void doAction(void);
```
We should only get a click when we are NOT editing. Hence, on clicks, we set the focus pointer to us. This will fire off the editing start sequence.

```
void setThisFocus(bool setLoose);
```
And here's the focus pointer call we need to start or end editing sequences.

```
void idle(void);
```
If the user clicks OK or cancel, the dataField obj actually doesn't see it. So, during idle time, we check to see if the editLabel has shut down on us. If so, we quietly set the focus pointer to NULL, triggering everyone to do their stop editing things. Then we shut off the editLabel's events. And turn our events back on (To wait for the next "start editing" click). Who knows what everyone else is doing?

bmpKeyboard

https://github.com/leftCoast/LC_keyboard/blob/main/bmpKeyboard.h

A much better looking keyboard. As of this writing, bmpKeyboard is pretty much hard coded to be used in the lilOS environment. In fact, there's a lot of hard coding in this class and it does need a bit of a cleanup. But it all works.

bmpKeyboard

There are two flavors of bmpKeyboard. The standard one that lives on the bottom of the screen. And, the modal one that lives on a modal dialog with ok & cancel buttons. You can have those in lilOS.

To speed things up, bmpKeyboards use offscreen drawing. There is a blank input key stored in RAM as a bitmap. When drawing the keyboard, or redrawing because of a modifier key being clicked. This blank is stamped over all the keys that will change so that they update ***much*** faster. But we're looking at 2,376 bytes of ram while the bmpKeyboard is running. Keep that in mind. You can't run this one on an UNO.

bmpKeyboard extends keyboard so you get all of keyboard's public and protected calls. We'll have a look at what we are adding.

Constructors

```
bmpKeyboard(editable* inEditObj,bool modal=false);
For the constructor it's looking for the initial editObj and if it is to be
modal or not. IE. Living on a modal dialog as opposed to the parent "panel".
The initial editObj can be set to NULL if desired.
```

Public methods

```
void loadKeys(void);
Keys are different, loading them is different. Hence this override call to
loadKeys().

int col(int col, int row);
Given the column & row key I want to draw. What's the column's pixel I start
with? Because the rows are different lengths.

bitmap* getKeyMap(void);
Returns a pointer to the blank keycap's bitmap.

colorObj* getKeyTextColor(void);
Returns a pointer to the color that's being used for the letters on the keys.
Meaning? You can change it.
```

bmpInputKey

The bmp version of an input key. Meaning inputted text. Acts like the keyboard input key but drawn differently. Public interface is pretty simple.

Constructors

```
bmpInputKey(const char* inLabel,const char* inNum,const char* inSym,int
inX,int inY,int inWidth,int inHeight,bmpKeyboard* inKeyboard);
It's looking for.. The letter, number or symbol to send, x,y location, size,
and a pointer to it's bmpKeyboard.

void drawSelf(void);
This does all the drawing. Uses the RAM based bitmap. Very tricky.
```

bmpControlKey

The bmp version of an input key. Meaning inputted text. Acts like the keyboard input key but drawn differently. Public interface is pretty simple.

Constructors

```
bmpControlKey(const char* inLabel,keyCommands inCom,int inX, int inY, int
inWidth,int inHeight,bmpKeyboard* inKeyboard,const char* bmpPath);
It's looking for.. The text label, the key command to send, x,y location,
size, A pointer to it's bmpKeyboard and a file path to it's .bmp file.
NOTE : Most control keys have custom bitmap files.

void drawSelf(void);
This does all the drawing. Typically control keys do not use RAM based bitmas.
They tend to have custom .bmp image files.
```

IOandKeys

https://github.com/leftCoast/LC_keyboard/blob/main/IOandKeys.h

This is for running something like a terminal. Or a chat window. It links to an outgoing message editLabel and a textView to display both incoming and outgoing messages.

Here is an example.

https://wokwi.com/projects/448479010495548417

And the public user interface..

Constructors

```
IOandKeys(editLabel* inEditField,textView* inTextField);
Hand this a pointer to an editLabel and a text view object it hands back your
keyboard with everything defaulted.
```

Public methods

```
void handleKey(keyCommands inEditCom);
Takes in the keystrokes looking for the enter key to send the message. All
other keystrokes are just passed on to the base classes.

int haveBuff(void);
Returns the number of bytes ready to send out.

bool getBuff(char* buff,int maxBytes);
When the return keystroke has been caught, this is called to fill the outgoing
buffer with the message.
```

SD Card files

LC_blockFile

https://github.com/leftCoast/LC_blockFile

blockFile

https://github.com/leftCoast/LC_blockFile/blob/master/src/blockFile.h

This is an object that will manage a file on an SD card much like dynamic memory. Give it a data buffer & number of bytes, its stores it for you and hands you back an ID number.. From that you can store, retrieve, delete or resize your block of data, as your whim takes you. Files can have pretty near an unlimited number of numbered blocks.

Now, when you start up your program, the big problem is.. In what block is anything stored? There are no guarantees as to the order of the block ID allocation. Basically, you are faced with the old problem, "The keys for the car are locked in the car".

What to do?

Here is how it this problem is addressed. The first block allocated is known as the "root" block ID. You use this file block to hold whatever information you need to decode the rest of your file. But what's this block's ID number? Doesn't matter. The readRootBlockID() method will hand it back to you.

Let's have a look at blockFile's public interface.

Public Definitions

```
#define BLOCKFILE_TAG              "BLOCKFILE"
#define BLOCKFILE_TAG_SIZE         12
#define CURRENT_BLOCKFILE_VERSION 1
#define INITIAL_BLOCKFILE_ID       1

#define BF_NO_ERR          0   // Everything's fine now, ain't it?
#define BF_MEM_ERR         1   // malloc() failed.
#define BF_VERSION_ERR     2   // Right kind of file. But, wrong version.
#define BF_FOPEN_ERR       3   // Tried to open from a file path but failed.
#define BF_FREAD_ERR       4   // Tried to read a buffer but failed.
```

```
#define BF_FWRITE_ERR     5   // Tried to write a buffer but failed.
#define BF_ISDIR_ERR      6   // Looking for a file, got path to directory.
#define BF_SEEK_ERR       7   // Trying to move the file pointer, failed.
```

Constructors

```
blockFile(char* inFilePath);
```

You start off with a full file path string. This associates our object with a file on the SD drive. It will create a new file if necessary.

Public methods

```
unsigned long readRootBlockID(void);
```

Returns the ID for your root block.

NOTE : You store the information you need to decode your file in here.

```
unsigned long getNewBlockID(void);
```

reserves a block ID and passes it back to you.

NOTE: remember, The first blockID issued to you, will be saved as your initial block. That can come from getNewBlockID() or addBlock().

```
unsigned long addBlock(char* buffPtr,unsigned long bytes);
```

Adds this new block of data to the SD card and returns a new block ID for it. Returning the new ID to you.

```
bool deleteBlock(unsigned long blockID);
```

If it can find this blockID, and it's NOT the initial block ID. It will free this block and return true. Else it will return false.

```
bool writeBlock(unsigned long blockID,char* buffPtr,unsigned long bytes);
```

I have my ID, save this buffer and return success or not.

```
unsigned long getBlockSize(unsigned long blockID);
```

Returns how large a block of data is, given a block ID. Zero if anything goes wrong.

```
bool getBlock(unsigned long blockID,char* buffPtr,unsigned long bytes);
```

Given an ID and a buffer, fills the buffer with this ID's data.

```
void cleanup(unsigned long allowedMs);
```

Planned trash collection method. Not written as of this writing.

```
void deleteBlockfile(void);
```

Mark file to be erased when object is deleted. As in, entire file is gone forever.

```
int checkErr(bool clearErr=false);
```

returns the last error value. Optionally clears the error after returning it.

