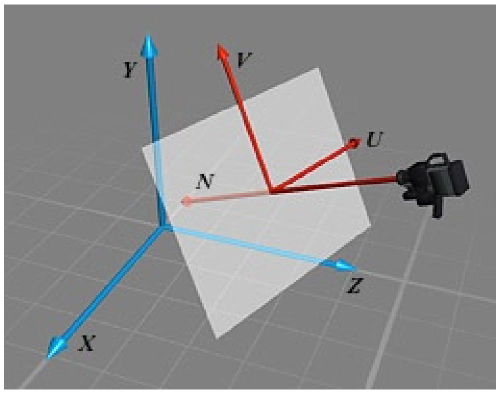
World to Screen Transformation

Author Ernie Wright
Date 29 Mar 2001

This page tells you how to convert from LightWave's world (*x*, *y*, *z*) coordinates to pixel coordinates in the rendered image. It relates parameters available from the LightWave plug-in API to components of the *synthetic camera model*, a method for defining the position of a virtual camera in 3D space and determining what it can see.



Pixels lie on the view plane in the camera's UVN coordinate system. The camera itself is set back from the view plane by an amount called the *eye distance* that controls the perspective foreshortening.

The unit vectors **u**, **v** and **n** correspond to the axes of the UVN system. Position vector **r** is the origin of UVN, sometimes called the view reference point, or VRP. Given these vectors and the eye distance *e*, we can express the world to image space transformation as a single 4 x 4 matrix *M*,

$$M = \begin{matrix} u_x & v_x & n_x & -n_x/e_n \\ u_y & v_y & n_y & -n_y/e_n \\ u_z & v_z & n_z & -n_z/e_n \\ r'_x & r'_y & r'_z & 1 - r'_z/e_n \end{matrix}$$

The upper left 3 x 3 submatrix rotates the world into the UVN orientation and contains the components of **u**, **v** and **n**. The bottom row translates the origin along **r**', derived from **r** by

$$\mathbf{r}' = (-\,\mathbf{r}\cdot\mathbf{u},\ -\,\mathbf{r}\cdot\mathbf{v},\ -\,\mathbf{r}\cdot\mathbf{n})$$

The rightmost column contains the perspective foreshortening terms.

To transform the point **p**, multiply its homogeneous position vector (p_x, p_y, p_z, 1) by matrix M. The result is the transformed point in viewing coordinates. For pixel coordinates, you need to divide by the width and height of a pixel.

Doing It from a Plug-in

In your activation function, get the globals you'll need.

```
LWItemInfo *lwi;
LWCameraInfo *lwc;
LWSceneInfo *lws;

lwi = global( LWITEMINFO_GLOBAL,   GFUSE_TRANSIENT );
lwc = global( LWCAMERAINFO_GLOBAL, GFUSE_TRANSIENT );
lws = global( LWSCENEINFO_GLOBAL,  GFUSE_TRANSIENT );
if ( !lwi || !lwc || !lws ) return AFUNC_BADGLOBAL;
```

When you're ready to do the transformation, get the camera's `RIGHT`, `UP`, `FORWARD` and `W_POSITION` vectors, and the camera zoom factor. These correspond to **u**, **v**, **n**, **r** (sort of) and e.

```
LWItemID id;
LWTime lwtime;
double u[ 3 ], v[ 3 ], n[ 3 ], r[ 3 ], e;

id = lws->renderCamera( lwtime );
lwi->param( id, LWIP_RIGHT,      lwtime, u );
lwi->param( id, LWIP_UP,         lwtime, v );
lwi->param( id, LWIP_FORWARD,    lwtime, n );
lwi->param( id, LWIP_W_POSITION, lwtime, r );
e = -lwc->zoomFactor( id, lwtime );
```

Fill in the transformation matrix.

```
typedef double MAT4[ 4 ][ 4 ];
MAT4 t;

t[ 0 ][ 0 ] = u[ 0 ];
t[ 0 ][ 1 ] = v[ 0 ];
t[ 0 ][ 2 ] = n[ 0 ];
t[ 1 ][ 0 ] = u[ 1 ];
t[ 1 ][ 1 ] = v[ 1 ];
t[ 1 ][ 2 ] = n[ 1 ];
t[ 2 ][ 0 ] = u[ 2 ];
t[ 2 ][ 1 ] = v[ 2 ];
t[ 2 ][ 2 ] = n[ 2 ];
t[ 3 ][ 0 ] = -dot( r, u );
t[ 3 ][ 1 ] = -dot( r, v );
t[ 3 ][ 2 ] = -dot( r, n ) + e;
t[ 0 ][ 3 ] = -n[ 0 ] / e;
t[ 1 ][ 3 ] = -n[ 1 ] / e;
t[ 2 ][ 3 ] = -n[ 2 ] / e;
t[ 3 ][ 3 ] = 1.0 - t[ 3 ][ 2 ] / e;
```

Multiply the point by the matrix.

```
double pt[ 3 ], tpt[ 3 ], w;

w = transform( pt, t, tpt );
```

At this point, you'll probably want to test whether the transformed point is actually visible in the image. It might be behind the camera, or in front of it but outside the boundaries of the viewport. The *z* coordinate returned by the transformation, however, is the *pseudodepth*, which has desireable mathematical properties but isn't very useful for visibility testing in LightWave.

We calculate the true *z* distance (the perpendicular distance from the camera plane) instead, using the homogeneous coordinate *w* returned by our `transform` function. `zdist` is the value that would be in the *z*-buffer for the point.

```
double zdist, frameAspect;

zdist = w * tpt.z - e;
if ( zdist <= 0 ) {
   // the point is behind the camera ...

frameAspect = lws->pixelAspect * lws->frameWidth / lws->frameHeight;
if (( fabs( tpt.y ) > 1 ) || ( fabs( tpt.x ) > frameAspect )) {
   // the point is outside the image rectangle ...
```

Finally, convert from meters (on the projection plane) to pixels.

```
double s, x, y;

s = lws->frameHeight * 0.5;
y = s - tpt.y * s;
x = lws->frameWidth * 0.5 + tpt.x * s / lws->pixelAspect;
```

The `transform` function looks like

```
double transform( double pt[ 3 ], MAT4 t, double tpt[ 3 ] )
{
   double w;

   tpt[ 0 ] = pt[ 0 ] * t[ 0 ][ 0 ]
            + pt[ 1 ] * t[ 1 ][ 0 ]
            + pt[ 2 ] * t[ 2 ][ 0 ] + t[ 3 ][ 0 ];
   tpt[ 1 ] = pt[ 0 ] * t[ 0 ][ 1 ]
            + pt[ 1 ] * t[ 1 ][ 1 ]
            + pt[ 2 ] * t[ 2 ][ 1 ] + t[ 3 ][ 1 ];
   tpt[ 2 ] = pt[ 0 ] * t[ 0 ][ 2 ]
            + pt[ 1 ] * t[ 1 ][ 2 ]
            + pt[ 2 ] * t[ 2 ][ 2 ] + t[ 3 ][ 2 ];
   w        = pt[ 0 ] * t[ 0 ][ 3 ]
```

```
            + pt[ 1 ] * t[ 1 ][ 3 ]
            + pt[ 2 ] * t[ 2 ][ 3 ] + t[ 3 ][ 3 ];

   if ( w != 0 ) {
      tpt[ 0 ] /= w;
      tpt[ 1 ] /= w;
      tpt[ 2 ] /= w;
   }

   return w;
}
```

and `dot` is just

```
double dot( double a[ 3 ], double b[ 3 ] )
{
   return ( a[ 0 ] * b[ 0 ] + a[ 1 ] * b[ 1 ] + a[ 2 ] * b[ 2 ] );
}
```

Let's Make a Box

Author Ernie Wright
Date 29 May 2001

This is an introductory level plug-in tutorial. We'll be discussing a Modeler plug-in that makes a box. The emphasis in this first installment is on the basic mechanics of writing and compiling a plug-in. We don't want to get bogged down in the specifics of creating geometry, so we'll make a simple shape, and we'll do it in the simplest available way, using the `MAKEBOX` command. But don't worry, we'll get to some of the cool stuff in later installments.

Our First Box Plug-in

Here's the entire source file for our first plug-in. This is included in the SDK samples as the file `sample/boxes/box1/box.c`.

```
#include <lwserver.h>
#include <lwcmdseq.h>
#include <stdio.h>

XCALL_( int )
Activate( long version, GlobalFunc *global, LWModCommand *local,
   void *serverData)
{
   char cmd[ 128 ];

   if ( version != LWMODCOMMAND_VERSION )
      return AFUNC_BADVERSION;
   sprintf( cmd, "MAKEBOX <%g %g %g> <%g %g %g> <%d %d %d>",
      -0.5, -0.5, -0.5,  0.5, 0.5, 0.5,  1, 1, 1 );
   local->evaluate( local->data, cmd );
   return AFUNC_OK;
}

ServerRecord ServerDesc[] = {
   { LWMODCOMMAND_CLASS, "Tutorial_Box1", Activate },
   { NULL }
};
```

A Closer Look

Let's break this down and examine the parts in detail.

```
#include <lwserver.h>
#include <lwcmdseq.h>
```

```
   #include <stdio.h>
```

These are the headers required by the plug-in. The first two, *lwserver.h* and *lwcmdseq.h*, are part of the LightWave SDK, in the include directory. *lwserver.h* contains definitions required by every plug-in. These are explained on the Common Elements page of the SDK, but I'll repeat some of that information less formally here. The second SDK header, *lwcmdseq.h*, contains the structure definitions and function prototypes comprising the CommandSequence plug-in class.

> **Classes**: LightWave plug-ins are divided into different types, called classes. Each class does something different, or plugs into LightWave in a different place. The three Modeler classes are called CommandSequence (the kind we're looking at now), MeshDataEdit and MeshEditTool. The CommandSequence class drives Modeler by issuing commands.

We also include the C standard header *stdio.h* to get the function prototype for `sprintf`, which we'll use to build the command string.

The order in which these are listed, meaning whether the C headers or the LightWave SDK headers come first, usually doesn't matter. I put the SDK headers inside angle brackets (`<` and `>`), which tells the compiler to look for them in the "usual place." Most compilers allow you to add directories to the search path for this usual place, and I use this to add the path to the SDK's include directory.

```
   XCALL_( int )
   Activate( long version, GlobalFunc *global, LWModCommand *local,
      void *serverData )
   {
```

This is the activation function, which happens to be the only function in our plug-in. Modeler will call this function when the user starts our plug-in. The actions of the plug-in are complete when this function returns.

> **Activation Function**: Every plug-in has one. This is the entry point of a plug-in, sort of like `main` in a standard C program. Activation functions have a standard list of four arguments. The type of the third one depends on the plug-in class, but the others are always the same. The function doesn't have to be called `Activate`.

The XCALL_ macro is defined in *lwserver.h*. It encloses the return type of the function. XCALL_ is a placeholder for any platform-specific or compiler-specific weirdness that might be required to get the calling conventions right. If you've programmed Microsoft Windows, you know that Win32 defines several macros (WINAPI and CALLBACK, for example) that serve a similar purpose.

Strictly speaking, XCALL_ should be used on every function in your source code that can be called by LightWave. But as of this writing, XCALL_ has no effect on any platform LightWave currently supports, and you might notice that I've gotten somewhat careless in the SDK samples about using it for some callbacks. Just be aware that it might be needed in the future.

For Modeler command plug-ins, the third argument to the activation function is a pointer to an LWModCommand structure, which is defined in *lwcmdseq.h*.

```
      char cmd[ 128 ];
```

Plug-ins can declare data in all the ways any other C program or library can. This is the string in which we'll build our command.

```
      if ( version != LWMODCOMMAND_VERSION )
         return AFUNC_BADVERSION;
```

The first thing an activation function should do is ensure that the version is correct. As LightWave and the SDK develop over time, the definition of LWModCommand can change. The version number passed to your activation function tells you, among other things, which version of LWModCommand Modeler is passing to you.

The headers define symbols for the version number of each class's local data. This is the version that matches the definition in the header. In other words, the version of LWModCommand in *lwcmdseq.h* is LWMODCOMMAND_VERSION. If you recompile your plug-in with newer headers, LWModCommand might be different, and if it is, LWMODCOMMAND_VERSION will be also.

LightWave tries to be backward-compatible with older plug-ins. It will call your activation function with different versions of the local data until you accept one of the versions (by not returning AFUNC_BADVERSION) or until it runs out of versions to try. With a couple of exceptions, it will start with the highest version first, so that you'll run with the highest version of the API you accept. For more about this, see the Compatibility page.

```
sprintf( cmd, "MAKEBOX <%g %g %g> <%g %g %g> <%d %d %d>",
   -0.5, -0.5, -0.5,  0.5, 0.5, 0.5,  1, 1, 1 );
local->evaluate( local->data, cmd );
```

This is where something actually happens. We build a command string containing the command and its arguments, then call the LWModCommand `evaluate` function to issue the command.

> **Function Pointers**: These may be new to you. The plug-in API makes extensive use of them as a means of allowing separate modules (in this case Modeler and the plug-in) to call each other's functions.
>
> A pointer to a function can be used like pointers to anything else. They can appear in arguments, arrays and structures. When you dereference a function pointer, you're calling the function. The `evaluate` member of the LWModCommand structure is a pointer to a function that takes two arguments and returns an int.
>
> A call to a function through a pointer is sometimes written with an explicit dereference operator, like this.
>
> ```
> result = ( *local->evaluate )( local->data, cmd );
> ```
>
> The parentheses are necessary to ensure that the `*` binds to the function pointer. The result of writing the call this way is exactly equivalent to writing it without the `*`, but it may clarify what's going on for human readers, and in rare cases it can prevent the C preprocessor from making incorrect macro substitutions.

The first argument to `evaluate` is the `data` field of the LWModCommand structure. `data` is an opaque pointer, meaning you're not supposed to know what it points to. This is data owned by Modeler, that Modeler uses to keep track of what it's doing. The second argument is the `MAKEBOX` command string.

The `MAKEBOX` command is described on the Modeler Commands page. It takes two required arguments, the coordinates of the low and high corners, and one optional argument, the number of segments along each axis. Each of these arguments is a vector, a triple of three numbers, delimited by angle brackets (`<` and `>`).

Since we're making a cube centered on the origin, all three numbers in each vector are the same. We could simplify the command string somewhat by writing only the first value in each vector. And since the values are constants, we don't need `sprintf` at all. We could have written instead,

```
   local->evaluate( local->data, "MAKEBOX <-.5> <.5> <1>" );
```

When vector components are omitted, their values are taken from the last one included.

```
      return AFUNC_OK;
   }
```

The activation function returns `AFUNC_OK` to indicate that it completed successfully.

The LWModCommand `evaluate` function returns 0 to indicate success, or a non-zero error code if something went wrong. For our simple plug-in, we're ignoring `evaluate`'s return value, but in non-trivial code, you'll probably want to react to errors, and this might involve returning something other than `AFUNC_OK` from your activation function.

```
   ServerRecord ServerDesc[] = {
      { LWMODCOMMAND_CLASS, "Tutorial_Box1", Activate },
      { NULL }
   };
```

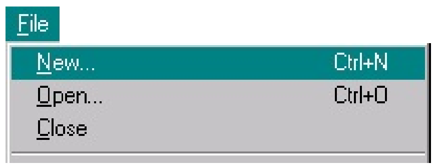
The ServerRecord array lists the plug-ins contained in a plug-in (.p) file. Our source code defines only one plug-in. Its class is `LWMODCOMMAND_CLASS`, its server name is "Tutorial_Box1" and its activation function is `Activate`. Although we don't use it here, the ServerRecord also allows you to specify a user name that differs from the server name, along with other information about the plug-in supplied as an array of tags.

On each platform LightWave supports, the operating system provides a standard method of loading the .p file as a shared or dynamically-linked library and of obtaining the address within the library of the ServerRecord array. On Windows, for example, LightWave uses the Win32 `LoadLibrary` and `GetProcAddress` functions. Once LightWave has the address of the ServerRecord array, it can read the list of plug-ins in the file and find the activation functions.

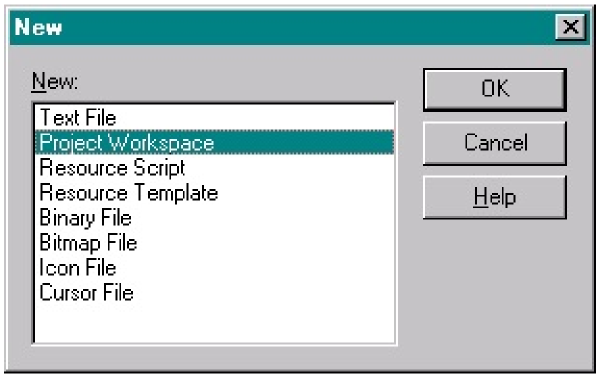
Building the Plug-in

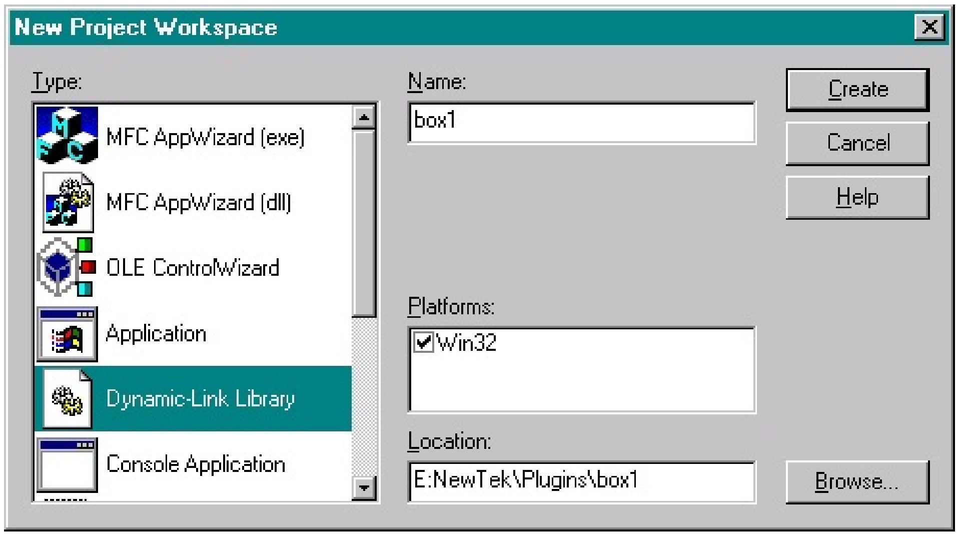
The Compiling page of the SDK gives detailed instructions for creating a .p file from your C source code. But the instructions there can be a little bewildering if your experience with setting compiler switches and building DLLs or shared libraries is limited. So I thought I'd show you how I build plug-ins on my own machine with the compiler I'm using, Microsoft Visual C++ version 4.0 Standard.

This obviously won't be so helpful for people using different compilers and platforms, but hopefully it will at least demystify the process a little for them and demonstrate that no rocket science is involved.



Begin by creating a new project workspace. Make sure the project type is Dynamic-Link Library.



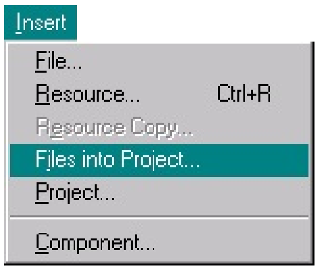

New Project Workspace

Type: MFC AppWizard (exe), MFC AppWizard (dll), OLE ControlWizard, Application, Dynamic-Link Library (selected), Console Application

Name: box1

Platforms: [✓] Win32

Location: E:\NewTek\Plugins\box1

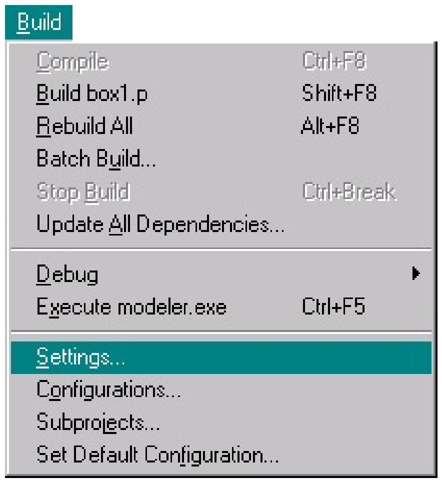


Insert your source files into the project. You also need to insert a small amount of code from the SDK.

The Compiling page tells you how to build the SDK code into a library called *server.lib*, and if you've done that already, you can just insert *server.lib* here. Or you can add *server.lib* to the library files listed on the Link tab of the Settings dialog. The effect is the same.

But if you haven't built *server.lib*, you can instead insert the SDK files *startup.c* and *shutdown.c* from the *source* directory. The advantage of this approach is that you get both debug and release builds of the SDK code without having to worry about keeping track of two versions of *server.lib*.

