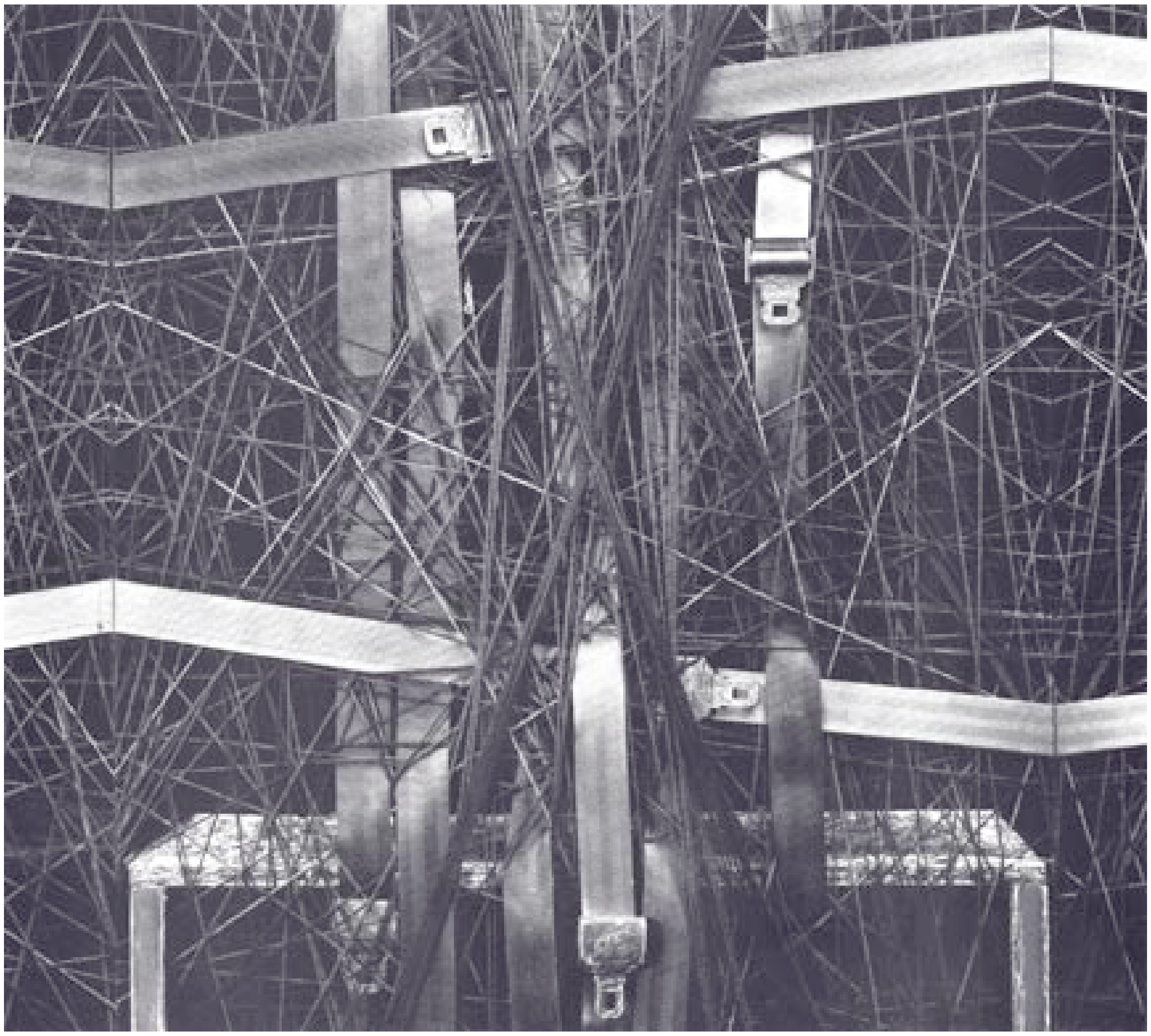
The pattr SDK



Revision 1.0 / 5 October 2005

Table of Contents

Copyright and Trademark Notices

Credits

SDK Documentation and Reference: Jeremy Bernstein

Cover Design: Lilli Wessling Hart

Cover Photo: Sue Costabile

Graphic Design: Gregory Taylor

How pattr functions (briefly)

The **pattr** object operates in two modes: unbound and bound. Unbound **pattr** objects are self-standing, and will not be discussed in this document. Bound **pattr** objects (and the **autopattr** object, which is essentially a multiply-bound **pattr** object), however, require some explanation.

When binding to another object's data, **pattr** queries the object to determine whether it supports the pattr API (which is simply used to send and receive data between **pattr** and the target object). Assuming that the object supports this API, the **pattr** binds. At that point, it is the responsibility of the target object to inform **pattr** of any state changes -- the **pattr** object WILL NOT automatically query target objects, in the interest of efficiency. The **pattrstorage** object also uses these notifications to update its state and interface.

The API for the pattr system is, in fact, very simple and most Max objects can quickly be 'retrofitted' for pattr-compatibility with very little trouble. Developers of objects which maintain very complex state information, however, should consider using attributes. Attributes are pattr-aware as-is. However, using attributes requires some extra preparation and changes in object definition.

Adding pattr-awareness when developing Max objects in C

Developers of normal Max objects in C can easily add pattr-compatibility to their objects. The first step is the add the following #include to your source code:

```
#include "ext_obex.h"
```

The "ext_obex.h" header defines a number of types and utility functions, the existence of which are assumed in the examples below and required for pattr-compatibility.

Optionally, add the source code file *commonsyms.c* (in *c74support/max-includes/common/*) to your project. This, plus a call to `common_symbols_init()` in `main()`, will initialize a number of useful symbols (see the header file, *commonsyms.h*, for a list). This document assumes that the common symbols are available.

The actual pattr-compatibility is simply accomplished by implementing 2 special functions as `A_CANT` methods.

Using the traditional Max object interface:

```
	addmess((method)myobj_getvalueof, "getvalueof", A_CANT, 0);
	addmess((method)myobj_setvalueof, "setvalueof", A_CANT, 0);
```

Using the obex object interface (discussed in further detail below):

```
	class_addmethod(myobj_class, (method)myobj_getvalueof,
		"getvalueof", A_CANT, 0);

	class_addmethod(myobj_class, (method)myobj_setvalueof,
		"setvalueof", A_CANT, 0);
```

Regardless of the object interface in use, the `getvalueof()` and `setvalueof()` methods should be prototyped as:

```
	t_max_err myobj_setvalueof(t_myobj *x, long ac, t_atom *av);
	t_max_err myobj_getvalueof(t_myobj *x, long *ac, t_atom **av);
```

The `setvalueof()` method is used to set your object's state, while `getvalueof()` queries it. A very simple sample implementation is shown below, for an object whose state is defined by a single floating point value.

```
t_max_err myobj_setvalueof(t_myobj *x, long ac, t_atom *av)
{
	if (ac && av) {
		// simulate receipt of a float value
		myobj_float(x, atom_getfloat(av));
	}
	return MAX_ERR_NONE;
}

t_max_err myobj_getvalueof(t_myobj *x, long *ac, t_atom **av)
{
	if (ac && av) {
		if (*ac && *av) {
			// memory has been passed in; use it.
		} else {
			*av = (t_atom *)getbytes(sizeof(t_atom));
		}
		*ac = 1; // our data is a single floating point value
		atom_setfloat(*av, x->objvalue);
	}
	return MAX_ERR_NONE;
}
```

Objects may, of course, contain more than a single value. Here are the same methods, modified to support an object whose value comprises 3 integer values: red, green and blue.

```
t_max_err myobj_setvalueof(t_myobj *x, long ac, t_atom *av)
{
	if (ac && av && ac >= 3) {
		myobj_red(x, atom_getlong(av));
		myobj_green(x, atom_getlong(av + 1));
		myobj_blue(x, atom_getlong(av + 2));
	}
	return MAX_ERR_NONE;
}

t_max_err myobj_getvalueof(t_myobj *x, long *ac, t_atom **av)
{
	if (ac && av) {
		if (*ac && *av) {
			// memory has been passed in; use it.
		} else {
			*av = (t_atom *)getbytes(sizeof(t_atom) * 3);
		}
		*ac = 3; // three integer values
		atom_setfloat(*av, x->red);
		atom_setfloat(*av + 1, x->green);
		atom_setfloat(*av + 2, x->blue);
	}
	return MAX_ERR_NONE;
}
```

IMPORTANT: Note that in the two `setvalueof` examples above, we use the `getbytes()` function to allocate memory. Users *must* use this function. Other memory allocation functions will almost certainly cause instability or crashing.

Finally, **pattr** needs to be informed of any object state changes, to know when it should call the `getvalueof()` method. With Max 4.5, a generalized server/client notification mechanism was introduced, which **pattr** uses for this purpose. The details of this mechanism are beyond the scope of this document (although further details are given in the Obex Reference in Section D), and for the purposes of **pattr** implementation are transparent.

In order to inform **pattr** that the object's state has changed, developers need only call the `object_notify()` function, more or less directly after the state change has occurred. However, the function should be called differently, depending on whether your object is a UI object or not.

If your object is a UI object, call the function simply as:

```
object_notify(x, _sym_modified, NULL);
```

`x`	The pointer to your object
`_sym_modified`	Defined in *commonsyms.c*, equivalent to `gensym("modified")`

The symbol `_sym_modified` is not explicitly part of the obex API, but can be used, along with a number of other useful, predefined symbols, by adding the *commonsyms.c* source code file to your project, and making a call to `common_symbols_init()` in `main()`. The examples in this SDK assume that the developer has done this.

If your object is *not* a UI object, call the function as:

```
object_notify(x->b, _sym_modified, NULL);
```

`x->b`	The pointer to your object's enclosing box.
`_sym_modified`	Defined in commonsyms.c, equivalent to `gensym("modified")`

In this case, you'll need to store the pointer to your object's enclosing box in your new() method. The value is located in the s_thing field of the t_symbol * gensym("#B"). For instance:

```
void *myobj_new(void)
{
      t_myobj *x = NULL;

      if (x = (t_myobj *)newobject(myobj_class)) {
            x->b = (t_box *)gensym("#B")->s_thing;
            ...
      }
      return x;
}
```

This will cause pattr to call your object's `getvalueof()` method. For instance, in the theoretical UI object `myobj`, the function `myobj_float()` might look like this:

```
void myobj_float(t_myobj *x, double d)
{
      x->objvalue = d;
      object_notify(x, _sym_modified, NULL);
      outlet_float(x->floatout, d);
}
```

You'll want to make sure that the call to `object_notify()` occurs *after* any code which would cause state changes to your object. Otherwise, your object's `getvalueof()` method might get called before the object has fully updated.

