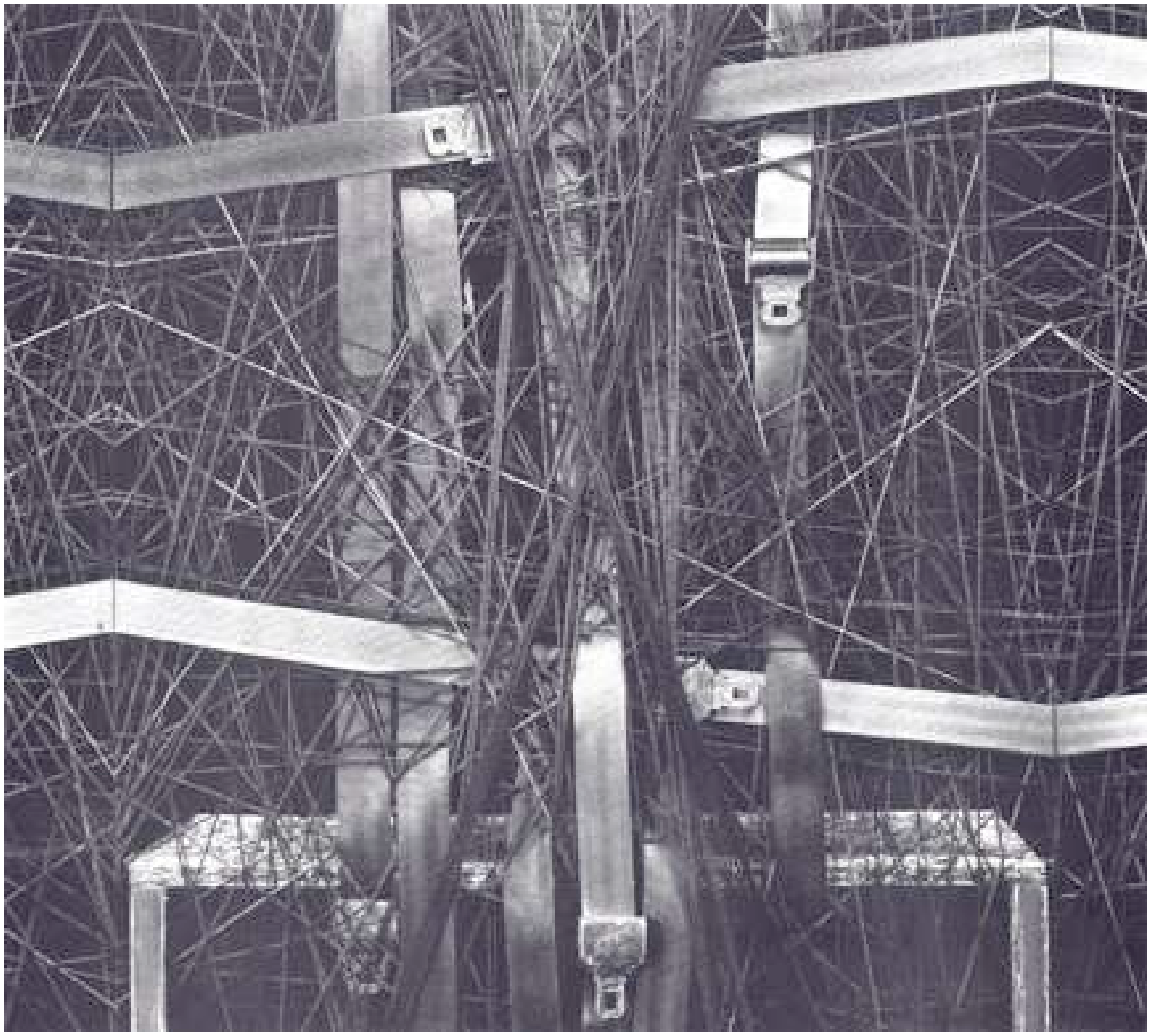
MAX



Writing External Objects for Max and MSP

Revision 12 / 3 August 2005

Copyright and Trademark Notices

Credits

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

Table of Contents

Chapter 1: Overview

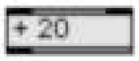
Writing External Objects for Max gives you an inside peek at the Max environment and shows how it can be extended by creating objects in the C language. This document assumes some familiarity with Max from a user's standpoint. However, we'll try to show the connection between the programming constructs presented and how they appear to the user.

In writing an external object for Max, your task is to write a shared library (on the Macintosh) or a dynamic link library (on Windows XP) that is loaded and called by the "master environment" and in turns calls upon helpful routines back in the master environment. You create a *class*, or template for the behavior of an object. Instances of this class "do the work" of the object, when they are sent messages. Your external object definition will:

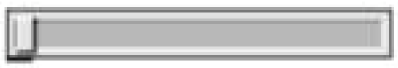
- Define the class: its data structure, size, and how instances are to be created and destroyed

- Define functions (called *methods*) that will respond to various messages, performing some action

There are several types of external objects you can write. *Normal Objects* show up in Patcher windows in New Object boxes with two lines at the top and bottom, like this:



User Interface Objects are a bit more complicated to write, but they can have any appearance and behavior in a Patcher window, such as this **hslider**:



In addition, you can write external objects for the Timeline that function either as Actions or Editors.

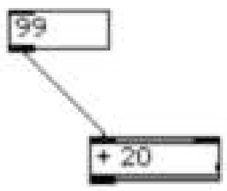
Typically, if you'll be interfacing Max to the outside world or performing some computation, you'll write a normal object. Normal objects can also open their own windows and dialog boxes. But they can't do any drawing or event handling in the Patcher window itself. To do any drawing or user interaction within a Patcher window, you'll need to do some extra work to make a user interface object.

As mentioned above, there are two phases of your object definition: class initialization time and what could be called object behavior time. Your shared library (Macintosh) or dynamic link library (Windows XP) can be loaded when Max starts up if it's placed in Max's startup folder (usually called *max-startup*), or it can be loaded whenever Max wants to create the first instance of your object. At this time your object's `main` function will be called, and it should initialize its class. The main function is the only entry point your object will define. By initializing the class, it will tell Max about all of the other functions defined in the object. We'll explain more about this process in chapter 4. After the class is initialized, Max will not call functions in your external until someone creates an object (instance) of your class. This happens…

- when a patcher file is read in
- when someone types the name of your object into a New Object box in a Patcher window
- when duplicating an existing object of your class

Your object's *instance creation function* will be called at this point. In this routine, you'll allocate memory for a new object of your class, and do additional initialization of the fields in the object.

After the object has been created, it can receive messages. When a number is sent in an object's inlet, the object receives an int message (or a float message, if the number is floating point).



You need to write a *method* to respond to this message. If you were an object that performed addition, your int method might add two numbers together and send the result out your outlet.

There are a number of predefined messages your object can respond to. You can also define your own messages. Defining messages and associating methods with them is done at initialization time when you're setting up your class.

Finally, if your object is deleted, your object's *free function* will be called. If you didn't allocate any extra memory inside your object (assigned to any of your object's fields), you need not have a free function. Otherwise, you should free the memory used by these fields in this function.

About This Manual

This manual should be used in conjunction with the Example Objects supplied with the Max Software Development Kit. Copying an example as the basis of your object is the preferred method to start developing a Max external.

Conventions

The task of writing an external also involves a choice of C language development environments. The examples assume the use of the Metrowerks CodeWarrior environment on the Macintosh, but the next chapter discusses getting started using either CodeWarrior or Apple's MPW environments on the Macintosh or other compilers on Windows XP.

In this manual, the names of Max messages are printed like this (message) and almost always lowercase. Names of existing (built-in) Max objects are printed like this (**object**). Other Max programming names and constructs are printed like this (`wind_drag`). Any messages you might see in the Max window will be printed like this (*Method not found).*

Chapter 2: Basics

The Choice of Development Environments

This chapter discusses preparing to write your object by choosing and configuring a development environment, the include files you'll need, and some of the general techniques specific to Max externals you'll need to use.

On Macintosh OSX, Max/MSP objects use the CFM object code format. (The ability to make Mach-O externals will be possible in a future release of Max, but for the time being Mach-O externs are not currently supported.) This means you have to use Code Warrior, not Project Builder, to make externs at the present time. One advantage of using CFM is that many existing externals made for Max 4.1 and earlier can be used unchanged in Max 4.2 and 4.3; as long as the externs don't make any calls to InterfaceLib (toolbox functions) or MathLib (math functions) they should work. All Macintosh files included with the SDK were built for version 8.x of Code Warrior. We suggest getting the latest release possible.

On Windows XP, The project files included in the SDK are designed for MS Visual Studio .NET 2002 or later. Sample project files for MS Visual Studio 6.0 are also included. We will also discuss the important compiler settings to help assist those who would like to set up projects using other development environments.

The c74support Folder

This folder contains all the headers needed for creating Max, MSP, and Jitter external objects for both Mac OS and Windows, plus some commonly used files. On the Macintosh this folder should be placed in your Metrowerks Code Warrior folder (also known as the "compiler" folder) in order to compile the examples provided. On Windows it should be placed at the same level as your Max external development directory although you could place it elsewhere if you specify the proper include folders in your compiler settings.

Creating Projects on Mac OSX Using Metrowerks Code Warrior 8

The SDK examples we've provided were built on the Macintosh using our new CodeWarrior Project Template files. These are XML files that work with Code Warrior 8 to allow you to easily set up a new project. First we'll explain how our new .mcp (Metrowerks CodeWarrior Project) files work to build externs for both Mac OS X and Mac OS 9. This may be of help to you in trying to maintain multiple versions of your objects.

If you open the project file hslider.mcp you will see that it contains four targets (check the menu at the top of the project list window), debug, release, debug-classic and release-classic. For OS X we are only concerned with debug and release. The classic targets allow you to make versions of an object that will work with Mac OS 9 versions of Max/MSP.

Make sure the debug target is selected and look at the list of files You'll see that a number of these files are not included in the debug target because they do not have a dot in the “target” column (the icon is a red-and-white target with an arrow pointing at it). These files are CarbonAccessors.o, MathLib, and InterfaceLib.

However, CarbonLib is included. Switching to one of the classic targets, you see that CarbonLib is not included, but these other three files are. This is the important step in building an external object for OS X — you have to use CarbonLib.

However, if you open the maximum.mcp file, you'll see that there is no separate classic-debug or classic-release target, and none of the "lib" files mentioned above is included. That's because the maximum object doesn't need any of these files in order to work. It is therefore a “universal” object that should work with both Mac OS X and Mac OS 9 versions. Any object that makes direct use of the Mac OS will not be universal, and you'll need to have separate classic and carbon targets. The good news is that our project template files can set up all of this for you.

To use the CodeWarrior Project Templates you first need to create a folder with your source (.c) files and resource files, which should be called objectname.c and objectname.rsrc. We suggest naming the folder the same as your object. Next, choose an appropriate project template. The following list should provide a good idea of which template file you'll want to use:

- If you have a Max (not MSP) object that doesn't call any Mac toolbox or math functions, try max.common.mcp.xml
- If you have a Max (not MSP) object that calls math functions but not toolbox functions, try max.mathlib.mcp.xml
- If you have a Max user-interface object, try max.startup.ACAF.mcp.xml
- If you have a Max object that makes toolbox graphics calls (but isn't a UI object), try max.interfacelib.ACAF.mcp.xml
- If you have a Max object that makes toolbox calls but doesn't deal with GrafPorts, Windows, or Dialogs, try max.interfacelib.mcp.xml
- If you have an MSP object that doesn't call any Mac toolbox or math functions, try msp.common.mcp.xml
- If you have an MSP object that calls math functions but not toolbox functions, try msp.mathlib.mcp.xml
- If you have an MSP user-interface object, try msp.startup.ACAF.mcp.xml
- If you have an MSP object that makes toolbox graphics calls (but isn't a UI object), try msp.interfacelib.ACAF.mcp.xml
- If you have an MSP object that makes toolbox calls but doesn't deal with GrafPorts, Windows, or Dialogs, try msp.interfacelib.mcp.xml

Note: The “ACAF” project templates include CarbonAccessors.o, plus they use the MacHeadersACAF.h, which defines `ACCESSOR_CALLS_ARE_FUNCTIONS` to allow Carbon Accessor functions to be used in classic targets. In many cases this allows you to write Carbonized code that works on both platforms.

After opening the desired template file, save it in a folder containing your .c and .rsrc files. Name the file objectname.xml, then choose Find and Replace... from the Search menu. Enter C74OBJECTNAME as the find string, and the name of your object as the replace string. Click Replace All. Save your .xml file and close its window. The .xml file is now set up to provide CodeWarrior all of the information needed to create a CodeWarrior project file (.mcp) that is correctly configured to build your external object.

From within CodeWarrior choose Import Project... from the File menu, and select the .xml file you just saved (i.e. myobjectname.xml). CodeWarrior will then prompt you to save the project file. Save the file as objectname.mcp in the same folder as your .c and .rsrc files. The new project will open. It will include your .c and .rsrc file. By default, the debug target will be selected. Now it's time to start working on getting your external to compile. Assuming you can get your object to compile, it will be built (along with a .xSYM file if you are compiling a non-optimized debug target — see below) in a folder called "build-mac" (or "build-startup-mac" for UI objects) at the same level as your project folder.

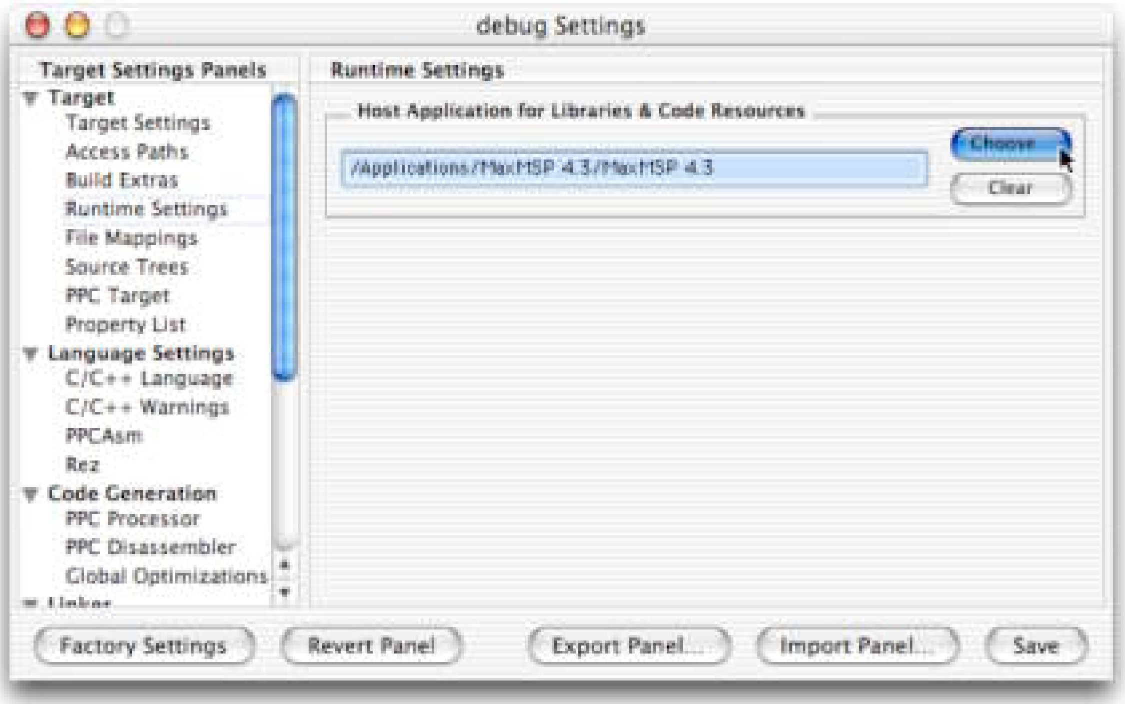
Debugging Under Mac OSX

The CodeWarrior debugger uses GDB, which is installed by the Mac OS X Developer Tools CD. If you don't have this CD, you won't be able to debug. And even if you do, getting the Code Warrior debugger to land in your code seems currently to be a hit-or-miss proposition. It works for some people but not others. There are a couple of general points we can make:

- Click the "debug" column for your .c file in the project. This isn't enabled by default.

- Objects are built in build-mac directory at the same level as the directory containing your source. (Startup UI objects are built in build-startup-mac.) Also included is the .xSYM file for each object. You'll need to copy the .xSYM file to wherever you copy the object (if you in fact copy it at all). Double-click on the .xSYM file to load it into the debugger, and try setting a breakpoint. As long as Max/MSP loads the copy of the extern in the same folder as the .xSYM file, it might just work.

You can also try setting the Host Application in the Runtime Settings panel of the target settings (it'll say debug Settings... in the Edit menu for the debug target). Choose your copy of Max/MSP or Max/MSP Runtime.



/Applications/MaxMSP 4.3/MaxMSP 4.3

Another useful debugging resource is the Crash Reporter log. It's in your user home directory's Library folder, inside the Logs folder, then inside Crash Reporter. Each application gets its own file, and each time your object crashes, it's ultimately Max/MSP that crashes so look in the Max/MSP file for details of what was on the stack when your object crashed. It might reveal something. However, note that Crash Reporter appends crash information to any existing file for Max/MSP, so read from the bottom rather than the top.

Note: You'll get the Crash Reporter log whether or not you are debugging with the Metrowerks debugger.

Using the QuickTime SDK for Windows on Windows XP

The QuickTime SDK for Windows from Apple has some very useful tools that may assist porting an external object from the mac to windows. You can optionally download this from http://developer.apple.com/quicktime/. The MaxMSP SDK zip file download available from Cycling '74 contains empty folders as an example of how the build environment could be set up to use the QuickTime SDK for Windows. The sample projects are configured to use this folder layout so we recommend you use it. Copy the header files from the QuickTime SDK to the win-sdks\QuickTime\inc folder, the .lib files to the win-sdks\QuickTime\lib folder, and the Tools (.exe) to the win-sdks\QuickTime\tools folder.

On the macintosh, resource files are used to add assistance strings, icons, and other data to external objects. The same macintosh style resources can be appended to a max external on Windows using the QuickTime SDK rezwack tool. The sample projects for Visual Studio .NET use build events to automate this process. Note that you can also simply add assistance strings by hard-coding them into the source code.

The QuickTime SDK for Windows can also be helpful in porting QuickDraw code. The QTML library contains much of the QuickDraw API and is an alternative to porting your drawing code to use native Win32 API calls. The xgui sample in the MaxMSP SDK includes projects that show both native drawing (xgui.vcproj) as well as drawing with QTML (qtxgui.vcproj).

Creating Projects on Windows XP Using MS Visual Studio

The quickest way to move your project from Macintosh to Windows (provided that your object has no operating system dependencies) or to start a new project on Windows, is to follow the following steps. Before you try building your own project, though, please make sure that the project files provided in the SDK are building properly for you.

First, copy an existing project file (.vcproj for Visual Studio .NET or .dsp for Visual Studio 6.0) such as minimum.vcproj to a new folder containing your object source code. For the project settings to work as is you will need to place your project files inside a folder at the same level as the other sample projects. In other words, create a folder inside the “projects” folder of the MaxMSP SDK. Also copy an existing definition file (.def) such as minimum.def to the same folder containing your object source code, and rename the copied project and definition files to something appropriate for your object (e.g. myobject.vcproj and myobject.def)

To help you decide which project file is most suited to use as a template, here is a brief overview of four of the SDK examples:

- minimum.vcproj is appropriate for non-UI externs which have no MacToolbox dependencies (not including simple things like standard math or string libraries, which are provided by the Windows standard libraries). (Later in this document we will see that max has built-in cross-platform methods for common things like file access, window management, memory allocation, and more.)

- lores~.vcproj is appropriate for non-UI MSP externs which have no MacToolbox dependencies (not including simple things like standard math or string libraries, which are provided by the Windows standard libraries).

- xgui.vcproj is appropriate for UI objects which have no MacToolbox dependencies (you will be writing windows native UI code). See the section concerning the integration of native Windows code in Max externals for more information.

- qtxgui.vcproj is appropriate for UI or non-UI externs which have MacToolbox dependencies and will be using QTML for the MacToolbox functions it provides. See the section on QTML for more information.

Next, open your newly named .vcproj (or .dsp) and .def as a *text* files in a text editor which supports Unix line breaks and can convert from Macintosh-style formatting. A good example is Metapad (a free download from http://www.liquidninja.com/metapad/). You will then need to find and replace all instances of the original object name (e.g. “minimum” in this case) with your object name (e.g. “myobject”) in both the .def and .vcproj files. Once you have done this, save the modified .vcproj and .def files, and then re-open as a project file in Visual Studio. Visual Studio will automatically generate a "solution" file (.sln) for your project.

At this point you will need to add any additional source files if necessary. The file <objectname>.c will already be included in the project (e.g. “myobject.c”). If your object uses a Macintosh resource file for things like assistance stings, we recommend recoding the assistance strings in C at least for the time being. If you would like to continue using Macintosh resource files for this information, you can do so by using a post-build step and the rezwack.exe program included in the QTML development materials. This will be covered in more detail in a future revision of this document. Windows resources could also be used for assistance strings, but the use of Windows resources is not covered in this document.

Now you can build your external object. Select Build->Build Solution. The final output file will be located in the "..\..\sysbuild\win_debug\externals\", "..\..\sysbuild\win_release\externals\" or similarly located “max-startup” folder. If there were no errors, place the output file in the appropriate folder in the max search path. Alternately, you could add your output folder to the max search path. Note that Max externals use the ".mxe" filename extension, so despite the fact that you build them the same way you would any other DLL, they should not have a ".dll" extension.

Although a comprehensive troubleshooting guide for the various errors you might see if your external does not compile immediately is beyond the scope of this document, we will nonetheless outline some common errors or problems and possible solutions or suggestions:

- Your code makes use of MacToolbox code and hence there are errors. Please refer the sections concerning the "Sys" APIs (Syswindow, Sysfile, Sysmem), native Windows code, and the use of QTML for a few ways to approach cross platform development of max externals.

- The compiler shows lots of warnings like "could not convert LPRECT to Rect *" or similar. Your code contains Macintosh functions which have the same name as Windows functions which operate differently.

- The compiler shows warnings like "implicit conversion from t_myobject * to t_object *" or "void * to t_messlist *". You should explicitly cast such pointers to the appropriate type.

- The compiler complains that it can't find Max header files or MaxAPI.dll or MaxExt.dll. The c74support folder needs to be located at "..\..\c74support" relative to your project file.

- The compiler complains that it can't find some QTML headers or libraries. Please read the section concerning the use of QTML and the assumed locations of the QuickTime for Windows SDK.

  Here are some other things to beware of and to keep in mind:

  - When implementing methods that accept floats from max make sure your function prototypes accept the arguments as double and not as float. This may work on the macintosh but will cause your function to receive incorrect data under windows.

  - One common source of problems when porting from the mac to windows seems to be uninitialized data. It seems the mac newly allocated memory tends to be zeroed more often on the mac.

- The compiler or linker complains of “unexpected tokens following preprocessor directive - expected a newline” or “unresolved external symbol main”. Most likely, one or more of the source files were copied directly from a Macintosh, and the line breaks are not appropriate for VC++. Use Metapad (or an equivalent) to convert the source file to unix or dos line breaks.

Creating Projects on Windows XP Using Other Development Environments

Creating a max external on other development environments should be fairly straightforward. Here is a discussion of the necessary compiler settings. First we will discuss a max-only external. Then we will discuss the necessary settings to add when using MSP features. Finally we will discuss the necessary settings to add when using the QuickTime SDK for Windows.

Basic settings for all externals:

Project Type: Win32 Dynamic Link Library

- Include directories: “..\..\c74support\max-includes”
- Preprocessor definitions: WIN_VERSION
- Struct member alignment: 2 Bytes. Don’t forget this!
- Output file: <name.mxe> (in other words, specify a .mxe extension rather than .dll).
- C Runtime: The Multithreaded DLL version of the C runtime library is recommended.

Linker libraries:

- Debug: include “..\..\c74support\max-includes\win-includes\debug\MaxAPI.lib”
- Release: include “..\..\c74support\max-includes\win-includes\release\MaxAPI.lib”
- DLL Exports: main is the only exported function. You can export via a .DEF file or via the __declspec(dllexport) specifier.

Additional settings for MSP projects (add these to the above settings):

- Include directories: “..\..\c74support\msp-includes”

- Linker libraries:
    - Debug: include "..\..\c74support\msp-includes\win-includes\debug\MaxAudio.lib"
    - Release: include "..\..\c74support\msp-includes\win-includes\release\MaxAudio.lib"

Additional settings for projects using QuickTime SDK for Windows:

- Preprocessor definitions: USE_QTML
- Include directories: "..\..\win-sdks\QuickTime\inc
- Linker libraries: include "..\..\win-sdks\QuickTime\lib\qtmlClient.lib"

Header Files

A number of necessary C header files are provided in the folder `c74support`. They should be included in your source files as follows:

ext.h	Required for all MAX external objects.
ext_proto.h	Used by ext.h when compiling a PowerPC external.
ext_support.h	Used only for 68K externals, and not maintained.
ext_mess.h	Not necessary to include directly. Used by ext.h.
ext_common.h	Commonly used macros and definitions.
ext_qtimage.h	Scaling definitions for Quicktime image handling.
ext_sndfile.h	A structure typedef for a sound file.
ext_string.h	Prototypes for string handling functions.
ext_path.h	Needed to provide system non-specific file path information (Chapter 8).
ext_wind.h	Needed for objects that put up their own windows (Chapter 10).
ext_user.h	Used when writing user interface objects (Chapter 11).

ext_colors.h	Color palette defitions for user interface objects (Chapter 11).
ext_menu.h	Needed for methods that respond to the chkmenu message (Chapter 10).
edit.h	The data structure for the text editor (Chapter 9).
ext_edit.h	Used when interfacing to the text editor window (Chapter 9).
ext_anim.h	Needed for objects that use the graphic window or sprites (Chapter 12).
ext_expr.h	Needed when using the interface to the **expr** object (Chapter 9).
ext_numc.h	Needed when using the Numerical routines (Chapter 10).
ext_midi.h	Needed when accessing MIDI Manager structures (see **timein** example code).
ext_event.h	Needed when writing Timeline Editors (Chapter 14).
ext_track.h	Needed when writing Timeline Editors (Chapter 14).

You may also need specific include files related to Macintosh data structures you deal with.

As you read through this document, you may notice that the type definitions for many of the Max structures discussed here are listed in ext.h as pointer to void (void *). This is done when the internal structure is not important for use.

Function Prototypes

Max provides you with a sizable number of functions to assist you in writing external objects. These are often the same functions that the objects built into Max use to carry out their work. These functions present a programming interface similar to the one you'd have available writing the object in Max itself.

Within the Max application, your object is dynamically linked at runtime when it is loaded. Loading occurs either when Max starts up and your object is in the max-startup folder, when it is loaded explicitly using the Install… command in the File menu, or when someone types the name of your object into an object box or reads in a patcher file containing a reference to your object.

This dynamic loading allows you to use prototyped Max function calls, with complete compile-time error checking. If you want to refer to these prototypes, open the file ext_proto.h in the `MAX includes` folder.

Object Header

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 extra-precision 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.

Chapter 3: Data Types and Argument Lists

Max can provide you with the service of type checking arguments to messages destined for your object. The two functions `setup` and `addmess` take argument type specification lists used for performing this task.

Your object creation function is called in response to a message sent to a special new object. You pass the same kind of argument type specification list to `setup` as you do when defining the arguments to your own messages. In the case of a message to the new object, the name of the class is the "message selector" itself, and the arguments are what follows the class name (the 20 in + 20, for example).

The function `addmess`, like `setup`, takes argument type specification lists. For example, suppose we want to define a message search that requires two long integers as arguments. The user might type such a message into a Max message box connected to our object.

In this case, search is the message, and 304 and 228 are the arguments. The argument type list would look like this:

```
A_LONG, A_LONG, 0
```

Argument type lists always end with a zero (defined as `A_NOTHING`).

You would declare the arguments to the function you write (that respond to the search message) as follows:

```
void myobject_search (myObject *x, long arg1, long arg2);
```

Floating point numbers can be specified with `A_FLOAT`, and Symbols (text words) with `A_SYM`.

If you want arguments to be optional, you can use `A_DEFLONG`, `A_DEFFLOAT`, and `A_DEFSYM`. These default missing arguments to 0, 0.0, and the empty symbol ("") respectively.

In each case where you declare explicit arguments, Max will pass arguments of the specified type directly to your method. If the arguments are of the wrong type, Max will indicate an error to the user.

To review, here are the 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.

An `Atom` has the following form:

```
union word       /* union for packing any data type */
{
    long w_long;
    float w_float;
    t_symbol *w_sym;
    t_object *w_obj;
};
typedef struct atom  /* an Atom that is a typed datum */
{
    short a_type; /* from the definitions above*/
    union word a_w;
} t_atom;
```

If you declare a method to receive its arguments with `A_GIMME`, they'll be passed as an *argc, argv list* (vaguely familiar to UNIX C programmers). `argc` is the number of arguments, and `argv` points to the first of `argc` t_atoms in an array. You're also passed the t_symbol containing the message itself. If your creation function gets its arguments a la `A_GIMME`, this t_symbol is the name of your class. This is the function declaration that corresponds to an `A_GIMME` type list.

```
void myMethod (myObject *x, t_symbol *s, short
argc, t_atom *argv);
```

`x`	Your object.
`s`	The message selector as a t_symbol.
`argc`	Count of t_atoms in argv.
`argv`	Array of arguments.

To type-check the arguments yourself, you look at the `a_type` field of a `t_atom`. The possible values are `A_LONG`, `A_SYM`, and `A_FLOAT` (never `A_DEFLONG`, `A_DEFSYM`, or `A_DEFFLOAT`).

Here’s an example method that prints out its arguments that could be used as a model for type checking and processing arguments:

```
void myMethod(myObject *x, t_symbol *s, short argc, t_atom
*argv)
{
    short i;
    for (i=0; i < argc; i++) {
        switch (argv[i].a_type) {
            case A_LONG:
                post("argument %ld is a long: %ld", (long)i,
                    argv[i].a_w.w_long);
                break;
            case A_SYM:
                post("argument %ld is a symbol: name %s",
(long)i,            argv[i].a_w.w_sym->s_name);
                break;
            case A_FLOAT:
                post("argument %ld is a float: %lf",
(long)i,                argv[i].a_w.w_float);
                break;
        }
    }
}
```

The character string (`t_symbol`) argument to your method is the name of the message that invoked it. You may have several message names bound to the same method. For example:

```
addmess(myObject_doit,"doit", A_GIMME, 0); /* bound to doit
*/
addmess(myObject_doit,"finger", A_GIMME, 0);     /* bound
to finger */
```

If you want to know when your method was invoked with the doit message, check to see if the `t_symbol s` is equal to the doit symbol, using the following technique:

```
void myobject_doit(myObject *x, t_symbol *s, short argc,
t_atom *argv)
{
    if (s == gensym("doit")) {
        post("Called as a result of receiving the message
doit");
    }
}
```

Note: `gensym` is a function that returns the t_symbol associated with a character string and is described in Chapter 7.

Your shared library contains only one entry point known to the outside world when it is loaded—the starting address of your function `main`. Your `main` function is called once and only once—when the code resource is loaded. It initializes the class for your object and should look something like this:

```
void *myobject_class;    /* points to your class */
void *myobject_create(void);
void myobject_free(myObject *x);
void main(void)
{
    setup((t_messlist **)&myobject_class,
(method)myobject_create,
        (method)myobject_free, (short)sizeof(myObject), 0L,
0);
    /* allocates class memory and sets up class */

    /* add messages here using addmess, addint, addbang,
etc. */
    /* copy any resources here using rescopy */
}
```

Routines for Defining Your Class

This section describes some of the functions you'll use within the initialization (main) routine.

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.

rescopy

Use `rescopy` to copy any resources from your external object's code resource file you want to use after your initialization routine is finished.

```
void rescopy (OSType resType, short resID)
```

`resType`	The four character resource type of the resource you wish to copy.
`resID`	The ID of the resource you wish to copy.

The file that contains your code resource along with any other resources is closed after initialization time. That means that if you wish to use any Macintosh resources such as dialogs or menus inside your object's methods, you'll need to copy them out of your code resource file. `rescopy` is a function that copies a resource you specify from your original file to a temporary resource file, called "Max Temp." This file is placed in the Temporary Items folder. If the computer crashes while Max is running, the file is placed in the trash.

`rescopy` is intended to be used only at initialization time. It should work even when running Max on locked media. After a resource has been copied, you can access it like you would any resource. `rescopy` returns 0 if successful, otherwise an error message will appear in the Max window and `rescopy` will return -1. The name of the resource, if there is one, is preserved when the resource is copied.

In order to make this work when compiling externals on Windows XP, you will need to use the `FOUR_CHAR_CODE` macro with the resource type. (This macro is defined to do nothing on the Macintosh.):

```
rescopy(FOUR_CHAR_CODE('STR#'), 7474);
```

We advise against using Macintosh resource files, when and where possible, since they are a Macintosh OS9 legacy and not always easily dealt with on other platforms.

resnamecopy

Use `resnamecopy` to copy a resource by name from your external object's code resource file that you want to use after your initialization routine is finished.

```
void resnamecopy (OSType resType, char *name)
```

`resType`	The four character resource type of the resource you wish to copy.
`name`	A C string naming the resource you wish to copy.

`resnamecopy` functions identically to `rescopy` except that it copies a resource specified by name rather than ID. The ID of the resource is preserved when the resource if copied.

As with rescopy, we advise against using Macintosh resource files when possible.

Reserved Resources

Resources have to be numbered in such a way as to avoid conflicts with Max's own resources and those of other external objects. If you're curious about Max's resource IDs for a given type, just look at the Max application file in ResEdit.

Here are reserved ranges in Max for some of most common resource types.

ALRT	1000-1099, 1300-1399
CNTL	1000-1099
DITL	257, 1000-1099, 1300-1399
MENU	128-255
DLOG	257, 1000-1099, 1300-1399
STR#	1000-1199, 3000-3799, 4000-4099, 5000-5099, 7000-7099
PICT	344, 501-509, 888, 4200-4299, 8800-8899, 14914

In addition, the externals that come with Max use IDs in the range from 3000-3700 for resources such as STR#, DITL, and DLOG. You would do well to avoid this range.

alias

Use the `alias` function to allow users to refer to your object by a name other than that of your shared library.

```
void alias (char *name)
```

`name`	An alternative name for the user to use to make an object of your class.

This function allows users to refer to your object by a name other than the name of the shared library. This might come in handy if you're writing an external object that could have a number of possible manifestations, such as a shape drawing object that could create both ovals or rectangles. If you call `setup` with a type list of `A_GIMME` (this is explained below) your object creation function will be able to find out the name the user typed to create the object.

However, it's not quite that simple! If the user wants to load your file dynamically when typing the name of your object, the filename has to be the same as the name of the object. Thus, if you use `alias`, you should inform the user that your object needs to be loaded at startup (or by choosing Install… from the File menu), otherwise the aliased names will result in “no such object” errors when they are first referenced.

This problem can be alleviated somewhat using the alias feature of the Mac OS, since Max can resolve file aliases to external objects. You can then create *file* aliases that correspond to the names you’ve provided with the Max alias function. If you have an external object henry with an alias name hank, select the Max external file named henry and choose Make Alias from the File menu in the Finder, then change the name from henry alias to hank.

class_setname

Use `class_setname` to associate you object’s name with it’s filename on disk.

```
void class_setname (char *obname, char *filename);
```

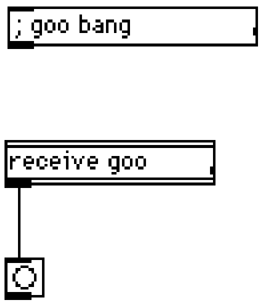
`obname`	A character string with the name of your object class as it appears in Max.
`filename`	A character string with the name of your external’s file as it appears on disk.

Because your object may have a differnet filename on disk than it would when typed into an object box, you may need to use `class_setname` to associate the object name with the filename. An example would be an obejct which contains non-alphanumeric characters that may be illegal to use in filenames on some operating systems (such as “**/~**”, whose filename is “**div~**”).

Chapter 5: Messages

Your object will do its work by responding to *messages*. Normally, objects receive numbers (int and float messages) in their inlets, process them, and send other messages out their outlets. However, arbitrary messages that begin with a character string can be sent to an object with a Max *message box* object.

The Max message box can also send messages to objects that are not directly connected to it. A Symbol after a semicolon will be the name of a receiver of the rest of the message. Any object that binds itself to this Symbol (see the explanation of binding in the description of the Symbol data structure below) will then receive the message. For example, the patch below shows the receive goo object can receive a bang message (and light up the button) without being directly connected to the message box that's sending it. When a receive object is created, it connects itself to its Symbol argument (in this case goo).



Basic Behavior

Normally, the "defining" thing your object does with a number should be in the method that responds to an integer in the left inlet (the int message). A common convention is that when a Max object receives a bang message , it repeats the action performed when an integer in the left inlet was received, using the most recently received value. If this makes sense for your object, you will want to store the value received in an int message inside your object.

You'll get a different message when a number is sent to your object through an inlet other than the leftmost one. The leftmost inlet gives you an int message whereas other inlets give you in1, in2, etc. (in1 is associated with the inlet next to the leftmost one, and the in-number increases as you move to the right, assuming you've done the proper set up work detailed below).

In most cases, you'll want the leftmost inlet to cause the object to output value or perform some kind of action, while the other inlets are used to store additional information needed when the action is taken.

As an example, consider the **noteout** object. Its leftmost inlet specifies the pitch of the note to be played, while the other inlets set the MIDI channel and velocity. The internal structure of **noteout** looks something like this:

```
typedef struct noteout {
    t_object n_ob;
    short pitch, velocity, channel;
} Noteout;
```

The **noteout** object has a method for each of its three inlets:

int	set pitch, send MIDI note message using current velocity and MIDI channel
in1	set velocity
in2	set MIDI channel

By the way, if you send a list of three integers to an object such as **noteout** that has three inlets, Max will separate the list into the three individual messages—the third number will be sent as an in2 message first, then the second number will be sent as in1, and finally the first number will be sent as int. This behavior will not occur if your object has a method that responds to a list message (see the discussion of `inlet_new` below for more information about list methods).

After writing our three integer methods, we'd also want to add a bang method, which sent a MIDI note message using the current values of pitch, velocity, and MIDI channel stored in the object.

Adding many ways to accomplish the same task adds flexibility to how your object can be used. For example, Max would be a lot harder to use if **noteout** had only one inlet and *required* a list of three numbers every time you wanted to play a note. On the other hand, allowing people to play notes by sending a list of three numbers might help someone accomplish what they want to do more easily.

Routines for Binding Messages

These routines are used in your initialization routine, after calling `setup`, to bind messages to functions you write in your class (what we refer to as methods). There are simplified routines for the most commonly used messages, as well as `addmess`, which is usable for binding any message.

addbang

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

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

`mp`	Function to be the bang method.

addfloat

Use `addfloat` to bind a function to the float message received in the leftmost inlet.

```
void addfloat (method mp);
```

`mp`	Function to be the float method.

addftx

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

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

`mp`	Function to be the float method.
`inlet`	Number of the inlet to connect with this method. 1 is the first inlet to the right of the left inlet.

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.

Standard Message Selectors

This section describes some of the standard messages Max objects send and receive besides int, bang, and float.

anything

If you want to have a method that responds to *any* message that wasn't handled by your other methods, you can bind a function to the name anything using `addmess`.

BINDING

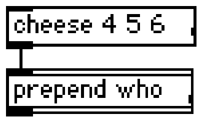
```
addmess ((method)myObject_anything, "anything",
A_GIMME, 0);
```

DECLARATION

```
myobject_anything (myObject *x, t_symbol *message,
short argc, t_atom *argv)
```

`message`	The name of the message received.
`argc`	Number of arguments (Atoms) in the `argv` array.
`argv`	Array of the message arguments.

As an example, in the following patch, the **prepend** object receives the message cheese 4 5 6. In its anything method, `argc` would be 3 and `argv` would contain t_atoms with the numbers 4, 5, and 6. `message` would be the symbol (not the t_atom) cheese.



list

This message is sent to your object when a message starts with a number. A list is an array of two or more ints, floats, or t_symbols. The first number will always be an int or float.

BINDING

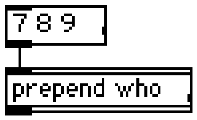
```
addmess ((method)myObject_list, "list", A_GIMME,
0);
```

DECLARATION

```
myobject_list (myObject *x, t_symbol *msg, short
argc, t_atom *argv)
```

`msg`	The name of the message received.
`argc`	Number of values (t_atoms) in the `argv` array.
`argv`	Array of t_atoms containing the list.

Note that lists may contain t_atoms of type `A_LONG`, `A_FLOAT`, and even `A_SYM`, although a t_symbol will never be the *first* element in a list. The `t_symbol` argument `msg` is unimportant and should be ignored. If the user clicked on the message box below the **prepend** object's list method would be called, and `argv` would contain t_atoms with the numbers 7, 8, and 9 (in order).



Other Standard Messages

If you're planning to use a specific word as a message, try to use a word that might already be in use in existing Max objects. These words include:

set N	Sets a value without causing output
start	Starts something, or use bang or a non-zero integer
stop	Stops something, or use 0
record	
append	
read S	Read a file (also use import), S is an optional name
write S	Write a file (also use export), S is an optional name

next	Output the next value, go to the next thing
prev	Output the previous value, go the previous thing
goto N	Set the next or prev counters to N
size N	Set size to N
zero	Set to all zeros
clear	Delete, erase, set to zero. Optionally, an argument might specify what should be cleared, while clear with no argument should clear (or zero) everything.
length	Output your length

Messages from Max

Here are some predefined messages you may want to implement.

enable and disable

These messages are sent when the user enables or disables MIDI using **pcontrol** (formerly, on Mac OS9 versions of Max this could be done by clicking on the now-defunct MIDI icon in a Patcher window title bar).

BINDING

```
addmess ((method)myObject_enable, "enable", 0);
```

DECLARATION

```
myobject_enable (myObject *x);myobject_disable
(myObject *x);
```

When MIDI is disabled in a patcher, the disable message is sent to all the objects in the patcher window. Of existing Max objects, only MIDI objects (i.e. **notein**, **noteout**) respond to this message, but if your object communicates directly with the outside world in a manner analogous to MIDI, you might consider disabling communication in response to a disable message and re-enabling it in response to an enable message. Always create your object in an enabled state because you'll never receive an initial enable message.

info

The info message is sent to your object if it is selected in an unlocked Patcher window and the user chooses Get Info… from the Max menu.

BINDING

```
addmess (myObject_info, "info", A_CANT, 0);
```

DECLARATION

```
myobject_info (myObject *x, t_patcher *parent,
t_box *container)
```

`parent`	The patcher that contains your object.
`container`	The box that contains your object (see Chapter 11 for more information about boxes).

This might be a time to put up a dialog box to set or display your object's parameters. You might also take this opportunity to put up an About box to brag about how great you are for being able to write a Max external object. Also, the info message can be used to display an Inspector patch (described in chapter 11).

The definitions of `t_patcher` and `t_box` are in *ext_user.h*. It is likely that you will not need to access your Patcher or your Box in this method, in which case you can declare your info method as:

```
void myobject_info (myObject *x, void *p, void *b);
```

preset

This message is sent by the **preset** object to request that your object provide its current state.

BINDING

```
addmess ((method)myObject_preset, "preset", A_CANT,
0);
```

DECLARATION

```
void myobject_preset (myObject *x);
```

You respond to a preset message by supplying a message that will restore your object to its current state. This may be done with a set or int message, or something more complex if your object has a lot of different data that can be changed. This will allow your object to work with the built-in **preset** object.

The functions `preset_int`, `preset_set`, and `preset_store` are useful in writing your preset method. See the section on Presets in Chapter 9 for descriptions of these routines.

loadbang

When a Patcher window is loaded from a file, each object it contains can receive the loadbang message.

BINDING

```
addmess ((method)myObject_loadbang, "loadbang",
A_CANT, 0);
```

DECLARATION

```
void myobject_loadbang (myObject *x);
```

There is a built-in Max object called **loadbang** that sends out a bang when a Patcher window is loaded. Your object can respond to the loadbang message in any way that it wants. Before implementing a method that responds to loadbang, keep in mind that the user can connect your object to the outlet of a **loadbang** object to perform initialization if necessary. Note that you do not get the loadbang message when the user creates a new instance of your object in the

Patcher window, only when a Max file containing your object is loaded from disk.

assist

This message is sent to your object when the user has positioned the cursor over one of your object's inlets or outlets and the Assistance area of the Patcher window is visible.

BINDING

```
addmess ((method)myObject_assist, "assist", A_CANT,
0);
```

DECLARATION

```
void myobject_assist (myObject *x, void *box, long
msg, long arg, char *dstString);
```

`box`	The box that contains your object (see Chapter 11 for more information about boxes). This argument is almost never used in responding to the assist message.
`msg`	One of two values, `ASSIST_INLET` (1) or `ASSIST_OUTLET` (2), indicating whether you're describing an inlet or an outlet.
`arg`	The inlet or outlet number, starting at 0 for the leftmost inlet or outlet.
dstString	Where you should copy a C string with the Assistance information for this inlet or outlet.

To respond to the assist message, you should tell the user about the function of the inlet or outlet in 60 characters or less. See example Assistance messages from existing Max objects. You can even get fancy and refer to specific arguments or the state of the instance, although you aren't sent new assist messages when your state changes.