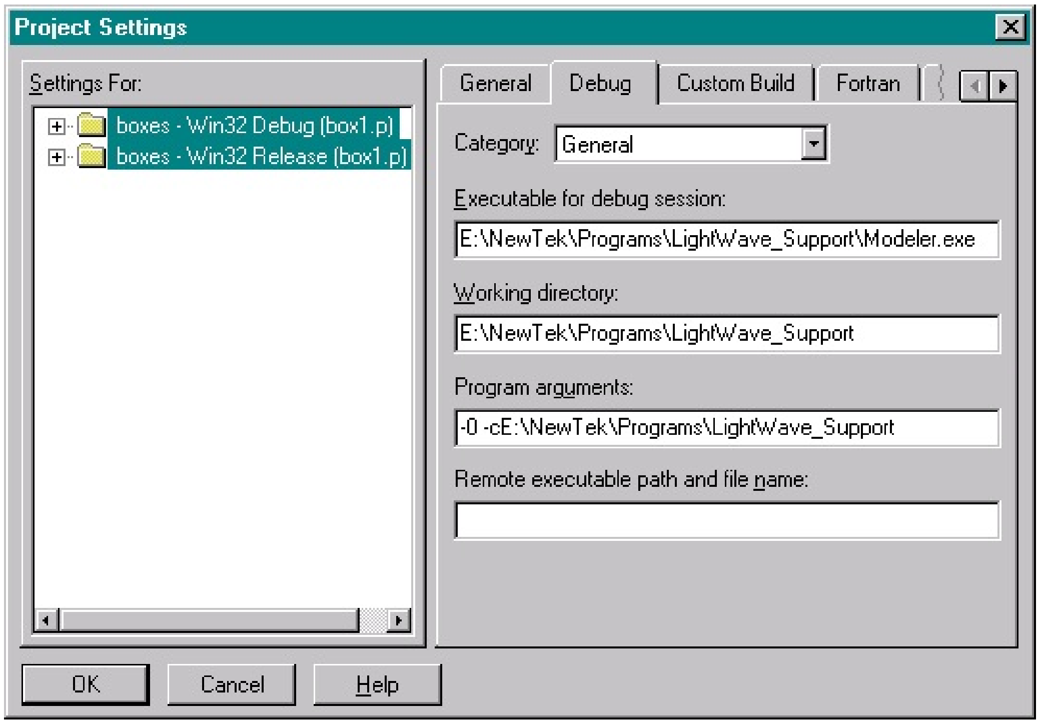
Either way, you'll also need to insert *servmain.c* and *serv.def*. The sole purpose of the .def file is to tell the linker to export the symbol `_mod_descrip`., which is defined in servmain.c. `_mod_descrip` contains your ServerDesc array, among other things, and



exporting it makes it visible to other programs like LightWave that call the Win32 `GetProcAddress` function.

Once you've gotten all your sources into your project, you're ready for the compile and link settings that will turn it into a .p file. Open the settings dialog.

We're only concerned here with settings on three of the tabs: the Debug, C/C++ and Link tabs. The information you enter on the Debug tab allows you to debug your plug-in using the windowed debugger in MSVC. You can set breakpoints in your code, step through it line by line, and examine the values of variables.



boxes - Win32 Debug (box1.p)
boxes - Win32 Release (box1.p)

Category: General

Executable for debug session: E:\NewTek\Programs\LightWave_Support\Modeler.exe

Working directory: E:\NewTek\Programs\LightWave_Support

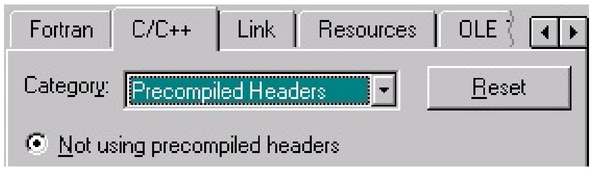
Program arguments: -0 -cE:\NewTek\Programs\LightWave_Support

Remote executable path and file name:

In the edit field labelled "Executable for debug session:", you'll enter the path to your installation of either Layout (Lightwav.exe) or Modeler (Modeler.exe), depending on which part of LightWave your plug-in is intended to run in. Our box plug-in runs in Modeler.

The "Working directory:" will typically be the same as the "Start in:" directory for your LightWave installation. To find that, look on the Shortcut tab of the Properties panel for the icon or the Startup Menu entry that launches LightWave on your system. Also look there for the program arguments you normally use. I usually set mine to run without the Hub and to write the config files in the directory where the program files are.

You might also want to add the -d switch, which runs the LightWave component in a "debug" mode. In this mode, plug-in files are always closed and detached when not in use, making it possible to recompile them while LightWave remains open. LightWave may also create a text file error dump when certain plug-in related problems occur.

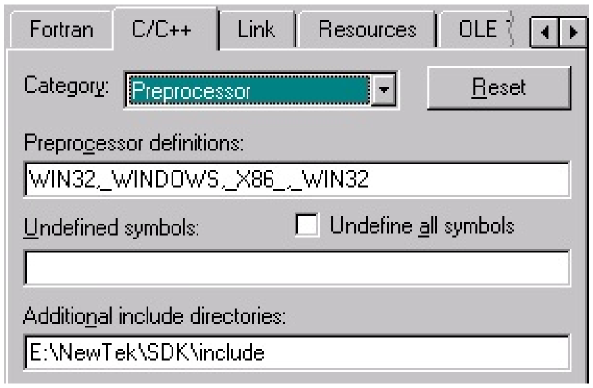


Category: Precompiled Headers

[✓ Not using precompiled headers]

Like a lot of compilers, MSVC gives you the option of precompiling the headers, and it does this by default. The headers for a project tend not to change as often as the C sources, so in theory this can save a small amount of time, especially when you're including the Win32 headers and compiling on a slow machine. But the .pch files MSVC creates can be surprisingly large, over a megabyte for the Win32 headers, and these are created redundantly for each project. And they don't really save that much time. So I always turn this option off.

I also turn off incremental linking, mostly for similar reasons, but also to avoid the small chance that it will introduce errors by failing to rebuild code affected by changes elsewhere in the project.



Category: Preprocessor

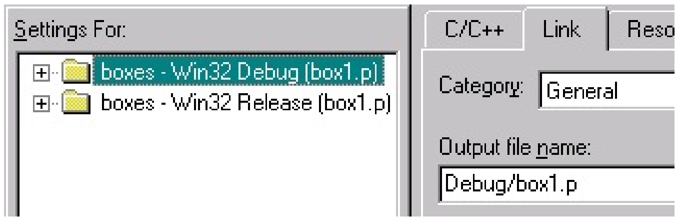
Preprocessor definitions: WIN32,_WINDOWS,_X86_,_WIN32

Undefined symbols:

Undefine all symbols

Additional include directories: E:\NewTek\SDK\include

On the C/C++ tab, under Preprocessor, add `_X86_` and `_WIN32` to the preprocessor definitions. The SDK source and the *lwdisplay.h* header use these symbols to select platform-specific code. In the "Additional include directories:" field, enter the path to the SDK include directory.



Settings For:

boxes - Win32 Debug (box1.p)

boxes - Win32 Release (box1.p)

Category: General

Output file name: Debug/box1.p

The last settings step is renaming the output file so that it has a .p extension. This isn't strictly necessary. You can rename the file later, or simply use it with its default .dll extension, which LightWave has no problem with. If you do rename the file here, make sure only one of the two builds is selected in the left pane before altering the filename. If both are selected, the debug and release builds will have the same name and will overwrite each other.

You're now ready to build. By default, the toolbar includes a dropdown list from which you can select the debug or release builds of your project. During development, you'll usually want to be building and testing the debug version. Once you've chosen the build, select "Build box1.p" (Shift + F8) or "Rebuild All" (Alt + F8) from the Build menu, or hit the little build icon on the toolbar.

To run the plug-in, hit "Execute Modeler.exe" in the same menu. If this is the first time you've run the plug-in, you'll need to install it in LightWave. Use the Add Plug-ins option (in the Modeler/Plug-ins or Layout/Plug-ins menu) and in the file dialog, navigate to the .p file you created. Unless you changed the path, your .p file will be in the Debug directory created by MSVC in your project directory.

To debug your plug-in, select one of the Debug submenu options, typically Go. MSVC will warn you that the LightWave component contains no debugging information. That's OK, since you're not debugging LightWave itself. The debug build of your plug-in does contain this information. Before hitting Debug/Go, you'll typically set one or more breakpoints, so that execution stops at those points and you can examine the state of the plug-in's data. Put the cursor in one of your source files in the MSVC editor and press F9 to set a breakpoint there.

What's Next

If all's gone well, you've learned how to write, compile and run a Modeler plug-in. In the next installment, we'll work on calling Modeler commands in a more sophisticated way, and we'll add a user interface.

Let's Make Another Box

Author Ernie Wright
Date 1 June 2001

In the first installment of this tutorial, we looked at a simple plug-in that creates a box in Modeler by calling the `MAKEBOX` command. In this installment we'll assume you're comfortable with the plug-in basics covered there. We'll call `MAKEBOX` in a different way, and we'll add a user interface. The complete source is in `sample/boxes/box2/box.c`.

Our Second Box Plug-in

In this plug-in, we'll move the box creation out of the activation function into its own function, and we'll use `lookup` and `execute` rather than `evaluate` to issue the `MAKEBOX` command.

Here's our new `makebox` function.

```
int makebox( LWModCommand *local, float *size, float *center,
   int nsegments )
{
   static LWCommandCode ccode = 0;
   DynaValue argv[ 3 ];

   argv[ 0 ].type = DY_VFLOAT;
   argv[ 0 ].fvec.val[ 0 ] = center[ 0 ] - 0.5f * size[ 0 ];
   argv[ 0 ].fvec.val[ 1 ] = center[ 1 ] - 0.5f * size[ 1 ];
   argv[ 0 ].fvec.val[ 2 ] = center[ 2 ] - 0.5f * size[ 2 ];

   argv[ 1 ].type = DY_VFLOAT;
   argv[ 1 ].fvec.val[ 0 ] = center[ 0 ] + 0.5f * size[ 0 ];
   argv[ 1 ].fvec.val[ 1 ] = center[ 1 ] + 0.5f * size[ 1 ];
   argv[ 1 ].fvec.val[ 2 ] = center[ 2 ] + 0.5f * size[ 2 ];

   if ( nsegments ) {
      argv[ 2 ].type = DY_VINT;
      argv[ 2 ].ivec.val[ 0 ] =
      argv[ 2 ].ivec.val[ 1 ] =
      argv[ 2 ].ivec.val[ 2 ] = nsegments;
   }
   else argv[ 2 ].type = DY_NULL;

   if ( !ccode )
      ccode = local->lookup( local->data, "MAKEBOX" );

   return local->execute( local->data, ccode, 3, argv, 0, NULL );
}
```

Now we're getting into some stuff! Before I explain it, you might wonder why we'd bother with this apparently more complicated method at all, when we can use the simpler `evaluate` function. One answer is speed.

The `lookup` and `execute` functions are Modeler's *native* mechanism for processing commands. When you use `evaluate` instead, Modeler parses the command string you pass and then calls the `lookup` and `execute` functions itself. You save some time by using `lookup` and `execute` directly, rather than building an `evaluate` string in your plug-in that Modeler is just going to take apart again.

Also on the plus side, you only have to write your `makebox` function once. After that, you can call it as often as you like with a single line of code, and you can cut and paste it into other plug-ins. The modlib SDK sample is a library of about a hundred functions like `makebox`, each of which issues one of the Modeler commands.

And if the low and high corner arguments to the `MAKEBOX` command are inconvenient, you can make up your own, as we did above with the size and center arguments. Obviously, your `makebox` could call `evaluate`, but if you're going to the trouble to write a `makebox` at all, you might as well go all the way and use the faster `lookup` and `execute`.

Arguments

Except for the last line, `makebox` spends all of its time constructing the arguments for the `execute` function, one of which is the argument list for the `MAKEBOX` command. Let's take a closer look at all of these arguments.

```
   int execute( void *mddata, LWCommandCode ccode, int argc,
      DynaValue *argv, EltOpSelect opsel, int *result );
```

mddata

As with `evaluate`, this is the `data` field of the LWModCommand structure, which provides Modeler the context in which the command will be processed. You never need to know what's in this data. You just have to pass it to the functions that require it.

ccode

A command code obtained by calling `lookup`. A code is used to specify the command instead of the name of the command (in our case, "MAKEBOX") because it's faster. Modeler doesn't have to do a lot of string comparisons to figure out which command you're issuing. The string search need only be done once per Modeler session, when you call `lookup`. In our plug-in, the command code returned by `lookup` is stored in a static variable. We only call `lookup` the first time through `makebox`, when our static variable hasn't been initialized yet.

argc, argv

The argument list for the `MAKEBOX` command. `argv` is an array of DynaValues, and `argc` is the number of elements in the `argv` array. DynaValues are the union of a number of different data types. To initialize a DynaValue, you set the type field, then set the value of the union member appropriate for the type. DynaValues allow you to create a single array in which each element can be a different data type.

opsel

This is the selection mode for the command. It determines which existing geometry your command will interact with. You want some commands to work only on the polygons the user has selected, or on all polygons regardless of the selection, and so on. This is ignored for `MAKEBOX`, so we just set it to 0.

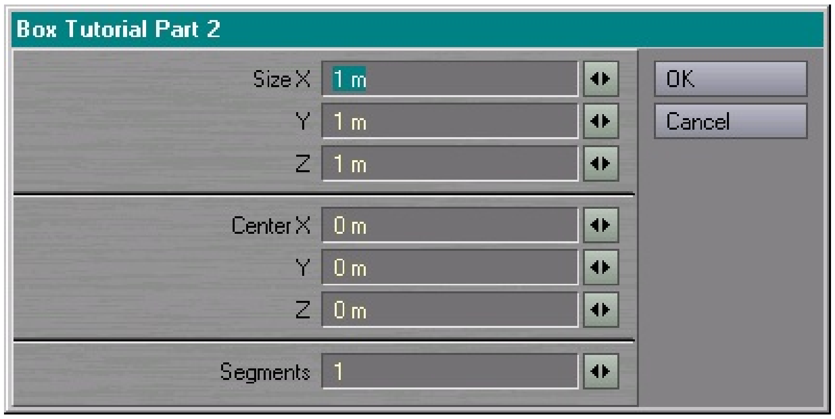
result

A few commands return command-specific result codes in this argument. `MAKEBOX` isn't one of them, so we set this to NULL.

Before we move on to the interface part, there's one other thing to notice here. The third argument to the `MAKEBOX` command is optional. You don't have to specify a number of segments. To support this in our `makebox` function, we allow the `nsegments` argument to be 0. In that case, the third element of the `argv` array is a DynaValue of type `DY_NULL`. The `DY_NULL` type serves as a placeholder when the argument list for a command contains an optional argument that you aren't supplying.

The Interface

Our interface function displays a modal input window, or panel, that looks like this.



Size X	1 m
Y	1 m
Z	1 m
Center X	0 m
Y	0 m
Z	0 m
Segments	1

This is built with XPanels, a component of the platform-independent user interface API. The panel layout and event handling in XPanels are highly automated, so to create this panel, all we have to do is create a list of controls and initialize them, display the panel, and then collect the results.

Here's the source code for our interface function.

```
int get_user( LWXPanelFuncs *xpanf, double *size, double *center,
   int *nsegments )
{
   LWXPanelID panel;
   int ok = 0;

   enum { ID_SIZE = 0x8001, ID_CENTER, ID_NSEGMENTS };
   LWXPanelControl ctl[] = {
      { ID_SIZE,      "Size",     "distance3" },
      { ID_CENTER,    "Center",   "distance3" },
      { ID_NSEGMENTS, "Segments", "integer"   },
      { 0 }
   };
   LWXPanelDataDesc cdata[] = {
      { ID_SIZE,      "Size",     "distance3" },
      { ID_CENTER,    "Center",   "distance3" },
      { ID_NSEGMENTS, "Segments", "integer"   },
      { 0 }
   };
   LWXPanelHint hint[] = {
      XpMIN( ID_NSEGMENTS, 0 ),
      XpMAX( ID_NSEGMENTS, 200 ),
      XpDIVADD( ID_SIZE ),
      XpDIVADD( ID_CENTER ),
      XpEND
   };

   panel = xpanf->create( LWXP_FORM, ctl );
   if ( !panel ) return 0;

   xpanf->describe( panel, cdata, NULL, NULL );
   xpanf->hint( panel, 0, hint );
   xpanf->formSet( panel, ID_SIZE, size );
```

```
      xpanf->formSet( panel, ID_CENTER, center );
      xpanf->formSet( panel, ID_NSEGMENTS, nsegments );

      ok = xpanf->post( panel );

      if ( ok ) {
         double *d;
         int *i;

         d = xpanf->formGet( panel, ID_SIZE );
         size[ 0 ] = d[ 0 ];
         size[ 1 ] = d[ 1 ];
         size[ 2 ] = d[ 2 ];

         d = xpanf->formGet( panel, ID_CENTER );
         center[ 0 ] = d[ 0 ];
         center[ 1 ] = d[ 1 ];
         center[ 2 ] = d[ 2 ];

         i = xpanf->formGet( panel, ID_NSEGMENTS );
         *nsegments = *i;
      }

      xpanf->destroy( panel );
      return ok;
   }
```

Dissecting the Interface

Let's take a closer look.

```
   int get_user( LWXPanelFuncs *xpanf, double *size, double *center,
      int *nsegments )
```

The first argument to `get_user` is a pointer to LWXPanelFuncs. When we get to our activation function, I'll explain where this comes from. But for now, it's a structure containing the functions we need to work with our panel. The other three arguments are used to initialize our controls, and they'll be modified with the values entered by the user. Note that `size` and `center` are arrays of three doubles, while `nsegments` is a pointer to a single integer.

```
      LWXPanelID panel;
      int ok = 0;
```

LWXPanelID is just an opaque pointer that LightWave uses to identify our panel. The double and integer pointers will be used when we retrieve the user's entries from the controls, and `ok` will be true if the user presses OK to dismiss the panel.

```
      enum { ID_SIZE = 0x8001, ID_CENTER, ID_NSEGMENTS };
```

XPanels uses integer codes to identify panel controls. User-defined controls have IDs that start at 0x8001.

```
LWXPanelControl ctl[] = {
   { ID_SIZE,      "Size",     "distance3" },
   { ID_CENTER,    "Center",   "distance3" },
   { ID_NSEGMENTS, "Segments", "integer"   },
   { 0 }
};
LWXPanelDataDesc cdata[] = {
   { ID_SIZE,      "Size",     "distance3" },
   { ID_CENTER,    "Center",   "distance3" },
   { ID_NSEGMENTS, "Segments", "integer"   },
   { 0 }
};
```

These two arrays define our controls. They look redundant, and to a certain extent, for what we're doing, they are. XPanels distinguishes between controls (the widgets drawn on your panel) and data descriptions, which define how you'll represent the values of your controls. You can define controls that don't have corresponding data descriptions, and vice versa. This amount of abstraction is useful for more sophisticated panels, but we're setting up the simplest kind of relationship between our controls and their values, so our control list and our data descriptions are parallel.

Note that XPanels, as of this writing, doesn't support controls of type "integer3". We could simulate one with three separate "integer" controls, but for the sake of simplicity I chose not to do this. As a result, our `makebox` will create the same number of segments along all three axes.

```
LWXPanelHint hint[] = {
   XpMIN( ID_NSEGMENTS, 1 ),
   XpMAX( ID_NSEGMENTS, 200 ),
   XpDIVADD( ID_SIZE ),
   XpDIVADD( ID_CENTER ),
   XpEND
};
```

XPanels automates most aspects of control layout. The rules it uses resemble those used to build LightWave's own interface, so your plug-in's panels are aesthetically and functionally consistent with the rest of the program. In exchange for this, you must sacrifice some low-level control over the appearance of your panel. You can't specify the pixel positions of your controls, for example.

Instead, you use hints to describe your controls and the appearance of your panel in a more abstract way. Here we define a sane range for the segments control and add dividers between the controls. The positions and sizes of the controls, their labels, and decorations like the dividers are all calculated for us. You can also use hints to group controls, put controls on different tabs, establish dependencies between controls, and lots of other things.

```
      panel = xpanf->create( LWXP_FORM, ctl );
      if ( !panel ) return 0;
```

This is where we create the panel. If panel creation fails for some reason, we return 0, which is also what we return when the user presses the Cancel button.

XPanels supports two kinds of panels, called *forms* and *views*. Views are designed primarily for the panels associated with handler class plug-ins in Layout. Views work with *instances*, the unique data pointers returned by each invocation of a handler plug-in. But you're free to choose. You can use forms in your handlers, and we could have used a view here. Forms are just a little easier to grasp initially.

```
      xpanf->describe( panel, cdata, NULL, NULL );
      xpanf->hint( panel, 0, hint );
      xpanf->formSet( panel, ID_SIZE, size );
      xpanf->formSet( panel, ID_CENTER, center );
      xpanf->formSet( panel, ID_NSEGMENTS, nsegments );
```

These calls initialize the panel. The last two arguments to the `describe` function are NULL because our panel is a form. If it had been a view, these arguments would contain pointers to our get and set callbacks. The IDs in the `formSet` calls are value IDs corresponding to entries in the data description array (as opposed to control IDs from our control array). This is a distinction without a difference for us now, but I wanted to plant this in the back of your mind for a time when it *will* make a difference.

```
      ok = xpanf->post( panel );
```

The `post` function displays the panel and waits for the user. Using `post` makes the panel modal, which just means that everything else stops until the user presses OK or Cancel on the panel.

```
      if ( ok ) {
         double *d;
         int *i

         d = xpanf->formGet( panel, ID_SIZE );
         size[ 0 ] = d[ 0 ];
         size[ 1 ] = d[ 1 ];
         size[ 2 ] = d[ 2 ];

         d = xpanf->formGet( panel, ID_CENTER );
         center[ 0 ] = d[ 0 ];
         center[ 1 ] = d[ 1 ];
         center[ 2 ] = d[ 2 ];

         i = xpanf->formGet( panel, ID_NSEGMENTS );
         nsegments[ 2 ] = nsegments[ 1 ] = nsegments[ 0 ] = *i;
      }
```

If the user presses OK, we retrieve the value of each control and store it for use later in our plug-in. The `formGet` function returns a pointer to a variable of the appropriate type for the value.

```
   xpanf->destroy( panel );
   return ok;
}
```

We're done with the panel, so we destroy it. If we needed to open this panel more than once, we could create it once, post it as many times as we need it, then destroy it when we exit.