```
bool isEmpty(void);
returns true if there are no bytes stored in this blockFile.
```

fileBuff

https://github.com/leftCoast/LC_blockFile/blob/master/src/fileBuff.h

There is a helpful base class included in this package called fileBuff. If your class is something that needs to be stored in a block file. You can inherit fileBuff, override three methods (calculateBuffSize(), writeToBuff() and loadFromBuff()) This handles the underlying nonsense of getting you in and out of the file. Other objects will suddenly know just how to save you in a file and pull you out.

Let's have a look at fileBuff's public interface.

Constructors

```
fileBuff(blockFile* inFile);
constructor for root ID.

fileBuff(blockFile* inFile,unsigned long blockID);
constructor for all those "other guys".
```

Public methods

```
unsigned long getID(void);
Returns our file ID.

unsigned long calculateBuffSize(void);
Returns how many bytes you will need to have stored. To be overridden and
filled out by you.

void writeToBuff(char* buffPtr,unsigned long maxBytes);
knowing how many bytes need to be stored to save yourself. Whrite these bytes
into this buffer. To be overridden and filled out by you.

unsigned long loadFromBuff(char* buffPtr,unsigned long maxBytes);
Given this buffer reconstruct yourself. To be overridden and filled out by
you.

bool saveSubFileBuffs(void);
If you manage sub objects that need to be stored, you do them here. Otherwise
ignore this one. To be overridden and filled out by you.

void eraseFromFile(void);
```

We've been deleted and need to erase ourselves from the file? This will do that for us.

```
bool saveToFile(void);
```
Called by whomever wants us put in a file. This manages that action.

```
bool readFromFile(void);
```
Called by whomever wants us created from a file. This manages that action.

LC_SDTools

https://github.com/leftCoast/LC_SDTools

SDTools.h

https://github.com/leftCoast/LC_SDTools/blob/main/src/SDTools.h

A grab bag of basic file tools that come in handy for working with SD cards & files. Lets go to the public interface first on this one.

First set is big/little Indian for file reads & writes. Some files are big, some or little. Arduino is little. These work for the reading & writing of different size integers using these file functions.

NOTE: All methods and functions in LC_SDTools that return string pointers, MUST be copied locally before use. We're only using one buffer here, so you don't want things getting mixed up down the line.

```
#define  BYTE_SWAP    bigIndian swap;
Put this at top of function, and for the duration of the function, all
integers will be swapped. Both reading and writing to files.

bool SDFileErr;
This is set if a big/small Indian function has a file error. The idea is to
clear it before doing your reads/writes. Then check it to see if everything
went ok.
```

bigIndian

Stack based class that flips the byte order for the calls while it's in scope. Automatically flips them back when going out of scope. NOT reentrant!

Constructors

```
bigIndian(void);
Stack based class. Causes bytes to be flipped when present.
```

Function calls - The integer reading and writing calls.

```
bool read16(void* result,File f);
For reading two byte numbers.

bool write16(uint16_t val, File f);
```

For writing two byte numbers.

```
bool read32(void* result,File f);
```
For reading four byte numbers.

```
bool write32(uint32_t val, File f);
```
For writing four byte numbers

Function calls - General purpose file calls.

```
bool MacOSFilter(pathItem* inItem);
```
Used for filtering out MacOS files. The ones Apple hides in there. Returns true if pathItem* is NOT one of the added system files.

```
bool createFolder(const char* folderPath);
```
Returns true if this folderPath can be found, or created.

```
char* numberedFilePath(const char* folderPath,const char* baseName,const char* extension);
```
Given a path, baseName and extension this hands back a string with a path to an unused numbered file. For example "/docs/NoName5.doc". IF it can not allocated this file it will return NULL.

```
File TRUNCATE_FILE(const char* path);
```
TOTAL HACK TO GET AROUND NO FILE truncate() CALL in SD library.

```
void fcpy(File dest,File src);
```
The file version of strcpy(). The dest file must be open for writing. The src file must be, at least, open for reading. (Writing is ok too) The dest file index is left pointing to the end of the file. The src file index is not changed.

```
void fcat(File dest,File src);
```
The file version of strcat(). The dest file must be open for writing. The src file must be, at least, open for reading. (Writing is ok too) The dest file index is left pointing to the end of the file. The src file index is not changed.

```
bool extensionMatch(const char* extension,const char* filePath);
```
Pass in your extension and a file path. Returns if the file extension matches.

filePath

https://github.com/leftCoast/LC_SDTools/blob/main/src/filePath.h

File path is actually the bases for file browsing. Be it command line or GUI based. What kind of thing are we pointing at right now? If its a directory what does the list of things we contain look like? And tools for filtering such lists.

File path includes several "helper" classes. We'll start with them.

A function

```
void clipTrialingSlash(char* instr);
If you look for a filename ending in '/'. 9 out of 10 times SD will crash your
program. So, we make sure the silly things are clipped off before asking.
That's what this does, drop in a prospective path and if it has a trailing
slash, snip! It's gone. No more crashing because of SD.
```

pathItem

This is the base class for the "things" you find on a file path. root, folders and files. It's actually a double linked list node. It holds the common things like what kind of thing it is and it;'s name. we can have a look at it's public interface.

Constructors

```
pathItem(void);
Gives you back a default path item with no type and no name. Seems silly now,
but it good for catching sloppy code. If things turn up with no type.

pathItem(pathItem* aGrandItem);
This give back a new copy of the path item that was passed in.
```

Public methods

```
pathItemType getType(void);
Passes back the type of this path item. Better not be "noType".

char* getName(void);
Returns a pointer to the name if this path item.

int getNumPathChars(void);
Returns the number of chars in this item's name. (Not including the /0.

void addNameToPath(char* path) =0;
Pure virtual. Adding a name to the path means you need to go out in the SD
world and ind out what type of thing name is attached to.
```

```
pathItem* getParent(void);
Returns the path item of the parent of this path item. Or NULL if this is
actually "root".

pathItem* getNext(void);
Returns the next item on the path list. "Child" item.
```

rootItem

Every path starts at "root' or "slash" /. Even relative paths have a root, because what their relative to has a root. You can think of "root" as the SD card itself.

filePath is is the end product of all of these and what the user would typically be interacting with. So lets have a look at filePath's user interface. Derived from pathItem is rootItem.

Constructors

```
rootItem(void);
Passes back a root item. type is rootType and the name is "/".
```

Public methods

```
int getNumPathChars(void);
Hard coded to give back 1.

void addNameToPath(char* path);
We are building a text path. The buffer is big enough, just add yourself to
the end. But NO! Root is the beginning of the path. You don't add the root
item to the path. You start the path WITH the root item. This erase the path
string and places "/" at the beginning.
```

fileItem

So handy, you can even store stuff in them. Files can only be at the tip of the path. Lets see what they have to offer.

Constructors

```
fileItem(const char* fileName);
Give it a name and it returns a new fileItem.
```

Public methods

```
void addNameToPath(char* path);
Concatenates it's name to the path string.
```

folderItem

This one is a little more complicated.

Constructors

```
folderItem(const char* folderName);
Give it a name and it returns a new folder item.
```

Public methods

```
int getNumPathChars(void);
Different than the file name in that it has a trailing slash. But this still
reutrns just the number of chars. Not the trailing /0.

void addNameToPath(char* path);
This adds our name, with a trailing slash, to the passed in path string
```

filePath

This is where everything is tied together. filePath is the interface that the user will interact with the most. But to go through directory lists your going to need to deal with double linked lists and all that so here's a link to the lists library.

https://github.com/leftCoast/LC_baseTools/blob/master/lists.h

And your public interface.

Types

```
enum pathItemType { noType, rootType, folderType, fileType };
```

Constructors

```
filePath(void);
```

Public methods

```
void reset(void);
Deletes all used memory and sets everything back to initial state.

int numPathBytes(void);
Returns the number of bytes needed to store this path in a c string.
```

NOTE: Including the \0.

```
pathItemType getPathType(void);
Returns the type of path this is. IE what kind of thing is it pointing to?

char* getPathName(void);
Returns the name of what this path is pointing to. Be it a folder or a file,
or root.

pathItemType checkPathPlus(const char* inPath);
```

If we were to add this name to our current pathList.. What do we end up with? A file? A folder? Nothing at all? Find out and return the answer.

```
bool addPath(const char* inPath);
```
Try to add this path segment to our existing path. If it works out? This new path will be our path. If not? No change. Returns true for success.