Although you can use the function `assist_string` in conjunction with an internationalizable STR# resource to do all the work necessary to respond to this message, we recommend the old-fashioned manual way of doing things, for ease of cross-platform compatibility, as in the following example.

```
void myobject_assist (myObject *x, void *b, long msg, long
arg,
                     char *dst)
{
   if (msg==ASSIST_INLET) {
       switch (arg) {
           case 0:
               strcpy(dst,"Start Clock Ticking");
               break;
           case 1:
               strcpy(dst,"Set Clock Speed in
Milliseconds");
               break;
           case 2:
               strcpy(dst,"Set Clock Erratic Factor (0-
99)");
               break;
       }
   } else if (msg==ASSIST_OUTLET) {
       strcpy(dst,"Clock Ticks");
   }
}
```

save

Having a save method permits you to save more of the state of your non user-interface object than was typed into an object box by the user.

BINDING

```
addmess ((method)myObject_save, "save", A_CANT, 0);
```

DECLARATION

```
void myobject_save (myObject *x, Binbuf *dest);
```

`dest`	The destination for the message you create to restore your object from a file.

The details of implementing this method will be provided in the discussion of `binbuf_vinsert` in Chapter 7. Essentially, the idea is to format a message that can be sent to the **new** object to recreate your object with its current parameters, then use a function to copy this message into the data buffer `dest`. Some save methods, such as the one used by the **coll** object to save its current data, can be more elaborate and often involve messages that are sent to the newly created instance of an object that serve to restore its internal state.

Note: If your object implements a save method, it might wish to mark its owning patcher window as having unsaved data when the user changes its internal state. See the routine `patcher_dirty` in Chapter 11 for information about how to do this.

Chapter 6: Writing the Instance Creation Function

Your instance creation function is called when a new copy of your object needs to be made, either as the result of a file being read in or the user typing its name into an object box (or if it's a user interface object, choosing it from the Patcher window palette).

As with all functions you write that will be called by Max, the arguments to your object's instance creation function will be based on an argument type list you specified. Unlike all other methods, you specify the types for the instance creation function's arguments with the call to `setup` at initialization time, rather than with `addmess`.

The instance creation function's arguments are identical to those described in the Data Types and Argument Lists chapter with one exception. Since the message that's being responded to was not sent to an object of your class, but to the special **new** object, the first argument will not be a pointer to an object of your class. Instead of passing a pointer to the new object, which would have served no purpose, Max simply skips the first argument altogether. For example, if your creation function's argument type list were…

```
A_DEFLONG, 0
```

your creation function should be declared as

```
void *myObject_new (long arg);
```

The task of your instance creation function is to make an instance of your class. The creation function should return a pointer to the created object or 0 if there was a problem.

The first thing you will typically do is to call `newobject` to allocate memory for an instance of your class and do system-level initialization needed by all Max objects. Next, you'll perform additional initialization of your object's fields. For example, you might want to add an Outlet, a Clock (so the object can schedule itself), or a Qelem (if the object will draw anything or perform any action that cannot take place at interrupt level). Some fields could be assigned values based on the arguments your object creation function received. For example, when you type…

…the + object's creation function takes its argument 20 and stores it inside the newly created object.

Note: If you want to access your object's Patcher or its `t_box` structure in the Patcher, you must grab a reference to it in your object creation function. The technique for doing this is shown in the description of the function `patcher_dirty` in Chapter 11.

Inlets and Outlets

Inlets and *Outlets* are the way your object normally communicates with other objects. They are structures that keep track of connections between objects and facilitate sending messages "through" these connections.

When your object is created, you're usually given an Inlet. It is referenced in the `t_object` structure that should be the first field of your object. Notice that some objects don't show inlets. This is because they set a special flag in their class field that says, "Don't give me an inlet." You can set this flag yourself immediately before or after calling `newobject` in your instance creation function.

```
void *myclass;    /* initialized by call to setup */
void *myobject_new (void)
{
       myObject *x;
       x = newobject(myclass);
       class_noinlet(myclass);
    /* additional initialization */
       return (x);
}
```

Routines for Instance Creation

These functions cover making a new instance of your class and giving it inlets and outlets.

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.

Routines for Creating Inlets

There are several functions for creating additional inlets that are used in your object's creation function. You don't need to ever touch an inlet once it is created, so these functions do not return pointers to the created inlets. The main purpose of an inlet is just to exist, so that outlets can sink their teeth into them. Communication between objects in Max is driven entirely by actions performed with outlets. Note that inlets do not need to be freed by your object in its free method; this is taken care of for you.

intin

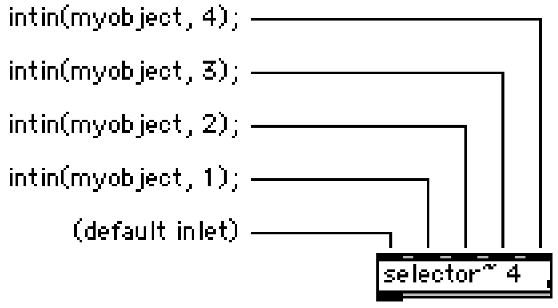
Use intin to create an inlet typed to receive only integers.

```
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:



floatin

Use `floatin` to create an inlet typed to receive only floats.

```
void floatin (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.

This function creates a floating-point inlet. It’s analogous to `intin` for floating point numbers.

inlet_new

Use `inlet_new` to create an inlet that can receive a specific message or any message.

```
void inlet_new (void *object, char *msg)
```

`object`	Your object.
`msg`	Character string of the message, or 0L to receive any message.

`inlet_new` ceates a general purpose inlet. You can use it in circumstances where you would like special messages to be received in inlets other than the leftmost one.

To create an inlet that receives a particular message, pass the message's character string. For example, to create an inlet that receives only bang messages, do the following…

```
inlet_new (myObject,"bang");
```

To create an inlet that can receive any message, pass `0` for `msg`…

```
inlet_new (myObject,0);
```

Proxies are an alternative method for general-purpose inlets that have a number of advantages. If you create multiple inlets as shown above, there would be no way to figure out which inlet received a message. See the discussion in the Using Proxies section below.

inlet_4

Use `inlet_4` to make an inlet that allows control over the translation of incoming messages.

```
void inlet_4 (void *owner, void *dst, t_symbol *in,
t_symbol *out);
```

`owner`	Your object.
`dst`	Object that will receive the message, or 0 if the inlet is "shut off." By convention, `dst` will be your object.
`in`	The incoming message to match.
`out`	The message produced by the inlet. This will be the message that will be matched to one of your object's methods, or the `t_symbol` received as the second argument to your object's anything method.

`inlet_4` is the most general inlet creation function. It allows the specification of your own translations of incoming messages. Whereas you should pass a pointer to your object for `owner`, you can specify another object to receive the translated message with the `dest` parameter. For example, you can specify one of your object's outlets.

The dst parameter can be changed (or set to 0) dynamically with the inlet_to function described below, but in order to do this, you must store the Inlet object inlet_4 returns inside your object. The **gate** object uses this technique to assign its inlet to one of several outlets, or no outlet at all.

As an example, here is how intin(x,1) would be implemented using inlet_4:

```
inlet_4 (myObject, myObject, gensym("int"),
gensym("in1"));
```

inlet_to

Use inlet_to to change the destination of an incoming message matched by an inlet.

```
void inlet_to (Inlet *in, t_object *newdest);
```

in	The inlet to change.
newdest	An object to be the new received of the messages matched by the inlet (and sent to the inlet's owner, as specified by inlet_4). If newdest is 0, the inlet is shut off and no object will receive its messages.

Note: This routine is seldom used in the initialization function, but it's described here because it's the only routine that affects inlets.

Routines for Creating Outlets

Your object is not created with any default outlets; routines must be used to create them. As with inlets, outlets can be typed with integers, floating-point numbers, or even specific messages. All the outlet creation functions listed below return a pointer to the created outlet and expect a pointer to your object as their first argument.

Since you need to reference outlets explicitly, it makes sense to store a pointer to them inside your object. The leftmost outlet can be accessed as…

```
myObject->m_ob.o_outlet
```

… assuming `m_ob` is a `t_object` that is the first field of your object.

Note that outlets do not need to be freed by your object in its free method; this is taken care of for you.

bangout

Use `bangout` to create an outlet that will always send the bang message.

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

`owner`	Your object.

You can send a bang message out a general purpose outlet, but creating an outlet using `bangout` allows Max to type-check the connection a user might make and refuse to connect the outlet to any object that cannot receive a bang message. `bangout` returns the created outlet.

floatout

Use `floatout` to create an outlet that will always send the float message.

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

`owner`	Your object.

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);
```

listout

Use `listout` to create an outlet that will always send the list message.

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

`owner`	Your object.

outlet_new

Use `outlet_new` to create an outlet that can send a specific non-standard message, or any message.

```
Outlet *outlet_new (void *owner, char *msgType);
```

`owner`	Your object.
`msgType`	C string specifying the message that will be sent out this outlet, or 0 to indicate the outlet will be used to send various messages.

If you will be sending many data types and messages through an outlet, use the `outlet_new` function and pass `0` for `msgType`. This creates the most basic type of outlet. The advantage of this kind of outlet's flexibility is balanced by the fact that Max must perform a message-lookup in real-time for every message sent through it, rather than when a patch is being constructed, as is true for other types of outlets. Patchers execute faster when outlets are typed, since the message lookup can be done before the program executes.

Here's an example of the use of `outlet_new`.

```
void *outpointer;
outpointer = outlet_new (mynewobject,0L);   /* a general
outlet */
```

Using Proxies

Recall that the mechanism Max objects use to locate the inlet that received a message is the translation of messages to other messages, such as int changing to in1 for the inlet immediately to the right of the leftmost inlet when you create an inlet with `intin`. This mechanism restricts the types of messages that can be received in inlets other than the leftmost one. Indeed, the leftmost "inlet" isn't really an inlet at all, but rather a direct reference to your object.

There are some situations where you may want to be able to receive all messages in your inlets, and then be able to determine the inlet where message arrived. For example, consider a "multi-track recorder" object that wanted to use each inlet as an independent track. Rather than having to send messages to the leftmost inlet such as record 3 to put track 3 into record, you could send the record message directly into the third inlet. In addition, the recorder could "record" any kind of message that might arrive at each inlet, not just integers.

This behavior can be achieved by using Proxy objects. Proxies are "intermediary objects" that intercept messages arriving at an inlet before your object sees them. Then, after storing the number of the receiving inlet, the Proxy sends the message on to you, where you can check this inlet number and take appropriate action. You create a Proxy object with `proxy_new`, but unlike inlets and outlets, you must explicitly get rid of a Proxy (using `freeobject`) in your object's free function.

Note: You cannot mix regular inlets and Proxies together in the same object.

Additionally, because of the new threading model in Max 4.3 and later and the ability of the low priority thread to interrupt the high priority thread, previous assumptions about proxies no longer hold true in all situations — i.e. the inlet number set in the "`id`" value will not necessarily reflect the correct inlet. When creating proxies, you should still pass in a pointer to an integer for the "`stuffLoc`" argument (see `proxy_new` below), but should not make use of it to detect which inlet in which a message was received. Instead you should use the proxy_getinlet function, which is threadsafe. There are still potential problems with feedback loops and the leftmost inlet which will are notresolvable at this time. If users complain about bizarre behaviors resulting from such a feedback loop please recommend they use delay or deferlow to prevent this unhandled reentrancy.

The following is a code example in which a Proxy is used to receive messages in three different inlets. Included is a sample bang method that prints out the inlet number where the bang message arrived.

Here is how we declare our object, making space for all our Proxy objects plus a long where the inlet number will be stored by the Proxy

```
typedef struct _myobject {
    t_object m_ob;
    void *m_proxy[2]; /* 3 inlets requires 2 proxies */
    long m_inletNumber;  /* where proxy will put inlet
number */
} myObject;
```

Here is the object creation function:

```
void *myObject_new (void)
{
    myObject *x;
    x = (myObject *)newobject(class);
    /* create proxy objects from right to left */
    x->m_proxy[1] = proxy_new(x,2,&x->m_inletNumber);
    x->m_proxy[0] = proxy_new(x,1,&x->m_inletNumber);
    return (x);
}
```

Now here is the method written in response to a bang message.

```
void myObject_bang (myObject *x;)
{
    post("message arrived at inlet %ld",
proxy_getinlet(x));
}
```

proxy_new

Use `proxy_new` to create a new Proxy object.

```
Proxy *proxy_new (t_object *owner, long id, long
*stuffLoc);
```

`owner`	Your object.
`id`	A non-zero number to be written into your object when a message is received in this particular Proxy. Normally, `id` will be the inlet "number" analogous to in1, in2 etc.
`stuffLoc`	A pointer to a location where the `id` value will be written.

This routine creates a new Proxy object (that includes an inlet). It allows you to identify messages based on an `id` value stored in the location specified by `stuffLoc`. You should store the pointer returned by `proxy_new` because you'll need to free all Proxies in your object's free function.

After your method has finished, Proxy sets the `stuffLoc` location back to 0, since it never sees messages coming in an object's leftmost inlet. You'll know you received a message in the leftmost inlet if the contents of `stuffLoc` is 0. As of Max 4.3, `stuffLoc` is not always guaranteed to be a correct indicator of the inlet in which a message was received. Use `proxy_getinlet` (below) to determine the inlet number.

proxy_getinlet

Use `proxy_getinlet` to get the inlet number in which a message was received.

```
long proxy_getinlet (t_object *owner);
```

`owner`	Your object.

This routine returns the inlet in which a message was received. Note that the "owner" argument should point to your external object's instance, not a proxy object.

The following macro may be useful for developers still building externs for Mac OS9. It makes the assumption that your external is "weak linked" to MaxLib, and tests the `proxy_getinlet` function to determine if it is present (not the case on Mac OS9).

```
#define PROXY_GETTHEINLET(x,id) (proxy_getinlet?
proxy_getinlet(x) : (id))
```

For more information on "weak linking" please consult the CodeWarrior documentation.

Chapter 7: Elements of Methods

This section documents four important structures you'll be using when writing your external object's methods. These are Outlets, Binbufs, Qelems, and Clocks. In addition, important utility routines you'll use to interface to Max are documented.

Routines for Using Outlets

These functions are used to send data to other objects. You don't need to access the fields of an Outlet data structure. All functions that send data out an outlet return 0 if a stack overflow occurred while sending the data. If you are performing repeated calls to an outlet function, you should stop if you see a 0 result returned. For example:

```
for (i=0; i < count; i++)
    if (!outlet_int(myObject, (long)i))
        break;
```

A non-zero result indicates that an error has *not* occurred.

outlet_bang

Use `outlet_bang` to send a bang message out an outlet.

```
void *outlet_bang (Outlet *theOutlet);
```

`theOutlet`	Outlet that will send the message.

outlet_float

Use `outlet_float` to send a float message out an outlet.

```
void *outlet_float (Outlet *theOutlet, double f);
```

`theOutlet`	Outlet that will send the message.
`f`	Float value to send.

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.

outlet_list

Use `outlet_list` to send a list message out an outlet.

```
void *outlet_list (Outlet *theOutlet, t_symbol
*msg, short argc, t_atom *argv);
```

`theOutlet`	Outlet that will send the message.
`msg`	Should be 0L, but can be the list Symbol.
`argc`	Number of elements in the list in argv.
`argv`	Atoms constituting the list.

`outlet_list` sends the list specified by `argv` and `argc` out the specified outlet. The outlet must have been created with `listout` or `outlet_new` in your object creation function (see above). You create the list as an array of Atoms, but the first item in the list *must* be an integer or float.

Here's an example of sending a list of three numbers.

```
t_atom myList[3];
long theNumbers[3];
short i;
theNumbers[0] = 23;
theNumbers[1] = 12;
theNumbers[2] = 5;
for (i=0; i < 3; i++) {
    SETLONG(myList+i,theNumbers[i]);   /* macro for setting
a t_atom */
}

outlet_list(myOutlet,0L,3,&myList);
```

It's not a good idea to pass large lists to `outlet_list` that are comprised of local (automatic) variables. If the list is small, as in the above example, there's no problem. If your object will regularly send lists, it might make sense to keep an array of t_atoms inside your object's data structure.

outlet_anything

Use `outlet_anything` to send any message out an outlet.

```
void *outlet_anything (Outlet *theOutlet, t_symbol
*msg, short argc, t_atom *argv);
```

`theOutlet`	Outlet that will send the message.
`msg`	The message selector t_symbol.
`argc`	Number of elements in the argument list in argv.
`argv`	t_atoms constituting the message arguments.

This function lets you send an arbitrary message out an outlet. Here are a couple of examples of its use.

First, here's a hard way to send the bang message (see `outlet_bang` above for an easier way):

```
outlet_anything(myOutlet, gensym("bang"), 0, NIL);
```

And here's an even harder way to send a single integer (instead of using `outlet_int`).

```
t_atom myNumber;
myNumber.a_type = A_LONG;   /* assign a type to the Atom */
myNumber.a_w.w_long = 432;  /* assign a value */
/* alternatively, you can use the macro SETLONG for the
    two lines above */

outlet_anything(myOutlet, gensym("int"), 1, &myNumber);
```

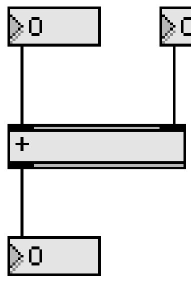
Notice that `outlet_anything` expects the message argument as a `t_symbol`, so you must use `gensym` (which transforms a C string into a `t_symbol`) on a character string. If you'll be sending the same message a lot, you might call `gensym` on the message string at initialization time and store the result in a global variable to save the (significant) overhead of calling `gensym` every time you want to send a message. Also, do not send lists using `outlet_anything` with list as the selector argument. Use the `outlet_list` function instead.

Binbufs and the Max File Format

By choosing Open As Text… from the File menu, you can open a Max binary file as a text file. Fascinating, but what does it mean? Well, one thing you can use it for is changing Max binary files without opening them into Patcher windows. For another, it provides a window into the mechanism Max uses to save messages, called Binbufs (short for *bin*ary *buf*fer).

Binbufs are an "Atomized" counterpart of the Max text file. When you copy or duplicate part of a patch, it's turned into a Binbuf. The Binbuf can then be "evaluated" because it consists of Max messages.

For example, here's a line-by-line annotation of a simplified Max file that puts a + object and three number boxes into a patcher window. (The example has been simplified by eliminating font and color information from certain objects).



`max v2;`	• What version of Max file this is
`#N vpatcher 50 38 450 338;`	• Send a vpatcher message with window coords to the new object (abbreviated #N). The resulting new Patcher window is put on a "stack" so it can be referred to as #P symbol (internally, the `s_thing` field of the #P symbol is the newly created Patcher window object).
`#P number 138 67 35 0;`	• Send a number message to the Patcher, with arguments: coordinates of the number box.
`#P number 98 67 35 0;`	• Same as above.
`#P number 98 168 35 0;`	• Same as above.
`#P newex 98 127 50 0 +;`	• Send a newex message (the name for a box for making normal objects), with box coordinates. If you had typed any arguments after + they would appear after the +.
`#P connect 0 0 1 0;`	• connect messages tell the patcher to make connections between the previously defined objects. The numbers are backwards from the listed order (0 refers to the + object), so this line says, "Connect outlet 0 of object 0 to inlet 0 of object 1."
`#P connect 3 0 0 1;`	
`#P connect 2 0 0 0;`	
`#P pop;`	• Pop the current meaning of the #P symbol off the stack and restore the previous binding (in this case, nothing).

When this file is read in, it is turned into a Binbuf consisting of Atoms: symbols, numbers, semicolons, etc. This can then be evaluated as a message, where the first Symbol is the receiver, the second the message, and the additional Atoms are arguments to the message. A semicolon is used to separate messages.

Why would you want to know about Binbufs? One reason is to use them to evaluate text. Once a bunch of text has been transformed into a Binbuf, it can be "evaluated" as a Max message. For example, the pop-up **umenu** object turns its text into a Binbuf, which can be evaluated as a message to be sent out its right outlet. The Binbuf can take care of separating a text stream into Atoms for you, generating Symbols and separating numbers from text as it goes. You'll also need to know about Binbufs if you want to do anything special to save the state of your object in a Max file. This is especially true if you're writing a user interface object (normal objects have their box coordinates and typed-in arguments saved for them).

Binbuf Routines

You won't need to know about the internal structure of a Binbuf, so you can use the `void *` type to refer to one.

When writing an object with a save method or a user interface object, you will often use `binbuf_vinsert` to store your object's creation information. Additional details are furnished in Chapter 11.

binbuf_new

Use `binbuf_new` to create and initialize a Binbuf.

```
Binbuf *binbuf_new (void);
```

`binbuf_new` returns the created Binbuf if successful, 0 if not. If you've created a Binbuf, you'll need to use `freeobject` to get rid of it.

binbuf_append

Use `binbuf_append` to append t_atoms to a Binbuf without modifying them.

```
void binbuf_append (Binbuf *bin, t_symbol *msg,
short argc, t_atom *argv);
```

`bin`	Binbuf to receive the items.
`msg`	Ignored. Pass 0.
`argc`	Count of items in the argv array.
`argv`	Array of atoms to add to the Binbuf.

binbuf_insert

Use `binbuf_insert` to append a Max message to a Binbuf adding a semicolon.

```
void binbuf_insert (Binbuf *bin, t_symbol *msg,
short argc, t_atom *argv);
```

`bin`	Binbuf to receive the items.
`msg`	Ignored. Pass 0.
`argc`	Count of items in the argv array.
`argv`	Array of t_atoms to add to the Binbuf.

You'll use `binbuf_insert` instead of `binbuf_append` if you were saving your object into a Binbuf and wanted a semicolon at the end. If the message is part of a file that will later be evaluated, such as a Patcher file, the first argument `argv[0]` will be the receiver of the message and must be a Symbol. `binbuf_vinsert` (see below) is easier to use than `binbuf_insert`, since you don't have to format your data into an array of Atoms first.

`binbuf_insert` will also convert the t_symbols #1 through #9 into $1 through $9. This is used for saving patcher files that take arguments; you will probably never save these symbols as part of anything you are doing.

binbuf_vinsert

Use `binbuf_vinsert` to append a Max message to a Binbuf adding a semicolon.

```
void binbuf_vinsert (Binbuf *bin, char *fmtString,
void *items, ...);
```

`bin`	Binbuf containing the desired Atom.
`fmtString`	C string containing one or more letters corresponding to the types of each element of the message. `s` for Symbol, `l` for long, or `f` for float.
`items`	Elements of the message, passed directly to the function as Symbols, longs, or floats.

binbuf_vinsert works somewhat like a printf for Binbufs. It allows you to pass a number of arguments of different types and insert them into a Binbuf. The entire message will then be terminated with a semicolon. Only 16 items can be passed to binbuf_vinsert.

The example below shows the implementation of a normal object's save method. The save method requires that you build a message that begins with #N (the new object) , followed by the name of your object (in this case, represented by the symbol myobject), followed by any arguments your instance creation function requires. In this example, we save the values of two fields m_val1 and m_val2 defined as longs.

```
void myobject_save (myObject *x, Binbuf *dstBuf)
{
    binbuf_vinsert(dstBuf, "ssll", gensym("#N"),
gensym("myobject"),
        x->m_val1, x->m_val2);
}
```

Suppose that such an object had written this data into a file. If you opened the file as text, you would see the following:

```
#N myobject 10 20;
#P newobj 218 82 30 myobject;
```

The first line will result in a new myobject object to be created; the creation function receives the arguments 10 and 20. The second line contains the text of the object box. The newobj message to a patcher creates the object box user interface object and attaches it to the previously created myobject object. Normally, the newex message is used. This causes the object to be created using the arguments that were typed into the object box.

binbuf_eval

Use `binbuf_eval` to evaluate a Max message in a Binbuf, passing it arguments.

```
short *binbuf_eval (Binbuf *bin, short argc, t_atom
*argv, void *receiver);
```

`bin`	Binbuf containing the message.
`argc`	Count of items in the argv array.
`argv`	Array of t_atoms as the arguments to the message.
`receiver`	Receiver of the message.

binbuf_eval is an advanced function that evaluates the message in a Binbuf with arguments in `argv`, and sends it to `receiver`. Returns the result of sending the message.

binbuf_getatom

Use `binbuf_getatom` to retrieve a single Atom from a Binbuf.

```
short binbuf_getatom (Binbuf *bin, long
*typeOffset, long *stuffOffset, t_atom *result);
```

`bin`	Binbuf containing the desired t_atom.
`typeOffset`	Offset into the Binbuf's array of types. Modified to point to the next t_atom.
`stuffOffset`	Offset into the Binbuf's array of data. Modified to point to the next t_atom.
`result`	Location of a t_atom where the retrieved data will be placed.

To get the first t_atom, set both `typeOffset` and `stuffOffset` to 0. `binbuf_getatom` returns 1 if there were no t_atoms at the specified offsets, 0 if there's a legitimate t_atom returned in `result`.

Here's an example of getting all the items in a Binbuf:

```
t_atom holder;
long to, so;
to = 0;
so = 0;
while (!binbuf_getatom(x, &to, &so, &holder));
    /* do something with the t_atom */
```

binbuf_set

Use `binbuf_set` to change the entire contents of a Binbuf.

```
void binbuf_set (Binbuf *bin, t_symbol *msg, short
argc, t_atom *argv);
```

`bin`	Binbuf to receive the items.
`msg`	Ignored. Pass 0.
`argc`	Count of items in the argv array.
`argv`	Array of t_atoms to put in the Binbuf.

The previous contents of the Binbuf are destroyed.

binbuf_text

Use `binbuf_text` to convert a text handle to a Binbuf.

```
short binbuf_text (Binbuf *bin, char **srcText,
long length);
```

`bin`	Binbuf to contain the converted text. It must have already been created with `binbuf_new`. Its previous contents are destroyed.
`srcText`	Handle to the text to be converted. It need not be terminated with a 0.
`length`	Number of characters in the text.

`binbuf_text` parses the text in the handle `srcText` and converts it into binary format. Use it to evaluate a text file or text line entry into a Binbuf. If `binbuf_text` encounters an error during its operation, a non-zero result is returned, otherwise it returns 0.

Note: Commas, symbols containing a dollar sign followed by a number 1-9, and semicolons are identified by special *pseudo-type* constants for you when your text is binbuf-ized.

The following constants in the `a_type` field of Atoms returned by `binbuf_getAtom` identify the special symbols `A_SEMI` (10), `A_COMMA` (11), and `A_DOLLAR` (12).

For a t_atom of the pseudo-type `A_DOLLAR`, the `a_w.w_long` field of the t_atom contains the number after the dollar sign in the original text or symbol.

Using these pseudo-types may be helpful in separating "sentences" and "phrases" in the input language you design. For example, the pop-up **umenu** object allows users to have spaces in between words by requiring the menu items be separated by commas. It's reasonably easy, using `binbuf_getatom`, to find the commas in a Binbuf in order to determine the beginning of a new item when reading the atomized text to be displayed in the menu.

If you want to use a literal comma or semicolon in a symbol, precede it with a backslash (\) character. The backslash character can be included by using two backslashes in a row.

binbuf_totext

Use `binbuf_totext` to convert a Binbuf into a text handle.

```
short binbuf_totext (Binbuf *bin, char **dstText,
long *size);
```

`bin`	Binbuf with data to convert to text.
`dstText`	Pre-existing handle where the text will be placed. `dstText` will be resized to accomodate the text.
`size`	Where `binbuf_totext` returns the number of characters in the converted text handle.

binbuf_totext converts a Binbuf into text and places it in a handle. Backslashes are added to protect literal commas and semicolons contained in symbols. The pseudo-types are converted into commas, semicolons, or dollar-sign and number, without backslashes preceding them. `binbuf_text` can read the output of `binbuf_totext` and make the same Binbuf. If `binbuf_totext` runs out of memory during its operation, it returns a non-zero result, otherwise it returns 0.

binbuf_read

Use `binbuf_read` to read a Max format binary or text file into a Binbuf.

```
short binbuf_read (Binbuf *bin, char *filename,
short volume, short binaryFlag);
```

`bin`	Binbuf into which the file will be read. It must have already been created with `binbuf_new`. Its previous contents are destroyed.
`filename`	C string containing the name of the file to read. Partial pathnames are acceptable.
`vol`	Volume or working directory reference number of the file.
`binaryFlag`	If non-zero, indicates the file is a binary format Max document (type maxb). If 0, indicates the file is a TEXT file. An error is returned if you try to read an old format (type 1) Max binary file.

binbuf_read opens and reads a file into a Binbuf. It returns a non-zero result if an error occurred, or 0 if there was no error during its operation.

binbuf_write

Use `binbuf_write` to write a Binbuf to a Max format binary or text file.

```
short binbuf_write (Binbuf *bin, char *filename,
short volume, short binaryFlag);
```

`bin`	Binbuf to write to disk.
`filename`	C string containing the name of the file to write. Partial pathnames are acceptable. If the file already exists, it will be overwriten.
`vol`	Volume or working directory reference number of the file.
`binaryFlag`	1 specifies that an old format Max binary file should be written. 2 specifies a new format Max binary file. 0

specifies a TEXT format file. An error is returned on the PowerPC if you try to write an old format (type 1) binary file.

`binbuf_write` creates a file (if necessary) and writes a Binbuf into it. `binbuf_write` returns a non-zero result if an error occurred, or 0 if there was no error during its operation.

readatom

Use `readatom` to read a single Atom from a text buffer.

```
short readatom (char *outstr, char **text, long
*index, long size, t_atom *result);
```

`outstr`	C string of 256 characters that will receive the next text item read from the buffer.
`text`	Handle to the text buffer to be read.
`index`	Starts at 0, and is modified by readatom to point to the next item in the text buffer.
`size`	Number of characters in `text`.
`result`	Where the resulting Atom read from the text buffer is placed.

This function provides access to the low-level Max text evaluator used by binbuf_text. It is designed to operate on a handle of characters (text) and called in a loop, as in the example shown below. readatom returns non-zero if there is more text to read, and zero if it has reached the end of the text. Note that this return value has the opposite logic from that of binbuf_getatom.

```
long index;
t_atom dst;
char outstr[256];
index = 0;
while (readatom(outstr,textHandle,&index,textLength,&dst))
{
    /* do something with the resulting Atom */
}
```

An alternative to using readatom is to turn your text into a Binbuf using binbuf_text, then call binbuf_getatom in a loop.

Routines for Atombufs

The Atombuf is an alternative to the Binbuf for temporary storage of atoms. Its principal advantage is that the internal structure is publicly available so you can manipulate the atoms in place. The standard Max text objects (message box, object box, comment) use the Atombuf structure to store their text (each word of text is stored as a Symbol or a number).

The data structure of an Atombuf is as follows:

```
typedef struct atombuf {
    long a_argc;
    t_atom a_argv[1];
} t_atombuf, Atombuf;
```

The array a_argv is of variable length specified by a_argc. The size of an Atombuf x is thus sizeof(long) + x->a_argc * sizeof(t_atom).

atombuf_new

Use atombuf_new to create a new Atombuf from an array of t_atoms.

```
t_atombuf *atombuf_new (long argc, t_atom *argv);
```

argc	Number of t_atoms in the argv array. May be 0.

`argv`	Array of t_atoms. If creating an empty Atombuf, you may pass 0.

`atombuf_new` create a new Atombuf and returns a pointer to it. If 0 is returned, insufficient memory was available.

atombuf_free

Use `atombuf_free` to dispose of the memory used by an Atombuf.

```
void atombuf_free (t_atombuf *ab);
```

`ab`	Atombuf to free.

You cannot use `freeobject` on an Atombuf, since it contains no object header information.

atombuf_text

Use `atombuf_text` to convert text to t_atoms in an Atombuf.

```
void atombuf_text (t_atombuf **ab, char **buffer,
long size);
```

`ab`	Pointer to existing atombuf variable. The variable will be replaced by a new Atombuf containing the converted text.
`buffer`	Handle to the text to be converted. It need not be zero-terminated.
`size`	Number of characters in the text.

To use this routine to create a new Atombuf from the text buffer, first create a new empty `t_atombuf` with a call to `atombuf_new(0L,0L)`.

Clock Routines

Clock objects are your interface to Max's scheduler. To use the scheduler, you create a new Clock object using `clock_new` in your instance creation function. You also have to write a clock function that will be executed when the clock goes off, declared as follows:

```
void myobject_tick (myobject *x);
```

The argument `x` is determined by the `arg` argument to `clock_new`. Almost always it will be pointer to your object.

Then, in one of your methods, use `clock_delay` or clock_fdelay to schedule yourself. If you want unschedule yourself, call `clock_unset`. To find out what time it is now, use `gettime` or `clock_getftime`. More advanced clock operations are possible with the **setclock** object interface described in Chapter 9. We suggest you take advantage of the higher timing precision of the floating-point clock routines—all standard Max 4 timing objects such as **metro** use them.

When the user has Overdrive mode enabled, your clock function will execute at interrupt level.

clock_new

Use clock_new to create a new Clock object.

```
Clock *clock_new (void *arg, method clockfun);
```

`arg`	Argument that will be passed to clock function `clockfun` when it is called. This will almost always be a pointer to your object.
`clockfun`	Function to be called when the clock goes off, declared to take a single argument as shown above.

`clock_new` returns a pointer to a newly created Clock object that will run function `clockfun` passing it argument `arg` when it goes off. Normally, `clock_new` is called in your instance creation function—and it cannot be called at interrupt level. To get rid of a clock object you created, use `freeobject`.

clock_delay

Use `clock_delay` to schedule the execution of a Clock.

```
void clock_delay (Clock *cl, long interval);
```

`cl`	Clock to schedule.
`interval`	Delay, in milliseconds, before the Clock will execute.

`clock_delay` sets a clock to go off at a certain number of milliseconds from the current logical time.

clock_fdelay

Use `clock_fdelay` to schedule the execution of a Clock using a floating-point argument.

```
void clock_fdelay(Clock *c, double time);
```

c	Clock to schedule.
time	Delay, in milliseconds, before the Clock will execute.

`clock_fdelay` is the floating-point equivalent of `clock_delay`.

clock_unset

Use `clock_unset` to cancel the scheduled execution of a Clock.

```
void clock_unset (Clock *cl);
```

`cl`	Clock to cancelled.

`clock_unset` will do nothing (and not complain) if the Clock passed to it has not been set.

gettime

Use `gettime` to find out the current logical time of the scheduler in milliseconds.

```
long gettime (void);
```

clock_getftime

Use `clock_getftime` to find out the current logical time of the scheduler into a floating point argument.

```
void clock_getftime(double *time)
```

time	Returns the current time.

`clock_getftime` is the floating-point equivalent of gettime.

Using Clocks

Under normal circumstances, `gettime` or `clock_getftime` will not be necessary for scheduling purposes if you use `clock_delay` or `clock_fdelay`, but it may be useful for recording the timing of messages or events .

As an example, here's a fragment of how one might go about writing a metronome using the Max scheduler. First, here's the data structure we'll use.

```
typedef struct mymetro {
    t_object *m_obj;
    void *m_clock;
    double m_interval;
    void *m_outlet;
} t_mymetro;
```

We'll assume that the class has been initialized already. Here's the instance creation function that will allocate a new Clock.

```
void *mymetro_create (double defaultInterval)
{
    t_mymetro *x;
    x = (t_mymetro *)newobject(mymetro_class);    /*
allocate space */
    x->m_clock = clock_new(x,(method)mymetro_tick); /* make
a clock */
    x->m_interval = defaultInterval;   /* store the interval
*/
    x->m_outlet = bangout(x);   /* outlet for ticks */
    return x;  /*return the new object */
}
```

Here's the method written to respond to the bang message that starts the metronome.

```
void mymetro_bang (t_mymetro *x)
{
    clock_fdelay(x->m_clock,0.);
}
```

Here's the Clock function.

```
void mymetro_tick(t_mymetro *x)
{
    clock_fdelay(x->m_clock, x->m_interval);
    /* schedule another metronome tick */
    outlet_bang(x->m_outlet);   /*send out a bang */
}
```

You may also want to stop the metronome at some point. Here's a method written to respond to the message stop. It uses `clock_unset`.

```
void mymetro_stop (t_mymetro *x)
{
    clock_unset(x->m_clock);
}
```

In your object's free function, you should call `freeobject` on any Clocks you've created.

```
void mymetro_free (MyMetro *x)
{
    freeobject((t_object *)x->m_clock);
}
```

Systime API

The Systime API provides the means of getting the system time, instead of the scheduler time (see `gettime`, above).

systime_ticks

Use `systime_ticks` to find out the operating system's time in ticks.

```
unsigned long systime_ticks (void);
```

Returns the system time in ticks.

systime_ms

Use `systime_ms` to find out the operating system's time in milliseconds.

```
unsigned long systime_ms (void);
```

Returns the system time in milliseconds.

systime_datetime

Use `systime_datetime` to find out the operating system's date and time.

```
unsigned long systime_datetime (t_datetime *d);
```

Returns the system's date and time in a `t_datetime` data structure:

```
typedef struct _datetime
{
   unsigned long year;
   unsigned long month;
   unsigned long day;
   unsigned long hour;
   unsigned long minute;
   unsigned long second;
   unsigned long millisecond;  // (reserved for future use)
} t_datetime;
```

Qelem Routines

Your object's methods may be called at interrupt level. This happens when the user has Overdrive mode enabled and one of your methods is called, directly or indirectly, from a scheduler Clock function. This means that you cannot count on doing certain things—like drawing, asking the user what file they would like opened, or calling any Macintosh toolbox trap that allocates or purges memory—from within any method that responds to any message that could be sent directly from another Max object. The mechanism you'll use to get around this limitation is the *Qelem* (queue element) structure. Qelems also allow processor-intensive tasks to be done at a lower priority than in an interrupt. As an example, drawing on the screen, especially in color, takes a long time in comparison with a task like sending MIDI data.

A Qelem works very much like a Clock. You create a new Qelem in your creation function with `qelem_new` and store a pointer to it in your object. Then you write a queue function, very much like the clock function (it takes the same single argument that will usually be a pointer to your object) that will be called when the Qelem has been set. You set the Qelem to run its function by calling `qelem_set`.

Often you'll want to use Qelems and Clocks together. For example, suppose you want to update the display for a counter that changes 20 times a second. This can be accomplished by writing a Clock function that calls `qelem_set` and then reschedules itself for 50 milliseconds later using the technique shown in the metronome example above. This scheme works even if you call `qelem_set` faster than the computer can draw the counter, because if a Qelem is already set, `qelem_set` will not set it again. However, when drawing the counter, you'll display its *current value*, not a specific value generated in the Clock function.

Note that the Qelem-based *defer* mechanism discussed later in this chapter may be easier for lowering the priority of one-time events, such as opening a standard file dialog box in response to a read message.

If your Qelem routine sends messages using `outlet_int` or any other of the outlet functions, it needs to use the lockout mechanism described in the Interrupt Level Considerations section.

qelem_new

Use `qelem_new` to create a new Qelem.

```
Qelem *qelem_new (void *arg, method fun);
```

`arg`	Argument to be passed to function fun when the Qelem executes. Normally a pointer to your object.
`fun`	Function to execute.

Any kind of drawing or calling of Macintosh Toolbox routines that allocate or purge memory should be done in a Qelem function. You need to store the return value of `qelem_new` to pass to `qelem_set`.

Note that in order to get rid of a Qelem, do *not* call `freeobject`; use `qelem_free` instead.

qelem_set

Use `qelem_set` to cause a Qelem to execute.

```
void qelem_set (Qelem *qe);
```

`qe`	The Qelem whose function will be exeucted at the main level.

The key behavior of `qelem_set` is this: if the Qelem object has already been set, it will not be set again. (If this is not what you want, see `defer` below.) This is useful if you want to redraw the state of some data when it changes, but not in response to changes that occur faster than can be drawn. A Qelem object is unset after its queue function has been called.

qelem_unset

Use `qelem_unset` to cancel a Qelem's execution.

```
void qelem_unset (Qelem *qe);
```

`qe`	The Qelem whose execution you wish to cancel.

If the Qelem's function is set to be called, `qelem_unset` will stop it from being called. Otherwise, `qelem_unset` does nothing.

qelem_front

Use `qelem_front` to cause a Qelem to execute at a high priority.

```
void qelem_front (Qelem *qe);
```

`qe`	The Qelem whose function will be exeucted at the main level.

This function is identical to `qelem_set`, except that the Qelem's function is placed at the front of the list of routines to execute at the main event level instead of the back. Be polite and only use `qelem_front` for special time-critical applications.

qelem_free

Use `qelem_free` to get rid of a Qelem in your object's free funtion.

```
void qelem_free (Qelem *qe);
```

`qe`	The Qelem to destroy.

This function frees a previously allocated Qelem object. Use this function instead of `freeobject` to free the memory used by a Qelem.

Interrupt Level Considerations

Your object may be responding to messages at interrupt level, and there are a few guidelines you'll need to follow when writing methods so that your objects work correctly. Interrupt Level processing is enabled when the user chooses Overdrive from the Options menu. The advantages of Interrupt Level processing are the increased accuracy of timing, the ability for a Max patch to continue to operate smoothly while the user is using the menu bar or dragging a window, and the ability to prioritize more time-critical clock and serial port operations over slower screen drawing.

Older versions of Max/MSP running on Mac OS9 used a hardware interrupt model, but Max/MSP 4.2 and later for both Windows XP and Mac OSX use a software interrupt model where Overdrive and audio processinng are concerned—i.e. using individual threads for the various tasks. In this new multithreaded model it is possible for the low priority thread to interrupt the high priority scheduler or audio threads as the operating system sees fit, regardless of thread priority settings. This was not the case under OS 9, in which the low priority thread could never interrupt the scheduler interrupt or audio interrupt, and the scheduler interrupt could never interrupt the audio interrupt. If you have assumptions in your code that rely on the aforementioned OS 9 properties, you will probably need to revisit the way your code is written (it certainly forced us to do so).

The basic rules for interrupt level operations are the following:

- Use critical regions, described below, to safeguard parts of your code that needed to be executed without interruption. This is usually only necessary when doing things like manipulating memory at low priority. Calling outlet functions at interrupt level (such as when using an outlet to send a message in a Qelem) is safe to do, since outlets are now threadsafe (any place in your code that formerly made use of Max's lockout mechanism for this purpose will not need to be placed in a critical region). See the section on critical regions, below, for further information.

- Don't call operating system memory allocation routines directly in response to a typed message (such as int or list). Nor should you use other system-specific routines that move or purge memory (see your operating system's developer documentation for a list). For memory management, you may be able to use the special interrupt-safe Max routines `getbytes` and `freebytes`, and defer other types of memory allocation. Don't do anything that creates, adds to or frees a Binbuf or an Expr—these structures use Macintosh memory management calls. Note that `getbytes` has only a limited supply of memory available at interrupt level, so don't overuse it. In addition, `getbytes` can only allocate a buffer of less than 16384 bytes at interrupt level. If you need larger buffers, allocate them in advance.

- Use the memory allocation calls supplied by Max in the SYSMEM API called `sysmem_newhandle` and `sysmem_disposhandle` instead of system-specific routines such as NewHandle and DisposeHandle. Large memory allocations should be done in a low priority thread (preferably in advance). Note that using system memory calls will no longer crash your computer, but it is still not recommended, as it will adversely affect Max's timing.

- Don't draw on the screen or put up a dialog box directly in response to a message. Use a Qelem, `defer` or `defer_low`. (Note that when using `defer`, events are placed at the *front* of the low priority queue, and consequently it is likely that message sequencing will be reversed. In cases where it the message ordering of a series of deferrals is important, use `defer_low`, which places events at the back of the low priority queue.) Note also that you can call `ouchstring` (for putting up error dialog box notices) in an interrupt, and the dialog will be queued to a lower priority for you.

Interrupt Level Routines

Here are a few additional routines for dealing with Interrupt-driven processing issues.

isr

Use `isr` to determine whether your code is executing in the Max timer interrupt

```
short isr (void);
```

This function returns non-zero if you are within a Max timer interrupt, zero otherwise. Note that if your code sets up other types of interrupt-level callbacks, such as for other types of device drivers used in asynchronous mode, `isr` will return false.

defer

Use `defer` to defer execution of a function to the main level if (and only if) your function is executing at interrupt level.

```
void defer (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.

This function uses 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 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` described below *is* appropriate however, because it *always* defers.

The deferred function should be declared as follows:

```
void myobject_do (myObject *client, t_symbol *s, short argc,
t_atom *argv);
```

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.

schedule

Use `schedule` to cause a function to be executed at the timer level at some time in the future.

```
void schedule (t_object *client, method fun, long
time, t_symbol *sel, short argc, t_atom *argv);
```

`client`	First argument to the function `fun` to be executed. By convention, this is a pointer to your object.
`fun`	Function to be executed. See below for how to declare it. This function may be called at interrupt level.
`time`	The logical time that the function `fun` will be executed.
`sel`	Second argument to the function `fun`.
`argc`	Count of Atoms in `argv`; third argument to the function `fun`.
`argv`	Additional arguments to the function `fun`, or 0L if there are none.

`schedule` calls a function at some time in the future. Unlike `defer`, the function is called in the scheduling loop when logical time is equal to the specified value `when`. This means that the function could be called at interrupt level, so it should follow the usual restrictions on interrupt-level conduct. The function `fun` passed to `schedule` should be declared as follows:

```
void myobject_do (myObject *client, t_symbol *s,
short argc, t_atom *argv);
```

One use of `schedule` is as an alternative to using the lockout flag. Here is an example click method that calls `schedule` instead of `outlet_int` surrounded by `lockout_set` calls.

schedule_delay

Use `schedule_delay` to cause a function to be executed at the timer level at some time in the future specified by a delay offset.

```
void schedule (t_object *client, method fun, long
delay, t_symbol *sel, short argc, t_atom *argv);
```

`client`	First argument to the function `fun` to be executed. By convention, this is a pointer to your object.
`fun`	Function to be executed. See below for how to declare it. This function may be called at interrupt level.
`delay`	The delay from the current time before the function will be executed.
`sel`	Second argument to the function `fun`.
`argc`	Count of Atoms in `argv`; third argument to the function `fun`.
`argv`	Additional arguments to the function `fun`, or 0L if there are none.