Please refer to the included source code example *myobj* for a working example of a traditional Max object with pattr-support using the techniques described above.

Adding attributes when developing Max objects in C

Attributes will be well-known to some developers from their use in Jitter. In Max 4.5, attributes were added to the kernel code, as well. A full discussion of attributes and their use, from a user-standpoint, can be found in the Max45GettingStarted.pdf document that ships with Max. From the developer standpoint, attributes simplify certain elements of object design, and provide automatic pattr-awareness for their data. However, implementing attributes requires some additional work on the part of the developer.

The **pattr** object is able to bind to any Max or Jitter attribute (Jitter attributes are supported in Jitter 1.5 and above, using the **pattr** objects which shipped with Max 4.5.5 or later), using the *bindto* syntax 'bindto [objectname]::[attributename]'. If the attribute [attributename] is found in the object [objectname], **pattr** will bind, *even if the object doesn't directly support* **pattr** *using the techniques described above*. This is quite useful.

Starting with Max 4.5.5 the **autopattr** object has an attribute (@greedy) which causes the object to expose any and all attributes within client objects connected to the **autopattr** object's include inlet. If you want your object to be **autopattr**-compatible, but don't want to use attributes or require a direct connection between the **autopattr** and your object, your object will need to directly support the pattr API using the `getvalueof()` and `setvalueof()` methods.

With Max 4.5, we introduced an extension to the Max object class, called *obex*, which must be used by developers who wish to support attributes. The traditional API for defining objects does not support the definition of attributes, and developers who wish to use them will need to move their objects over to the obex API.

What is obex?

The traditional means of declaring Max classes with `setup()`, `addmess()`, and friends is limited to declaring a single class per external, whose class name is the same name as the external's file on disk, and which is instantiable in object boxes. To create any other classes, class definitions need to be filled out by hand, and knowledge of these "boxless" classes is restricted to the external which defined them. An example of such a manually created "boxless" class is demonstrated in the *coll* external source code provided with the SDK.

Another limitation of the traditional Max class is that the object state for such classes is somewhat difficult to identify and access, as there is only the notion of a verb (message), and not the notion of a noun (attribute) which would represent some portion of object state.

The obex extensions to Max classes solve these two problems—i.e. they permit the creation and registration of "boxless" and "box" classes in such a way that is independent from the way in which externals are loaded, and permit the definition of attributes as object state.

Additionally, objects can be further extended via the "obex" pointer, which is essentially a hashtable containing keyword/object pairs. This has been used for storing objects related to the dump outlet, attributes, and is otherwise used internal to object maintenance.

Using the obex API, it is also possible to register objects, to which other objects may attach as clients. Such clients are notified when the registered objects they are attached to change. This object registration feature is heavily used by the pattr system.

In order to implement these changes, the API for defining and accessing classes needed to change. Note that if you do not wish to make use of any of the new extensions, the traditional API is still supported and compatible in most situations.

Following is an overview of steps developers should take to obex-ify their objects, followed by a more detailed reference.

First Steps

1. Add the following include to your code:

   ```
   #include "ext_obex.h"
   ```

2. Add an obex pointer to your object's struct:

   The obex member provides additional functionality to your Max object, such as attribute management, etc., and is defined as a void *. Conventionally, you'll add obex as the second item of the object struct, but that position is not required:

   ```
   typedef struct _myobex
   {
         t_object    ob;
         void        *obex;
         ...
   } t_myobex;
   ```

In the main() Function

3. Define your object class:

In order to define a class for use with the obex API, the traditional function `setup()` must be replaced with `class_new()`. `class_new()` is prototyped as:

```
t_class *class_new(char *name, method mnew, method mfree,
      long size, method mmenu, short type, ...);
```

`class_new()` is similar to `setup()`, but note that:

- `class_new()` returns a `t_class *`, which must be registered (see below).
- `class_new()` takes a C-string 'name' argument as its first argument, rather than a `t_messlist **`.

For example, to create a new obex class, *myobex*, which takes an `A_GIMME` argument list, the following `class_new()` function could be used:

```
t_class *c = class_new("myobex", (method)myobex_new,
      (method)myobex_free, sizeof(t_myobex), (method)0L,
      A_GIMME, 0);
```

4. Set the obex offset:

The byte offset of the obex class member must be registered with the class using the `class_obexoffset_set()` function. The helper macro `calcoffset()` is typically used:

```
class_obexoffset_set(c, calcoffset(t_myobex, obex));
```

4a. (optional) Initialize common symbols:

Add the source code file *commonsyms.c* (in *c74support/max-includes/common/*) to your project, and add the call

```
common_symbols_init();
```

to `main()`. This will initialize a number of useful symbols (see the header file, *commonsyms.h*, for a list). This document assumes that the common symbols are available.

5. Define and register methods:

Instead of using `addmess()`, `addint()`, `addfloat()`, etc. to define methods, classes using obex must use the function `class_addmethod()` to define and register traditional-style Max methods:

```
t_max_err class_addmethod(t_class *x, method m, char *name, ...);
```

For developers porting older objects, the `addmess()` call can be easily modified. For instance:

```
addmess((method)myobj_test, "test", A_GIMME, 0);
```

becomes, using the obex interface:

```
class_addmethod(c, (method)myobex_test, "test", A_GIMME, 0);
```

The first argument (`c`, in this example) is the `t_class *` returned by `class_new()`, above.

All of the types supported by addmess() are also supported by class_addmethod() – `A_LONG`, `A_FLOAT`, `A_SYM`, `A_DEFLONG`, `A_DEFFLOAT`, `A_DEFSYM`, `A_GIMME`, `A_CANT`.

Note: `addint()`, `addfloat()` and `addbang()` need to be replaced with `class_addmethod()` calls, as well:

```
class_addmethod(c, (method)myobex_bang, "bang", 0L);
class_addmethod(c, (method)myobex_int, "int", A_LONG, 0L);
class_addmethod(c, (method)myobex_float, "float", A_FLOAT, 0L);
```

Technical Note: Similarly, the `addinx()` and `addftx()` functions are replaced by `class_addmethod()`. The special method names "in1", "in2", … "in9" are used to bind methods for integers; "ft1", "ft2", … "ft9" are used for floats. For example:

```
// bind integer at 2nd inlet to myobex_in1 method
class_addmethod(c, (method)myobex_in1, "in1", A_LONG, 0L);
// bind float at 3rd inlet to myobex_ft1 method
class_addmethod(c, (method)myobex_ft1, "ft2", A_FLOAT, 0L);
```

Developers building new objects are advised to use Max's proxy inlets.

6. Define and register attributes:

Defining attributes is slightly more complex than defining methods. There are, in fact, three different types of attributes: *attribute, attr_offset* and *attr_offset_array*. In most situations, you'll use the *attr_offset* or *attr_offset_array* types. Attributes are, in fact, objects, and respond to various functions and methods. See the API reference, below, for more information.

The *attribute* type is used for an attribute which stores its own data—the data may be of any supported type or length. The *attr_offset* and *attr_offset_array* types **do not store their own data**, but instead refer to some other data, usually a value or values in the object struct, defined by an offset into the struct. The *attr_offset* type is used for an offset-attribute which returns a single value of any type supported. The *attr_offset_array* type is used for an offset-attribute which returns multiple values (a list or vector).

IMPORTANT: in order to support attribute arguments (attribute setting on the object's "command line" during instantiation using the @attribute syntax), objects using attributes **MUST BE DEFINED** as `A_GIMME` in `class_new()`.

To instantiate an *attribute* object, the function `attribute_new()` is used:

```
t_object *attribute_new(char *name, t_symbol *type, long flags,
     method mget, method mset);
```

To instantiate an *attr_offset* object, the function `attr_offset_new()` is used:

```
t_object *attr_offset_new(char *name, t_symbol *type, long flags,
     method mget, method mset, long offset);
```

To instantiate an *attr_offset_array* object, the function `attr_offset_array_new()` is used:

```
t_object *attr_offset_array_new(char *name, t_symbol *type,
     long flags, method mget, method mset, long offsetcount,
     long offset);
```

If the `mget` or `mset` arguments are `NULL`, default get and set methods will be used. In the case of the *attr_offset* and *attr_offset_array* attributes, the `offset` argument is the offset, in bytes, of the start of the data within the object struct. In the *attr_offset_array* type, the `offsetcount` argument is the offset, in bytes, of the number of members in the data vector (you'd store this number in your object's struct, as well). The macro `calcoffset()` can be used to calculate these offsets easily.

Regardless of the attribute type, the newly created attribute must be registered with the object class, or it will be unusable:

```
class_addattr(c, the_new_attr);
```

For instance, suppose that our *myobex* object is defined as so:

```
typedef struct _myobex
{
      t_object   ob;
      void       *obex;
      long       ldata;     // single long value
      float      fdata[4];  // vector of 4 float values
      long       fdatacnt;  // number of values in fdata (== 4)
      t_object   *attr;     // attribute, manages own data
} t_myobex;
```

We can set up 3 different attributes, as in the following example:

```
t_object *attr;
long attrflags = 0;

attr = attr_offset_array_new("float_data",  // attr name
                   _sym_float32, // attr type (32-bit float)
                   4,            // max size of array
                   attrflags,    // flags
                   (method)0L,   // default get method
                   (method)0L,   // default set method
                   calcoffset(t_myobex, fdatacnt), // count
                   calcoffset(t_myobex, fdata)); // data
class_addattr(c, attr);

attr = attr_offset_new ("long_data",   // attr name
                   _sym_long,    // attr type (long)
                   attrflags,    // flags
                   (method)0L,   // default get method
                   (method)0L,   // default set method
                   calcoffset(t_myobex, ldata)); // data
class_addattr(c, attr);

attr = attribute_new ("atom_data",     // attr name
                   _sym_atom,    // attr type (atom)
                   attrflags,    // flags
                   (method)0L,   // default get method
                   (method)0L);  // default set method
class_addattr(c, attr);
```

Note that the value of attributes created with the `attribute_new()` call, and added to the class with `class_addattr()`, is globally available to the entire class. Setting its value from any instantiated object of your class will change its value for all class members. Because the other two types of attributes merely reference data (based on the data's offset within the object's struct), they do not exhibit this behavior. To create an object-specific attribute with the `attribute_new()` function, you'll want to register the attribute with the object, instead of the class. See below in the obex API reference for more details.

