Breve guida di riferimento per l'uso delle librerie maxmspsdk in semplici external objects

t_object

Each Max external object needs a C structure definition. If you're defining a normal object, it needs to start with a structure called a `t_object`. This field is not a pointer, but an entire structure contained inside your object. Typically, Max objects have followed the UNIX convention of starting fields of data structures with a lowercase letter followed by an underscore. The letter is normally the first letter of the name of the structure. Here's a hypothetical structure for an t_alarmclock object.

```
typedef struct alarmclock {
t_object a_ob; /* must be first in any non UI object
*/
long a_hours;
long a_minutes;
long a_seconds;
long a_alarmset;
} t_alarmclock;
```

The `t_object` contains references to your object's class definition as well as some other information. This class reference allows an instance of your class to respond to messages in the right way. You're free to use any data type you wish as a field in your object's structure. Keep in mind that Max stores floating point numbers internally as type `float`, so any extra precision not contained in a `float` may be lost. (This doesn't mean you can't perform extraprecision computation inside your object.) In addition, integers are passed from Max to functions you write as longs, and communicated to outlets and most other Max structures as longs.

setup

Use the `setup` function to initialize your class by informing Max of its size, the name of your functions that create and destroy instances, and the types of arguments passed to the instance creation function.

```
void setup (t_messlist **class, method createfun,
method freefun, short classSize, method menufun,
short types...);
```

`class`	A global variable in your code resource that points to the initialized class.
`createfun`	Your instance creation function.
`freefun`	Your instance free function (see Chapter 7).
`classSize`	The size of your objects data structure in bytes. Usually you use the C `sizeof` operator here.
`menufun`	Used only when you're defining a user interface object. It's the function that gets called when the user creates a new object of your class from the Patcher window's palette. Pass `0` if you're not defining a user interface object (how to write this function is discussed in Chapter 11).
`types`	A standard Max type list as explained in Chapter 3. The final argument of the type list should be a 0. This list specifies the arguments that are expected when a new instance of your class is created. These would be the arguments that the user types in after the name of your class.

As an example, imagine that you want to define a class for an object called + to accept one integer as an argument. The value 20 will be passed to the object's instance creation function. Here's the structure definition for such a class.

```
typedef struct _myobj {
struct object m_ob; long m_watchtower;
} t_myobj;
```

Here are the prototypes of the creation and free functions.

```
void *myobj_new (long arg);
void myobj_free (t_myobj *x);
```

Here is the global variable that points to the class .

```
void *myobj_class;
```

Here is beginning of the initialization routine, with the call to `setup`.

```
void main(void)
{
setup ((t_messlist **)&myobj_class, (method)myobj_new,
(method)myobj_free, (short)sizeof(t_myobj), 0L, A_DEFLONG, 0);
/* additional code will go here */
}
```

After calling `setup`, you'll have a well-defined class that doesn't know how to do anything. In order to make it useful, you need to make the class respond to messages. This means that a t_symbol (such as the word bang) is bound to a function you write (often called a method). We'll discuss the functions you'll use to do this in a moment. There are functions are designed to make it easy to add standard messages to your class, along with `addmess`, which allows you to specify novel messages and give them arguments that will be type-checked for you by Max.

Basic type list specifiers:

`A_NOTHING`	Ends the type list.
`A_LONG`	Type-checked integer argument
`A_FLOAT`	Type-checked float argument
`A_SYM`	Type-checked symbol argument
`A_OBJ`	Pointer argument (obsolete)
`A_DEFLONG`	Type-checked integer argument that defaults to 0
`A_DEFFLOAT`	Type-checked float argument that defaults to 0
`A_DEFSYM`	Type-checked symbol argument that defaults to 0
`A_GIMME`	You can only specify up to seven arguments in a list.

However, you can specify that Max just hand you the arguments as an array of t_atoms (a structured type defined below) if you use the following type list:

```
A_GIMME, 0
```

This allows you to type check the arguments yourself, and no limit is placed on the number of arguments that can be included in such a message.