`schedule_delay` is similar to schedule but allows you to specify the time as a delay rather than a specific logical time.

One use of `schedule` or `schedule_delay` is as an alternative to using the lockout flag. Here is an example click method that calls `schedule` instead of `outlet_int` surrounded by `lockout_set` calls.

```
void myobject_click (t_myobject *x, Point pt, short
modifiers)
{
    t_atom a[1];
    a[0].a_type = A_LONG;
    a[0].a_w.w_long = Random();
    schedule_delay(x, myobject_sched, 0 ,0, 1, a);
}
void myobject_sched (t_myobject *x, t_symbol *s, short ac,
t_atom *av)
{
    outlet_int(x->m_out,av->a_w.w_long);
}
```

Critical Regions

Under Max 4.1 and earlier there was a simple protective mechanism called "lockout" that would prevent the scheduler from interrupting the low priority thread during sensitive operations such as sending data out an outlet or modifying members of a linked list. This lockout mechanism has been deprecated, and under the Mac OS X and Windows XP versions (Max 4.2 and later) does nothing. So how do you protect thread sensitive operations? Use critical regions (also known as critical sections). However, it is very important to mention that all outlet calls are now thread safe and should never be contained inside a critical region. Otherwise, this could result in serious timing problems. For other tasks which are not thread safe, such as accessing a linked list, critical regions or some other thread protection mechanism are appropriate.

A critical region is a simple mechanism that prevents multiple threads from accessing at once code protected by the same critical region. The code fragments could be different, and in completely different modules, but as long as the critical region is the same, no two threads should call the protected code at the same time. If one thread is inside a critical region, and another thread wants to execute code protected by the same critical region, the second thread must wait for the first thread to exit the critical region. In some implementations a critical region can be set so that if it takes too long for the first thread to exit said critical region, the second thread is allowed to execute, dangerously and potentially causing crashes. This is the case for the critical regions exposed by Max and the default upper limit for a given thread to remain inside a critical region is two seconds. Despite the fact that there are two seconds of leeway provided before two threads can dangerously enter a critical region, it is important to only protect as small a portion of code as necessary with a critical region.

In Max, the `critical_enter` function is used to enter a critical region, and the `critical_exit` function is used to exit a critical region. It is important that in any function which uses critical regions, all control paths protected by the critical region, exit the critical region (watch out for goto or return statements). The `critical_enter` and `critical_exit` functions take a critical region as an argument. However, for almost all purposes, we recommend using the global critical region in which case this argument is zero. The use of multiple critical regions can cause problems such as deadlock, i.e. when thread #1 is inside critical region A waiting on critical region B, but thread #2 is inside critical region B and is waiting on critical region A. In a flexible programming environment such as Max, deadlock conditions are easier to generate than you might think. So unless you are completely sure of what you are doing, and absolutely need to make use of multiple critical regions to protect your code, we suggest you use the global critical region.

In the following example code we show how one might use critical regions to protect the traversal of a linked list, testing to find the first element whose values is equal to "val". If this code were not protected, another thread which was modifying the linked list could invalidate assumptions in the traversal code.

```
critical_enter(0);
for (p = head; p; p = p->next) {
    if (p->value == val)
    break;
}
critical_exit(0);
return p;
```

And just to illustrate how to ensure a critical region is exited when multiple control paths are protected by a critical region, here's a slight variant.

```
critical_enter(0);
for (p = head; p; p = p->next) {
    if (p->value == val) {
        critical_exit(0);
        return p;
    }
}
critical_exit(0);
return NULL;
```

For more information on multi-threaded programming, hardware interrupts, and related topics, we suggest you perform some research online or read the relevant chapters of “Modern Operating Systems” by Andrew S. Tanenbaum (Prentice Hall). At the time of writing, some relevant chapters from this book are available for download in PDF format on Prentice Hall's web site. See:

www.prenhall.com/divisions/esm/app/author_tanenbaum/custom/mo s2e/

Look under "sample sections".

critical_new

Use `critical_new` to create a new critical region.

```
void critical_new (t_critical *cr);
```

cr	A t_critical struct will be returned to you via this pointer.

Normally, you do not need to create your own critical region, because you can use Max's global critical region. Only use this function (in your object's instance creation method) if you are certain you are not able to use the global critical region.

critical_free

Use `critical_free` to free a critical region created with `critical_new`.

```
void critical_free (t_critical cr);
```

cr	The t_critical struct that will be freed.

If you created your own critical region, you will need to free it in your object's free method.

critical_enter

Use `critical_enter` to enter a critical region.

```
void critical_enter (t_critical cr);
```

cr	A pointer to a t_critical struct, or zero to uses Max's global critical region.

Typically you will want the argument to be zero to enter the global critical region, although you could pass your own critical created with `critical_new`. It is important to try to keep the amount of code in the critical region to a minimum.

critical_exit

Use `critical_exit` to leave a critical region.

```
void critical_exit (t_critical cr);
```

cr	A pointer to a t_critical struct, or zero to uses Max's global critical region.

Typically you will want the argument to be zero to exit the global critical region, although, you if you are using your own critical regions you will want to pass the same one that you previously passed to `critical_enter`.

Chapter8: Essential Max Utilities

Here are some important functions for interacting with the Max environment.

Throughout this and remaining chapters of this document you will probably notice groups of related functions beginning with the letters "sys…" Max has a number of these "Sys" APIs to take the place of what would otherwise be operating system specific tasks. These APIs include working with files, windows, memory allocation, and other OS specific tasks. Due to the Macintosh heritage of Max, there is sometimes a Macintosh leaning to the way the API is named or implemented, but these functions should work the same regardless of the operating system on which they are being used. These APIs are available in both the Macintosh OS X and Windows versions of Max, and more may be added in the future.

General Utilities

freeobject

Use `freeobject` to release the memory used by a Max object.

```
void freeobject (t_object *obj);
```

`obj`	Object to free.

`freeobject` calls an object's free function, if any, then disposes the memory used by the object itself. `freeobject` should be used on any instance of a standard Max object data structure, *with the exception of Boxes, Qelems and Atombufs*. Clocks, Binbufs, Proxies, Toolfiles, Exprs, Eds, etc. should be freed with `freeobject`.

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
```

postatom

Use `postatom` to print multiple items in the same line of text in the Max window.

```
void postatom (t_atom *item);
```

`item`	Atom to be printed. The proper formatting is performed depending on the Atom's `a_type` field.

This function prints a single Atom on a line in the Max window without a carriage return afterwards, as post does. Each Atom printed is followed by a space character. The Max **print** object uses `postatom` to print lists.

ouchstring

Use `ouchstring` to put up an error or advisory alert box on the screen.

```
void ouchstring (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`.

This function performs an sprintf on `fmtstring` and `items`, then puts up an alert box. `ouchstring` will queue the message to a lower priority level if it's called in an interrupt and there is no alert box request already pending.

An example of the use of `ouchstring` you might have seen in Max at one time or another:

The Max user-interface style suggests that error dialogs be used as seldom as possible in favor of error messages in the Max window.

sprintf

Use `sprintf` to format text strings.

```
short sprintf (char *dest, char *fmtString, void
*items...);
```

`dest`	C string where the resulting formatted text will be placed.
`fmtstring`	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`.

This provides access to the commonly used C library function sprintf used in the Max kernel so you can avoid linking it into your external code resource.

sscanf

Use `sscanf` to convert text to binary data.

```
short sscanf (char *src, char *fmtString, void
*items...);
```

`src`	C string where the text is read from.
`fmtString`	C string containing text and printf-like codes specifying the sizes of the additional arguments.
`items`	One or more addresses of data where you want the converted binary values to be placed.

This provides access to the C library function sscanf used in the Max kernel so you can avoid linking it into your external code resource.

maxversion

Use `maxversion` to determine information about the current Max environment.

```
short maxversion (void);
```

This function returns the version number of Max. In Max versions 2.1.4 and later, this number is the version number of the Max kernel application in binary-coded decimal. Thus, 2.1.4 would return 214 hex or 532 decimal. Version 3.0 returns 300 hex. Use this to check for the existence of particular function macros that are only present in more recent Max versions. Versions before 2.1.4 returned 1, except for versions 2.1.1 - 2.1.3 which returned 2. Bit 14 (counting from left) will be set if Max is running as a standalone application, so you should mask the lower 12 bits to get the version number.

assist_string

Use `assist_string` to provide information about an inlet or outlet of your object to the user.

```
void assist_string (short rsrcID, long message,
long arg, short firstin, short firstout, char
*dstString, ...);
```

`rsrcID`	ID of a 'STR#' resource containing assistance information. This resource must have been copied to Max's temporary resource file with `rescopy`.
`message`	Either `ASSIST_INLET` or `ASSIST_OUTLET` specifying whether the assistance is for an inlet or an outlet. This value is passed to your assist method as an argument.
`arg`	Inlet or outlet number to describe, beginning at 0. This value is passed to your assist method as an argument.
`firstin`	Index (1-relative) of the first string in the 'STR#' resource that describes an inlet. This is almost always 1.
`firstout`	Index of the first string in the 'STR#' resource that describes an outlet.
`dstString`	Location where the string extracted from the resource should be copied. This pointer is passed to your assist method as an argument, although you may wish to call assist string on a temporary character array and then perform additional processing on it. The result is stored as a C string.
`...`	You can pass up to 16 bytes worth of additional arguments to assist_string and they will be passed to sprintf, which uses the string copied from the resource as a format string.

This routine is useful in implementing your assist method. Here's an example. Suppose we've stored the following two strings for our object in a STR# resource ID = 4534. The stored strings are:

```
Sets the Value of the Bludgeon (Currently
%ld)Outputs a bludgeon Message
```

We can document this object in our assist method as follows:

```
void myobject_assist(MyObject *x, void *b, long msg, long
arg, char *s)
{
   assist_string(4534,msg,arg,1,2,s,x->m_bludgeon);
}
```

drawstr

Use `drawstr` to draw a C string.

```
void drawstr (char *str);
```

`str`	C string to draw at the current pen location using the current font and size.

quittask_install

Use `quittask_install` to register a function that will be called with Max exits.

```
void quittask_install(method m, void *a);
```

m	A function that will be called on Max exit.
a	Argument to be used with method m.

`quittask_install` provides a mechanism for your external to register a routine to be called prior to Max shutdown. This is useful for objects that need to provide disk-based persistance outside the standard Max storage mechanisms, or need to shut down hardware or their connection to system software and cannot do so in the termination routine of a code fragment.

quittask_remove

Use `quittask_remove` to unregister a function previously registered with `quittask_install`.

```
void quittask_remove(method m);
```

m	Function to be removed as a shutdown method.

This routineallows an object to remove any previously registered shutdown methods.

object_subpatcher

Use `object_subpatcher` to determine if an object contains any subpatchers.

```
void *object_subpatcher(t_object *theobject, long
*index, void *arg);
```

theobject	An object to query.
index	The index of the returned subpatcher. Set this to 0 on the initial call.
arg	An argument to be passed to the patcher routine. This is primarily for internal Max use.

`object_subpatcher` lists any Patchers that are “contained” by an object. For instance, the patcher and bpatcher objects contain Patchers, as does the MSP object **poly~**..

The index argument is set during the call to object_subpatcher. If a non-zero result is returned (meaning that a subpatcher was found), the index argument will be set to the index number of the returned pointer to a Patcher. To find all the Patchers associated with an object, call object_subpatcher until it returns 0.

Memory Management Routines

Here are some functions provided for memory allocation that work within the real-time Max environment. Max maintains a small amount of memory that can be allocated at interrupt level, because you can’t use standard Macintosh calls to allocate memory at interrupt level because the Memory Manager is not re-entrant. If the amount of memory allocated at interrupt level is reduced by more than 50%, the supply is replenished when Max returns to the main event level.

The newhandle and disposhandle routines can be used in place of the Macintosh traps NewHandle and DisposeHandle. They work with the Max scheme for preventing out of memory errors by failing if the memory allocated would dip into an emergency reserve left for operating system use. They also fail if called at interrupt level.

newhandle

Use `newhandle` to allocate relocatable memory.

```
char **newhandle (long size);
```

size	The size to allocate in bytes.

This function is a substitute for NewHandle that performs some error checking and won’t call NewHandle if is called at interrupt level.

disposhandle

Use `disposhandle` to free the memory used by a handle you no longer need.

```
void disposhandle (char **handle);
```

`handle`	Macintosh Handle to be disposed.

This function calls the Macintosh trap DisposeHandle unless it is called at interrupt level in which it returns a NIL value.

growhandle

Use `growhandle` to change the size of a handle.

```
void growhandle (char **handle, long size);
```

`handle`	Macintosh Handle whose size is to be changed.
`size`	The new size in bytes.

This function is a substitute for SetHandleSize with some error checking that refuses to work at interrupt level.

getbytes

Use `getbytes` to allocate small amounts of non-relocatable memory.

```
void *getbytes (short size);
```

`size`	The size to allocate in bytes.

`getbytes` is a substitute for NewPtr that takes the memory from a pool maintained by Max. It can be called for a request up to 32767 bytes. If size is greater than 16384 bytes, getbytes calls the Macintosh routine NewPtr. If this size request is made at interrupt level, `getbytes` returns 0 and prints the following message in the Max window.

- `check failed: t_newptr in overdrive`

The memory pool used by `getbytes` is limited to 256K for any particular interrupt in version 4, 128K in version 3 and 32K in previous versions. The same "t_newptr in overdrive" may appear when you try to allocate too many small chunks of memory at interrupt level, since `getbytes` uses NewPtr to replenish its non-relocatable memory pool. Always free memory allocated with `getbytes` with `freebytes`, and note that `freebytes` requires that you pass it the size of the block allocated with `getbytes`.

freebytes

Use `freebytes` to free memory allocated with `getbytes`.

```
void freebytes (void *ptr, short size);
```

`ptr`	A pointer to the block of memory previously allocated that you want to free.
`size`	The size of this block in bytes.

Like `getbytes`, `freebytes` may be called at interrupt level for blocks up to 16384 bytes.

getbytes16

Use `getbytes16` to allocate small amounts of non-relocatable memory that is aligned on a 16-byte boundary for use with vector optimization.

```
void *getbytes16 (short size);
```

`size`	The size to allocate in bytes.

`getbytes16` is identical to getbytes except that it returns memory that is aligned to a 16-byte boundary. This allows you to allocate storage for vector-optimized memory at interrupt level. Note that any memory allocated with `getbytes16` must be freed with `freebytes16`, not `freebytes`.

freebytes16

Use `freebytes16` to free memory allocated with `getbytes16`.

```
void freebytes16 (void *ptr, short size);
```

`ptr`	A pointer to the block of memory previously allocated with `getbytes16` that you want to free.
`size`	The size of this block in bytes.

Note that `freebytes16` will cause memory corruption if you pass it memory that was allocated with `getbytes`. Use it only with memory allocated with `getbytes16`.

Sysmem API

The Sysmem API provides a number of utilities for allocating and managing memory. It is relatively similar to some of the Macintosh Memory Manager API, and not too different from Standard C library memory functions. It is not safe to mix these routines with other memory routines (e.g. don't use `malloc` to allocate a pointer, and `sysmem_freeptr` to free it).

sysmem_newptr

Use `sysmem_newptr` to allocate memory.

```
t_ptr sysmem_newptr(long size);
```

`size`	The amount of memory in bytes that will be allocated.

This function is similar to `NewPtr` or `malloc`. It allocates a pointer of a given number of bytes and returns a pointer to the memory allocated.

sysmem_newptrclear

Use `sysmem_newptrclear` to allocate memory and set it to zero.

```
t_ptr sysmem_newptrclear(long size);
```

`size`	The amount of memory in bytes that will be allocated.

This function is similar to `NewPtrClear` or `calloc`. It allocates a pointer of a given number of bytes, zeroing all memory, and returns a pointer to the memory allocated.

sysmem_resizeptr

Use `sysmem_resizeptr` to resize an existing pointer.

```
t_ptr sysmem_resizeptr(void *ptr, long newsize);
```

`ptr`	The pointer to the memory that will be resized.
`newsize`	The new size of the pointer in bytes.

This function is similar to `realloc`. It resizes an existing pointer and returns a new pointer to the resized memory.

sysmem_ptrsize

Use `sysmem_ptrsize` to find the size of a pointer.

```
long sysmem_ptrsize(void *ptr);
```

`ptr`	The pointer whose size will be queried.

This function is similar to `_msize`. It returns the number of bytes allocated to the pointer specified.

sysmem_freeptr

Use `sysmem_freeptr` to free memory allocated with `sysmem_newptr`.

```
void sysmem_freeptr(void *ptr);
```

`ptr`	The pointer whose memory will be freed.

This function is similar to `DisposePtr` or `free`. It frees the memory that had been allocated to the given pointer.

sysmem_copyptr

Use `sysmem_copyptr` to write part of a file to disk.

```
void sysmem_copyptr (const void *src, void *dst,
long bytes);
```

`src`	A pointer to the memory whose bytes will be copied.
`dst`	A pointer to the memory where the data will be copied.
`bytes`	The size in bytes of the data to be copied

This function is similar to `BlockMove` or `memcpy`. It copies the contents of the memory from the source to the destination pointer.

sysmem_newhandle

Use `sysmem_newhandle` to allocate a handle (a pointer to a pointer).

```
t_handle sysmem_newhandle(long size);
```

`size`	The size of the handle in bytes that will be allocated.

This function is similar to `NewHandle`. It allocates a handle of a given number of bytes and returns a t_handle.

sysmem_resizehandle

Use `sysmem_resizehandle` to resize an existing handle.

```
long sysmem_resizehandle(t_handle handle, long
newsize);
```

`handle`	The handle that will be resized.
`newsize`	The new size of the handle in bytes.

This function is similar to `SetHandleSize`. It resizes an existing handle to the size specified, and returns the number of bytes allocated to the specified handle.

sysmem_handlesize

Use `sysmem_handlesize` to find the size of a handle.

```
long sysmem_handlesize(t_handle handle);
```

`handle`	The handle whose size will be queried.

This function is similar to `GetHandleSize`. It returns the number of bytes allocated to the specified handle.

sysmem_freehandle

Use `sysmem_freehandle` to free memory allocated with `sysmem_newhandle`.

```
void sysmem_freehandle(t_handle *handle);
```

`handle`	The handle whose memory will be freed.

This function is similar to `DisposeHandle`. It frees the memory that had been allocated to the given handle.

sysmem_lockhandle

Use `sysmem_lockhandle` to set the locked/unlocked state of a handle.

```
long sysmem_lockhandle(t_handle handle, long lock);
```

`handle`	The handle that will be locked.
`lock`	The new lock state of the handle.

This function is similar to `HLock or HUnlock`. It sets the lock state of a handle, using a zero or non-zero number, and returns the previous lock state.

sysmem_ptrandhand

Use `sysmem_ptrandhand` to add memory to an existing handle and copy memory to the resized portion from a pointer.

```
long sysmem_ ptrandhand (void *p, t_handle h, long
size);
```

`p`	The existing pointer whose data will be copied into the resized handle.
`h`	The handle which will be enlarged by the size of the pointer.
`size`	The size in bytes that will be added to the handle.

This function is similar to `PtrAndHand`. It resizes an existing handle by adding a given number of bytes to it and copies data from a pointer into those bytes. It returns the number of bytes allocated to the specified handle.

File Routines

These routines assist your object in opening and saving files using the standard file package, as well as locating the user's files in the Max search path. There have been a significant number of changes to these routines (as well as the addition of many functions), so some history may be useful in understanding their use.

Prior to version 4, Max used a feature of Mac OS 9 called "working directories" to specify files. When you used the `locatefile` service routine, you would get back a file name and a volume number. This name (converted to a Pascal string) and the volume number could be passed to FSOpen to open the located file for reading. The `open_dialog` routine worked similarly.

In Mac OSX, working directories are no longer supported. In addition, the use of these "volume" numbers makes it somewhat difficult to port Max file routines to other operating systems, such as Windows XP, that specify files using complete pathnames (i.e., "C:\dir1\dir2\file.pat").

However, it is useful to be able to refer to the path and the name of the file separately. The solution in Max 4 involves the retention of the volume number (now called Path ID), but with a platform-independent wrapper that determines its meaning. There are now calls to locate, open, and choose files using C filename strings and Path IDs, as well as routines to convert between a ”native” format for specifying a file (such as a full pathname on Windows or an FSSpec on the Macintosh) to the C string and Path ID.

The path handling system in Max 4 works in two modes. In compatibility mode, a Path ID is a working directory references. This mode is specified when the user has a file called Path_Compatibility in their Max folder. If the file is not present, the Path ID is an index into a table of paths maintained by the Max kernel. These paths are stored internally in the native format of the host operating system (FSSpec on the Mac, full pathnames on Windows).

Path Compatibility mode is only needed for objects that have not been updated for Max 4's file handling. Once all objects have been updated, there will be no need to use the compatibility mode.

The native path format is called a `PATH_SPEC`; it will be defined differently for each target platform. Any code that deals directly with a `PATH_SPEC` must be considered platform-specific (as will code that reads and writes file contents, which may dealt with at a later date).

Now that paths in Max have changed to use the slash style, as opposed to the old Macintosh colon style (see the Max 4.3 documentation for a description of the file path styles), there is one function in particular that you will find useful for converting between the various ways paths can be represented, including operating system native paths. This function is `path_nameconform`. Note that for compatibility purposes Path API functions accept paths in any number of styles, but will typically return paths, or modify paths inline to use the new slash style. In addition to absolute paths, paths relative to the Max Folder, the “Cycling '74” folder and the boot volume are also supported. See the conformpath.help and ext_path.h files for more information on the various styles and types of paths. See the “filebyte” SDK example project for a demonstration of how to use the path functions to convert a Max name and path ref pair to a Windows native path for use with CreateFile.

There are a large number of service routine in the Max 4 kernel that support files, but only a handful will be needed by most external objects. In addition to the descriptions that follow, you should consult the movie, folder and filedate examples included with the SDK.

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.

saveas_dialog

Use `saveas_dialog` to present the user with the standard save file dialog.

```
short saveas_dialog (char *filename, short *path,
short format);
```

`filename`	A C string containing a default name for the file to save. If the user decides to save a file, its name is returned here.
`path`	If the user decides to save the file, the Path ID of the location chosen is returned here.
`format`	The default Max file format for saving the file. If `format` is set to 2, the Normal binary mode will be selected. If `format` is 0, Text will be selected. When the user decides to save the file, the choice of file format is returned here. If you pass 0L for `format` instead of a pointer to a short, the choice for saving the file in binary or text formats is not presented to the user. This is appropriate when you always save your object's files in a specialized format. format 1 was used in previous version of Max to save in "Old Format", which is no longer supported.

This function is a convenient wrapper for using Navigation Services or Standard File for saving files. It is appropriate when you are saving Binbufs, since it provides the user with the option of saving the file as text or in a binary format. `saveas_dialog` returns 0 if the user chose to save the file. If the user cancelled, a non-zero value is returned.

saveasdialog_extended

Use `saveasdialog_extended` to present the user with a save file dialog with your own list of file types.

```
short saveasdialog_extended(char *filename, short
*path, long *type, long *typelist, short numtypes);
```

`filename`	A C string containing a default name for the file to save. If the user decides to save a file, its name is returned here.
`path`	If the user decides to save the file, the Path ID of the location chosen is returned here.
type	Returns the type of file chosen by the user.
typelist	The list of types provided to the user.
numtypes	The number of types to be found in `typelist`.

saveasdialog_extended is similar to `saveas_dialog`, but allows the additional feature of specifying a list of possible types. These will be displayed in a pop-up menu.

File types found in the typelist argument that match known Max types will be displayed with descriptive text. Unmatched types will simply display the type name (for example, "foXx" is not a standard type so it would be shown in the pop-up menu as foXx)

Known file types are:

`TEXT—text file`

`maxb—Max binary patcher`

`maxc—Max collective`

`Midi—MIDI file`

`Sd2f—Sound Designer II audio file`

`NxTS—NeXT/Sun audio file`

`WAVE—WAVE audio file.`

`AIFF—AIFF audio file`

```
mP3f—Max preference file

PICT—PICT graphic file

MooV—Quicktime movie file

aPcs—VST plug-in

AFxP—VST effect patch data file

AFxB—VST effect bank data file

DATA—Raw data audio file

ULAW—NeXT/Sun audio file
```

open_promptset

Use `open_promptset` to add a prompt message to the open file dialog displayed by `open_dialog`.

```
short open_promptset (char *prompt);
```

`prompt`	A C string containing the prompt you wish to display in the dialog box.

Calling this function before `open_dialog` permits a string to displayed in the dialog box instructing the user as to the purpose of the file being opened. It will only apply to the call of `open_dialog` that immediately follows `open_promptset`.

saveas_promptset

Use `saveas_promptset` to add a prompt message to the open file dialog displayed by `saveas_dialog` or `saveasdialog_extended`..

```
short saveas_promptset (char *prompt);
```

`prompt`	A C string containing the prompt you wish to display in the dialog box.

Calling this function before `saveas_dialog` permits a string to displayed in the dialog box instructing the user as to the purpose of the file being saved. It will only apply to the call of `saveas_dialog` that immediately follows `saveas_promptset`.

defvolume

Use `defvolume` to get the volume or directory the user accessed most recently.

```
short defvolume (void);
```

This function returns the Path ID of the volume or folder in which the most recent file was opened. This routine performs the same function as the routine `path_getdefault`.

locatefile

Use `locatefile` to find a Max document by name in the search path.

```
short locatefile (char *filename, short *path,
short *binary);
```

`filename`	A C string that is the name of the file to look for.
`path`	The Path ID containing the location of the file if it is found.
`binary`	If the file found is in binary format (it's of type 'maxb') 1 is returned here; if it's in text format, 0 is returned.

`locatefile` searches through the directories specified by the user for Patcher files and tables in the File Preferences dialog as well as the current default path (see `path_getdefault`) and the directory containing the Max application. If a file is found with the name specified by `filename`, `locatefile` returns 0, otherwise it returns non-zero. The file's Path ID is returned in `path`. `binary` is non-zero if file is in Max binary format, 0 if it's in text format. `filename` and `vol` can then be passed to `binbuf_read` to read and open file the file. When using MAXplay, the search path consists of all subdirectories of the directory containing the MAXplay application. `locatefile` only searches for files of type 'maxb' and 'TEXT.'

locatefiletype

Use `locatefiletype` to find a file by name and/or filetype and creator in the search path.

```
short locatefiletype (char *filename, short *path,
OSType filetype, OSType creator);
```

`filename`	A C string that is the name of the file to look for.
`path`	The Path ID containing the location of the file if it is found.
`filetype`	The filetype of the file to look for. If you pass 0L, files of all filetypes are considered.
`creator`	The creator of the file to look for. If you pass 0L, files with any creator are considered.

This function searches through the same directories as `locatefile`, but allows you to specify a type and creator of your own. `locatefile`, in contrast, searches for only the standard types of Max files 'maxb' and 'TEXT'. If `locatefiletype` has a successful match, it returns 0, otherwise it returns non-zero.

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);
```

nameinpath

Use `nameinpath` to find a folder with a specific name in the Max search path.

```
short nameinpath(char *name, short *path);
```

`name`	The name of the file for the search.
`path`	A Path ID to search.

genpath

Use `genpath` to create a Path ID from a PATH_SPEC.

```
short genpath(PATH_SPEC *fs);
```

`fs`	A valid PATH_SPEC.

`genpath` returns a Path ID for a valid PATH_SPEC. If the PATH_SPEC is already found in the Max path table, the existing Path ID is returned. Otherwise, a new Path ID will be created, and the path will be added to the Max path table.

path_lookup

Use `path_lookup` to determine if a PATH_SPEC is already in the Max path table.

```
short path_lookup(PATH_SPEC *fs);
```

`fs`	A valid PATH_SPEC.

If the PATH_SPEC is found in the path table, the current Path ID will be returned. If it is not found, path_lookup will return 0.

path_new

Use path_new to add a PATH_SPEC to the Max path table.

```
short path_new(PATH_SPEC *fs);
```

`fs`	A valid PATH_SPEC

`path_new` will add the PATH_SPEC to the path table, and will return the Path ID. If there is an error, or if memory cannot be allocated for another path table entry, the routine will return 0.

path_tospec

Use `path_tospec` to load a PATH_SPEC from a Path ID/Name combination.

```
short path_tospec(short path, char *name, PATH_SPEC
*fs);
```

`path`	A Path ID, or 0 (for a fully qualified name argument).
`name`	A file name (which can include partial or full qualification).
`fs`	A PATH_SPEC structure to be loaded by this routine.

Given a Path ID and file name, path_tospec will return the complete PATH_SPEC for a file. If name contains a fully qualified file name, path can be set to 0. The return value is similar to MacOS file system calls—a value of 0 represents successful completion, while a non-zero value represents failure (where the value may represent an error code).

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.

path_namefromspec

Use `path_namefromspec` to retrieve a file name from a PATH_SPEC.

```
void path_namefromspec(PATH_SPEC *fs, char *name);
```

`fs`	A valid PATH_SPEC.
`name`	A pointer to a character string that will receive the file name.

The name of the file is retrieved from the PATH_SPEC and copied to the name parameter.

path_resolvefile

Use `path_resolvefile` to resolve a Path ID plus a (possibly extended) file name into a path that identifies the file’s directory and a filename.

```
short path_resolvefile(char *name, short path,
short *outpath);
```

`name`	A file name (which may be fully or partially qualified), will contain the file name on return.
`path`	The Path ID to be resolved.
`outpath`	The Path ID of the returned file name.

This routine converts a name and Path ID to a standard form in which the name has no path information and does not refer to an aliased file. It returns 0 if successful.

path_fileinfo

Use `path_fileinfo` to retrive a `t_fileinfo` structure from a file/path combination.

```
short path_fileinfo(char *name, short path, void
*info);
```

`name`	The file name to be queried.
`path`	The Path ID of the file.
`info`	A structure of type t_fileinfo containing file information.

`path_fileinfo` retrieves OS-specific information about a file and returns it in a OS-neutral `t_fileinfo` structure, declared as follows:

```
typedef struct _fileinfo
{
   long type;
   long creator;    // Mac-only
   long date;
   long flags;
} t_fileinfo;
```

The `flags` variable may contain the following flags:

```
// fileinfo flags
enum {
    FILEINFO_ALIAS = 1,
    FILEINFO_FOLDER = 2
};
```

This routine returns 0 if successful, otherwise it returns an OS-specific error code.

path_opensysfile

Use `path_opensysfile` to open a file given a filename and Path ID.

```
short path_opensysfile(char *name, short path,
t_filehandle *ref, short perm);
```

`name`	The name of the file to be opened.
`path`	The Path ID of the file to be opened.
`ref`	A FILE_REF that will contain the OS-specific pointer to a file. On the Mac OS it is a file refNum that you can pass to FSRead, etc.
`perm`	The file permission for the opened file.

This routine opens a file for reading or writing using Max's platform-neutral Sysfile routines (see below). The perm argument must contain one of the enumerated values of READ_PERM, WRITE_PERM or RW_PERM. Like other file system calls, this routine returns 0 if successful, and an OS-specific error code if unsuccessful.

path_createsysfile

Use `path_createsysfile` to create a file given a type code, a filename and a Path ID.

```
short path_createsysfile(char *name, short path,
long type, t_filehandle *ref);
```

`name`	The name of the file to be created.
`path`	The Path ID of the file to be created.
`type`	The file type of the created file.
`ref`	A t_filehandle containing a pointer to the opened file.

`path_createsysfile` will create a new file in an OS-neutral manner compatible with Max's Sysfile routines (see below), and open it for reading and writing. If the file already exists, a new file is created in its place. Like other file system calls, this routine returns 0 if successful, and an OS-specific error code if unsuccessful.

path_translate

Use `path_translate` to create a valid Path ID and file name from a PATH_SPEC, including optional alias resolution.

```
short path_translate(PATH_SPEC *fs, char *name,
short *vol, short resolvealias);
```

`fs`	The PATH_SPEC to translate.
`name`	The name of the file contained in PATH_SPEC fs.
`vol`	The Path ID of the file identified by PATH_SPEC fs.
`resolvealias`	If non-zero, and if the PATH_SPEC contains an alias, the returned name/Path ID will refer to the file pointed to by the alias.

`path_translate` is used to completely convert a PATH_SPEC into a Path ID and filename combination. Unlike `path_namefromspec` and `genpath`, resolution of aliases is available. Like other file system calls, this routine returns 0 if successful, and an error code if unsuccessful.

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_topotentialname

Use `path_topotentialname` to create a fully qualified file name from a Path ID/file name combination, regardless of whether or not the file exists on disk.

```
short path_topotentialname(short path, char *file,
char *name, short check);
```

`path`	The path to be used.
`file`	The file name to be used.
`name`	Loaded with the fully extended file name on return.
`check`	Flag to check if a file with the given path exists .

If the `check` flag is false, this function will not check to see if the path is valid, and always return a full path string. If it is true, this routine behaves exactly like `path_topathname`. Returns 0 if successful, and an error code if unsuccessful

path_frompathname

Use `path_frompathname` to create a filename and Path ID combination from a fully qualified file name.

```
short path_frompathname(char *name, short *path,
char *filename);
```

name	The extended file path to be converted.
`path`	Contains the Path ID on return.
`filename`	Contains the file name on return.

path_frompathname will return the Path ID and filename from a file path. It performs alias resolution in the conversion. This routine returns 0 if successful, and an error code if unsuccessful. Note that path_frompathname does not require that the file actually exist. In this way you can use it to convert a full path you may have received as an argument to a file writing message to a form appropriate to provide to a routine such as path_createfile.

path_setdefault

Use `path_setdefault` to install a path as the default search path.

```
void path_setdefault(short path, short recursive);
```

`path`	The path to use as the search path.
`recursive`	If non-zero, all subdirectories will be installed in the default search list.

The default path is searched before the Max search path. For instance, when loading a patcher from a directory outside the search path, the pathcer's directory is searched for files before the search path. path_setdefault allows you to set a path as the default.

If path is already part of the Max Search path, it will not be added (since, by default, it will be searched during file searches). Be very careful with the use of the recursive argument—it has the capacity to slow down file searches dramatically as the list of folders is being built. Max itself never creates a hierarchical default search path.

path_getdefault

Use `path_getdefault` to retrieve the Path ID of the default search path.

```
short path_getdefault(void);
```

`path_getdefault` returns the Path ID of the last path passed to `path_setdefault`. The routine for retrieving the default path in previous versions, `defvolume`, is still available and has the same effect as calling `path_getdefault`.

path_getmoddate

Use `path_getmoddate` to determine the modification date of the selected path.

```
short path_getmoddate(short path, unsigned long
*date);
```

`path`	The Path ID of the directory to check.
`date`	The last modification date of the directory.

path_getfilemoddate

Use `path_getfilemoddate` to retrieve the modification date of a specified file.

```
short path_getfilemoddate(char *filename, short
path, unsigned long *date);
```

`filename`	The name of the file to query.
`path`	The Path ID of the file.
`date`	Contains the last modification date on return.

path_getapppath

Use `path_getapppath` to retrieve the Path ID of the Max application.

```
short path_getapppath(void);
```

The return value is the Path ID of the Max application or runtime.

Routines for Iterating Through Folders

The following routines allow you to iterate through all of the files in a path.

path_openfolder

Use `path_openfolder` to prepare a directory for iteration.

```
void *path_openfolder(short path);
```

`path`	The directory Path ID to open.

The return value of this routine is an internal “folder state” structure used for further folder manipulation. It should be saved and used for calls to `path_foldernextfile` and `path_closefolder`. If the folder cannot be found or accessed, `path_openfolder` returns 0.

path_foldernextfile

Use `path_foldernextfile` to get the next file in the directory.

```
short path_foldernextfile(void *xx, long *filetype,
char *name, short descend);
```

`xx`	The "folder state" value returned by `path_openfolder`.
`filetype`	Contains the file type of the file type on return.
`name`	Contains the file name of the next file on return.
`descend`	Unused.

In conjunction with `path_openfolder` and `path_closefolder`, this routine allows you to iterate through all of the files in a path. `path_foldernextfile` may return a folder, in which case the filetype argument will contain 'fold'. This routine returns 0 if successful, and an error code if unsuccessful.

path_foldergetspec

Use `path_foldergetspec` to retrieve more information from a file in a directory.

```
short path_foldergetspec(void *xx, PATH_SPEC *spec,
short resolve);
```

`xx`	The "folder state" value returned by `path_openfolder`.
`spec`	The PATH_SPEC to contain additional information.
`resolve`	If non-zero, will resolve a file alias into an actual file.

Use path_foldergetspec to retrieve information held in a PATH_SPEC structure for the file at the current position in a folder iteration. This routine returns 0 if successful, and an error code if unsuccessful.

path_closefolder

Used `path_closefolder` to complete a directory iteration.

```
void path_closefolder(void *x);
```

x	The “folder state” value originally returned by `path_openfolder`.

This routine should be used whenever a directory iteration has been completed.

Sysfile API

The Sysfile API provides the means of reading and writing files opened by path_createsysfile and similar. These functions all make use of a new opaque structure, t_filehandle. The old path functions like path_createfile are Macintosh specific, and for cross platform development are obsolete. See the new path functions `path_opensysfile` and `path_createsysfile` described earlier in this chapter for more information. The Sysfile API is relatively similar to parts of the old Macintosh File Manager API, and not too different from Standard C library file functions. The “filebyte” example project in the SDK shows how to use these functions to read from a file. It is not safe to mix these routines with other file routines (e.g. don’t use `fopen` to open a file and `sysfile_close` to close it).

In addition to being able to use these routines to write cross-platform code in your max externals, another advantage of the Sysfile API is that it is able to read files stored in Max 4.3’s new collective file format on both Windows XP and Mac OSX.

sysfile_read

Use `sysfile_read` to read a file from disk.

```
long sysfile_read(t_filehandle fh, long *count,
void *bufptr);
```

`fh`	The t_filehandle structure of the file the user wants to open.
`count`	Pointer to the number of bytes that will be read from the file at the current file position. The byte count actually read is returned to this value.
`bufptr`	Pointer to the buffer that the data will be read into.

This function is similar to FSRead or fread. It should be used instead of these functions (or other system-specific file reading routines) in order to make max external code that will compile cross-platform. It reads "count" bytes from file handle at current file position into "bufptr". The byte count actually read is set in "count", and the file position is updated by the actual byte count read. The return value is an error code.

sysfile_write

Use `sysfile_write` to write part of a file to disk.

```
long sysfile_write(t_filehandle fh, long *count,
const void *bufptr);
```

`fh`	The t_filehandle structure of the file to which the user wants to write.
`count`	Pointer to the number of bytes that will be written to the file at the current file position. The byte count actually written is returned to this value.
`bufptr`	Pointer to the buffer that the data will be read from.

This function is similar to FSWrite or fwrite. It should be used instead of these functions (or other system-specific file reading routines) in order to make max external code that will compile cross-platform. The function writes "count" bytes from "bufptr" into file handle at current file position. The byte count actually written is set in "count", and the file position is updated by the actual byte count written. The return value is an error code.

sysfile_close

Use `sysfile_close` to close a file opened with `sysfile_open.`

```
long sysfile_close(t_filehandle fh);
```

`fh`	The t_filehandle structure of the file the user wants to close.

This function is similar to FSClose or fclose. It should be used instead of system-specific file closing routines in order to make max external code that will compile cross-platform. The return value is an error code.

sysfile_geteof

Use `sysfile_geteof` to get the size of a file handle.

```
long sysfile_geteof(t_filehandle fh, long *logeof);
```

`fh`	The file's t_filehandle structure.
`logeof`	The file size in bytes is returned to this vaue.

This function is similar to and should be used instead of GetEOF. The function gets the size of file handle in bytes, and places it in "logeof". The return value is an error code.

sysfile_seteof

Use `sysfile_seteof` to set the size of a file handle.

```
long sysfile_seteof(t_filehandle fh, long logeof);
```

`fh`	The file's t_filehandle structure.
`logeof`	The file size in bytes.

This function is similar to and should be used instead of SetEOF. The function sets the size of file handle in bytes, specified by "logeof". The return value is an error code.

sysfile_getpos

Use `sysfile_getpos` to get the current file position of a file handle.

```
long sysfile_getpos(t_filehandle fh, long
*filepos);
```

`fh`	The file's t_filehandle structure.
`filepos`	The current read/write position in bytes is returned to this vaue.

This function is similar to and should be used instead of GetFPos. The function gets the current file position of file handle in bytes, and places it in "filepos". The return value is an error code.

sysfile_setpos

Use `sysfile_setpos` to get the current file position of a file handle.

```
long sysfile_setpos(t_filehandle fh, long mode,
long offset);
```

`fh`	The file's `t_filehandle` structure.
`mode`	Mode from which the offset will be calculated.
`offset`	The offset in bytes relative to the mode.

This function is similar to and should be used instead of SetFPos. It is used to set the current file position of file handle to by the given number of offset bytes relative to the mode used. The three modes are:

`SYSFILE_FROMSTART`	Calculate the file position from the start of the file.
`SYSFILE_FROMLEOF`	Calculate the file position from the logical end of the file.
`SYSFILE_FROMMARK`	Calculate the file position from the current file position.

The return value is an error code.

sysfile_readtextfile

Use `sysfile_readtextfile` to read a text file from disk.

```
long sysfile_readtextfile(t_filehandle fh, t_handle
htext, long maxlen, long flags);
```

`fh`	The t_filehandle structure of the text file the user wants to open.
`htext`	Handle that the data will be read into.
`maxlen`	The maximum length in bytes to be read into the handle. Passing the value 0L indicates no maximum (i.e. read the entire file).
`flags`	Flags to set linebreak translation.

This function reads up to the maximum number of bytes given by `maxlen` from file handle at current file position into the `htext` handle, performing linebreak translation if set in `flags`. The four possible linebreak flags are as follows:

`TEXT_LB_NATIVE`	Use the linebreak format native to the current platform.
`TEXT_LB_MAC`	Use Macintosh linebreaks.
`TEXT_LB_PC`	Use PC linebreaks.
`TEXT_LB_UNIX`	Use Unix linebreaks.

The return value is an error code.

sysfile_writetextfile

Use `sysfile_writetextfile` to write a text file to disk.

```
long sysfile_writetextfile(t_filehandle fh,
t_handle htext, long flags);
```

`fh`	The t_filehandle structure of the file to which the user wants to write.
`htext`	Handle that the data that will be read from.

`flags`	Flags to set linebreak translation.

This function writes a text handle to a text file performing linebreak translation if set in `flags`. For a list of the available flags, please see `sysfile_readtextfile`, above. The return value is an error code.

A File Handling Example

Below is an example where we use an object's read method in conjunction with `open_dialog` and `locatefile_extended`. This is how we've bound the read method in the initialization routine.

```
addmess((method)myobject_read, "read", A_DEFSYM, 0);
// optional symbol argument to specify name
```

Here is the first part of the actual read method, deferred to a routine called `myobject_doread`.

```
void myobject_read(t_myobject *x, t_symbol *s)
{
   defer(x,(method)myobject_doread,s,0,0);    // always defer
this message
}
void myobject_doread(t_myobject *x, t_symbol *s, short argc,
t_atom *argv)
{
   char filename[256];
   short path, err;
   long type = 'DATA';      // some file type you're looking
for
   long outtype, count;
   Byte data[128];
   t_filehandle fh; // you probably will want this to be an
instance
                    // variable in your object's data struct
   if (!s->s_name[0]) { // empty symbol
      if (open_dialog(filename, &path, &outtype, &type, 1))
          return;   // user cancelled
   } else {
      strcpy(filename, s->s_name);
      // important: copy symbol arg to local string
      if (locatefile_extended(filename, &path, &outtype,
&type, 1))
          return;   // not found
   }
   // at this point, a valid name is in filename and
   // a valid path is in path
   err = path_opensysfile(ps, path, &fh, READ_PERM);
   if (err) {
      fh = 0;
      error("error %d opening file", err);
      return;
   }
```

A `t_filehandle` is a cross-platform of referring to an open file. It is an opaque structure, meaning you don't have access to the individual elements of the data structure. You can use a `t_filehandle` only with the file routines in the Sysfile API. Do not use other platform-specific file functions in conjunction with these functions. The perm parameter can be either READ_PERM, WRITE_PERM, or RW_PERM.

When writing files, you will typically want to use `path_createsysfile` instead of `path_opensysfile,` as `path_createsysfile` takes care of creating the file if it doesn't already exist—this replaces a lot of code which is often necessary with system-specific file routines, since you receive an error if you attempt to create a file with an existing name, then have to try again to sucessfully open the file. `path_createsysfile` provides all of this functionality.

Continuing with the `myobject_doread` example:

```
    // open file for reading
    err = path_opensysfile(filename, path, &fh, READ_PERM);
    if (err) {
        // report any errors
        error("%s: error %d opening file", filename, err);
        return;
    }
    // read 128 bytes from the file...
    count = 128;
    err = sysfile_read(fh, &count, &data);
    // do something with the data you've just read
    // close the file when you're done...
    sysfile_close(fh);
}
```

Chapter 9: Advanced Facilities

The following sections discuss capabilities that most objects will not need to use, but may be of interest to advanced programmers.

Advanced Object Creation and Message Routines

These routines allow you to create your own instances of classes—either existing ones or those you define—and send them "untyped" messages. 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.

newinstance

Use `newinstance` to make a new instance of an existing Max class.