Attribute flags are discussed in more detail below.

7. Add support for dumpout and quickref functionality:

In order for your object to use the dumpout outlet and display attributes in the quickref pop-up menu, you simply need to define these two methods:

```
class_addmethod(c, (method)object_obex_dumpout, "dumpout",
     A_CANT, 0);

class_addmethod(c, (method)object_obex_quickref, "quickref",
     A_CANT, 0);
```

Note that the developer need not define the `object_obex_dumpout()` and `object_obex_quickref()` functions – they are exported from the Max kernel.

8. Register your class:

Finally, in order for Max to become *aware* of your class, it must be registered, using the `class_register()` function:

```
t_max_err class_register(t_symbol *name_space, t_class *x);
```

The `name_space` argument should either be the constant **CLASS_BOX**, for obex classes which will instantiate inside of a Max patcher (e.g. boxes, UI objects, etc.), or the constant **CLASS_NOBOX**, for classes which will only be used internally. In general, the name_space `CLASS_BOX` will be used.

IMPORTANT: The `class_register()` function should be called at the end of `main()`, only after all methods and attributes have been defined.

So, in the case of the above *myobex* class, the following is called at the end of `main()`:

```
class_register(CLASS_BOX, c);
```

As with the old class model, you'll want to store your object's class in a global variable.

In the new() Method

9. Create an instance of your object class:

For classes defined with the obex API function `class_new()`, the traditional function `newobject()`, used to instantiate an object class, must be replaced with `object_alloc()`:

```
void *object_alloc(t_class *c);
```

The usage is identical.

10. Create a dumpout outlet:

The dumpout outlet is used by objects with attributes to output responses to 'get' queries. It is, in fact, just a general outlet, but it needs to be registered with the obex object. It is best created as below:

```
object_obex_store(x, _sym_dumpout, outlet_new(x, NULL));
```

The first argument (`x`, here), is the object pointer returned by `object_alloc()`. Conventionally, the dumpout outlet is the rightmost outlet on an object. Consequently, you'll typically create your dumpout outlet first, before any other outlets.

11. Process attribute arguments:

Attributes are typically specified directly in an object's box using the @attrname syntax. In order for attributes to be properly read, the following function must be called:

```
attr_args_process(x, ac, av);
```

The first argument (`x`) is the object pointer returned by `object_alloc()`. The second and third arguments (`ac`, `av`) are the atom list arguments of your `new()` method. Remember that your object's message list must have been defined as `A_GIMME` in the call to `class_new()` in `main()`.

A Sample Listing

Here's a very simple, but complete, code listing for a bare-bones new-style Max object which simply multiplies an incoming value (int or float) by an operand defined by an attribute:

```
#include "ext.h"
#include "ext_obex.h"

typedef struct _testobex
{
      t_object   ob;
      void       *obex;
      float      operand;
      void       *out;
} t_testobex;

void *class_testobex;

void *testobex_new(t_symbol *s, long ac, t_atom *av);
void testobex_free(t_testobex *x);
void testobex_int(t_testobex *x, long d);
void testobex_float(t_testobex *x, double f);

void main(void)
{
      t_class *c;
      t_object *attr;
      long attrflags = 0;

      // define the class - A_GIMME required for attributes
      c = class_new("testobex", (method)testobex_new,
            (method)testobex_free, sizeof(t_testobex), (method)0L,
            A_GIMME, 0);

      // register the obex object's offset
      class_obexoffset_set(c, calcoffset(t_testobex, obex));

      // create and register an attribute (attr_offset)
      attr = attr_offset_new("operand", _sym_float32, attrflags,
            (method)0L, (method)0L, calcoffset(t_testobex, operand));
      class_addattr(c, attr);

      // create and register some methods
      class_addmethod(c, (method)testobex_int, "int", A_LONG, 0);
      class_addmethod(c, (method)testobex_float, "float", A_FLOAT, 0);


      // add dumpout and quickref methods
      class_addmethod(c, (method)object_obex_dumpout, "dumpout",
            A_CANT, 0);
      class_addmethod(c, (method)object_obex_quickref, "quickref",
            A_CANT, 0);

      // register the class as an instantiable box
      class_register(CLASS_BOX, c);

      // set the global class variable
```

```
    testobex_class = c;
}

void *testobex_new(t_symbol *s, long ac, t_atom *av)
{
    t_testobex *x = NULL;

    // allocate a new object
    if (x = (t_testobex *)object_alloc(testobex_class)) {
        // important: initialize class members before processing
        // attribute args. otherwise, you might replace data.
        x->data = 0;

        //handle attribute arguments (e.g. @operand 0.5)
        attr_args_process(x, ac, av);

        // create a dumpout outlet, used by attributes
        object_obex_store(x, _sym_dumpout, outlet_new(x, NULL));

        // create another outlet
        x->out = outlet_new(x, NULL);

    }
    return x;
}

void testobex_free(t_testobex *x)
{
    ; // nix
}

void testobex_int(t_testobex *x, long d)
{
    // int in, int out
    outlet_int(x->out, (long)((float)d * x->data));
}

void testobex_float(t_testobex *x, double f)
{
    // float in, float out
    outlet_float(x->out, f * x->data);
}
```

Please refer to the included source code example *myobex* for a more complete working example of a new-style Max object with methods and attributes (with overridden get and set methods).

The obex API reference (preliminary)

The following preliminary API reference is not exhaustive. But it describes the principal points of the new obex API for Max objects in C.

Header Files (in addition to those required by the Max API)

ext_obex.h

Optionally, *commonsyms.c* can be added to your project file, for additional symbol definitions. You'll find this file in " The header for commonsyms.c (commonsyms.h) is included by *ext_obex.h*. Although symbols are declared, they will not be defined or useful (although your code may compile) unless you call

```
common_symbols_init();
```

in `main()`.

Class Routines

class_new

Use the `class_new` function to initialize your class by informing Max of its name, instance creation and free functions, size and argument types. Developers wishing to use obex class features (attributes, etc.) *must* use `class_new` instead of the traditional `setup` routine.

```
t_class *class_new(char *name, method mnew,
     method mfree, long size, method mmenu,
     short type, ...);
```

`name`	Your class's name, as a C-string
`mnew`	Your instance creation function
`mfree`	Your instance free function
`size`	The size of your object's data structure in bytes. Usually you use the C sizeof operator here.

`mmenu`	The function called when the user creates a new object of your class from the Patch window's palette (UI objects only). Pass 0L if you're not defining a UI object.
`type`	A standard Max *type list* as explained in Chapter 3 of the Writing Externals in Max document (in the Max SDK). The final argument of the type list should be a 0. *Generally, obex objects have a single type argument*, `A_GIMME`, followed by a 0.

This routine returns the class pointer for your new object class. *This pointer is used by numerous other functions and should be stored in a global variable.*

class_free

Use the `class_free` function to free a previously defined object class. *This routine is not typically used by external developers.*

```
t_max_err class_free(t_class *c);
```

`c`	Your class pointer

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

class_register

Use the `class_register` function to register a previously defined object class. This routine is required, and should be called at the end of `main()`.

```
t_max_err class_register(t_symbol *name_space,
     t_class *c);
```

`name_space`	Either the constant `CLASS_BOX`, for obex classes which will instantiate inside of a Max patcher (e.g. boxes, UI objects, etc.), or the constant `CLASS_NOBOX`, for classes which will only be used internally. In general, the name_space `CLASS_BOX` will be used.

`c`	Your class pointer

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

class_addmethod

Use the `class_addmethod` function to bind a method to a previously defined object class.

```
t_max_err class_addmethod(t_class *c, method m,
      char *name, ...);
```

`c`	Your class pointer
`m`	Function to be called when your method is invoked
`name`	C-string defining the message (message selector)
`...`	One or more integers specifying the arguments to the message, in the standard Max type list format (see Chapter 3 of the Writing Externals in Max document for more information).

The `class_addmethod` routine functions essentially like the traditional `addmess` routine, adding the function pointed to by `m`, to respond to the message string `name` in the leftmost inlet of your object. This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

class_addattr

Use the `class_addattr` function to bind an attribute to a previously defined object class.

```
t_max_err class_addattr(t_class *c, t_object *attr);
```

`c`	Your class pointer

`attr`	The attribute to bind. The attribute will be a pointer returned by `attribute_new`, `attr_offset_new` or `attr_offset_array_new`.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

class_obexoffset_set

Use the `class_obexoffset_set` function to register the byte-offset of the obex member of your class's data structure with the previously defined object class. Use of this routine is required for obex-class objects. It must be called from `main()`.

```
void class_obexoffset_set(t_class *c, long offset);
```

`c`	Your class pointer
`offset`	The byte-offset to the obex member of your object's data structure. Conventionally, the macro `calcoffset` is used to calculate the offset.

class_obexoffset_get

Use the `class_obexoffset_get` function to learn the byte-offset of the obex member of your class's data structure.

```
long class_obexoffset_get(t_class *c);
```

`c`	Your class pointer

This routine returns the byte-offset of the obex member of your class's data structure.

class_name_get

Use the `class_name_get` function to learn the name of a class, given the class's pointer.

```
t_symbol *class_nameget(t_class *c);
```

`c`	Your class pointer

If successful, this routine returns the name of the class as a t_symbol *.

class_findbyname

Use the `class_findbyname` function to find the class pointer for a class, given the class's namespace and name.

```
t_class *class_findbyname(t_symbol *name_space,
     t_symbol *classname);
```

`name_space`	The desired class's name space. Typically, either the constant `CLASS_BOX`, for obex classes which can instantiate inside of a Max patcher (e.g. boxes, UI objects, etc.), or the constant `CLASS_NOBOX`, for classes which will only be used internally. Developers can define their own name spaces as well, but this functionality is currently undocumented.
`classname`	The name of the class to be looked up

If successful, this routine returns the class's data pointer. Otherwise, it returns NULL.

calcoffset

The macro `calcoffset` is used as follows:

```
long calcoffset(void classtype, void varname);
```

`classtype`	Your class's defined type (e.g. `t_myobject`)

`varname`	The name of a variable in your class's data structure.