Activation

All that remains is our activation function, which really hasn't changed very much.

```
XCALL_( int )
Activate( long version, GlobalFunc *global, LWModCommand *local,
   void *serverData )
{
   LWXPanelFuncs *xpanf;
   double size[ 3 ]   = { 1.0, 1.0, 1.0 };
   double center[ 3 ] = { 0.0, 0.0, 0.0 };
   int nsegments = 1;

   if ( version != LWMODCOMMAND_VERSION )
      return AFUNC_BADVERSION;

   xpanf = global( LWXPANELFUNCS_GLOBAL, GFUSE_TRANSIENT );
   if ( !xpanf ) return AFUNC_BADGLOBAL;

   if ( get_user( xpanf, size, center, &nsegments ))
      makebox( local, size, center, nsegments );

   return AFUNC_OK;
}
```

We've added storage for the box parameters so that these can be user-defined, and we call our `get_user` and `makebox` functions rather than `sprintf` and `evaluate`. We've also added a couple of lines having to do with that LWXPanelFuncs pointer, and as promised, I'll now explain where that pointer comes from.

```
   LWXPanelFuncs *xpanf;
   xpanf = global( LWXPANELFUNCS_GLOBAL, GFUSE_TRANSIENT );
   if ( !xpanf ) return AFUNC_BADGLOBAL;
```

The second argument to every activation function is the *global function*. The globals returned by this function are services provided by LightWave. The XPanels global is a common example. When called with an `LWXPANELFUNCS_GLOBAL` argument, the global function returns a pointer to an LWXPanelFuncs structure containing the functions you need to build and display an XPanels interface. The Globals page of the SDK describes the global function in detail and lists the globals available in LightWave.

What's Next

We'll be taking our user interface with us into the next installment, but we'll be leaving behind the `MAKEBOX` command as we explore the construction of boxes from individual points and polygons.

A Mesh Edit Box

Author Ernie Wright
Date 11 June 2001

In the previous installment of this tutorial, we created a user interface and a function that calls Modeler's MAKEBOX command. In this installment, we'll leave the MAKEBOX command behind and instead create our box from its constituent points and polygons. In LightWave nomenclature, creating, deleting and modifying points and polygons is called *mesh editing*, and we'll be using the functions in a MeshEditOp structure provided by Modeler.

We'll also cover the use of the Surface Functions global to build a menu of surface names on our panel, and I'll introduce command line processing, which allows our box plug-in to be called with arguments by other plug-ins.

We're taking a significant step up in complexity, so I've divided the source into three separate files. You can find it in `sample/boxes/box3/box.c`, `ui.c` and `cmdline.c`.

Some Data

With the MAKEBOX command, we didn't need explicit definitions of point positions and polygon vertices, but we do need these in some form now.

```
double vert[ 8 ][ 3 ] = {   /* a unit cube */
   -0.5, -0.5, -0.5,
    0.5, -0.5, -0.5,
    0.5, -0.5,  0.5,
   -0.5, -0.5,  0.5,
   -0.5,  0.5, -0.5,
    0.5,  0.5, -0.5,
    0.5,  0.5,  0.5,
   -0.5,  0.5,  0.5
};

int face[ 6 ][ 4 ] = {     /* vertex indexes */
   0, 1, 2, 3,
   0, 4, 5, 1,
   1, 5, 6, 2,
   3, 2, 6, 7,
   0, 3, 7, 4,
   4, 7, 6, 5
};
```

The `vert` array contains the (*x*, *y*, *z*) coordinates of the eight corner points of a unit cube, which we'll scale to create the points of the box. The `face` array lists the vertices defining each of the six rectangular faces of our box. The numbers correspond to indexes into the `vert` array.

We're also going to define a UV map for the box. (UV mapping is a texture projection method. It associates specific points in 3D space with specific points on a 2D texture, typically an image.)

```
float cuv[ 8 ][ 2 ] = {    /* continuous UVs (spherical mapping) */
   .125f, .304f,
   .375f, .304f,
   .625f, .304f,
   .875f, .304f,
   .125f, .696f,
   .375f, .696f,
   .625f, .696f,
   .875f, .696f
};

float duv[ 2 ][ 2 ] = {    /* discontinuous UVs */
   -0.125f, 0.304f,
   -0.125f, 0.696f
};
```

This is the UV mapping Modeler generates when it uses spherical mapping to initialize a new vertex map.

> **Vertex Map**: Associates a set of vectors with a set of points. UV vmaps contain two floats for each point, the *u* and *v* coordinates. Color vmaps are made up of RGB or RGBA vectors. Weight maps have a single value per point. Point selection sets are implemented as vmaps with no value at all. Type codes for the most common vmap types are defined in *lwmeshes.h*, but you can also define your own custom vmaps.

Mesh Editing

The mesh edit version of our `makebox` function uses the `vert`, `face`, `cuv` and `duv` arrays to create the points and polygons that comprise our box.

```
void makebox( MeshEditOp *edit, double *size, double *center,
   char *surfname, char *vmapname )
{
   LWDVector pos;
   LWPntID pt[ 8 ], vt[ 4 ];
```

```
      LWPolID pol[ 6 ];
      int i, j;

      for ( i = 0; i < 8; i++ ) {
         for ( j = 0; j < 3; j++ )
            pos[ j ] = size[ j ] * vert[ i ][ j ] + center[ j ];
         pt[ i ] = edit->addPoint( edit->state, pos );
         edit->pntVMap( edit->state, pt[ i ],
            LWVMAP_TXUV, vmapname, 2, cuv[ i ] );
      }

      for ( i = 0; i < 6; i++ ) {
         for ( j = 0; j < 4; j++ )
            vt[ j ] = pt[ face[ i ][ j ]];
         pol[ i ] = edit->addFace( edit->state, surfname, 4, vt );
      }

      edit->pntVPMap( edit->state, pt[ 3 ], pol[ 4 ],
         LWVMAP_TXUV, vmapname, 2, duv[ 0 ] );
      edit->pntVPMap( edit->state, pt[ 7 ], pol[ 4 ],
         LWVMAP_TXUV, vmapname, 2, duv[ 1 ] );
   }
```

Let's go through it one step at a time.

```
   void makebox( MeshEditOp *edit, double *size, double *center,
      char *surfname, char *vmapname )
   {
```

Instead of the LWModCommand structure we passed to the previous version of `makebox`, the first argument to this one is a MeshEditOp, which contains all of the mesh editing functions. We'll be getting this from our activation function. The other arguments control the size and center of the box, the surface for the box faces, and the name of the vertex map that will hold our UVs. To simplify this a bit, there's no argument for the number of segments, nor will we support more than one.

```
      LWDVector pos;
      LWPntID pt[ 8 ], vt[ 4 ];
      LWPolID pol[ 6 ];
      int i, j;
```

The LWPntID and LWPolID types are used to identify points and polygons. They're returned from functions that create these elements, and they're later passed as arguments when you need to refer to them. The LWDVector type is just an array of three doubles.

```
      for ( i = 0; i < 8; i++ ) {
         for ( j = 0; j < 3; j++ )
            pos[ j ] = size[ j ] * vert[ i ][ j ] + center[ j ];
```

The position of each point is the size multiplied by the coordinates for a unit cube, offset by the center position.

```
         pt[ i ] = edit->addPoint( edit->state, pos );
```

The `addPoint` function creates a point at the specified position. We'll need to refer to the points we create when we connect them together to form the faces, so we store the point IDs.

```
         edit->pntVMap( edit->state, pt[ i ],
            LWVMAP_TXUV, vmapname, 2, cuv[ i ] );
      }
```

While we're in the points loop, we also initialize the UV values for each point. `pntVMap` takes a point ID, a vertex map, and a vector of two floats containing the UV coordinates.

Vmaps are defined by a name, a type code and a vector dimension. The name is what the user sees in the interface when vmaps are listed. Type codes for common vmap types like texture UV maps are defined in lwmeshes.h, but it's also possible to create custom vmap types. The vector is an array of floats associated with a point, and the dimension is just the number of elements in the vector. UV vmaps contain two floats for each point, the *u* and *v* coordinates.

If the vmap doesn't exist at the time of the call, `pntVMap` creates it.

```
      for ( i = 0; i < 6; i++ ) {
         for ( j = 0; j < 4; j++ )
            vt[ j ] = pt[ face[ i ][ j ]];
```

The `vt` array contains four point IDs, one for each vertex of a box face. The direction of the polygon normal depends on the order in which the points are listed. The point indexes in the `face` array are listed in clockwise order as seen from the polygon's visible side.

```
         pol[ i ] = edit->addFace( edit->state, surfname, 4, vt );
      }
```

The `addFace` function creates a polygon with the given surface name and vertex list. If the surface doesn't exist, `addFace` creates it.

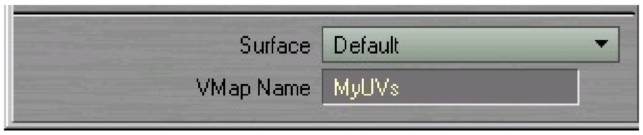
```
      edit->pntVPMap( edit->state, pt[ 3 ], pol[ 4 ],
         LWVMAP_TXUV, vmapname, 2, duv[ 0 ] );
      edit->pntVPMap( edit->state, pt[ 7 ], pol[ 4 ],
         LWVMAP_TXUV, vmapname, 2, duv[ 1 ] );
   }
```

Finally, we add two discontinuous UV values to the vmap. Most points have a single UV value. The (*u*, *v*) is the same at a given point for all faces that use the point as a vertex. Discontinuous UVs override this value, but only for one of the polygons that shares the point. This fixes the seam problem, where two points are on opposite sides of a discontinuity, or seam, in the texture.

In our case, we're fixing up the -X face of the box, where the left and right sides of an image map would meet if it used our vmap. Without this fix, the interpolation of *u* across this face would be "backwards," the reverse of that across the -Z, +X and +Z faces.

Surface Name List

In our interface, we need to give the user a way to specify the surface and vmap names. For vmap names, we'll provide a simple text edit field. But to show what else we can do, we'll build a popup menu for the surface names.



Surface: Default
VMap Name: MyUVs

The declaration of our Surface popup control and its data description in our `get_user` function looks like this.

```
LWXPanelControl ctl[] = {
   ...
   { ID_SURFLIST, "Surface", "iPopChoice" }, ...

LWXPanelDataDesc cdata[] = {
   ...
   { ID_SURFLIST, "Surface", "integer" }, ...
```

The value of a popup control is a 0-based integer index into the list of menu items. We need a way to give XPanels our list of surface names. Although there are other ways to do it, we'll populate the menu using an `XpSTRLIST` hint.

```
LWXPanelHint hint[] = {
   ...
   XpSTRLIST( ID_SURFLIST, surflist ), ...
```

The second argument to the `XpSTRLIST` macro is an array of strings. The last element of the array is NULL to mark the end of the list.

If we knew the item list in advance, we could simply declare it like this:

```
char *menulist[] = { "Apples", "Oranges", "Bananas", NULL };
```

But we don't know in advance what surfaces exist in Modeler, so we have to allocate and initialize one of these string arrays dynamically, when our plug-in is executed.

To build the surface name list, we'll use the `first`, `next` and `name` routines provided by the Surface Functions global. `first` and `next` walk you through the linked list of surface descriptions in Modeler, and `name` returns the name of a surface, given its LWSurfaceID. The function in our plug-in that allocates and initializes the surface name list is called `init_surflist`.

```
int init_surflist( LWSurfaceFuncs *surff )
{
   LWSurfaceID surfid;
   const char *name;
   int i, count = 0;
```

The first thing it does is count the surfaces.

```
   surfid = surff->first();
   while ( surfid ) {
      ++count;
      surfid = surff->next( surfid );
   }
```

It's possible for the count to be 0. In that case, we create a list with a single entry, "Default", and return a count of 1.

```
   if ( !count ) {
      surflist = calloc( 2, sizeof( char * ));
      surflist[ 0 ] = malloc( 8 );
      strcpy( surflist[ 0 ], "Default" );
      return 1;
   }
```

Otherwise, we allocate an array of `count` + 1 strings. The extra one is the NULL string that marks the end of the list.

```
   surflist = calloc( count + 1, sizeof( char * ));
   if ( !surflist ) return 0;
```

Now we loop through the surface list again using first and next, this time copying the surface name into our string array. If anything goes wrong while we're doing this, we call our `free_surflist` function, which frees each string and the string array, and then return a count of 0.

```
      surfid = surff->first();
      for ( i = 0; i < count; i++ ) {
         name = surff->name( surfid );
         if ( !name ) {
            free_surflist();
            return 0;
         }
         surflist[ i ] = malloc( strlen( name ) + 1 );
         if ( !surflist[ i ] ) {
            free_surflist();
            return 0;
         }
         strcpy( surflist[ i ], name );
         surfid = surff->next( surfid );
      }
```

We're done.

```
      return count;
   }
```

This function is fairly typical of the way you'll get and use information from LightWave. The Surface Functions global doesn't provide a canned `getSurfaceNameArray` function, and XPanels doesn't have an "iPopSurfaceName" control type. This arguably places a greater burden on plug-in authors, but it also offers greater flexibility. Suppose you only want to list green surfaces, or surface names starting with the letter B?

Doing It Differently

Before we leave the surface list, I want to call attention to things we can and can't do differently with it. We can't call `init_surflist` from within `get_user`. At that point, it's already too late. The (not yet initialized) `surflist` has already been written into the hint array. For the same reason, we can't declare the hint array `static`.

It's also not easy to write the correct value for `surflist` into the hint array after it's been declared, because it's hard to know what its array index will be after the various `Xp` macros have been expanded. For example, our hint array after expansion looks like the following.

```
   LWXPanelHint hint[] = {
      (( void * )( 0x3D000B03 )),             /* XPTAG_LABEL   */
      (( void * )( 0 )),
      (( void * )( "Box Tutorial Part 3" )),
      (( void * )( 0 )),                      /* XPTAG_NULL    */
      (( void * )( 0x3D021481 )),             /* XPTAG_DIVADD  */
      (( void * )( 0x00008001 )),             /* ID_SIZE       */
      (( void * )( 0 )),                      /* XPTAG_NULL    */
      (( void * )( 0x3D021481 )),             /* XPTAG_DIVADD  */
      (( void * )( 0x00008002 )),             /* ID_CENTER     */
      (( void * )( 0 )),                      /* XPTAG_NULL    */
      (( void * )( 0x3D014503 )),             /* XPTAG_STRLIST */
      (( void * )( 0x00008003 )),             /* ID_SURFLIST   */
      (( void * )( surflist )),
      (( void * )( 0 )),                      /* XPTAG_NULL    */
      (( void * )( 0 )),                      /* XPTAG_END     */
   };
```

Without expanding it by hand like this, it's not at all obvious that `surflist` ends up in `hint[12]`.

There's another way to give XPanels the items in a popup, however. Instead of the `XpSTRLIST` macro, you can use `XpPOPFUNCS` to pass a pair of callbacks that XPanels will call when it needs to know the item count and the name of each item. Since it doesn't paint you into a corner the way `XpSTRLIST` can, this is the preferred method for item lists that must be built at runtime. I chose not to use it here because I decided, somewhat arbitrarily, that an array would be easier to understand than the callbacks would be. But we will use `XpPOPFUNCS` in Part 4.

Command Line Processing

Command sequence plug-ins can call other command sequence plug-ins using Modeler's `CMDSEQ` command. `CMDSEQ` allows you to pass arguments to the called plug-in.

Our box plug-in, in other words, can be called by other Modeler plug-ins. When used this way, it becomes just another command! For this to be really useful, we need to process the command line so that we can accept arguments. Modeler passes the command line to us in the `argument` field of the LWModCommand structure.

Our parameters are the box size and center, the surface name, and the vmap name. The obvious command line for us would be

```
   <size> <center> surfname vmapname
```

where the size and center arguments are vectors enclosed in angle brackets, and the other two arguments are strings, possibly enclosed in double quotes. For example,

```
<1.5 2.5 3.5> <0> "Bram Stoker" Dracula
```

Modeler passes this to us as a single string. It's up to us to divide it into an array of tokens similar to the `argv` array passed to a C console program's `main` function. It's a little tricky. We can't just call the C runtime function `strtok`, since spaces are only delimiters if they're not inside double quotes or angle brackets, and angle brackets are only delimiters if they're not inside double quotes.

We'd also like to support some of the same conventions Modeler itself does for command arguments: Vector components after the first are optional, and if omitted, are assigned the value of the last component present. Strings that don't contain spaces don't have to be enclosed in double quotes.

It might seem like we're making work for ourselves by supporting a more complicated command line. But keep in mind that users can also write a command line for our plug-in when they assign it to a key or a menu, so conforming to Modeler command conventions is usually a good idea. We'll also get some help from LightWave for converting the vectors.

Our `get_argv` function breaks the command line into an array of token strings. It just looks at each character in the command string and decides whether to add it to the existing token or start a new one. Tokenizing a string is covered in numerous general programming texts, so I won't go into detail about how `get_argv` is implemented.

The function that calls `get_argv` is `parse_cmdline`.

```
int parse_cmdline( DynaConvertFunc *convert, const char *cmdline,
   double *size, double *center, char *surfname, char *vmapname )
{
   DynaValue from = { DY_STRING }, to = { DY_VDIST };
   int argc;
   char **argv;
```

The first argument is the function returned by the Dynamic Conversion global. This function takes a DynaValue of one type (in our case, a string) and returns one of a different type (a 3-vector of distance values). We'll use this to convert the size and center vector strings into arrays of three doubles. This gives us automatic support for the default values of missing vector components.

```
      argv = get_argv( cmdline, 4, &argc );

      if ( argc == 4 ) {
         from.str.buf = argv[ 0 ];
         to.fvec.defVal = 1.0;
         convert( &from, &to, NULL );

         size[ 0 ] = to.fvec.val[ 0 ];
         size[ 1 ] = to.fvec.val[ 1 ];
         size[ 2 ] = to.fvec.val[ 2 ];

         from.str.buf = argv[ 1 ];
         to.fvec.defVal = 0.0;
         convert( &from, &to, NULL );

         center[ 0 ] = to.fvec.val[ 0 ];
         center[ 1 ] = to.fvec.val[ 1 ];
         center[ 2 ] = to.fvec.val[ 2 ];

         strcpy( surfname, argv[ 2 ] );
         strcpy( vmapname, argv[ 3 ] );
      }

      free_argv( argc, argv );
      return ( argc == 4 );
   }
```

If `get_argv` finds four tokens in the command string, the first two are assumed to be vectors and are assigned to the size and center arrays after conversion. The last two are assumed to be surface and vmap names. The function returns TRUE if the argument count is 4.

Activation

The activation function is where we pull all of this together.

```
   XCALL_( int )
   Activate( long version, GlobalFunc *global, LWModCommand *local,
      void *serverData )
   {
      DynaConvertFunc *dynaf;
      LWXPanelFuncs *xpanf;
      LWSurfaceFuncs *surff;
      MeshEditOp *edit;
```

We'll get these four things by calling functions in Modeler.

```
      double size[ 3 ]   = { 1.0, 1.0, 1.0 };
      double center[ 3 ] = { 0.0, 0.0, 0.0 };
      char surfname[ 128 ];
      char vmapname[ 128 ] = "MyUVs";
      int ok = 0;
```

This is where our parameters are kept.

```
      if ( version != LWMODCOMMAND_VERSION )
         return AFUNC_BADVERSION;
```

Like always, the first thing we do is make sure Modeler is calling us with the right version of LWModCommand.

```
      if ( local->argument[ 0 ] ) {
```

The `argument` string is always valid. To decide whether we've received a command line, we need to see whether the string is empty.

```
         dynaf = global( LWDYNACONVERTFUNC_GLOBAL, GFUSE_TRANSIENT );
         if ( !dynaf ) return AFUNC_BADGLOBAL;
         ok = parse_cmdline( dynaf, local->argument,
            size, center, surfname, vmapname );
         if ( !ok ) return AFUNC_BADLOCAL;
      }
```

If it isn't empty, we get our parameters from the command line instead of displaying our interface.

```
      else {
         xpanf = global( LWXPANELFUNCS_GLOBAL, GFUSE_TRANSIENT );
         surff = global( LWSURFACEFUNCS_GLOBAL, GFUSE_TRANSIENT );
         if ( !xpanf || !surff ) return AFUNC_BADGLOBAL;
         if ( !init_surflist( surff )) return AFUNC_BADGLOBAL;
         ok = get_user( xpanf, size, center, surfname, vmapname );
         free_surflist();
      }
```

If we don't have a command line, we display our interface as before.

```
      if ( ok ) {
         edit = local->editBegin( 0, 0, OPSEL_GLOBAL );
         if ( edit ) {
            makebox( edit, size, center, surfname, vmapname );
            edit->done( edit->state, EDERR_NONE, 0 );
         }
      }
```

If we got parameters from *somewhere*, either the command line or our interface, we perform the mesh edit that creates our box. Between the calls to `local->editBegin` and `edit->done`, we can't call any commands. These calls are the boundaries of a single undo atom. Mesh edits aren't actually applied until you call `done`, so from the point of view of commands, the geometry database is in an indeterminate state.

We should probably track errors that might occur in `makebox` and pass something other than `EDERR_NONE` to `done` if something goes wrong, but I left that out because we had a lot of ground to cover. Don't be lazy like me. Stuff *can* go wrong.

```
      return AFUNC_OK;
   }
```

But life is good.

What's Next

Up to now, we've been writing imperative code. It marches from beginning to end, pausing only once to allow the user to type some numbers. In the final installment, we'll see how to turn our plug-in into an event-driven tool that allows the user to size and center the box interactively.