```
t_object *newinstance (t_symbol *className, short
argc, t_atom *argv);
```

`className`	Symbol specifying the name of the class of the instance to be created.
`argc`	Count of arguments in `argv`.
`argv`	Array of t_atoms; arguments to the class's instance creation function.

This function creates a new instance of the specified class. Using `newinstance` is equivalent to typing something in a New Object box when using Max. The difference is that no object box is created in any Patcher window, and you can send messages to the object directly without connecting any patch cords. The messages can either be type-checked (using `typedmess`) or non-type-checked (using the members of the `getfn` family).

`newinstance` returns a pointer to the created object, or 0 if the class didn't exist or there was another type of error in creating the instance. This function is useful for taking advantage of other already-defined objects that you would like to use "privately" in your object, such as tables. See the source code for the coll object for an example of using a privately defined class.

typedmess

Use `typedmess` to send a typed message directly to a Max object.

```
void *typedmess (void *receiver, t_symbol *message,
short argc, t_atom *argv);
```

`receiver`	Max object that will receive the message.
`message`	The message selector.
`argc`	Count of message arguments in `argv`.
`argv`	Array of t_atoms; the message arguments.

`typedmess` sends a message to a Max object (`receiver`) a message with arguments. If the `receiver` object can respond to the message, `typedmess` returns the result. Otherwise, an error message will be seen in the Max window and 0 will be returned. Note that the message must be a t_symbol, not a character string, so you must call `gensym` on a string before passing it to `typedmess`. Also, note that untyped messages defined for classes with the argument list `A_CANT` cannot be sent using `typedmess`. You must use `getfn` etc. instead described below.

Example: If you want to send a **bang** message to the object `bang_me`…

```
void *bangResult;
bangResult = typedmess(bang_me,gensym("bang"),0,0L);
```

Routines for Sending Untyped Messages

The following three routines send non type-checked messages to objects. You are responsible for passing the message arguments correctly. System errors, rather than error messages in the Max window, are likely if you don't. These functions could be useful with an object created with `newinstance`.

getfn

Use `getfn` to send an untyped message to a Max object with error checking.

```
method getfn (t_object *obj, t_symbol *msg);
```

`obj`	Receiver of the message.
`msg`	Message selector.

`getfn` returns a pointer to the method bound to the message selector `msg` in the receiver's message list. It returns 0 and prints an error message in Max Window if the method can't be found.

egetfn

Use `egetfn` to send an untyped message to a Max object that always works.

```
method egetfn (t_object *obj, t_symbol *msg);
```

`obj`	Receiver of the message.
`msg`	Message selector.

`egetfn` returns a pointer to the method bound to the message selector `msg` in the receiver's message list. If the method can't be found, a pointer to a do-nothing function is returned.

zgetfn

Use `zgetfn` to send an untyped message to a Max object without error checking.

```
method zgetfn (t_object *obj, t_symbol *msg);
```

`obj`	Receiver of the message.
`msg`	Message selector.

zgetfn returns a pointer to the method bound to the message selector msg in the receiver's message list. It returns 0 but doesn't print an error message in Max Window if the method can't be found.

Using Untyped Messages

The macros `mess0`, `mess1`, `mess2`, etc. defined in *ext_mess.h* are useful for sending non type-checked messages. Here's an illustration of using `newinstance` and non type-checked messages.

In this example, the **bob** object's info method creates an instance of the **joe** class, sends it an info message (which presumably does something) and destroys the instance of **joe**.

```
void bob_info(Bob *x, void *p, void *b)
{
   void *joe;
   joe = newinstance(gensym("joe"),0,0L);  /* create a joe
*/
   mess2(joe,gensym("info"),p,b);  /* send untyped message
*/
   freeobject(joe);  /* kiss joe goodbye */
}
```

Table Access

You can use these functions to access named **table** objects. Tables have names when the user creates a **table** with an argument, such as…

table norris

The scenario for knowing the name of a **table** but not the object itself is if you were passed a Symbol, either as an argument to your creation function or in some message, with the implication being "do your thing with the data in the **table** named norris."

table_get

Use `table_get` to get a handle to the data in a named table object.

```
short table_get (t_symbol *tableName, long
***dstHandle, long *dstSize);
```

`tableName`	Symbol containing the name of the **table** object to find.
`dstHandle`	Address of a handle where the table's data will be returned if the named **table** object is found.
`dstSize`	Number of elements in the table (its size in longs).

`table_get` searches for a **table** associated with the `t_symbol tableName`. If one is found, a Handle to its elements (stored as an array of long integers) is returned and the function returns 0. If no **table** object is associated with the symbol `tableName`, `table_get` returns a non-zero result. Never count on a **table** to exist across calls to one of your methods. Call `table_get` and check the result each time you wish to use a **table**.

Here is an example of retrieving the 40th element of a **table**:

```
long **storage,size,value;
if (!table_get(gensym("somename"),&storage,&size)) {
    if (size > 40)
        value = *((*storage)+40);
}
```

table_dirty

Use `table_dirty` to mark a **table** object as having changed data.

```
short table_dirty (t_symbol *tableName);
```

`tableName`	Symbol containing the name of a **table** object.

Given the name of a **table** object in tableName, `table_dirty` sets its dirty bit, so the user will asked to save changes if the **table** is closed. If no **table** is associated with `tableName`, `table_dirty` returns a non-zero result.

Text Editor Windows

Max has a simple built-in text editor object *Ed* that can display and edit text in conjunction with your object. The routines described below let you create a text editor.

When the editor window is about to be closed, your object could receive as many as three messages. The first one, okclose, will be sent if the user has changed the text in the window. This is the standard okclose message that is sent to all “dirty” windows when they are about to be closed, but the text editor window object passes it on to you instead of doing anything itself. Refer to the section on Window Messages for a description of how to write a method for the okclose message. It’s not required that you write one—if you don’t, the behavior of the window will be determined by the setting of the window’s `w_scratch` bit (described in Chapter 10). If it’s set, no confirmation will be asked when a dirty window is closed (and no okclose message will be sent to the text editor either). The second message, edclose, requires a method that should be added to your object at initialization time. The third message, edSave, allows you to gain access to the text before it is saved, or save it yourself.

edclose

edclose will be sent to your object in the following two cases:

- The window's `w_scratch` bit has been set
- The user clicked on Save in the "Save Changes?" alert placed on the screen when the window was about to be closed. The window's `w_scratch` bit was not set in this case.

BINDING

```
addmess (myObject_edclose, "edclose", A_CANT, 0);
```

DECLARATION

```
void myobject_edclose (Myobject *obj, char **text,
long size);
```

`text`	A handle to the edited text.
`size`	The size of the handle in bytes.

In this method, the editor will hand you the text in the editing window. If you set the window's `w_scratch` bit, you might want to know if the text in the window was modified by the user. The definition of the text editing object `t_ed` is in *ext_edit.h*. Use the following code to check the dirty bit of the editor's window (assume that `x->m_edit` points to your text editor object).

```
if (x->m_edit->e_wind->w_dirty)
    /* has been modified */
```

In the case where the text has been modified, you can update the state of your object using the text arguments of the edclose message.

After receiving an edclose message, the text editor window is destroyed, so it's important to note this in the internal state of your object. It's a good idea to set the field of your object that points to the text editor to 0 at this point.

edsave

The edsave message allows your object to customize the file saving of a text-editing window.

BINDING

```
addmess (myObject_edsave, "edsave", A_CANT, 0);
```

DECLARATION

```
void *myobject_edsave (myObject *x, char **text,
long size, char *filename, short vol);
```

`text`	Handle to the text buffer.
`size`	Length of the text buffer.
`filename`	C string containing the file's name. The file may need to be created.
`vol`	Volume reference number specifying the location of the file.

Your object will receive this message when a text editor file is about to be saved. Your method can save the text in the specified file in any format it desires. If you just wanted to gain access to the text without saving it, return 0 from this function, and the normal file saving procedure will be used. Otherwise, return any non-zero value.

ed_new

Use `ed_new` to make a new text editing window.

```
t_ed *ed_new (t_object *assoc);
```

`assoc`	The object associated with the text editing window. Normally this is a pointer to your object.

This function creates a new text-editing window. The text editor will be visible immediately. `assoc` should be a pointer to your object, so it can receive the all-important edclose message. You should store the result of `ed_new` in your object for two reasons. First, so you can call `ed_settext` to set the text in the window and second, so you can call `ed_vis` to bring the window to the front when your object receives a dblclick message.

ed_settext

Use `ed_settext` to set the contents of a text-editing window.

```
void ed_settext (t_ed *ew, Handle textHandle, long
textSize);
```

`ew`	The text editing window object.
`textHandle`	Handle containing the text to put in the window.
`textSize`	Number of characters in the `textHandle.`

`ed_settext` the text in a text editing window to the characters in `textHandle`. The window will be refreshed to show the new text.

ed_vis

Use `ed_vis` to bring the text-editing window to the front.

```
void ed_vis (t_ed *ew);
```

`ew`	The text editing window object.

If you want to implement the standard behavior for text editor windows, you'll make yours visible only after receiving a dblclick message. Since the editor window is immediately visible when created with `ed_new`, don't create an editor in your object creation function. Rather, initialize the slot for storing your editor to 0 when your object is created. Then you can determine whether you need to create a new editor in your dblclick method, or bring one that already exists to the front.

Here's an example dblclick method. We assume that `x->m_edit` is a field in your object that contains a text editor.

```
void myobject_dblclick(Myobject *x)
{
    if (x->m_edit)
        ed_vis(x->m_edit);
    else {
        x->m_edit = ed_new(x);
    /* set text of editor here with ed_settext */
    }
}
```

Access to expr Objects

If you want to use C-like variable expressions that are entered by a user of your object, you can use the "guts" of Max's **expr** object in your object. For example, the **if** object uses **expr** routines for evaluating a conditional expression, so it can decide whether to send the message after the words then or else. The following functions provide an interface to **expr**.

Note: As with Binbuf, Clock, and Qelem, the Expr type is just pointer to void. Constants and other declarations needed to use Expr are found in *ext_expr.h*.

expr_new

Use `expr_new` to create a new **expr** object.

```
Expr *expr_new (short argc, t_atom *argv, t_atom
*types);
```

`argc`	Count of arguments in argv.
`argv`	Arguments that are used to create the expr. See the example below for details.
`types`	A pre-existing array of nine t_atoms, that will hold the types of any variable arguments created in the expr. The types are returned in the a_type field of each t_atom. If an argument was not present, `A_NOTHING` is returned.

`expr_new` creates an **expr** object from the arguments in `argv` and returns the type of any **expr**-style arguments contained in `argv` (i.e.

$i1, etc.) in atoms in an array pointed to by `types`. `types` should already exist as an array of nine Atoms, all of which will be filled in by `expr_new`. If an argument was not present, it will set to type `A_NOTHING`. For example, suppose `argv` pointed to the following atoms:

```
$i1    (A_SYM)
+      (A_SYM)
$f3    (A_SYM)
+      (A_SYM)
3      (A_LONG)
```

After calling `expr_new`, `types` would contain the following:

Index	*Argument*	*Type*	*Value*
0	1 ($i1)	`A_LONG`	0
1	2	`A_NOTHING`	0
2	3 ($f3)	`A_FLOAT`	0.0
3	4	`A_NOTHING`	0
4	5	`A_NOTHING`	0
5	6	`A_NOTHING`	0
6	7	`A_NOTHING`	0
7	8	`A_NOTHING`	0
8	9	`A_NOTHING`	0

expr_eval

Use `expr_eval` to evaluate an expression in an **expr** object.

```
void expr_eval (Expr *ex, short argc, t_atom *argv,
t_atom *result);
```

`ex`	**expr** object to evaluate.
`argc`	Count of arguments in `argv`.

`argv`	Array of nine Atoms that will be substituted for variable arguments (such as $i1) in the expression. Unused arguments should be of type `A_NOTHING`.
`result`	A pre-existing Atom that will hold the type and value of the result of evaluating the expression.

Evaluates the expression in an **expr** object with arguments in `argv` and returns the type and value of the evaluated expression as a t_atom in `result`. `result` need only point to a single t_atom, but `argv` should contain at least `argc` Atoms. If, as in the example shown above under `expr_new`, there are "gaps" between arguments, they should be filled in with t_atom of type `A_NOTHING`.

Presets

Max contains a **preset** object that has the ability to send preset messages to some or all of the objects (*clients*) in a Patcher window. The preset message, sent when the user is *storing* a preset, is just a request for your object to tell the **preset** object how to restore your internal state to what it is now. Later, when the user *executes* a preset, the **preset** object will send you back the message you had previously said you wanted.

The dialog goes something like this:

- During a store…**preset** *object to Client object(s)*: hello, this is the preset message—tell me how to restore your state*Client object to* **preset** *object:* send me int 34 (for example)
- During an execute…**preset** *object to Client object*: int 34

The client object won't know the difference between receiving int 34 from a **preset** object and receiving a 34 in its leftmost inlet.

It's not mandatory for your object to respond to the preset message, but it is something that will make users happy. All Max user interface objects currently respond to preset messages. Note that if your object is *not* a user interface object and implements a preset method, the user will need to connect the outlet of the **preset** object to its leftmost inlet in order for it to be sent a preset message when the user stores a preset.

Here are routines you can use when responding to the preset message.

preset_int

Use `preset_int` to restore the state of your object with an int message.

```
void preset_int (t_object *obj, long value);
```

`obj`	Your object.
`value`	Current value of your object.

This function causes an int message with the argument `value` to be sent to your object from the preset object when the user executes a preset. All of the existing user interface objects use the int message for restoring their state when a preset is executed.

preset_set

Use `preset_set` to restore the state of your object with a set message.

```
void preset_set (t_object *obj, long value);
```

`obj`	Your object.
`value`	Current value of your object.

This function causes a set message with the argument `value` to be sent to your object from the preset object when the user executes a preset.

preset_store

Use `preset_store` to give the **preset** object a general message to restore the current state of your object.

```
void preset_store (char *format, void *items, ...);
```

`format`	C string containing one or more letters corresponding to the types of each element of the message. `s` for Symbol, `l` for long, or `f` for float.
`items`	Elements of the message used to restore the state of your object, passed directly to the function as Symbols, longs, or floats. See below for an example that conforms to what the **preset** object expects.

This is a general preset function for use when your object's state cannot be restored with a simple int or set message. The example below shows the expected format for specifying what your current state is to a **preset** object. The first thing you supply is your object itself, followed by the symbol that is the name of your object's class (which you can retrieve from your object using the macro `ob_sym`, declared in *ext_mess.h*). Next, supply the symbol that specifies the message you want receive (a method for which had better be defined in your class), followed by the arguments to this message—the current values of your object's fields.

Here's an example of using `preset_store` that specifies that the object would like to receive a set message. We assume it has one field, `myvalue`, which it would like to save and restore.

```
void myobject_preset(myobject *x)
{
   preset_store("ossl",x,ob_sym(x),gensym("set"),x-
>myvalue);
}
```

When this preset is executed, the object will receive a set message whose argument will be the value of `myvalue`. Note that the same thing can be accomplished more easily with `preset_set` and `preset_int` discussed below.

Don't pass more than 12 items to `preset_store`. If you want to store a huge amount of data in a preset, use `binbuf_insert`.

The following example locates the Binbuf into which the preset data is being collected, then calls `binbuf_insert` on a previously prepared array of Atoms. It assumes that the state of your object can be restored with a set message.

```
void myobject_preset(myObject *x)
{
    void *preset_buf; /* Binbuf that stores the preset */
    short atomCount;  /*number of atoms you’re storing */
    t_atom atomArray[SOMESIZE]; /* array of atoms to be
stored */
    /* 1. prepare the preset "header" information */
    SETOBJ(atomArray,x);
    SETSYM(atomArray+1,ob_sym(x));
    SETSYM(atomArray+2,gensym("set"));
    /**fill atomArray+3 with object's state here and set
atomCount*/
    /* 2. find the Binbuf */
    preset_buf = gensym("_preset")->s_thing;
    /* 3. store the data */
    if (preset_buf) {
        binbuf_insert(preset_buf,NIL,atomCount,atomArray);
    }
}
```

Event and File Serial Numbers

If you call `outlet_int`, `outlet_float`, `outlet_list`, or `outlet_anything` inside a Qelem or during some idle or interrupt time, you should increment Max’s Event Serial Number beforehand. This number can be read by objects that want to know if two messages they have received occurred at the same logical “time” (in response to the same event). Max increments the serial number for each tick of the clock, each key press, mouse click, and MIDI event. Note that this is different from the *file serial number* returned by the `serialno` function. The file serial number is only incremented when patchers are saved in files. If more than one patcher is saved in a file, the file serial number will change but the event serial number will not.

evnum_incr

Use `evnum_incr` to increment the event serial number.

```
void evnum_incr (void);
```

evnum_get

Use `evnum_get` to get the current value of the event serial number.

```
long evnum_get (void);
```

serialno

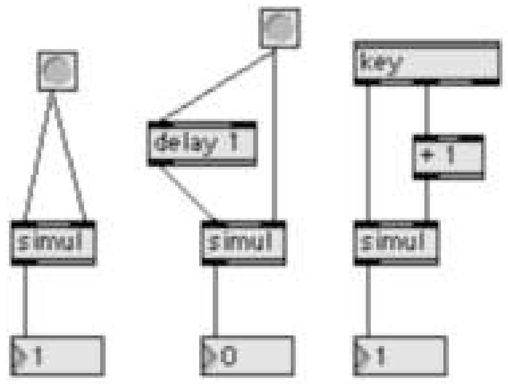
Use `serialno` to get a unique number for each Patcher file saved.

```
short serialno (void);
```

This function returns a serial number that is incremented each time a Patcher file is saved. This routine is useful for objects like **table** and **coll** that have multiple objects that refer to the same data, and can "embed" the data inside a Patcher file. If the serial number hasn't changed since your object was last saved, you can detect this and avoid saving multiple copies of the object's data.

Using Event Serial Numbers

Here is a Max patch that includes an object called **simul** that would use the information returned by `evnum_get` to return a 1 if the right and left inlets receive messages at the same time, 0 if not. The number boxes below show the results of clicking on the **button** objects or typing a key.



Loading Max Files

Several high-level functions permit you to load patcher files. These can be used in sophisticated objects that use Patcher objects to perform specific tasks.

stringload

Use `stringload` to load a patcher file located in the Max search path by name.

```
void *stringload (char *name);
```

`name`	Filename of the patcher file to load (C string).

This function searches for a binary or text patcher file, opens it, evaluates it as a patcher file, opens a window for the patcher and brings it to the front. You need only specify a filename and Max will look through its search path for the file. The search path begins with the current "default volume" that is often the volume of the last opened patcher file, then the folders specified in the File Preferences dialog, searched depth first, then finally the folder that contains the Max application. If `stringload` returns a non-zero result, you can later use `freeobject` to close the patcher, or just let users do it themselves. If `stringload` returns zero, no file with the specified name was found or there was insufficient memory to open it.

fileload

Use `fileload` to load a patcher file by name and volume reference number.

```
void *fileload (char *name, short path);
```

`name`	Filename of the patcher file to load (C string).
path	Path ID specifying the location of the file.

`fileload` requires that you specify a Path ID for the path argument, such as is returned from `open_dialog` or `locatefile_extended`. If the file is found, `fileload` tries to open the file, evaluate it, open a window, and bring it to the front. A pointer to the newly created Patcher is returned if loading is successful, otherwise, if the file is not found or there is insufficient memory, zero is returned.

readtohandle

Use `readtohandle` to load a data file into a handle.

```
short readtohandle (char *name, short path, char
***hp, long *size);
```

`name`	Name of the patcher file to load.
`path`	Path ID specifying the location of the file.
`hp`	Pointer to a handle variable that will receive the handle that contains the data in the file.
`size`	Size of the handle returned in `hp`.

This is a low-level routine used for reading text and data files. You specify the file's name and Path ID, as well as a pointer to a Handle. If the file is found, `readtohandle` creates a Handle, reads all the data in the file into it, assigns the handle to the variable `hp`, and returns the size of the data in `size`. `readtohandle` returns 0 if the file was opened and read successfully, and non-zero if there was an error.

lowload

Use `lowload` to pass arguments to Max files when you open them.

```
void *lowload (char *filename, short path, short
argc, t_atom *argv, short couldedit);
```

`filename`	Name of the file to open.
`path`	Path ID specifying the location of the file.
`argc`	Count of t_atoms in `argv`. To properly open a patcher file, `argc` should be 9.
`argv`	Array of t_atoms that will replace the changeable arguments #1-#9. The default behavior could be to set all these to t_atoms of type `A_LONG` with a value of 0.
`couldedit`	If non-zero and the file is not a patcher file, the file is opened as a text file.

This function loads the specified file and returns a pointer to the created object. Generally, `lowload` is used to open patcher files, whether they are in text or Max binary format. It can also open table files whose contents begin with the word "table."

If couldedit is non-zero and the file is not a patcher file, it is made into a text editor, and lowload returns 0. If couldedit is non-zero, lowload will just alert the user to an error and return 0. If there is no error, the value returned will be a pointer to a patcher or table object.

Connecting Objects As Clients and Servers

The Connection facility in Max allows two or more objects that may not be created or destroyed at the same time to be linked together via a standard set of routines and messages. This might be useful if you wish to provide an editor for a named data structure (such as a **coll** or **table** object) that automatically displays the named data when a corresponding object is loaded, and is updated when the corresponding data is changed in some way.

Connections involve *clients*, the objects that wish to access the data, and *servers*, objects that can be found by name when they attach themselves to a particular symbol. A **table** object could be a server (and in fact is—you can use the Connection routines to communicate with one). To establish your object is a client, you call `connection_client`. To establish your object as a server, call `connection_server`. If and when one or more clients and a server exist for the same name, the client objects will receive the newserver message. When a server attached to clients is freed, it calls `connection_delete` and its clients receive the freeserver message. In addition, the server can define messages to send to its clients. For example, the **table** object sends the message tabchanged to its clients when its data changes. This is done through the function `connection_send` so that the server does not need to keep track of its clients.

connection_client

Use `connection_client` to register a client with a symbolic name.

```
void *connection_client (void *client, t_symbol
*name, t_symbol *class, method traverse);
```

`client`	Client object to be registered.
`name`	Name under which the client will be registered.
`class`	Name of the class of the server. For a Max object `obj`, this is `ob_sym(obj)`.
`traverse`	A function that allows the connection facility to link multiple clients together by returning a pointer to a field within the client object that can be used for this purpose. Thus, in order to use the connection routines, the client object's data structure must include a link pointer variable to this same data structure. See below for an example of a traversal function.

This function registers a client with a name. If a server with this name already exists and has already registered by calling `connection_server`, `connection_client` will cause the object client to receive the newserver message (see below). Otherwise, the newserver message will be sent to the client whenever a server object with the specified name calls `connection_server`.

Here's an example of a traversal function you'd pass as the traverse argument. First, here's a data structure with a link pointer in it.

```
typedef struct myclient {
    t_object c_ob;
    struct myclient *c_next;
    void *c_data;
} t_myclient;
```

The traverse method is declared as:

```
void *myobject_traverse(t_myobject *x, t_myobject
***ptr);
```

The function should set `ptr` to the address of the "points to next" field in the data structure, and then return the current contents of this field.

For the Myclient data structure shown above, the traverse method would look like this:

```
void *myclient_traverse(t_myclient *x, t_myclient ***addr)
{
   *addr = &x->c_next;
   return (x->c_next);
}
```

connection_server

Use `connection_server` to register a server with a symbolic name.

```
void connection_server (void *server, t_symbol
*name);
```

`server`	Server object to be registered.
`name`	Name under which the server will be registered.

This function registers an object `server` with a name (`name`). If client objects already exist that are attached to this name and whose class is the same as the `server` object's, they are informed of the presence of the server object with the newserver message. This method should be declared as follows:

```
void myobject_newserver (t_myobject *x, void
*server);
```

Using this method, the client can store a reference to the server if it needs direct access to the object.

After calling `connection_server`, a server can send messages to all its clients using `connection_send`.

connection_send

Use `connection_send` to send messages from a server to all its clients.

```
void connection_send (void *server, t_symbol *name,
t_symbol *message, void *arg);
```

`server`	Registered server object.
`name`	Name under which the server is registered.
`message`	Message selector.
`arg`	Message argument.

`connection_send` verifies the connection status of the object `server` bound to the symbol `name`, then sends the untyped message specified by the symbol `message` (along with `arg`) to any currently connected clients. Since the server never knows when it actually has clients, it should call `connection_send` in all possible situations. If there are no clients, `connection_send` will do nothing (safely).

connection_delete

Use `connection_delete` to remove a client or server from a connection.

```
void connection_delete (void *obj, t_symbol *name);
```

`obj`	Registered client or server object.
`name`	Name under which the client or server is registered.

Both clients and servers use `connection_delete` (passing themselves as the `object` argument) when they want to break a connection (usually in an object's free function). The name of the connection is supplied in `name`. If `connection_delete` is called by a server, all connected clients will receive the freeserver message. This message should be implemented as follows:

```
void myobject_freeserver (t_myobject *x, void
*server);
```

After receiving this message, a client should make no further direct references to the object server, since it is likely being disposed of.

Error Message Subscription

In certain cases, it may be desireable to receive error messages that are sent to the Max window.

error_subscribe

Use `error_subscribe` to receive messages from the error handler.

```
void error_subscribe(t_object *myobject);
```

myobject	The object subscribed to the error handler.

`error_subscribe` enables your object to receive a message (**error**), followed by the list of atoms in the error message posted to the Max window.

Prior to calling `error_subscribe`, you should bind the error message to an internal error handling routine:

```
addmess((method)myobject_error, "error", A_GIMME,
0);
```

Your error handling routine should be declared as follows:

```
void myobject_error(t_myobject *x, t_symbol *s,
short argc, t_atom *argv);
```

error_unsubscribe

Use `error_unsubscribe` to remove an object as an error message recipient.

```
void error_unsubscribe(t_object *myobject);
```

`myobject`	The object to unsubscribe.

myobject will no longer receive error messages after this call.

Scheduling with setclock Objects

The **setclock** object allows a more general way of scheduling Clocks by generalizing the advancement of the time associated with a scheduler. Each **setclock** object's "time" can be changed by a process other than the internal millisecond clock. In addition, the object implements routines that modify the mapping of the internal millisecond clock onto the current value of time in an object. Your object can call a set of routines that use either **setclock** or the normal millisecond clock transparently. Many Max objects accept the message clock followed by an optional symbol to set their internal scheduling to a named **setclock** object. The typical implementation passes the binding of a Symbol (the `s_thing` field) to the Setclock functions. By default, the empty symbol is passed. If the binding has been linked to a **setclock** object, it will be used to schedule the Clock. Otherwise, the Clock is scheduled using the main internal millisecond scheduler. The Setclock data structure is a replacement for `void *` since there will be no reason for external objects to access it directly.

setclock_delay

Use `setclock_delay` to schedule a Clock on a scheduler.

```
void setclock_delay (Setclock *scheduler, Clock
*clk, long time);
```

`scheduler`	A **setclock** object to be used for scheduling this clock.
`clk`	Clock object containing the function to be executed.
`time`	Time delay (in the units of the Setclock) from the current time when the Clock will be executed.

Schedules the Clock `clk` to execute at `time` units after the current time. If `scheduler` is 0 or does not point to a **setclock** object, the internal millisecond scheduler is used. Otherwise `clk` is scheduled on the **setclock** object's list of Clocks. The Clock should be created with `clock_new`, the same as for a Clock passed to `clock_delay`.

setclock_fdelay

Use setclock_fdelay to schedule a Clock on a scheduler, using a floating-point time argument.

```
void setclock_fdelay(Setclock *scheduler, Clock
*clk, double time);
```

scheduler	A setclock object to be used for scheduling this clock.
clk	Clock object containing the function to be executed.
time	Time delay from the current time when the Clock will be executed.

`setclock_fdelay` is the floating-point equivalent of `setclock_delay`.

setclock_unset

Use `setclock_unset` to remove a Clock from a scheduler.

```
void setclock_unset (Setclock *scheduler, Clock
*clk);
```

`scheduler`	The **setclock** object that was used to schedule this clock. If 0, the clock is unscheduled from the internal millisecond scheduler.
`clk`	Clock object to be removed from the scheduler.

This function unschedules the Clock `clk` in the list of Clocks in the **setclock** object, or the internal millisecond scheduler if `scheduler` is 0.

setclock_gettime

Use `setclock_gettime` to find out the current time value of a **setclock** object.

```
long setclock_gettime (Setclock *scheduler);
```

`scheduler`	A **setclock** object.

Returns the current time value of the **setclock** object `scheduler`. If `scheduler` is 0, `setclock_gettime` is equivalent to the function `gettime` that returns the current value of the internal millisecond clock.

setclock_getftime

Use `setclock_getftime` to find out the current time value of a **setclock** object in floating-point milliseconds.

```
void setclock_getftime(Setclock *scheduler, double
*time);
```

scheduler	A **setclock** object.
time	The current time in milliseconds.

`setclock_getftime` is the floating-point equivalent of `setclock_gettime`.

Using the setclock Object Routines

Here's an example implementation of the relevant methods of a metronome object using the Setclock routines.

```
typedef struct metro
{
    t_object m_ob;
    long m_interval;
    long m_running;
    void *m_clock;
    t_symbol *m_setclock;
} t_metro;
```

Here's the implementation of the routines for turning the metronome on and off. Assume that in the instance creation function, the t_symbol `m_setclock` has been set to the empty symbol (`gensym ("")`) and `m_clock` has been created; the clock function `metro_tick` is defined further on.

```
void metro_bang(Metro *x)   /* turn metronome on */
{
    x->m_running = 1;
    setclock_delay(x->m_setclock->s_thing,x->m_clock,0);
}
void metro_stop(Metro *x)
{
    x->m_running = 0;
    setclock_unset(x->m_setclock->s_thing,x->m_clock);
}
```

Here is the implementation of the clock function `metro_tick` that runs periodically.

```
void metro_tick(Metro *x)
{
    outlet_bang(x->m_ob.o_outlet);
    if (x->m_running)
        setclock_delay(x->m_setclock->s_thing,x->m_clock,
x->m_interval);
}
```

Finally, here is an implementation of the method to respond to the clock message. Note that the function tries to verify that a non-zero value bound to the `t_symbol` passed as an argument is in fact an instance of **setclock** by checking to see if it responds to the unset message. If not, the metronome refuses to assign the t_symbol to its internal `m_setclock` field.

```
void metro_clock(Metro *x, t_symbol *s)
{
    void *old = x->m_setclock->s_thing;
    void *c = 0;

/*  the line below can be restated as:
    if s is the empty symbol
    or s->s_thing is zero
    or s->s_thing is non-zero and a setclock object
*/
    if ((s == gensym("")) || ((c = s->s_thing) &&
zgetfn(c,&s_unset)))
    {
        if (old)
            setclock_unset(old,x->m_clock);
        x->m_setclock = s;
        if (x->m_running)
            setclock_delay(c,x->m_clock,0L);
    }
}
```

Creating Schedulers

If you want to schedule events independently of the time of the global Max scheduler, you can create your own scheduler with `scheduler_new`. By calling `scheduler_set` with the newly created scheduler, calls to `clock_new` will create Clocks tied to your scheduler instead of Max's global one. You can then control the time of the scheduler (using `scheduler_settime`) as well as when it executes clock functions (using `scheduler_run`). This is a more general facility than the setclock object routines, but unlike using the time from a setclock object to determine when a Clock function runs, once a Clock is tied to a scheduler, it CreatingCreatingCreating. By calling scheduler_set with the newly created scheduler, calls to clock_new will create Clocks tied to your scheduler instead of Max's global one. You can then control the time of the scheduler (using scheduler_settime) as well

scheduler_new

Use `scheduler_new` to create a new local scheduler.

```
Void *scheduler_new(void);
```

This call returns a pointer to the newly created scheduler.

scheduler_set

Use `scheduler_set` to make as when it executes clock functions (using scheduler_run).

```
void *scheduler_set(t_scheduler *scheduler);
```

scheduler	The scheduler to make current.

Make a scheduler current, so that future related calls (such as `clock_delay`) will affect the appropriate scheduler. This routine returns a pointer to the previously current scheduler, which should be saved and restored when local scheduling is complete.

scheduler_run

Use `scheduler_run` to run scheduler events to a selected time.

```
void scheduler_run(t_scheduler *scheduler, double
until);
```

scheduler	The scheduler to advance.
until	The ending time for this run (in milliseconds).

scheduler_settime

Use `scheduler_settime` to set the current time of the scheduler.

```
void scheduler_settime(t_scheduler *scheduler,
double time);
```

scheduler	The scheduler to set.

time	The new current time for the selected scheduler (in milliseconds).

scheduler_gettime

Use `scheduler_gettime` to retrieve the current time of the selected scheduler.

```
void scheduler_gettime(t_scheduler *scheduler,
double *time);
```

scheduler	The scheduler to query.
time	The current time of the selected scheduler.

Operating System Access Routines

When creating external objects, you may need to have access to operating system functions and data elements. The operating system access routines provide a Max-safe method for access to this information.

event_process

Use `event_process` to send events to the operating system for processing.

```
void event_process(void *event, t_wind *win);
```

event	An event structure (EventRecord on Mac OS) to be processed.
win	The window context in which to process the record.

You might use `event_process` when implementing a filter proc for a dialog box; Max can handle the event and do things such as redraw windows if your dialog box moves.

event_run

Use `event_run` to run Max's global event loop.

```
void event_run(void);
```

Quittasks

Sometimes it is necessary to be notified when a program is quitting so that your code can perform clean up of global resources or some other task. Max now has a mechanism to support such a thing, which is called a "quittask". A quittask may be installed with a function pointer and possible pointer argument that will be passed to the quittask function when Max is quitting.

quittask_install

Use `quittask_install` to send events to the operating system for processing.

```
void quittask_install(method *m, void *a);
```

m	A function pointer.
a	Optional arguments (pass 0L for none).

Installs a quittask with the given function pointer and arguments which will be passed to the function when Max quits.

quittask_remove

Use `quittask_remove` to remove an existing quittask.

```
void quittask_remove(method *m);
```

m	A function pointer.

Removes a previously-installed quittask assicoated with the given function pointer.

Chapter 10: Objects With Windows

Max allows external objects to create their own windows and handle operating system events. Generally, these tools simplify the task of writing a user interface and they're a bit simpler to manage than writing user interface objects that exist within Patcher windows.

If your window will be an "editor" for the data in a Max object, you should open it when your object receives a dblclick message (Max doesn't send normal objects a single click message).

Max will send messages to your external object in response to various operating system events. These message are described in detail later in this chapter. These messages work consistently on both Macintosh and Windows XP platforms and thus help develop cross platform externals with windows.

On the Macintosh your object will be able to respond to window messages because Max will install a reference to it inside each Macintosh window record. When Max detects a Macintosh event in the window, it sends the appropriate message to the object that "owns" the window.

On Windows XP you have two choices for working with a window. Using the QuickTime SDK for Windows you can draw using a Macintosh style QuickDraw API. Alternatively, you could use native WIN32 API calls by retrieving the HWND for the window from max. For maximum flexibility you could use this HWND to subclass the WNDPROC to override the default behavior that max provides for you. Some details on how to do this will be provided towards the end of this chapter.

There are two things you'll need to know in order to work with objects that own their own windows in Max.

- A set of Max functions you'll use to allow your window to exist within the Max world.

- A set of special window messages your object will be sent while its window is open. You write methods to respond to these messages and perform actions like drawing the contents of the window or handling a mouse click. At initialization time, use `addmess` to install these methods in your class. All of the window methods should use the special argument list…

```
A_CANT, 0
```

…that specifies that Max can't and shouldn't type check the arguments of the message (because they're not passed as type-checkable t_atoms).

Note that window messages will never be sent to your object at interrupt level.

The basic window structure is called a `t_wind`. You'll create one of these in your creation function (or whenever you want a window to open) by calling `wind_new`. The `t_wind` structure definition and the flags to pass to `wind_new` are declared in the include file *ext_wind.h*. After the window is created and made visible, it will cause messages to be sent to its owning Max object.

The messages sent by the window system are listed on the following pages. In each case, you should frame any action inside calls to set the current GrafPort to your window, and restore it using when done drawing. On the Macintosh use `wind_setport` to set the port and `wind_restoreport` to restore the old port. On Windows XP, using the QuickTime SDK, use the max exported function `XQT_wind_setport` to set the port and `XQT_SetPort` as linked from qtmlclient.lib to restore the port.

The examples below assume a pointer to the `t_windpub` returned from calling `wind_new` is stored in the object's `m_wind` field.

On the Macintosh, or when using native WIN32 drawing routines:

```
GrafPtr sp;
if (sp = wind_setport(myobject->m_wind)) {
    /* draw something here */
    wind_restoreport(sp);
}
```

On Windows XP, using the QuickTime SDK:

```
        if (sp = XQT_wind_setport(myobject->m_wind)) {
            /* draw something here */
            XQT_SetPort(sp);
        }
```

Window Messages

The following messages are available for implementing a window for a Max object. They are presented (somewhat) in order of importance.

click

The click message is sent when a mouse-down event occurs in your window.

BINDING

```
addmess (myobject_click, "click", A_CANT, 0);
```

DECLARATION

```
void myobject_click (t_myobject *x, Point pt, short
dblClick, short modifiers);
```

`pt`	Location of the mouse click in local coordinates.
`dblClick`	Non-zero if this is a double-click, zero otherwise.
`modifiers`	The modifers field of the Mac OS EventRecord returned by GetNextEvent for this mouse down event, indicating whether the shift, option, command, caps lock, or control keys were pressed.

Your click method should handle a mouse down event at the specified location. The function `wind_defaultscroll` helps your object handle events involving scroll bars, and `wind_drag` should be used to follow the dragging action of the mouse. These two functions are described below.

update

The update message is sent when your window should be redrawn.

BINDING

```
addmess (myobject_update, "update", A_CANT, 0);
```

DECLARATION

```
void myobject_update (t_myobject *x);
```

This message indicates that you need to respond to an update event by drawing the contents of your window. Note that Max takes care of drawing the scroll bars (Max will call DrawControls on your window) and grow icon of your window if it contains those items. Also, the affected area of the window will have been erased for you.

key

The key message is sent when the user presses a key and your window is frontmost.

BINDING

```
addmess (myobject_key, "key", A_CANT, 0);
```

DECLARATION

```
void myobject_key (t_myobject *x, short key, short
modifiers        short keycode);
```

`key`	The ASCII code of the key pressed.
`modifiers`	The modifers field of the Macintosh EventRecord returned by GetNextEvent for this key event, indicating whether the shift, option, command, caps lock, or control keys were pressed.
`keycode`	The Macintosh key code of the key pressed.

This message allows you to respond to key down or auto-key event. Note that on Windows XP key events are translated to the Macintosh modifiers and keycode. Refer to the relevant Macintosh developer documentation for details.

cursor

The cursor message allows you to adjust the cursor or display of your window to reflect the current location of the mouse while it is over your window.

BINDING

```
addmess (myobject_cursor, "cursor", A_CANT, 0);
```

DECLARATION

```
void myobject_cursor (t_myobject *x, Point pt,
short modifiers, short active);
```

`pt`	Current location of the mouse in local coordinates.
`modifiers`	The modifers field of the Macintosh EventRecord returned by GetNextEvent for this key event, indicating whether the shift, option, command, caps lock, or control keys were pressed.
`active`	Zero if your window is not the active (frontmost) window, non-zero if it is.

Your cursor method is called repeatedly while the cursor happens to be over your window. The location of the mouse is passed in local coordinates in `pt`, and you are reported the state of the modifier keys and the active/non-active status of the window via `modifiers` and `active`, respectively. Typically, windows will use the cursor method to adjust the cursor when it falls over specific locations, such as TextEdit fields. You can set the cursor with `wind_setcursor`.

The Patcher also uses its cursor method to call TEIdle if text is being edited inside its window, and to highlight inlets and outlets if the mouse is over them. Note that your cursor function is also called during the time the mouse is down and you've called `wind_drag`.

cursorout

The cursorout message allows you to adjust the cursor or display of your window in response to the cursor leaving your window's bounding rectangle.

BINDING

```
addmess (myobject_cursor, "cursorout", A_CANT, 0);
```

DECLARATION

```
void myobject_cursorout (t_myobject *x, Point pt,
short modifiers, short active);
```

`pt`	Location of the mouse in local coordinates where it first left your window.
`modifiers`	The modifers field of the Macintosh EventRecord returned by GetNextEvent for this key event, indicating whether the shift, option, command, caps lock, or control keys were pressed.
`active`	Zero if your window is not the active (frontmost) window, non-zero if it is.

Your cursorout method is called once when the cursor leaves from being over your window. The location of the mouse is passed in local coordinates in `pt`, and you are reported the state of the modifier keys and the active/non-active status of the window via `modifiers` and `active`, respectively. Typically, windows will use the cursorout method to adjust the cursor when it leaves your window's bounding rect. You can set the cursor with `wind_setcursor`.

widle

The widle message lets you know if the cursor has moved over your object's window at idle time.

BINDING

```
addmess (myobject_widle, "widle", A_CANT, 0);
```

DECLARATION

```
void myobject_widle (t_myobject *x);
```

I have no idea why you'd want to use this if you bind a message to the cursor message.

activate

The activate message allows you to change the appearance of your window when it becomes the frontmost window, or when it is no longer the frontmost window.

BINDING

```
addmess (myobject_activate, "activate", A_CANT, 0);
```

DECLARATION

```
void myobject_activate (t_myobject *x, short
active);
```

`active`	Non-zero if the window is becoming active, zero if it is being deactivated.

Typically you'll respond to an activate event by highlighting or unhighlighting something that's selected, according to the value of active. Max takes care of enabling and disabling scroll bars.

close

The close message is sent when your window should be closed.

BINDING

```
addmess (myobject_close, "close", A_CANT, 0);
```

DECLARATION

```
void myobject_close (t_myobject *x);
```

This message is sent to your object when the user wants to close your window. It's suggested that you write your object so that closing the window will not destroy your object's data, so the user can close the window without worrying about losing anything. If you plan on creating another window with `wind_new` at a later time, dispose of the window's memory by calling `freeobject` (which will perform a CloseWindow). Otherwise, you should use syswindow_hide on the t_syswind as follows:

```
syswindow_hide(wind_syswind(myobject->m_wind)));
```

Max just tells you that the user wants to close your window. You can respond in any manner you like, although if you leave the window open, the user may experience problems when quitting.

scroll

The scroll message is sent when a scroll bar is moved and your window's contents should scroll.

BINDING

```
addmess (myobject_scroll, "scroll", A_CANT, 0);
```

DECLARATION

```
void myobject_scroll (myObject *x);
```

Max calls this routine when the user is moving a scroll bar. You should check the `w_xoffset` (horizontal) and `w_yoffset` (vertical) fields of your `t_wind`, compare against your stored previous values, and scroll the window accordingly.

vis and invis

The vis message is sent when your window has just become visible. The invis message is sent when your window is just about to become invisible.

BINDING

```
addmess (myobject_vis, "vis", A_CANT, 0);addmess
(myobject_invis, "invis", A_CANT, 0);
```

DECLARATION

```
void myobject_vis (t_myobject *x);void
myobject_invis (t_myobject *x);
```

You can perform any appropriate action in response to these messages, such as initializing the window's user interface when receiving the vis message.

oksize

The oksize message is sent to confirm a new size for your window.

BINDING

```
addmess (myobject_oksize, "oksize", A_CANT, 0);
```

DECLARATION

```
void myObject_oksize (t_myobject *x, short *hsize,
short *vsize);
```

`hsize`	The proposed horizontal size of the window. If you wish to modify the horizontal size, return a new value in `hsize`, otherwise leave it unchanged.

`vsize` The proposed vertical size of the window. If you wish to modify the vertical size, return a new value in `vsize`, otherwise leave it unchanged.

You can implement an oksize method for your window that will allow you to check and possibly adjust the size of a window before it is actually resized (see the wsize message below). The proposed size is passed in `hSize` and `vSize`. You can set these values to whatever you like. You might use the oksize message if your window contains “cells” and you want the size of the window to be an exact multiple of the number of cells. Obviously, the oksize message is sent before the wsize message described below.

wsize

The wsize message is sent when your window has changed size.

BINDING

```
addmess (myobject_wsize, "wsize", A_CANT, 0);
```

DECLARATION

```
void myObject_wsize (t_myobject *x, short hsize,
short vsize);
```

`hsize` The new horizontal size of the window.

`vsize` The new vertical size of the window.

If the user resizes a window, you’ll be informed via the wsize message. The new dimensions of the window are passed in `hSize` and `vSize`. If your window has scroll bars, you will need to move, resize, redraw them here.

otclick

The otclick message is sent when a user option-clicks on the title bar of your window.

BINDING

```
addmess (myobject_otclick, "otclick", A_CANT, 0);
```

DECLARATION

```
void myObject_otclick (MyObject *x);
```

otclick stands for Option-Title-Click. As an example of an object that implements this method, the Patcher window has a pop-up menu that allows you to return to the parent window of a subpatch window. If you don't implement an otclick method, the user will get the normal dragging action that they'd expect when option-clicking on a title bar.

mouseup

The mouseup message is sent when there is a mouse up event in your window.

BINDING

```
addmess (myobject_mouseup, "mouseup", A_CANT, 0);
```

DECLARATION

```
void myobject_mouseup (t_myobject *x, Point where,
short modifiers);
```

`where`	The location of the mouse up event in local coordinates.
`modifiers`	The modifers field of the EventRecord returned by GetNextEvent for this mouse up event, indicating whether the shift, option, command, caps lock, or control keys were pressed.

Menu Messages

These messages are sent to your window when the user chooses an item from a menu and your window is the active window.

chkmenu

The chkmenu message is sent immediately before the menus are drawn when there has been an event that will cause them to be drawn or used.

The chkmenu message is sent in order to allow you to specify standard menu items you would like enabled or disabled.

BINDING