```
bool setPath(const char* inPath);
```
Used to set the initial path for browsing. If the path is NOT found on the SD card, this fails and gives back a false. Returns true for success.

NOTE: Paths must start with '/' because they all start at root. They Must also fit in 8.3 file names, because the SD library is brain dead and we're stuck with it for now.

```
char* getPath(void);
```
Returns the string representing our full path.

```
pathItem* getCurrItem(void);
```
Returns a pointer to the last (current) item on our path.

```
char* getCurrItemName(void);
```
Same as getPathName(). Returns the name of the last item in our path.

```
void dumpChildList(void);
```
If we are pointing at a directory, dump our list of what's in that directory.

```
void refreshChildList(void);
```
If we are pointing at a directory, reload our child list from current disk data.

```
int numChildItems(void);
```
If we are pointing at a directory, this will return the number of items in our directory.

```
pathItem* getChildItemByName(const char* name);
```
If we are pointing at a directory, find the item on our child list with this name and pass back a pointer to it. NULL on failure.

```
bool pushChildItemByName(const char* name);
```
If we are pointing at a directory, and we have a child item with this name? We add this to our path. Basically changing our path to point at that item. Returns true for success.

```
bool pushItem(pathItem* theNewGuy);
```
This does that actual adding of the new item to our name for the pushChildItemByName() call above. Returns true for success.

```
void popItem(void);
Strips one item off the end of our path. Has the effect of going up one
directory.

bool clearDirectory(void);
If we are pointing at a directory, will recursively clear our directory, on
the SD card, of all files & folders. Returns true for success.

bool deleteCurrentItem(void);
Delete what we are pointing at from the SD card. That includes all files and
folders empty or not. Returns true for success.
```

Public Variable

```
pathItem* childList;
This is the child list. Probably should make this a method to get this. Mabe
later..
```

Here is an example of these tools in action. A command line file browser.

https://wokwi.com/projects/401643432890805249

LC_docTools

https://github.com/leftCoast/LC_docTools

docFileObj

https://github.com/leftCoast/LC_docTools/blob/main/src/docFileObj.h

The docFileObj is a class that supports opening, reading and editing files. When opening a file for editing, it will create a temporary copy of your file. and let you edit teh copy. This gives you the open of discarding changes, saving changes, or saving your changes to a new file.

This is the base object of any filetype that would need this kind of functionality. As of this writing .bmp files are based on this. Meaing that can be created, opened, read and if desired edited.

Let's have a look at the public interface.

Definitions

```
#define TEMP_FOLDER       "/temp/" // Default folder for temporary files.
#define FILE_SEARCH_MS    500      // Time allowed to find a temp file name.

enum fileModes {
     fClosed,
     fOpenToEdit,
     fEdited,
     fOpenToRead
};
Modes used internally.
```

Constructors

```
docFileObj(const char* filePath);
```

Public methods

```
bool createNewDocFile(void) = 0;
Pure virtual method filled in be descendant classes to create a new file of
their type.

bool openDocFile(int openMode);
```

Attempts to open a doc file and returns success or not.

```
bool saveDocFile(const char* newFilePath=NULL);
```
Attempts to save a doc file, with optional new file path. returns success or not.

```
void closeDocFile(void);
```
Closes a doc file.

```
bool changeDocFile(const char* newPath);
```
Attempt to change our association to a different doc file. Returns success or not.

```
void setAsAutoGen(bool trueFalse=true);
```
Internally used. Save off whether our document has an auto generated name or not.

```
bool fileEdited(void);
```
Return if we have unsaved changes or not.

```
char* getName(void);
```
Returns the name of the file for our document.

```
char* getFolder(void);
```
Returns the folder path that our file is saved in.

```
byte peek(void);
```
Returns the byte of data that our file index is pointing to.

```
uint32_t position(void);
```
Returns the index value of our file. returns 0 if there is no file open..

```
bool seek(uint32_t index);
```
Moves the index of our file to this location. Returns success or not.

```
uint32_t size(void);
```
Returns the size of the file we're associated with.

```
int read(void);
```
Returns the value of the byte of data that our file index is pointing to. Increments the index.

```
uint16_t read(byte* buff,uint16_t numBytes);
```
Reads up to numBytes from the file index to the supplied buff. Returns the number of bytes read.

```
size_t write(uint8_t aByte);
```

Writes a byte to the file at index. Increments the index and returns how many bytes were written. 0 or 1.

`size_t write(byte* buff,size_t numBytes);`
Writes a numBytes from buff to the file at index. Increments the index and returns how many bytes were written. 0..numBytes.

Protected methods

`bool checkDoc(File inFile);`
When the inherited opens up a document, this will be called to see if its parsable as the kind of file the class is looking for.

`bool createEditPath(void);`
Find an unused temp file and assign it to the edit file path.

baseImage

https://github.com/leftCoast/LC_docTools/blob/main/src/baseImage.h

This is the base class for files that are images. Contains the basics methods getWidth(), getPixel(), setPixel(). these are the public calls for the user of any image file.

Public interface.

Constructors

```
baseImage(const char* filePath);
Creates a blank baseImage with a file path.
```

Public methods

```
int getWidth(void);
Returns the width in pixels of this image.

int getHeight(void);
Returns the height in pixels of this image.

colorObj getPixel(int x,int y);
Returns a colorObj representing the color at this pixel.

RGBpack getRawPixel(int x,int y)=0;
Pure virtual to be filled in by descendants. Returns the RGBPack
representation of the color of this pixel.

bool setPixel(int x,int y,colorObj* aColor);
Sets the color of the pixel at location x,y.

void setRawPixel(int x,int y,RGBpack* anRGBPack)=0;
Pure virtual to be filled in by descendants. Sets the color of the pixel at
location x,y.

bool getRow(int x,int y,int numPix,RGBpack* RGBArray);
Copies a section, of a row of pixels, starting at x,y, to a RGBPack array.
Returns success or not.

bool setRow(int x,int y,int numPix,RGBpack* RGBArray);
Sets a section, of a row of pixels, starting at x,y, from a RGBPack array.
Returns success or not.

bool checkXYLmits(int x, int y);
Returns if this position is within limits or not.
```

bmpImage

https://github.com/leftCoast/LC_docTools/blob/main/src/bmpImage.h

This is the specific base class for reading, creating and writing .bmp files. Based on baseImage based on docFileObj. All if the image drawing for the Left Coast GUI uses this for reading the images and icons from the SD cards. Let's have a look and what it has to use.

Functions

```
bool createNewBMPFile(const char* newPath,int inWidth,int inHeight);
Given a path ending in a .bmp file name. A width and height. This will create
a new initialized .bmp image file of that size, at that location.
```

Constructors

```
bmpImage(const char* filePath);
Given a file path to a .bmp file, this creates your bmpImage object.
```

Public methods

```
bool setNewBMPFile(const char* BMPPath,int w,int h);
Closes this object's .bmp file. Creates a new .bmp file of w x h size. Then
switches to this new file. Returns true if successful.

bool createNewDocFile(void);
We get a general "We need a new file now" call. So using defaults.. Where do
the defaults come from? See below.

void setPWH(const char* imgPath,int w,int h);
Setup for the defaults for the next createNewDocFile() call.