A Tool Box

Author Ernie Wright
Date 3 July 2001

In the first three installments of this tutorial, I introduced the basics of plug-in creation, including the organization of plug-ins into classes, the use of the SDK headers, the activation function, the server record, and function pointers. We walked through the build process with a specific compiler. We used globals that allowed us to create a user interface, query the surface list, and convert between text and binary representations of numbers. We learned how to process a command line so that our plug-in can be run in batch mode. And we created a box in Modeler, both by issuing a command (in two different ways) and by calling mesh edit functions.

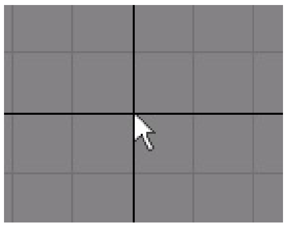
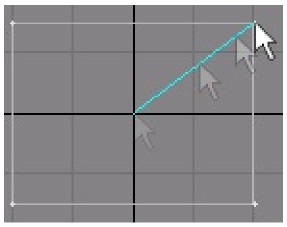
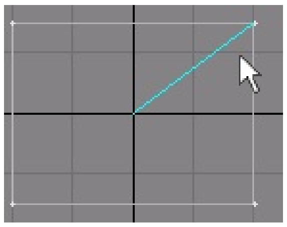
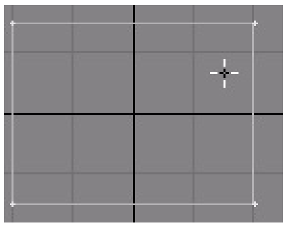
In this final installment, we'll apply what we've learned to create a *tool*, a plug-in that interacts with the user in the same way that Modeler's native tools do. The user will be able to click and drag in Modeler's interface to position and size our box, and our non-modal panel will open when the user requests our numeric options.

Unlike the first three versions of our box plug-in, this one isn't a CommandSequence plug-in. Modeler tools are of the MeshEditTool class. The complete source code for the box tool can be found in `sample/boxes/box4/box.c`, `tool.c` and `ui.c`. Because it's difficult to use a windowed debugger to trace the execution of code that responds to mouse clicks and drags, I've also written debug versions of the tool and interface modules called `wdbtool.c` and `wdbui.c` that write event information to a file. `wdbtool.c` contains a few lines of Windows-specific code related to hooking mouse events. Hopefully they can be easily replaced for use with other operating systems.

The Basic Idea

Tool plug-ins supply a set of callbacks, functions that Modeler calls while the tool is active. These callbacks respond to user actions by drawing the tool and generating the geometry that the tool creates or modifies.

A *handle* is a point that the user can grab and move to change the operation of the tool. Our plug-in will support two handles, one for the center of the box, and the other at the (+*x*, +*y*, +*z*) corner to control the size. (More sophisticated tools will usually support many more handles.) The following table shows the user clicking to establish the box center, then dragging to pull out the corner handle. The callbacks are listed in the order in which they're typically called during each part of this operation.

mouse down	**mouse move**	**mouse up**	**spacebar**
			
• `count` • `start` • `dirty` • `test`	• `handle` • `adjust` • `dirty` • `test` • `build` • `draw`	• `end`	• `done`

We'll cover the implementation of each of these callbacks, pretty much in the same order. But before we do that, note that all of the callbacks take an LWInstance (a pointer to void) as their first argument. This is the tool's *instance*, a structure we design to hold all of the information we need to maintain the tool's state and generate the geometry. Our instance data structure is called BoxData. One of these is allocated in our activation function, and it persists until the user is finished with the tool.

Count, Start

These two callbacks are related. They're only called when the user clicks the left mouse button to begin dragging the tool, and `start` is only called if `count` returns 0.

```
static int Count( BoxData *box, LWToolEvent *event )
{
   return box->active ? 2 : 0;
}
```

From our tool's point of view, there are two different kinds of mouse down events. The first is the initial mouse down, before any box has been dragged out and before we've drawn the handles. For that case, our `box->active` is FALSE, and our `Count` returns 0, so that our `Start` will be called. The other kind of mouse down occurs after the first one. The user is modifying an existing box, rather than starting a new one. In this second case, `Count` returns 2 (because we have 2 handles), and Modeler doesn't call `Start`.

```
static int Start( BoxData *box, LWToolEvent *event )
{
   int i;

   if ( !box->active )
      box->active = 1;

   for ( i = 0; i < 3; i++ ) {
      box->center[ i ] = event->posSnap[ i ];
      box->size[ i ] = 0.0;
   }
   calc_handles( box );

   return 1;
}
```

When Modeler calls `Start`, the user has just clicked the left mouse button to begin a new box. We make sure `box->active` is now TRUE, and we set the size of the box to 0 and the center to the point at which the user clicked. We initialize the precalculated handle positions, then return 1, the index of the second handle, to indicate that the user has grabbed the sizing handle. While the left mouse button remains down, the `handle` callback will only be called for this handle.

Dirty, Test

These two callbacks are also somewhat related. The `dirty` callback tells Modeler whether the tool needs to be redrawn on the screen. The `test` callback tells Modeler whether the tool needs to create new geometry or discard existing geometry.

```
static int Dirty( BoxData *box )
{
   return box->dirty ? LWT_DIRTY_WIREFRAME | LWT_DIRTY_HELPTEXT : 0;
}
```

`Dirty` is only concerned with the tool's appearance to the user. After the initial mouse down for a new box, `box->dirty` is FALSE, since we haven't drawn anything yet, and we tell Modeler that nothing needs to be redrawn. During mouse move events, our `Adjust` callback is called, and this sets `box->dirty` to TRUE so that we get redrawn to follow the user's mouse moves. We're also dirty after receiving reset and activate events in our `Event` callback. Our `Draw` callback sets `box->dirty` to FALSE again after redrawing the tool.

```
   static int Test( BoxData *box )
   {
      return box->update;
   }
```

Like `box->dirty`, our `Adjust` and `Event` callbacks set `box->update`, depending on our tool's state at that point. `Build` also sets it (to `LWT_TEST_NOTHING`) after creating the box geometry. Our `Test` just returns the value in `box->update`.

Handle

This callback tells Modeler about one of our handles.

```
   static int Handle( BoxData *box, LWToolEvent *event, int handle,
      LWDVector pos )
   {
      if ( handle >= 0 && handle < 2 ) {
         pos[ 0 ] = box->hpos[ handle ][ 0 ];
         pos[ 1 ] = box->hpos[ handle ][ 1 ];
         pos[ 2 ] = box->hpos[ handle ][ 2 ];
      }
      return handle + 1;
   }
```

`Handle` is called during mouse moves, but only for the handle the user is currently moving. It's also called right after mouse down, if `Count` returns a non-zero number of handles. In that case, it's called for every handle, and Modeler uses the positions to determine which handle the user has selected. The return value is the priority of the handle, which is used to decide between handles that overlap visually (have the same apparent position in the viewport). When the user points to two or more overlapping handles, Modeler chooses the one with the highest priority.

Adjust

The `adjust` callback is called during mouse moves to tell you that a handle is being dragged.

```
   static int Adjust( BoxData *box, LWToolEvent *event, int handle )
   {
      if ( event->portAxis >= 0 ) {
         if ( event->flags & LWTOOLF_CONSTRAIN ) {
            int x, y, xaxis[] = { 1, 2, 0 }, yaxis[] = { 2, 0, 1 };
            x = xaxis[ event->portAxis ];
            y = yaxis[ event->portAxis ];
            if ( event->flags & LWTOOLF_CONS_X )
               event->posSnap[ x ] -= event->deltaSnap[ x ];
            else if ( event->flags & LWTOOLF_CONS_Y )
               event->posSnap[ y ] -= event->deltaSnap[ y ];
         }
      }
```

Before we move the handle, we check whether its new position should be quantized or fixed by a constraint. Typically, this is to account for the user holding down the Ctrl key. The fact that Modeler doesn't do this for us means that we aren't *required* to honor this convention, but in our case (and in most cases), we have no reason not to.

```
      if ( handle == 0 ) {  /* center */
         box->center[ 0 ] = event->posSnap[ 0 ];
         box->center[ 1 ] = event->posSnap[ 1 ];
         box->center[ 2 ] = event->posSnap[ 2 ];
      }
      else if ( handle == 1 ) {  /* corner */
         box->size[ 0 ] = 2.0 * fabs( event->posSnap[ 0 ]
            - box->center[ 0 ] );
         box->size[ 1 ] = 2.0 * fabs( event->posSnap[ 1 ]
            - box->center[ 1 ] );
         box->size[ 2 ] = 2.0 * fabs( event->posSnap[ 2 ]
            - box->center[ 2 ] );
      }

      calc_handles( box );
      box->dirty = 1;
      box->update = LWT_TEST_UPDATE;
      return handle;
   }
```

If the user's moving the center handle, we set the box center to the new position, and if the user is moving the size handle, we recalculate the size. In both cases, we precalculate the handle positions for the next `Handle` and `Draw` calls, and we tell Modeler that we need to be both redrawn and rebuilt.

Build

Finally! This callback creates geometry based on what the user is doing.

```
   static LWError Build( BoxData *box, MeshEditOp *edit )
   {
      makebox( edit, box );
      box->update = LWT_TEST_NOTHING;
      return NULL;
   }
```

All we have to do here is call our old friend `makebox`, passing it the MeshEditOp and the size and center set by the user. And since we've just built the geometry, we set `box->update` to `NOTHING`.

Draw

Here we draw the tool itself. We don't have to draw the geometry we create, since Modeler takes care of that for us.

```
static void Draw( BoxData *box, LWWireDrawAccess *draw )
{
   if ( !box->active ) return;
   draw->moveTo( draw->data, box->hpos[ 0 ], LWWIRE_SOLID );
   draw->lineTo( draw->data, box->hpos[ 1 ], LWWIRE_ABSOLUTE );
   box->dirty = 0;
}
```

To keep this simple, we're drawing a single line segment connecting our two handles. More typically, you'll draw a bounding box or some other representation of the scope of your tool's effects, and you'll draw the handles in some way, so that the user knows where they are.

Help

The `help` callback returns a line of text that Modeler draws while the tool is selected. Modeler calls `Help` whenever `Dirty` returns the `LWT_DIRTY_HELPTEXT` bit. It also calls `Help` each time the user moves the mouse cursor to a new viewport, so that you can return a different string for each view.

```
static const char *Help( BoxData *box, LWToolEvent *event )
{
   static char buf[] = "Box Tool Plug-in Tutorial";
   return buf;
}
```

Event

This is called when the user drops, resets or re-activates the tool.

```
static void Event( BoxData *box, int code )
{
   switch ( code )
   {
      case LWT_EVENT_DROP:
      if ( box->active ) {
         box->update = LWT_TEST_REJECT;
         break;
      }
```

The user can drop a tool by clicking in a blank area of Modeler's interface outside the viewports. Generally this means that the user wants to discard the geometry created with the tool, so if we've created some geometry (`box->active` is TRUE), we set `box->update` to `LWT_TEST_REJECT`, so that Modeler will discard the geometry the next time it calls `Test`. If `box->active` is FALSE, we fall through to the next case, treating a drop like a reset.

```
         case LWT_EVENT_RESET:
            box->size[ 0 ] = box->size[ 1 ] = box->size[ 2 ] = 1.0;
            box->center[ 0 ] = box->center[ 1 ] = box->center[ 2 ] = 0.0;
            strcpy( box->surfname, "Default" );
            strcpy( box->vmapname, "MyUVs" );
            box->update = LWT_TEST_UPDATE;
            box->dirty = 1;
            calc_handles( box );
            break;
```

A reset event occurs when the user selects the Reset action on Modeler's Numeric panel. We set all of the box parameters to default values and set our state variables so that Modeler will both rebuild and redraw us.

```
         case LWT_EVENT_ACTIVATE:
            box->update = LWT_TEST_UPDATE;
            box->active = 1;
            box->dirty = 1;
            break;
      }
   }
```

An activate event can be triggered from the Numeric window or with a keystroke, and it should restart the edit operation with its current settings.

End, Done

These sound confusingly alike. The `end` callback is called at the completion of a mouse down, mouse move, mouse up sequence. While the tool is selected, you may get any number of `end` calls. The `done` callback is called when the user is finished with the tool and has deselected it, and it's typically used to free memory allocated by the activation function.

```
   static void End( BoxData *box, int keep )
   {
      box->update = LWT_TEST_NOTHING;
      box->active = 0;
   }
```

Our `End` sets `box->update` to `NOTHING` and `box->active` to FALSE, the state we want our tool data to be in the next time `Count` is called.

```
static void Done( BoxData *box )
{
   free( box );
}
```

Our `Done` frees the BoxData structure.

The Interface

The panel we create for a tool is displayed inside Modeler's Numeric panel when the tool is active. We don't open it ourselves. We create the panel in yet another callback, and Modeler takes care of opening or closing it. The panel becomes just another way for the user to interact with the tool. As settings are changed on the panel, the geometry is changed and the tool is redrawn, just as if the user were dragging the mouse in the viewport.

So our panel is now non-modal. It differs from previous incarnations of our interface in a couple of other ways, too. Since tools use instances (our BoxData structure), it's more natural to make our panel an `LWXP_VIEW` instead of an `LWXP_FORM`. And the surface name list is built with the popup callbacks I avoided in Part 3.

```
LWXPanelID Panel( BoxData *box )
{
   LWXPanelID panel;

   static LWXPanelControl ctl[] = {
      { ID_SIZE,     "Size",      "distance3"  },
      { ID_CENTER,   "Center",    "distance3"  },
      { ID_SURFLIST, "Surface",   "iPopChoice" },
      { ID_VMAPNAME, "VMap Name", "string"     },
      { 0 }
   };
   static LWXPanelDataDesc cdata[] = {
      { ID_SIZE,     "Size",      "distance3" },
      { ID_CENTER,   "Center",    "distance3" },
      { ID_SURFLIST, "Surface",   "integer"   },
      { ID_VMAPNAME, "VMap Name", "string"    },
      { 0 }
   };
   LWXPanelHint hint[] = {
      XpLABEL( 0, "Box Tutorial Part 4" ),
      XpPOPFUNCS( ID_SURFLIST, get_surfcount, get_surfname ),
      XpDIVADD( ID_SIZE ),
      XpDIVADD( ID_CENTER ),
      XpEND
   };
```

The control and data description arrays are the same as before, with one important difference: they've been declared static. Our panel is no longer modal. It persists after the `Panel` function returns, and the control and data descriptions must also.

The `XpSTRLIST` hint has been replaced by an `XpPOPFUNCS` hint that tells XPanels to use the `get_surfcount` and `get_surfname` callbacks with the surface name popup. These callbacks will be called to initialize the list each time the user clicks on it to open it. They use the same techniques for enumerating the surfaces in Modeler that `init_surflist` used in Part 3.

```
      panel = xpanf->create( LWXP_VIEW, ctl );
      if ( !panel ) return NULL;

      xpanf->describe( panel, cdata, Get, Set );
      xpanf->hint( panel, 0, hint );

      return panel;
   }
```

Recall that in Part 3, the third and fourth arguments to `describe` were NULL. Since our panel is a view, we now pass get and set callbacks.

Get, Set

It's easy to get these two mixed up. Just try to remember that the names are from LightWave's point of view, not yours (you're the server, LightWave is the client). XPanels calls the `Get` callback when it wants to get the value of a control from you. It calls the `Set` callback when it wants you to write the value of a control into your instance data.

```
   static void *Get( BoxData *box, unsigned long vid )
   {
      static int i;

      switch ( vid ) {
         case ID_SIZE:      return &box->size;
         case ID_CENTER:    return &box->center;
         case ID_SURFLIST:  i = get_surfindex( box->surfname );
                            return &i;
         case ID_VMAPNAME:  return &box->vmapname;
         default:           return NULL;
      }
   }
```

`Get` is usually pretty straightforward. Just return a pointer to the appropriate element of your instance data.

```
   static int Set( BoxData *box, unsigned long vid, void *value )
   {
      const char *a;
      double *d;
      int i;
      switch ( vid )
      {
         case ID_SIZE:
            d = ( double * ) value;
            sbox.size[ 0 ] = box->size[ 0 ] = d[ 0 ];
            sbox.size[ 1 ] = box->size[ 1 ] = d[ 1 ];
            sbox.size[ 2 ] = box->size[ 2 ] = d[ 2 ];
            break;

         case ID_CENTER:
            ...
```

`Set` adds a few wrinkles. The first is that you generally need to cast the value argument before assigning its contents to your instance data, so it's convenient to have temporary pointers of the right type handy. The second, for us, is that we'd like to keep a local copy of the instance, so that we can use it to initialize the tool instance the next time the user activates the tool. The user's perception of this is that the tool "remembers" what was done previously. So all of our assignments are duplicated for the local copy.

```
         default:
            return LWXPRC_NONE;
      }

      box->update = LWT_TEST_UPDATE;
      box->dirty = 1;
      calc_handles( box );
      return LWXPRC_DRAW;
   }
```

Lastly, when the value of a control changes, we want to tell Modeler to redraw and rebuild us the next time it calls `Dirty` and `Test`, so we set `box->update` and `box->dirty` accordingly and precalculate the positions of our handles.

The Activation Function

Our activation function is significantly different from the ones in previous installments of this tutorial. Instead of being finished when the function returns, tool plug-ins haven't really begun yet. The only thing a tool's activation function is required to do, and all ours does, is create an instance and tell Modeler where to find the callbacks. In this sense, Modeler tools are like Layout handlers, which supply callbacks that Layout later calls during animation and rendering.

```
   XCALL_( int )
   Activate( long version, GlobalFunc *global, LWMeshEditTool *local,
      void *serverData )
   {
      BoxData *box;

      if ( version != LWMESHEDITTOOL_VERSION )
         return AFUNC_BADVERSION;
```

Note that the third argument is now LWMeshEditTool instead of LWModCommand. Each plug-in class gets its own local data. As always, the first thing we do is ensure that the version of this structure in our copy of the headers is the same as the version being passed to us by Modeler.

```
      if ( !get_xpanf( global )) return AFUNC_BADGLOBAL;
      box = new_box();
      if ( !box ) return AFUNC_OK;
      local->instance = box;
```

The `get_xpanf` and `new_box` functions are in `ui.c`, since that's where the LWXPanelFuncs and LWSurfaceFuncs pointers and the local copy of the box settings are stored and used. `get_xpanf` gets the globals used by the interface, and `new_box` allocates a BoxData and initializes it with default values (or values remembered from previous invocations). The BoxData will be freed when `Done` is called.

```
      local->tool->done   = Done;
      local->tool->help   = Help;
      local->tool->count  = Count;
      local->tool->handle = Handle;
      local->tool->adjust = Adjust;
      local->tool->start  = Start;
      local->tool->draw   = Draw;
      local->tool->dirty  = Dirty;
      local->tool->event  = Event;
      local->tool->panel  = Panel;

      local->build        = Build;
      local->test         = Test;
      local->end          = End;

      return AFUNC_OK;
   }
```

And we're done! After returning from the activation function, Modeler will start calling your callbacks through the function pointers you've supplied.

Server Tags

Finally, note that we've added server tags to the ServerRecord array.

```
static ServerTagInfo srvtag[] = {
   { "Tutorial: Box 4",    SRVTAG_USERNAME | LANGID_USENGLISH },
   { "create",             SRVTAG_CMDGROUP },
   { "objects/primitives", SRVTAG_MENU },
   { "Tut Box 4",          SRVTAG_BUTTONNAME },
   { "", 0 }
};
```

These are explained in detail on the Common Elements page of the SDK. The user name appears in the interface in plug-in lists and popup menus. The server name is used if this isn't supplied, but there are lexical constraints on server names (they can't contain spaces, for example) that make them less user-friendly. Modeler is currently ignoring the `MENU` and `CMDGROUP` tags, but it may not in the future.

What's Next

Unless you had the evidence in front of you, you might not believe that a 40-page tutorial about writing box plug-ins was possible. But on this thin pretext, we've briefly visited most of the important techniques used to write plug-ins for LightWave Modeler. In the future, we might be seeing even more boxes on a similar tour of Layout...

AnimLoaderHandler
AnimLoaderInterface

Availability LightWave 6.0
Component Layout, Modeler
Header lwanimlod.h

An animation loader loads frames from an animation file. An animation file is a file that contains a time sequence of still images, or data that can be interpreted that way. Animation loaders must be prepared to provide random access to the frames in an animation file. They must also be able to distinguish between files they can load and those they can't. LightWave relies on this to choose the proper loader for an animation file.

Handler Activation Function

```
   XCALL_( int ) MyAnimLoader( long version, GlobalFunc *global,
      LWAnimLoaderHandler *local, void *serverData );
```

The `local` argument to an anim loader's activation function is an LWAnimLoaderHandler.

```
   typedef struct st_LWAnimLoaderHandler {
      LWInstanceFuncs *inst;
      int              (*frameCount) (LWInstance);
      double           (*frameRate)  (LWInstance);
      double           (*aspect)     (LWInstance, int *w, int *h,
                                        double *pixAspect);
      void             (*evaluate)   (LWInstance, double,
                                        LWAnimFrameAccess *);
   } LWAnimLoaderHandler;
```

The first member of this structure points to the standard instance handler functions. An anim loader also provides functions that return image pixels and other information from the file.

The `context` argument to the `create` function is the filename. The `create` function should open the file and determine whether it's in a format the loader can load. If the format isn't recognized, `create` should return NULL, without setting the error string. LightWave will understand this to mean that the file should be handled by a different anim loader.

```
count = frameCount( instance )
```

Returns the number of frames in the file.

`fps = `**`frameRate`**`( instance )`

Returns the animation's playback speed, in frames per second.

frame_aspect = **aspect**(instance, w, h, pixel_aspect)

Fills in the width and height of the frames and the pixel aspect ratio and returns the frame aspect ratio. The *aspect ratio* of a rectangle describes its shape--whether it's short and broad, tall and thin, or square--and is expressed as width / height. The aspect ratios of each pixel and of the image as a whole are related in the following way.

frame_aspect = w * pixel_aspect / h
pixel_aspect = h * frame_aspect / w

The `aspect` function therefore asks for redundant information, so to ensure that what you're returning is self-consistent, you should calculate one of the aspects in terms of the other.