```
addmess (myobject_chkmenu, "chkmenu", A_CANT, 0);
```

DECLARATION

```
void myobject_chkmenu (t_myobject *x, t_menuinfo
*mi);
```

`mi`	A t_menuinfo structure that you will fill in with those items that should be enabled. All items are disabled by default. See below for a description of this structure.

When your window becomes the active window, Max will update its menus based on those items your window could respond to. You are passed a pointer to a Menuinfo structure initialized to all zero values. If you wish to be sent a message when the user chooses a particular menu item, set the corresponding item in the Menuinfo structure to 1, and the item will be enabled. The Menuinfo structure is an array of short integers and is declared in *ext_menu.h*. Here are the menu commands available to you:

Field	*Message*	*Commands*
i_cut	cut, copy, clr	Cut, Copy, and Clear–Edit menu
i_paste	paste	Paste—Edit menu
i_dup	dup	Duplicate—Edit menu
i_save	saveto	Save and Save As…—File menu
i_pastepic	pastepic	Paste Picture—Edit menu
i_edit	edit	Edit—View menu
i_range	dialog	Get Info…—Object menu
i_lineup	lineup	Align—Max menu
i_fixwidth	fixwidth	Fix Width—Object menu
i_size	font	A Size in the Font menu
i_hide	hide	Hide on Lock—Object menu
i_show	show	Show on Lock—Object menu
i_selectall	selectall	Select All—Edit menu
i_find	find	Find…—Edit menu
i_findagain	find	Find Again—Edit menu

Field	*Message*	*Commands*
i_replace	find	Replace—Edit menu
i_print	print	Print…—File menu
i_font	font	A Font in the Font menu
i_color	wcolor	Color… —Object menu
i_savecoll	savecoll	Save As Collective…—File menu
i_noclick	noclick	Ignore Click—Object menu
i_respond	respondtoclick	Respond to Click—Object menu
i_front	bfront	Bring to Front—Object menu
i_back	back	Send to Back—Object menu
i_lockbg	lockbg	Lock Background—View menu
i_showbg	showbg	Show Background—View menu
i_includebg	includebg	Include in Background—Object menu
i_excludebg	excludebg	Rm. from Background–Object menu
i_coloritem	N/A	current color item—Object menu
i_setorigin	setorigin	Set Origin—View menu
i_restoreorigin	restoreorigin	Restore Origin—View menu
i_name	wobjectname	Name…—Object menu
i_showconnections	showconnections	Show/Hide Connections—View menu
i_showpal menu	showpalette	Show/Hide Object Palette—View
i_pastereplace	pastereplace	Paste Replace—Edit menu

Of the menu messages listed above, only the saveto, font, savecoll, showconnections, showpalette, and find messages contain additional arguments beyond the normal pointer to your object. These messages are detailed below along with those menu messages that deserve additional remarks.

undo

The undo message is sent when the user chooses Undo from the Edit menu.

BINDING

```
addmess (myobject_undo, "undo", A_CANT, 0);
```

DECLARATION

```
void myobject_undo (t_myobject *x);
```

In order to receive this message, you need to explicitly enable and set the text of the Undo menu item, which you can do by responding to the undoitem message discussed below. It's your responsibility to keep track, when your object receives this message, if you should perform an undo or a redo—but this shouldn't be hard, since you set the text of the menu item yourself. If you don't implement an undoitem method, the Undo menu item will always be disabled when your window is the active window.

undoitem

The undoitem message allows you to enable the Undo item in the Edit menu and set its text.

BINDING

```
addmess (myobject_undoitem, "undoitem", A_CANT, 0);
```

DECLARATION

```
void myobject_undoitem (t_myobject *x, char *text);
```

`text`	C string that you set to contain the text of the Undo item in the Edit menu. Set the string to be empty (`text[0] = 0`) to disable the Undo item. In that case, the text will be Undo.

pastepic

The pastepic message is sent when the user chooses Paste Picture from the Edit menu.

BINDING

```
addmess (myobject_pastepic, "pastepic", A_CANT, 0);
```

DECLARATION

```
void myobject_pastepic (t_myobject *x);
```

The picture data is still on the clipboard, and it's your responsibility to copy the data (using Mac OS routine GetScrap) and do something with it.

saveto

The saveto message is sent when the user chooses Save or Save As… from the File menu.

BINDING

```
addmess (myobject_saveto, "saveto", A_CANT, 0);
```

DECLARATION

```
void myobject_saveto (t_myobject *x, char
*filename, short path);
```

`filename`	C string containing the name of the file to write your data into. It may need to be created.
`path`	PathID specifying the location of the file.

If the user chooses Save As… from the File menu when your window is the active window, you will receive this message after the user has specified a file name and volume in a standard save file dialog box. The file has been neither opened nor created. You should create, open, write out, and close the file. You might also want to change the title of the window using `wind_settitle`, although this is done automatically unless you set the `WKEEPT` bit in the flags argument passed to `wind_new`.

If the user chooses Save from the File menu, the filename and volume are already known to Max, and your saveto method will be called.

You're not required to implement this method if there is nothing worth saving in your window. In this case, don't enable the saveto message in your chkmenu method.

dialog

The dialog message is sent when the user chooses Get Info… from the Max menu.

BINDING

```
addmess (myobject_dialog, "dialog", A_CANT, 0);
```

DECLARATION

```
void myobject_dialog (t_myobject *x);
```

The dialog method is used in the Patcher window to send the info message to any selected Patcher objects. Your object can respond to the info message, as we've mentioned above, usually by putting up a dialog box to set some internal values.

The dialog message will be sent to you when your window is the active window and the user chooses Get Info… from the Max menu. Your method might put up the same dialog box as your object's info method. Or if your window contains selectable items, the dialog might depend on what's selected.

font

The font message is sent when the user chooses a new font or font size from the Font menu.

BINDING

```
addmess (myobject_font, "font", A_CANT, 0);
```

DECLARATION

```
void myobject_font (myObject *x, short size, short
fontnum);
```

`size`	The new font size if it has been changed, otherwise -1.
`fontnum`	The number of the new font if one has been chosen from the menu, otherwise -1.

Your object will receive this message after the user has changed the window’s default font or font size by choosing from the Size menu. You can determine the current size by checking the `w_fontsize` field of your `t_wind`. The default font is stored in the `w_realfont` field. To get the font and size information directly from the font message, your method should be declared as shown above. If either `fontnum` or `size` is -1, its value has not been changed by the user and you should not change this aspect of the selected or unselected text in your window.

When responding to the chkmenu message, you can specify that a particular font or size be checked when the Font menu is displayed by setting the values of `i_font` and `i_size` to the current values used in your window.

help

The help message is sent when the user chooses Help… from the Max menu.

BINDING

```
addmess (myobject_help, "help", A_CANT, 0);
```

DECLARATION

```
void myobject_help (t_myobject *x);
```

Your help method can respond by opening a help file or any other helpful action.

find

The find message is sent when the user is using the Find Dialog.

BINDING

```
addmess (myobject_find, "find", A_CANT, 0);
```

DECLARATION

```
void myobject_find (myObject *x, void *search, void
*replace);
```

`search`	A Binbuf specifying what to search for in your window. If `search` is 0L, you should only replace.

`replace`	A Binbuf specifying the replacement for the current selection in your window. If `replace` is 0, you should only perform a search.

This message will be sent to your window after the user has clicked Find in the Find Dialog with your window active, or chosen Find Again or Replace from the Edit menu. If you "find" something, you should probably enable the Find Again… item the next time you receive a chkmenu message.

If both `search` and `replace` are non-zero, first replace what's selected in your window, then search again. If you want to change `search` or `replace` to text, use `binbuf_totext`.

okclose

The okclose message is sent before a standard "Save changes before closing?" alert is displayed.

BINDING

```
addmess (myobject_okclose, "okclose", A_CANT, 0);
```

DECLARATION

```
void myobject_okclose (t_myobject *x, char *prompt,
short *result);
```

`prompt`	Modify this C string to contain the text of the "Save changes?" alert if you want to modify its standard appearance.
`result`	A code from 0 to 4 as described below indicating the action that should be taken in closing the window.

Implementing this message allows your object to override the default behavior of putting up this alert when the window's `w_dirty` field is non-zero. You can, for example, change the text of the alert by modifying `prompt`. Or you can return certain values in `result` that will cause the alert to be skipped or for the window not to be closed at all (unless it is being closed because its owning object is being freed). These latter actions would be appropriate if you want to put up your own Save Changes alert.

The allowable values for `result` are:

0 Take normal action (display alert if `w_dirty` is non-zero, otherwise close the window)

1 Same as 0 except that Max is informed that `dialogString` has been changed.

2 Don't put up an alert and clear `w_dirty` (used when custom alert has resulted in a save).

3 Don't put up an alert, leave `w_dirty` unchanged.

4 Act as if the user cancelled (useful when the user cancels out of a custom alert).

print

The print message is sent after the user chooses Print… from the File menu and the standard print dialog has been displayed.

BINDING

```
addmess (myobject_print, "print", A_CANT, 0);
```

DECLARATION

```
void myobject_print (t_myobject *x, THPrint hPrint,
GrafPtr yourPort, short *result);
```

`hPrint`	A standard Mac OS `THPrint` as described in *Inside Macintosh*.
`yourPort`	Your window's GrafPort. You won't draw in this port when printing, but it may be convenient to access port information such as the port rectangle.
`result`	Set this to a non-zero value if you encounter an error while printing. It's set to zero when your method is called.

This message is sent to your window when the user has chosen Print… from the File menu, seen the standard printing dialog and clicked OK. You will receive a print message for every copy of the document the user wants to print.

In this method, you print your window as through it were a "document." If there's an error, set `result` to a non-zero value.

Here is an outline of some simple code that would print out a window on a single page without worrying if the window is too big for the size of the paper. Note the calls to the Printing Manager that are required before and after drawing a page.

```
void myobject_print (myObject *x, THPrint hp, GrafPtr
port, short *res)
{
    TPPrPort printPort;
    Rect printRect;
    printPort = PrOpenDoc(hp,0L,0L);
    SetPort(printPort);
    TextFont(port->txFont);
    TextSize(port->txSize);
    printRect = (**hp).prInfo.rPage;
    PrOpenPage(printPort,0L);
    /* print window here */

    PrClosePage(printPort);
    PrCloseDoc(printPort);
    *res = 0;
}
```

You may want to be nice and add code that checks for command-period being typed. In this case, `result` should be set to a non-zero value, so that additional copies of the document are not printed.

Window Routines

The following functions are for use in conjunction with your object's window. Here is the t_wind structure used by most of these routines.

```
typedef struct wind
{
    t object w_ob           // object header
    short w_x1;             // location of window
    short w_x2;
    short w_y1;
    short w_y2;
    short w_xoffset;        // scroll offsets
    short w_yoffset;
    short w_scrollgrain;    // scroll grain in pixels
    short w_refcount;       // reference count
    char w_vis;             // visible
    char w_titled;          // has a title
    char w_grow;            // has a grow region
    char w_close;           // has a close region
    char w_scrollx;         // has an x scroll region
    char w_scrolly;         // has a y scroll region
    char w_dirty;           // dirty flag (can save)
    char w_scratch;         // no complain on close
    char w_bin;             // binary save
    char w_font;            // text font
    char w_fsize;           // font size
    char w_fontindex;       // old font index field (unused)
    WindowRecord w_wind;    // Mac OS window data (not always
                            // present, need to check w_local)
    short w_vsmax;          // vertical scroll max
    ControlHandle w_vscroll;    // vertical scroll bar
    short w_hsmax;          // horizontal scroll max
    ControlHandle w_hscroll;    // horizontal scroll bar
    void *w_assoc;          // associated object
    void *(*(w_idle))();    // window idle function (unused)
    char w_name[80];        // filename = window title
    short w_vol;            // Path ID file location
    short  w_proc;          // window proc id (0 = normal)
    char w_keeptitle;       // set window title on saveas
    char w_canon;           // slot in canonical list of
                            // locations
    char w_silentgrow;      // don't draw grow icon but allow
                            //  grow
    char w_color;           // try to make color window if you
                            //  can
    char w_bits;            // number of bits (i.e. 2 for
                            //  b&w)
    char w_divscrollx;      // divided horiz scroll bar
    char w_zoom;            // has zoom rect
    short w_realfont;       // real font index
    short w_hsleft;         // left of scroll bar
    Rect w_oldsize;          // internal use
    short w_oldproc;         // internal use
    char w_select;           // always select on click
    char w_frame;            // internal use
    long w_flags;            // internal use
    WindowPtr w_wptr;        // contains pre-existing window or
                             // ptr to w_wind
    long w_local;            // is OS window stored in w_wind?
    Rect w_growbounds;       // optional grow bounds for a
                          // b window
    char w_helper;           // is part of Extras menu
} t_wind;
```

wind_new

Use `wind_new` to make a new window.

```
t_wind *wind_new (void *assoc, short left, short
top, short right, short bottom, short flags);
```

`assoc`	Owning object. This will usually be a pointer to your object. However, if you want your object to have multiple windows, you may wish to create intermediary objects that receive the window messages so you can distinguish which window they're for.
`left`	Left global coordinate of the window. If both `left` and `top` are 0, the window is placed in an ordered "canonical" location relative to other windows.
`top`	Top global coordinate of the window.
`bottom`	Bottom global coordinate of the window.
right	Right global coordinate of the window.
`flags`	A bitmap of constants determining the window's behavior and appearance from the list below.

`#define WVIS`	1	window will be visible
`#define WGROW`	2	has a Grow region
`#define WSCROLLX`	4	horizontal scrollbar
`#define WSCROLLY`	8	vertical scrollbar
`#define WCLOSE`	16	has a close box

`#define WKEEPT`	32	don’t change window title after saving
`#define WSGROW`	64	invisible grow region
`#define WCOLOR`	128	color window
`#define WPATCHPROC`	256	patcher window defproc with extra title gadgets
`#define WSHADOWPROC`	512	window plain box procID used in new object list window
`#define WDIVSCROLLX`	1024	divided horizontal scroll bar (min 140 pixels)
`#define WZOOM`	2048	has a zoom box
`#define WSELECT`	4096	always select on click (disobey All Windows Active)
`#define WFROZEN`	8192	prevent patcher control over this window
`#define WFLOATING`	16384	create a floating window

`wind_new` returns a new t_wind object. The actual Mac OS or Windows window will not be created unless the visible flag `WVIS` is set.

wind_vis

Use `wind_vis` to make a window visible or bring it to the front.

```
void wind_vis (t_wind *window);
```

`window`	Window to make visible.

`wind_vis` makes a window visible. If it’s already visible, `wind_vis` calls SelectWindow to make the window the active window. If you want to make a Mac OS window visible, use `syswindow_show`.

wind_invis

Use `wind_invis` to make a window invisible.

```
void wind_invis (t_wind *window);
```

`window`	Window to make invisible.

`wind_invis` hides the window if it's visible. If the window isn't visible, `wind_invis` does nothing. Note that `wind_invis` destroys the actual OS window object. If you want to defeat the system and keep your window alive but invisible, use `syswindow_hide` on the return value of `wind_syswind` instead. Note that `wind_invis` does not actually get rid of the memory occupied by the Wind structure. After using wind_invis, call `wind_vis` to create another Mac OS or Windows XP window.

wind_setgrowbounds

Use `wind_setgrowbounds` to limit the minimum and maximum bounds of the selected window.

```
void wind_setgrowbounds(t_wind *window, short minx,
short miny, short maxx, short maxy);
```

window	Window to set bounds.
minx	The minimum width of the window.
miny	The minimum height of the window.
maxx	The maximum width of the window.
maxy	The maximum height of the window.

wind_defaultscroll

Use `wind_defaultscroll` to see if a mouse click was on a scrollbar, and if so, handle it in the default manner.

```
void wind_defaultscroll (t_wind *window, Point pt,
short pagesize);
```

`window`	Window in which the mouse was clicked.
`pt`	Location of the mouse click in local coordinates, passed to your click method.
`pagesize`	Value to increment or decrement the scroll bar when the user pages a scroll bar up or down. Paging is clicking on the dotted part of the bar outside the thumb. If you pass 0 for `pageSize`, the default paging routine is used, which goes to the maximum or minimum of the scroll bar's value when the user clicks in the dotted area of the bar.

If your window has scroll bars, call `wind_defaultscroll` in your click method. It will check if `pt` lies within a scroll bar. If so, `wind_defaultscroll` executes the default scroll bar routine and returns 1. If not, `wind_defaultscroll` returns 0.

wind_dirty

Use `wind_dirty` to mark a window as having unsaved data.

```
void wind_dirty (t_wind *window);
```

`window`	Window to dirty.

`wind_dirty` sets the window's dirty bit, so the user will be asked to save changes if the window is closed. Your saveto method will be called if the user wants to save the changes.

wind_drag

Use `wind_drag` to track mouse dragging in a window.

```
void wind_drag (method dragfun, void *arg, Point
start);
```

`dragfun`	Procedure that will handle tracking the cursor and the mouse button. See below for its definition.
`arg`	Argument passed to the drag procedure. Normally this is your object.
`start`	The starting location of the drag. This is usually the Point you receive as an argument to your click method.

Use of `wind_drag` replaces a typical program's loop that usually looks like:

```
do {
    GetMouse(&pt);
    /* do something here to track the mouse*/
} while (StillDown());
```

You pass a pointer to a function (`dragfun`) you want called every time the mouse moves. It will call `dragfun` with the specified argument `arg`, the location of the mouse, and whether the mouse button is down. When the mouse button goes up, your drag function is called one last time. Your drag function should be declared as follows:

```
void myobject_drag (void *dragarg, Point pt, short
button);
```

`dragarg`	Argument passed to wind_drag. Usually it will be a pointer to your object.
`pt`	Current cursor location in local coordinates.
`button`	Non-zero if the mouse button is down, zero otherwise.

As mentioned above, `wind_drag` will normally only call your drag function when the mouse moves. If you want your drag function to be called even if the mouse hasn't moved, call `wind_noworrymove` before calling `wind_drag`. The dragroutine will be called one final time when the mouse button is released (and `button` will be zero). Your drag routine should use `wind_setport` (see below) to ensure that drawing takes place in the correct GrafPort.

wind_inhscroll

Use `wind_inhscroll` to test whether a Point lies within a horizontal scroll bar of a window.

```
short wind_inhscroll (t_wind *window, Point pt);
```

`window`	Window containing the scroll bar(s) to test.
`pt`	Mouse click location.

`wind_inhscroll` returns true if `pt` lies within the horizontal scroll bar and false if it doesn't. This can be used to distinguish a click in the horizontal scrollbar from one in the vertical scrollbar for the purpose of passing a different `pagesize` argument to `wind_defaultscroll`.

wind_noworrymove

Use `wind_noworrymove` to set the next invocation of `wind_drag` to call your drag function even if the cursor hasn't changed.

```
void wind_noworrymove (void);
```

See `wind_drag` for more information.

wind_setcursor

Use `wind_setcursor` to change the cursor.

```
void wind_setcursor (short curs);
```

`curs` One of the following predefined cursors:

```
#define C_ARROW    1#define C_WATCH
  2#define C_IBEAM 3#define C_HAN
  4#define C_CROSS 5#define C_PENCIL
  6#define C_GROW  8
```

`wind_setcursor` keeps track of what the cursor was previously set to, so if something else has changed the cursor, you may not see a new cursor if you set it to the previous argument to `wind_setcursor`. The solution is to call `wind_setcursor(0)` before calling it with the desired cursor constant. Use wind_setcursor(-1) to tell Max you'll set the cursor to your own cursor directly.

wind_setport

Use `wind_setport` to set the current GrafPort to a window.

```
GrafPtr wind_setport (t_wind *window);
```

`window` Window to be made the current GrafPort.

A convenience function that sets the current GrafPort to the port associated with a window. A pointer to the previous GrafPort is returned by the function if successful, otherwise, `wind_setport` returns NIL. You should call this function before drawing or handling events in a window, and call `wind_restoreport` on the result when you're through. Here's an example.

```
GrafPtr sp;
if (sp = wind_setport(myWind)) {
    /* draw things here */
    wind_restoreport(sp);
}
```

Note: If you are using the Quicktime SDK for Windows XP, you should use XQT_wind_setport instead (see below).

wind_restoreport

Use `wind_restoreport` to restore the graphics port to its previous value returned by `wind_setport`.

```
void wind_setport (GrafPtr gp);
```

`gp` Opaque structure returned by `wind_setport`.

This function sets the current `GrafPort` to the port returned by `wind_setport`.

XQT_wind_setport (Windows XP with QuickTime SDK Only)

Use `XQT_wind_setport` to set the current GrafPort to a window.

```
GrafPtr XQT_wind_setport(t_wind *window);
```

`window` Window to be made the current GrafPort.

A convenience function that sets the current QuickTime SDK QuickDraw GrafPort to the port associated with a window. A pointer to the previous GrafPort is returned by the function if successful, otherwise, `XQT_wind_setport` returns NIL. You should call this function before drawing or handling events in a window, and call `XQT_SetPort` on the result when you're through. Here's an example.

```
GrafPtr sp;
if (sp = XQT_wind_setport(myWind)) {
    /* draw things here */
    XQT_SetPort(sp);
}
```

wind_syswind

Use `wind_syswind` to retrieve a `t_syswind` from a `t_wind`.

```
t_syswind *wind_syswind(t_wind *window);
```

`window`	Window to query.

See the section below on the Syswindow API for uses of the returned `t_syswind`.

wind_gethwnd (Windows XP Only)

Use `wind_gethwnd` to get the Windows-specific HWND window handle associated with a Max `t_wind` when writing externals using Windows-specific functions.

```
HWND wind_gethwnd (t_wind *window);
```

`window`	t_wind to query.

You can use the returned HWND in any WIN32 API function call that requires the HWND for your window.

wind_fromhwnd (Windows XP Only)

Use `wind_fromhwind` to get the Max `t_wind` structure associated with a Windows-specific HWND when writing externals using Windows-specific functions.

```
t_wind *wind_fromhwnd (HWND window);
```

`window`	Windows window handle to query.

If no max t_wind is associated with the given HWND the return value is NULL.

wind_installidletask

Use `wind_installidletask` to install an idle-time task for your window.

```
void *wind_installidletask (t_wind *w, method m,
void *myobj, long flags);
```

`w`	Window associated with the idle task.
`method`	method that will handle the task.
`myobj`	Pointer to your object's instance.
`flags`	Options for when the idle task is called. The flags argument can be comprised of the following options:

```
#define IDLETASK_IDLE    1#define
IDLETASK_CURSOR    2
```

`wind_installidletask` installs an idle-time task for your object's window, and returns a pointer to the task. The task calls the method you provide either in the widle function or in the window's cursor function, or both, as indicated by `flags`. If you are calling the task in the window's cursor function, the task function will also receive a `Point` argument. You will probably want to call this routine in your object's instance creation function, and remove the task with `wind_removeidletask`, in your object's free function.

wind_removeidletask

Use `wind_removeidletask` to remove an idle-time task that you installed with `wind_installidletask`.

```
void *wind_removeidletask (t_wind *w, void *task);
```

`task`	The task pointer returned by `wind_installidletask`.

`wind_removeidletask` removes the task you pass it..

wind_setsmax

Use `wind_setsmax` to set the maximum values of a window's scrollbars.

```
void wind_setsmax (t_wind *window, short hmax,
short vmax);
```

`window`	Window containing the scroll bars.
`hmax`	New maximum for the horizontal scroll bar.
`vmax`	New maximum for the vertical scroll bar.

The *minimum* values of the scroll bars are always 0. This function can be used whether or not the window is visible.

wind_setsval

Use `wind_setsval` to set the values of a window's scroll bars.

```
void wind_setsval (t_wind *window, short hval,
short vval);
```

`window`	Window containing the scroll bars.
`hval`	New value for the horizontal scroll bar.
`vval`	New value for the vertical scroll bar.

This function can be used whether or not the window is visible.

wind_settitle

Use `wind_settitle` to change a window's title.

```
void wind_settitle (t_wind *window, char *title,
short brackets);
```

`window`	Window containing the scroll bars.
`title`	C string containing the new title.
`brackets`	If non-zero, the title will appear within square brackets.

This function can be used whether or not the window is visible.

wind_setundo

Use `wind_setundo` to set the text of the Undo item in the Edit menu.

```
void wind_setundo (char *string, short undoable);
```

`string`	New text of the Undo item (C string).
`undoable`	If non-zero, the Undo item is enabled, otherwise it's disabled.

wind_filename

Use `wind_filename` to change title and filename stored with the window.

```
void wind_filename (t_wind *window, char *filename,
short path, short bin);
```

`window`	Window whose filename is to be changed.
`filename`	The new filename (C string).
`path`	The new Path ID specifying the file's location.
`bin`	The new default setting of the file format. If `bin` is 0, Text is selected in the Save As dialog; if `bin` is 2, Normal is selected. A `bin` value of 1 was used for "Old Format" binary files in Max -- this format is no longer supported. If `bin` is -1, the choice of file formats is not presented to the user in the Save As dialog.

This function changes the title of your window and gives it a filename and a volume that is automatically passed as an argument to the saveto message if the user chooses Save from the File menu.

wind_setbin

Use `wind_setbin` to change the file format setting of a window.

```
void wind_setbin (t_wind *queenie, short way);
```

`queenie`	Window whose file format setting is to be changed.

`way`	The new default setting of the file format. If `way` is 0, Text is selected in the Save As dialog; if `way` is 2, Normal is selected. A `bin` value of 1 was used for "Old Format" binary files in Max - this format is no longer supported. If `way` is -1, the choice of file formats is not presented to the user in the Save As dialog.

wind_close

Use `wind_close` to begin the process of closing a window.

```
short wind_close (t_wind *window);
```

`window`	Window to be closed.

Normally this function's actions are performed when the user clicks the close box or chooses Close from the File menu. `wind_close` first checks if the window's data has been changed. If it has, and the window's `w_scratch` field has not been set, the Save Changes dialog is presented and the desired action is taken. If `wind_close` returns -1, it means the user cancelled out of the Save Changes dialog or the file saving dialog and the window was not closed.

If `wind_close` returns 0, it means that the window was closed. This happens in three circumstances. First, when there was no data to save. Second, when the user specified that changes were to be saved and they were properly saved. Third, when the user specified that changes were not to be saved.

This function might be used in conjunction with `wind_nocancel` described below if you want to ask the user whether to save data in a window owned by your object when your object's free function is called.

wind_nocancel

Use `wind_nocancel` before `wind_close` to eliminate the possibility of the user being able to cancel out of a "Save changes?" dialog before a window is closed.

```
short wind_nocancel (void);
```

`wind_nocancel` only affects the "Save changes?" dialog that appears immediately after it has been called.

Syswindow API

The Syswindow API provides a number of utility functions for managing windows. It is relatively similar to some of the Macintosh Window Manager API. On Mac OS9 a `t_syswind` pointer was actually a Macintosh `WindowPtr`, but on both Mac OSX and Windows XP, it is an opaque type. This means that it is no longer safe to mix these routines with other window routines. For a Max `t_wind` object, the associated `t_syswind` value can be obtained by calling the `wind_syswind` function.

syswindow_inlist

Use `syswindow_inlist` to find if a window is in the Max window list.

```
Boolean syswindow_inlist (t_syswind *wnd);
```

`wnd`	A pointer to a `t_syswind` struct.

Returns TRUE if the window is in the Max window list, otherwise FALSE.

syswindow_isvisible

Use `syswindow_isvisible` to find if a window is in the Max window list.

```
Boolean syswindow_isvisible (t_syswind *wnd);
```

`wnd`	A pointer to a `t_syswind` struct.

Returns TRUE if the window is visible, otherwise FALSE.

syswindow_isfloating

Use `syswindow_isfloating` to find if a window is in the Max window list.

```
Boolean syswindow_isfloating (t_syswind *wnd);
```

`wnd`	A pointer to a `t_syswind` struct.

Returns TRUE if the window is a floating window, otherwise FALSE.

syswindow_hide

Use `syswindow_hide` to hide a window.

```
void syswindow_hide (t_syswind *wnd);
```

`wnd`	A pointer to a `t_syswind` struct.

Hides the specified window.

syswindow_show

Use `syswindow_show` to show a hidden window.

```
void syswindow_show (t_syswind *wnd);
```

`wnd`	A pointer to a `t_syswind` struct.

Shows the specified window.

syswindow_move

Use sys`window_move` to move the selected window.

```
void syswindow_move(t_syswind *wnd, long x, long y,
Boolean front);
```

`wnd`	Window to move.
`x`	Horizonatal screen position.
`y`	Vertical screen position.
`front`	Flag to bring window to the front.

This function moves the given window to the screen location defined by the (x, y) pixel coordinates. The window may be brought to the front if the front flag is non-zero. Note that on Windows XP, the window will be repositioned in relation to the Max application's MDI frame window client area, unless it is a floating window, in which case it is positioned according to screen coordinates.

syswindow_size

Use `syswindow_size` to resize a window.

```
void syswindow_size (t_syswind *wnd, long width,
long height, Boolean update);
```

`wnd`	Window to resize.
`width`	New window width.
`height`	New window height.
`update`	Flag to update the window.

This function resizes the given window to the height and width given in pixels. The window is subsequently updated if the front flag is non-zero.

syswindow_wind

Use `syswindow_wind` to retrieve the t_wind structure associated with a t_syswind pointer.

```
struct wind *syswindow_wind(t_syswind *wnd);
```

`wnd`	Window to query.

Returns a pointer to the Max t_wind object associated with t_syswind pointer.

Subclassing the Window (Windows XP Only)

On Windows XP max provides the implementation of the WNDPROC to handle all of the basic functionality. However, you may want to receive messages that max does not handle or you may want to change the default implementation that max provides. You can do this by subclassing the window.   A complete description of how to subclass the window is beyond the scope of this document. However, we will provide some guidelines.

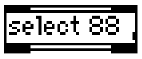
- To subclass the window use the WIN32 API `SetWindowLong()` function with the `GWL_WNDPROC` index. Make sure you store the prior `WNDPROC` and use the WIN32 API `CallWindowProc()`

function to pass appropriate messages on for default handling by max.

- Note that `SetWindowLong()` will need the `HWND` of your window. This `HWND` does not exist until the window has been made visible by the `wind_vis()` call, or immediately after `wind_new()` with the `WVIS` flag. Once the `HWND` does exist you can get it by calling `wind_gethwnd().`

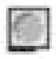
Chapter11: Writing User Interface Objects

Thus far, we've talked only about writing normal external objects, ones that show up in boxes with two lines at the top and bottom.

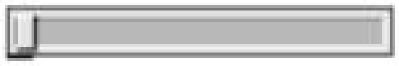


Now you'll learn the secrets to writing external objects that have a custom appearance and behavior in a Patcher window.

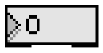
The normal object is just one of several objects you have to choose from in the Patcher window's palette. Others include buttons…



…sliders…



… and number boxes.



These objects have inlets and outlets just like normal objects, but they present a friendlier face to the world. Make sure before writing a custom user interface object that your task might not be better suited to a normal object that creates its own window. Your user interface object should enhance what a user can do within a Patcher window.

Windows XP Considerations for UI Objects

This chapter discusses writing user interface objects for Max. This API originated in a Macintosh QuickDraw environment. On Windows XP you can either use native WIN32 drawing or you can use Apple’s QuickTime SDK for Windows which contains an emulation of QuickDraw. Please see the section titled, *XQT and QTML for Windows UI Externals,* at the end of this chapter for important information on using QTML in the Max environment.

The Box

Patcher windows are made up of a collection of t_box structures.

A `t_box` structure contains the user interface object’s enclosing rectangle and a bunch of flags that determine the object’s appearance and behavior. When you create a structure for your user interface object, a Box must be the first thing contained within it. Not a pointer to a t_box—the `t_box` structure itself. This structure replaces the `t_object` that normally begins an external object structure definition. `t_box` and other useful user interface data types and constants are defined in the Max include file *ext_user.h*.

Here’s an example user interface object structure definition. We’ll be using this example throughout our explanation of the various routines that help your write user-interface objects.

```
typedef struct myuserobject {
    t_box my_box;
    long my_data1;
    long my_data2;
    void *my_qelem;
} t_myuserobject;
```

Using this technique, the patcher window is able to treat your user interface object as a `t_box`, even though it contains additional fields, while Max can treat it as an `t_object`.

The SICN

How do you tell Max that your object is a user interface object? You include a SICN resource in the file that contains your shared library. In CodeWarrior, a resource file can be added to your project if its file type is rsrc. In the MPW environment, a resource file (of type rsrc) is run through DeRez to create a text-based resource (.r) file, then rebuild using Rez.

The SICN resource should be an icon of the object's appearance. Take a cue from the size and style of the existing Max object icons (look at the SICN resources in the Max application file in ResEdit) when designing yours. Give the SICN the same resource ID number as the mAxL resource with which it's associated.

In addition, you should name the SICN resource with a brief description of your object, for example, "Horizontal Slider." This description will show up in the Assistance portion of the Patcher window when the user clicks on your SICN in the Patcher palette.

On Windows XP you will need to create this resource and append it to your external object DLL using rezwack from the QuickTime SDK for Windows tools. A future version of the Max SDK will allow the icon to be specified using a Windows ICON resource.

User Interface Object Creation Functions

Remember the `menufun` argument to the `setup` function, called at initialization time? `menufun` is a function that will be called if the user chooses your user interface object from the Patcher window palette. It should create a "default" object in size and appearance, and return a pointer to your newly created object. It is declared as follows:

```
void *myuserobject_menu (t_patcher *p, long left,
long top, long font);
```

`p`	The patcher your object is in.
`left`	Left coordinate where the object should be created.
`top`	Top coordinate where the object should be used.
`font`	Current window's default font size in the lower 16 bits, and the current default font index in the upper 16 bits.

You need only use this information if your object displays text.

User interface objects create two instance creation routines. One, declared above, is called to make a default object and the other is called when an instance of your object is created when a patcher file is being read in. The exact format of the second instance creation routine is somewhat up to you, since your object will respond to the psave message to write out its state and location within a Patcher. Before discussing the psave method, here is how you declare your file-based instance creation routine.

```
void *myuserobject_new (t_symbol *sym, short argc,
            t_atom *argv);
```

`sym`	The name of your object.
`argc`	Count of t_atoms in `argv`.
`argv`	Array of t_atoms that describe your object (coordinates, etc.).

The first t_atom in the argv array points to the Patcher your object is in. You'll need this information in order to initialize the t_box structure that is the header for all user interface objects. Here is what your object should do first in creating your user interface object instance:

```
void *myuserobject_new (t_symbol *sym, short argc, t_atom
*argv)
{
    t_myuserobject *x;
    void *p;      // your patcher
    short left, top, right, bottom;
    x = newobject(myuserobject_class);
    p = argv->a_w.w_obj;        // get patcher out of argv
```

Having gotten your patcher, you should now look at the additional arguments that have been passed to your routine. In particular, we recommend that the next few arguments specify the coordinates of your object's rectangle within the Patcher. We'll show you how to do this in the psave method in a moment, but let's assume that the object we're creating has stored its coordinates in this order: left, top, right, bottom.

Continuing with our myuserobject_new example, we would retrieve the coordinates out of the argv array:

```
left = argv[1]->a_w.w_long;
top = argv[2]->a_w.w_long;
right = argv[3]->a_w.w_long;
bottom = argv[4]->a_w.w_long;
```

Now we're ready to learn about box_new, which initializes a t_box contained in our user-interface object.

box_new

Use box_new to initialize a Box.

```
void box_new (t_box *b, t_patcher *p, short flags,
short left, short top, short right, short bottom);
```

b	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.
p	The patcher argument passed to your creation function.
flags	A combination of constants that control the appearance and behavior of the box, discussed below.
left	Left coordinate of the box; should be the left coordinate passed to your creation function.
top	Top coordinate of the box; should be the top coordinate passed to your creation function.
right	Right coordinate of the box; should be based on the left coordinate and the width passed to your creation function.
bottom	Bottom coordinate of the box; should be based on the top coordinate and the height passed to your creation function.

`box_new` doesn’t allocate memory for a t_box. Instead, it initializes an existing t_box inside your object. The `flags` argument is a combination of the following constants:

`#define F_DRAWFIRSTIN`	1	draw first inlet
`#define F_GROWY`	2	can grow in y direction only
`#define F_NODRAWBOX`	4	don't draw the box
`#define F_MOVING`	8	Object can draw or reveal other objects outside its defined clipping region
`#define F_GROWBOTH`	32	can grow independently in x and y
`#define F_IGNORELOCKCLICK`	64	don’t send a click msg if patcher is locked
`#define F_NOGROW`	128	don’t draw a grow marker because box can’t change size
`#define F_HILITE`	256	highlightable object (for typing into)
`#define F_DRAWINLAST`	512	Draw inlets after box draws (useful for colorized objects)
`#define F_TRANSPARENT`	1024	Allows the creation of transparent objects.
`#define F_SAVVY`	2048	Tells Max any queue function you make calls box_enddraw
`#define F_BACKGROUND`	4096	Immediately set Box into the background after creating object
`#define F_NOFLOATINSPECTOR`	8192	Prevent object from being edited with the floating inspector

The most commonly used flags are `F_DRAWFIRSTIN`, and `F_GROWY` or `F_GROWBOTH`.

Continuing with our example, here is the t_myuserobject call to `box_new`:

```
box_new((t_box *)x, p, F_DRAWFIRSTIN | F_SAVVY, left, top,
right, bottom);
```

After having initialized the Box, you'll want to initialize the other fields of your object. It's required, because of interrupt level considerations, that you use a Qelem for redrawing your object's state in response to messages that can be sent to your object by a user (such as int or bang). Since you can't draw yourself immediately in response to a message, it often helps to keep track of both your object's logical value and its most recently drawn value. This way, you'll know when your object needs to be updated to reflect a new value.

The next step in a user interface creation routine is to create any Outlets or additional Inlets it may need. Next, you must assign the Box's `b_firstin` field to point to your object. Here our example sets up a Qelem and does the required assignment. We'll discuss the `myobject_redraw` function shortly.

```
x->my_qelem = qelem_new(x, (method)myobject_redraw);
x->my_box.b_firstin = (void *)x;
```

Finally, you'll want to call `box_ready` so that the Patcher window knows to draw your newly initialized object.

box_ready

Use `box_ready` to prepare your object to be drawn.

```
void box_ready (t_box *b);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

`box_ready` calculates where to draw your new Box's inlets and outlets. Then, if the new object is being created by the user (rather than being read in from a file), `box_ready` will visually select it in the Patcher window, which is handy if the first thing the user wishes to do is choose Get Info… from the Max menu to set additional parameters, such as the range for a slider-type object.

After calling box_ready, the user interface object's creation routine should return a pointer to its newly created instance.

```
box_ready((t_box *)x);
return x;
```

Now we've discussed the basic steps in writing a user-interface object creation routine. Let's return to the first creation routine we discused, the one called by the Patcher when a default instance of our object is being created. We can implement this routine so that it sets up a proper argv array with our default values, and then calls the creation routine we just wrote. This avoids any duplication in our code.

```
void *myuserobject_menu (t_patcher *p, long  left, long
top, long font)
{
    t_atom argv[20];
    // set up argv the way myuserobject_new wants it…
    // first the patcher
    argv[0].a_type = A_OBJ;
    argv[0].a_w.w_obj = (t_object *)p;
    // now the coordinates (font info not used)
    argv[1].a_type = argv[2].a_type =
        argv[3].a_type = argv[4].a_type = A_LONG;
    argv[1].a_w.w_long = left;
    argv[2].a_w.w_long = top;
    argv[3].a_w.w_long = right;
    argv[4].a_w.w_long = bottom;
    return myuserobject_new(0, 5, argv);
}
```

User Interface Object Free Function

User interface objects must define a free function, and it should call `box_free` to free any data associated with the Box. Do not call `freeobject` on your Box. This will produce an infinite recursion because the user interface object free function has been called within `freeobject`, and calling `freeobject` on the same pointer will call your free function again, which will call `freeobject`, and so on. On the other hand, objects that are *not* user interface objects should never call `box_free`.

After calling `box_free`, your free function should do any other memory disposal or cleanup as you would in the free function for a normal object.

box_free

Use `box_free` to free data structures used by a Box.

```
void box_free (t_box *b)
```

`b`	The Box structure that is the header of your user interface object. You can just pass a pointer to your object here.

This function frees data structures associated with a box, such as patch cords. If the box is visible, it also causes the area of the patcher window underneath the box to be redrawn.

Messages for User Interface Objects

There are three messages you must implement for a user interface object: click, update, and psave. Use `addmess` at initialization time to add methods to respond to these messages. Two optional messages, key and bfont, are useful for objects that display text or numbers.

click

The click message is sent when the user clicks the mouse on your object.

BINDING

```
addmess (myobject_click, "click", A_CANT, 0);
```

DECLARATION

```
void myobject_click (myObject *x, Point pt, short
modifiers);
```

`pt`	Location (in local coordinates) where the mouse was clicked.
`modifiers`	The modifers field of the EventRecord returned by GetNextEvent for this mouse up event, indicating whether the shift, option, command, caps lock, or control keys were pressed.

You'll get this message when the patcher window has detected a click within your Box's rectangle. You should respond by tracking the mouse in some appropriate way. `pt` is in local coordinates and modifiers is the standard Macintosh EventRecord modifiers field. Rather than implement a loop in which you wait for the mouse button to go up, use `wind_drag` to do mouse tracking. Use `patcher_setport` to set the current GrafPort before doing any drawing or mouse tracking.

Important: If you send any messages out your outlets in your click method, you must set the `lockout` flag before doing so, and restore it afterward. Here's an example of a click routine which sends 0 out an outlet:

```
void myObject_click (t_myuserobject *x, Point pt, short
modifiers)
{
    short savelock;
    savelock = lockout_set(1);
    outlet_int (x->m_ob.o_outlet,0);
    lockout_set (savelock);
}
```

update

The update message is sent when your object needs to be redrawn.

BINDING

```
addmess (myobject_update, "update", A_CANT, 0);
```

DECLARATION

```
void myobject_update (t_myuserobject *x);
```

The update method draws your user interface object inside its box. In addition, if the user can grow or shrink your object's box rectangle, the update method is where you should check to see if your object has changed in size. For example, a slider object might want to reconstruct a bit map of its "slider handle" based on a new width. You'll need to store some aspect of the Box's "old" size within your object to make this comparison. Also, you can use the `box_size` function to resize a Box to keep it from getting too small or too large.

If you wish to make a highlightable user interface object, you need to look at the `b_hilited` field of your `t_box` in your update method. If this field is non-zero, you should draw your object in a highlighted state. Otherwise, draw it in a non-highlighted state. In addition, you should implement a method to respond to the key message described below so the user can type into your object. The Number box is an example of a highlightable user interface object.

If you plan on doing any drawing in a routine other than your update method, see the information about the routines patcher_setport and box_nodraw below. Also, if you will support indexed colors for your user interface object, you should refer to the *Color And User Interface Objects* section for information on the color and wcolor messages.

psave

The psave message is sent when a patcher is saving your object, either because it's being copied to the clipboard or because it's being saved in a file.

BINDING

```
addmess (myobject_psave, "psave", A_CANT, 0);
```

DECLARATION

```
void myobject_psave (t_myuserobject *x, Binbuf
*dest);
```

`dest`	The Binbuf where you should write out a message to save your object.

Writing the psave method is probably the most arcane thing you'll have to do in writing your user interface object. You can use either `binbuf_vinsert` (described in Chapter 7) or `binbuf_insert` to create a message that can recreate your object when a Max file is opened. First, you need to determine whether your object has been set to be hidden by checking your `t_box`'s `b_hidden` field. The user can hide any Box or Patchline by choosing Hide on Lock from the Max menu.

Then you'll save your object's internal data in a manner something like the example below. The only things you should change from the example presented below involve the additional arguments you save, along with the format string passed to `binbuf_vinsert`.

You should also know the name of your object's class. In the example below, our object is called **myuserobject**. Assume that we're saving a user interface object with the following structure and we'd like to save the my_range and my_value fields along with the coordinates of our object.

```
typedef struct myuserobject {
    t_box my_box;
    long my_range;
    long my_value;
} t_myuserobject;
```

Here's an appropriate psave method.

```
void myObject_psave(myObject *x, void *w)
{
    short hidden;
    Rect *r;
    hidden = myObj.my_box.b_hidden;
    r = &myObj.my_box.b_rect;
    if (hidden) {
        binbuf_vinsert
(w,"sssslllll",gensym("#P"),gensym("hidden"),
    gensym("user"),gensym("myobject"), (long)(r->left),
        (long)(r->top), (long)(r->right - r->left),
                   (long)(r->bottom - r->top), (long)(x-
>my_range));
    } else {
        binbuf_vinsert(w,"ssslllll",
gensym("#P"),gensym("user"),           gensym("myobject"),
               (long)(r->left),(long)(r->top),
    (long)(r->right - r->left),
    (long)(r->bottom - r->top),
                        (long)(x->my_range));
    }
}
```

We'll save the coordinates first in a form that can be used by the myuserobject_new function we defined earlier: first the left, then the top, etc.

A few points of explanation. hidden is a special keyword that tells Max to create the Box with its `b_hidden` attribute set. This will be done automatically for you when a Max file is opened. user is a special keyword that is required for an externally written user interface object. If you forget to include user, your object will is likely to generate an error something like…

```
patcher: doesn't understand 'myobject'
```

…when it is loaded from a file.

The rest of the arguments to `binbuf_vinsert` are the coordinates of your object's Box and other information you want to save.

If we were view as text the result of executing the psave method shown above, it would look something like this:

```
#P user myobject 200 200 18 18 128;
```

bfont

The bfont message is sent when your object is selected and the user chooses a new font or font size from the Font menu.

BINDING

```
addmess (myobject_bfont, "bfont", A_CANT, 0);
```

DECLARATION

```
void myobject_bfont (myObject *x, short size, short
font);
```

`size`	The new size chosen by the user, or -1 if the font size has not changed.
`font`	The number of the new font chosen by the user, or -1 if the font has not changed.