The macro returns the byte-offset of the specified variable within the class data structure for the specified class type.

Object Routines: instantiation and freeing

object_alloc

Use the `object_alloc` function to allocate the memory for an instance of an object class and initialize its object header. It is used like the traditional routine `newobject`, inside of an object's `new` method, but its use is required with obex-class objects.

```
void *object_alloc(t_class *c);
```

`c`	Your class pointer, returned by `class_new`

This routine returns a new instance of your object class if successful, or NULL if unsuccessful.

object_new

Use the `object_new` function to allocate the memory for an instance of an object class and initialize its object header *internal to Max*. It is used similarly to the traditional routine `newinstance`, but its use is required with obex-class objects.

```
void *object_new(t_symbol *name_space,
      t_symbol *classname, ...);
```

`name_space`	The desired object's name space. Typically, either the constant `CLASS_BOX`, for obex classes which can instantiate inside of a Max patcher (e.g. boxes, UI objects, etc.), or the constant `CLASS_NOBOX`, for classes which will only be used internally. Developers can define their own name spaces as well, but this functionality is currently undocumented.
`classname`	The name of the class of the object to be created

`...`	Any arguments expected by the object class being instantiated

This routine returns a new instance of the object class if successful, or NULL if unsuccessful.

object_new_typed

Use the `object_new_typed` function to allocate the memory for an instance of an object class and initialize its object header *internal to Max*. It is used similarly to the traditional routine `newinstance`, but its use is required with obex-class objects. The `object_new_typed` routine differs from `object_new` by its use of an atom list for object arguments—in this way, it more resembles the effect of typing something into an object box from the Max interface.

```
void *object_new_typed(t_symbol *name_space,
     t_symbol *classname, long ac, t_atom *av);
```

`name_space`	The desired object's name space. Typically, either the constant `CLASS_BOX`, for obex classes which can instantiate inside of a Max patcher (e.g. boxes, UI objects, etc.), or the constant `CLASS_NOBOX`, for classes which will only be used internally. Developers can define their own name spaces as well, but this functionality is currently undocumented.
`classname`	The name of the class of the object to be created
`ac`	Count of arguments in `av`
`av`	Array of t_atoms; arguments to the class's instance creation function.

This routine returns a new instance of the object class if successful, or NULL if unsuccessful.

object_free

Use the `object_free` function to call the free function and release the memory for an instance of an internal object class previously instantiated using `object_new`, `object_new_typed` or other new-style object constructor functions (e.g. `hashtab_new`). It is, at the time of this writing, a wrapper for the traditional routine `freeobject`, but its use is suggested with obex-class objects.

```
t_max_err object_free(void *x);
```

`x`	The pointer to the object to be freed.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Object Routines: sending messages

object_method

Use the `object_method` function to send an untyped message to an object.

```
void *object_method(void *x, t_symbol *s, ...);
```

`x`	The object that will receive the message
`s`	The message selector
`...`	Any arguments to the message

If the receiver object can respond to the message, `object_method` returns the result. Otherwise, the function will return 0.

Example: To send the message `bang` to the object `bang_me`:

```
void *bang_result;

bang_result = object_method(bang_me, gensym("bang"));
```

object_method_typed

Use the `object_method_typed` function to send a type-checked message to an object.

```
t_max_err object_method_typed(void *x, t_symbol *s,
      long ac, t_atom *av, t_atom *rv);
```

`x`	The object that will receive the message
`s`	The message selector
`ac`	Count of message arguments in `av`
`av`	Array of t_atoms; the message arguments
`rv`	Return value of function, if available

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

If the receiver object can respond to the message, `object_method_typed` returns the result in `rv`. Otherwise, `rv` will contain an `A_NOTHING` atom.

object_getmethod

Use the `object_getmethod` function to retrieve an object's `method` for a particular message selector.

```
method object_getmethod(void *x, t_symbol *s);
```

`x`	The object whose method is being queried
`s`	The message selector

This routine returns the `method` if successful, or 0 if unsuccessful.

Object Routines: obex

object_obex_store

Use the `object_obex_store` function to store data in your object's obex.

```
t_max_err object_obex_store(void *x, t_symbol *key,
     t_object *val);
```

`x`	Your object pointer. This routine should only be called on instantiated objects (i.e. in the `new` method or later), not directly on classes (i.e. in `main()`).
`key`	A symbolic name for the data to be stored
`val`	A `t_object *`, to be stored in the obex, referenced under the `key`.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Most developers will need to use this routine for the specific purpose of storing the dumpout outlet in the obex (the dumpout outlet is used by attributes to report data in response to 'get' queries). For this, the developer should use something like the following in the object's `new` method:

```
object_obex_store(x, _sym_dumpout,
     outlet_new(x, NULL));
```

object_obex_lookup

Use the `object_obex_lookup` function to learn the value of a data stored in the obex.

```
t_max_err object_obex_lookup(void *x, t_symbol *key,
      t_object **val);
```

`x`	Your object pointer. This routine should only be called on instantiated objects (i.e. in the `new` method or later), not directly on classes (i.e. in `main()`).
`key`	The symbolic name for the data to be retrieved
`val`	A pointer to a `t_object *`, to be filled with the data retrieved from the obex.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

By default, pointers to your object's containing t_patcher and t_box objects are stored in the obex, under the keys '#P' and '#B', respectively. To retrieve them, the developer could do something like the following:

```
void post_containers(t_obexobj *x)
{
      t_patcher *p;
      t_box *b;

      p = object_obex_lookup(x, gensym("#P"),
            (t_object **)&p);
      b = object_obex_lookup(x, gensym("#B"),
            (t_object **)&b);

      post("my patcher is located at 0x%X", p);
      post("my box is located at 0x%X", b);
}
```

object_obex_dumpout

Use the `object_obex_dumpout` function to send data from your object's dumpout outlet. The dumpout outlet is stored in the obex using the `object_obex_store` routine (see above) It is used approximately like `outlet_anything`.

```
void object_obex_dumpout(void *x, t_symbol *s,
      long argc, t_atom *argv);
```

`x`	Your object pointer. This routine should only be called on instantiated objects (i.e. in the `new` method or later), not directly on classes (i.e. in `main()`).
`s`	The message selector t_symbol *
`argc`	Number of elements in the argument list in argv
`argv`	t_atoms constituting the message arguments

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_obex_set

The `object_obex_set` function is currently undocumented.

```
t_max_err object_obex_set(void *x, t_hashtab *obex);
```

`x`	*undocumented*
`obex`	*undocumented*

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_obex_get

The `object_obex_get` function is currently undocumented.

```
t_hashtab *object_obex_get(void *x);
```

`x`	*undocumented*

The return value is undocumented.

Object Routines: miscellaneous

object_classname_compare

Use the `object_classname_compare` function to determine if a particular object is an instance of a given class.

```
long object_classname_compare(void *x,
      t_symbol *name);
```

`x`	The object to test
`name`	The name of the class to test the object against

This routine returns 1 if the object is an instance of the named class. Otherwise, 0 is returned.

For instance, to determine whether an unknown object pointer is a pointer to a print object, you'd call:

```
long isprint = object_classname_compare(x,
      gensym("print"));
```

object_class

Use the `object_class` function to determine the class of a given object.

```
t_class *object_class(void *x);
```

`x`	The object to test

This routine returns the t_class * of the object's class, if successful, or `NULL`, if unsuccessful.

object_getvalueof

Use the `object_getvalueof` function to retrieve the value of an object which supports the `getvalueof/setvalueof` interface. See part 2 of the pattr SDK for more information on this interface.

```
t_max_err object_getvalueof(void *x, long *ac,
      t_atom **av);
```

x	The object whose value is of interest
ac	Pointer to a long variable to receive the count of arguments in av. The long variable itself should be set to 0 previous to calling this routine.
av	Pointer to a t_atom *, to receive object data. The t_atom * itself should be set to NULL previous to calling this routine.

This routine returns the error code MAX_ERR_NONE if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Calling the object_getvalueof routine allocates memory for any data it returns. It is the developer's responsibility to free it, using the freebytes function.

Developers wishing to design objects which will support this function being called on them must define and implement a special method, getvalueof, like so:

```
class_addmethod(c, (method)myobject_getvalueof,
      "getvalueof", A_CANT, 0);
```

The getvalueof method should be prototyped as:

```
t_max_err myobject_getvalueof(t_myobject *x, long *ac,
      t_atom **av);
```

And implemented, generally, as:

```
t_max_err myobj_getvalueof(t_myobj *x, long *ac,
   t_atom **av)
{
   if (ac && av) {
      if (*ac && *av) {
         // memory has been passed in; use it.
      } else {
         // allocate enough memory for your data
         *av = (t_atom *)getbytes(sizeof(t_atom));
      }
      *ac = 1; // our data is a single floating point
         value
      atom_setfloat(*av, x->objvalue);
   }
   return MAX_ERR_NONE;
}
```

By convention, and to permit the interoperability of objects using the obex API, developers should allocate memory in their `getvalueof` methods using the `getbytes` function.

object_setvalueof

Use the `object_setvalueof` function to set the value of an object which supports the `getvalueof/setvalueof` interface. See part 2 of the pattr SDK for more information on this interface.

```
t_max_err object_setvalueof(void *x, long ac,
      t_atom *av);
```

`x`	The object whose value is of interest
`ac`	The count of arguments in `av`
`av`	Array of t_atoms; the new desired data for the object

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Developers wishing to design objects which will support this function being called on them must define and implement a special method, `setvalueof`, like so:

```
class_addmethod(c, (method)myobject_setvalueof,
      "setvalueof", A_CANT, 0);
```

The `setvalueof` method should be prototyped as:

```
t_max_err myobject_setvalueof(t_myobject *x, long *ac,
      t_atom **av);
```

And implemented, generally, as:

```
t_max_err myobject_setvalueof(t_myobject *x, long ac,
      t_atom *av)
{
      if (ac && av) {
            // simulate receipt of a float value
            myobject_float(x, atom_getfloat(av));
```

```
        }
        return MAX_ERR_NONE;
}
```

Object Routines: registration/notification

Starting in Max 4.5, objects can be registered under a symbolic name inside a developer-defined *namespace*. The namespace is essentially a hash table which provides fast lookup for member data. The principal usage for namespaces, from the perspective of developers, is client registration and notification.