`evaluate``( instance, time, access )`

Load an image at the specified running time from the file. At the loader's discretion, the image can be the frame nearest to the time or an interpolation of two or more frames. The access structure, described below, provides the functions the loader uses to pass image data to Layout.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. LightWave currently doesn't give the user an interface for animation loaders, although it may in a future version.

Anim Frame Access

The access structure passed to the loader's evaluation function uses the image I/O mechanism to pass image data to Layout.

```
   typedef struct st_LWAnimFrameAccess {
      void              *priv_data;
      LWImageProtocolID (*begin) (void *, int type);
      void              (*done)  (void *, LWImageProtocolID);
   } LWAnimFrameAccess;
```

`priv_data`

An opaque pointer to data used by Layout. Pass this to `begin` and `done`.

`protocol = `**`begin`**`( priv_data, type )`

Call this to tell Layout that you're about to send it image data for a frame. The type argument describes the kind of pixel data you'll send and can be any of the image I/O pixel types. Layout returns an LWImageProtocolID containing the functions used to pass the data.

`done``( priv_data, protocol )`

Call this to tell Layout that you've finished sending the image.

Example

The ancounter sample is a simple animation loader that draws its frames on the fly based on a small amount of information in a text file. The text file is the "animation file" the user selects in order to invoke this loader.

Every animation file passes through every anim loader's `create` function until one of the loaders claims the file as its own. AnCounter reads the first line of each file and compares it to a phrase that identifies the file as an AnCounter text file. If the phrase isn't present at the start of the file, the `create` function quietly fails by returning NULL.

```
   fp = fopen( filename, "r" );
   if ( !fp ) {
      *emsg = "Couldn't open anim file.";
      return NULL;
   }

   fgets( buf, sizeof( buf ), fp );

   if ( strncmp( buf, "Counter AnimLoader File", 23 )) {
      fclose( fp );
      return NULL;
   }
```

If the phrase is there, `create` allocates an instance structure and initializes it using the information in the file. The evaluation function later uses this information to make a "counter" image. A string of the form "hh:mm:ss:ff" (hours, minutes, seconds, frames) is made from the time argument, and this is rasterized, using the font information in the text file, and passed to LightWave as the current frame.

```
if ( !getTextImage( counter, text )) return;

ip = access->begin( access->priv_data, LWIMTYP_GREY8 );
if ( !ip ) return;

LWIP_SETSIZE( ip, counter->w, counter->h );
LWIP_SETPARAM( ip, LWIMPAR_ASPECT, 0, 1.0f );

for ( i = 0; i < counter->h; i++ ) {
   result = LWIP_SENDLINE( ip, i, counter->buf + i * counter->w );
   if ( result != IPSTAT_OK ) break;
}

LWIP_DONE( ip, result );
access->done( access->priv_data, ip );
```

AnimSaverHandler
AnimSaverInterface

Availability LightWave 6.0
Component Layout
Header lwanimsav.h

Animation savers write out a sequence of rendered images as an animation file. Anim savers add frames to the animation file as each frame is rendered. The rendered image is passed to the saver in the same way it's passed to frame buffer display plug-ins, except that animation savers are given a filename, and there is no pause after each frame.

Handler Activation Function

```
XCALL_( int ) MyAnimSaver( long version, GlobalFunc *global,
   LWAnimSaverHandler *local, void *serverData );
```

The `local` argument to an anim saver's activation function is an LWAnimSaverHandler.

```
typedef struct st_LWAnimSaverHandler {
   LWInstanceFuncs *inst;
   LWItemFuncs     *item;
   int              type;
   LWError         (*open)  (LWInstance, int w, int h,
                              const char *filename);
   void            (*close) (LWInstance);
   LWError         (*begin) (LWInstance);
   LWError         (*write) (LWInstance, const void *R, const void *G,
                              const void *B, const void *alpha);
} LWAnimSaverHandler;
```

The first two member of this structure point to the standard handler functions. In addition to these, an anim loader also provides functions for opening and closing the file and for writing a frame, and it specifies what type of data it wants to receive. The `context` argument to the `inst->create` function is currently unused.

`type`
: The type of pixel data Layout should send to the `write` function. This can be either `LWAST_UBYTE` or `LWAST_FLOAT`.

`error = open( instance, width, height, filename )`

Open the file. This function receives the width and height of the frame in pixels, and the name of the file. Called when a rendering session begins. Returns an error message string if an error occurs, otherwise it returns NULL.

`close( instance )`

Close the file. This is called when rendering is complete.

`error = begin( instance )`

Prepare to receive the next frame. Returns an error message string if an error occurs, otherwise it returns NULL.

`error = write( instance, R, G, B, alpha )`

Write the next scanline of the current frame. The scanlines for each frame are sent in order from top to bottom. The buffer arguments point to arrays of color channel values. There are exactly `width` values for each channel, one for each pixel in a scanline, and the values are either unsigned bytes or floats, depending on the `type` code. Returns an error message string or NULL.

You'll need a way to know when you can write each frame. You can initialize a scanline index to 0 in your `begin` and then increment it in `write` until you've received the last scanline. Or you can save the last completed frame in `begin` (save frame 1 when `begin` is called for frame 2, and so on) and save the last frame in `close`.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. The saver's interface is invoked by Layout when the user selects the saver from the anim saver list on the Render panel.

Example

The SDK avisave sample is an anim saver for Windows AVI files.

ChannelHandler

Availability LightWave 6.0
Component Layout, Modeler
Header lwchannel.h

A channel is a value that can vary continuously with time. Channels are everywhere in LightWave. Any animation parameter that can be enveloped is associated with an underlying channel. Channel handlers dip into the stream of a parameter and alter its value.

Handler Activation Function

```
   XCALL_( int ) MyChannel( long version, GlobalFunc *global,
      LWChannelHandler *local, void *serverData );
```

The `local` argument to a channel handler's activation function is an LWChannelHandler.

```
   typedef struct st_LWChannelHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      void            (*evaluate) (LWInstance, const LWChannelAccess *);
      unsigned int    (*flags)    (LWInstance);
   } LWChannelHandler;
```

The first two members of this structure are standard instance handler functions. The `context` argument to the `create` function is the LWChannelID of the associated channel. *When the plug-in is activated by Modeler, the `item` member of the LWChannelHandler will be NULL.* Check for this before assigning the `item` callbacks.

A channel handler also provides an evaluation function and a flags function.

evaluate(instance, access)
: The channel value is examined and modified at each time step using functions in the channel access structure, described below.

f = **flags**(instance)
: Returns an integer containing bit flags combined using bitwise-or. No flags are currently defined for channel handlers, so this should return

0.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. Channel handlers are selected on the graph editor panel, and their non-modal interfaces will be drawn there.

Channel Access

This is the structure passed to the handler's evaluation function.

```
   typedef struct st_LWChannelAccess {
      LWChannelID   chan;
      LWFrame       frame;
      LWTime        time;
      double        value;
      void         (*getChannel)  (LWChannelID chan, LWTime t,
                                     double *value);
      void         (*setChannel)  (LWChannelID chan, const double value);
      const char * (*channelName) (LWChannelID chan);
   } LWChannelAccess;
```

`chan`
    The channel ID.

`frame`
    The frame number of the evaluation.

`time`
    The time of the evaluation, in seconds.

`value`
    The current value of the channel at the given time.

`getChannel``( channel, time, value )`
    Retrieves a value from a channel.

`setChannel``( channel, value )`
    Sets the value of the channel.

`name = `**`channelName`**`( channel )`
    Returns the name associated with the channel ID.

Example

Several of the SDK samples are channel handlers. NoisyChan uses the texture system's noise function to modify a channel. A channel handler is one of four classes demonstrated in txchan, which also uses textures as channel modifiers. xpanchan is a channel handler that demonstrates four ways of displaying the same XPanels interface.

ColorPicker

Availability LightWave 6.0
Component Layout, Modeler
Header lwdialog.h

Color pickers prompt the user for a color selection. They replace the operating system's default color selection mechanism, or provide one if no default exists.

See the Color Picker global for a discussion of color pick requests from the host's point of view.

Activation Function

```
   XCALL_( int ) MyColorPick( long version, GlobalFunc *global,
      LWColorPickLocal *local, void *serverData );
```

The `local` argument to a color picker's activation function is an LWColorPickLocal.

```
   typedef void LWHotColorFunc( void *data, float r, float g, float b );

   typedef struct st_LWColorPickLocal {
      int             result;
      const char     *title;
      float           red, green, blue;
      void           *data;
      LWHotColorFunc *hotFunc;
   } LWColorPickLocal;
```

`result`
: The result of the request. Set this to 1 if the user selects a color, 0 if the user cancels the request, and a negative number to indicate an error.

`title`
: The title string. This is generally displayed near the top of the color dialog and tells the user the context of the color request.

`red`, `green`, `blue`
: The initial color. If the user selects a color, the selected color should be written in these fields. The nominal range for RGB levels is 0.0 to 1.0, but they can be outside this range.

`data`

A pointer to data that should be passed to the caller's color callback.

`hotFunc``( data, r, g, b )`

A color callback supplied by the host. Call this while the user is playing with any of your color selection mechanisms. This allows the host to update its display interactively as the user selects a color. (The "hot" part of the name refers to this dynamic interaction. This isn't a video color gamut test.)

Example

The simpcolr SDK sample is a simple example of a color picker.

CommandSequence

Availability LightWave 6.0
Component Modeler
Header lwcmdseq.h

Command sequence plug-ins issue commands to create and manipulate geometry in Modeler. They also have access to the same mesh editing functions as the MeshDataEdit class.

Activation Function

```
XCALL_( int ) MyCmdSeq( long version, GlobalFunc *global,
   LWModCommand *local, void *serverData );
```

The `local` argument to a command sequence's activation function is an LWModCommand.

```
typedef struct st_LWModCommand {
   void          *data;
   const char    *argument;
   LWCommandCode (*lookup)   (void *, const char *cmdName);
   int           (*execute)  (void *, LWCommandCode cmd, int argc,
                                const DynaValue *argv,
                                EltOpSelect, DynaValue *result);
   MeshEditBegin *editBegin;
   int           (*evaluate) (void *, const char *command);
} LWModCommand;
```

data

An opaque pointer to data used internally by Modeler. Pass this as the first argument to the `lookup`, `execute` and `evaluate` functions.

argument

Users and other plug-ins can invoke your plug-in with arguments, which are stored here as a string.

cmdcode = **lookup**(data, cmdname)

Returns an integer code corresponding to the command name. The command is issued by passing the command code to the `execute` function. Command codes are constant for a given Modeler session, so this only needs to be called once per command, after which the codes can be cached and then used in any number of calls to `execute`.

result = **execute**(data, cmdcode, argc, argv, selection, cmdresult)

Issue the command given by the command code argument. `argv` is an array of DynaValue arguments. `argc` is the number of arguments in the `argv` array. The selection determines which geometry will be affected by the command and can be any one of the EltOpSelect codes except `OPSEL_MODIFY`. The result of the command is written in `cmdresult`. The function returns `CSERR_NONE` (0) if it succeeds or one of the following non-zero error codes.

CSERR_MEMORY
CSERR_IO
CSERR_USERABORT
CSERR_ARGCOUNT
CSERR_ARGTYPE
CSERR_ARGVALUE
CSERR_OPFAILURE
CSERR_BADSEL

edit = **editBegin**(pnt_bufsize, pol_bufsize, opsel)

Begin a mesh edit. The buffer sizes are used to create temporary buffers associated with each point and polygon. Modeler allocates and frees this memory for you, and you can use it for any per-point or per-polygon data you might need during the edit. Points and polygons are flagged as selected according to the code you pass in `opsel`.

The `editBegin` function is identical to the function passed as the local data to mesh edit plug-ins. See that page for complete documentation of the MeshEditOp structure it returns. Command sequence plug-ins can perform multiple mesh edits. Each edit begins by calling this function to get a MeshEditOp and ends when the MeshEditOp's `done` function is called. No commands can be issued during a mesh edit.

result = **evaluate**(data, cmdstring)

Issue the command with the name and arguments in the command string. This is an alternative to using `lookup` and `execute`. The command and its arguments are written to a single string and delimited by spaces.

See the Commands pages for a complete list of the commands that can be issued in Modeler, as well as a detailed explanation of the formatting of command arguments for both the `execute` and `evaluate` methods.

Example

The DNA sample is a CommandSequence plug-in that builds classic Watson-Crick DNA molecules. It uses the ModLib static-link library, which greatly simplifies command execution by translating commands into function calls. The library currently contains about 170 functions that cover Modeler commands, mesh edit functions, and globals.

This ModLib function executes the `MAKEBALL` command, building the DynaValue argument list and calling the `lookup` and `execute` functions. (ModData is a ModLib structure that caches the LWModCommand pointer and the data returned from a number of globals.)

```
int csMakeBall( double *radius, int nsides, int nsegments,
   double *center )
{
   static LWCommandCode ccode;
   ModData *md = getModData();
   DynaValue argv[ 4 ];

   assert( md->edit == NULL );

   argv[ 0 ].type = DY_VFLOAT;
   argv[ 0 ].fvec.val[ 0 ] = radius[ 0 ];
   argv[ 0 ].fvec.val[ 1 ] = radius[ 1 ];
   argv[ 0 ].fvec.val[ 2 ] = radius[ 2 ];

   argv[ 1 ].type = DY_INTEGER;
   argv[ 1 ].intv.value = nsides;

   argv[ 2 ].type = DY_INTEGER;
   argv[ 2 ].intv.value = nsegments;

   if ( center ) {
      argv[ 3 ].type = DY_VFLOAT;
      argv[ 3 ].fvec.val[ 0 ] = center[ 0 ];
      argv[ 3 ].fvec.val[ 1 ] = center[ 1 ];
      argv[ 3 ].fvec.val[ 2 ] = center[ 2 ];
   }
   else argv[ 3 ].type = DY_NULL;

   if ( !ccode )
      ccode = md->local->lookup( md->local->data, "MAKEBALL" );

   md->cmderror = md->local->execute( md->local->data, ccode,
      4, argv, md->opsel, &md->result );

   return md->cmderror == CSERR_NONE;
}
```

Using ModLib makes DNA's command processing almost as simple as scripting. Below is a code fragment from the function in the DNA plug-in that creates the cylinders representing atomic bonds.

```
   csSetLayer( layer2 );
   csSetDefaultSurface( surface_name( snum ));
   csMakeDisc( r, h, 0, "Y", bond_nsides, bond_nsegments, c );
   csRotate( xrot, "X", NULL );
   csRotate( yrot, "Y", NULL );
   csMove( pt );
   rot = 36 * j;
   csRotate( rot, "Y", NULL );
   csCut();
   csSetLayer( layer1 );
   csPaste();
```

CustomObjHandler
CustomObjInterface

Availability LightWave 6.0

Component Layout
Header lwcustobj.h

Layout uses null objects as placeholders for animation data. Nulls can be used as parents to add degrees of freedom to the motion of other objects, or as references for texturing, or as camera targets. Plug-ins can also rely on nulls as a way for users to interactively set parameters.

A custom object handler can be associated with a null to customize its appearance in Layout's interface. This is useful for providing visual feedback about, for example, the range or magnitude of an effect controlled by the null. Custom nulls will often be an adjunct to a plug-in of another class that uses nulls for data entry, but they can also be used by themselves for things like guides and rulers.

When applied to non-null objects, a custom object plug-in supplements LightWave's drawing of the object in the interface. Hypervoxels, for example, uses a custom object handler to draw wireframe bounding spheres around the particles associated with an object.

Handler Activation Function

```
XCALL_( int ) MyCustomObj( long version, GlobalFunc *global,
   LWCustomObjHandler *local, void *serverData );
```

The `local` argument to a custom object's activation function is an LWCustomObjHandler.

```
typedef struct st_LWCustomObjHandler {
   LWInstanceFuncs *inst;
   LWItemFuncs     *item;
   LWRenderFuncs   *rend;
   void            (*evaluate)(LWInstance, const LWCustomObjAccess *);
   unsigned int    (*flags)   (LWInstance);
} LWCustomObjHandler;
```

The first three members of this structure are the standard handler functions. In addition to these, a custom object provides an evaluation function and a flags function.

The `context` argument to the `inst->create` function is the LWItemID of the associated object.

evaluate`( instance, access )`

Draw the object on the interface using the information in the access structure, described below.

`f =` **flags**`( instance )`

Returns bit flags combined using bitwise-or.

`LWCOF_SCHEMA_OK`

Tells Layout that you support drawing in schematic viewports.

`LWCOF_VIEWPORT_INDEX`

Tells layout to use the viewport number instead of its type in the LWCustomObjAccess view element

`LWCOF_NO_DEPTH_BUFFER`

Causes drawing to occur in front of other OpenGL elements, regardless of Z position.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. Users open a custom object's interface by pressing an Options button on the Geometry tab of the Object Properties panel.

Custom Object Access

The access structure contains drawing functions and fields indicating which of the interface views the object will be drawn in and whether the object is currently selected.

Within the limitations of the drawing functions, there aren't any restrictions on what your custom object may look like. But in most cases it will be helpful to users if your object's appearance is consistent in color and style with the rest of Layout's interface.

```
   typedef struct st_LWCustomObjAccess {
      int   view;
      int   flags;
      void *dispData;
      void (*setColor)   (void *, float rgba[4]);
      void (*setPattern) (void *, int lpat);
      void (*setTexture) (void *, int, unsigned char *);
      void (*setUVs)     (void *, double[2], double[2], double[2],
                            double[2]);
      void (*point)      (void *, double[3], int csys);
      void (*line)       (void *, double[3], double[3], int csys);
      void (*triangle)   (void *, double[3], double[3], double[3],
                            int csys);
      void (*quad)       (void *, double[3], double[3], double[3],
                            double[3], int csys);
      void (*circle)     (void *, double[3], double, int csys);
      void (*text)       (void *, double[3], const char *, int just,
                            int csys);
      LWDVector viewPos, viewDir;
   } LWCustomObjAccess;
```

view

The view the object will be drawn in. Possible values are

`LWVIEW_ZY`

`LWVIEW_XZ`

`LWVIEW_XY`

`LWVIEW_PERSP`

`LWVIEW_LIGHT`

`LWVIEW_CAMERA`
`LWVIEW_SCHEMA`

These refer to the orthographic, perspective, light, camera and schematic views available to the user in the Layout interface.

flags

Contains bitfields with information about the context of the render request. Currently the only flag defined is `LWCOFL_SELECTED`, which tells you whether the object should be rendered in its selected state.

dispData

An opaque pointer to private data used by Layout. Pass this as the first argument to the drawing functions.

setColor`( dispData, rgba )`

Set the current drawing color, including the alpha level. Calling this is optional. By default, all drawing is done in the color set by the user in the Scene panel when the custom object isn't selected, and in yellow when the object is selected. Color settings don't persist between calls to the evaluation function, nor do they change the settings in the Scene panel.

setPattern(dispData, linepat)

Set the current line pattern. The pattern codes are

LWLPAT_SOLID

LWLPAT_DOT

LWLPAT_DASH
LWLPAT_LONGDOT

As with `setColor`, calling `setPattern` is optional. By default, all drawing is done with solid lines. Line pattern settings don't persist between evaluations.

setTexture(dispData, size, imagebytes)

Set the current image for texture mapping. This image is mapped onto quads drawn by the `quad` function. The `size` is the width (and height) of the image in pixels and must be a power of 2. The pixel data is an OpenGL image in `GL_RGBA` format and `GL_UNSIGNED_BYTE` data type. Each pixel is represented by a set of four contiguous bytes containing red, green, blue and alpha values ranging from 0 to 255.

setUVs(dispData, uv0, uv1, uv2, uv3)

Set the UVs for texture mapping. This sets the position of the texture image on each polygon drawn by the `quad` function until the next call to `setUVs`.

point(dispData, xyz, coord_sys)

Draw a point at the specified position. The point will be drawn in the color set by the most recent `setColor` call, or in the default color if no color was set. The coordinate system argument identifies the coordinates in which the position is expressed and may be one of the following.

LWCSYS_WORLD

"Absolute" coordinates, unaffected by the position, rotation and scale of the underlying null object.

LWCSYS_OBJECT

"Relative" coordinates. Layout will transform the point.

`LWCSYS_ICON`
: A special coordinate system that works like `LWCSYS_OBJECT` but scales with the grid size. Layout's camera and light images are examples of the use of this mode.

line`( dispData, end1, end2, coord_sys )`
: Draw a line between the specified endpoints using the current color and line pattern.

triangle`( dispData, v1, v2, v3, coord_sys )`
: Draw a solid triangle with the specified vertices using the current color.

quad`( dispData, v1, v2, v3, v4, coord_sys )`
: Draw a solid quadrangle with the specified vertices using the current color or texture.

circle`( dispData, center, radius, coord_sys )`
: Draw a circle of the given radius around the specified center point using the current color and line pattern. Circles can only be drawn in the orthographic views, not in the light, camera or perspective views.

text`( dispData, pos, textline, justify, coord_sys )`
: Draw a single line of text using the current color and line pattern. The `justify` argument determines whether the text will be drawn to the left or right of the position, or centered on it:

```
LWJUST_LEFT
LWJUST_CENTER
LWJUST_RIGHT
```

History

In LightWave 7.0, `LWCUSTOMOBJ_VERSION` was incremented to 5 because of significant changes to the LWCustomObjAccess structure. The previous version of the structure looked like this.

```
typedef struct st_LWCustomObjAccess_V4 {
    int   view;
    int   flags;
    void *dispData;
    void (*setColor)   (void *, float rgb[3]);
    void (*setPattern) (void *, int lpat);
    void (*point)      (void *, double[3], int csys);
    void (*line)       (void *, double[3], double[3], int csys);
    void (*triangle)   (void *, double[3], double[3], double[3],
                          int csys);
    void (*circle)     (void *, double[3], double, int csys);
```

```
      void (*text)       (void *, double[3], const char *, int csys);
   } LWCustomObjAccess_V4;
```

The `setTexture`, `setUVs` and `quad` functions are missing, and the `text` function lacks the justification argument.