If you want your user interface object's Box to change its size or appearance when it is selected and the user choose a new font or font size from the Font menu, you should implement a bfont method that will be called after such an action is performed. **IncDec** and **umenu** are examples of Max user interface objects which implement this method.

key

The key message is sent when your object is highlighted and the user presses a key.

BINDING

```
addmess (myobject_key, "key", A_CANT, 0);
```

DECLARATION

```
void myobject_key (myObject *x, short keyvalue);
```

`keyvalue`	The ASCII value of the key pressed.

User interface objects that set the F_HILITE bit in the flags argument to box_new will receive the key message when the user highlights the object and presses a key on the Macintosh keyboard. Your object should respond by changing its state appropriate to the key's ASCII value passed in `keyvalue` and redrawing itself.

enter

The enter message is sent when the user does something to indicate that the text typed into your object should be "entered" as permanent.

BINDING

```
addmess (myobject_enter, "enter", A_CANT, 0);
```

DECLARATION

```
void myobject_enter (myObject *x);
```

This message is sent when the user presses the Return or Enter keys, highlights another box, or clicks outside your box in the patcher window. In response to this message, you should eliminate anything in your display that appears to be "pending" (such as the three dots shown by the Number box) and accept the typed-in value.

clipregion

The clipregion message is sent when the Patcher wants to know how your object should overlap other objects. If you don't implement this method, it's assumed that your object fills its rectangle.

BINDING

```
addmess (myobject_clipregion, "clipregion", A_CANT,
0);
```

DECLARATION

```
void myobject_clipregion (t_myobject *x, RgnHandle
*rgn, short *result);
```

This message allows your object to define a non-rectangular subset of its rectangle as its drawing area. For instance, the **umenu** object displays the current menu item in a rounded-corner rectangle. You set the result parameter to one of four values:

- `CLIPRGN_RGN` indicates you have set the `rgn` parameter to a Mac OS region (an example is shown below) and that the object draws only within the defined region.

- `CLIPRGN_RECT` means your object will fill its entire rectangle. In this case you do not modify the `rgn` parameter.

- `CLIPRGN_EMPTY` means your object isn't going to draw anything. Transparent objects such as **dropfile** and **ubutton** could use this setting. The `rgn` parameter should not be modified.

- `CLIPRGN_COMPLEX` means that your object's clipping region is so complicated it isn't worth definining as a region. Instead, the Patcher should always make your object draw on top of any object underneath it. The **comment** object, which draws text transparently on top of other objects, uses this setting. The `rgn` parameter should not be modified.

Here's an example of an object that defines a circular clipping region in its clipregion method.

```
void myobject_clipregion (t_myobject *x, RgnHandle
*rgn,short *result)
{
    *result = CLIPRGN_RGN;
    OpenRgn();
    FrameOval(&x->my_box.b_rect);   // use box's rect
    CloseRgn(*rgn = NewRgn());
}
```

For more information on the clipregion method, see the Transparent Objects section below.

Note also that we currently do not have a way to specify a region for UI objects which is not Macintosh-specific. Therefore the CLIPRGN_RGN mode is currently not supported on Windows XP.

bidle

The bidle message is sent when the cursor is over your object's rectangle in a locked Patcher window. This method is optional, since not all objects need to adjust or track the cursor.

BINDING

```
addmess (myobject_bidle, "bidle", A_CANT, 0);
```

DECLARATION

```
void myobject_bidle (t_myobject *x, Point pt, short
modifiers, short active);
```

`pt`	Current location of the mouse in local coordinates.
`modifiers`	The modifers field of the Macintosh EventRecord returned by GetNextEvent for this key event, indicating whether the shift, option, command, caps lock, or control keys were pressed.
`active`	Zero if your window is not the active (frontmost) window, non-zero if it is.

The bidle message is sent when the cursor is over your object's rectangle in a locked Patcher window. You'll typically track the cursor—for example on Macintosh to find its location, call the Mac OS function GetMouse. If you want to set the cursor to a different shape, call wind_setcursor with an argument of –1 before returning.

bidleout

The bidleout message is sent when the cursor moves off of your object's rectangle in a locked Patcher window. This lets you stop doing whatever you were doing in response to the bidle message.

BINDING

```
addmess (myobject_bidleout, "bidleout", A_CANT, 0);
```

DECLARATION

```
void myobject_bidleout (t_myobject *x, Point pt);
```

`pt`	Location of the mouse where it left the object's rectangle, in local coordinates.

The bidleout message is sent when the cursor moves off of your object's rectangle in a locked Patcher window.

Routines for User Interface Objects

Here are some helpful functions for writing your user interface object. As was mentioned above, most user interface objects will need to use a Qelem to change their displayed state in response to a message such as int or set. In your queue function it is important to call `patcher_setport` and `box_nodraw` before doing any drawing, since your object may be in a closed patcher window or have been hidden by the user. However, you should not call these routines in your update method. If you want to call your update method from within a queue function, do the necessary preparation outside of the update method and then call it. There is an example under the description of the **color** message.

box_ownerlocked

Use `box_ownerlocked` to determine if a patcher that contains a box is locked.

```
short box_ownerlocked (t_box *b);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

This function returns non-zero if the box's owning patcher window is locked, zero if it's unlocked. (Generally, an unlocked patcher shows the UI object toolbar at the top of a patcher window.)

User interface objects may want to draw themselves differently if the Patcher window is locked. You may have noticed that a box's outlets are extended downward one pixel when a Patcher is unlocked. This is done automatically for you.

box_size

Use `box_size` to resize a box.

```
void box_size (t_box *b, short hsize, short vsize);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.
`hsize`	The new horizontal size of the box.
`vsize`	The new vertical size of the box.

box_nodraw

Use `box_nodraw` to determine if a user interface object should be drawn.

```
short box_nodraw (t_box *b);
```

`b`	The Box structure that is the header of your user interface object. You can just pass a pointer to your object here.

`box_nodraw` should be used when drawing outside of a user interface object's update method. It returns non-zero if the Box's Patcher is locked and the Box is hidden (because the user has chosen Hide on Lock from the Max menu). It can also set up clipping regions so your object draws properly with other objects. For instance, if your object is partially hidden by the edge of a bpatcher, box_nodraw will handle clipping for you. However, it will only provide these services if you pass the F_SAVVY flag to `box_new`, which indicates you have used box_nodraw and box_enddraw in any non-update drawing.

Your object should not draw in response to any typed messages it receives (i.e., messages other than those with an A_CANT argument type specifier). For example, a slider draws its changed value in a queue function set in response to an int messages. Inside the queue function, it checks `box_nodraw` before drawing. You should only call `box_nodraw` after you've first determined that your object's patcher window is visible (with `patcher_setport)`. `box_nodraw` will crash if you call it in your user interface object's update method. All clipping is set up for you in the update method whether or not you passed the F_SAVVY flag to `box_new`.

box_enddraw

Use `box_enddraw` to tell a Patcher you are finished drawing after having called `box_nodraw`.

```
void box_enddraw (t_box *b);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

`box_enddraw` is used when drawing outside of your update method. It is called after you are finished drawing, and *only* if a previous call to `box_nodraw` returns false. An example draw routine might be:

```
if (gp = patcher_setport(x->m_box.b_patcher)) {
    if (!box_nodraw((t_box *)x)) {
        // draw here
        box_enddraw((t_box *)x);
    }
    patcher_restoreport(gp);
}
```

To get the maximum benefit from box_nodraw and box_enddraw, add the F_SAVVY flag to your call to box_new in the new instance routine.

```
box_new((t_box *)x, patcher, F_DRAWFIRSTIN | F_NODRAWBOX |
           F_GROWBOTH | F_SAVVY, l, t, r, b);
```

box_redraw

Use `box_redraw` to cause a box's frame and contents to be redrawn.

```
void box_redraw (t_box *b);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

This function erases and redraws the entire Box as well as any other Boxes or Patchlines in the affected area. Your object will receive an update message as a result.

box_visible

Use `box_visible` to determine if a box is visible (whether in a visible patcher, or visible within a bpatcher).

```
short box_visible (t_box *b);
```

`b`	The Box structure that is the header of your user interface object. You can just pass a pointer to your object here.

This function returns non-zero if the Box is visible, or zero if the Box is not visible on-screen.

patcher_deselectbox

Use `patcher_deselectbox` to visually deselect a box in a patcher.

```
void patcher_deselectbox (t_patcher *p, t_box *b);
```

p	The box's owning patcher (`b->b_patcher`).
b	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

patcher_dirty

Use `patcher_dirty` to mark a patcher as having unsaved data.

```
void patcher_dirty (t_patcher *p);
```

p	Your object's owning patcher.

This function sets the dirty bit in a Patcher's window, so that the user will asked to save changes if the window is closed. The only time you have to do this is if your object stores its data inside its owning patcher (for example, a **table** object can store its elements inside a Patcher) and the data is changed.

Note: If your object is *not* a user interface object, you can find your Box's "owning patcher" in your object creation function (this is the *only* time this will work) by getting the object bound to the symbol #P.

```
void *mypatcher;
mypatcher = ((t_symbol *)gensym("#P"))->s_thing;
```

patcher_selectbox

Use `patcher_selectbox` to visually select a box.

```
void patcher_selectbox (t_patcher *p, t_box *b);
```

p	The box's owning patcher (`b->b_patcher`).
b	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

patcher_setport

Use `patcher_setport` to set the current GrafPort to a patcher window.

```
GrafPtr patcher_setport (t_patcher *p);
```

`p`	Your object's owning patcher.

This function sets the current `GrafPort` to the Patcher's window. `patcher_setport` returns the previous port, or `0` if the Patcher window is not currently visible, in which case you should not draw anything. After you're through drawing in the patcher window, call SetPort with the result of a successful `patcher_setport`. Note that this function is *not* equivalent to `wind_setport`, though the way you'd use it is similar to the example shown under `wind_setport` in Chapter 10. Do not use `wind_setport` instead of `patcher_setport` in a user interface object. If you think you're clever and pass the patcher's `p_wind` field to `wind_setport`, you will get burned eventually, since patchers contained within **bpatcher** objects contain invalid information in this field.

Note: On Windows XP it is also important to call this function in order for the clipping regions to work properly. However, the returned `GrafPtr` is an opaque type that should only be used when you restore the port with a call to `patcher_restoreport`.

Further note: On Windows XP when using QTML you need to use XQT_patcher_setport and XQT_patcher_restoreport. For additional details see the section below on the XQT API.

patcher_restoreport

Use `patcher_restoreport` to set the GrafPort returned by `patcher_setport`. On the Macintosh this currently calls MacSetPort() through a C macro.

```
void patcher_restoreport (GrafPtr gp);
```

`gp`	Opaque structure returned by `patcher_setport`.

This function sets the current `GrafPort` to the port returned by `patcher_setport`.

patcher_okclose

Use `patcher_okclose` to set the receiver of a patcher window's okclose message.

```
void patcher_okclose (t_patcher *p, t_object
*target);
```

`p`	Your object's owning patcher.
`target`	Your object.

After calling `patcher_okclose`, the patcher window will send any okclose message it receives on to the object `target`. In the case where your object opens a Patcher window itself, you might want to handle how the window is saved or closed in a non-standard way. You should then implement an okclose method and pass yourself to `patcher_okclose` after you've opened the patcher window. See Chapter 10 for a description of writing an okclose method.

ispatcher

Use is`patcher` to determine if an object is a patcher.

```
short ispatcher (t_object *obj);
```

`obj`	The mystery object that could be a patcher.

Returns non-zero if an object is a Patcher, zero if not. You might use this to locate all the patcher windows from the window list. On the Macintosh first use GetWRefCon to get a pointer to the Max t_wind object associated with a Macintosh window. On Windows XP you can call wind_fromhwnd to get the Max t_wind object associated with a Windows window. Then call `ispatcher` on the `w_assoc` field of the t_wind if a valid t_wind is returned.

Or, if you're traversing through the list of a patcher's boxes, you might be interested to know if an object associated with the box (in the `b_firstin` field) is a subpatcher. Instead of passing the box itself, check to see if the `b_firstin` field of a box is non-zero, and if so, pass it to `ispatcher`. You need to pass a genuine Max object to `ispatcher`.

isnewex

Use `isnewex` to determine if an object is an object box.

```
short isnewex (t_object *obj)
```

`obj`	The mystery object that could be an object box.

Returns non-zero if an object is an object box that holds the text of normal objects. You can then find the pointer to the object itself in the `t_box`'s `b_firstin` field. Make sure you pass a genuine Max object to `isnewex`.

Note: You can always find the type of an object using the macros `ob_name` or `ob_class` defined in *ext_mess.h*, then comparing the string (in the case of `ob_name`) or symbol (in the case of `ob_class`) to the one you're looking for. `ispatcher` and `isnewex` are slightly faster.

newex_knows

Use `newex_knows` to determine if the object in an object box can respond to a message.

```
short newex_knows (t_box *b, t_symbol *msg);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.
`msg`	The message selector that the object may or may not understand.

This function returns non-zero if the object is a box that holds the text of normal objects *and* its associated object has a method for `msg`.

patcher_eachdo

Use `patcher_eachdo` to call a function on every patcher in memory.

```
void patcher_eachdo (method fun, void *arg);
```

`fun`	The function you want called. See below for how to declare it.

`arg`	An argument you want passed to the function.

The function `fun` will be called for every patcher in memory, including hidden subpatchers. It should be declared as follows:

```
short myEachFun (t_patcher *patcher, void *arg);
```

`patcher`	A patcher window.
`arg`	The arg argument passed to patcher_eachdo.

Your function needs to return 0 if you wish to continue being called for more patchers, non-zero if you want to stop. You might use `patcher_eachdo` to implement a search routine for a particular object. It is used by the built-in objects **send** and **receive** in response to a dblclick message. These objects search all Patchers for any other **send** or **receive** objects bound to the same t_symbol, and bring the corresponding object's window to the front.

assist_drawstring

Use `assist_drawstring` to draw a string in the assistance area of a patcher window.

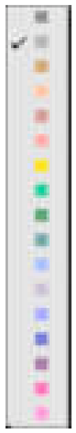
```
void assist_drawstring (void *patcher, char
*string);
```

`patcher`	The patcher where you want to draw the string.
`string`	The C string to draw.

The assistance area of a patcher window can be used for any helpful messages while the user is clicking on your object. An example is the display of the current horizontal and vertical offsets when scrolling what is visible inside a bpatcher object.

Color And User Interface Objects

If your user-interface object responds to the color message, users can take advantage of a standard submenu for assigning it one of sixteen colors. The color menu previews each color and allows color selection.



The color menu sets the b_color field of your object's t_box to a number between 0 and 15. Then you are sent an update message to redraw your object. This means that if you implement the color feature, your update method needs to check for changes in the b_color field of the box. You use the function `box_color` to get the current RGB color associated with the value of the b_color field.

color

The color message has a number of uses. First, you must respond to it if the color menu will be enabled when your object is selected. Second, the color message is sent to your object when the user sets the color to 0 (usually black or gray). It can also be sent directly by the user to change the object's color. You should queue a redraw of your object when receiving this message, as well as setting the b_color field of your box to the value you receive as an argument. You'll need to constrain this value between 0 and 15 (an index into the set of standard colors), since a Max user could send any argument to the color message.

BINDING

```
addmess (myobject_color, "color", A_LONG, 0);
```

DECLARATION

```
void myobject_color (myObject *x, long color);
```

`color`	A number between 0 and 15 representing the color chosen.

In response to this message, you should set the b_color field of your object to the color value, then redraw your object. Here's an example of how to redraw an object using the `defer` function.

```
void myuserobject_color (t_myuserobject *x, long color)
{
    x->m_box.b_color = color;   // set the box's color field
    defer(x,(method)myuserobject_docolor,0,0,0); // cause
redraw
}
void myuserobject_docolor (t_myuserobject *x)
{
    GrafPtr gp;
    if (gp = patcher_setport(x->m_box.b_patcher)) {
        if (!box_nodraw((t_box *)x)) {
            myuserobject_update(x);
            box_enddraw((t_box *)x);
        }
        SetPort(gp);
    }
}
```

In an update message, you need to translate a color value in the b_color field of a box into an index that references the standard set of colors. This can be performed with the `box_color` function.

frgb, brgb, rgb1 etc.

These messages set colors for various aspects of your object. They are not sent by Max but exist only as conventions, so users can expect that if your object supports RGB colors it will work with these messages. A standard component of an inspector contains a swatch object that is designed to work with these messages. The frgb message should correspond to the "content" in your object (such as the color of the text), while the brgb message should set the background color behind the content. Additional colors, if your object uses them, should be set with rgb1, rgb2, etc.

BINDING

```
addmess (myobject_frgb, "frgb", A_LONG, A_LONG,
A_LONG, 0);
```

DECLARATION

```
void myobject_frgb (t_myuserobject *x, long r, long
g, long b);
```

`r, g, b`	Components of an RGB color as values between 0-255.

This message should change the color of some component of your object, and then cause the object to be redrawn at a lower priority. Here's an example. Assume an object has a field m_fg that is an RGBColor that determines the foreground color used by an object. The frgb method shown below converts between the Max representation (8 bits per component) and the Mac OS RGBColor representation (16 bits per component), then defers a redraw of the object.

```
void myuserobject_frgb (t_myuserobject *x, long r, long g,
long b)
{
    x->m_fg.red = CLIP(r*257, 0, 65535);
    x->m_fg.green = CLIP(g*257, 0, 65535);
    x->m_fg.blue = CLIP(b*257, 0, 65535);
    defer(x,(method)myuserobject_doredraw,0,0,0);
}
```

Here is the myuserobject_doredraw function, which should look fairly familiar from other examples you've seen.

```
void myuserobject_doredraw(t_myuserobject *x)
{
    GrafPtr gp;
    if (gp = patcher_setport(x->m_box.b_patcher)) {
        if (!box_nodraw((t_box *)x)) {
            myuserobject_update(x);
            box_enddraw((t_box *)x);
        }
        SetPort(gp);
    }
}
```

box_color

Use `box_color` to set the current foreground color to the color of your box.

```
void box_color (t_box *b);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

`box_color` assumes you've stored the color index (0-15) in the `b_color` field of your box. It checks to see if the patcher window is currently in color and, if so, it looks up the RGB color that corresponds to the `b_color` field and sets the foreground color to that value. After calling `box_color` you can draw the colored parts of your object. Then you'll want to restore the foreground color to black by calling RGBForeColor.

Note on Windows XP you should use `box_getcolor` in conjunction with appropriate native WIN32 API calls or appropriate QTML QuickTime calls. Calling `box_color` on Windows XP will have unexpected results since there is no call to `RGBForeColor` to reset the foreground color to black.

box_usecolor

Use `box_usecolor` to determine if your box should be drawn in color.

```
long box_usecolor (t_box *b);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.

`box_usecolor` returns a non-zero value if your object box is to be drawn in color, and zero otherwise. It takes into account multiple monitors of different depths in making this determination. It would be a good idea to call this each time your object receives an update message.

box_getcolor

Use `box_getcolor` to get the RGB values of the color associated with a box color index.

```
void box_usecolor (t_box *b, short index, RGBColor
*rgb);
```

`b`	The t_box structure that is the header of your user interface object. You can just pass a pointer to your object here.
`index`	The color index (0-15) for which you want the RGB values.
`rgb`	Where the RGB values for the index will be placed.

box_getcolor returns in rgb the RGB values of the color with the color index (0-15) supplied in its index argument

Transparent Objects

You can create objects that do not completely fill in their drawing rectangle, exposing other objects beneath them. A common example is the **comment** object that draws its text over objects beneath it.

First, let's look at a simple case: an object that wants to draw text over a background without erasing or filling a rectangle. This example object draws the fixed string "Max" within its rectangle. It also responds to the "frgb" message to set the color of the text, and therefore needs to redraw itself in a qelem. The data structure for this object would be :

```
typedef struct _drawmax
{
   t_box d_box;
   void *d_qelem;
   RGBColor d_color;
} t_drawmax;
```

Next, the new instance routine. The object stores its rectangle coordinates and the red, blue, and green color values.

```
void *drawmax_new(t_symbol *s, short argc, t_atom *argv)
{
   t_drawmax *x = (t_drawmax *)newobject(drawmax_class);
   t_patcher *p = argv[0]->a_w.w_obj;
   short top, left, bottom, right;
   top = argv[1]->a_w.w_long;
   left = argv[2]->a_w.w_long;
   bottom = argv[3]->a_w.w_long;
   right = argv[4]->a_w.w_long;
   box_new(x,p,DRAWFIRSTIN | F_NODRAWBOX | F_GROWBOTH |
   F_SAVVY |
      F_TRANSPARENT,left,top,right,bottom);
   x->d_color.red = argv[5]->a_w.w_long;
   x->d_color.green = argv[6]->a_w.w_long;
   x->d_color.blue = argv[7]->a_w.w_long;
   x->d_qelem = qelem_new(x,(method)drawmax_redraw);     //
define qelem
   box_ready((t_box *)x);
   return x;
}
```

Next is the object's update method. It clips to the box's rectangle so it doesn't draw the word "Max" outside of its box. Notice that it doesn't erase the box first—the text will appear on top of the existing background.

```
void drawmax_update(t_drawmax *x)
{
   RgnHandle old,boxclip;
   RGBColor saveColor;
   GetClip(old = NewRgn());    // get existing clip region
   RectRgn(boxclip = NewRgn(), &x->d_box.b_rect);   // box's
rect as region
   SectRgn(old,boxclip,boxclip);  // intersect them
   SetClip(boxclip);    // make current clip region
   // get ready to draw text
   MoveTo(x->d_box.left + 4, x->d_box.bottom – 4);
   TextFont(0);
   TextSize(12);
   GetForeColor(&saveColor);   // get existing color
   RGBForeColor(&x->d_color);  // set to your color
   DrawString("\pMax");    // draw the text
   RGBForeColor(&saveColor);   // restore everything…
   SetClip(old);
   DisposeRgn(old);
   DisposeRgn(boxclip);
}
```

This is how the object handles changes to color.

```
void drawmax_frgb(t_drawmax *x, long r, long g, long b)
{
    x->d_color.red = CLIP(r*257, 0, 65535);         // get the
new color
    x->d_color.green = CLIP(g*257, 0, 65535);
    x->d_color.blue = CLIP(b*257, 0, 65535);;
    qelem_set(x->d_qelem);      // make it redraw
}
```

Next is the routine `drawmax_redraw`, the routine called from within a qelem. It's a wrapper around `drawmax_update` with all of the goodies—`box_nodraw`, `box_enddraw` etc.

```
void drawmax_redraw(t_drawmax *x)
{
    GrafPtr gp;
    if (gp = patcher_setport(x->d_box.b_patcher)) {
        if (!box_nodraw((t_box *)x)) {
            drawmax_update(x);
            box_enddraw((t_box *)x);
        }
        SetPort(gp);
    }
}
```

One thing transparent objects can do to help Max optimize their redrawing is to have a "clipregion" method. When objects below yours are redrawing, Max clips out your object. If your object is transparent, you may not want your entire box rectangle clipped out. With the clipregion method you can specify what you want clipped.

```
addmess((method)myobject_clipregion, "clipregion", A_CANT, 0);
```

This method tells Max the region in which your object has content. For the drawing of text, the region masking the text is too complicated to define, so we return a special constant `CLIPRGN_COMPLEX`. Basically, this means it's impossible to avoid redrawing the text on top of objects below it when they need to be redrawn. Objects that are not transparent shouldn't use this constant—indeed, there's no reason for them to supply the default `CLIPRGN_RECT` answer to the clipregion method at all.

```
void myobject_clipregion(t_drawmax *x, RgnHandle *rgn, short
*result)
{
   *result = CLIPRGN_COMPLEX;
}
```

If your object doesn't define a clipregion method, it's assumed you clip to the object's rectangle.

Another possibility is an object that draws a shape that does not entirely fill the object's rectangle. The umenu object is one example. Here is an object that draws a rectangle that is three pixels smaller than its bounding rectangle.

```
void smallrect_update(t_smallrect *x)
{
   Rect r = x->s_box.b_rect;
   InsetRect(&r,3,3);
   if (!EmptyRect(&r))
      PaintRect(&r);
}
```

This object's clipregion method would use the following code. It sets the region handle to the region that defines a small rectangle, and sets the result to `CLIPRGN_REGION`, indicating there's a region to consider. Your method needs to define the region handle, but it should not use it in any other way, because Max will dispose of it shortly after having called your clipregion method.

```
void smallrect_clipregion(t_smallrect *r, RgnHandle *r, short
*result)
{
   Rect rect;
   rect = x->s_box.b_rect;
   InsetRect(&rect,3,3);   // make a rectangle the same
                        // size as what you draw
   *r = NewRgn();
   RectRgn(*r,&rect);   // turn it into a region
   *result = CLIPRGN_REGION;
}
```

The example circle object in the SDK, which is heavily commented, shows another, more elaborate use of clipping regions. It also uses the `box_invalnow` method in its qelem routine to draw what it may have revealed by moving, and it demonstrates the proper way to handle dragging for transparent objects.

Inspectors

User interface objects (and, possibly, certain non-UI objects that have their own windows) may wish to implement a way for users to change settings using a patcher rather than a dialog box. This is possible using the new inspector mechanism in Max 4. An inspector is a specially named patch, opened when the user selects your object in an unlocked patcher and chooses "Get Info…", that can edit the settings of an instance of your object.

Inspectors are intended to make Max more customizable (users can change the inspector patches as desired, and/or use features from inspectors in their own patches), as well as to make it easier to port the software to other platforms. In order to implement an inspector, your object has to do the following:

- bind the function inspector_open to the word info
- add a call to notify_free in your object's free method
- if it is not possible to parse the object's output in a psave method, implement a pstate method with a simpler format.
- if necessary, implement methods to change the internal settings of your object that are saved in its psave output. For instance, if your object has a "maximum" value saved in a patcher that was changeable in a traditional Get Info dialog, you will need to add a message to change it.

After the object has been properly prepared, you can write an inspector patch named *myobject-insp.pat* (substitute the name of your object for *myobject*). You then place the inspector patch anywhere in the search path (preferably in the inspectors subfolder of the patches folder). The inspector patch uses the **thisobject** object to communicate with an instance of your object. **thisobject** outputs the messages generated by your object's psave or pstate method. You use this information to show the object's current state. Your patch can then send thisobject messages to change the settings of the instance; the messages are simply passed on to your object. A commented example of an inspector patch is provided with the **hslider** object source code in the SDK.

inspector_open

To implement an inspector, your object binds the info message to the function inspector_open, as shown below:

```
addmess(inspector_open, "info", A_CANT, 0);
```

The `inspector_open` function is built into Max; it will open a patcher file called **myobject-insp.pat** found in the search path. Opening a patch using inspector_open performs special binding so that the **thisobject** object within the patch is linked to the instance being inspected. Inspector patches need to be written using special techniques illustrated in the **hslider** example.

notify_free

Use `notify_free` in the free method of an object that uses `inspector_open`.

```
void notify_free(t_object *owner);
```

`owner`	The object associated with the inspector.

Call this in the free method for any object that has an inspector; it disconnects an object from its inspector if open. `notify_free` should be called before `box_free`.

pstate

You can use the pstate method to provide a alternative to your existing psave method that makes it easier to write an inspector patch. For instance, some psave methods need to include large amounts of data that will not apply to an inspector, or they save the data in a bitfield format that is difficult to parse. If your object has a pstate method, it will be called in preference to the psave or save methods.

BINDING

```
addmess (myobject_pstate, "pstate", A_CANT, 0);
```

DECLARATION

```
void myobject_pstate (myobject *x, void *w);
```

`w`	A pointer to a binbuf where your object will construct its saved output message.

The pstate output should consist of a symbol that names your object's class, followed by values that reflect its settings that can be saved and restored. For instance, here's a pstate method that saves three settings inside an object m_multipler, m_weight, and m_length.

```
void myobject_pstate(t_myobject *x, void *w)
{
    binbuf_vinsert(w, "slll", gensym("myobject"), x-
>m_multiplier,
        x->m_weight, x->m_length);
}
```

QuickTime Image Routines

The Max QuickTime image routines can be used by externals that need to use features of QuickTime to open files. QuickTime has function s that can open many file types, including image files (jpeg, gif, pict and photoshop files), time-based media files (movies, Flash and 3-D files) and non-image files (text and audio) with scaling and timing factor manipulation. For a complete list of files openable by QuickTime, refer to the QuickTime Developer documentation.

qtimage_open

Use `qtimage_open` to open a file and render it into a GWorld.

```
long qtimage_open(char *name, short path, CGrafPtr
*gp, void *extra);
```

`name`	File name to open.
`path`	File path for the selected file.
`gp`	GWorld that will receive the rendered file.
`extra`	Information that may be used to render the file (see `qti_extra`).

Given a filename, path and GWorld(gp), `qtimage_open` will open a file and render it in the given GWorld. Extra information may be used in rendering the image, such as scaling and timeoffset (for time-based media).

If gp is NULL, `qtimage_open` will allocate a new 32-bit offscreen gworld to contain the image (scaled, if necessary). It is the calling object's responsibility to free this memory. If the calling object needs to use an on-screen GWorld, a non 32-bit GWorld or the autofit scaling mode, gp must be a valid pointer.

If quicktime is not installed, this routine will only open PICT files.

On Windows XP the CGrafPtr works with QTML Graphics Worlds. For this to work the QTI_FLAG_QTML_GWORLD field needs to be set in the extra->flags. You can set this using the qtimage_extra_flags_set method documented below.

qtimage_getrect

Use `qtimage_getrect` to return the size needed to render an image file (scaled, if necessary).

```
long qtimage_getrect(char *name, short path, Rect
*r, void *extra);
```

`name`	File name to query.
`path`	File path for the selected file.
`r`	A Rect structure that will be filled with the required render size.
`extra`	Information that may be used to render the file (see qti_extra).

qti_extra_new

Use `qti_extra_new` to create a new `qti_extra` object.

```
void *qti_extra_new(void);
```

qti_extra_free

Use `qti_extra_free` to free the memory allocated by `qti_extra_new`.

```
void qti_extra_free(void *extra)
```

`extra`	`qti_extra` object to free.

qti_extra_matrix_get

Use `qti_extra_matrix_get` to copy the contents of a `qti_extra` object's matrix member into a MatrixRecord.

```
long qti_extra_matrix_get(void *extra, MatrixRecord
*matrix);
```

`extra`	The `qti_extra` object that is the copy source.
`matrix`	A MatrixRecord that is the copy destination.

`qti_extra_matrix_get` will copy the contents of a `qti_extra` object's matrix member into a MatrixRecord. See the Apple QuickTime developer documentation for more information on MatrixRecords and how they are used for transforming images. Both `extra` and `matrix` must be valid pointers.

qti_extra_matrix_set

Use qti_extra_matrix_set to copy the contents of a MatrixRecord into a `qti_extra` object's matrix member.

```
long qti_extra_matrix_set(void *extra, MatrixRecord
matrix);
```

`extra`	The `qti_extra` object that is the copy destination.
`matrix`	A MatrixRecord that is the copy source.

`qti_extra_matrix_set` will copy the contents of a MatrixRecord into a `qti_extra` object's matrix member. See the Apple QuickTime developer documentation for more information on MatrixRecords and how they can be used for transforming images. For the `qti_extra` matrix memver to be used for scaling in `qtimage_open` and `qtimage_rect`, the `qti_extra` object's scalemode must be set to QTI_SCALEMODE_MARTIX (see `qti_extra_scalemode_set`). `extra` and `matrix` must be valid pointers.

qti_extra_rect_get

Use `qti_extra_rect_get` to copy the contents of a `qti_extra` object's rect member into a Rect struct.

```
long qti_extra_rect_get(void *extra, Rect *rct);
```

`extra`	The qti_extra object that is the copy source.
`rct`	The Rect struct that is the copy destination.

`qti_extra_rect_get` will copy the contents of a `qti_extra` object's rect member into a standard Rect structure. See the Apple QuickTime developer documentation for more information on using Rect structures to transform images. Both `extra` and `rct` must be valid pointers.

qti_extra_rect_set

Use `qti_extra_rect_set` to copy the contents of a Rect struct into a `qti_extra` object's rect member.

```
long qti_extra_rect_set(void *extra, Rect *rct);
```

`extra`	The `qti_extra` object that is the copy destination.
`rct`	The Rect struct that is the copy source.

`qti_extra_rect_set` will copy the contents of a Rect struct into a `qti_extra` object's rect member. See the Apple QuickTime developer documentation for more information on using Rect structures to transform images. For the `qti_extra` object's rect member to be used for scaling in `qtimage_open` and `qtimage_rect`, the `qti_extra` objects scalemode member variable must be set to QTI_SCALEMODE_RECT (see `qti_extra_scalemode_set`). Both `extra` and `rct` must be valid pointers.

qti_extra_scalemode_get

Use `qti_extra_scalemode_get` to copy the contents of a `qti_extra` object's rect member into a scalemode (long).

```
long qti_extra_scalemode_get(void *extra, long
*scalemode);
```

`extra`	The qti_extra object that is the copy source.
`scalemode`	The scalemode (implemented as a long) that is the copy destination.

Both `extra` and `scalemode` must be valid pointers.

qti_extra_scalemode_set

Use `qti_extra_scalemode_set` to copy a scalemode into a `qti_extra` object's rect member variable.

```
long qti_extra_scalemode_set(void *extra, long
scalemode);
```

`extra`	The qti_extra object that is the copy destination.
`scalemode`	The scalemode (implemented as a long) that is the copy source.

`qti_extra_scalemode_set` will copy a long value into a `qti_extra` object's rect member variable. `extra` must be a valid pointer. Possible values for scalemode are:

QTI_SCALEMODE_NONE

QTI_SCALEMODE_MATRIX

QTI_SCALEMODE_RECT

QTI_SCALEMODE_AUTOFIT

If using QTI_SCALEMODE_AUTOFIT, qtimage_open reqires a valid GWorld pointer(gp). Also, when using QTI_SCALEMODE_AUTOFIT, the `qtimage_rect` function is not used, since the rect required to render the image will be the same as the dimensions of the GWorld passto to `qtimage_open`.

qti_extra_time_get

Use `qti_extra_time_get` to copy the contents of a `qti_extra`'s time member variable into a double.

```
long qti_extra_time_get(void *extra, double *time);
```

`extra`	The `qti_extra` object that is the copy source.
`time`	The double that is the copy destination.

See the Apple QuickTime developer documentation for more information on time units. Both `extra` and `time` must be valid pointers.

qti_extra_time_set

Use `qti_extra_time_set` to copy a double value into a `qti-extra` time member variable.

```
long qti_extra_time_set(void *extra, double time);
```

`extra`	The qti_extra object that is the copy destination.
`time`	The double value that is the copy source.

`extra` must be a valid pointer. See the Apple QuickTime developer documentation for more information on time units.

qti_extra_flags_get

Use `qti_extra_flags_get` to copy the contents of a `qti_extra`'s flags member variable into a long.

```
long qti_extra_flags_get(void *extra, long *flags);
```

`extra`	The `qti_extra` object that is the copy source.
`flags`	The long that is the copy destination.

Both `extra` and `flags` must be valid pointers.

qti_extra_flags_set

Use `qti_extra_flags_set` to copy a long value into a `qti-extra` flags member variable.

```
long qti_extra_flags_set(void *extra, long flags);
```

`extra`	The qti_extra object that is the copy destination.
`flags`	The long value that is the copy source.

`extra` must be a valid pointer.

XQT and QTML for Windows UI Externals

For Macintosh developers with a heavy reliance upon QuickDraw, QuickTime or some other part of the Macintosh Toolbox supported by Apple's QuickTime Media Layer (QTML) for windows, it might make sense to use the QTML libraries for porting your object to Windows. The "qtxgui" example project in the SDK shows how to make use of QTML's QuickDraw implementation inside a Max UI object. In order for this to work properly, there are a few helper functions and some caveats about the way the functions are called.

These helper functions are defined in ext_qtstubs.h and have the prefix "XQT", and so we'll refer to these calls as the XQT calls or the XQT API.

- In order to link to QTML you'll need to download the QuickTime SDK from Apple. The "qtxgui" project example requires that these headers are located in "..\..\win-sdks\QuickTime\inc" and libraries located in "..\..\win-sdks\QuickTime\lib", so if you are basing your project from that template, you should move the appropriate files from the QT SDK to this location. You'll also need to have the USE_QTML preprocessor definition defined for your project.

- Do not call InitializeQTML directly, but rather XQT_InitializeQTML. If you are making any XQT calls, this will be done for you if it hasn't already been initialized, but if you are making ordinary calls into the QTML libraries prior to making any other XQT calls, you should call XQT_InitializeQTML somewhere appropriate — e.g. main.

- Do not call TerminateQTML.

- If you are using QTML's QuickDraw implementation in a Max extern you will need to use the XQT_patcher_setport, XQT_patcher_restoreport, XQT_box_nodraw and XQT_box_enddraw as is shown in the "qtxgui" example project. These make the appropriate conversions to preserve clipping regions and other QD state, as well as allocate the necessary QT PortAssociation if one has not already been created for your patcher window.

- Certain QTML functions now have a "Mac" prefix in order to avoid conflict with Windows functions of the same name. Please consult the QuickTime for Windows developer documentation to see a complete list, but things to look out for are SetPort (now MacSetPort) and functions dealing with rectangles or regions.

- Do not make any QTML calls inside dll_main if you are using your own dll_main code (note that this is different from your main function in which QTML calls are allowed).

XQT_GetQuickTimeVersion

Use `XQT_GetQuickTimeVersion` to get the current version of Quicktime.

```
long XQT_GetQuickTimeVersion (void);
```

Returns Quicktime version if installed. Return value is zero if Quicktime is not installed.

XQT_CheckFunPtr

Use `XQT_CheckFunPtr` to check a Quicktime function pointer.

```
long XQT_CheckFunPtr(FARPROC *ppfn, const char
*name
```

`ppfn`	Function pointer.
`name`	Name of function.

If function pointer pointed to by ppfn is NULL, attempt to load function specified by

"name" from QuickTime library. Return value is an error code.

XQT_OpenMovieFilePathSpec

Use `XQT_OpenMovieFilePathSpec` to open a movie file.

```
OSErr XQT_OpenMovieFilePathSpec(const PATH_SPEC
*pathSpec, short *resRefNum, SInt8 permission);
```

`pathSpec`	A PATH_SPEC.
`resRefNum`	Reference number.
`permission`	Flag for read/write permission.

Similar to OpenMovieFile, but takes a Max PATH_SPEC rather than a QT FSSpec. On Macintosh these are identical, but not on Windows. Return value is error code.

XQT_QT2MaxSpecCopy

Use `XQT_QT2MaxSpecCopy` to copy a Quickeime FSSpec to a Max PATH_SPEC.

```
long XQT_QT2MaxSpecCopy(FSSpec *qt_spec, PATH_SPEC
*max_spec);
```

`qt_spec`	A pointer to a source Quicktime FSSpec.
`max_spec`	A pointer to a destination Max-specific PATH_SPEC.

Convert QT FSSpec to Max PATH_SPEC. Return value is error code..

XQT_ Max2QTSpecCopy

Use `XQT_Max2QTSpecCopy` to open a movie file.

```
long XQT_Max2QTSpecCopy(PATH_SPEC *max_spec, FSSpec
*qt_spec);
```

`max_spec`	A pointer to a source max PATH_SPEC.
`qt_spec`	A pointer to a destination Quicktime FSSpec.

Convert Max PATH_SPEC to QT FSSpec. Return value is error code..

XQT_patcher_setport

Use `XQT_patcher_setport` in place of `patcher_setport` when making QTML-based UI externs.

```
GrafPtr XQT_patcher_setport(struct patcher *p)
```

`p`	A pointer to your UI object’s owning patcher.

Replacement for `patcher_setport` required for QTML based UI externs. Unlike Macintosh version this must also be called inside the update method.

XQT_patcher_getport

Use `XQT_patcher_getport` to get a Max patcher's QT port..

```
GrafPtr XQT_patcher_getport(struct patcher *p)
```

`p`	A pointer to your UI object's owning patcher.

Return a Max patcher's QT port.

XQT_patcher_restoreport

Use `XQT_patcher_restoreport` in place of `patcher_restoreport` when making QTML-based UI externs.

```
void XQT_patcher_restoreport(GrafPtr gp)
```

`gp`	A pointer to an opaque GrafPort structure returned by `XQT_patcher_setport.`

Replacement for SetPort as used to restore the previous port. This is required for QTML based UI externs, and should be paired with any use of XQT_patcher_setport.

XQT_wind_setport

Use `XQT_wind_setport` in place of `wind_setport` when making QTML-based UI externs.

```
GrafPtr XQT_wind_setport(struct wind *w)
```

`w`	A pointer to your UI object's owning patcher's window.

Replacement for `wind_setport` required for QTML based UI externs. Unlike Macintosh version this must also be called inside the update method.

XQT_wind_getport

Use `XQT_wind_getport` to get a Max patcher's QT port.

```
GrafPtr XQT_wind_getport(struct wind *w)
```

`w`	A pointer to your UI object's owning patcher's window.

Return a Max window's QT port. Note: make sure you restore the port with XQT_SetPort. Restoring using wind_restoreport can lead to unexpected results.

XQT_box_nodraw

Use `XQT_box_nodraw` in place of `wind_setport` when making QTML-based UI externs.

```
GrafPtr XQT_box_nodraw(struct box *b)
```

`b`	A pointer to your UI object's Box.

Replacement for `box_nodraw` required for QTML based UI externs.

XQT_box_enddraw

Use `XQT_box_enddraw` to get a Max patcher's QT port..

```
GrafPtr XQT_box_enddraw(struct box *b)
```

`b`	A pointer to your UI object's Box.

Replacement for `box_enddraw` required for QTML based UI externs.

Chapter 12: Graphics Windows

Note: these routines are not supported on Windows XP and may also be obsolete on Mac OSX at some point in the near future. We recommend using the **lcd** object for simple graphics rather than implementing objects that use these routines. For higher level graphics manipularion, we suggest developing external objects for use with Jitter .

Max contains a set of routines for managing bit map graphics and offscreen "sprites" needed to do animation and paint-program interaction. Graphics windows, known as GWinds, are created by the Max user with the **graphic** object, but they can also be created directly from your external object by calling `gwind_new`. Graphics windows are primarily display vehicles—if you want to create a specialized user interface, you're better off creating your own window (see Chapter 10). Max programmers can access the mouse in a graphics window using the **MouseState** object, but a typical object that uses a graphics window will only draw something—animation, shapes, text, etc.

The t_gwind structure and pertinent constants are declared in *ext_anim.h*.

Graphics Window Routines

Here are routines for creating graphics windows, associating them with symbols, and drawing in them.

colorinfo

Use `colorinfo` to get information about the current color environment.

```
void colorinfo (CInfoRec *cinfo);
```

`cinfo`	A CInfoRec data structure that will hold the information about the color environment. See below for the definition of a CInfoRec.

`colorinfo` fills in the fields of an existing CInfoRec (defined in *ext_anim.h*), which is defined as follows:

```
typedef struct {
    short c_hasColorQD;
    short c_depth;
    short c_has32bitQD;
    short c_inColor;
    short c_curDevH;
    short c_curDevV;
} CInfoRec;
```

`c_hasColorQD` is non-zero if the machine has Color Quickdraw in ROM. `c_depth` is the bit depth of the screen (1-24). `c_has32bitQD` is non-zero if the system has 32-bit Quickdraw installed. `c_inColor` is non-zero if there are colors, 0 if the monitor is set to display gray scales. `c_curDevH` and `c_curDevV` are width and height, respectively, of the current GDevice.

Typically, you'll call `colorinfo` in your Initialization routine, and store the results in global variables. Note that `c_depth` and `c_inColor` could change over time. If these are important to you, consider calling `colorinfo` more regularly. If you're writing a user interface object, you can use the `box_usecolor` function to determine whether you should draw your object in color or not.

gwind_new

Use `gwind_new` to make a new graphic window object.

```
t_gwind *gwind_new (t_object *assoc, t_symbol
*name, short flags, short left, short top, short
right, short bottom);
```

`assoc`	A pointer to your object.
`name`	A name for other objects to be able to access the GWind. All access of GWinds is done through binding to Symbols, similar to the way the Max table object is bound to a Symbol.
`flags`	1 if you want the window created without a title bar, 0 otherwise.
`left`	Left global coordinate of the window.
`top`	Top global coordinate of the window.
`right`	Right global coordinate of the window.
`bottom`	Bottom global coordinate of the window.

`gwind_new` creates a new GWind object. Unlike `wind_new`, all graphics windows you create are immediately visible.

gwind_offscreen

Use `gwind_offscreen` to initialize an Offscreen buffer for a graphics window.

```
void gwind_offscreen (t_gwind *gw);
```

`gw`	A graphics window.

You can also send the offscreen message to a GWind to perform this function.

gwind_get

Use `gwind_get` to return the GWind associated with a symbolic name.

```
t_gwind *gwind_get (t_symbol *name);
```

`name`	Name associated with the graphics window.

If a GWind object is associated with the Symbol name, `gwind_get` returns a pointer to it. Otherwise, it returns 0. You should call `gwind_get` every time one of your methods wants to start accessing a GWind, because you never know whether the GWind object still exists or not. See the example under `gwind_setport` for a typical use of `gwind_get`.

gwind_setport

Use `gwind_setport` to set the current GrafPort to a graphics window.

```
t_gwind *gwind_setport (t_gwind *gw);
```

`gw`	A graphics window.

If the GWind `gw` is visible, `gwind_setport` does a SetPort to its associated Macintosh window and returns the previous GrafPort. If the window is not visible, `gwind_setport` returns 0.

All drawing into a graphics window should be prefaced by a call to gwind_setport, as in the example below. Note also the correct use of gwind_get, assuming the m_windsym field of myObject is a Symbol which (supposedly) holds the name of a graphics window object.

