

USING THE LEFT COAST LIBRARIES



LC_baseTools	3
timeObj	3
mapper	4
multiMap	6
runningAvg	7
strTools	9
tempStr	10
colorObj	11
idler	15
blinker	16
mechButton	18
autoPOT	21
serialStr	23
textBuff	25
squareWave	27
resizeBuff	29
maxBuff	30
lists	32
linkListObj	32
linkList	33
Stack & queue	34
dblLinkListObj	35
LC_lilParser	38
LC_slowServo	41
LC_neoPixel	43
LC_SDTools	47
bigIndian	47
filePath	48
LC_blockFile	52

LC_baseTools

timeObj

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

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, double y1, 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 public consumption.

multiMap

Multi map is used like a mapper in that 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

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

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

Function calls

Pass in a string and this makes all the letters uppercase.

```
void upCase(char* inStr);
```

Pass in a string and this makes all the letters lowercase.

```
void lwrCase(char* inStr);
```

Allocates and makes a copy of inStr.

```
bool heapStr(char** resultStr, const char* inStr);
```

Recycles resultStr.

```
void freeStr(char** 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 external 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

colorObj gives you a common class for passing colors about from screens to neoPixels 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.

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);  
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.)

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

```

Then there are the color mappers. First 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);`
Returns a mapped color for every value of 0..100 entered.

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.

i d l e r

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 placed as the first line of your `global loop()` routine. Each object that is to become an idler needs to have its `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

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 and by default 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().

```
// Some defaults in case the user just doesn't care..  
#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

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(*func)(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

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.setCallback(gotChange);  
}
```

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

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

autoPOT's public interface..

Constructors

Set the analog pin number with the constructor.

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

Public methods

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

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

Sets a +/- value 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 setWindow(int plusMinus);
```

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.

```
void idle(void);
```

serialStr

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);
```

Public methods

```
void setCallback(void(*funct)(char*));  
void idle(void);  
bool hadOverrun(void);
```

textBuff

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.  
}
```

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);
```

Add a char.

```
bool addStr(char* inCStr, bool andNULL=true);
```

Add c string. (With or without the null terminator.)

```
char peekHead(void);
```

Look at the next char to come out.

`char peekIndex(int index);`

Have a look at the char at index. '\0' for no char.

`char readChar(void);`

Read out the next char. (removes it)

`int strlen(void);`

strlen() for the next string to read. Not counting the '\0'.

`char* readStr(void);`

Read out a c string. (removes it)

NOTE: copy result, 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?

`void clear(void);`

Dump all the chars, reset to empty.

squareWave

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 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 is 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

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 `fcatt()`, a file concatenation function. Given destination and source files, Concatenate the source onto the end of destination.

```
void fcatt(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

Pass in the total amount of bytes to move along with the minimum buffer size before calling it quits.

```
maxBuff(unsigned long numBytes,unsigned long minBytes=BYTE_CUTOFF);
```

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).

l i s t s

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.

l i n k L i s t O b j

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 add 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 an 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 an 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

dblLinkedListObj(void);
Creates a ready to use dblLinkedListObj .

Public methods

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

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

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

dblLinkedListObj* getLast(void);
Runs up the list 'till dllNext == NULL. Returns link to that node.

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

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

dblLinkedListObj* 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.

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 really 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. How to set this up?

First you need to write an 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 character.  
        Serial.print(inChar);                // Echo the character.  
        command = myParser.addChar(inChar); // Try parsing what we have.
```

```

switch (command) {
    case noCommand : break; // Check the results.
    case firstCom : handleFirst(); break; // Nothing to report, move along.
    case secondCom : handleSecond(); break; // Call handler.
    default : Serial.println("What?"); break; // Call handler.
}
}
}
}

```

```

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 {
        Serial.print("LED is ");
        Serial.println(getLEDState());
    }
}

```

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 param 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

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 to move slower than it can. This call follows the commanded angle over time not the actual angle.

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. 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 for 4 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 below.

The second class in this library? Well, set of classes actually. This second set of classes are chainPixels. Imagine you have a project that includes a bunch of pixel groups. Each needs to "play" a different pattern. Maybe some patterns 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 its 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 cool 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 are connected to the data line. Done!

Here's an example of chain pixels :

<https://wokwi.com/projects/444957517359016961>

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

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 object 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.

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 set a 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_SDTools

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 it 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.

```
extern 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 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 fcopy(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

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 that the user doesn't actually interact with directly. So I think, for now, I'm going to leave them out. See the source code for info. on the rest of the classes. pathItem, rootItem, fileItem, folderItem.

filePath is the end product of all of these and what the user would typically be interacting with. So let's have a look at filePath's user 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.

```
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.

Here is an example of these tools in action a command line file browser.

<https://wokwi.com/projects/401643432890805249>

LC_blockFile

blockFile.. 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, it 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 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.

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.

Error codes:

```
#define BF_NO_ERR          0    // Everything's fine now, ain't it?
#define BF_MEM_ERR        1    // malloc() failed.
#define BF_VERSION_ERR    2    // We have the 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, was handed path
// to directory.
#define BF_SEEK_ERR       7    // Trying to reposition the file
// pointer failed.
```

```
bool isEmpty(void);
```

returns true if there are no bytes stored in this blockFile.

Now there is also 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 saveSubFileBufs(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.