Untyped messages are those whose type list contains the constant A_CANT and cannot be sent directly by a user using a message box connected to an inlet of your object.

addbang

Used to bind a function to the common triggering message bang.

```
void addbang (method mp);
```

`mp` Function to be the bang method.

addint

Use `addint` to bind a function to the int message received in the leftmost inlet.

```
void addint (method mp);
```

`mp` Function to be the int method.

addinx

Use `addinx` to bind a function to a int message that will be received in an inlet other than the leftmost one.

```
void addinx (method mp; short inlet);
```

`mp` Function to be the int method.

`inlet` Number of the inlet connected to this method. 1 is the first inlet to the right of the left inlet.

Note: This correspondence between inlet locations and messages is not automatic, but it is strongly suggested that you follow existing practice. You must set the correspondence up when creating an object of your class with proper use of `intin` and `floatin` in your instance creation function (see Chapter 6).

addmess

Use `addmess` to bind a function to a message other than the standard ones described above.

```
void addmess (method mp; char *sym; short types...);
```

`mp` Function you want to be the method.
`sym` C string defining the message.
`types` One or more integers specifying the arguments to the message, in the standard Max type list format (see Chapter 3).

The `addmess` function adds the function pointed to by `mp` to respond to the message string `sym` in the leftmost inlet of your object. Type checking of the message's arguments can be done by passing a list of argument type specifiers. The list must end with a 0 (`A_NOTHING`). The maximum number of type-checked arguments is 7.

gensym

Use `gensym` to convert a character string into a t_symbol.

```
t_symbol *gensym (char *string)
```

`string` C string to be looked up in Max's symbol table. If the string is not present, a new symbol is created.

`gensym` takes a C string and returns a pointer to the `t_symbol` associated with the string. Max maintains a symbol table of all strings to speed lookup for message passing. If you want to access the bang symbol for example, you'll have to use the expression `gensym("bang")`. You may need to use `gensym` in writing a User Interface object's psave method to save extra data besides the object's box location and arguments. Or `gensym` may be needed when sending messages directly to other Max objects such as with `typedmess` and `outlet_anything`. These functions expect t_symbols—they don't `gensym` character strings for you.
The `t_symbol` data structure contains a place to store an arbitrary value. The following example shows how you can use this feature to use symbols to share values among two different external object classes. (Objects of the same class can use the code resource's global variable space to share data.) The idea is that the `s_thing` field of a `t_symbol` can be set to some value, and `gensym` will return a reference to the Symbol. Thus, the two classes just have to agree about the character string to be used. Alternatively, each could be passed a `t_symbol` that will be used to share data.
Storing a value:

```
t_symbol *s;
s = gensym("some_weird_string");
s->s_thing = (t_object *)someValue;
```

Retrieving a value:

```
t_symbol *s;
s = gensym("some_weird_string");
someValue = s->s_thing;
```

post

Use `post` to print text in the Max window.

```
void post (char *fmtstring, void *items...);
```

`fmtstring` A C string containing text and printf-like codes specifying the sizes and formatting of the additional arguments.
`items` Arguments of any type that correspond to the format codes in `fmtString`.

`post` is a printf for the Max window. It even works at interrupt level, queuing up to four lines of text to be printed when main event level processing resumes. `post` can be quite useful in debugging your external object. Note that `post` only passes 16 bytes of arguments to `sprintf`, so if you want additional formatted items on a single line, use `postatom`.

Example:

```
short whatIsIt;
whatIsIt = 999;
post ("the variable is %ld",(long)whatIsIt);
```

The Max Window output when this code is executed.

```
the variable is 999
```

error

Use `error` to print an error message in the Max window.

```
void error (char *fmtstring, void *items...);
```

`fmtstring` A C string containing text and printf-like codes specifying the sizes and formatting of the additional arguments.

`items` Arguments of any type that correspond to the format codes in `fmtString`.
The `error` function writes a line of text printf-style into the Max window like `post`, preceded by the attention-getting string `* error`. Note that by using this routine to post errors, you let users trap the messages using the **error** object.

