The NeXT computer offers complete preemptive multi-tasking along with a graphical user interface that is conducive to having many programs running at the same time. Additionally, it is very easy to create custom applications. It just seems natural to want to access the features of Mesa from a custom application. The Mesa Application Program Interface, Mesa API, allows a programmer to access the features of Mesa to transfer data or make Mesa perform certain tasks (like saving a worksheet, etc.) This document is a brief description of the API. The API is changing and growing. If you have suggestions that would make using the API easier or better for you, please forward them to dpp@athena.com.

To access the API, you must include the `MesaAPI.h` header file in Objective-C program that accesses Mesa. Additionally, you must link the `MesaAPI.o` file into your project. The API contains two classes: `MesaAPI` and `MesaListen`. The `MesaAPI` class allows you to manipulate information in a Mesa worksheet. You will probably not need to sub-class this object in order to effectively communicate with Mesa. `MesaListen` is a class that listens for and receives information from a `MESSAGEIF()` function. You will need to sub-class `MesaListen` to all take advantage of its functionality.

MesaListen

The MESSAGEIF() function sends a range of cells to a waiting MesaListen object running in another program. For example, let's say we have a spreadsheet that is being fed gold prices from exchanges around the world (we could easily build this with the MesaAPI object, and this is similar to the MAPIDemo program.) We have built a simple model to identify arbitraging opportunities. If the price of gold in Zurich is more than 1% greater than the price of gold in London, we want to be alerted. The following formula would be entered into a cell in the worksheet receiving the gold price updates:

```
=messageif(zurich_price / 101% > london_price,"buyzurich",zurich_price)
```

When the price of gold in Zurich was more than 1% higher than the price in London, a "Mach Message" will be send to a "port" named "buyzurich" containing the information in the range `zurich_price`. To "listen" to that port, simply subclass the `MesaListen` class, initialize an object, and when the message is sent, your object will receive it. Here's the code:

MyListen.h

```
@interface MyListen : MesaListen
{
    id myWindow;
    id priceTextField;
}

- gotMessage:(MAPIValue *)mv num:(int)num forUpperRow:(int)ur upperCol:(int)uc
    lowerRow:(int)lr lowerCol:(int)lc;
@end
```

MyListen.m

```
@implementation MyListen
- gotMessage:(MAPIValue *)mv num:(int)num forUpperRow:(int)ur upperCol:(int)uc
    lowerRow:(int)lr lowerCol:(int)lc
{
    [super gotMessage:mv num:num forUpperRow:ur upperCol:uc lowerRow:lr
    lowerCol:lc];

    [myWindow makeKeyAndOrderFront:self];  // bring application to front

    // display the current gold price
    [priceTextField setDoubleValue:mv -> values.number];

    return self;
    }
@end
```

in your `appDidInit:` method, instantiate the object:

```
    myListen = [[MyListen alloc] initToPort:"buyzurich"];
```

and in your `appWillTerminate:` method, free the object:

```
    [myListen free];
```

And that's all there is too it.

MesaAPI

The Mesa API object allows you to connect to a worksheet and then perform various operations on the worksheet.   When you initialize a MesaAPI object, you must specify a path to a valid Mesa file or initialize to a new worksheet.   If Mesa is not running, the Workspace Manager is asked to run a copy of Mesa.   If Mesa is loaded and can open the worksheet, the object is initialized, otherwise a NULL pointer is returned.

```
- initToWorksheet:(const char *)ws;
- initToNewWorksheet;
```

Used instead of the `init` method.   Connects the object to a worksheet with the name `ws` or to a new worksheet (an Untitled worksheet.)   If Mesa cannot be run or the worksheet cannot be opened, `NULL` is returned.

```
- free;
```

Disconnects the object from the worksheet and frees the object.

```
- (int)recalc;
```

Recalculates and re-displays the worksheet.   The worksheet is not recalculated or re-displayed after information is changed using an API method.   To keep the information on the worksheet current, you much either re-display or recalculate the sheet after you have changed it.

```
- (int)displaySheet;
```

Re-displays the worksheet.   The worksheet is not recalculated or re-displayed after information is changed using an API method.   To keep the information on the worksheet current, you much either re-display or recalculate the sheet after you have changed it.

```
- (int)saveSheet;
```

Saves the worksheet to disk using its current file name.

```
- (int)saveSheetAs:(const char *)name;
```

Renames the worksheet and saves it to disk.

```
- (int)hide;
```

Hides the Mesa application.

```
- (int)makeKeyAndOrderFront;
```

Brings a window in the worksheet to the front of the window list.   If Mesa is not the active application, it becomes the active application.

```
- (int)getValues:(MAPIValue **)mv num:(int *)n
    upperRow:(int)ur col:(int)uc lowerRow:(int)lr col:(int)lc;
```