```
void myobject_draw(myObject *x)
{
    GrafPtr save;
    t_gwind *g;
    if (g = gwind_get(x->m_windsym)) { /* does it exist? */
        if (save = gwind_setport(g)) { /* is it visible? */
        /* draw something in the GWind here */
            SetPort(save);
        }
    }
}
```

Offscreen Routines

Although GWinds are designed to use the offscreen and sprite routines, there's no reason why you can't use them in your own window if you wish. As mentioned above, gwind_offscreen will initialize an Offscreen structure for a GWind. You can initialize the offscreen structure for your own window with off_new.

The Offscreen routines take care of "buffering" drawing to minimize unsightly screen flicker. This facility is similar to that provided by 32-bit Quickdraw GWorld functions, and the Offscreen structure transparently uses 32-bit Quickdraw available. The user of the routines does not need to worry about whether GWorlds are being used or not.

Typically, you'll use the Sprite routines to draw into an Offscreen structure. The Offscreen data structure is declared in *ext_anim.h*.

off_new

Use off_new to create a new Offscreen structure.

```
Offscreen *off_new (GrafPtr dest);
```

dest	The window that the Offscreen will draw into.

off_new creates a new Offscreen structure which will have the same dimensions as the GrafPort dest it is linked to. If the size of your window changes, call off_resize. You must create an Offscreen before you can create or use any Sprites. Free an Offscreen with `off_free`, since it's not a Max object.

off_free

Use `off_free` to dispose of an Offscreen.

```
void off_free (Offscreen *os);
```

`os`	The Offscreen structure you want to free.

off_copy

Use `off_copy` to copy an entire Offscreen to its associated GrafPort.

```
void off_copy (Offscreen *os);
```

`os`	The Offscreen structure to be copied.

off_copyrect

Use `off_copyrect` to copy a portion of an Offscreen to its associated GrafPort.

```
void off_copyrect (Offscreen *os, Rect *src);
```

`os`	The Offscreen structure to be copied.
`src`	Rectangle to copy.

The rectangle `src` in the Offscreen will be copied to the same location in the Offscreen's destination GrafPort.

off_maxrect

Use `off_maxrect` to return a rectangle that covers two source rectangles within an Offscreen.

```
void off_maxrect (Offscreen *os, Rect *src1, Rect
*src2, Rect *result);
```

`os`	An Offscreen structure.
`src1`	Rectangle to be covered.

`src2`	Another rectangle to be covered.
`result`	The resulting rectangle that includes both `src1` and `src2` and is within the bounding rectangle of `os` will be placed here.

off_tooff

Use `off_tooff` to copy a BitMap to an Offscreen.

```
void off_tooff (Offscreen *os, PixMapHandle *src,
Rect *srcRect, Rect *dstRect);
```

`os`	An Offscreen structure.
`src`	PixMap or BitMap containing the bits to copy.
`srcRect`	Portion of `src` you want copied.
`dstRect`	Location in the Offscreen where the bits should be copied.

This function copies the pixels in `src` to the Offscreen buffer without copying them to the screen. `src` can also be a pointer to a BitMap.

off_resize

Use `off_resize` to change the size of an Offscreen to match its associated GrafPort.

```
void off_resize (Offscreen *os);
```

`os`	An Offscreen structure.

Call this routine when the user resizes a window containing an Offscreen.

Sprite Routines

Sprites are independent entities that draw an image inside a set rectangle. A Sprite system is owned by an Offscreen object, so for example, there will be a different set of Sprites for each active Graphics Window in Max. Each Sprite has a *priority*, which is used to layer objects from front to back. Sprites can change their priority dynamically. There is no set limit to the number of Sprites or different priority levels. If two or more Sprites are at the same priority level, the one which joined the Offscreen structure first will be drawn in front of the more recent arrival.

The Sprite structure is defined in *ext_anim.h*. You can get away with being ignorant of the fields of a Sprite object, but it can be helpful in some circumstances, such as knowing the Sprite's rectangle.

```
typedef struct sprt {
    struct object s_ob;
    GrafPtr s_dest;      /* screen dest */
    Rect s_rect;         /* rectangle */
    BitMapHandle s_mask; /* mask */
    RgnHandle s_rgn;     /* mask rgn */
    int s_number;        /* sprite number (priority) */
    char s_drawn;        /* is it drawn */
    char s_change;       /* message to sprite proc to go to
                            "next" frame */
    void *(*s_proc)();   /* procedure that draws */
    long s_frame;        /* current frame, used by s_proc*/
    long s_misc;         /* used by s_proc */
    void *s_assoc;       /* an associated object */
    OffScreen *s_owner;  /* owning system */
    struct sprt *s_prev; /* link */
    struct sprt *s_next; /* link */
} Sprite;
```

All of the Sprite drawing routines discussed below (`sprite_move`, `sprite_rect`, etc.) automatically redraw the other sprites on the screen if necessary when the Offscreen owner of the Sprite's destination GrafPort is visible.

sprite_new

Use `sprite_new` to create a new Sprite.

```
Sprite *sprite_new (t_object *assoc, Offscreen
*owner, long priority, Rect *frame, ProcPtr
*drawProc);
```

`assoc`	A pointer to your object.
`owner`	The associated Offscreen structure—where the Sprite will draw.
`priority`	The Sprite's priority number. 0 is the background and higher numbers are more in the foreground.

`frame`	The rectangle in which the Sprite will draw.
`drawProc`	The function that draws the Sprite based on its current state. See below for how to declare it.

This function creates a new Sprite object that will draw in the Offscreen environment of `owner`. The draw procedure `drawProc` should be declared as follows:

```
void myObject_spritedraw (myObject *obj, Sprite
*spr);
```

`obj`	Your object.
`spr`	The Sprite to draw.

In this routine, you can make normal Quickdraw calls (such as PaintRect) and the image will be recorded Offscreen, then copied to its associated GrafPort at the proper time to assure the layering of all the Sprites.

sprite_move

Use `sprite_move` to change a Sprite's relative location.

```
void sprite_move (Sprite *spr, short deltaH, short
deltaV);
```

`spr`	The Sprite to move.
`deltaH`	Horizontal distance in pixels to move the Sprite.
`deltaV`	Vertical distance in pixels to move the Sprite.

This function moves a Sprite's rectangle by `deltaH` pixels horizontally and `deltaV` pixels vertically. The Sprite is erased at its old location and redrawn at the new location. Any other Sprites affected by the move are also redrawn. If you want to redraw your sprite at the same location but with a different appearance, you could use:

```
sprite_move(mySprite, 0,0);
```

sprite_moveto

Use `sprite_moveto` to move a Sprite to a specific location.

```
void sprite_move (Sprite *spr, short h, short v);
```

`spr`	The Sprite to move.
`h`	New left coordinate of the Sprite's rectangle.
`v`	New top coordinate of the Sprite's rectangle.

This function moves the Sprite's rectangle to the specified location. The sprite is erased at its old location and redrawn at the new location. Any other Sprites affected by the move are also redrawn.

sprite_rect

Use `sprite_rect` to change a Sprite's rectangle.

```
void sprite_rect (Sprite *spr, Rect *newRect, short
change, short next);
```

`spr`	The Sprite whose rectangle you want to change.
`newRect`	The new rectangle.
`change`	Value to store in the Sprite's `s_change` field that can be interpreted by your Sprite's drawing procedure in any way it wants.
`next`	Value to store in the Sprite's `s_next` field that can be interpreted by your Sprite's drawing procedure in any way it wants.

This function changes the Sprite's rectangle to `newRect` and redraws it.

sprite_redraw

Use `sprite_redraw` to redraw a Sprite.

```
void sprite_redraw (Sprite *spr, short deltaH,
short deltaV, short change, short next);
```

`spr`	The Sprite to redraw.
`deltaH`	Horizontal distance in pixels to move the Sprite.
`deltaV`	Vertical distance in pixels to move the Sprite.
`change`	Value to store in the Sprite's `s_change` field that can be interpreted by your Sprite's drawing procedure in any way it wants.
`next`	Value to store in the Sprite's `s_next` field that can be interpreted by your Sprite's drawing procedure in any way it wants.

sprite_redraw is like `sprite_move`, but also allows you to set the change and next fields of the Sprite.

sprite_erase

Use `sprite_erase` to erase a Sprite.

```
void sprite_erase (Sprite *spr);
```

`spr`	The Sprite to erase.

This function erases a Sprite, filling in any other Sprites which may have been lurking behind it.

sprite_newpriority

Use `sprite_newpriority` to change the priority of a Sprite.

```
void sprite_newpriority (Sprite *spr, long
priority);
```

`spr`	A Sprite.
`priority`	The Sprite's new priority. 0 is background and higher numbers are increasingly in the foreground.

This function assigns a new priority to a Sprite and redraws all the elements of the Offscreen structure that owns the Sprite necessary to reflect the change in priorities.

A Sprite Example

The following example includes the key methods of an object that draws ovals using Sprites. It shows several useful techniques, such as isolating the data structures used for drawing in a Qelem from those changed in int methods.

This object has four inlets, for each coordinate of an oval's rectangle. The data structure is defined as follows:

```
typedef struct oval {
    struct object o_ob;
    long o_priority;     /* sprite priority */
    Sprite *o_sprite;    /* the sprite */
    Rect o_bounds;       /* where it is */
    Rect o_dbounds;      /* where it is drawing */
    void *o_qelem;       /* Qelem */
    t_symbol *o_sym;     /* symbol of GWind */
} Oval;
```

Here's the Initialization routine:

```
void main(void *p)
{
    setup((t_messlist **)&OvalClass, oval_new, oval_free,
    (short)sizeof(Oval), 0L, A_SYM, A_LONG, A_DEFLONG,
    A_DEFLONG, A_DEFLONG, A_DEFLONG, 0);
    addint(oval_int);
    addinx(oval_in1,1);
    addinx(oval_in2,2);
    finder_addclass("Graphics","oval");
}
```

Here is the object's instance creation function. Note that its Sprite isn't created until it's needed, within the queue function `oval_qfn`. We always reference the GWind through a Symbol each time we draw, since we have no guarantee the GWind is still around when we want to draw in it.

```
void *oval_new(t_symbol *windName, long priority, long
left, long top, long bottom, long right)
{
    Oval *x;
    x = (Oval *)newobject(OvalClass);
    intin(x,3);
    intin(x,2);
    intin(x,1);
    SetRect(&x->o_bounds, (short)left, (short)top,
            (short)bottom, (short)right);
    x->o_dbounds = x->o_bounds;
    x->o_sym = windName;
    x->o_priority = priority;
    x->o_qelem = qelem_new(x,oval_qfn);
    x->o_sprite = 0;
    return (x);
}
```

Here are the int methods. These set the coordinates of the o_bounds rectangle and the leftmost one sets the Qelem to draw the oval. You can't draw on the screen directly in response to an int or bang message, since your method may be executing at interrupt level.

```
void oval_bang(Oval *x)
{
    qelem_set(x->o_qelem);
}
void oval_int(Oval *x, long left)
{
    x->o_bounds.left = left;
    oval_bang(x);
}
void oval_in1(Oval *x, long top)
{
    x->o_bounds.top = top;
}
void oval_in2(Oval *x, long right)
{
    x->o_bounds.right = right;
}
void oval_in3(Oval *x, long bottom)
{
    x->o_bounds.bottom = bottom;
}
```

The queue function is where all the action is.

```
void oval_qfn(Oval *x)
{
    GrafPtr gp;
    t_gwind *it;

    x->o_dbounds = x->o_bounds; /* make a copy of the new 
rect */
    it = gwind_get(x->o_sym);

    if (!it || !(gp = gwind_setport(it))) {
                        /* doesn’t exist or not visible? 
*/
        return;
    }
    if (it->g_off) {     /* if there’s an Offscreen */
        if (!x->o_sprite)    /* need to make a new Sprite? 
*/
            x->o_sprite = sprite_new(x,it->g_off,x-
>o_priority,
                &x->o_dbounds,oval_spritedraw);
        sprite_rect(x->o_sprite,&x->o_dbounds,0,0);    /* 
draw */
    }
    SetPort(gp);
}
```

Here is the Sprite’s `drawProc`. Note that we get the drawing bounds from the Sprite’s rectangle `s_rect`. This isn’t necessary, but ensures that you’ll always be drawing where the Sprite thinks you’re supposed to be drawing.

```
void oval_spritedraw(Oval *x, Sprite *s)
{
    EnterCallback();
    x->o_dbounds = s->s_rect;
    PaintOval(&x->o_dbounds);
    ExitCallback();
}
```

Finally, here’s the object’s free function.

```
void oval_free(Oval *x)
{
    EnterCallback();
    qelem_free(x->o_qelem);
    if (x->o_sprite)
        freeobject(x->o_sprite);
    ExitCallback();
}
```

This example should demonstrate how to draw things in GWinds with Sprites. The actual Max oval object is a bit more complicated than this one (it can draw in different shapes and colors) but the Sprite techniques it uses are identical to those presented in this example.

Chapter13: Writing Objects for the Timeline

The Max Timeline object is, at the most basic level, a system for sending messages at pre-determined times. The Timeline consists of an editing window of events and numerous auxiliary objects that allow communication between Patchers and the events in the window.

There are several ways in which an external object can extend the capabilities of the Max Timeline object. First, you can write an object that resides inside a patcher that is used as a Timeline action. An example is the external object tiCmd, the source for which is distributed in the SDK. This object "registers" itself with the Timeline, which causes it to receive data held in the events in a Timeline track. It then passes the data to other Max objects via its outlets.

Another possibility is an object that controls a Timeline, such as the objects thisTimeline and thisTrack. Here, it is just a matter of looking for an object bound to a specific symbol when your object is created. Then you can send this object messages to control it. A third opportunity is to write an external that is itself an Action. This is similar to writing an object like tiCmd but involves an extra step of displaying an icon in the Timeline window and, optionally, some type of configuration window or dialog when the user double-clicks on this icon. The key thing to note about all the work needed to make an object that interfaces with the Timeline described in this section is that all these features can be added to an existing normal object. If the object is not being used in the context of the Timeline, the object can be written so that it still does something and its Timeline interface is disabled.

The next chapter describes the process of writing an Editor object for the Timeline. Editors *are* specific to the Timeline world and cannot lead a double life as a user interface object in a Patcher window. However, the way an Editor works will seem familiar to anyone who has written a user-interface object for the Patcher.

Registration

The key concept in writing an object that interfaces to the Timeline is that of *registration*, which is the process of advertising that your object accepts a certain symbol as a message. After having registered one or more messages, the timeline knows enough to guide the user into making events that will send to your object only those messages for which it has registered.

Registration is performed in your object creation function using the routine `message_patcherRegister` (if your object is loaded directly as an Action, it uses `message_register`, to be discussed later). You can register for more than one message. In your object free function, you must unregister every message that you've registered using `message_patcherUnregister`. This will disconnect your object from any Timeline Events that could potentially send it a message.

Each message you register has two important components. The first is the *message name*, which determines the symbol that an Event will send back to you. This can be any symbol and will be descriptive of a command or parameter. In the tiCmd object, the message name is passed as an argument. The second important component is the *message data type*. This is also a symbol, but it determines the sort of Editor that can be used to display the Event that will hold the message in the Timeline window. It also determines the arguments to the message your object will receive when time passes by the left edge of an Event in the Timeline. Several standard Editors are included in the Timeline object, as well as a few external object editors. The standard message box edits the generic data type message, and the Timeline version of the number box edits either int or float data types. The external editors etable and efunc edit int data types, and the external editor edetonate edits the list data type.

As an example, if you make an object with int as its message data type, the user will have a choice, assuming the standard configuration, of making an event with either the int, efunc or etable editors. It is important to understand the difference between a message and a data type. The data type does *not* determine the format of the message you will receive. All messages from the timeline to your object are sent by the function `typedmess`, which will respect the argument list you provide to the `addmess` function. While is true that an editor for the int data type will always send a message that contains one integer as an argument, other data types can be defined by your own convention.

A final option in message registration is to provide a *receiver name* for your message. The movie object does this when it registers the start message for the movie data type.

As with any other specification of a message in an external object, you need to write some method that is bound to the symbol you have registered, or provide an anything method. If you've registered a message foo with a data type of int, you can write the accompanying method as:

```
void myobject_foo (myObject *x, long n);
```

…and binding it at initialization time with:

```
addmess(myobject_foo, "foo", A_LONG, 0);
```

Other data types with multiple message arguments will often use the A_GIMME form.

eventEnd

The eventEnd message is sent by the message editor when time passes the trailing edge of its event rectangle.

BINDING

```
addmess (myobject_eventEnd, "eventEnd", A_CANT, 0);
```

DECLARATION

```
void myobject_eventEnd (myObject *x, t_symbol
*message);
```

`message`	The same message that was sent by the message editor when time passed the leading edge of this event.

Don't confuse this eventEnd message sent by the message editor with the eventEnd message that the Timeline sends to editors. The message editor's eventEnd message is in fact its response to receiving an eventEnd message from the Timeline.

message_patcherRegister

Use `message_patcherRegister` to register your object for a particular message and data type.

```
void message_patcherRegister (t_symbol *message,
t_symbol *dataType, t_object *theObject, t_symbol
*objectName, t_patcher *p);
```

`messsage`	The message your object wants to receive. This will appear in the pop-up menu when the user makes a new event in a Timeline track.
`dataType`	A standard message data type: int, float, or the generic message. This determines the editor that can display the data sent to your object in the Timeline window.
`theObject`	A pointer to your object, or the receiver of this message.
`objectName`	If your object doesn’t have a name, pass 0L, otherwise pass a symbolic name for your object. A Timeline editor will receive this name but would have to be specially written to pass the name back to you.
`p`	Your object’s parent patcher, bound to the symbol #P in your instance creation function. You should save the patcher value in your object because you will need it to unregister the message when your object is freed.

For objects created in a patcher that is being used as a Timeline Action, this function registers a message that can be sent to the object from the Timeline. It is normally called in your object’s instance creation function. If the patcher `p` is not connected to a Timeline, `message_patcherRegister` will do nothing.

Here is an example of registering a message foo of data type int using `message_patcherRegister`.

```
message_patcherRegister(gensym("foo"), gensym("int"),
myObject, 0L,
       myObject->m_patcher = gensym("#P")->s_thing);
```

message_patcherUnregister

Use `message_patcherUnregister` to unregister an object previously registered with `message_patcherRegister`.

```
void message_patcherUnregister (t_symbol *message,
t_symbol *dataType, t_object *theObject, t_symbol
*objectName, t_patcher *p);
```

`messsage`	The message your object registered.
`dataType`	A standard message data type: int, float, or the generic message.
`theObject`	A pointer to your object, or the receiver of this message.
`objectName`	The objectName argument you passed to `message_patcherRegister`.
`p`	Your object's parent patcher, bound to the symbol #P in your instance creation function.

This function expects the same arguments that were previously passed to `message_patcherRegister`. It removes all references to the message message of data type `dataType` to sent to the object `theObject` from the track connected with the patcher `p` so that when the Timeline plays no such messages will be sent. You must balance each call to `message_patcherRegister` with a call to `message_patcherUnregister`. If the patcher `p` is not connected to a Timeline, this function does nothing.

Writing an Action External

Instead of loading a patcher to use as an Action, a Timeline user can place an external object in the *tiAction* folder and choose it from the Track menu. Such an object could also be usable as a normal patcher object. To think of it another way, it is possible to add the Action capability to an existing external object. The additional steps for writing an Action external is relatively simple. In general the concept is similar to the Timeline-compatible external objects discussed above: you write methods bound to symbols with `addmess`, then register these symbols with the Timeline.

The first additional step is to check to see if something is bound to the symbol #A in your object creation function. If there is, it will be a pointer to the Timeline track for which your external object is an Action. You will need to pass this value to the function `message_register`—analogous to `message_patcherRegister` for Actions without a patcher.

Next, you must implement a method to respond to the actionIcon message to display an icon for your action in the Timeline window.

actionIcon

The actionIcon message is a request by the Timeline to draw an icon representing your action.

BINDING

```
addmess (myobject_actionIcon, "actionIcon", A_CANT,
0);
```

DECLARATION

```
void myobject_actionIcon (myObject *x, Rect
*drawHere);
```

`drawHere`	Draw something icon-like within this 16x16 pixel rectangle.

message_register

Use `message_register` to register a message for an Action object.

```
void message_register (t_symbol *message, t_symbol
*dataType, t_object *theObject, t_symbol
*objectName, void *track);
```

`messsage`	The message your object wants to receive. This will appear in the pop-up menu when the user makes a new event in a Timeline track.
`dataType`	A standard message data type: int, float, or the generic message. This determines the editor that can display the data sent to your object in the Timeline window.
`theObject`	A pointer to your object, or the receiver of this message.

`objectName`	If your object doesn't have a name, pass 0L, otherwise pass a symbolic name for your object. A Timeline editor will receive this name but would have to be specially written to pass the name back to you.
`track`	Your object's parent track, bound to the symbol #A in your instance creation function. You should save this value in your object because you will need it to unregister the message when your object is freed. If the value bound to #A is 0, you should not call message_register.

This function registers a message for an instance of an object that is an Action.

message_unregister

Use `message_unregister` to unregister an object previously registered with `message_register`.

```
void message_unregister (t_symbol *message,
t_symbol *dataType, t_object *theObject, t_symbol
*objectName, void *track);
```

`messsage`	The message your object registered.
`dataType`	A standard message data type: int, float, or the generic message.
`theObject`	A pointer to your object, or the receiver of this message.
`objectName`	The objectName argument you passed to `message_patcherRegister`.
`track`	Your object's parent track, bound to the symbol #A in your instance creation function.

This function shold be called in your instance free function for all messages previously registered with `message_register`.

Chapter 14: Writing Editors for the Timeline

The Timeline window has a similar structure to the Patcher window. A Timeline object holds a linked list of a data structure called an Event, analogous to a Box in the Patcher. Events hold both the data (time-tagged messages) in the Timeline and refer to the objects that know how to edit and display it. Each Event is the header of an instance of a special type of Max object known as an Editor, just as each Box is the header of a user interface object. In fact, the Event borrows a number of fields from the `t_box` structure—which allows certain Patcher window routines to be used in the Timeline object. Many of the same concepts used in Patcher user interface objects apply to writing Editors for the Timeline, although it is somewhat awkward to combine both roles in the same external object (and the exact techniques for doing it are not discussed here).

Registering A Timeline Editor

Essentially, there are three basic steps to writing an Editor. First, there is the process of registering the Editor for one or more data types in the initialization routine. Next, you must initialize the Event structure when a new instance of your object is created. And finally, messages to support the required and optional messages for Timeline Editors must be written. These messages concern drawing, user interaction, sending the data to the objects in Actions linked to the Timeline window, and saving the Event's data in a Timeline file.

The first step in creating an editor object that works with the Timeline is to connect it with particular dataTypes that you can edit using `editor_register`.

editor_register

Use `editor_register` to register an editor for a particular data type.

```
void editor_register (t_symbol *dataType, t_symbol
*name                method new, method menu, method
update);
```

`dataType`	The type of data your editor can edit and display. You can invent any name you like for a data type, but your editor will never be usable in the Timeline window unless there is an object in an Action that can register a message of this type. Editors currently exist for the data types int, float, list, and the generic type message.
`name`	The name of your editor, used to identify it when saving a Timeline file and in the new event pop-up menu when multiple editors exist for the same `dataType`.
`new`	Method for making a new event from a file. Described in the instance creation section below.
`menu`	Method for making a new event from a new event pop-up menu. Described in the instance creation section below.
`update`	Method for updating an event. Described in the messags section below.

In your editor object's initialization routine, after calling `setup` to initialize your class, you call `editor_register` to make the existence of your editor known to the Timeline object. The Timeline classifies editors by data type, which is a symbol that describes a type of message that the Editor would send to an object at any particular moment in time. The QuickTime movie Editor invented a type called movie that is used in conjunction with the revised Max movie object that works in the context of the Timeline. Your editor can register for multiple data types, and when new instances are created, your object is told what data type the instance requires.

Editor Instance Creation and the Event Structure

Your Editor object must begin with the Event data structure (declared in the include file *ext_event.h*):

```
typedef struct myEditor {
    Event m_event;
    ...rest of your editor fields here
} myEditor;
```

This structure holds the location of the Editor instance in the timeline window as well as other information needed by both the Timeline object and your object. Here is a description of each of the fields in an Event.

```
typedef struct event {
```

`t_object e_obj;`	
`struct smallbox *e_box;`	Pointer to Smallbox (or Box) that holds the rectangle.
`struct event *e_upd;`	Update list link (used internally for spooling events).
`Rect *e_rect;`	Pointer to rectangle within e_box.
`struct event *e_next;`	Linked list of Events in the Timeline Track.
`void *e_track;`	Pointer to owning Track.
`struct oList *e_assoc;`	List of associated objects (internal use).
`t_object *e_o;`	Pointer to object in unitary receiver case.
`t_symbol *e_label;`	Descriptive text for Event locate pop-up menu.
`long e_start;`	Event start time (in milliseconds).
`long e_duration;`	Event duration (in milliseconds).
`t_symbol *e_dataType;`	Data type of the message.
`t_symbol *e_message;`	Message selector that is sent.
`t_symbol *e_editor;`	Editor name.
`struct editor *e_edit;`	Internal information about the Editor.
`void *e_saveThing;`	Internal temporary variable.
`void *e_thing;`	Internal unused variable.
`short e_wantOffset;`	If the track containing the editor has been collapsed, this field stores the previous vertical offset of the Event's rectangle from the top of the track.
`Boolean e_active;`	Currently unused.
`Boolean e_preview;`	Should be non-zero, not currently used.
`Boolean e_constantWidth;`	If this flag is non-zero, the Event's duration is recalculated so that the Event rectangle stays the same width at every scale change. This mode is used by the Number box editor.
`Boolean e_editable;`	Flag is non-zero if clicking on the Event modifies its contents.
`Boolean e_smallbox;`	Flag is non-zero if the Event uses the Smallbox data structure (defined in ext_user.h) rather than the Box data structure. Only the Number box editor currently uses the Box data structure in order to reuse code from the Number box user interface object.

```
} Event;
```

Your editor's instance creation function is responsible for initializing the Event's data structures. To do this it calls `event_new` and `event_box`. The first several arguments to the Event's object creation function are standardized (by convention) and many of these can be directly passed to the two Event initialization functions. The Event's instance creation function should be of the following form:

```
void *myEditor_new (t_symbol *dataType, short argc,
t_atom *argv);
```

`dataType`	The data type for the event being created. If your Editor object has only registered for one data type, you won't need to pay too much attention to this argument, although you should pass the argument to `event_new` rather than a hard-coded pointer to the Symbol under which you registered.
`argc`	Count of Atoms in `argv`.
`argv`	An array containing the following data:

Index	*Constant*	*Description*
0	`ED_TRACK`	Pointer to the Event's parent Track (`A_OBJ`)
1	`ED_MESSAGE`	Symbol that specifies the message you'll send to an Object when the Timeline tells you to (`A_SYM`)
2	`ED_START`	Start time of the Event in milliseconds (`A_LONG`)
3	`ED_DURATION`	Duration of the Event in milliseconds (`A_LONG`)
4	`ED_TOP`	Top of Event rectangle in the Track (`A_LONG`)
5	`ED_BOTTOM`	Bottom of Event rectangle in the Track (`A_LONG`)

There may be additional arguments (if `argc` is greater than 6) that are your own user-defined parameters, including the contents of the Event message data.

event_new

Use `event_new` to initialize an Event.

```
void event_new (Event *evnt, void *track, t_symbol
*dataType, t_symbol *message, t_symbol *editor,
t_symbol *label, long start, long duration, long
flags, t_box *box);
```

`evnt`	The Event to initialize. It should be a pointer to your object, since it begins with an Event header.
`track`	Pass the `a_w.w_obj` field of the `ED_TRACK` argument received by your instance creation function.
`dataType`	Pass the dataType argument passed to your instance creation function.
`message`	Pass the `a_w.w_sym` field of the `ED_MESSAGE` argument received by your instance creation function.
`editor`	The name of your editor, the same Symbol passed to `editor_register`.
`label`	Any text; often the same Symbol passed as the editor argument is used here.
`start`	Pass the `a_w.w_long` field of the `ED_START` argument received by your instance creation function.
`duration`	Pass the `a_w.w_long` field of the `ED_DURATION` argument received by your instance creation function.
`flags`	See the constants listed below.
`box`	This argument is 0L if you are using the normal Smallbox structure to hold the Event display information. If the argument is non-zero, a Smallbox is not allocated. Instead, an already allocated and initialized Box you pass is used instead. You're responsible for freeing this Box when your free function is called. If you pass 0L, the memory used by the Smallbox is freed for you.

This function initializes most of the fields of an Event (like `box_new`, it is passed an existing Event, it does not create one). The constants for `flags` are:

`#define F_GROWY`	2	Can grow in y direction by dragging, appropriate for text-based objects
`#define F_GROWBOTH`	32	Can grow independently in both x and y dimensions
`#define F_CONSTANTWIDTH`	16	Duration is sensitive to display scaling, sets the flag `e_constantWidth` in the Event structure

You should use either F_GROWY or F_GROWBOTH, and optionally F_CONSTANTWIDTH.

event_box

Use `event_box` to set the rectangle of an Event.

```
void event_box (Event *evnt, short top, short
bottom);
```

`evnt`	A pointer to your object.
`top`	Pass the `a_w.w_long` field of the `ED_TOP` argument received by your instance creation function.
`bottom`	Pass the `a_w.w_long` field of the `ED_BOTTOM` argument received by your instance creation function.

Editor Instance Creation Example

This function initializes the Event rectangle, about which more will be said shortly. It must be called after `event_new` so that the start and duration values can be used to calculate the current position of the Event rectangle.

Here's a standard implementation of the instance creation function showing the use of `event_new` and `event_box`. Here we're initializing an Event for an editor whose name is the same as the data type that uses the standard Smallbox.

```
void *myEditor_new(t_symbol *dataType, short argc, t_atom
*argv)
{
    MyEditor *x;
    x = (MyEditor *)newobject(myEditor_class); /* create
instance */
    /* initialize the Event */
    event_new((Event)x,         /* event */
        (void *)argv[ED_TRACK].a_w.w_obj, /* track */
        dataType,               /* data type */
        argv[ED_MESSAGE].a_w.w_sym,    /* message */
        dataType,               /* editor name */
        argv[ED_MESSAGE].a_w.w_sym,    /* label */
        argv[ED_START].a_w.w_long,  /* start */
        argv[ED_DURATION].a_w.w_long,  /* duration */
        (long)F_GROWY | F_CONSTANTWIDTH, /* flags */
        0L);                           /* box */
    event_box((Event)x, (short)argv[ED_TOP].a_w.w_long, /*
top */
        (short)argv[ED_BOTTOM].a_w.w_long); /* bottom */
    /* do other initialization here */
    return (x);
}
```

Editor Menu Function

This function, having been supplied to `editor_register` at initialization time, is called when the user creates a new event in a Timeline Track.

```
void *myEditor_menu (t_symbol *dataType, t_symbol
*message, void *track, void *obj, long start, Point
pt);
```

dataType	The data type your editor has registered to edit. Pass this to your instance creation function.
message	The message for this event. Pass this to your instance creation function.
track	Parent track holding the event. Pass this to your instance creation function.

obj	Receiver of the messages sent by this event. In most cases, you can ignore the `obj` argument, since it will also be passed to your editor when it receives the eventStart and eventEnd messages. However, if your editor will be displaying data that is contained in the object, it will be important to store this reference. The QuickTime movie editor stores this information because it needs to access the movie stored in the Max movie object in order to display its thumbnails.
start	Start of this event (in milliseconds).
pt	Location where the user clicked to place this event. It should be the upper left-hand corner of your event rectangle.

This function, having been supplied to `editor_register` at initialization time, is called when the user creates a new event in a Timeline Track. The menu function must return the result of the creation routine: either a pointer to the newly created object or 0 if there was an error in creating it.

Typically, in the menu function you will assemble an array of Atoms to pass to your object's creation routine. This array will take the same argument format as the creation function would receive directly from the timeline object when a file is being read in.

Here is an example of a typical menu function for an Editor that deals with a message for a data type of int. The constants used are declared in *timelineEvent.h*.

```
void *myEditor_menu(t_symbol *dataType, t_symbol *message,
    void *track, void *obj, long start, Point pt)
{
    MyEditor *x;
    long dur;
    t_atom a[18];

    SETOBJ(a + ED_TRACK,(void*)track);  /* event’s track */
    SETSYM(a + ED_MESSAGE,message);    /* event’s message */
    SETLONG(a + ED_START,start);    /*event start */
    dur = track_pixToMS(track,132);
    SETLONG(a + ED_DURATION,dur);   /*event duration */
    SETLONG(a + ED_TOP,(long)pt.v); /*box top */
    SETLONG(a + ED_BOTTOM,(long)pt.v+64);   /* box bottom */
    x = myEditor_new(dataType,6,a);
    return (x);
}
```

event_spool

Use `event_spool` to cause an Event to be redrawn.

```
void event_spool (Event *evnt);
```

`evnt`	Event to be redrawn.

Call `event_spool` after making changes that would affect the appearance of an event. You need not do this in your instance creation routine. Note that if you just want to redraw the state of your object in a function outside of the context of the standard Editor messages, you can call `event_spool` without needing to do any of the setup discussed in the section above.

Messages Sent to Editors By the Timeline

In order to have a working editor, you must implement the psave, eventStart, and update messages. The concepts behind most of these messages is quite similar to those used for writing user interface objects for the patcher. This section describes each message, along with the Timeline routines that will be useful in writing a method to respond to the message.

psave

The psave message is sent when your editor needs to save an event.

BINDING

```
addmess (myobject_psave, "psave", A_CANT, 0);
```

DECLARATION

```
void myobject_psave(myObject *x, Binbuf *dest);
```

`dest`	The Binbuf where you should write out a message to save your object.

This message is sent to your object to save its contents when an event is being copied or a Timeline file is being saved. The first nine arguments that you must save are standard for every Event, and can be placed in an array of Atoms for you with the function `event_save`. After that, you can put additional data needed to restore your Editor instance. You then pass this array to `binbuf_insert` using `dest` as the first argument. See the example after the description of the `event_save` function. Note that the arguments to the psave method are the same as for the Patcher user interface object case and the method for the save message for normal objects.

eventStart

The eventStart message is sent when time passes over the left edge of your Event's rectangle.

BINDING

```
addmess (myobject_eventStart, "eventStart", A_CANT,
0);
```

DECLARATION

```
void myobject_eventStart(myObject *x, t_object
*receiver);
```

`receiver`	An object inside an Action Patcher or an Action External that has been linked to your editor. You should send it the data your editor contains. See the example below.

This message may be sent to you at interrupt level; therefore the usual restrictions on the behavior of your method apply. For an Editor object that holds a single data element, the usual implementation of eventStart involves sending a message to the receiver argument. For a more complex object that will schedule a series of events over the duration of the Event, a call to a Timeline-relative scheduling facility is made. First let's look at how the simple case is implemented. The following eventStart method sends a single integer to a receiver. This is similar to the implementation used by the Number box Editor for the int data type. Note that it does not send the message selector int to the receiver but rather uses the `e_message` field of the Event.

```
void myNumberEditor_eventStart(MyEditor *x, t_object
*receiver)
{
    t_atom val;
    val.a_w.w_type = A_LONG;
    val.a_w.w_long = x->m_value;
    typedmess(receiver,x->m_event.e_message,1,&val);
}
```

event_save

Use `event_save` to prepare the first nine arguments of an Event for saving.

```
void event_save (Event *evnt, t_atom *buf);
```

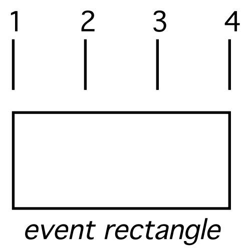
`evnt`	Event to be saved.
`buf`	Array of at least nine Atoms where `event_save` will place the standard information needed for saving an Event.

This function copies the standard first nine items of an Event to an array of Atoms so you need not worry about the details of saving the Event data structure. Here's an example implementation of a psave method that uses `event_save` and then adds an additional piece of data before calling `binbuf_insert`.

```
void myEditor_psave(MyEditor *x, void *buf)
{
    t_atom buffer[10];
    event_save((Event)x,buffer);
    SETLONG(buffer+9,x->m_value);
    binbuf_insert(buf,0L,10,buffer);
}
```

Scheduling Events

Now, let's imagine we're writing an Editor for an object that holds four integer values that will be sent out over the duration of an Event, as follows:



The Timeline object knows nothing about the internal structure of an Editor, so it won't automatically call our eventStart function for the last three messages we want to send. And we can't just create a Clock to schedule these events, since the "current time" used by the Timeline object is not simply the time of the internal Max scheduler, but can be manipulated by other processes. Even in the simplest case, we would want these messages not to be sent if the user stops the Timeline from playing in the middle of the Event. To handle these situations, the Timeline object keeps an internal list of tasks to do that are scheduled to occur at Timeline-relative times. You put tasks on this list by using the function `event_schedule`.

event_schedule

Use `event_schedule` to schedule a message for a later Timeline-relative time.

```
void event_schedule (Event *evnt, method fun,
t_object *receiver, void *arg, long delay, long
flags);
```

`evnt`	Event scheduling this function.
`fun`	Function you want to be called when the Timeline reaches the specified time. See below for how to declare this function.
`receiver`	The receiver of the message you'll be sending.
`arg`	Any additional argument that will be passed to your function.
`delay`	The delay, in timeline "milliseconds," until the function should be called.
`flags`	Optionally, one of the following constants. Either `L_DIEONSTOP` (1) if the function should not be called if the Timeline stops before time reaches the specified point, or `L_MUSTHAPPEN` (2) if the function should be called even if the Timeline stops before time reaches the specified point. Generally, the latter is used when scheduling things like MIDI note-off messages.

Here is the implementation of sending the four evenly spaced messages based on the duration of an Event that uses `event_schedule`. Assume that the four integer values are stored in an array `m_values`. To know which message we're sending we need a counter into this array `m_counter`.

The implementation consists of two functions, one that responds to the message eventStart that sends out the first value and the other that is scheduled by `event_schedule` that sends the other three.

```
void myEditor_eventStart(myEditor *x, t_object *receiver)
{
    t_atom at;

    at.a_w.w_long = x->m_values[0];    /* send first value
*/
    at.a_type = A_LONG;
    typedmess(receiver,x->m_event.e_message,1,&at);
    x->m_delay = (long)((double)x->m_event.e_duration/3.0);
        /* calculate interval between events */
    x->m_counter = 1; /* next value to send */
    event_schedule(x,myEditor_tick,receiver,0L,x->m_delay,
(long)L_DIEONSTOP);
}
void myEditor_tick(myEditor *x, t_object *receiver)
{
    t_atom at;

    at.a_w.w_long = x->m_values[x->m_counter++]; /* send
next value */
    at.a_type = A_LONG;
    typedmess(receiver,x->m_event.e_message,1,&at);
    if (x->m_counter < 4)                          /*
reschedule */
        event_schedule(x,myEditor_tick,receiver,0L, x-
>m_delay,     (long)L_DIEONSTOP);
}
```

update

The update message is sent when your editor should redraw an Event.

BINDING

```
addmess (myobject_update, "update", A_CANT, 0);
```

DECLARATION

```
void myobject_update (myObject *x, Rect *updateBox,
Boolean preview);
```

`updateBox`	The part of the Event's rectangle to redraw. This may not be your entire Event rectangle.
`preview`	Currently always non-zero. If zero, it indicates you should draw your object in a faster way.

This message is sent when you're Editor is to draw its data in the Timeline window. One difference between the update message in the Timeline context and the one in the Patcher window is that you are passed an update rectangle `updateBox` which covers the area of the window being updated. If only a part of your Event rectangle intersects the `updateBox`, you can avoid drawing all of your data, speeding up the drawing of the Timeline window. It is especially critical to pay attention to the `updateBox` if your object draws its data slowly (as is the case with the QuickTime movie editor). If your object's Event rectangle is entirely outside of an area of the Timeline window being drawn, your object's update method will not be called.

When the update method is called, the Event rectangle (`*e->e_rect`, note that it is a pointer, unlike the Box rectangle), has been properly offset so that you can draw into it. However, be careful not to draw outside of the rectangle or the `updateBox`. You need not draw the rectangle frame, just the contents. If you calculate any internal variables based on the size of the rectangle, be prepared for the size to change between update messages (for example, when the user zooms in or out). Generally, you should always check for a change of the rectangle's size in the update method before drawing.

info

The info message is sent when your event is selected and the user chooses Get Info… from the Max menu.

BINDING

```
addmess (myobject_info, "info", A_CANT, 0);
```

DECLARATION

```
void myobject_info (myObject *x);
```

If you bind a method to the info message, the Get Info... item in the Max menu will automatically be enabled when your object is selected. Typically, you will put up a dialog box allowing the user to change some aspect of the data stored in the editor or parameters of the editor's display. Note that if the dialog box changes the appearance of your Event, you must tell the Timeline object to redraw it by calling `event_spool`.

event_avoidRect

Use `event_avoidRect` to position a dialog box relative to your Event.

```
void event_avoidRect (Event *evnt, short dialogID);
```

`evnt`	Your Event.
`dialogID`	Resource ID of a DLOG resource. `event_avoidRect` will modify the resource's dialog window rectangle.

Analogous to `patcher_avoidbox`, `event_avoidRect` changes the coordinates of a dialog box so that will be positioned, when possible, directly below the Event being edited. If your Editor uses a dialog box in its info method, use `event_avoidRect` before calling GetNewDialog.

dblclick

The dblclick message is sent when the user double-clicks on your event.

BINDING

```
addmess (myobject_dblclick, "dblclick", A_CANT, 0);
```

DECLARATION

```
void myobject_dblclick (myObject *x, Point pt,
short mods);
```

`pt`	Location of the double-click.
`mods`	The modifers field of the EventRecord returned by GetNextEvent for this mouse click event, indicating whether the shift, option, command, caps lock, or control keys were pressed.

This message is sent to your editor when the user double-clicks on the Event rectangle. If your object displays or edits data in an auxiliary window, you could display the window in response to this message.

Messages for Editors of Editable Events

An Editor can edit the data contained in an Event directly in the Timeline window, in an auxiliary window, or not at all. Examples of the first type of Editor are the message box, the number box, and the **efunc** editor for the **funbuff** object. Examples of the second type are **etable** and **edetonate**, and an example of the third is **emovie**. If an editor responds to either the click or key messages, the Timeline treats its events as “editable” and allows mouse clicks within the Event rectangle to be passed to the Editor, rather than used for dragging the Event around in the Track to change its start time or vertical position.

Note that the names given to the messages idle, click, and key are the same messages that an object would receive were it to put up its own window. Thus, if you wish your object to be editable and have its own auxiliary window, you must make the auxiliary window “owned” by another object. However, since the Max text editor window belongs to its own instance of the `t_ed` object class, it would be possible to use a text editor as an auxiliary window for a Timeline editor without worrying about these considerations.

The *target event* is the event that will receive keyboard input and thus must be “editable” according to the criteria discussed above. If an event is the Target it can also receive the click message. It can also handle menu commands such as cut, copy, and paste if it defines these messages as well as a chkmenu message to enable them (see the description of the chkmenu message in Chapter 10 above). Typically the Target event is the last event that the user clicked on and selected. A Target event will always have a marquee around it in the Timeline window.

idle

The idle message is sent to track the cursor when it is over your Event rectangle.

BINDING

```
addmess (myobject_idle, "idle", A_CANT, 0);
```

DECLARATION

```
void myobject_idle (myObject *x, Point pt, short
*within);
```

`pt`	Current location of the cursor in local coordinates.
`within`	Set `within` to 1 if the mouse is within the “editing” portion of the event rectangle and you therefore would like a mouse click in this region to be passed to your editor in a click message. `within` should be set to -1 if you have changed the cursor in this call to the idle method and/or do not wish to receive the click message under any circumstances. `within` should be set to 0 if you have not set the cursor and wish to have the Timeline use its standard technique of treating a mouse click in the Event rectangle. The standard technique passes a click message to the Editor if the click was on the border of the Event rectangle or one pixel in from the border. This decision about what to do with a click on an event is made in the Editor’s idle method immediately before the Editor would receive the click message.

The idle message is sent to your object to track the cursor when it is within the Event rectangle. You can change the cursor depending upon the location of `pt` or display information in the legend with `track_drawDragParam` (see below).

Before sending you the idle message, the Timeline object adjusts your Event rectangle so it is relative to the top of its window rather than the top of your Event's track.

click

The click message is sent when the user clicks on your editable Event.

BINDING

```
addmess (myobject_click, "click", A_CANT, 0);
```

DECLARATION

```
void myobject_click (myObject *x, Point pt, short
dbl, short modifiers);
```

`pt`	Location of the mouse click in local coordinates.
`dbl`	Non-zero if this is a double-click. Note that you will never receive a dblclick message if your object responds to a click message.
`modifiers`	The modifers field of the EventRecord returned by GetNextEvent for this mouse click event, indicating whether the shift, option, command, caps lock, or control keys were pressed.

An Editor receives this message when the user clicks or double-clicks in your Event rectangle. There are numerous strategies for what to do in response to a click message. If your object is a text editor, for example, it might call TEClick. If you will allow editing the contents of an object by drawing, as with the editor **efunc**, you'll use `wind_drag` and supply it with a drag function as is done in Patcher user interface objects.

Before sending you the click message, the Timeline object adjusts your Event rectangle so it is relative to the top of its window rather than the top of the track. However, the top and bottom of the Event rectangle will not be correct if you make a drag function to handle continuous mouse movement in your object. In order to relocate it to coordinates that reflect the screen, use the function `event_offsetRect` described below. During this function you may find it useful to call the routine for Event position conversation and drawing in the Timeline legend described below.

key

The key message is sent when the user presses a key when your Event is the Target Event.

BINDING

```
addmess (myobject_key, "key", A_CANT, 0);
```

DECLARATION

```
void myobject_key (myObject *x, short key, short
modifiers, short code);
```

`key`	ASCII code of the key pressed.
`modifiers`	State of the modifier keys.
`code`	Macintosh key code.

selected

The selected message allows you to inform the Timeline about your selection state.

BINDING

```
addmess (myobject_selected, "selected", A_CANT, 0);
```

DECLARATION

```
void myobject_selected (myObject *x, short *state);
```

`state`	Set `state` to 1 if your the data is entirely selected for editing (and thus should be, for example, copied or duplicated as a whole). An Editor that contained text for editing would set `state` to 0 if a subset of its text were selected (or if no text were selected), but it would set it to 1 if all the text were selected.

When your Editor receives this message you should set `state` according to the criteria listed above. Often editors either do not allow a subset of their data to be selected (such as **efunc**) or always edit it as a logical whole (such as the number box), and these should always set `state` to 1.

Routines For Drawing in Editors

These routines are used to set up any drawing you might do in an editor in situations such as a Qelem function or in a mouse drag tracking function called by `wind_drag`.

track_setport

Use `track_setport` to set the current GrafPort to the window containin a Timeline track.