Example:

```
error ("bad arguments to %s",myclassname);
```

Max Window output:

```
• error: bad arguments to myclass
```

outlet_int

Use `outlet_int` to send a int message out an outlet.

```
void *outlet_int (Outlet *theOutlet, long n);
```

`theOutlet` Outlet that will send the message. `n` Integer value to send.

newobject

Use `newobject` to allocate the space for an instance of your class and initialize its object header.

```
void *newobject (void *class);
```

`class` The global class variable initialized in your main routine by the `setup` function.

You call `newobject` when creating an instance of your class in your creation function. `newobject` allocates the proper amount of memory for an object of your class and installs a pointer to your class in the object, so that it can respond with your class's methods if it receives a message.

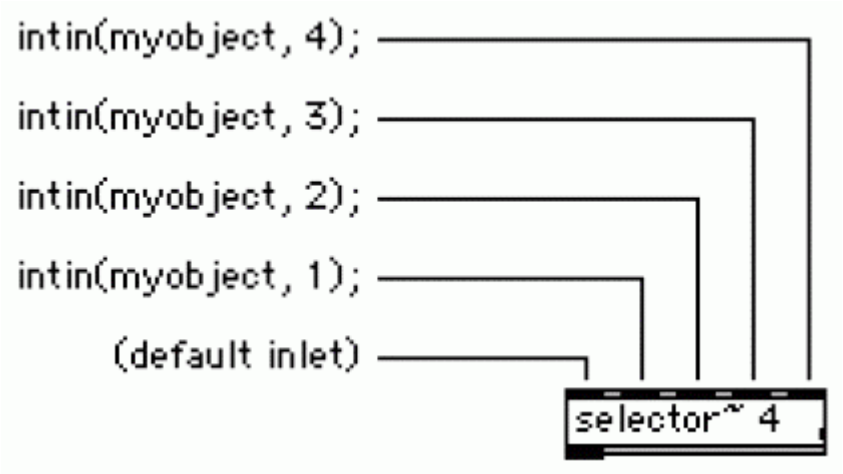
intin

Use intin to create an inlet typed to receive only integers.

```
void intin (void *object, short index)
```

`object` Your object. `index` Location of the inlet from 1 to 9. 1 is immediately to the right of the leftmost inlet.

`intin` creates integer inlets. It takes a pointer to your newly created object and an integer `n`, from 1 to 9. The number specifies the message type you'll get, so you can distinguish one inlet from another. For example, an integer sent in inlet 1 will be of message type in1 and a floating point number sent in inlet 4 will be of type ft4. You use `addinx` and `addftx` to add methods to respond to these messages. The order you create additional inlets is important. If you want the rightmost inlet to be the have the highest number in- or ft- message (which is usually the case), you should create the highest number message inlet first. Creating four additional integer inlets (for a total of five) would provide the following:



intout

Use `intout` to create an outlet that will always send the int message.

```
Outlet *intout (void *owner);
```

`Owner` Your object.

Here's an example of using `intout` that creates an outlet that will be used to send integers:

```
mynewobject->m_intout = intout(mynewobect);
```

defer_low

Use `defer_low` to defer execution of a function to the main level.

```
void defer_low (t_object *client, method fun, t_symbol *s, short argc,
t_atom *argv);
```

`client`	First argument passed to the function `fun` when it executes.
`fun`	Function to be called, see below for how it should be declared.
`s`	Second argument passed to the function `fun` when it executes.
`argc`	Count of arguments in `argv`. `argc` is also the third argument passed to the function fun when it executes.
`argv`	Array containing a variable number of function arguments. If this argument is non-zero, `defer` allocates memory to make a copy of the arguments (according to the size passed in `argc`) and passes the copied array to the function `fun` when it executes as the fourth argument.

`defer_low` always defers a call to the function `fun` whether you are at interrupt level or not, and uses `qelem_set`, not `qelem_front`. This function is recommended for responding to messages that will cause your object to open a dialog box, such as read and write.