Example

The barn sample draws a simple barn or house shape. It's easy to tell which way this shape is pointing, which makes it useful for quickly testing programming assumptions about the effects of animation parameters on the orientation of objects.

DisplacementHandler
DisplacementInterface

Availability LightWave 6.0
Component Layout
Header lwdisplce.h

Displacement plug-ins deform objects by moving their points at each time step.

Handler Activation Function

```
   XCALL_( int ) MyDisplacement( long version, GlobalFunc *global,
      LWDisplacementHandler *local, void *serverData );
```

The `local` argument to a displacement plug-in's activation function is an LWDisplacementHandler.

```
   typedef struct st_LWDisplacementHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      LWRenderFuncs   *rend;
      void            (*evaluate) (LWInstance, LWDisplacementAccess *);
      unsigned int    (*flags)    (LWInstance);
   } LWDisplacementHandler;
```

The first three members of this structure are the standard handler functions. The `context` argument to the `inst->create` function is the LWItemID of the object associated with this instance.

In addition to the standard functions, a displacement plug-in provides an evaluation function and a flags function.

`evaluate( instance, access )`

This is where the displacement happens. At each time step, the evaluation function is called for each vertex in the object. The position of the vertex is examined and modified through the access structure described below.

`f = flags( instance )`

Returns bit flags combined using bitwise-or. The flags tell Layout whether the displacement will be in world coordinates and whether it should occur before or after the object has been deformed by bones. Only one of these flags should be set.

LWDMF_WORLD
LWDMF_BEFOREBONES

Interface Activation Function

```
XCALL_( int ) MyInterface( long version, GlobalFunc *global,
   LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Displacement Access

The LWDisplacementAccess passed to the evaluation function contains two point positions, the point ID, and a mesh info for the object the point belongs to.

```
typedef struct st_LWDisplacementAccess {
   LWDVector    oPos;
   LWDVector    source;
   LWPntID      point;
   LWMeshInfo  *info;
} LWDisplacementAccess;
```

oPos
The original point location in object coordinates. This is read-only.

source
The location to be transformed in place by the displacement. If the flags function returned the LWDMF_WORLD bit, the source is in world coordinates and has already been modified by morphing, bones and object motion. Otherwise the source is in object coordinates (after morphing, before item motion, and before or after bone effects, depending on whether the flags function returned LWDMF_BEFOREBONES).

point
The point ID. This can be used to retrieve other information about the point from the mesh info structure.

info
A mesh info structure for the object.

History

In LightWave 7.0, `LWDISPLACEMENT_VERSION` was incremented to 5. This reflects additions to the LWMeshInfo structure, but in all other respects, displacement handlers were unchanged.

Example

The inertia sample is a displacement handler that causes points to "lag behind" as the object moves. This plug-in was formerly known as LazyPoints.

EnvironmentHandler
EnvironmentInterface

Availability LightWave 6.0
Component Layout
Header lwenviron.h

Environment handlers render the backdrop, the points at infinity that aren't covered by anything in the scene. This is the natural place to draw the sky, the distant horizon, or a procedural nebula in space.

Handler Activation Function

```
   XCALL_( int ) MyEnvironment( long version, GlobalFunc *global,
      LWEnvironmentHandler *local, void *serverData );
```

The `local` argument to an environment handler's activation function is an LWEnvironmentHandler.

```
   typedef struct st_LWEnvironmentHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      LWRenderFuncs   *rend;
      LWError         (*evaluate) (LWInstance, LWEnvironmentAccess *);
      unsigned int    (*flags) (LWInstance);
   } LWEnvironmentHandler;
```

The first three members of this structure point to the standard handler functions. In addition to these, an environment handler provides an evaluation function and a flags function.

`errmsg = `**`evaluate`**`( instance, access )`

> This is where the environment handler does its work. At each time step in the animation, the evaluation function is called for each affected pixel in the image. The access argument, described below, contains information about the environment to be colored.

`f = `**`flags`**`( instance )`

> Returns flag bits combined using bitwise-or.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. An environment's non-modal xpanel interface is drawn on the Backdrop tab of the Effects panel.

Environment Access

This is the structure passed to the handler's evaluation function.

```
   typedef struct st_LWEnvironmentAccess {
      LWEnvironmentMode mode;
      double            h[2], p[2];
      LWDVector         dir;
      double            colRect[4][3];
      double            color[3];
   } LWEnvironmentAccess;
```

`mode`
: The context of the evaluation. Currently this distinguishes between rendering (`EHMODE_REAL`) and lower quality previewing (`EHMODE_PREVIEW`).

`h, p`
: The heading and pitch extents of a rectangular area of the backdrop. They're both expressed in radians. In preview mode, these form a bounding box in spherical coordinates of an area to be colored. They should be ignored in real mode.

dir

A vector pointing toward a point on the backdrop to be colored. Use this when evaluating in real mode.

`colRect`
: In preview mode, this is where the evaluation function returns the color at the corners of the rectangular area defined by `h` and `p`. The preview display interpolates between these at points inside the rectangle.

`colRect[0]` gets the color of `(h[0], p[0])`
`colRect[1]` gets the color of `(h[0], p[1])`
`colRect[2]` gets the color of `(h[1], p[0])`
`colRect[3]` gets the color of `(h[1], p[1])`

`color`
: In real mode, this is where the evaluation function returns the color of

the point defined by the direction vector `dir`.

Example

The horizon sample duplicates Layout's gradient backdrop settings. It can also produce a grid backdrop. Be sure to look for the haiku in the `newTime` function.

The following code can be used to draw a simple representation of the sky and ground that includes a disk for the sun.

Drawing the sun requires knowing whether a point on the backdrop is inside the sun's disk. The point's angular separation from the center of the sun must be less than the sun's angular radius. So we need a function for calculating the angular separation between two spherical coordinates.

```
static double angsep( double h1, double p1, double h2, double p2 )
{
   double cd;

   if ( h1 == h2 && p1 == p2 )
      return 0.0;

   cd = cos( p1 ) * cos( p2 ) * cos( fabs( h2 - h1 ))
      + sin( p1 ) * sin( p2 );

   /* catch small roundoff errors */

   if ( cd >  1.0 ) cd =  1.0;
   if ( cd < -1.0 ) cd = -1.0;

   return acos( cd );
}
```

We'd also like to write a sampling function that uses the same representation for the point being sampled, regardless of whether it's called in preview or real mode, so we need to be able to convert the direction vector into spherical (heading and pitch) coordinates. If the vector points straight up or down, the heading is undefined, so we set it arbitrarily to 0. To avoid problems with roundoff, we say that the vector is straight up or down if the magnitude of the y component is within some epsilon of 1.0.

```
static void vec2hp( LWDVector n, double *h, double *p )
{
   *p = asin( -n[ 1 ] );

   if ( 1.0 - fabs( n[ 1 ] ) > 1e-5 ) {
      /* not straight up or down */
      *h = acos( n[ 2 ] / cos( *p ));
      if ( n[ 0 ] < 0.0 ) *h = 2 * PI - *h;
   }
   else *h = 0;
}
```

The sampling function decides whether the point is in the sky, the ground, or the sun, and colors the point accordingly. If the backdrop point is below the horizon, the ground color is used. If both the point and the sun are above the horizon, the point is compared to the sun's position to decide whether the sun or the sky color is used. `hsun` and `psun` are the heading and pitch of the sun's position. In preview mode, the Manhattan distance between the point and the sun's center is sufficient, while in real mode we do the more expensive angular separation calculation.

```
static void sample( MyInstance *inst, double h, double p,
   double *color, int mode )
{
   int insun;

   if ( p >= 0.0 ) {
      color[ 0 ] = inst->gndcolor[ 0 ];
      color[ 1 ] = inst->gndcolor[ 1 ];
      color[ 2 ] = inst->gndcolor[ 2 ];
      return;
   }

   insun = inst->psun - inst->sunrad < 0.0;
   if ( insun ) {
      if ( mode == EHMODE_PREVIEW )
         insun =  ( fabs( h - inst->hsun ) < inst->sunrad )
               && ( fabs( p - inst->psun ) < inst->sunrad );
      else
         insun = angsep( h, p, inst->hsun, inst->psun )
               < inst->sunrad;
   }

   if ( insun ) {
      color[ 0 ] = inst->suncolor[ 0 ];
      color[ 1 ] = inst->suncolor[ 1 ];
      color[ 2 ] = inst->suncolor[ 2 ];
   }
   else {
      color[ 0 ] = inst->skycolor[ 0 ];
      color[ 1 ] = inst->skycolor[ 1 ];
      color[ 2 ] = inst->skycolor[ 2 ];
   }
}
```

The evaluation function uses the sampling function to find the right color and returns the color to the renderer.

```
XCALL_( static LWError )
Evaluate( MyInstance *inst, LWEnvironmentAccess *access )
{
   double h, p;
   switch ( access->mode )
   {
      case EHMODE_PREVIEW:
         sample( inst, access->h[ 0 ], access->p[ 0 ],
            access->colRect[ 0 ], access->mode );
         sample( inst, access->h[ 0 ], access->p[ 1 ],
            access->colRect[ 1 ], access->mode );
         sample( inst, access->h[ 1 ], access->p[ 0 ],
            access->colRect[ 2 ], access->mode );
         sample( inst, access->h[ 1 ], access->p[ 1 ],
            access->colRect[ 3 ], access->mode );
      break;

      case EHMODE_REAL:
         vec2hp( access->dir, &h, &p );
         sample( inst, h, p, access->color, access->mode );
         break;

      default:
         break;
   }

   return NULL;
}
```

FileRequester

Availability LightWave 6.0
Component Layout, Modeler
Header lwdialog.h

File request plug-ins prompt the user for a file selection. At a minimum, they should provide the same functionality as the operating system's default file dialog, allowing users to browse their file systems to select or enter a file name.

See the File Request 2 global for a discussion of file requests from the host's point of view.

Handler Activation Function

```
   XCALL_( int ) MyFileReq( long version, GlobalFunc *global,
      LWFileReqLocal *local, void *serverData );
```

The `local` argument to a file request plug-in's activation function is an LWFileReqLocal.

```
   typedef struct st_LWFileReqLocal {
      int         reqType;
      int         result;
      const char *title;
      const char *fileType;
      char       *path;
      char       *baseName;
      char       *fullName;
      int         bufLen;
      int        (*pickName) (void);
   } LWFileReqLocal;
```

`reqType`

Indicates the type of file request. Possible values are

FREQ_LOAD
FREQ_SAVE
FREQ_DIRECTORY
A request for a path.
FREQ_MULTILOAD
A request for one or more files to load.

`result`

The result of the request. Set this to 1 if the user selects a file, 0 if the user cancels the request, and a negative number to indicate an error.

`title`

The title string. This is generally displayed near the top of the file dialog and tells the user what kind of file is being requested.

`fileType`

A string identifying a file type filter. This should be used to filter the names displayed in the dialog. The string will generally contain one of the file type names used in LightWave's configuration files, rather than a literal, platform-specific list of type IDs or wildcards. See the File Type Pattern global for more information about what the file type string might contain. You can use the global to translate this into a literal filter string.

`path`

The initial path on entry. This is where browsing of the file system should begin. The initial path can be either absolute (fully qualified) or relative to the operating system's current default path, also sometimes called the current working directory. If the user completes the file request, the plug-in should write the fully qualified path of the selected file in this field.

If the operation of the file request plug-in changes the current working directory, this should be restored before the file request is completed.

`baseName`

The initial file name, not including the path. This may be empty, or it may contain a default name. If the user selects a file, the initial name should be replaced with the name (not including the path) of the selected file.

`fullName`

The file request returns the selected file name, including the path, in this string. The initial contents are ignored.

`bufLen`

The size in bytes of the `baseName`, `path` and `fullName` strings. When writing to these strings, the file request plug-in should not write more than `bufLen` characters, including the NULL terminating byte.

`error = **pickName**()`

A callback function provided by the host for `FREQ_MULTILOAD` requests. This function should be called for each selected file when the request type is `FREQ_MULTILOAD`, even if only one file is selected. For each call, the `baseName`, `path` and `fullName` fields should be filled with the data for the next selected file in the list. The function returns 0 to continue and any non-zero value to stop processing the files in a multiple file selection.

Example

The wfilereq sample is a FileRequester that uses the Windows common file dialog.

FrameBufferHandler
FrameBufferInterface

Availability LightWave 6.0
Component Layout
Header lwframbuf.h

Frame buffer plug-ins display the current rendered frame, either on the screen or to another output device.

Handler Activation Function

```
   XCALL_( int ) MyFrameBuffer( long version, GlobalFunc *global,
      LWFrameBufferHandler *local, void *serverData );
```

The `local` argument to a frame buffer's activation function is an LWFrameBufferHandler.

```
   typedef struct st_LWFrameBufferHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      int              type;
      LWError         (*open)  (LWInstance, int w, int h);
      void            (*close) (LWInstance);
      LWError         (*begin) (LWInstance);
      LWError         (*write) (LWInstance, const void *R, const void *G,
                                  const void *B, const void *alpha);
      void            (*pause) (LWInstance);
   } LWFrameBufferHandler;
```

The first two member of this structure point to the standard handler functions. In addition to these, a frame buffer also provides functions for the start and end of a rendering session, for receiving a frame's scanlines, and for displaying the frame, and it specifies what type of data it wants to receive.

`type`
: The type of pixel data Layout should send to the `write` function. This can be either `LWFBT_UBYTE` or `LWFBT_FLOAT`.

`error = **open**( instance, width, height )`
: Initialize the frame buffer. Called when a rendering session begins. Returns an error message string if an error occurs, otherwise it returns NULL.

`**close**( instance )`

Close the frame buffer. Called when the rendering session is complete.

`error = **begin**( instance )`

Prepare to receive the next frame. Returns an error message string if an error occurs, otherwise it returns NULL.

`error = **write**( instance, R, G, B, alpha )`

Receive the next scanline of the current frame. The scanlines for each frame are sent in order from top to bottom. The buffer arguments point to arrays of color channel values. There are exactly `width` values for each channel, one for each pixel in a scanline, and the values are either unsigned bytes or floats, depending on the `type` code. Returns an error message string or NULL.

`**pause**( instance )`

Pause awaiting user input. This is called for F9 and manually advanced frames, but not during automatic rendering. Typically the frame buffer displays the image here and then waits for the user to dismiss the display before returning from this function.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. A frame buffer's interface is invoked when the plug-in is selected as the render display on the Render Options panel.

Example

The framebuf sample is a simple example of a frame buffer. It uses the Raster Functions and Panels globals to store and display the rendered image.

Global

Availability LightWave 6.0
Component Layout, Modeler
Header lwglobsrv.h

Global class plug-ins provide services that other plug-ins can use. They extend the list of globals that are part of the plug-in API.

Other plug-ins call your global class plug-in by calling the GlobalFunc with your server name as the first argument. LightWave calls your activation function, which fills in the `local->data` field. This is then passed back to the caller as the return value of the GlobalFunc call.

Activation Function

The `local` argument to a global's activation function is an LWGlobalService.

```
   typedef struct st_LWGlobalService {
      const char *id;
      void       *data;
   } LWGlobalService;
```

`id`

The server name. This will be the same as the `name` field of the plug-in's server record. It's also the string that the requesting plug-in passed as the first argument to the GlobalFunc. If the module contains more than one global plug-in and they share a single activation function, the `id` can be used to tell which global is being requested.

`data`

The return value of the global. Fill this in with whatever is appropriate to satisfy the global request, or NULL to indicate failure. The value is typically a pointer to static data.

Global class plug-ins are available in both Modeler and Layout by default. If you don't want to run in one of these components, call the System ID global in your activation function and return `AFUNC_BADAPP` if the `LWSYS_TYPEBITS` of the return value don't match a program you will run in. The following fragment will allow your global to be activated in Layout and Screamernet, but not in Modeler.

```
unsigned long sysid, app;

sysid = ( unsigned long ) global( LWSYSTEMID_GLOBAL,
   GFUSE_TRANSIENT );
app = sysid & LWSYS_TYPEBITS;
if ( app != LWSYS_LAYOUT && app != LWSYS_SCREAMERNET )
   return AFUNC_BADAPP;
```

Example

The vecmath sample is a Global class plug-in that provides a library of vector and matrix routines. Information on how to use this library in your plug-ins is given in the comments at the top of the source file.

ImageFilterHandler
ImageFilterInterface

Availability LightWave 6.0
Component Layout, Modeler
Header lwfilter.h

Image filter plug-ins apply image post processing (filtering) effects to the rendered image.

Handler Activation Function

```
   XCALL_( int ) MyImageFilter( long version, GlobalFunc *global,
      LWImageFilterHandler *local, void *serverData );
```

The `local` argument to an image filter's activation function is an LWImageFilterHandler.

```
   typedef struct st_LWImageFilterHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      void            (*process)(LWInstance, const LWFilterAccess *);
      unsigned int    (*flags)  (LWInstance);
   } LWImageFilterHandler;
```

The first two members of this structure contain standard handler functions. In addition to these, an image filter provides a processing function and a flags function.

The `context` argument to the `inst->create` function is a pointer to an integer containing context flags. If the `LWFCF_PREPROCESS` flag is set, the instance is being created for an image other than the rendered output, and buffers other than the RGBA of the image won't be available.

An image filter can be activated by both Layout and Modeler. When activated by Modeler, the LWItemFuncs pointer in the local data is NULL. Be sure to test for this before filling in the `useItems` and `changeID` fields. Note too that if your image filter relies on Layout-only globals, those won't be available when Modeler calls your callbacks.

process(instance, access)
This is where the image filter does its work. For each frame, the filter

is given access to the red, green, blue and alpha channels of the rendered image, along with any other image buffers requested by the flags function. The access structure, described below, provides image information and functions for examining the buffers and writing new RGB and alpha values.

flags`( instance )`

Returns an int that tells the renderer which buffers the image filter will need to examine. The return value contains bitfields combined using bitwise-or. The symbols listed here and in `lwfilter.h` are bit positions, not the flags themselves, so you'll need to form the expression `(1 << LWBUF_WHATEVER)` to create the flags before or-ing them together.

The renderer may ignore requests from the processing function for access to any buffers you haven't asked for here. The buffers are

LWBUF_RED
LWBUF_GREEN
LWBUF_BLUE
LWBUF_ALPHA

The final output of the rendering pass. These form the image to be modified by the filter. They are always provided to every image filter (it isn't necessary to return flags for them in the flags function).

LWBUF_LUMINOUS
LWBUF_DIFFUSE
LWBUF_SPECULAR
LWBUF_MIRROR
LWBUF_TRANS
LWBUF_RAW_RED
LWBUF_RAW_GREEN
LWBUF_RAW_BLUE

The raw, unshaded values of the surface parameters at each pixel.

LWBUF_SHADING

A picture of the diffuse shading and specular highlights applied to the objects in the scene. This is a component of the rendering calculations that depends solely on the angle of incidence of the lights on a surface. It doesn't include the effects of explicit shadow calculations.

LWBUF_DIFFSHADE
LWBUF_SPECSHADE
: Like the `LWBUF_SHADING` buffer, but these store the amount of diffuse and specular shading (highlighting) separately, rather than adding them together. They should not be confused with the `LWBUF_DIFFUSE` and `LWBUF_SPECULAR` buffers, which store the unshaded surface values for those parameters.

LWBUF_SHADOW
: Indicates where shadows are falling in the final image. It may also be thought of as an illumination map, showing what parts of the image are visible to the lights in the scene.

LWBUF_GEOMETRY
: The values in this buffer are the dot-products of the surface normals with the eye vector (or the cosine of the angle of the surfaces to the eye). They reveal something about the underlying shape of the objects in the image. Where the value is 1.0, the surface is facing directly toward the camera, and where it's 0, the surface is edge-on to the camera.

LWBUF_DEPTH
: The distance from the camera to the nearest object visible in a pixel. Strictly speaking, this is the perpendicular distance from the plane defined by the camera's position and view vector. Also known as the z-buffer.

LWBUF_MOTION_X
LWBUF_MOTION_Y
: Support for 2D vector-based motion blur. These buffers contain the pixel distance moved by the item visible in each pixel. The amount of movement depends on the amount of time the shutter was open (controlled by the blur length user setting) and

includes the effects of the camera's own motion.

LWBUF_REFL_RED
LWBUF_REFL_GREEN
LWBUF_REFL_BLUE
The RGB levels of the contribution from mirror reflections calculated by raytracing or environment mapping.

LWBUF_SPECIAL
Contains user-assigned values that are unique for each surface.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Filter Access

This is the access structure passed to the processing function. The data members are read-only. The functions provide the means to get and set pixel values, and the optional monitor informs the user of the filter's progress.