RGBpack getRawPixel(int x,int y);
Returns the color of the pixel at file location (x,y) in the form of a
RGBPack.

void setRawPixel(int x,int y,RGBpack* anRGBPack);
Sets the color of this pixel at file locatin (x,y) from a RGBpack.

bool checkDoc(File inFile);
Open the doc file and see if its what we expect. Then grab out the info. we
need to "deal" with it.

uint32_t fileIndex(int x,int y);
Given an (x,y) location of a pixel. Hand back the file index of its location.
Used internally.
```

LC_lilOS

Need GitHub link to repository

One of the issues with microprocessors is that they're limited to a single application. lilOS effectively erases that limitation. Now anyone can write applications for your project.

How this all works:

Arudino's limited RAM is the driving factor behind the single application limitation. The goal is to only load one application into RAM at a time while having this code switching process be as transparent as possible to the user.

Arduino programs start with two functions to fill in: setup() and loop(). lilOS gives you the panel class that has setup() and loop() methods. One derives an application as a subclass of panel, then codes it just as if it were a stand alone application. The main difference is to make sure your application globals are all members of your application class. In this way creating an instance of your application brings everything it needs into RAM. Deleting your application will clear everything out, making room for the next application to load.

To help with the unloading of your application, there are two more methods: close() and closing(). close() can be called by anything to delete your application. closing() will be called after close() has been called, and before you application gets deleted. This allows you to do whatever cleanup and saving of information you need before closing up shop.

The primary limitation is having to gather together all your selected applications and compile them into one code base. You can not dynamically load them from disk during runtime like a desktop computer; this really isn't a big an issue as it would seem. As these applications are all written using the LC_libraries, they should not add a lot to the code space because they will all be sharing the same base code.

Along with this code swapping feature comes a common set of icons with tools to make locating and using them convent and uniform over all applications. Tools for setting up common hardware resources keeping hardware private are also available.

Passing messages to and from the different applications as they go in and out of memory is almost trivial, seeing as all actual globals are always active. These can be setup and used for passing information back and forth.

Setting up background processes is almost trivial as well. Any class can spawn an idler() object which can run in the background independent of the applications. An idler can bring

applications to the foreground if needed. For example, a background idler that watches for phone connections and brings up the phone application when a call comes in.

lilOS

Link to .h file

lilOS

This is the base class that runs everything and maintains the common interface for the applications. For your project you will need to extend lilOS to create your own OS tailored to your hardware. lilOS supplies two very important global variables. OSPtr, a pointer to your OS. Need an OS call? Use this pointer to make it. And, nextPanel. Anything can write a panelID to next panel and whatever panel is running will close pu shop and the panel with panelID matching nextPanel will be loaded into memory. Now if lilOS can not find the panel of that ID? There is a special “homePanel” That you will create and that will be loaded as the default;

Lets have a look at the user interface.

Definitions

```
#define  NO_PANEL_ID      0                    // You have to have this guy.
#define  HOME_PANEL_ID    1                    // Home panel wants this one.
#define STD_ICON_FLDR     "icons/standard/"    // Where inside system folder.

// The list of standard icons supplied by the OS for the applications to use.
enum stdIcons {
     mask22, mask32, app32, check22, check32, choice32, copy32, cross22,
     cross32, cut32, doc16, edit22, edit32, fNew22, fNew32, folder16,
     fldrRet16, fSave22, fSave32, fOpen22, fOpen32, FdrNew22, FdrNew32, note32,
     paste32, pref22, pref32, SDCard16, search22, search32, sort22, sort32,
     trashC22, trashC32, trashR22, trashR32, warn32, x22, x32
};

enum menuBarChoices { noMenuBar, emptyMenuBar, closeBoxMenuBar };
```

Panels get their choice of menu bar starting points.

Globals

```
int      nextPanel;   // Set to the next panelID to swap panels.
lilOS*   OSPtr;       // Need OS things? Here's the address.
panel*   ourPanel;    // The current application that’s running at the moment.
int      panelWith;   // These two may be deleted soon.
int      panelHeight; //
```

Constructors

```
lilOS(void);
```

You’re basic lilOS class constructor.

Public methods

`bool begin(void);`
The global world is online, do hookups. Returns success.

`panel* createPanel(int panelID);`
Inherit and fill in to create your own choice of application, given the inputted panelID.

`void launchPanel(void);`
Dispose of current and launch a newly created panel.

`void loop(void);`
Tell the current panel its loop time.

`int getPanelWidth(void) = 0;`
What is being called panel width is display width in pixels. Fill this in to return the correct value.

`int getPanelHeight(void) = 0;`
Same as above but for display height in pixels.

`void beep(void) = 0;`
Fill this in to make your standard beep sound when clicking icons.

`int getTonePin(void) = 0;`
At this time we're looking for the pin number that the piezo speaker is attached to.

`void setBrightness(byte brightness) = 0;`
TFT displays typically have a pin to set brightness. Fill this in to return which of our pins this is connected to.

`const char* getSystemFolder() = 0;`
Fill this one in to return the full path of the location for our system folder.

`const char* getPanelFolder(int panelID) = 0;`
Knowing where our system folder is located, fill this in to return the full path of any given application's folder.

`const char* stdIconPath(stdIcons theIcon);`
Also knowing the full path of our system folder, return the full path of any given standard icon file.
NOTE: This one is not pure virtual. It actually has a default method that in most cases will work fine with no alterations.

panel

This is the base class for application classes. The one that contains the setup() and loop() methods for your to fill in.

To create a working application, it will need a folder to place on the SD card to contain it's icon and whatever else it needs. Typically the application name, folder name and icon name are the same. And sadly, at this time, limited eight characters or less. For example the breakout's application folder is named, breakout containing an icon called breakout.bmp.

Besides the application named icon, the contents of this folder is completely up to the creator of the application. Place this folder in the ***../system/appFiles/*** folder. Then lilOS will know how to find it, and let the application know where it is as well.

In the Arduino library folder the application source files are typically placed in a LCP_***appName*** folder along side of the ***appName*** folder itself. To install an application you will need to copy the ***appName*** folder into the ***../system/appFiles/*** folder on the SD card.

NOTE: Both source code & appName folder are stored in the LC_appName folder. Just figured I'd try to make that clear.

Panel's public interface.

Constructors

```
panel(int panelID,menuBarChoices menuBarChoice=closeBoxMenuBar,eventSet
inEventSet=noEvents);
```

The panel constructor will be called by your OS object. panelID is assigned by the OS and passed in here. menuBarChoice and eventSet are defaulted to sane choices. You get a menu bar with at least a close box. The event set only applies to the background of the panel. So typically it's set to noEvents. Because it's the things on the background that need the events.

Public methods

```
int getPanelID();
```

Returns the panelID that's been assigned to this panel.

```
bool setFilePath(const char* inName);
```

Given a filename from this panel's folder, generate the fullpath to it. This returns of it was successful. And, if successful, the full path will be stored in the panel's variable mFilePath.

```
void setup(void);
```

This is your setup() function to fill in. Have fun with it.

```
void loop(void);
```

This is your loop() function to fill in. Have fun with it as well.

```
void drawSelf(void);
```
This is typically used to draw the background of your application.

```
void close(void);
```
Used to tell the OS “it’s time to close this panel”. It’s defaulted to open the home panel. Typically this is the action you will want. Basically the home panel that has all the other panel’s icons listed on it.

```
void closing(void);
```
This is called when the panel is about to be deleted. This gives the panel a chance to clean up and do whatever state saving it needs before going out of scope.

```
void handleCom(stdComs comID);
```
There are stdComs listeners can respond to. We’ve not gone over those yet. Panels are also listeners, so they can respond to stdComs. By default, panels can only respond to the close command. You application may need to respond to more than that. If so, you will fill this out to deal with them.

homePanel

Default base home panel with panelID of **HOME_PANEL_ID**. This is basically your “Desktop” panel. Typically has no menu bar and lists all your panel’s icons buttons. You extend this to create your own home panel.

Public user interface.

Constructors

```
homePanel(void);
```
Creates a home panel. Handled internally.

Public methods

```
void setup(void);
```
This is one you will fill in to locate all your panel’s custom icons and display them in some way.

```
void loop(void);
```
Typically left blank.