This defer functions use the isr routine to determine whether you're at the Max timer interrupt level. If so, defer creates a Qelem, calls `qelem_front`, and its queue function calls the function `fun` you passed with the specified arguments. If you're not at the Max timer interrupt level, the simple defer function is executed immediately with the arguments. Note that this implies that defer is not appropriate for using in situations such as Device or File manager I/0 completion routines. `defer_low` is appropriate because it always defers and places events at the back of the low priority queue.

open_dialog

Use `open_dialog` to present the user with the standard open file dialog.

```
short open_dialog (char *filename, short *path,
OSType *dstType, SFTypeList typelist, short
numtypes);
```

`filename`	A C string that will receive the name of the file the user wants to open.
`Path`	Receives the Path ID of the file the user wants to open.
`DstType`	The file type of the file the user wants to open.
`Typelist`	A list of file types to display. This is not limited to 4 types as in the SFGetFile trap. Pass 0L to display all types.
`Numtypes`	The number of file types in `typelist`. Pass 0 to display all types.

This function is convenient wrapper for using Mac OS Navigation Services or Standard File for opening files. `open_dialog` returns 0 if the user clicked Open in the dialog box, and returns the name of the file picked as a C string in `filename`, its volume reference number in `vol`, and its file type in `dstType`. If the user cancelled, `open_dialog` returns a non-zero value.
The standard types to use for Max files are ‘maxb’ for binary files and ‘TEXT’ for text files.

locatefile_extended

Use `locatefile` to find a Max document by name in the search path. This is the preferred method for file searching in Max 4.

```
short locatefile_extended(char *name, short
*outpath, long *outtype, long *typelist, short
numtypes);
```

`name` The file name for the search, receives actual filename.
`outpath` The Path ID of the file (if found).
`outtype` The file type of the file (if found).
`typelist` The file type(s) that you are searching for.
`numtypes` The number of file types in the typelist array (1 if a single entry).

The existing file search routines `locatefile` and `locatefiletype` are still supported in Max 4, but the use of a new routine `locatefile_extended` is highly recommended. However, `locatefile_extended` has an important difference from `locatefile` and `locatefiletype` that may require some rewriting of your code. It modifies its name parameter in certain cases, while `locatefile` and `locatefiletype` do not. The two cases it where it could modify the incoming filename string are 1) when an alias is specified, the file pointed to by the alias is returned; and 2) when a full path is specified, the output is the filename plus the path number of the folder it's in. This is important because many people

pass the s_name field of a `t_symbol` to `locatefile`. If the name field of a symbol were to be modified, the symbol table would be corrupted. To avoid this problem, use strcpy to copy the contents of a t_symbol to a character string first, as shown below:

```
char filename[256];
strcpy(filename,str->s_name);
result = locatefile_extended(filename,&path,&type,typelist,1);
```

path_topathname

Use `path_topathname` to create a fully qualified file name from a Path ID/file name combination.

```
short path_topathname(short path, char *file, char *name);
```

`path` The path to be used.
`File` The file name to be used.
`Name` Loaded with the fully extended file name on return.

Unlike `path_topotentialname`, this routine will only convert a pathname pair to a valid path string if the path exists. Returns 0 if successful, and an error code if unsuccessful.

path_nameconform

Use `path_nameconform` to convert from one filepath style to another.

```
short path_nameconform(char *src, char *dst, long style, long type);
```

`src` A pointer to source character string to be converted.
`dst` A pointer to destination character string.
`style` The destination filepath style (slash style, colon style, etc…).
`type` The destination filepath type (absolute, relative, etc…).

Converts a source path string to destination path string using the specified style and type. The available path styles are as follows:

`PATH_STYLE_MAX` As of version 4.3, Max's path style is the slash style.
`PATH_STYLE_NATIVE` This is the path style native to the operating system.
`PATH_STYLE_COLON` The Mac OS 9 path style used by Max 4.1 and earlier.
`PATH_STYLE_SLASH` The cross-platform path style used by Max 4.3 and later.
`PATH_STYLE_NATIVE_WIN` The Windows backslash path style (not recommended, since the backslash is a special character in Max).

The available path types are:

`PATH_TYPE_IGNORE`	Do not use a path type.
`PATH_TYPE_ABSOLUTE`	A full pathname.
`PATH_TYPE_RELATIVE`	A path relative to the Max application folder.
`PATH_TYPE_BOOT`	A path relative to the boot volume
`PATH_TYPE_C74`	A path relative to the “Cycling’74” folder.

The function’s return value is an error code.