```
GrafPort *track_setport (void *track)
```

`track`	An Event's parent track (`evnt->e_track`).

This function is required when drawing out of the context of the standard Editor messages. `track_setport` is analogous to `wind_setport` or `patcher_setport`. Given a track it ensures that drawing will occur in the Track's GrafPort. If `track_setport` returns 0 it means that the Timeline window containing your Event is not currently visible, and you should not draw anything. This situation is entirely possible if the user has created a Timeline object within the context of a patcher and closed the window. You should also check `track_setport` in your instance creation function. If it returns a non-zero value, it is safe to draw or use GrafPort-relative calls (such as TextWidth). If not, you need to defer such calls until your Editor receives the first update message for this Event. When you are finished drawing, pass the non-zero value returned from `track_setport` to the Macintosh routine SetPort.

event_offsetRect

Use `event_offsetRect` to adjust your event rectangle before drawing in it.

```
short event_offsetRect (Event *evnt)
```

`evnt`	Your Event.

Before drawing in a non-standard situation, such as in a mouse tracking function called from `wind_drag` or a queue function, you need to offset the Event's rectangle so that it is relative to the top of the window. `event_offsetRect` does this and returns the value you can use as the vertical coordinate of the Macintosh routine OffsetRect to restore the Event rectangle. Here is a typical use of `event_offsetRect`:

```
short offset;
offset = event_offsetRect((Event *)x);
/* draw here */
OffsetRect(x->m_event.e_rect,0,-offset);    /* must negate
to restore */
```

track_clipBegin

Use `track_clipBegin` to restrict drawing to the current Track rectangle, in case your event rectangle is partially hidden by a track boundary.

```
void track_clipBegin (void *track, Rect *clip);
```

`track`	An Event's parent track (`evnt->e_track`).
`clip`	Where the current track rectangle will be placed. You can use this to avoid drawing the portion of your event that is not visible by only drawing what intersects `clip`.

Before drawing in your Event rectangle during a mouse tracking function called from `wind_drag`, it is necessary to restrict your drawing to the current track rectangle, since the Event may be partially outside the visible portion of a track. This is done by framing all drawing with calls to `track_clipBegin` and `track_clipEnd`. If you draw in any other situations where you are not receiving a direct message listed above from the Timeline, such as in a queue function set by your eventStart message, you must also use `track_clipBegin` and `track_clipEnd`. Clipping has already been set to the track when your Editor receives any of the standard set of messages described above (update, click, etc.) so you need not use these functions at those time.

track_clipEnd

Use `track_clipEnd` to restore a clipping region set by `track_clipBegin`.

```
void track_clipEnd (void *track);
```

`track`	An Event’s parent track (`evnt->e_track`).

Mysterious things have been known to happen to drawing in the Timeline window if each call of `track_clipBegin` isn’t matched with a call to `track_clipEnd`.

Using Editor Drawing Routines

The proper order for all of these setup routines is shown in this example:

```
void myEditor_draw (Event *e)
{
    GrafPort *savePort;
    Rect clipRect;
    void *eventTrack;
    short offset;
    eventTrack = e->e_track;
    if (savePort = track_setport(eventTrack)) {
        offset = event_offsetRect(e);
        track_clipBegin(eventTrack,&clipRect);
        /* draw here */
        track_clipEnd(eventTrack);
        OffsetRect(e->e_rect,0,-offset);
        SetPort(savePort);
    }
}
```

Event Position Conversion Routines

These routines allow conversion between a x coordinate location in the Timeline window and an event time.

track_pixToMS

Use `track_pixToMS` to convert a pixel distance in the Timeline window to a millisecond time value.

```
long track_pixToMS (void *track, short pix);
```

`track`	An Event's parent track (`evnt->e_track`).
`pix`	Pixel value you want converted to milliseconds.

Given a track and a distance in pixels in `pix`, `track_pixToMS` returns the number of milliseconds currently associated with this number of pixels, according to the Timeline's current zoom level.

track_MSToPix

Use `track_MSToPix` to convert a time value to a pixel distance in the Timeline window.

```
short track_pixToMS (void *track, long time);
```

`track`	An Event's parent track (`evnt->e_track`).
`time`	Time value in milliseconds you want converted to pixels.

Given a duration in milliseconds, `track_MSToPix` returns the number of pixels currently associated with this duration, according to the Timeline's current zoom level.

track_posToMS

Use `track_posToMS` to convert from a location in the Timeline window to milliseconds from the start of the Timeline.

```
long track_pixToMS (void *track, short pix);
```

`track`	An Event's parent track (`evnt->e_track`).
`pix`	Pixel value you want converted to milliseconds.

Given a track and an x coordinate location, `track_posToMS` returns the event time in milliseconds currently associated with this position, according to the Timeline's current zoom level.

track_MSToPos

Use `track_MSToPos` to convert a time value to a location in the Timeline window.

```
short track_MSToPos (void *track, long time);
```

`track`	An Event's parent track (`evnt->e_track`).
`time`	Time value in milliseconds you want converted to pixels.

Given a track and an event time in milliseconds, `track_MSToPos` returns the coordinate location on in the Timeline window associated with this time, according to the Timeline's current zoom level.

Routines for Drawing in the Timeline Legend

These routines are used in a drag function set up by `wind_drag`, initiated in the method that responds to a click message. They allow the user's editing action occurring within your Event rectangle to be guided in terms of its current time position, or with any other sort of information. You can also call the routines during an Event's idle message for guidance in the style of the Patcher window's assistance. Each function takes a track argument that is used to access the Timeline window.

track_drawDragTime

Use `track_drawDragTime` to draw two numbers related to an Event.

```
void track_drawDragTime (void *track, long time1,
long time2);
```

`track`	An Event's parent track (`evnt->e_track`).
`time1`	Number you want drawn on the left.
`time2`	Number you want drawn on the right.

This function accepts two values that are converted into a string according to the current time format and displayed in the Timeline legend. Normally, it is used to display begin and end times when dragging an event rectangle. The value `time1` is generally taken to be the left side of the item being moved and `time2` is the right side. If you only want to draw one value you can use `track_drawTime`.

track_drawDragParam

Use `track_drawDragParam` to draw a string describing an Event.

```
void track_drawDragParam (void *track, char
*string);
```

`track`	An Event's parent track (`evnt->e_track`).
`string`	C string containing information to display about the Event.

This function allows you to draw any character string in the legend portion of the Timeline window to describe the current value of a parameter that might be changing in response to an editing action within the Event rectangle. The efunc object uses the routine to display the current X and Y value of a point being dragged.

track_drawTime

Use `track_drawTime` to draw a single value related to an Event.

```
void track_drawTime (void *track, long time);
```

`track`	An Event’s parent track (`evnt->e_track`).
`time`	Number to draw.

This function draws a single time value in the legend portion of the Timeline window.

track_eraseDragTime

Use `track_eraseDragTime` erase the Timeline window legend.

```
void track_eraseDragTime (void *track);
```

`track`	An Event’s parent track (`evnt->e_track`).

This function erases anything drawn with any of the above three functions.

Chapter 15: MSP Development Basics

The next few chapters describe how to write signal processing externals for Max using the API of the MSP signal processing environment. MSP externals are very similar to Max externals, but they have two additional features specific to signal processing. One is the *perform routine* that performs signal processing on one or more buffers of audio. MSP assembles calls to objects' perform routines into a *DSP call chain* connected by signal buffers. The second additional method you need to write, called when MSP builds the DSP call chain and sends your object the dsp message, tells MSP the address of your perform routine and the arguments it requires. We'll refer to it as the *dsp method.*

In addition to the perform and dsp methods, there are calls you need to make in the initialization, new instance, and free routines. There are two sets of calls, depending on whether you are writing a normal object or a user interface object. However, there are no differences between normal and user interface objects in writing the dsp and perform methods.

The MSP Library

The MSP functions described here reside in a shared library called *Max Audio Library*, that in Max 4 / MSP 2 is included in the Max/MSP application. This shared library exports a number of functions and globals used by signal processing objects. It transforms a graph of signal objects into a series of function calls, handles audio I/O and interfacing, and manages signal buffers.

Creating MSP Projects

In addition to your source file and any resource files you be using, you will need to include the following files:

- MaxAudioLib stub library
- MaxLib stub library
- InterfaceLib stub library
- MathLib (or libmoto, if you want much faster but possibly 603-incompatible math routines)
- MSL ShLibRuntime.Lib (MWCRuntime.Lib for CodeWarrior 5 and lower)
- SoundLib, if you use newer Sound Manager routines

While they are not always needed, there is no penalty for including MathLib and SoundLib if they're not needed, so it's a good idea to include them.

Refer to Chapter 2 when creating a development project. However, it may easier to modify a copy of one of the example MSP projects included in the software development kit. Then all the right files are included and the settings will be correct except for the name of your object in the settings for PPC Project. When creating your own project (or extening an existing one), you should add the *MSP includes* folder to the CodeWarrior Access Paths or MPW includes variables.

Project Resource File

In addition to the libraries listed above and the source files you write, a good citizen MSP project will contain a resource file that contains at least two items. The first is the STR# resource used by your object's assist method (refer to chapter 5 if you are not familiar with this), and a *mAxL* resource that contains a small amount of 68K code. When loaded, this code reports that the object does not work on a 68K processor. You can use the mAxL resource that does this very thing found in the file *nono.68K* included in the *MSP includes* folder of the software development kit. Open nono.68K in ResEdit and copy the mAxL resource to your project's resource file. Then select the resource in your project's file and choose Get Resource Info… from the Resource menu. The resource's ID doesn't matter, but the name must be changed from “nono” to the name of your object. 68K Max finds your object using the name of the mAxL resource.

Adding the mAxL resource allows your PowerPC object to be found when it is inside a collective or a standalone application. If you look at a standalone application created with MSP external objects, you’ll see a bunch of small mAxL resources, one for each external that is included.

Chapter 16: Writing MSP Code

This chapter covers the basic information you need to write an MSP external.

Include Files

For a typical MSP object, you should have the following at the beginning of your source file.

```
#include "ext.h"  // standard include file for Max
externals
#include "z_dsp.h" // contains MSP info
```

The include file *z_dsp.h* references a number of other include files; they will be mentioned when relevant below.

Defining Your Object Structure

An MSP object has a `t_pxobject` as its first field rather than `t_object`. `t_pxobject` is a `t_object` with some additional fields, most notably a place for an array of *proxies*, used to allow inlets to MSP objects to accept either signals or floats as input. If you're not familiar with proxies, refer to Chapter 6 and the *buddy* external object sample code. In general, MSP handles most of the details of using proxies for you. User interface objects use a similar header, called a `t_pxbox`, that combines the standard `t_box` user interface object header with the fields of a `t_pxobject`. Both structures are defined in the include file *z_proxy.h*.

Here's an example declaration of an MSP external object:

```
typedef struct _sigobj {
    t_pxobject x_obj; // header
    float x_val; // additional fields
} t_sigobj;
```

Writing the Initialization Routine

The initialization routine sets up the class information for a Max external. In the call to the `setup` function which initializes your class—generally the first thing you do in any Max external—you should pass `dsp_free` as your free routine unless you need to write your own free routine for memory you allocate in your new instance routine. Here's an example for an object that doesn't allocate any memory and doesn't take any initial arguments.

```
setup(&sigobj_class, sigobj_new, (method)dsp_free,
(short)sizeof(t_sigobj), 0L, 0);
```

After the call to `setup`, your initialization routine needs to bind your object's dsp method (discussed below), using the `A_CANT` argument type specifier as follows:

```
addmess(sigobj_dsp, "dsp", A_CANT, 0);
```

You also need to call `dsp_initclass` to finish setting up your MSP external's class.

dsp_initclass

Use `dsp_initclass` to set up your object's class to work with MSP.

```
void dsp_initclass(void);
```

This routine must be called in your object's initialization routine. It adds a set of methods to your object's class that are called by MSP to build the DSP call chain. These methods function entirely transparently to your object so you don't have to worry about them. However, you should avoid binding anything to their names: signal, drawline, userconnect, and enable. This routine is for normal (non-user-interface objects).

dsp_initboxclass

Use `dsp_initboxclass` to set up your user interface object's class to work with MSP.

```
void dsp_initboxclass(void)
```

Call this routine in a user interface object's initialization (main) routine instead of `dsp_initclass`. In addition adding the four methods bound to the names listed above, `dsp_initboxclass` also uses the name **bxdsp**.

New Instance Routine

Typical Max new instance routines specify how many inlets and outlets an object will have. An MSP signal object is no execption, but it uses proxies if you want more than a single signal inlet. You specify how many signal inlets you want with the `dsp_setup` call (or `dsp_setupbox` for user-interface signal objects). There is a requirement that signal inlets must be to the left of all non-signal inlets. Similarly, all signal outlets—declared simply with a type of "signal"—must be to the left of all non-signal outlets.

Here is an example of the initialization routine for an object that has two signal inlets and two signal outlets.

```
void *sigobj_new(void)
{
    t_sigobj *x;
    x = newobject(sigobj_class);
    dsp_setup((t_pxobject *)x,2);  // set up object and
inlets
    outlet_new((t_object *)x,"signal"); // and outlets
    outlet_new((t_object *)x,"signal");
    return x;
}
```

Note that unlike the initialization routine in a typical Max object, the example routine above doesn't store pointers to its outlets. An MSP object almost never directly references its signal outlets. The MSP signal compiler accesses the outlets via the a pointer to all your object’s outlets stored inside the t_object structure that begins all Max objects.

dsp_setup

Use `dsp_setup` to initialize an instance of your class and tell MSP how many signal inlets it has.

```
void dsp_setup(t_pxobject *x, short
num_signal_inputs);
```

Call this routine after creating your object in the new instance routine with `newobject`. Cast your object to `t_pxobject` as the first argument, then specify the number of signal inputs your object will have. `dsp_setup` initializes fields of the `t_pxobject` header and allocates any proxies needed (if `num_signal_inputs` is greater than 1). Some signal objects have no inputs; you should pass 0 for `num_signal_inputs` in this case. After calling `dsp_setup`, you can create additional non-signal inlets using `intin`, `floatin`, or `inlet_new`.

dsp_setupbox

Use `dsp_setupbox` to initialize an instance of your user interface object class and tell MSP how many signal inlets it has.

```
void dsp_setupbox(t_pxbox *x, short
num_signal_inputs);
```

This routine is a version of `dsp_setup` for user interface signal objects.

Special Bits in the t_pxobject Header

There are three bits you can set in the `t_pxobject` or `t_pxbox` header that affect how your object is treated when MSP builds the DSP call chain. The explanation of these settings will make more sense once you have read more about the **dsp** and **perform** methods, but they are explained here because you need to set them in your new instance routine. Both `t_pxobject` and `t_pxbox` contain a field called `z_misc`; by default it is 0 meaning that all of the following settings are disabled.

```
#define Z_NO_INPLACE 1
```

If you set this bit in `z_misc`, the compiler will guarantee that all the signal vectors passed to your object will be unique. It is common that one or more of the output vectors your object will use in its perform method will be the same as one or more of its input vectors. Some objects are unable to handle this restriction; typically, this occurs when an object has pairs of inputs and outputs and writes an entire output on the basis of a single input before moving on to another input-output pair.

`#define Z_PUT_LAST 2`

If you set this bit in `z_misc`, the compiler puts your object as far back as possible on the DSP call chain. This is useful in two situations. First, your object's dsp routine might require that another object's dsp routine is called first in order to work properly. Second, if your object wants another object's perform routine to run before its own perform routine. For example, to minimize delay times, a delay line reading object probably wants the delay line writing object to run first. However, setting this flag does not guarantee any particular ordering result.

`#define Z_PUT_FIRST 4`

If you set this bit in `z_misc`, the compiler puts your object as close to the beginning of the DSP call chain as possible. This setting is not currently used by any standard MSP object.

The dsp Method

The dsp message is sent to your object when MSP is building the DSP call chain. If you want to add something to the chain, your dsp method should call `dsp_add`, which adds your perform method to the DSP call chain. Your method should be declared as follows:

dsp

Called by MSP to include your object in the DSP call chain.

BINDING

```
addmess (mysigobject_dsp, "dsp", A_CANT, 0);
```

DECLARATION

```
void mysigobject_dsp (t_sigobj *x, t_signal **sp,
short *count);
```

Your dsp method is passed an array of `t_signal` structures that define your perform method's signal inputs and outputs. A `t_signal` contains a buffer of floats and a size `s_n`, which specifies the number of samples computed during any particular call to your perform routine. (This size is sometimes referred to as a *vector size.)* Currently, the vector size will be the same for all the signals you receive, although future versions of MSP may allow special objects that accept vectors of different or variable sizes. A signal also has a *sampling rate*; it is very important that if your object makes sampling-rate-dependent calculations, it use the sampling rate in one of the signals rather than use the global sampling rate obtained by a call to sys_getsr. The `t_signal` structure is defined in *z_dsp.h*.

In addition to the array of `t_signals`, your dsp method is passed an array that specifies the number of connections to each input and output. Some MSP objects use this information to put different perform methods on the DSP call chain. For instance, the *~ object does some optimizing by using a simpler routine that multiplies a signal signal by a constant if there is no signal connected to one of its inputs. In this case it uses the object's internal value, set either as an argument or via a float sent to the right inlet.

You may wish to use the dsp method initialize other internal variables used by your perform routine. For example, many objects require dividing by the sampling rate. Rather than dividing during the perform routine, which is expensive, you can calculate the reciprocal of the sampling rate in the dsp routine, store it, and then multiply by the reciprocal in the perform routine. Again, remember that you must obtain the sampling rate from one of your signal arguments, rather than assuming the sampling rate of your object's perform routine will be the same as the global sampling rate.

dsp_add

Use `dsp_add` to add your object's perform routine to the DSP call chain.

```
void dsp_add(t_perfroutine p, long argc, ...);
```

This function adds your object's perform method to the DSP call chain and specifies the arguments it will be passed. `argc`, the number of arguments to your perform method, should be followed by `argc` additional arguments, all of which must be the size of a pointer or a long.

dsp_addv

Use dsp_addv to add your object's perform routine to the DSP call chain and specify its arguments in an array rather than as arguments to a function.

```
void dsp_addv(t_perfroutine p, long argc, void
**vector)
```

This function is a variant of dsp_add that allows you to construct an array of the arguments you wish to pass to your perform routine.

Here's an example of dsp method that doesn't pay attention to the connection count information might do. It has two inputs and two outputs. The inputs appear first in the array of signals, followed by the outputs, so sp[0] is the left input, sp[1] is the right input, sp[2] is the left output, and sp[3] is the right output. It also storesthe reciprocal of the sampling rate to use in its perform method calculation.

```
void sigobj_dsp(t_sigobj *x, t_signal **sp, short *count)
{
    x->s_1oversr = 1. / (double)sp[0]->s_sr;
    dsp_add(sigobj_perform, 5, sp[0]->s_vec, sp[1]->s_vec,
    sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
}
```

The above call to dsp_add specifies the name of the perform routine, followed by the number of arguments that will be passed to it, followed by each argument. The s_vec field of a signal is its array of floats. In this case, the two input arrays are passed, followed by the two output arrays, followed by the vector size. You can pick any t_signal to use for the vector size. By convention, most MSP objects use the first input signal.

Next, here's a more complex dsp method that uses a different perform routine if its right input and right output are disconnected. One object that does something similar is fft~, where a routine that calculates only the real part of an FFT is used if the imaginary input and output are disconnected. In this example, `sigobj_perform2` takes only three arguments, the signal vectors for the left input and left output, plus the vector size.

```
void sigobj_dsp(t_sigobj *x, t_signal **sp, short *count)
{
   if (count[1] || count[3]) // right input or right
output connected
       dsp_add(sigobj_perform, 5, sp[0]->s_vec, sp[1]-
>s_vec,       sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
   else
       dsp_add(sigobj_perform2, 3, sp[0]->s_vec, sp[2]-
>s_vec,       sp[0]->s_n);
}
```

Even if, according to the information in the count array passed to your dsp method, a `t_signal` is not connected to your object, the `t_signal` still contains a valid vector as well as valid sample rate and vector size information.

If for some reason you want to put several functions on the DSP call chain, you can do so: just make as many calls to `dsp_add` as you want. However, keep in mind there are subtle issues when doing this. For instance, if the input and output signals buffers point to the same memory, if the first function writes data to an output signal buffer, the input signal buffer will have been overwritten for all subsequent perform routines that use the same signal buffers.

The Perform Routine

Your perform routine is called repeatedly to calculate signal values. MSP calls each perform method in its DSP call chain in order with the arguments that were specified by the object's call to `dsp_add`. However, the arguments are not passed on the stack; instead, a pointer to an array containing the arguments is passed to the object. The perform routine must return a pointer into the array just after the last argument specified by its `dsp_add` call. If this is not done, MSP will crash. Your method should be declared as follows:

```
t_int *sigobj_perform(t_int *w);
```

MSP generally calls perform routines at interrupt time. As with any interrupt routine, your perform routine should be written as efficiently as possible. It cannot call routines that would move memory, nor should it call `post` (for debugging), since at a 44.1 kHz sampling rate and vector size of 256 samples, each perform routine is called about every 5.8 milliseconds. You can however, set a qelem or, if you're careful, use `defer_low` (*not* `defer`, since the MSP interrupt is not the Max scheduler interrupt, and thus `defer` doesn't know that it is being executed at interrupt level) to delay a function until the main level. You need to be careful because, if you `defer` a call every 5.8 milliseconds, you will cause a huge backlog of main event level functions that need to be run, as well as allocate a large amount of memory at interrupt level.

Here is a sample perform method that takes two signals as input, adds them together to produce one output and subtracts them to produce the other. The method is written so that it would be compatible with the `sigobj_dsp` example shown above. The type `t_int` is the same size as a pointer (int or long on the PowerPC); it is used for some degree of source code compatibility with `Pd` perform routines.

Note that the first argument that you specified in your call to `dsp_add` is at offset 1 in the array passed to your perform routine. Offset 0 contains the address of your perform routine.

```
t_int *sigobj_perform(t_int *w)
{
    float *in1,*in2,*out1,*out2,val,val2;
    long n;
    in1 = (float *)(w[1]);   // input 1
    in2 = (float *)(w[2]); // input 2, second arg
    out1 = (float *)(w[3]); // arg 3, first output
    out2 = (float *)(w[4]); // arg 4, second output
    n = w[5]; // vector size
    // calculation loop
    while (n--) {
        val = *in1++;
        val2 = *in2++;
        *out1++ = val + val2
        *out2++ = val - val2;
    }
    return w + 6; // always return a pointer to one more
than the
                  // highest argument index
}
```

In the calculation loop, the code is written so that even if the output and input signal vectors are the same, the result is still correct. However, if for some reason you can't do this, you can specify that the input and output signal vectors should be unique with the `Z_NOINPLACE` flag. How to do this is explained above in the section entitled Special Bits in the `t_pxobject` header. (Only two standard MSP objects—**fft~/ifft~** and **tapout~**—require this feature.)

The Free Routine

If your normal object doesn't allocate any memory or need anything to be turned off when an instance is freed, you can pass `dsp_free` as the free method to `setup` in your initialization (main) routine. (User interface objects, even if they don't allocate memory themselves, require a free routine because they need to call `box_free`.)

If you do write your own free routine, your normal object should call `dsp_free` in it, and your user interface object should call `dsp_freebox`.

dsp_free

Use `dsp_free` in your object's free routine.

```
void dsp_free(t_pxobject *x);
```

This function disposes of any memory used by proxies allocated by `dsp_setup`. It also notifies the signal compiler that the DSP call chain needs to be rebuilt if signal processing is active. You should be sure to call this before de-allocating any memory that might be in use by your object's perform routine, in the event that signal processing is on when your object is freed.

dsp_freebox

For user interface objects, use dsp_freebox instead of dsp_free.

```
void dsp_freebox(t_pxbox *x);
```

This function disposes of any memory used by proxies allocated by dsp_setupbox. It also notifies the signal compiler that the DSP call chain needs to be rebuilt if signal processing is active. You should be sure to call this before de-allocating any memory that might be in use by your object's perform routine, in the event that signal processing is on when your object is freed.

Chapter 17: Handling MSP Parameters

Real-time signal processing isn't just about calculating signals. You also want your DSP routines to respond to changes in input parameters. This task is often referred to as parameter updating. In Max, DSP parameters are typically control messages. You can either use a **sig~** or **line~** object to convert control messages to signals, or you can update the internal state of a signal object directly. The latter approach has the disadvantage of possible discontinuities in the output, but for many applications or when the user is experimenting, it is easier, not to mention more efficient.

Many MSP objects need to pass a pointer to themselves to the perform method to access internal state information. For example, a filter object that accepts floats to specify coefficients would need to pass itself to the perform routine so that these coefficients can be accessed.

A Filter Example

As an example, here are the dsp and perform methods of a simple lowpass filter object called **lop~** that uses a coefficient stored in the object. Let's first assume that the coefficient, which is specified via the right inlet of the object, can only be passed as a float, not a signal. This means you'll have to declare an additional inlet and a method to accept the parameter. Here is the object declaration:

```
typedef struct _lop {
    t_pxobject x_obj;    // header
    float x_coeff;           // coefficient
    float x_m1;              // filter memory
} t_lop;
```

Here is the initialization routine:

```
void main(void)
{
    setup(&lop_class, lop_new, (method)dsp_free,
(short)sizeof(t_lop), 0L, A_DEFFLOAT, 0);
    addftx(lop_ft1,1);   // bind right inlet method
    addmess(lop_dsp,"dsp", A_CANT, 0);
    dsp_initclass();
}
```

Here is the right inlet method..

```
void lop_ft1(t_lop *x, double f)
{
    x->x_coeff = f;
}
```

Here is the new instance routine. There is only a single signal input, in the left inlet, so 1 is passed as the signal input count to `dsp_setup`.

```
void *lop_new(double initial_coeff)
{
    t_lop *x = newobject(lop_class);
    dsp_setup((t_pxobject *)x, 1);
    floatin((t_object *)x,1);
    outlet_new((t_object *)x,"signal");
    x->x_coeff = initial_coeff; // initialize coefficient
    x->x_m1 = 0.;    // initialize previous state
    return x;
}
```

Here is the dsp method. It instructs MSP to pass the object's pointer, the input vector, the output vector, and the vector size to the perform routine. No signal connection counting is required; indeed, you could declare the method without the `count` parameter if you wanted to.

```
void lop_dsp(t_lop *x, t_signal **sp, short *count)
{
    dsp_add(lop_perform1, 4, x, sp[0]->s_vec, sp[1]->s_vec,
               sp[0]->s_n);
}
```

Finally, here is the perform routine. We have called it `lop_perform1` because we'll be writing alternative perform methods as we continue with the example. Note how we get the filter coefficient out of the object's structure and place it in a local variable. This is far more efficient than reading it out of the object during the calculation loop since the vector could be up to 2048 samples. Since the perform routine is executing at interrupt level, we are guaranteed that the coefficient won't change in the middle of the routine. The same is true for the filter's memory that is also stored inside the object.

```
t_int *lop_perform1(t_int *w)
{
    t_lop *x = (t_lop *)(w[1]);     // object is first arg
    float *in = (float *)(w[2]);    // input is next
    float *out = (float *)(w[3]); // followed by the output
    long n = w[4];              // and the vector size
    loat xm1 = x->x_m1;  // local to keep track of previous 
state
    float coeff = x->x_coeff,val; // and coefficient
    // filter calculation
    while (n--) {
        val = *in++;
        *out++ = coeff * (val + xm1);
        xm1 = val;
    }
    x->x_m1 = xm1; // re-save old state for the next time
    return w + 5;
}
```

Now we will rewrite the filter to accept either a float or a signal for the coefficient value. There are two strategies for doing this depending on how often you want to read the coefficient value from a signal vector. First, let’s write it with a single perform routine that makes a decision about whether to get the coefficient from a signal or from the float value stored inside the object. In this implementation, the coefficient is only read from the first value of the signal vector, and the rest of the vector is ignored.

We will add a field to our object that tells the perform routine whether a the dsp routine found that a signal was connected to the right inlet or not.

```
typedef struct _lop {
    t_pxobject x_obj;
    float x_coeff;
    float x_m1;
    short x_connected;
} t_lop;
```

Since MSP will be using a proxy to get the signal and the float in the right inlet, we need to change our initialization routine sightly. We replace

```
addftx(lop_ft1,1);
```

with

```
addfloat(lop_float);
```

Other than being renamed, the float method remains the same as the one above.

Here is the revised new instance routine that specifies two signal inlets. We have removed the creation of the additional inlet and changed the number of signal inlets specified in the call to `dsp_setup` to 2. `dsp_setup`, using a proxy, creates the right inlet for us.

```
void *lop_new(double initial_coeff)
{
    t_lop *x = newobject(lop_class);
    dsp_setup((t_pxobject *)x,2); // changed from previous
example
    outlet_new((t_object *)x,"signal");
    x->x_coeff = initial_coeff; // initialize coefficient
    x->x_m1 = 0.; // initialize previous state
    return x;
}
```

Here is the revised dsp method. Since there are now two signal inlets, the output vector is at `sp[2]` rather than `sp[1]`.

```
void lop_dsp(t_lop *x, t_signal **sp, short *count)
{
    x->x_connected = count[1];  // save whether right inlet
has a signal
                                // going into it
    dsp_add(lop_perform2, 5, x, sp[0]->s_vec, sp[1]->s_vec,
               sp[2]->s_vec,
        sp[0]->s_n);
}
```

Here is the revised perform routine. Depending on the value of the `x_connected` field of the object, it uses either the first value from the signal vector passed on the stack or the stored float value.

```
t_int *lop_perform2(t_int *w)
{
    t_lop *x = (t_lop *)(w[1]);
    float *in = (float *)(w[2]);
    // use either signal or stored coefficient
    float coeff = x->x_connected? *(float *)(w[3]) : x-
>x_coeff;
    float *out = (float *)(w[4]);
    long n = w[5];
    float xm1 = x->x_xm1,val;
    while (n--) {
        val = *in++;
        *out++ = coeff * (val + xm1);
        xm1 = val;
    }
    x->x_m1 = xm1; // re-save old state for the next time
    return w + 6;     // 6 because there were now five
arguments
}
```

The second strategy uses two different perform routines. The dsp method decides which one to use based on the count of signals connected to the right input. Other than the elimination of the `x_connected` field of the `t_lop` structure, only the dsp and perform methods change from the previous implementation of **lop~.** Here is the revised dsp method, which makes reference to the original `lop_perform1` method defined above. Even though there is an additional input signal to the object now, we can still use `lop_perform1` by passing only the left input signal vector and the output signal vectors. `lop_perform1` has no idea that there was another input signal vector.

```
void lop_dsp(t_lop *x, t_signal **sp, short *count)
{
    if (count[1])
        dsp_add(lop_perform3,5,x, sp[0]->s_vec, sp[1]-
>s_vec,
            sp[2]->s_vec, sp[0]->s_n);
    else
        dsp_add(lop_perform1, 4, x, sp[0]->s_vec, sp[2]-
>s_vec,
            sp[0]->s_n);
            // skip unused sp[1] signal
}
```

Finally, here is `lop_perform3`, which uses all of the values from the input coefficient signal in calculating the low-pass filter output. It is ignorant that there is a stored internal coefficient.

```
t_int *lop_perform3(t_int *w)
{
    t_lop *x = (t_lop *)(w[1]);
    float *in = (float *)(w[2]);
    float *coeff = (float *)(w[3]);
    float *out = (float *)(w[4]);
    long n = w[5];
    float xm1 = x->x_xm1,val;
    while (n--) {
        val = *in++;
        // use each value in the coefficient signal vector
        *out++ = *coeff++ * (val + xm1);
        xm1 = val;
    }
    x->x_m1 = xm1; // re-save old state for the next time
    return w + 6;
}
```

Chapter 18: Access to MSP Global Information

The following routines provide access to the global state of the DSP environment. The results may not be valid in your object's main (initialization) routine, and they may change between your object's new instance routine and its dsp method.

sys_getblksize

Use `sys_getblksize` to find out the current DSP vector size.

```
long sys_getblksize(void);
```

sys_getsr

Use `sys_getsr` to find out the current sampling rate.

```
float sys_getsr(void);
```

However, do not use sys_getsr() within an object’s dsp method or its perform routine. Instead, use the sampling rate of one of the signal vectors passed to the dsp method, and store this value in your object for access by the perform routine if it needs it. This is because your object may be used inside an object such as **poly~** or **pfft~** that runs at a higher or lower sampling rate than the global rate. The following code example shows how to store the sampling rate from a t_signal.

```
void *myobject_dsp(t_myobject *x, t_signal **sp, short *count)
{
   x->x_sr = sp[0]->s_sr;      // store sampling rate
   // more code here
}
```

sys_getch

Use `sys_getch` to find out the current maximum number of channels.

```
long sys_getch(void);
```

sys_getdspstate

Use `sys_getdspstate` to find out whether the DSP is active or not.

```
long sys_getdspstate(void);
```

This function returns 1 if the DSP is active, 0 if it is not.

The following function returns information about an object's context in a DSP network.

dsp_isconnected

Use `dsp_isconnected` to determine whether two signal objects are connected.

```
short dsp_isconnected(t_object *src, t_object *dst,
short *index);
```

This function is useful only if you call it in your **dsp** method. It can be used to determine whether there is a signal connection from an outlet of src to an inlet of dst. The function returns a non-zero result if there is a connection, and zero if there isn't. The result is a count of the number of objects in between `src` and `dst` plus one. For example, if `dst` were directly below `src`, `dsp_isconnected` would return 1. `dsp_isconnected` returns in `index` the inlet number of `dst` (starting at 0) where the connection occurs. If there is more than one connection, information about the leftmost connection is returned.

Appendix A: Reserved Messages

For All Objects

Max makes certain assumptions if external objects have methods bound to some symbols. If you bind methods to these symbols, your methods need to do what Max expects. In addition, there are certain “internal” messages not described here that are used by Max or by certain objects. You need to avoid binding methods to these symbols. Below is a review of both the documented messages for which your method needs to play by the rules, as well as the secret internal messages you need to avoid altogether.

Many words are ill-advised only in certain contexts, such as for user-interface objects, non-user-interface objects, or objects that own windows. Thus this list is categorized by the context in which the message may be sent to your object by the system.

assist	Max asks an object to descibe itself. Avoid for other purposes.
checkin	Used by inlets to do type-checking. Avoid using this word.
disable	Sent to every object in a patcher when the user changes the enable icon in the title bar of a patcher window to the "X." For MIDI objects, this causes them to stop input or output. If your object talks directly to a piece of hardware, you may want to implement this method, otherwise don’t use the word.
enable	Sent to every object in a patcher when the user changes the enable icon in the title bar of a patcher window to the MIDI symbol. All objects are assumed to be enabled when created. For MIDI objects, this causes them to enable input or output. If your object talks

	directly to a piece of hardware, you may want to implement this method, otherwise don't.
loadbang	Sent to all objects by the patcher after a file has been loaded. Avoid for anything other than special initialization in this context.
preset	Used to support the preset object. Avoid for any other purpose.
repo	Internal message used by MIDI objects but sent to all objects. Sent every time the OMS setup changes. Don't use it.
info	Called when your object is selected in an unlocked patcher and the user chooses Get Info... from the Max menu. Don't use it for anything else than bragging or putting up a dialog to edit your object's settings.

For Non User-Interface Objects Only

For Any Object that Can Be Loaded from a File (eg. patcher, table, timeline)

save	If a method is bound to "save", Max assumes it is the object's way of saving itself in a Patcher file. (An example of how this is done is found in the sample code for the simp object or the coll object). Don't use in any other context.
dblclick	Sent to an object when the user double-clicks on an object box. Many objects open editing windows at this time. If you implement this for something else, the user will experience unpredictable behavior when double-clicking on your object's box.
imbed	Internal message used by patcher objects. Avoid it.
front	Used to bring patcher and table windows to the front. Don't use unless you're implementing an object whose files can be opened directly.

For User-Interface Objects Only

filename	Used to set a filename for a window after it has been loaded. OK only for this purpose.

setport	If a method is bound to setport, Max assumes it is a user interface object which requires special treatment when doing a patcher_setport. Don’t use it.
invis	For specialized user interface objects like bpatcher. Don’t use it. Note that the invis message is used by window-owning objects for something entirely different.
vis	For specialized user-interface objects like bpatcher. Don’t use it. Note that the vis message is used by window-owning objects for something entirely different.
eval	Used by user-interface text objects, like the message box and the object box, to re-evaluate their binbufs to create a new object. Takes no arguments. Best to avoid.
offset	Also used by bpatcher. Objects that don't set b_checkinvis field of the box can use this.
psave	Patcher save method. Avoid for any other purpose.
key	Patcher key method. Avoid for any other purpose.
click	Patcher mouse click method. Avoid for any other purpose.
update	Patcher update method. Avoid for any other purpose.
bfont	Patcher method to change box’s font. Avoid for any other purpose.
clipregion	Sent to an object to determine its shape. Avoid for any other purpose.

Objects that Own Text Editors Only

pname	For user interface objects whose b_firstin field is a patcher. Takes no arguments, and requests the name of the patcher, which should be returned as a symbol as the function result. Avoid for other purposes if you store a patcher this way.

edsave	Used by the ed object to allow the owner of a text editor to save (or not save) text in a special way. Avoid for other purposes.
edclose	Used to pass text to an object when window closes. Avoid for other purposes.

Objects that Own Patcher Windows Only

okclose	Used by the Ed object to allow the owner of a text editor to put up an alert asking whether the user wishes to save changes. Avoid for other purposes.
pclose	Sent to an object that has associated itself with a patcher in the p_assoc field when the patcher window is closing. Avoid for other purposes if you modify this field.

Objects that Own Windows Only

okclose	Sent to an object associated with a patcher window (that used patcher_okclose) when the window is about to be closed. Make sure your okclose method conforms if you call patcher_okclose.
pclose	Sent to window owner to override save changes dialog. Avoid for other purposes.
saveto	Sent to window owner to save a file. Avoid for other purposes.
otclick	Sent to window owner on option-title-click. Avoid for other purposes.
oksize	Sent to window owner to confirm size change. Avoid for other purposes.
mouseup	Sent to window owner on mouse up event. Avoid for other purposes.
print	Sent to window owner when user wants to print. Avoid for other purposes.
help	Sent to window owner when Help is chosen from Max menu. Avoid for other purposes.

font	Sent to window owner when Font menu is used. Avoid for other purposes.
wsize	Sent to window owner when window size changes. Avoid for other purposes.
key	Sent to window owner on key down event. Avoid for other purposes.
activate	Sent to window owner on activate event. Avoid for other purposes.
update	Sent to window owner on update event. Avoid for other purposes.
click	Sent to window owner on mouse down event. Avoid for other purposes.
idle	Sent to window owner during main event loop idle time. Avoid for other purposes.
find	Sent to window owner when Find item is used in Edit menu. Avoid for other purposes.
invis	Sent to window owner when window will become invisible. Avoid for other purposes.
close	Sent to window owner to close window. Avoid for other purposes.
scroll	Sent to window owner to scroll window contents. Avoid for other purposes.
dialog	Sent to window owner when Get Info… is chosen from Max menu. Avoid for other purposes.
pastepic	Sent to window owner when Paste Picture is chosen from Edit menu. Avoid for other purposes.
wcolor	Sent to window owner when Color… is chosen from Max menu. Avoid for other purposes.
chkmenu	Sent to window owner to enable and disable menu items. Avoid for other purposes.
undoitem	Sent to window owner to set text of Undo item. Avoid for other purposes.

edit	Sent to window owner when Edit is chosen from Max menu. OK for other purposes if you don't enable field in chkmenu message.
lineup	Sent to window owner when Align is chosen from Max menu. OK for other purposes if you don't enable field in chkmenu message.
fixwidth	Sent to window owner when Fix Width is chosen from Max menu. OK for other purposes if you don't enable field in chkmenu message.
hide	Sent to window owner when Hide On Lock is chosen from Max menu. OK for other purposes if you don't enable field in chkmenu message.
show	Sent to window owner when Show On Lock is chosen from Max menu. OK for other purposes if you don't enable field in chkmenu message.
undo	Sent to window owner when Undo is chosen from Edit menu. OK for other purposes if you don't implement an undoitem method.
cut	Sent to window owner when Cut is chosen from Edit menu. OK for other purposes if you don't enable field in chkmenu message.
copy	Sent to window owner when Copy is chosen from Edit menu. OK for other purposes if you don't enable field in chkmenu message.
paste	Sent to window owner when Paste is chosen from Edit menu. OK for other purposes if you don't enable field in chkmenu message.
clr	Sent to window owner when Clear is chosen from Edit menu. OK for other purposes if you don't enable field in chkmenu message.
dup	Sent to window owner when Duplicate is chosen from Edit menu. OK for other purposes if you don't enable field in chkmenu message.

selall	Sent to window owner when Select All is chosen from Edit menu. OK for other purposes if you don’t enable field in chkmenu message.

Appendix B: Useful Symbols

"Binding" to a symbol means that there is something of value in the Symbol's `s_thing` field (the `t_symbol`'s `s_name` field is a pointer to a C string). These bindings exist for specific limited times, as specified below.

#I When your object's initialization function is called, the name of your object (as a symbol) is bound to the symbol #I. This is important for objects such as the led object that can be modified by changing its resources. Because led looks at its current name bound to #I, different versions of the led object, with different names, can co-exist, since led uses this name in its psave method.

#B A non-user-interface object can get its Box when its creation function is called by looking at what is bound to #B. The Box is not necessarily connected to other objects, visible, or anything else in particular at the time the object is created. Another use of #B is the color message that is added to set the color of the new object box if it is not the default color. Change the color of a new object box, save a file containing it, and open it as text. You'll see something like…

```
#P newex 20 54 18 18 19932 funbuff;
#B color 10;
```

#P Any object can access its patcher in its creation function by looking at what is bound to this symbol.

#A An object that is loaded as an Action into a Timeline gains access to its parent Track by looking at what is bound to #A.

Appendix C: Using Native Windows APIs in Max

Nothing prevents the use of arbitrary Windows APIs from within a Max external, but there are certain times where you will need to convert some data from Max's internal representation to something that Windows can understand. One example of such a thing is a file's location found in the Max search path or selected via the Max open file dialog. The means of converting this to Windows format is demonstrated in the "filebyte" example project. This project shows both the Max cross platform and Windows specific means of accessing a file on disk. The cross platform approach uses the Sysfile API to open the disk. The Windows specific approach uses the Max function `path_nameconform` to convert Max's name and path pair to a Windows style path specifier for opening via the Windows CreateFile call. Another example would be to obtain the HWND from a Max window or patcher object so that it could be drawn in or possibly manipulated in some way. The "xgui" example project shows how to do this using the `patcher_gethwnd` function, and performs drawing in the HWND. Also shown in this example is the QuickDraw equivalent for comparison (both Macintosh and QTML based code). If you find some other native data structures that you need access to, or if Max seems to be preventing you from making certain Windows specific calls, please let us know at: maxsdk@cycling74.com.

Appendix D: Obsolete Functions

This section is provided as a reference of obsolete functions and what should replace them when developing objects for Max 4.3 and later.

finder_addclass

The method `finder_addclass` was originally used to add your object's name to the New Object List window that appears when the user inserts an object box into a patcher window. Max 4.3 introduces a better way of dealing with the New Object List, by including a text file with oblist messages in Max's init folder. Details on how to do this can be found in the Max 4.3 Getting Started manual.

lockout_set

The method `lockout_set` was formerly used from the main event level to prevent interrupt-level processing during critical regions of non-interrupt code. Critical regions are now handled by critical_enter, critical_exit. Please refer to the section on Interrupt Level Considerations and Critical Regions in Chapter 7 for more information. When using Max 4.3 and later, lockout_set does nothing.

path_createfile

The method `path_createfile` was used to create a file given a type code, a filename and a Path ID. As of Max 4.3 you should be using the newer `path_createsysfile` which is compatible with Max's built-in platform-independent Sysfile functions to deal with file handling. Please refer to the section on the Sysfile API in Chapter 8 for more information.

path_openfile

The method `path_openfile` was used to open a file given a filename and Path ID. As of Max 4.3 you should be using `path_opensysfile`, which is compatible with Max’s built-in platform-independent Sysfile functions, to deal with file handling. Please refer to the section on the Sysfile API in Chapter 8 for more information.

path_openresfile

The method `path_openresfile` was used on Mac OS9 to open the resource fork of a file given a filename and Path ID. Since resource files are platform-dependant, they will not work on Windows XP and consequently are no longer supported for cross-platform external development.

path_createresfile

The method `path_createresfile` was used on Mac OS9 to create an empty resource fork given a type code, a filename and a Path ID. Since resource files are platform-dependant, they will not work on Windows XP and consequently are no longer supported for cross-platform external development.

Index