In order to create a client notification scheme using this system, the following steps should be followed:

1. The *server* object needs to be registered within a namespace, using the `object_register` function.

2. *Client* objects need to be *attached* to the server object, using the `object_attach` function.

3. Client objects require a special class method, `notify`. The prototype for this method can be found below, under the reference for the `object_notify` routine.

4. In order to broadcast a message, the server object need simply call the `object_notify` function. All client objects, via their `notify` method, will receive the message.

5. In order to detach clients from a server, the `object_detach` routine is called.

6. Finally, to remove a registered object from its namespace, the `object_unregister` routine is called.

Following is a reference for the necessary routines. The included source code example, *myregob*, demonstrates the principles in use.

object_register

Use the `object_register` function to register an object in a namespace.

```
void *object_register(t_symbol *name_space,
        t_symbol *s, void *x);
```

`name_space`	The namespace in which to register the object. The namespace can be any symbol. If the namespace does not already exist, it is created automatically.
`s`	The name of the object in the namespace. This name will be used by other objects to attach and detach from the registered object.
`x`	The object to register

The function returns a pointer to the registered object. Under some circumstances, object_register will *duplicate* your object, and return a pointer to the duplicate—the developer should not assume that the pointer passed in is the same pointer that has been registered. To be safe, the returned pointer should be stored and used with the `object_unregister()` routine.

object_unregister

Use the `object_unregister` function to remove a registered object from a namespace.

```
t_max_err object_unregister(void *x);
```

`x`	The object to unregister. This should be the pointer returned from the `object_register` function.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attach

Use the `object_attach` function to attach a client to a registered object. Once attached, the object will receive notifications sent from the registered object (via the `object_notify` function), if it has a `notify` method defined and implemented. See below for more information, in the reference for `object_notify`.

```
t_max_err object_attach(t_symbol *name_space,
     t_symbol *s, void *x);
```

`name_space`	The namespace of the registered object. This should be the same value used in `object_register` to register the object. If you don't know the registered object's namespace, the `object_findregisteredbyptr` routine can be used to determine it.
`s`	The name of the registered object in the namespace. If you don't know the name of the registered object, the `object_findregisteredbyptr` routine can be used to determine it.
`x`	The object to attach. Generally, this is the pointer to your Max object.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_detach

Use the `object_detach` function to detach a client from a registered object.

```
t_max_err object_detach(t_symbol *name_space,
      t_symbol *s, void *x);
```

`name_space`	The namespace of the registered object. This should be the same value used in `object_register` to register the object. If you don't know the registered object's namespace, the `object_findregisteredbyptr` routine can be used to determine it.
`s`	The name of the registered object in the namespace. If you don't know the name of the registered object, the `object_findregisteredbyptr` routine can be used to determine it.
`x`	The object to detach. Generally, this is the pointer to your Max object.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_notify

Use the `object_notify` function to broadcast a message (with an optional argument) from a registered object to any attached client objects.

```
t_max_err object_notify(void *x, t_symbol *s,
     void *data);
```

`x`	Pointer to the registered object
`s`	The message to send
`data`	An optional argument which will be passed with the message. Set this argument to `NULL` if it will be unused.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

In order for client objects to receive notifications, they must define and implement a special method, `notify`, like so:

```
class_addmethod(c, (method)myobject_notify, "notify",
     A_CANT, 0);
```

The `notify` method should be prototyped as:

```
void myobject_notify(t_myobject *x, t_symbol *s,
     t_symbol *msg, void *sender, void *data);
```

`x`	Pointer to the receiving object
`s`	The name of the sending (registered) object in its namespace
`msg`	The sent message
`sender`	Pointer to the sending object.
`data`	An optional argument sent with the message. This value corresponds to the data argument in the `object_notify` method.

object_findregistered

Use the `object_findregistered` function to determine a registered object's pointer, given its namespace and name.

```
void *object_findregistered(t_symbol *name_space,
     t_symbol *s);
```

`name_space`	The namespace of the registered object
`s`	The name of the registered object in the namespace

This routine returns the pointer of the registered object, if successful, or NULL, if unsuccessful.

object_findregisteredbyptr

Use the `object_findregisteredbyptr` function to determine the namespace and/or name of a registered object, given the object's pointer.

```
t_max_err object_findregisteredbyptr(
     t_symbol **name_space, t_symbol **s, void *x);
```

`name_space`	Pointer to a t_symbol *, to receive the namespace of the registered object
`s`	Pointer to a t_symbol *, to receive the name of the registered object within the namespace
`x`	Pointer to the registered object

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Attribute Routines: attribute creation and filters

Attributes have been described at length elsewhere. They can be added to a class using the `class_addattr` function. The following functions provide means of creating and modifying attribute filter properties.

attribute_new

Use the `attribute_new` function to create a new attribute. The attribute will allocate memory and store its own data. Attributes created using `attribute_new` can be assigned either to classes (using the `class_addattr` function) or to objects (using the `object_addattr` function).

```
t_object *attribute_new(char *name, t_symbol *type,
     long flags, method mget, method mset);
```

`name`	A name for the attribute, as a C-string
`type`	A t_symbol * representing a valid attribute type. At the time of this writing, the valid type-symbols are: `_sym_char` (char), `_sym_long` (long), `_sym_float32` (32-bit float), `_sym_float64` (64-bit float), `_sym_atom` (Max t_atom pointer), `_sym_symbol` (Max t_symbol pointer), `_sym_pointer` (generic pointer) and `_sym_object` (Max t_object pointer).

`flags`	Any attribute flags, expressed as a bitfield. Attribute flags are used to determine if an attribute is accessible for setting or querying. The following accessor flags are currently available: ATTR_GET_OPAQUE: The attribute cannot be queried by either max message when used inside of a CLASS_BOX object, nor from C code. ATTR_SET_OPAQUE: The attribute cannot be set by either max message when used inside of a CLASS_BOX object, nor from C code. ATTR_GET_OPAQUE_USER: The attribute cannot be queried by max message when used inside of a CLASS_BOX object, but *can* be queried from C code. ATTR_SET_OPAQUE_USER: The attribute cannot be set by max message when used inside of a CLASS_BOX object, but *can* be set from C code.
`mget`	The method to use for the attribute's `get` functionality. If `mget` is `NULL`, the default method is used.
`mset`	The method to use for the attribute's `set` functionality. If `mset` is `NULL`, the default method is used.

This routine returns the new attribute's object pointer if successful, or `NULL` if unsuccessful.

Developers wishing to define custom methods for `get` or `set` functionality need to prototype them as:

```
t_max_err myobject_myattr_get(t_myobject *x,
     void *attr, long *ac, t_atom **av);

t_max_err myobject_myattr_set(t_myobject *x,
     void *attr, long ac, t_atom *av);
```

Implementation will vary, of course, but need to follow the following basic models. Note that, as with custom `getvalueof` and `setvalueof` methods for your object, assumptions are made throughout Max that getbytes has been used for memory allocation. Developers are strongly urged to do the same:

```
t_max_err myobject_myattr_get(t_myobject *x,
      void *attr, long *ac, t_atom **av)
{
      if (*ac && *av)
            // memory passed in; use it
      else {
            *ac = 1; // size of attr data
            *av = (t_atom *)getbytes(sizeof(t_atom) *
                  (*ac));
            if (!(*av)) {
                  *ac = 0;
                  return MAX_ERR_OUT_OF_MEM;
            }
      }
      atom_setlong(*av, x->some_value);
      return MAX_ERR_NONE;
}

t_max_err myobject_myattr_set(t_myobject *x,
      void *attr, long ac, t_atom *av)
{
      if (ac && av) {
            x->some_value = atom_getlong(av);
      }
      return MAX_ERR_NONE;
}
```

attr_offset_new

Use the attr_offset_new function to create a new attribute. The attribute references memory stored outside of itself, in your object's data structure. Attributes created using attr_offset_new can be assigned either to classes (using the class_addattr function) or to objects (using the object_addattr function).

```
t_object *attr_offset_new(char *name, t_symbol *type,
      long flags, method mget, method mset,
      long offset);
```

name	A name for the attribute, as a C-string
type	A t_symbol * representing a valid attribute type. At the time of this writing, the valid type-symbols are: _sym_char (char), _sym_long (long),

	`_sym_float32` (32-bit float), `_sym_float64` (64-bit float), `_sym_atom` (Max t_atom pointer), `_sym_symbol` (Max t_symbol pointer), `_sym_pointer` (generic pointer) and `_sym_object` (Max t_object pointer). This type should match the size of the data at the byte offset specified in the `offset` argument.
`flags`	Any attribute flags, expressed as a bitfield. See the discussion of attribute flags above, under `attribute_new().`
`mget`	The method to use for the attribute's `get` functionality. If `mget` is `NULL`, the default method is used. See the discussion under `attribute_new`, above, for more information.
`mset`	The method to use for the attribute's `set` functionality. If `mset` is `NULL`, the default method is used. See the discussion under `attribute_new`, above, for more information.
`offset`	Byte offset into the class data structure of the object which will "own" the attribute. The offset should point to the data to be referenced by the attribute. Typically, the `calcoffset` macro (described above) is used to calculate this offset.

This routine returns the new attribute's object pointer if successful, or `NULL` if unsuccessful.

For instance, to create a new attribute which references the value of a double variable (`val`) in an object class's data structure:

```
t_object *attr = attr_offset_new("myattr",
      _sym_float64 /* matches data size */,
      0 /* no flags */, (method)0L, (method)0L,
      calcoffset(t_myobject, val));
```

attr_offset_array_new

Use the `attr_offset_array_new` function to create a new attribute. The attribute references an array of memory stored outside of itself, in your object's data structure. Attributes created using `attr_offset_array_new` can be assigned either to classes (using the `class_addattr` function) or to objects (using the `object_addattr` function).

```
t_object *attr_offset_array_new(char *name,
     t_symbol *type, long size, long flags,
     method mget, method mset, long offsetcount,
     long offset);
```

`name`	A name for the attribute, as a C-string
`type`	A t_symbol * representing a valid attribute type. At the time of this writing, the valid type-symbols are: `_sym_char` (char), `_sym_long` (long), `_sym_float32` (32-bit float), `_sym_float64` (64-bit float), `_sym_atom` (Max t_atom pointer), `_sym_symbol` (Max t_symbol pointer), `_sym_pointer` (generic pointer) and `_sym_object` (Max t_object pointer). This type should match the size of the data at the byte offset specified in the `offset` argument.
`size`	The maximum number of elements the array of data will contain.
`flags`	Any attribute flags, expressed as a bitfield. See the discussion of attribute flags above, under `attribute_new()`.
`mget`	The method to use for the attribute's `get` functionality. If `mget` is `NULL`, the default method is used. See the discussion under `attribute_new`, above, for more information.
`mset`	The method to use for the attribute's `set` functionality. If `mset` is `NULL`, the default method is used. See the

discussion under `attribute_new`, above, for more information.