```
void drawSelf(void);
```
Write the drawing commands to draw your home panel’s background.

appIcon

We keep going on about using application icons and displaying them on your home panel. appIcon can make that a really easy thing to do. You give it a x,y location of the icon,

panelID as a message then the calculated panel's icon's path and let it go. You're home panel's setup() typically ends up being a list of appIcon constructors and handoffs to addObj() to hand them off.

Constructors

```
appIcon(int xLoc,int yLoc,int message,const char* path,int pix=32);
Pass in the x,y pixel location for the application icon, the panelID of that
application and the fullpath to it's icon's .bmp file. This will construct an
appIcon you can pass into addObj() to place our your home screen.

void doAction(void);
Opens the application given by the inputted panelID.
```

menuBar

menuBar.h link

Most applications need at least a close box. Some like to have a bunch of choices. Hence, the menuBar class. By default panels get a menuBar with a close box. If you would like a more complicated menu bar, this is the base class to start with.

NOTE: menuBar is a drawGroup, so you can add stuff like buttons to it and it'll manage them for you.

Public interface.

Definitions

```
#define MENU_BAR_H    24   // Because we have 22x22 icons to stick on it.
#define CLOSE_X       0
#define CLOSE_Y       1
#define CLOSE_SIZE    icon22
```

Globals

```
colorObj menuBarColor;
```

Constructors

```
menuBar(panel* inPanel,bool closeBox=true);
```

Typically your panel base object will call this. Or not, depending on your choice.

Public methods

```
void drawSelf(void);
```

Want a different looking menu bar? Fill this in for the new look.

documentPanel

documentPanel.h file link

Base framework for a document editing panel. This ties in all the state and dialog boxes for doing something like this. New file, open file, save file, do you really want to loose unsaved changes? All that jazz. So this ties in the modalAlert stuff that will come up next.

Seeing its for a document file, you can use it as a base class for anything derived from the document class.

documentPanel

Public interface with a note from the documentPanel.h file comments :

```
Note on createDocObj(void). This is one that the class that inherits this,
creates the document object of the class that it will edit. Use this to create
the fileObj of your design and be able to edit it.

Note on createNewDocFile(void). This is also one that the class that inherits
this fills out. The result of this function is to send back an okCmd or
cancelCmd through the command/listener channel. Depending on whether or not a
new doc was created and loaded or not.

Why is this written this way?

The documentPanel class has no idea what kind of document any child class will
be dealing with. So it has no idea what is involved in creating a new one.
There is a very good chance that this may involve asking the user for
information using an alert Object. So, this gives the child object the option
to dovetail it's own alert object into the UI.

NOTE: Sadly you have to put -something- in your newly created file or the SD
stuff doesn't work and everything falls apart. No empty file allowed, I guess.
```

Definitions

```
enum docPanelStates {
fileClosed, haveFileNoNameNoEdits, haveNamedFileNoEdits, hasEditsNoName,
hasEditsNamed, selectOpen, saveOpen, askOpen, newDocFileOpen
};
```

Functions

```
char* docStateStr(docPanelStates aState);
```

Returns a string representation of the document’s state.

```
bool hasExtension(char* inStr,const char* extension);
```

Pass in a string representing the extension you are checking for without the '.' Into extension. Now you can put in a file name or path into inStr and it'll pass back true or false whether this file has that extension.

Constructors

documentPanel(int panelID,menuBarChoices menuBarChoice=closeBoxMenuBar,eventSet inEventSet=noEvents);
This is very much like the default panel constructor as that it needs a panelID but the rest are defaulted to reasonable values.

Public methods

void createDocObj(void)=0;
Pure virtual, you have to fill out to create your docObj to edit.

bool createNewDocFile(void)=0;
Pure virtual, you have to fill this out to create a new document file of your chosen file type.

void setup(void);
Default setup method, calls your createDocObj method then populates the meme bar with typical edit commands.

void closing(void);
Closes the docObj starting the entire "Save changes?" Chain.

void setFilter(bool(*funct)(const char*));
Let the children set the filter function.

void setDefaultPath(const char* inFolder);
Let the descendants set a default folder path for saving/retrieving documents.

void handleCom(stdComs comID);
When a command comes in, this selects, from our current state, what handler should deal with it. Following are the handler methods.

void handleComFileClosed(stdComs comID);
We have a file, but it's closed.

void handleComHaveFileNoNameNoEdits(stdComs comID);
We have an auto generated new file that's un-edited.

void handleComHaveNamedFileNoEdits(stdComs comID);
We have a named file that is un-edited.

void handleComHasEditsNoName(stdComs comID);
We have a un-named file that is edited.

```
void handleComHasEditsNamed(stdComs comID);
We have a named file with edits.

void handleComSelectOpen(stdComs comID);
We have an incoming open file command.

void handleComSaveOpen(stdComs comID);
We have an incoming save file command.

void handleComAskOpen(stdComs comID);
Receiving a yes or no answer from an alert box.

void handleNewDocFileOpen(stdComs);
Receiving a yes or no answer from an open file dialog box.
```

LC_modalAlerts

Need link for library

Alert boxes, dialog boxes, file browsers, stdComs.. modalAlerts. In a nutshell, an application needs a choice by the user. It puts up a modal to receive that selection. The application is a listener and the choice comes back to the application via a stdComs message.

We'll start with stdComs.

stdComs

Need link to stdComs.h

Not all standard icons are used for commands. This library links those that are commands to the command set that they represent. And supplies a handy function for creating new ready to use ones.

Public interface.

Definitions

```
enum stdComs {
cutCmd, copyCmd, closeCmd, pasteCmd, cancelCmd, okCmd, newFileCmd,
newFolderCmd, newItemCmd, openFileCmd, saveFileCmd, deleteItemCmd, searchCmd,
sortCmd, editCmd
};

enum stdLabels {
warnLbl, choiceLbl, noteLbl, folderLbl, folderRetLbl, docLbl, SDCardLbl
};

enum iconSize { icon16, icon22, icon32 };
```

Functions

```
bmpObj* newStdLbl(int x,int y,iconSize inSize,stdLabels iconType);
Tell it the x,y location you want the icon placed, the icon size and what icon
you want there. This creates the bmpObj ready for you to hand off to your
local addObj() method.

stdComBtn* newStdBtn(int x,int y,iconSize inSize,stdComs iconType,listener*
inListener);
Tell this the x,y position, icon size, what stdComs you want and who wants to
receive the message. This creates the stdComBtn ready for you to hand off to
your local addObj() method.
```

listener

This is the base class for a listener. Basically the bit that's mixed into another class that needs to respond to stdCams messages.

Constructors

```
listener(void);
Basically nothin to do here but create a listener.
```

Public methods

```
void handleCom(stdComs comID);
This is to be inherited and filled out by the including class.
```

stdComBtn

Buttons and command tend to go together. It made sense to create a class that could pop them out without drama.

Public interface.

Constructors

```
stdComBtn(int xLoc,int yLoc,const char* path,stdComs iconType,listener*
inListener,int pixles);
Give it x,y location, fullpath to the image file, the message to pass, the
listener to receive the message and the size of icon to choose. This creates a
stdComBtn ready to hand off to your addObj() method.

void doAction(void);
This will pass it's chosen message to the saved listener.

void active(bool trueFalse);
Set this button to active or not active. Will show non-active by greying out
the button colors.
```

modalMgr

Need link to .h file.

Why this manager for modals? Here's the issue. Modals are typically placed "above" the rest of your screen items, and that causes all sorts of redraw and click through issues. Also, there is the issue of modal's needing to be deleted because they also tend to be temporary in nature. So this gives us three issues to deal with.

1) Need to draw after everything below is completed drawing.

2) Need to block other screen items below from getting clicks.

3) Need to have a mechanism for auto deletion. Because deleting oneself leads to nasty crashing.

modal

Base class for building modal dialog boxes. Let’s see the public interface.

Constructors

```
modal(rect* inRect,eventSet inEventSet=noEvents);
modal(int x, int y, int width,int height,eventSet inEventSet=noEvents);

                void init(void);
void checkIfReady(void);
void draw(void);
bool acceptEvent(event* inEvent,point* locaPt);
```

Marine navigation

LC_navTools

https://github.com/leftCoast/LC_navTools

Oh boy! I didn't want to go here yet. But, there's bits of this document that are going to rely on what's in here. So here goes..

navTools was created to hold low level tools for developing marine navigation software. As of this writing, it only has the globalPos class. This is a gathering of about everything one could do with latitude and longitude. Different ways to enter & read them, along with the math for course and bearing.

Lets take a look at the user interface for globalPos.

Definitions

```
enum quad {
     north,
     south,
     east,
     west
};
```

What quadrant are we talking about when reading a Lat & Lon.