```
   typedef struct st_LWFilterAccess {
      int        width, height;
      LWFrame    frame;
      LWTime     start, end;
      float *   (*getLine)  (int type, int y);
      void      (*setRGB)   (int x, int y, const LWFVector rgb);
      void      (*setAlpha) (int x, int y, float alpha);
      LWMonitor *monitor;
   } LWFilterAccess;
```

width, **height**
The dimensions, in pixels, of all of the image buffers.

frame
The frame number.

start, **end**
The start and end times for the frame. These times are the same unless the frame was rendered with motion blur, in which case the difference between them can be considered the exposure time for the frame.

`buf = `**`getLine`**`( buftype, y )`

Returns a pointer to the start of a scanline from the specified buffer. `y=0` is the top line of the buffer, and `y=height-1` is the bottom line. Don't try to look past the end of a scanline. Layout may not store scanlines contiguously for a given buffer. In fact, scanlines aren't guaranteed to exist at all until they're requested through these functions. The type codes are the same as those used by the flags function. NULL is returned for invalid type codes, or type codes for buffers not requested by the flags function.

`setRGB``( x, y, rgb )`
`setAlpha``( x, y, a )`

Use these functions to set the output values of the filter. The input RGBA buffers do not change as the output buffers are modified. A filter must set every pixel in the output image even if it does not alter the value, but it can set them in any order.

`monitor`

The filter can use this to update the user about its progress through the frame. This also allows the user to cancel the rendering during the filter's processing. The monitor mechanism is the same one provided by the monitor global. As with all monitors, the number of steps should be kept fairly low since checking for abort can have significant overhead on some systems.

Example

The negative sample is a simple filter that inverts the colors of the image. The convolve sample is a somewhat more complete example. It applies a 3 x 3 convolution to the image and includes an interface that allows the user to set the values of the filter kernel. These values are stored and retrieved using the handler `save` and `load` functions. The zcomp sample includes an image filter that saves the current `LWBUF_DEPTH` buffer to a file.

ImageLoader

Availability LightWave 6.0
Component Layout, Modeler
Header lwimageio.h

Image loaders read image files. Each of them typically supports a single format.

When an image loader's activation function is called, it should open the image file and try to recognize its contents. LightWave calls all of the installed image loaders in sequence until one of them recognizes the file. Each image loader is therefore responsible for identifying the files it can load. If the file isn't one the loader understands, the loader sets the `result` field of the `local` structure to `IPSTAT_NOREC` and returns `AFUNC_OK`.

If, on the other hand, the loader understands the image file, it calls `local->begin` to get the image protocol functions and then loads the file.

Activation Function

```
   XCALL_( int ) MyImageLoader( long version, GlobalFunc *global,
      LWImageLoaderLocal *local, void *serverData );
```

The `local` argument to an image loader's activation function is an LWImageLoaderLocal.

```
   typedef struct st_LWImageLoaderLocal {
      void              *priv_data;
      int                result;
      const char        *filename;
      LWMonitor         *monitor;
      LWImageProtocolID (*begin) (void *, LWImageType);
      void              (*done)  (void *, LWImageProtocolID);
   } LWImageLoaderLocal;
```

`priv_data`
Pass this as the first argument to the `begin` and `done` functions. It's an opaque pointer to data used internally by LightWave.

`result`
Set this to indicate whether the image was loaded successfully. The result codes are

`IPSTAT_OK`
: The image was loaded successfully.

`IPSTAT_NOREC`
: The loader didn't recognize the file. This can happen frequently, since all loaders are called in sequence until one of them *doesn't* return this result.

`IPSTAT_BADFILE`
: The loader couldn't open the file. If the loader is able to open the file but believes it has found an error in the contents, it should return `IPSTAT_NOREC`.

`IPSTAT_ABORT`
: Use this to indicate that the user cancelled the load operation. This can happen if you use the monitor to indicate the progress of a lengthy image loading operation.

`IPSTAT_FAILED`
: An error occurred during loading, for example a memory allocation failed.

`filename`
: The name of the file to load.

`monitor`
: A monitor for displaying the progress of the load to the user. You don't have to use this, but you're encouraged to if your image loading takes an unusual amount of time. This is the same structure as that returned by the monitor global's `create` function.

`protocol =` **`begin`**`( priv_data, pixeltype )`
: Call this when you're ready to begin sending image data to LightWave. This will be after you've opened and examined the image file and know what it contains. The pixel type code tells LightWave what kind of pixel data you will be sending, and this will in general depend on what kind of pixel data the file contains, although it doesn't have to. Pixel type codes are listed on the Image I/O page.

  The `begin` function returns a pointer to an LWImageProtocol, which is the structure you'll use to actually transfer the image data. See the Image I/O page. If you call `begin`, you must also call `done` so that LightWave can free resources associated with the LWImageProtocol it allocates for you.

`done``( priv_data, protocol )`

Completes the image loading process. The protocol is the LWImageProtocolID returned by `begin`. Only call `done` if you previously called `begin`.

Example

The iff sample is a complete IFF ILBM loader and saver. The zcomp sample includes an image loader that creates a floating-point grayscale image from the values in a previously saved `LWBUF_DEPTH buffer`.

ImageSaver

Availability LightWave 6.0
Component Layout
Header lwimageio.h

Image savers write image files. Each of them typically supports a single format.

When a saver's activation function is called, it should try to open the output file named in the `local` structure. If the open fails, the saver can set `local->result` to `IPSTAT_BADFILE` and return immediately. Otherwise, the saver creates and initializes an image protocol and calls `sendData` to tell LightWave it's ready to receive image data. LightWave then calls the saver's callbacks to transfer the data. `sendData` doesn't return until LightWave calls the saver's `done` callback.

Activation Function

```
   XCALL_( int ) MyImageSaver( long version, GlobalFunc *global,
      LWImageSaverLocal *local, void *serverData );
```

The `local` argument to an image saver's activation function is an LWImageSaverLocal.

```
   typedef struct st_LWImageSaverLocal {
      void        *priv_data;
      int          result;
      LWImageType  type;
      const char  *filename;
      LWMonitor   *monitor;
      int         (*sendData) (void *, LWImageProtocolID, int flags);
   } LWImageSaverLocal;
```

`priv_data`
: Pass this to the `sendData` function. It's an opaque pointer to data used internally by LightWave.

`result`
: Set this to indicate whether the image was saved successfully. The result codes are

  `IPSTAT_OK`

The image was saved successfully.

`IPSTAT_BADFILE`
: The saver couldn't open the file.

`IPSTAT_ABORT`
: Use this to indicate that the user cancelled the save operation. This can happen if you use the monitor to indicate the progress of a lengthy image saving operation.

`IPSTAT_FAILED`
: An error occurred during saving.

`type`

The kind of pixel data to be saved. Pixel types are listed on the Image I/O page. The most common types will be `LWIMTYP_RGBAFP` for color images and `LWIMTYP_GREYFP` for grayscale images. Use this to decide what kind of pixel data you want to receive. If your file format supports 24-bit color and 8-bit grayscale, for example, you would set your image protocol type to `LWIMTYP_RGB24` when the local `type` field contains any of the RGB types, and `LWIMTYP_GREY8` when it contained either `LWIMTYP_GREYFP` or `LWIMTYP_GREY8`.

`filename`

The name of the image file to write.

`monitor`

A monitor for displaying the progress of the save to the user. You don't have to use this, but you're encouraged to if your image saving takes an unusual amount of time. This is the same structure returned by the monitor global.

`result = `**`sendData`**`( priv_data, protocol, flags )`

Call this when you're ready to begin receiving image data from LightWave. This will be after you've filled in the fields of an appropriate LWImageProtocol structure, which is described on the Image I/O page. The only flag currently defined is `IMGF_REVERSE`, which instructs LightWave to send scanlines in bottom-to-top order. When you call `sendData`, LightWave calls the functions you provided in your image protocol structure to actually save the image. `sendData` won't return until the image is saved.

Example

The iff sample is a complete IFF ILBM loader and saver.

ItemMotionHandler
ItemMotionInterface

Availability LightWave 6.0
Component Layout
Header lwmotion.h

Motion handlers apply procedural translation, rotation and scaling to an item. They can be associated with any item in a scene that can be keyframed (objects, lights, cameras, bones).

Handler Activation Function

```
   XCALL_( int ) MyItemMotion( long version, GlobalFunc *global,
      LWItemMotionHandler *local, void *serverData );
```

The `local` argument to a motion handler's activation function is an LWItemMotionHandler.

```
   typedef struct st_LWItemMotionHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      void            (*evaluate)(LWInstance, const LWItemMotionAccess *);
      unsigned int    (*flags)   (LWInstance);
   } LWItemMotionHandler;
```

The first two members of this structure are standard handler functions. The `context` argument to the `inst->create` function is the LWItemID of the item associated with this instance.

In addition to the standard handler functions, a motion handler provides an evaluation function and a flags function.

evaluate(instance, access)
: This is where the motion handler does its work. LightWave calls the evaluation function at every point in the animation at which an item's motion parameters need to be calculated. The access structure, described below, tells you the item being animated and the frame and time of the evaluation, and provides functions to set motion parameters for the current time and to get the item's motion parameters for any time.

`f = `**`flags`**`( instance )`

Returns an integer containing flags combined using bitwise-or. Currently the only flag is `LWIMF_AFTERIK`, which specifies that the plug-in will be evaluated after LightWave has performed the inverse kinematics calculations for the item.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

The interface activation's local data is the standard interface structure for handlers.

Motion Access

The evaluation function receives an LWItemMotionAccess structure. The data members are read-only. The functions provide the means to get and set motion parameters.

```
   typedef struct st_LWItemMotionAccess {
      LWItemID  item;
      LWFrame   frame;
      LWTime    time;
      void     (*getParam) (LWItemParam, LWTime, LWDVector);
      void     (*setParam) (LWItemParam, const LWDVector);
   } LWItemMotionAccess;
```

`item`

The ID for the item to be affected by the procedural motion.

`frame`

The frame number at which the motion should be evaluated.

`time`

The animation time for which the motion should be evaluated.

`getParam``( param, lwtime, vec )`

Returns a motion parameter for the item at any given time. Only the `LWIP_POSITION`, `LWIP_ROTATION` and `LWIP_SCALING` parameters may be queried.

`setParam``( param, vec )`

Used by the evaluation function to set the computed motion of the item at the current time. Only the `LWIP_POSITION`, `LWIP_ROTATION` and `LWIP_SCALING` parameters may be set.

Example

If you want to modify an item's motion, rather than completely replace it, call `getParam` for the current time to find out what the item's unmodified motion would be, then calculate a new motion based on that and call `setParam`.

```
   XCALL_( static void )
   Evaluate( MyInstance *inst, const LWItemMotionAccess *access )
   {
      LWDVector pos;

      access->getParam( LWIP_POSITION, access->time, pos );
      ...do something to pos[]...
      access->setParam( LWIP_POSITION, pos );
   }
```

The kepler sample is a motion handler that moves an item in an elliptical orbit

LayoutGeneric

Availability LightWave 6.0
Component Layout
Header lwgeneric.h

Layout generic plug-ins can issue commands to alter the scene. They can also manipulate scene settings at a lower level by saving, altering, and reloading scene files. Generics also supply general-purpose, non-rendering functionality for doing things like configuring external devices, performing scratch calculations, or displaying scene information.

Activation Function

```
XCALL_( int ) MyGeneric( long version, GlobalFunc *global,
   LWLayoutGeneric *local, void *serverData );
```

The `local` argument to a generic's activation function is an LWLayoutGeneric.

```
typedef struct st_LWLayoutGeneric {
   int           (*saveScene)(const char *file);
   int           (*loadScene)(const char *file, const char *name);
   void          *data;
   LWCommandCode (*lookup)   (void *, const char *cmdName);
   int           (*execute)  (void *, LWCommandCode cmd, int argc,
                                const DynaValue *argv,
                                DynaValue *result);
   int           (*evaluate) (void *, const char *command);
} LWLayoutGeneric;
```

ok = **saveScene**(filename)
: Save the scene in its current state as a LightWave scene file.

ok = **loadScene**(filename, newname)
: Load a scene file. The scene is loaded from the file named in the first argument. The second argument is the default filename for subsequent saving of the scene and the name that will be displayed to the user.

data
: An opaque pointer to data used internally by Layout. Pass this as the first argument to the `lookup`, `execute` and `evaluate` functions.

cmdcode = **lookup**(data, cmdname)

Returns an integer code corresponding to the command name. The command is issued by passing the command code to the `execute` function. Command codes are constant for a given Layout session, so this only needs to be called once per command, after which the codes can be cached and used multiple times.

result = **execute**(data, cmdcode, argc, argv, cmdresult)

Issue the command given by the command code argument. `argv` is an array of DynaValue arguments. `argc` is the number of arguments in the `argv` array. The result of the command is written in `cmdresult`. The function returns 1 if it succeeds or 0 if it does not.

result = **evaluate**(data, cmdstring)

Issue the command with the name and arguments in the command string. This is an alternative to using `lookup` and `execute`. The command and its arguments are written to a single string and delimited by spaces. The function returns 1 if it succeeds or 0 if it does not.

See the Commands pages for a complete list of the commands that can be issued in Layout, as well as a detailed explanation of the formatting of command arguments for both the `execute` and `evaluate` methods.

Example

The hello sample is the LightWave version of everybody's favorite "Hello, World!" program. It opens a panel with an edit field, displays messages, and issues a command.

LayoutTool

Availability Future
Component Layout
Header lwlaytool.h

Layout tool plug-ins are just custom Layout tools. To the user, they behave like Layout's built-in tools (the tools activated by the Move, Rotate and Scale buttons, for example).

Support for this class hasn't been implemented yet.

Activation Function

```
XCALL_( int ) MyLayoutTool( long version, GlobalFunc *global,
   LWLayoutTool *local, void *serverData );
```

The `local` argument to a Layout tool's activation function is an LWLayoutTool.

```
typedef struct st_LWLayoutTool {
   LWInstance         instance;
   LWLayoutToolFuncs *tool;
} LWLayoutTool;
```

The activation function fills in the `instance` field and the callbacks of the `tool` field and returns. As with handlers, the remaining interaction between Layout and the plug-in takes place through the callbacks.

instance
A pointer to your user data. This will be passed to each of the tool callbacks.

tool
Points to a structure containing function pointers for your callbacks, described below.

Tool Functions

The `tool` field of the LWLayoutTool is a pointer to an LWLayoutToolFuncs.

```
typedef struct st_LWLayoutToolFuncs {
   void         (*done)   (LWInstance);
   void         (*draw)   (LWInstance, LWCustomObjAccess *);
   const char * (*help)   (LWInstance, LWToolEvent *);
   int          (*dirty)  (LWInstance);
   int          (*count)  (LWInstance, LWToolEvent *);
   int          (*handle) (LWInstance, LWToolEvent *, int i,
                              LWDVector pos);
   int          (*start)  (LWInstance, LWToolEvent *);
   int          (*adjust) (LWInstance, LWToolEvent *, int i);
   int          (*down)   (LWInstance, LWToolEvent *);
   void         (*move)   (LWInstance, LWToolEvent *);
   void         (*up)     (LWInstance, LWToolEvent *);
   void         (*event)  (LWInstance, int code);
   LWXPanelID   (*panel)  (LWInstance);
} LWLayoutToolFuncs;
```

done(inst)

Destroy the instance. Called when the user discards the tool.

draw(inst, custobj_access)

Display a wireframe representation of the tool in a 3D viewport. Typically this draws the handles.

helptext = **help**(inst, event)

Returns a text string to be displayed as a help tip for this tool.

dirty = **dirty**(inst)

Returns flag bit if either the wireframe or help string need to be refreshed.

nhandles = **count**(inst, event)

Returns the number of handles. If zero, then `start` is used to set the initial handle point.

priority = **handle**(inst, event, handle, pos)

Returns the 3D location and priority of the handle, or zero if the handle is currently invalid.

handle = **start**(inst, event)

Take an initial mouse-down position and return the index of the handle that should be dragged.

handle = **adjust**(inst, event, handle)

Drag the given handle to a new location and return the index of the handle that should continue being dragged (often the same as the input).

rawmouse = **down**(inst, event)
Process a mouse-down event. If this function returns false, handle processing will be done instead of raw mouse event processing.

move(inst, event)
Process a mouse-move event. This is only called if the `down` function returned true.

up(inst, event)
Process a mouse-up event. This is only called if the `down` function returned true.

event(inst, code)
Process a general event: DROP, RESET or ACTIVATE

panel = **panel**(inst)
Create and return a view-type xPanel for the tool instance.

Example

.

MasterHandler

Availability LightWave 6.0 **Component** Layout
Header lwmaster.h

Masters can issue commands like generics, but unlike generics, masters can respond to the user's changes to a scene as the scene is being composed. Masters are handlers, so they have persistent instances that can be saved in scene files. Masters can be used to record a sequence of commands for scripting or as a central point of control for a suite of handler plug-ins.

Activation Function

```
   XCALL_( int ) MyMaster( long version, GlobalFunc *global,
      LWMasterHandler *local, void *serverData );
```

The `local` argument to a master's activation function is an LWMasterHandler.

```
   typedef struct st_LWMasterHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      int              type;
      double          (*event) (LWInstance, const LWMasterAccess *);
      unsigned int    (*flags) (LWInstance);
   } LWMasterHandler;
```

The first two members of this structure point to standard handler functions. In addition to these, a master handler provides a type code, an event function and a flags function.

`type`

This can be one of the following.

`LWMAST_SCENE`
`LWMAST_OBJECTS`
`LWMAST_EFFECTS`

`LWMAST_LAYOUT`

The SCENE type is the most common. OBJECTS and EFFECTS types are reserved for future enhancement of the class. LAYOUT masters are like SCENE masters, but they survive scene clearing and can therefore be used to automate scene management.

`val = event( instance, access )`

The event function is called to notify the handler that something has happened. Information about the event is included in the access structure, described below. The handler can respond to the event by issuing commands through functions provided in the access structure. The return value is currently ignored and should be set to 0.

`f = flags( instance )`

Returns flag bits combined using bitwise-or. No flags are currently defined, so this function should simply return 0.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Master Access

This is the structure passed to the handler's event function.

```
   typedef struct st_LWMasterAccess {
      int            eventCode;
      void          *eventData;
      void          *data;
      LWCommandCode (*lookup)   (void *, const char *cmdName);
      int           (*execute)  (void *, LWCommandCode cmd, int argc,
                                   const DynaValue *argv,
                                   DynaValue *result);
      int           (*evaluate) (void *, const char *command);
   } LWMasterAccess;
```

`eventCode`

`eventData`

The type of event and its associated data. The event can be one of the following.

`LWEVNT_NOTHING`

Not currently used.

`LWEVNT_COMMAND`

A user action corresponding to a command. The `eventData` is a string containing the command and its arguments, written in the same format used by the `evaluate` function to issue commands.

`LWEVNT_TIME`

Sent whenever the frame slider is moved, which includes playing the scene, but not playing back a preview. This allows masters to remain synchronized in time with the Layout interface.

`LWEVNT_SELECT`
: Sent when the item selection has changed.

`LWEVNT_RENDER_DONE`
: Sent when a render has been completed.

`data`
: An opaque pointer to data used internally by Layout. Pass this as the first argument to the `lookup`, `execute` and `evaluate` functions.

`cmdcode = `**`lookup`**`( data, cmdname )`
: Returns an integer code corresponding to the command name. The command is issued by passing the command code to the `execute` function. Command codes are constant for a given Layout session, so this only needs to be called once per command, after which the codes can be cached and used multiple times.

`result = `**`execute`**`( data, cmdcode, argc, argv, cmdresult )`
: Issue the command given by the command code argument. `argv` is an array of DynaValue arguments. `argc` is the number of arguments in the `argv` array. The result of the command is written in `cmdresult`. The function returns 1 if it succeeds or 0 if it does not.

`result = `**`evaluate`**`( data, cmdstring )`
: Issue the command with the name and arguments in the command string. This is an alternative to using `lookup` and `execute`. The command and its arguments are written to a single string and delimited by spaces. The function returns 1 if it succeeds or 0 if it does not.

See the Commands pages for a complete list of the commands that can be issued in Layout, as well as a detailed explanation of the formatting of command arguments for both the `execute` and `evaluate` methods.

History

The `LWEVNT_SELECT` event code was added in LightWave 7.0. The `LWEVNT_RENDER_DONE` event code was added in LightWave 7.5.

Example

The macro sample is a master that records a sequence of commands and saves it as an LScript.

MeshDataEdit

Availability LightWave 6.0
Component Modeler
Header lwmeshedt.h

Mesh edit plug-ins create and modify geometry at the point and polygon level. This class is a subset of the CommandSequence class, which provides access to both mesh editing and commands, and of the MeshEditTool class, an interactive version of MeshDataEdit.

Activation Function

```
   XCALL_( int ) MyMeshEdit( long version, GlobalFunc *global,
      MeshEditBegin *local, void *serverData );
```

The `local` argument to a mesh edit plug-in's activation function is a MeshEditBegin.