`offsetcount`	Byte offset into your object class's data structure of a long variable describing how many array elements (up to `size`) comprise the data to be referenced by the attribute. Typically, the `calcoffset` macro (described above) is used to calculate this offset.
`offset`	Byte offset into the class data structure of the object which will "own" the attribute. The offset should point to the data to be referenced by the attribute. Typically, the `calcoffset` macro (described above) is used to calculate this offset.

This routine returns the new attribute's object pointer if successful, or `NULL` if unsuccessful.

For instance, to create a new attribute which references an array of 10 t_atoms (`atm`; the current number of "active" elements in the array is held in the variable `atmcount`) in an object class's data structure:

```
t_object *attr = attr_offset_array_new("myattrarray",
     _sym_atom /* matches data size */, 10 /* max */,
     0 /* no flags */, (method)0L, (method)0L,
     calcoffset(t_myobject, atmcount) /* count */,
     calcoffset(t_myobject, atm) /* data */);
```

attr_addfilter_clip

Use the `attr_addfilter_clip` function to attach a clip filter to an attribute. The filter will clip any values sent to or retrieved from the attribute using the attribute's `get` and `set` functions.

```
t_max_err attr_addfilter_clip(void *x, double min,
     double max, long usemin, long usemax);
```

`x`	Pointer to the attribute to receive the filter
`min`	Minimum value for the clip filter
`max`	Maximum value for the clip filter

`usemin`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.
`usemax`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

attr_addfilter_clip_scale

Use the `attr_addfilter_clip_scale` function to attach a clip/scale filter to an attribute. The filter will clip and scale any values sent to or retrieved from the attribute using the attribute's `get` and `set` functions.

```
t_max_err attr_addfilter_clip_scale(void *x,
     double scale, double min, double max,
     long usemin, long usemax);
```

`x`	Pointer to the attribute to receive the filter
`scale`	Scale value. Data sent to the attribute will be scaled by this amount. Data retrieved from the attribute will be scaled by its reciprocal. *Scaling occurs previous to clipping.*
`min`	Minimum value for the clip filter
`max`	Maximum value for the clip filter
`usemin`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.
`usemax`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

attr_addfilterset_clip

Use the `attr_addfilterset_clip` function to attach a clip filter to an attribute. The filter will *only* clip values sent to the attribute using the attribute's `set` function.

```
t_max_err attr_addfilterset_clip(void *x, double min,
     double max, long usemin, long usemax);
```

`x`	Pointer to the attribute to receive the filter
`min`	Minimum value for the clip filter
`max`	Maximum value for the clip filter
`usemin`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.
`usemax`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

attr_addfilterset_clip_scale

Use the `attr_addfilterset_clip_scale` function to attach a clip/scale filter to an attribute. The filter will *only* clip and scale values sent to the attribute using the attribute's `set` function.

```
t_max_err attr_addfilterset_clip_scale(void *x,
     double scale, double min, double max,
     long usemin, long usemax);
```

`x`	Pointer to the attribute to receive the filter
`scale`	Scale value. Data sent to the attribute will be scaled by this amount. *Scaling occurs previous to clipping.*
`min`	Minimum value for the clip filter

`max`	Maximum value for the clip filter
`usemin`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.
`usemax`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

attr_addfilterget_clip

Use the `attr_addfilterget_clip` function to attach a clip filter to an attribute. The filter will *only* clip values retrieved from the attribute using the attribute's `get` function.

```
t_max_err attr_addfilterget_clip(void *x, double min,
        double max, long usemin, long usemax);
```

`x`	Pointer to the attribute to receive the filter
`min`	Minimum value for the clip filter
`max`	Maximum value for the clip filter
`usemin`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.
`usemax`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

attr_addfilterget_clip_scale

Use the `attr_addfilterget_clip_scale` function to attach a clip/scale filter to an attribute. The filter will *only* clip and scale values retrieved from the attribute using the attribute's `get` function.

```
t_max_err attr_addfilterget_clip_scale(void *x,
     double scale, double min, double max,
     long usemin, long usemax);
```

`x`	Pointer to the attribute to receive the filter
`scale`	Scale value. Data retrieved from the attribute will be scaled by this amount. *Scaling occurs previous to clipping.*
`min`	Minimum value for the clip filter
`max`	Maximum value for the clip filter
`usemin`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.
`usemax`	Set this value to 0 if the minimum clip value should *not* be used. Otherwise, set the value to non-zero.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

attr_addfilterset_proc

Use the `attr_addfilterset_proc` function to attach a custom filter method to your attribute. The filter will *only* be called for values retrieved from the attribute using the attribute's `set` function.

```
t_max_err attr_addfilterset_proc(void *x,
     method proc);
```

`x`	Pointer to the attribute to receive the filter

`proc`	A filter method

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

The filter method should be prototyped and implemented as follows:

```
t_max_err myfiltermethod(void *parent, void *attr,
      long ac, t_atom *av);

t_max_err myfiltermethod(void *parent, void *attr,
      long ac, t_atom *av)
{
      long i;
      float temp,

      // this filter rounds off all values
      // assumes that the data is float
      for (i = 0; i < ac; i++) {
            temp = atom_getfloat(av + i);
            temp = (float)((long)(temp + 0.5));
            atom_setfloat(av + i, temp);
      }
      return MAX_ERR_NONE;
}
```

attr_addfilterget_proc

Use the `attr_addfilterget_proc` function to attach a custom filter method to your attribute. The filter will *only* be called for values retrieved from the attribute using the attribute's `get` function.

```
t_max_err attr_addfilterget_proc(void *x,
      method proc);
```

`x`	Pointer to the attribute to receive the filter
`proc`	A filter method

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

The filter method should be prototyped and implemented as described above for the `attr_addfilterset_proc` function.

Object Routines: attributes (general)

object_attr_getvalueof

Use the `object_attr_getvalueof` function to retrieve the value of an object's attribute.

```
t_max_err object_attr_getvalueof(void *x, t_symbol *s,
     long *argc, t_atom **argv);
```

`x`	Pointer to the object whose attribute is of interest
`s`	The attribute's name
`argc`	Pointer to a long variable to receive the count of arguments in `argv`. The long variable itself should be set to 0 previous to calling this routine.
`argv`	Pointer to a t_atom *, to receive object data. The t_atom * itself should be set to `NULL` previous to calling this routine.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Calling the `object_getvalueof` routine allocates memory for any data it returns. It is the developer's responsibility to free it, using the `freebytes` function.

object_attr_setvalueof

Use the `object_attr_setvalueof` function to set the value of an object's attribute.

```
t_max_err object_attr_setvalueof(void *x, t_symbol *s,
     long argc, t_atom *argv);
```

`x`	Pointer to the object whose attribute is of interest
`s`	The attribute's name

`argc`	The count of arguments in `argv`
`argv`	Array of t_atoms; the new desired data for the attribute

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_get

Use the `object_attr_get` function to return the pointer to an attribute, given its name.

```
void *object_attr_get(void *x, t_symbol *attrname);
```

`x`	Pointer to the object whose attribute is of interest
`attrname`	The attribute's name

This routine returns a pointer to the attribute, if successful, or `NULL`, if unsuccessful.

object_attr_method

Use the `object_attr_method` function to return the method of an attribute's `get` or `set` function, as well as a pointer to the attribute itself, from a message name.

```
method object_attr_method(void *x,
      t_symbol *methodname, void **attr, long *get);
```

`x`	Pointer to the object whose attribute is of interest
`methodname`	The Max message used to call the attribute's `get` or `set` function. For example, `gensym("mode")` or `gensym("getthresh")`.
`attr`	A pointer to a void *, which will be set to the attribute pointer upon successful completion of the function

`get`	A pointer to a long variable, which will be set to 1 upon successful completion of the function, if the queried method corresponds to the `get` function of the attribute.

This routine returns the requested method, if successful, or `NULL`, if unsuccessful.

object_attr_usercanset

Use the `object_attr_usercanset` function to determine if an object's attribute can be set from the Max interface (i.e. if its `ATTR_SET_OPAQUE_USER` flag is set).

```
long object_attr_usercanset(void *x, t_symbol *s);
```

`x`	Pointer to the object whose attribute is of interest
`s`	The attribute's name

This routine returns 1 if the attribute can be set from the Max interface. Otherwise, it returns 0.

object_attr_usercanget

Use the `object_attr_usercanget` function to determine if the value of an object's attribute can be queried from the Max interface (i.e. if its `ATTR_GET_OPAQUE_USER` flag is set).

```
long object_attr_usercanget(void *x, t_symbol *s);
```

`x`	Pointer to the object whose attribute is of interest
`s`	The attribute's name

This routine returns 1 if the value of the attribute can be queried from the Max interface. Otherwise, it returns 0.

object_attr_getdump

Use the `object_attr_getdump` function to force a specified object's attribute to send it's value from the object's dumpout outlet in the Max interface.

```
void object_attr_getdump(void *x, t_symbol *s,
      long argc, t_atom *argv);
```

`x`	Pointer to the object whose attribute is of interest
`s`	The attribute's name
`argc`	Unused
`argv`	Unused

Object Routines: attributes (object attributes)

Typically, attributes, like methods, are attached to an object's class, and available to any instance of the class. However, it is also possible for a particular instantiation of an object to possess its own attribute, not shared by all members of the class. The following functions implement this functionality.

object_addattr

Use the `object_addattr` function to attach an attribute directly to an object.

```
t_max_err object_addattr(void *x, t_object *attr);
```

`x`	An object to which the attribute should be attached
`attr`	The attribute's pointer—this should be a pointer returned from `attribute_new`, `attr_offset_new` or `attr_offset_array_new`.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_deleteattr

Use the `object_deleteattr` function to detach an attribute from an object that was previously attached with `object_addattr`. The function will also free all memory associated with the attribute. If you only wish to detach the attribute, without freeing it, see the `object_chuckattr` function, below.

```
t_max_err object_deleteattr(void *x,
     t_symbol *attrsym);
```

`x`	The object to which the attribute is attached
`attrsym`	The attribute's name

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_chuckattr

Use the `object_deleteattr` function to detach an attribute from an object that was previously attached with `object_addattr`. This function will not free the attribute (use `object_free` to do this manually).

```
t_max_err object_chuckattr(void *x,
     t_symbol *attrsym);
```

`x`	The object to which the attribute is attached
`attrsym`	The attribute's name

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Attribute Routines: attribute arguments

Most objects which implement attributes support the use of attributes on a Max object's "command line"—written directly in the object box using the form `@attributename value`. The following routines implement this functionality.