```
#define  G_POS_BUFF_BYTES 40
```

How large of a char buffer do we allow for a string version of a lat. lon. location.

Functions

```
bool checkLatDeg(int degrees);
```

Returns true if the inputted degree value would be valid for a latitude angle.

```
bool checkLonDeg(int degrees);
```

Returns true if the inputted degree value would be valid for a longitude angle.

```
bool checkMin(double minutes);
```

Returns true if the inputted minute value would be valid for a minute value.

```
double rad2deg(double angleRad);
```
Returns a degree value of inputted radians angle.

```
double deg2rad(double angleDeg);
```
Returns radian value for inputted degree angle.

class globalPos

Constructors
```
globalPos(void);
```
Creates an empty non-valid position object.

Public methods
```
bool valid(void);
```
Returns true if the stored position checks out as a valid position.

```
int writeToEEPROM(int addr);
```
Writes this position to EPROM storage and retuens the number of bytes used.

```
int copyFromEEPROM(int addr);
```
Reads this position from EPROM and returns how many bytes were read.

```
void copyPos(globalPos* aLatLon);
```
Makes this position the same as the passed in position.

```
void copyLat(globalPos* aLatLon);
```
Makes this latitude the same as the passed in position's latitude.

```
void copyLon(globalPos* aLatLon);
```
Makes this longitude the same as the passed in position's longitude.

```
void setLatValue(const char* inLatStr);
```
This is looking for a string in the formatted like DD MM.MMM. This does NOT look for quadrant. It just sets the numerical value, if possible.

```
void setLatQuad(const char* inQuad);
```
This is looking for "N" or "NORTH", "S" or "SOUTH". and only those case does not matter. If found, this will set our north south quadrant.

```
void setLonValue(const char* inLonStr);
```
This is looking for a string in the formatted like DD MM.MMM. This does NOT look for quadrant. It just sets the numerical value, if possible.

```
void setLonQuad(const char* inQuad);
```

This is looking for "E" or "EAST", "W" or "WEST". and only those. Case does not matter. If found, this will set our east west quadrant.

```
void setLat(double inLat);
```
This takes a signed latitude angle and sets our latitude from that.

```
void setLon(double inLon);
```
This takes a signed longitude angle and sets our latitude from that.

```
void setPosValues(const char* latStr,const char* lonStr);
```
Hell I don't know. I really have to re-write this library and simplify it a bunch.

```
void setQuads(const char*  inLatQuad,const char*  inLonQuad);
```
Looking for N or North or S or South for latitude, along with E or East or W or West for longitude.

```
void setPos(double inLat, double inLon);
```
Given signed latitude and longitude values, set this as our location.

```
void setLatValue(int inLatDeg, double inLatMin);
```
Given latitude degrees and minutes, set this as our latitude.

```
void setLatQuad(quad inLatQuad);
```
Given a latitude quad, set this as our latitude quadrant.

```
void setLonValue(int inLonDeg, double inLonMin);
```
Given longitude degrees and minutes, set this as our longitude.

```
void setLonQuad(quad inLonQuad);
```
Given a longitude quad, set this as our longitude quadrant.

```
void setQuads(quad inLatQuad,quad inLonQuad);
```
Given both latitude and longitude quads, set them as our own.

```
void setPosition(int inLatDeg, double inLatMin, quad inLatQuad, int inLonDeg, double inLonMin, quad inLonQuad);
```
Given all the damn bits set this as our position.

```
double trueBearingTo(globalPos* inDest);
```
Given a destination position return the true bearing (in degrees) to that position.

```
double distanceTo(globalPos* inDest);
```
Given a destination position return the true distance (in knots) to that position.

```
char* getLatStr(void);
```

Return a formatted latitude string of our position value.

```
char* getLatQuadStr(void);
```
Return a formatted string of our position's latitude's quadrant.

```
char* getLonStr(void);
```
Return a formatted longitude string of our position value.

```
char* getLonQuadStr(void);
```
Return a formatted string of our position's longitude quadrant.
```
char* showLatStr(void);
```
Returns a displayable latitude string.

```
char* showLonStr(void);
```
Returns a displayable longitude string.

```
int getLatDeg(void);
```
Returns the integer portion of latitude degrees.

```
double getLatMin(void);
```
Returns latitude minutes. Decimal value.

```
quad getLatQuad(void);
```
Returns the quad value of our latitude position.

```
int getLonDeg(void);
```
Returns the integer portion of longitude degrees.

```
double getLonMin(void);
```
Returns longitude minutes. Decimal value.

```
quad getLonQuad(void);
```
Returns the quad value of our longitude position.

```
double getLatAsDbl(void);
```
Returns latitude as a signed degree value.

```
double getLonAsDbl(void);
```
Returns longitude as a signed degree value.

```
int32_t  getLatAsInt32(void);
```
Returns latitude formatted as an int32 for passing into a NMEA2k message.

```
int32_t  getLonAsInt32(void);
```
Returns longitude formatted as an int32 for passing into a NMEA2k message.

```
int64_t  getLatAsInt64(void);
```

Returns latitude formatted as an int64 (Kinda') for passing into a NMEA2k message.

```
int64_t  getLonAsInt64(void);
```
Returns longitude formatted as an int64 (Kinda') for passing into a NMEA2k message.

LC_Adafruit_GPS

The Left coast version of GPS drivers, originally written for the Adafruit Ultimate GPS. I needed a GPS to add to our NMEA2000 Bus on our sailboat. The one I had was the Adafruit version. It streams out NMEA 0183 text data. Seems that's a pretty common thing. So this driver was written to parse out all the GPS data possible into a GPS object, that could later be read out like values from a struct..

Now, behind the scenes, each type of message has it's own handler class to decode it. The GPSReader class is basically a struct with enough smarts to choose handlers as the data flows by.

The data is read and broken into values by the inherited numStream class. The numStream class syncs the data, grabs the first token and asks the GPSReader if it has a handler for this type of data. If so, the handler is loaded and all the rest of the message is passed to it to be decoded. When the message has competed, either a success and the data is passed to the GPSReader, or failure, the gathered data ignored. Then the process starts all over again with the next message.

Since the GPSReader is the part that most people interact with, we'll have a look it it's public interface.

GPSReader

Definitions

```
enum fixQuality {
     fixInvalid,
     fixByGPS,
     fixByDGPS
};

enum mode {
     manual,
     automatic
};

enum posModes {       // From novatel documents.
     Autonomous,
     Differential,
     Estimated,       // Means dead reckoning.
     Manual,
```

```
    notValid
};

enum modeII {
    noFix,
    twoD,
    threeD
};
```

Constructors

```
GPSReader(Stream* inStream=DEF_IN_PORT,int tokenBuffBytes=DEF_TOKEN_BYTES);
```

Public methods

```
void begin(void);
```

Used to start the decoding process.

```
void reset(void);
```

Returns internal data back to initial state.

```
bool canHandle(const char* param);
```

Used internally by the inherited numStram class to see if this next message can be decoded or not.

```
bool addValue(char* param,int paramIndex,bool isLast);
```

Used internally to pass a parameter to a handler. Also passes in if this is the last parameter of a message. It returns if decoding this parameter was a success.

Public variables

```
GPSMsgHandler* handlers[NUM_HANDLERS];
```

This is our internal array of handlers. Best to ignore this one.

```
int theHandler;
```

Current active handler. Best to ignore this one as well.

NOTE: These data values won't valid without a current "GPS fix". Actually, if you can "see" one satellite, you may get a valid dat & time.

