USING THE LEFT COAST LIBRARIES

Basic tools & packages

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,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

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

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 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.

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. 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.
```

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

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

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(*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

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 +/- 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 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

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

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

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);
        virtual         ~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

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

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

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).

lists

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.

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

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

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 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 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 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/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 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.

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_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.

```
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 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

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 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.

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, 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.

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   // 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.

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 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_cardIndex

If youo 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.

A look at the user interface :

Constructors

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

Public methods

```
void loadList(void);
This is used to re-shuffle the deck. Think of it like a reset.

void omitCard(int value);
Used for removing cards from the deck.
```

NOTE: Only for this current deal. Once loadList() is called it will be a full list 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 getNumRemain(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_numStream

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.
```

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. 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 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 s 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 bascialy sets up the drawing environment for the object. the drwaw() methos 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)

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 together 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 this 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.

colorRect

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 TFT 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

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

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

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

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.

Marine navigation

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 quadrent 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 adapt 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);
        virtual bool decodeParam(char* inParam,int paramIndex,bool lastParam);

#ifdef SHOW_DATA        // Optional
        virtual 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_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);

    virtual    bool  begin(byte inAddr,addrCat inAddrCat,int inCS); // Whatever needed here.
    virtual    void  sendMsg(message* outMsg);                      // Sends out message obj.
    virtual    void  recieveMsg(void);                              // Receives message obj.
    virtual    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.
    virtual  bool     handleMsg(message* inMsg);

                // Fill in to create outgoing messages.
    virtual  void     newMsg(void);

                // This one just sends messages on their way.
    virtual  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.
    virtual  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);
   virtual  ~waterSpeedObj(void);

            float getSpeed(void);
   virtual  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.