NOTE: Because attribute arguments rely on atom lists of the `argc`/`argv` variety, developers who wish to support them must implement their objects as `A_GIMME` in their call to `class_new`.

attr_args_process

Use the `attr_args_process` function to take an atom list and properly set any attributes described within. This function is typically used in an object's `new` method to conveniently process attribute arguments.

```
void attr_args_process(void *x, short ac, t_atom *av);
```

`x`	The object whose attributes will be processed
`ac`	The count of t_atoms in `av`
`av`	An atom list

Here is a typical example of usage:

```
void *myobject_new(t_symbol *s, long ac, t_atom *av)
{
      t_myobject *x = NULL;

      if (x=(t_myobject *)object_alloc(myobject_class))
      {
            // initialize any data before processing
            // attributes to avoid overwriting
            // attribute argument-set values
            x->data = 0;

            // process attr args, if any
            attr_args_process(x, ac, av);
      }
      return x;
}
```

attr_args_offset

Use the `attr_args_offset` function to determine the point in an atom list where attribute arguments begin. Developers can use this routine to assist in the manual processing of attribute arguments, when attr_args_process doesn't provide the correct functionality for a particular purpose.

```
long attr_args_offset(short ac, t_atom *av);
```

`ac`	The count of t_atoms in `av`
`av`	An atom list

This routine returns an offset into the atom list, where the first attribute argument occurs. For instance, the atom list `foo bar 3.0 @mode 6` would cause `attr_args_offset` to return 3 (the attribute `mode` appears at position 3 in the atom list).

Object Routines: attribute utility functions (single value functions)

The following routines provide simple get and set access to long, float and t_symbol * attribute values. They can be used with any attribute, but will only return the first element of an attribute's data array in the instance of attributes which contain more than 1 value.

object_attr_getlong

Use the `object_attr_getlong` function to retrieve the value of an attribute, given its parent object and name.

```
long object_attr_getlong(void *x, t_symbol *s);
```

`x`	The attribute's parent object
`s`	The attribute's name

This routine returns the value of the specified attribute, if successful, or 0, if unsuccessful.

If the attribute is not of the type specified by the function, the routine will attempt to coerce a valid value from the attribute.

object_attr_getfloat

Use the `object_attr_getfloat` function to retrieve the value of an attribute, given its parent object and name.

```
float object_attr_getfloat(void *x, t_symbol *s);
```

`x`	The attribute's parent object
`s`	The attribute's name

This routine returns the value of the specified attribute, if successful, or 0, if unsuccessful.

If the attribute is not of the type specified by the function, the routine will attempt to coerce a valid value from the attribute.

object_attr_getsym

Use the `object_attr_getsym` function to retrieve the value of an attribute, given its parent object and name.

```
t_symbol *object_attr_getsym(void *x, t_symbol *s);
```

`x`	The attribute's parent object
`s`	The attribute's name

This routine returns the value of the specified attribute, if successful, or the empty symbol (equivalent to `gensym("")` or `_sym_nothing`), if unsuccessful.

object_attr_setlong

Use the `object_attr_setlong` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setlong(void *x, t_symbol *s,
     long c);
```

`x`	The attribute's parent object
`s`	The attribute's name
`c`	An integer value; the new value for the attribute

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_setfloat

Use the `object_attr_setfloat` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setfloat(void *x, t_symbol *s,
      float c);
```

`x`	The attribute's parent object
`s`	The attribute's name
`c`	An floating point value; the new value for the attribute

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_setsym

Use the `object_attr_setsym` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setsym(void *x, t_symbol *s,
      t_symbol *c);
```

`x`	The attribute's parent object
`s`	The attribute's name

`c`	A t_symbol *; the new value for the attribute

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Object Routines: attribute utility functions (multiple value functions)

The following routines provide access to arrays of attribute values. They can be used with any attribute.

object_attr_getlong_array

Use the `object_attr_getlong_array` function to retrieve the value of an attribute, given its parent object and name. This routine uses a developer-allocated array to copy data to. Developers wishing to retrieve the value of an attribute without pre-allocating memory should refer to the `object_attr_getvalueof` function.

```
long object_attr_getlong_array(void *x, t_symbol *s,
      long max, long *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`max`	The number of array elements in `vals`. The routine will take care not to overwrite the bounds of the array.
`vals`	Pointer to the first element of a pre-allocated array of long data.

This routine returns the number of elements copied into `vals`.

If the attribute is not of the type specified by the function, the routine will attempt to coerce a valid value from the attribute.

object_attr_getchar_array

Use the `object_attr_getchar_array` function to retrieve the value of an attribute, given its parent object and name. This routine uses a developer-allocated array to copy data to. Developers wishing to retrieve the value of an attribute without pre-allocating memory should refer to the `object_attr_getvalueof` function.

```
long object_attr_getchar_array(void *x, t_symbol *s,
      long max, uchar *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`max`	The number of array elements in `vals`. The routine will take care not to overwrite the bounds of the array.
`vals`	Pointer to the first element of a pre-allocated array of unsigned char data.

This routine returns the number of elements copied into `vals`.

If the attribute is not of the type specified by the function, the routine will attempt to coerce a valid value from the attribute.

object_attr_getfloat_array

Use the `object_attr_getfloat_array` function to retrieve the value of an attribute, given its parent object and name. This routine uses a developer-allocated array to copy data to. Developers wishing to retrieve the value of an attribute without pre-allocating memory should refer to the `object_attr_getvalueof` function.

```
long object_attr_getfloat_array(void *x, t_symbol *s,
      long max, float *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`max`	The number of array elements in `vals`. The routine will take care not to overwrite the bounds of the array.

`vals`	Pointer to the first element of a pre-allocated array of float data.

This routine returns the number of elements copied into `vals`.

If the attribute is not of the type specified by the function, the routine will attempt to coerce a valid value from the attribute.

object_attr_getdouble_array

Use the `object_attr_getdouble_array` function to retrieve the value of an attribute, given its parent object and name. This routine uses a developer-allocated array to copy data to. Developers wishing to retrieve the value of an attribute without pre-allocating memory should refer to the `object_attr_getvalueof` function.

```
long object_attr_getdouble_array(void *x, t_symbol *s,
     long max, double *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`max`	The number of array elements in `vals`. The routine will take care not to overwrite the bounds of the array.
`vals`	Pointer to the first element of a pre-allocated array of double data.

This routine returns the number of elements copied into `vals`.

If the attribute is not of the type specified by the function, the routine will attempt to coerce a valid value from the attribute.

object_attr_getsym_array

Use the `object_attr_getsym_array` function to retrieve the value of an attribute, given its parent object and name. This routine uses a developer-allocated array to copy data to. Developers wishing to retrieve the value of an attribute without pre-allocating memory should refer to the `object_attr_getvalueof` function.

```
long object_attr_getsym_array(void *x, t_symbol *s,
      long max, t_symbol **vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`max`	The number of array elements in `vals`. The routine will take care not to overwrite the bounds of the array.
`vals`	Pointer to the first element of a pre-allocated array of t_symbol *s.

This routine returns the number of elements copied into `vals`.

object_attr_setlong_array

Use the `object_attr_setlong_array` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setlong_array(void *x,
      t_symbol *s, long count, long *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`count`	The number of array elements in vals
`vals`	Pointer to the first element of an array of long data

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_setchar_array

Use the `object_attr_setchar_array` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setchar_array(void *x,
      t_symbol *s, long count, uchar *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`count`	The number of array elements in vals
`vals`	Pointer to the first element of an array of unsigned char data

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_setfloat_array

Use the `object_attr_setfloat_array` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setfloat_array(void *x,
      t_symbol *s, long count, float *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`count`	The number of array elements in vals
`vals`	Pointer to the first element of an array of float data

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_setdouble_array

Use the `object_attr_setdouble_array` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setlong_array(void *x,
      t_symbol *s, long count, double *vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`count`	The number of array elements in vals
`vals`	Pointer to the first element of an array of double data

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_attr_setsym_array

Use the `object_attr_setsym_array` function to set the value of an attribute, given its parent object and name. The routine will call the attribute's `set` method, using the data provided.

```
t_max_err object_attr_setlong_array(void *x,
      t_symbol *s, long count, t_symbol **vals);
```

`x`	The attribute's parent object
`s`	The attribute's name
`count`	The number of array elements in vals
`vals`	Pointer to the first element of an array of t_symbol *s

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

Atom Utilities

Also present in "ext_obex.c" are some new atom utilities, used to get and set t_atom values. Direct access to t_atoms continues to be supported, but many developers may find these utilities more succinct and safe.

atom_setlong

Use the `atom_setlong` function to insert an integer into a t_atom and change the t_atom's type to `A_LONG`.

```
t_max_err atom_setlong(t_atom *a, long b);
```

`a`	Pointer to a t_atom whose value and type will be modified
`b`	Integer value to copy into the t_atom

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

atom_setfloat

Use the `atom_setfloat` function to insert a floating point number into a t_atom and change the t_atom's type to `A_FLOAT`.

```
t_max_err atom_setfloat(t_atom *a, double b);
```

`a`	Pointer to a t_atom whose value and type will be modified
`b`	Floating point value to copy into the t_atom

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

atom_setsym

Use the `atom_setsym` function to insert a t_symbol * into a t_atom and change the t_atom's type to `A_SYM`.

```
t_max_err atom_setsym(t_atom *a, t_symbol *b);
```

`a`	Pointer to a t_atom whose value and type will be modified
`b`	Pointer to a t_symbol to copy into the t_atom

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

atom_setobj

Use the `atom_setobj` function to insert a generic pointer value into a t_atom and change the t_atom's type to `A_OBJ`.

```
t_max_err atom_setobj(t_atom *a, void *b);
```

`a`	Pointer to a t_atom whose value and type will be modified
`b`	Pointer value to copy into the t_atom

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

atom_getlong

Use the `atom_getlong` function to retrieve a long integer value from a t_atom.

```
long atom_getlong(t_atom *a);
```

`a`	Pointer to a t_atom whose value is of interest

This routine returns the value of the specified t_atom as an integer, if possible. Otherwise, it returns 0.