```
int         year;    // These three are the current date.
int         month;
int         day;

int         hours;   // And, of course these three are the current time.
int         min;
float       sec;
```

```
globalPos    latLon;      // Current position (GPS Fix)
float        altitude;    // Current altitude in feet.
fixQuality   qualVal;     // See definitions.
bool         valid;       // Is this fix valid?
posModes     posMode;     // See definitions.

float        trueCourse;          // Reliable value.
float        magCourse;           // When magnetic offset is available.
float        groudSpeedKnots;     // Speed over ground, knots.
float        groundSpeedKilos;    // Speed over ground, kilometers/hour.

uint16_t     magVar;           // Magnetic variance, never seen one.
char         vEastWest;        // Again, never seen it.
float        GeoidalHeight;    // How much does the earth bulge here?
float        ageOfDGPSData;    // Not see this one.
int          DGPSStationID;    // Differential station ID.
mode         operationMode;    // See definitions.
modeII       fixType;          // NoFix, 2D, 3D.
int          numSatellites;    // Number of satellites in view.
int          SVID[11];         // IDs of satellites used in fix.
float        PDOP;             // Position (3D) delusion of precision.
float        HDOP;             // Horizontal delusion of precision.
float        VDOP;             // Vertical delusion of precision.
linkList     satInViewList;    // List of all satellites in view.
```

Some notes about DOP ratings.

DOP Value	Rating	Description
< 1	Ideal	Highest possible confidence level to be used for applications demanding the highest possible precision at all times.
1–2	Excellent	At this confidence level, positional measurements are considered accurate enough to meet all but the most sensitive applications.
2–5	Good	Represents a level that marks the minimum appropriate for making accurate decisions. Positional measurements could be used to make reliable in-route navigation suggestions to the user.
5–10	Moderate	Positional measurements could be used for calculations, but the fix quality could still be improved. A more open view of the sky is recommended.

10–20	Fair	Represents a low confidence level. Positional measurements should be discarded or used only to indicate a very rough estimate of the current location.
> 20	Poor	At this level, measurements should be discarded.

GPSMsgHandler & offspring..

GPSMsgHandler is the base class of all the GPS NMEA 0183 Handlers. There are currently 5 handlers because I only saw 5 different types of messages ever come through the stream. GPVTG, GPGGA, GPGSA, GPGSV, GPRMC. To write a handler, you must inherit GPSMsgHandler and fill out constructor, destructor, clearValues() and decodeParam() methods.

Well take GPVTG as an example.

```
// Track Made Good and Ground Speed.
class GPVTG :  public GPSMsgHandler {

      public:
                  GPVTG(GPSReader* inReader);
      virtual     ~GPVTG(void);

       virtual    void clearValues(void);
bool decodeParam(char* inParam,int paramIndex,bool lastParam);

#ifdef SHOW_DATA              // Optional
void showData(void);
#endif

      float             trueCourse;     // Variables to hold one message of data.
      float             magCourse;
      float             groudSpeedKnots;
      float             groundSpeedKilos;
      posModes          posMode;
```

```
};
```

See source code for examples of how this code works.

LC_numStream

https://github.com/leftCoast/LC_numStream/tree/main

There is a class of streaming data that contains messages starting with certain sync characters, have values that are separated by certain separator characters and are ended with end of data characters. An example of this would be the NMEA 0183 text streaming one sees from many GPS modules. In fact this is what the numStream's default values are set for.

numStreamIn is a base class used for creating classes that gather streaming data. By itself it doesn't do a lot but keep track of incoming data and the location of data values.

The public user interface.

Definitions

```
#define DEF_IN_PORT      &Serial1  // Change to fit your hardware.
#define SYNK_CHAR        '$'       // Marker of the start of a data set.
#define DELEM_CHAR       ','       // Marker between data items.
#define END_CHAR         '\n'      // Noting the end of a data set.
#define DEF_TOKEN_BYTES 20         // Size of the token buffer.
#define MAX_MS           50        // How long we allow a search.
#define MAX_MSG_BYTES    200       // Max bytes for complete message.
Default values for constructor.

enum exitStates {
     noData,
     completed,
     finalValue,
     erroredOut
};
Internally used. Different ways a search can complete.

enum errType {
     unknownErr,
     synkTimOut,
     tokenTimOut,
     noHandler,
     addValueFail,
     badChecksum  ,
     tokenOverflow,
     msgBufOverflow
};
Different ways things can go wrong.

enum states {
     lookForSynk,
```

```
    readingType,
    readingParam
};
```
Internally used. Different modes of operation. Or, "What is it up to now?".

Constructors

```
numStreamIn(Stream* inStream=DEF_IN_PORT,int tokenBuffBytes=DEF_TOKEN_BYTES);
```
Given a stream to draw from and a maximum token buffer size, this creates your numStream object.

Public methods

```
void begin(void);
```
Starts the process by hooking the object into the idle queue.

```
void reset(void);
```
Clears buffers resets values to zero and restarts the search for a sync character.

```
bool canHandle(const char* inType);
```
The first token of a message is handed in. As this is written, this returns false. Derived classes will decide if there is a handler for this current message or not.

```
bool addValue(char* param,int paramIndex,bool isLast);
```
Each subsequent token from a message is passed to this message's handler with the bool telling if this is the token of the message or not. Again, as this is written, it just returns false. The derived classes will supply handlers that will handle this method.

```
bool dataChar(char inChar);
```
returns true of this incoming char is NOT a special character. But merely a data character.

```
bool checkTheSum(void);
```
Returns true of this passes a checksum.
NOTE: This one should NOT be here! It's written for NMEA strings. It was hacked in at the end and needs to be rooted out.

```
void setSpew(bool onOff);
```
Allows the data stream to be repeated out the Serial port or not.

NOTE: Added at the last moment when we needed a working chart plotter but had no GPS but the one that was running this code. I found that by streaming the GPS data we were reading with this device out the Serial port

to the laptop. We were able to get the chart plotter to read it and show where we were on the chart. (In realtime)

```
exitStates findSynkChar(void);
Internal, returns the exit state from a sync char search of the incoming
stream.

exitStates readToken(void);
Internal, returns the exit state of a token search from the incoming stream.

void idle(void);
Internal, the engine that runs all this stuff.

void errMsg(errType inErr);
If we're allowing debug messages, this will print out a error message to the
Serial port when an error is logged.
```

LC_llama2000

SAE_J1939

OK, this one.. Can be a little complicated.

NMEA2000 Is currently a very popular protocol for marine applications. Stuff data in here, and a display over there can show it. Device control, sensors and display. You name it, it'll pipe it around your boat. Well, small bits of information. It's not good for large data like video streaming. It's more for control and sensing. The other bit is, they try to cover every conceivable data set one could ever need on a boat/ship. Then, every manufacturer of devices (That pays to use this stuff) is listed as well. So there is a lot involved here.

There is a book available that actually covers how the SAE J1939 protocol works.

A Comprehensible Guide to J1939

By Wilfred Voss.

This is the book I bought used on Amazon for writing this library. I thought it was a pretty rough learning curve. But pretty much it's all there. All the websites I've seen, that say they are a guide to this stuff, are just copies of parts of that book.

How does this all work?

NMEA2000, for boats, runs on the SAE_J1939 protocol, for heavy equipment, tractors and trucks. And all of that is actually running on an extended set of CAN Bus messages, from cars. You know, error codes, gas gauges, and suchlike.

So lets look at it from the bottom up.

Your CAN bus is a chip to chip all hardware messaging service. The extended messages are picked up by the SAE J1939 code. Some messages user doesn't see, because they are handled internally by the SAE J1939 code. The other messages, that can not be handled internally, are handed out.. To YOUR NMEA2000 handlers. Yes, handler are the bit you'll probably have to write. Unless you can find what you need here.

How do decode a NMEA2000 message? You can look at this website where a bunch of them have been decoded by others.

https://canboat.github.io/canboat/canboat.html

Each NMEA2000 message has a PGN (**P**arameter **G**roup **N**umber) They are used to tell you what kind of message is coming through. This is kinda' a lie, but from the NMEA200 point of view? Or someone setting up handlers for NMEA2000 messages? That's what they are used for.

How do I create a NMEA2000 device? (Napkin overview)

Attachment to the bus.

The NMEA2000 bus has 12V Power, Ground, Data Hi, Data Low. You will need hardware that brings this power in to run your Arduino and typically a chip that brings in your CAN bus messages from the Data lines. typically these CAN chips communicate on SPI and you will need driver code to communicate with it. Once you are able to send and receive CAN bus messages, you can configure the J1939 library for communication.

Gather information for code setup.

First you'll need your name information.

Can you change your address while running? yes/no?

What group are you in? Tractor, boat, truck?

What kind of system are you for? Control, Propulsion, Navigation?

Which function are you? AC Bus, Engine, Actuator?

What instance of this function are you? Engine 0, Engine 1, Fuel tank 0?

What is your manufacturer's ID? NMEA assigns these. I made mine up. (It's my boat!)

What is your device ID. You make these up. You get 21 bits.

What is your ECU (Controller) instance. I'd go with zero for now. Unless this has clones?

You will end up putting all this info. into your name. There are methods available to do this. And the functions, systems, group names, are all in the SAE_J1939.h file. You can choose what you need from the lists. The interesting bit is.. If your device gets into a fight over address on the bus? This name is what decides who wins and who goes and finds another address to call their own.

Go figure..

This brings us to addresses. Every device on a CAN/J1939/NMEA2k bus has a one byte address.

Destination specific addresses can be 0..239.

Broadcast to everyone sends with return address of 255.

Sending messages from single OR multiple sources to single destination use 240..255.

Sending messages from single OR multiple sources to mutiple destination use 240..255.

I have no address, use 254.

Some of this addressing stuff seems illogical. And to make things muddier.. The sending address is mixed into the PGN (**P**arameter **G**roup **N**umber) than NMEA uses to sort out message types.

So, there are three types of addressing model one can choose.

Static address : You hard code a number and that's it. everyone else on the bus must just deal with it.

Service config : You can hook "something" to the network and change to your address.

Command config : Other things on the network can change your address. (How rude!)

Self config : We set our own by looking over the network setup. (Find an unused one?)

Arbitrary config : We can do the arbitrary addressing dance. Basically fight over address numbers with others using names as a measure as to who wins the fights.

No address : We have no address, possibly just a passive listener?

Once you have your hardware, name and address figured out. You should probably create a class, derived from netObj that can:

initializes your hardware with a begin() method,

Receive CAN bus messages and encode them into message class messages. with sendMsg() method.

Send CAN bus messages from message class messages using receiveMsg() method.

And this should be an idler that polls the CAN bus hardware for messages during it's idle() method.

For example :