Gets an array of values from the spreadsheet from the range.   Supply the bounding rows and columns.   The number of items will be placed in *n and the items will be in an array pointed to by mv.   Each element in the array is in the format:

```
enum {stringMAPIValue, numberMAPIValue, errorMAPIValue};

typedef struct _MAPIValue {
    int type;
    short row,col;
    union {
        char *string;
        double number;
        int error;
        } values;
    } MAPIValue;
```

When you are done with the information, call `freeMAPIValue(int n,MAPIValue *mv)` to free the array.

```
- (int)getValues:(MAPIValue **)mv  forLabel:(const char *)l num:(int *)n
    upperRow:(int *)ur col:(int *)uc lowerRow:(int *)lr col:(int *)lc;
```

Gets an array of values from the spreadsheet from the named range.   Supply the range name.   The number of items will be placed in *n and the items will be in an array pointed to by mv, and the bounds of the range will be returned in `ur`, `uc`, `lr`, and `lc`.   Each element in the array is in the format:

```
enum {stringMAPIValue, numberMAPIValue, errorMAPIValue};

typedef struct _MAPIValue {
    int type;
    short row,col;
    union {
        char *string;
        double number;
        int error;
        } values;
    } MAPIValue;
```

When you are done with the information, call `freeMAPIValue(int n,MAPIValue *mv)` to free the array.

```
- (int)putValues:(MAPIValue *)mv num:(int)n
```

```
    upperRow:(int)ur col:(int)uc lowerRow:(int)lr col:(int)lc;
```

Takes a array of values and places it in the worksheet. mv is the array of values. There are n of them. They are to be placed in the worksheet in the range `ur`, `uc`, `lr`, and `lc`. Mesa removes all cells from that range and then inserts the supplied cells. Make sure to set the `row` and `col` for each cell. If a cell is not in the range, it will not be inserted.

```
- (int)getLabels:(char ***)lp ranges:(short **)r num:(int *)num;
```

Gets all the labels for the worksheet. An example follows:

```
char **labels;
short *ranges;
int num,x;

if ([myobject getLabels:&label ranges:&ranges num:&num] == MAPISuccess) {
    for (x = 0; x < num; x++) {
        //  print the info
        printf("%d. %s upperRow:%d col:%d lowerRow:%d col:%d\n",
            x + 1,labels[x],ranges[x * 4],ranges[x * 4 + 1],
            ranges[x * 4 + 2], ranges[x * 4 + 3]);

        // free the string
        free(labels[x]);
        }

    // free the arrays
    free(labels);
    free(ranges);
    }
```

Please remember that you must manually free the label strings and the arrays or else you'll have memory leaks.

```
- (int)associateLabel:(const char *)l withItems:(char **)it
```

```
    andValues:(MAPIValue *)mv offset:(int)off numItems:(int)ni orientation:(int)or;
```

Associates a set of string items with a set of values. The label is a Mesa named range defining where the look-up should take place on the worksheet. Items is an array of strings containing, for example, stock prices. The values are values to be placed at offset rows or columns from the top or left of the matching cell. There are ni items in the list and the association will be either `horizontalOrientation` or `verticalOrientation`. An example of this is used in the MAPIDemo program to pump stock prices into the Mesa worksheet.

The following three methods get an EPS image for a range of cells or a named graph in the worksheet. vp is set to an NXImage object if the message is successful or NULL if not. You can use -`composite:toPoint:` to place the image into a view in your custom app. Remember to free the NXImage object after you are done using it (Mesa allocates the object, it your responsibility to free it.)

```
- (int)getEPSForLabel:(char *)l in:(id *)vp;
- (int)getEPSForUpperRow:(int)ur col:(int)uc lowerRow:(int)lr col:(int)lc in:(id
*)vp;
- (int)getEPSForGraph:(char *)l in:(id *)vp;
```

This method prints a named report from Mesa. You must create the named report first. This is useful to create a good looking report, have the custom app place report data into the worksheet and then print it out.

```
- (int)printReport:(char *)r;
```

Additional function:

```
void freeMAPIValue(int num,MAPIValue *mv);
```

This function frees the memory allocated for returned values from a message to `getValues:num:upperRow:col:lowerRow:col:` or `getValues:forLabel:num:upperRow:col:lowerRow:col:`. After you are done with the information generated by these methods, call this function with the number of items returned and the array that they were placed in. Please remember to do this to avoid nasty memory leaks.

Errors:

`MAPISuccess` - the operation was successful

`MAPISendError` - the message could not be sent to Mesa. The application was probably terminated

`MAPIReceiveError` - no answer was received in 30 seconds. The application may have crashed or is very busy.

`MAPILabelNotFoundError` - the named range was not found in the range look-up table.

`MAPISheetClosedError` - the sheet was closed and the link should be shut down.

`MAPINotImplError` - this method is currently not implemented.

This is an impassioned plea: Please, please, please, use only the functions and methods documented. These are the only ones that will be supported in the future. If you need functionality that is not provided, please send e-mail or call. The last thing any of us need is a program that doesn't work at all because it took advantage of an undocumented feature!!