If the t_atom is not of the type specified by the function, the routine will attempt to coerce a valid value from the t_atom. For instance, if the t_atom `at` is set to type `A_FLOAT` with a value of `3.7`, the `atom_getlong` function will return the truncated integer value of `at`, or `3`. An attempt is also made to coerce t_symbol data.

atom_getfloat

Use the `atom_getfloat` function to retrieve a floating point value from a t_atom.

```
float atom_getfloat(t_atom *a);
```

`a`	Pointer to a t_atom whose value is of interest

This routine returns the value of the specified t_atom as a floating point number, if possible. Otherwise, it returns 0.

If the t_atom is not of the type specified by the function, the routine will attempt to coerce a valid value from the t_atom. For instance, if the t_atom `at` is set to type `A_LONG` with a value of `5`, the `atom_getfloat` function will return the value of `at` as a float, or `5.0`. An attempt is also made to coerce t_symbol data.

atom_getsym

Use the `atom_getsym` function to retrieve a t_symbol * value from a t_atom.

```
t_symbol *atom_getsym(t_atom *a);
```

`a`	Pointer to a t_atom whose value is of interest

This routine returns the value of the specified `A_SYM`-typed t_atom, if possible. Otherwise, it returns an empty, but valid, t_symbol *, equivalent to `gensym("")`, or `_sym_nothing`.

No attempt is made to coerce non-matching data types.

atom_getobj

Use the `atom_getobj` function to retrieve a generic pointer value from a t_atom.

```
void *atom_getobj(t_atom *a);
```

`a`	Pointer to a t_atom whose value is of interest

This routine returns the value of the specified `A_OBJ`-typed t_atom, if possible. Otherwise, it returns `NULL`.

No attempt is made to coerce non-matching data types.

atom_getcharfix

Use the `atom_getcharfix` function to retrieve an unsigned integer value between 0 and 255 from a t_atom.

```
long atom_getcharfix(t_atom *a);
```

`a`	Pointer to a t_atom whose value is of interest

This routine returns the value of the specified t_atom as an integer between 0 and 255, if possible. Otherwise, it returns 0.

If the t_atom is typed `A_LONG`, but the data falls outside of the range 0-255, the data is truncated to that range before output.

If the t_atom is typed `A_FLOAT`, the floating point value is multiplied by 255. and truncated to the range 0-255 before output. For example, the floating point value `0.5` would be output from atom_getcharfix as `127` (0.5 * 255. = 127.5).

No attempt is also made to coerce t_symbol data.

atom_arg_getlong

Use the `atom_arg_getlong` function to retrieve the integer value of a particular t_atom from an atom list, if the atom exists.

```
t_max_err atom_arg_getlong(long *c, long idx, long ac,
      t_atom *av);
```

`c`	Pointer to a long variable to receive the atom's data if the function is successful.
`idx`	Offset into the atom list of the atom of interest, starting from 0. For instance, if you want data from the 3rd atom in the atom list, `idx` should be set to 2.
`ac`	Count of av.
`av`	Pointer to the first t_atom of an atom list.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

The `atom_arg_getlong` function only changes the value of `c` if the function is successful. For instance, the following code snippet illustrates a simple, but typical use:

```
void myobject_mymessage(t_myobject *x, t_symbol *s,
      long ac, t_atom *av)
{
      long var = -1;

      // here, we are expecting a value of 0 or greater
      atom_arg_getlong(&var, 0, ac, av);
      if (val == -1) // i.e. unchanged
            post("it is likely that the user did not
                  provide a valid argument");
      else {
            ...
      }
}
```

atom_arg_getfloat

Use the `atom_arg_getfloat` function to retrieve the floating point value of a particular t_atom from an atom list, if the atom exists.

```
t_max_err atom_arg_getfloat(float *c, long idx,
      long ac, t_atom *av);
```

`c`	Pointer to a float variable to receive the atom's data if the function is successful. Otherwise, the value is left unchanged.
`idx`	Offset into the atom list of the atom of interest, starting from 0. For instance, if you want data from the 3rd atom in the atom list, `idx` should be set to 2.
`ac`	Count of av.
`av`	Pointer to the first t_atom of an atom list.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

atom_arg_getdouble

Use the `atom_arg_getdouble` function to retrieve the floating point value, as a double, of a particular t_atom from an atom list, if the atom exists.

```
t_max_err atom_arg_getdouble(double *c, long idx,
      long ac, t_atom *av);
```

`c`	Pointer to a double variable to receive the atom's data if the function is successful. Otherwise the value is left unchanged.
`idx`	Offset into the atom list of the atom of interest, starting from 0. For instance, if you want data from the 3rd atom in the atom list, `idx` should be set to 2.

`ac`	Count of av.
`av`	Pointer to the first t_atom of an atom list.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

atom_arg_getsym

Use the `atom_arg_getsym` function to retrieve the t_symbol * value of a particular t_atom from an atom list, if the atom exists.

```
t_max_err atom_arg_getsym(t_symbol **c, long idx, long
      ac, t_atom *av);
```

`c`	Pointer to a t_symbol * variable to receive the atom's data if the function is successful. Otherwise, the value is left unchanged.
`idx`	Offset into the atom list of the atom of interest, starting from 0. For instance, if you want data from the 3rd atom in the atom list, `idx` should be set to 2.
`ac`	Count of av.
`av`	Pointer to the first t_atom of an atom list.

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

The `atom_arg_getsym` function only changes the value of `c` if the function is successful. For instance, the following code snippet illustrates a simple, but typical use:

```
void myobject_open(t_myobject *x, t_symbol *s,
      long ac, t_atom *av)
{
      t_symbol *filename = _sym_nothing;

      // here, we are expecting a file name.
      // if we don't get it, open a dialog box
      atom_arg_getsym(&filename, 0, ac, av);
      if (filename == _sym_nothing) { // i.e. unchanged
```

```
                // open the file dialog box,
                // get a value for filename
        }
        // do something with the filename
}
```

Miscellaneous Utilities

symbol_unique

Use the `symbol_unique` function to generate a unique t_symbol *. The symbol will be formatted somewhat like "u123456789".

```
t_symbol atom_setlong(void);
```

This routine returns a unique t_symbol *.

error_sym

Use the `error_sym` function to post an error message to the Max window. This function is interrupt safe.

```
void error_sym(void *x, t_symbol *s);
```

`x`	Your object's pointer
`s`	Symbol to be posted as an error in the Max window

post_sym

Use the `post_sym` function to post a message to the Max window. This function is interrupt safe.

```
void post_sym(void *x, t_symbol *s);
```

`x`	Your object's pointer
`s`	Symbol to be posted in the Max window

symbolarray_sort

Use the `symbolarray_sort` function to perform an ASCII sort on an array of t_symbol *s.

```
t_max_err symbolarray_sort(long ac, t_symbol **av);
```

`ac`	The count of t_symbol *s in `av`
`av`	An array of t_symbol *s to be sorted

This routine returns the error code `MAX_ERR_NONE` if successful, or one of the other error codes defined in "ext_obex.h" if unsuccessful.

object_obex_quickref

Developers do not need to directly use the `object_obex_quickref` function. However, to take advantage of a new quickref appearance, which provides support for attributes, developers should define the following method in `main()`.

```
class_addmethod(c /* your object class */,
      (method)object_obex_quickref, "quickref",
      A_CANT,0);
```

Making JavaScript code for Max pattr-aware

Adding pattr-compatibility to JavaScript code when using the **js** and **jsui** objects introduced in Max 4.5 is extremely straightforward. Since these objects already take advantage of the new obex class architecture, no major changes are required.

Just as with Max objects in C, two special functions need to be added to your JavaScript code in order for **pattr** and **autopattr** to attach to your object: `setvalueof()` and `getvalueof()`. These two functions are briefly discussed in the *Special Function Names* sections of the *JavaScript in Max* document.

The `setvalueof()` function is used to set your object's state, while `getvalueof()` queries it. A very simple sample implementation is shown below, for a JavaScript object whose state is defined by a single floating point value.

```
var the_object_value = 0.5;

function setvalueof(v)
{
	the_object_value = v;
}

function getvalueof()
{
	return the_object_value;
}
```

In cases where your data is more complicated, `setvalueof()` will receive multiple arguments (the developer can use JavaScript's `arguments` property to access a variable number of incoming arguments, or the **jsthis** object's `arrayfromargs()` method). Likewise, you can return an Array of data from `getvalueof()`. For instance:

```
var object_val_one = 0.5;
var object_val_two = 0.25;
var object_val_three = 0.125;

function setvalueof()
{
	if (arguments.length >= 3) {
		object_val_one = arguments[0];
		object_val_two = arguments[1];
		object_val_three = arguments[2];
	}
	do_something(); // some function to process the data
}

function getvalueof()
{
      var return_val = new Array();

      return_val[0] = object_val_one;
      return_val[1] = object_val_two;
      return_val[2] = object_val_three;

      return return_val;
}
```

Finally, **pattr** needs to be informed of any object state changes, to know when it should call the `getvalueof()` method. In JavaScript, the **jsthis** object's `notifyclients()` method is used for this purpose.

For instance, in a theoretical object, the special float-input function `msg_float()` might look like this:

```
function msg_float(d)
{
      the_object_value = d;
      notifyclients();
      outlet(0, d);
}
```

Please refer to the included source code example *jsob.js* for a simple example of JavaScript code with pattr-support using the techniques described above.

Beginning with Max 4.5.5, it is possible for objects written in JavaScript to contain attributes. These attributes may be bound to from the **pattr** and **autopattr** objects, just like Max and Jitter attributes.