```
class NMEA2k : public netObj {

      public:
                  NMEA2k(int inResetPin,int inIntPin);
                  ~ NMEA2k(void);

bool begin(byte inAddr,addrCat inAddrCat,int inCS);     // Whatever needed here.
void sendMsg(message* outMsg);                          // Sends out message obj.
void recieveMsg(void);                                  // Receives message obj.
void idle(void);                                        // Runs everything.

      protected:
                  int   resetPin; // Reset pin, you need this.
                   int   intPin;   // interrupt pin. Optional.
};
```

Example recieveMsg from what I'm using.

```
void NMEA2k::recieveMsg(void) {
       message newMsg;
      int          i;
      if (CAN.parsePacket()) {                                  // If we got a packet..
            newMsg.setCANID(CAN.packetId());                    // Decode and store.
           newMsg.setNumBytes(CAN.packetDlc());                 // Set up buffer.
           i = 0;                                               // Starting at zero..
           while (CAN.available() && i < newMsg.getNumBytes()) {// While we have a bytes..
                        newMsg.setDataByte(i, CAN.read());                  // Read
and store.
```

```
                i++;                                                    // Bump
index.
            }                                                   //
             incomingMsg(&newMsg);                                  // Let netObj deal
with it.
        }
}
```

Example sendMsg from what I'm using.

```
void NMEA2k::sendMsg(message* outMsg) {

    uint32_t CANID;
    int      numBytes;

    if (outMsg) {                                       // Sanity, if this is not null.
        numBytes = outMsg->getNumBytes();               // Read num of bytes.
        if (numBytes <= 8) {                            // Only 8 bytes with CAN.
            CANID = outMsg->getCANID();                 // Grab the formatted CAN ID.
            CAN.beginExtendedPacket(CANID);             // Hardware wants this CAN
ID.
            for (int i = 0; i < numBytes; i++) {        // For each data byte..
                CAN.write(outMsg->getDataByte(i));      // Send the data byte out.
            }                                           //
            CAN.endPacket();                            // All done, close up shop.
        }
    }
}
```

Of course your methods will probably be different, depending on how your hardware works.

But in some way, you will have to read CAN messages. Filter for extended CAN messages. Then decode them into message objects to be dropped into netObj's incomingMsg() method.

And, be able to decode message object messages into extended CAN messages to send them out the CAN bus.

The next thing you will need is to decide on, are the PGNs you plan on handling. Either you will be reading or transmitting data. Decoding and encoding this data is what the msgHandlers do.

handlers for NMEA2k messages all are derived from the msgHandler class. This class knows how to hook up to your NMEA2k object. Pass message objects to and from it. Do scheduled broadcasts if necessary and will be your interface to data that flows over the NMEA2k network. You can attach as many different handlers to your NMEA2k object as you like, and have resources for.

Lets have a look at the msgHandler class you will need to extend for your different handlers.

```
// Base class for handling and creation of SAE J1939 network messages. Inherit
// this create your handler object and add them using the netObj call
// addMsgHandler().

class msgHandler :    public linkListObj {

    public:
                  msgHandler(netObj* inNetObj);
                  ~msgHandler(void);

                        // Fill in to handle incoming messages.
bool      handleMsg(message* inMsg);

                        // Fill in to create outgoing messages.
void      newMsg(void);

                        // This one just sends messages on their way.
void      sendMsg(message* inMsg);

                        // Used for broadcasting. (Zero for off)
            void      setSendInterval(float inMs);

                        // Forgot the the timer setting? This'll let you know.
            float     getSendInterval(void);

                        // Same as idle, but called by the netObj.
void      idleTime(void);

                        // Pointer to our boss!
                        netObj*  ourNetObj;
```

```
                        // If broadcasting, how often do we broadcast? (Ms)
                        timeObj  intervaTimer;
};
```

An example of a handler. This one is for water speed.

```
// ************* waterSpeedObj *************

// This listens for boat speed messages and outputs a boatspeed in knots.
// transducer used for this test was an AIRMAR DST810

class waterSpeedObj  : public msgHandler {

   public:
            waterSpeedObj(netObj* inNetObj);
  ~waterSpeedObj(void);

            float getSpeed(void);
  bool  handleMsg(message* inMsg);

            mapper   speedMap;
            float    knots;
  };
```

And the actual implementation..

```
// ************* waterSpeedObj *************


waterSpeedObj::waterSpeedObj(netObj* inNetObj)
     : msgHandler(inNetObj) {

     knots     = 0;
     speedMap.setValues(0,1023,0,(1023*1.943844)*0.01);
}

waterSpeedObj::~waterSpeedObj(void) {  }

bool waterSpeedObj::handleMsg(message* inMsg) {

     unsigned int rawSpeed;

     if (inMsg->getPGN()==0x1F503) {
          rawSpeed = inMsg->getIntFromData(1);
          knots = speedMap.map(rawSpeed);
          return true;
     }
```

```
    return false;
}

float waterSpeedObj::getSpeed(void) { return knots; }
```

So to sum up? To set up a NMEA2k system you will need..

•Hardware to bring CAN messages into your processor.

•Drives for this hardware.

•A way to get the extended packet CAN ID & 8 byte data pack from the CAN message into a messageObj from the J1939 code.

•The parameters and info needed to setup your J1939 system & code.

•Your selection of PGNs to code ans/or encode for NMEA2k messages.

•Write or locate handlers for your set of PGNs

And that should do it. As for looking at the public interface? IN this case it would probably be best to actually look at the code itself. There's a lot in there but you only interact with..

The #define constanats.

message is a very important one. You will deal that one a lot.

netName will need to used to do your setup.

netObj is huge but you should really never have to deal with it beyond extending it to tie into your hardware.

You will need to deal with msgHandlers creating them, attaching them, using them.

And all the other stuff in the middle of the set of SAE_J1939 code? For now I'd just ignore it. It'll break your brain to try to learn all that stuff for busting up large message, dealing with address fights etc etc. That should all be done behind the scenes anyway. That's what all this nonsense was built for anyway.