```
   typedef MeshEditOp *
      MeshEditBegin (int pntBuf, int polBuf, EltOpSelect);
```

This function returns a MeshEditOp structure containing the mesh editing functions. It can be called only once for each activation.

The MeshEditBegin function can allocate a user data buffer for each point and polygon. This is memory you can use to store per-point and per-polygon information during the edit. Modeler automatically frees these buffers when the edit is completed. The `pntBuf` and `polBuf` arguments set the sizes of the buffers.

EltOpSelect

When the edit begins, Modeler sets a selection flag for each point and polygon. The EltOpSelect code determines which geometry is flagged as selected, and it can be one of the following.

OPSEL_GLOBAL
: All elements, whether or not they're selected by the user.

OPSEL_USER

    Only those elements selected by the user. This includes the implicit selection of all elements when nothing is explicitly selected, and selections by volume.

OPSEL_DIRECT

    Elements selected directly with the point or polygon selection tools. This applies to both points and polygons regardless of which is currently active in the interface.

OPSEL_MODIFY

    This activates a special mesh edit mode that can change the selection state of specific points and polygons. The mesh editing functions for adding and changing geometry aren't available in this mode, but the query functions can be used. The selection state of a point or polygon is modified by calling the MeshEditOp `pntSelect` or `polSelect` functions, typically within a `pointScan` or `polyScan` callback. `OPSEL_MODIFY` must be combined with one of the other selection modes in the MeshEditBegin call.

EltOpLayer

Many of the MeshEditOp functions operate on a specific set of layers, and these are identified by an EltOpLayer code.

`OPLYR_PRIMARY`

    The primary layer. This is the single active layer affected by mesh edits, normally the lowest numbered foreground layer.

`OPLYR_FG`

    Foreground layers, which are active and displayed.

`OPLYR_BG`

    Background layers, which are inactive but still displayed.

`OPLYR_SELECT`

    Both foreground and background layers.

`OPLYR_ALL`

    All layers in the Modeler system whether they contain data or not.

`OPLYR_EMPTY`

    Empty layers are those that contain no geometry.

`OPLYR_NONEMPTY`

    Non-empty layers are any layers which contain some data (the complement of `OPLYR_EMPTY`).

Individual Layers

In addition to the defined values, codes starting at 101 (for layer 1) can be used to select the individual layers by number.

Error Codes

Most of the mesh edit functions return an error state defined by one of the following codes. One of these is also passed to the MeshEditOp `done` function.

EDERR_NONE
: Success.

EDERR_NOMEMORY
: A memory allocation failed.

EDERR_BADLAYER
: An operation was attempted in an invalid layer.

EDERR_BADSURF
: The edit created an invalid surface name.

EDERR_USERABORT
: The user (or the plug-in) ended the edit before it was completed.

`EDERR_BADVMAP`
: The operation involved an invalid vertex map.

EDERR_BADARGS
: The function failed for a reason not covered by the other error codes.

MeshEditOp

The MeshEditBegin function returns a MeshEditOp containing data and functions for performing mesh edits.

```
   typedef struct st_MeshEditOp {
      EDStateRef       state;
      int              layerNum;
      void            (*done)      (EDStateRef, EDError, int selm);
      int             (*pointCount)(EDStateRef, EltOpLayer, int mode);
      int             (*polyCount) (EDStateRef, EltOpLayer, int mode);
      EDError         (*pointScan) (EDStateRef, EDPointScanFunc *,
                                      void *, EltOpLayer);
      EDError         (*polyScan)  (EDStateRef, EDPolyScanFunc *,
                                      void *, EltOpLayer);
      EDPointInfo *   (*pointInfo) (EDStateRef, LWPntID);
      EDPolygonInfo * (*polyInfo)  (EDStateRef, LWPolID);
      int             (*polyNormal)(EDStateRef, LWPolID, double[3]);
      LWPntID         (*addPoint)  (EDStateRef, double *xyz);
      LWPolID         (*addFace)   (EDStateRef, const char *surf,
                                      int numPnt, const LWPntID *);
      LWPolID         (*addCurve)  (EDStateRef, const char *surf,
                                      int numPnt, const LWPntID *,
                                int flags);
      EDError        (*addQuad)   (EDStateRef, LWPntID, LWPntID,
                                LWPntID, LWPntID);
      EDError        (*addTri)    (EDStateRef, LWPntID, LWPntID,
                                LWPntID);
      EDError        (*addPatch)  (EDStateRef, int nr, int nc, int lr,
                                int lc, EDBoundCv *r0,
                                EDBoundCv *r1, EDBoundCv *c0,
                                EDBoundCv *c1);
      EDError        (*remPoint)  (EDStateRef, LWPntID);
      EDError        (*remPoly)   (EDStateRef, LWPolID);
      EDError        (*pntMove)   (EDStateRef, LWPntID, const double *);
      EDError        (*polSurf)   (EDStateRef, LWPolID, const char *);
      EDError        (*polPnts)   (EDStateRef, LWPolID, int,
                                const LWPntID *);
      EDError        (*polFlag)   (EDStateRef, LWPolID, int mask,
                                int value);
      EDError        (*polTag)    (EDStateRef, LWPolID, LWID,
                                const char *);
      EDError        (*pntVMap)   (EDStateRef, LWPntID, LWID,
                                const char *, int, float *);
      LWPolID        (*addPoly)   (EDStateRef, LWID type, LWPolID,
                                const char *surf, int numPnt,
                                const LWPntID *);
      LWPntID        (*addIPnt)   (EDStateRef, double *xyz, int numPnt,
                                const LWPntID *, const double *wt);
      EDError        (*initUV)    (EDStateRef, float *uv);
      void *         (*pointVSet) (EDStateRef, void *, LWID,
                                const char *);
      int            (*pointVGet) (EDStateRef, LWPntID, float *);
      const char *   (*polyTag)   (EDStateRef, LWPolID, LWID);
      EDError        (*pntSelect) (EDStateRef, LWPntID, int);
      EDError        (*polSelect) (EDStateRef, LWPolID, int);
      int            (*pointVPGet)(EDStateRef, LWPntID, LWPolID,
                                float *);
      int            (*pointVEval)(EDStateRef, LWPntID, LWPolID,
                                float *);
      EDError        (*pntVPMap)  (EDStateRef, LWPntID, LWPolID,
                                LWID, const char *, int, float *);
   } MeshEditOp;
```

state

An opaque pointer to data used internally by Modeler during the mesh edit. Pass this as the first argument to all of the edit functions.

layerNum

Points and polygons may only be created or modified in the primary active layer, which is given by this layer number. The primary layer is the lowest numbered foreground layer.

done(state, error, selset)

Call this when the edit is complete. As changes are made during an edit, they are buffered through Modeler's undo mechanism, so they are not reflected in the data until done is called, and not at all if `done` sets the error argument.

In general, if one of the edit functions returns an error, you'll pass that error to `done`. If you just want the edit to stop or be discarded, possibly because the user pressed the Cancel button in a progress monitor, you'll pass `EDERR_USERABORT`. If an error occurs in your plug-in, you'll pass `EDERR_NOMEMORY` (for memory allocation errors) or `EDERR_BADARGS` (for everything else). And if the edit was successful, you'll use `EDERR_NONE`.

The `selset` argument tells Modeler how you want the selection to appear to the user after the edit has been applied. It contains flags combined using bitwise-or, and can include the following.

EDSELM_CLEARCURRENT	Deselect elements that were selected when the edit began.
EDSELM_SELECTNEW	Select elements created by the edit.
EDSELM_FORCEVRTS	Force selection of newly created vertices.
EDSELM_FORCEPOLS	Force selection of newly created polygons.

A value of 0 leaves all directly selected elements selected after the edit. The `CLEARCURRENT` and `SELECTNEW` flags are polite hints; they won't override selection settings made by the user. The force flags will always force direct selection of the points or polygons created by the edit.

npoints = **pointCount**(state, oplayer, selmode)
npolygons = **polyCount**(state, oplayer, selmode)

Returns the number of points or polygons that meet the layer and selection criteria. The selection mode can be one of the following.

EDCOUNT_ALL	Both selected and unselected points or polygons.
EDCOUNT_SELECT	Only selected points or polygons.
EDCOUNT_DELETE	

Only points or polygons flagged for deletion by this mesh edit.

err = **pointScan**(state, scanfunc, userdata, oplayer)
err = **polyScan**(state, scanfunc, userdata, oplayer)

Enumerate the points or polygons in the specified layers. For each element, Modeler calls the callback function you supply. The callbacks are defined this way.

```
typedef EDError EDPointScanFunc (void *, const EDPointInfo *);
typedef EDError EDPolyScanFunc (void *, const EDPolygonInfo *);
```

The `userdata` pointer is passed as the first argument to your callbacks, and it can be whatever is useful to you. The point and polygon info structures passed as the second argument are described later. If your callback returns an error, the scan is stopped and the callback's error is returned.

Point and polygon scans will enumerate all of the geometry in the layers you request, regardless of what geometry is selected, even if you begin the edit with `OPSEL_USER` or `OPSEL_DIRECT`. To find out whether a given element is selected (as defined by your choice of EltOpSelect), you need to test the EDPointInfo or EDPolygonInfo `flags` field for the `EDDF_SELECT` bit. Similarly, if you've deleted geometry during the mesh edit, it will still be enumerated, but the flags field of the info structure will contain `EDDF_DELETE`.

info = **pointInfo**(state, point)
info = **polyInfo**(state, polygon)

Returns information about a point or polygon. These return the same EDPointInfo and EDPolygonInfo structures that are passed to the scan callbacks.

ok = **polyNormal**(state, polygon, norm)

Get a polygon's normal. The normal is a unit vector perpendicular to the plane defined by the first, second and last vertex of the polygon. If the polygon has fewer than three vertices, or is somehow degenerate, `norm` isn't changed and the function returns 0. Otherwise it returns 1 and `norm` receives the normal.

point = **addPoint**(state, pos)

Create a point.

polygon = **addFace**(state, surfname, npoints, point_array)
Create a polygon. If the surface name is NULL, the polygon will be assigned the current default surface. The vertices are defined by an array of point IDs listed in clockwise order. The polygon normal will be inferred from the first, second and last points.

polygon = **addCurve**(state, surfname, npoints, point_array, flags)
Create a curve (a polygon of type `LWPOLTYPE_CURV`). The `EDPF_CCSTART` and `EDPF_CCEND` flags specify that the first and last points in the point array should serve as control points and not be included in the curve itself. To create a closed curve, both of these flags must be set, and the first and last point must overlap.

err = **addQuad**(state, v1, v2, v3, v4)
err = **addTri**(state, v1, v2, v3)
Create a quadrangle or a triangle with the current default surface. These two functions respect the user's settings for new geometry. Two collocated polygons with opposite normals will be created if the user has set the double-sided option, and quads will be split into two triangles if the user has set the triangles-only option.

err = **addPatch**(state, nr, nc, lr, lc, r0, r1, c0, c1)
Create a polygonal patch defined by three or four bounding curves. The first two arguments (after the EditStateRef) give the number of patch divisions in the R (row) and C (column) directions. The second two arguments are booleans; if 0, the divisions are equally spaced along the curve, and if 1, the spacing is determined by the positions of the curve knots. The last four arguments are the bounding curves, each defined by an EDBoundCV structure.

```
typedef struct st_EDBoundCv {
   LWPolID curve;
   int     start, end;
} EDBoundCv;
```

The `start` and `end` indexes are the points on the curve that should be used as endpoints for patching. The first and second curves define the R boundaries. The third and optional fourth curve define the C boundaries.

err = **remPoint**(state, point)

Delete the point. Modeler will flag the point as deleted, but will actually remove it from the database only after `done` is called.

err = **remPoly**(state, polygon)

Delete the polygon.

err = **pntMove**(state, point, pos)

Put a point in a new position.

err = **polSurf**(state, polygon, surfname)

Change the surface assigned to a polygon.

err = **polPnts**(state, polygon, npoints, point_array)

Replace the point list of a polygon.

err = **polFlag**(state, polygon, mask, value)

Set polygon flags. The mask contains 1 bits for each flag you want to change, and the value contains the new flag settings (0 or 1 for each 1 bit in the mask). Currently, the flags that can be changed are the `EDPF_START` and `EDPF_END` flags for curves.

err = **polTag**(state, polygon, tagtype, tag)

Add a polygon tag to a polygon, or change an existing tag. If the tag type is `LWPTAG_SURF`, the tag is the surface name. If the tag type is `LWPTAG_PART`, the tag is the part (or group) name. For anything other than surface tags, passing a NULL `tag` will remove an existing tag of the specified type.

err = **pntVMap**(state, point, type, name, nvalues, val_array)

Add a vmap vector to a point. The vmap type can be one of the following, or something else.

`LWVMAP_PICK` - selection set
`LWVMAP_WGHT` - weight map
`LWVMAP_MNVW` - subpatch weight map
`LWVMAP_TXUV` - texture UV coordinates
`LWVMAP_MORF` - relative vertex displacement (morph)
`LWVMAP_SPOT` - absolute vertex displacement (morph)

`LWVMAP_RGB`, `LWVMAP_RGBA` - vertex color

Pass a NULL `val_array` to *remove* a vmap vector from the point.

polygon = **addPoly**(state, type, template, surf, npoints, point_array)

Create a polygon. If a template polygon is supplied, Modeler copies the polygon tags for the new polygon from the template. If the surface name is NULL, the surface will be that of the template, or the current default surface if the template is NULL. The vertices of the new polygon are listed in clockwise order, and the normal will be inferred from the first, second and last vertex.

point = **addIPnt**(state, pos, npoints, point_array, weight_array)

Create an "interpolated" point. The new point's vmap values are calculated as a weighted average of the vmaps of the points in the points array. If `pos` is NULL, the position is also computed as a weighted average. If the weight array is NULL, the averaging over the point list is uniform. The weights are renormalized to sum to 1.0.

err = **initUV**(state, uv)

Set the texture UV for a point or polygon you're about to create. If a texture map is selected in Modeler's interface, the UVs will be assigned to that map as points or polygons are created. You'll typically want to give the user the option of whether or not to create UVs for new points and polygons.

When creating points, pass `initUV` an array of two floats and then call any of the functions that create a point. The two floats will be used as the U and V for the point, after which the `initUV` state will be cleared so that subsequent points have no UV unless the function is called again.

To initialize per-polygon, or discontinuous, UVs, call `initUV` with a pointer to 2*n* floats before creating a polygon with *n* vertices. For each vertex, if the point's continuous UV value is different from the UV in the array, then a polygon-specific UV is set for the vertex. If the point has no continuous UV, then the continuous value for the point is set to the polygon UV.

Any combination of these two methods can be used to assign UVs to new data. If only polygon UVs are specified, continuous UVs will still be created where polygons share UV values. Alternately, plug-ins can assign UVs to points and only specify polygon UVs along seam polygons.

vmapID = **pointVSet**(state, vmapID, vmaptype, vmapname)

Select a vmap for reading vectors. Returns an opaque pointer that can be used to select the same vmap in later calls to this function. The first time this is called for a given vmap, the pointer can be NULL, and Modeler will locate and select the vmap using the type and name arguments.

ismapped = **pointVGet**(state, point, val)

Read the vmap vector for a point. The vector is read from the vmap selected by a previous call to `pointVSet`. If the point is mapped (has a vmap value in the selected vmap), the `val` array is filled with the vmap vector for the point, and `pointVGet` returns true. If you don't already know the dimension of the vmap (the number of values per point, and therefore the required size of the `val` array), you can use the scene objects global to find out.

See also `pointVPGet` and `pointVEval`. `pointVGet` is equivalent to reading values from a VMAP chunk in an object file. It returns the continuous, or per-point, vector. For the raw discontinuous, or per-polygon-vertex value, use `pointVPGet`, and for the combined value accounting for both sources, use `pointVEval`.

tag = **polyTag**(state, polygon, tagtype)

Returns a tag string associated with the polygon. For the `LWPTAG_SURF` tag type, the surface name is returned.

err = **pntSelect**(state, point, setsel)
err = **polSelect**(state, polygon, setsel)

Set the selection state of a point or polygon. These can only be called during `OPSEL_MODIFY` mesh edits. The element is selected if `setsel` is true and deselected if it's false.

ismapped = **pointVPGet**(state, point, polygon, val)

Read the vmap vector for a polygon vertex. This is like `pointVGet`, but it returns the discontinuous vmap value, equivalent to reading entries in a VMAD chunk.

ismapped = **pointVEval**(state, point, polygon, val)

Read the vmap vector for a point, accounting for both continuous and discontinuous values. Generally, if a discontinuous value exists for the point, that value will be returned. Otherwise the continuous value is used.

err = **pntVPMap**(state, point, polygon, type, name, dim, val)

Add a discontinuous vmap vector to a polygon vertex. This is the vector returned by `pointVPGet`. See `pntVMap` for a partial list of vmap types.

Point and Polygon Info

The info and scan functions use EDPointInfo and EDPolygonInfo structures to provide information about points and polygons. Modeler maintains only one of each of these. It overwrites the structure each time data for a different point or polygon is required, so if you need to keep data for multiple points or polygons, you must copy it from the structure and store it locally.

```
typedef struct st_EDPointInfo {
   LWPntID  pnt;
   void    *userData;
   int      layer;
   int      flags;
   double   position[3];
   float   *vmapVec;
} EDPointInfo;
```

pnt

The ID of the point.

userData

Your per-point data buffer, allocated by the MeshEditBegin call.

layer

The layer in which the point resides.

flags

Flags for the point. The `EDDF_SELECT` bit is set if the selection state of the point matches the EltOpSelect passed to the MeshEditBegin function. The `EDDF_DELETE` bit is set if the point has been deleted during this mesh edit.

position

The point's position.

vmapVec

The vmap values associated with the point.

```
   typedef struct st_EDPolygonInfo {
      LWPolID        pol;
      void          *userData;
      int            layer;
      int            flags;
      int            numPnts;
      const LWPntID *points;
      const char    *surface;
      unsigned long  type;
   } EDPolygonInfo;
```

pol

The polygon ID.

userData

Your per-polygon data buffer, allocated by the MeshEditBegin call.

layer

The layer in which the polygon resides.

flags

Flags for the polygon. These include the `EDPF_CCSTART` and `EDDF_CCEND` bits for curves.

numPnts

The number of vertices in the polygon.

points

An array of point IDs for the vertices of the polygon.

surface

The polygon's surface.

type

The polygon type, which will usually be one of the following.

`LWPOLTYPE_FACE` - face
`LWPOLTYPE_CURV` - higher order curve
`LWPOLTYPE_PTCH` - subdivision control cage polygon
`LWPOLTYPE_MBAL` - metaball
`LWPOLTYPE_BONE` - bone

Example

The zfacing sample demonstrates `OPSEL_MODIFY` edits. This method of altering the selection is especially useful in CommandSequence plug-ins, so `zfacing.c` contains both edit and command versions of the activation function. The vidscape sample uses mesh editing to enumerate the geometry of an object before exporting it to a VideoScape format file. Many former mesh edit sample plug-ins, notably superq and spikey, have been converted to interactive mesh edit tools.

MeshEditTool

Availability LightWave 6.0 **Component** Modeler
Header lwmodtool.h, lwtool.h

Mesh edit tools are interactive versions of MeshDataEdit plug-ins. For users they behave like Modeler's built-in tools. You supply callbacks for drawing the tool, for creating "handles" that can be manipulated by the user, and for generating the geometry the tool creates or modifies.

Activation Function

```
   XCALL_( int ) MyMETool( long version, GlobalFunc *global,
      LWMeshEditTool *local, void *serverData );
```

The `local` argument to a mesh edit plug-in's activation function is an LWMeshEditTool.

```
   typedef struct st_LWMeshEditTool {
      LWInstance   instance;
      LWToolFuncs *tool;
      int          (*test)  (LWInstance);
      LWError      (*build) (LWInstance, MeshEditOp *);
      void         (*end)   (LWInstance, int keep);
   } LWMeshEditTool;
```

`instance`
: Set this to point to your instance data. Typically this is a structure that holds all of the data your plug-in needs to perform its function.

`tool`
: A set of tool callbacks you need to define. See below.

`action = `**`test`**`( inst )`
: Returns a code for the edit action that should be performed. The action can be one of the following.

`LWT_TEST_NOTHING`
: Do nothing. The edit state remains unchanged.

`LWT_TEST_UPDATE`
: Reapply the operation with new settings. The `build` function will be called.

`LWT_TEST_ACCEPT`

Keep the last operation. The `end` callback is called with a nonzero `keep` value.

`LWT_TEST_REJECT`
: Discard the last operation. The `end` callback is called with a `keep` value of 0.

`LWT_TEST_CLONE`
: Keep the last operation and begin a new one. The `end` callback is called with a nonzero `keep` value, and then the `build` function is called again.

The return value can also encode a memory size that will be allocated for each point and each polygon. These user memory sizes would be passed to the `begin` function of the MeshEditOp structure passed to MeshDataEdit plug-ins. For MeshEditTool class plug-ins, they are encoded in the value returned from `test` using the `LWT_VMEM` and `LWT_PMEM` macros for vertex and polygon sizes respectively.

`lwerr =` **`build`**`( inst, edit )`
: Perform the tool's mesh edit operation. A tool that creates a primitive would add the points and polygons of the primitive within this callback. The `edit` argument points to the same MeshEditOp structure passed to MeshDataEdit plug-ins.

`end``( inst, keep )`
: Clear the state when the last edit action is completed. This can be a result of a call to `test`, or it can be triggered by an external action.

Tool Functions

Your plug-in fills in an LWToolFuncs structure to tell Modeler where your tool callbacks are located.

```
typedef struct st_LWToolFuncs {
   void         (*done)   (LWInstance);
   void         (*draw)   (LWInstance, LWWireDrawAccess *);
   const char * (*help)   (LWInstance, LWToolEvent *);
   int          (*dirty)  (LWInstance);
   int          (*count)  (LWInstance, LWToolEvent *);
   int          (*handle) (LWInstance, LWToolEvent *, int i,
                             LWDVector pos);
   int          (*start)  (LWInstance, LWToolEvent *);
   int          (*adjust) (LWInstance, LWToolEvent *, int i);
   int          (*down)   (LWInstance, LWToolEvent *);
   void         (*move)   (LWInstance, LWToolEvent *);
   void         (*up)     (LWInstance, LWToolEvent *);
   void         (*event)  (LWInstance, int code);
```

```
      LWXPanelID   (*panel)  (LWInstance);
   } LWToolFuncs;
```

done(instance)

Destroy the instance. Called when the user discards the tool.

draw(instance, draw_access)

Display a wireframe representation of the tool in a 3D viewport using the drawing functions in the LWWireDrawAccess, described below.

helptext = **help**(instance, eventinfo)

Returns a text string to be displayed as a help tip for this tool.

dcode = **dirty**(instance)

Returns flag bits indicating whether the wireframe or the help string need to be refreshed. The bits are combined using bitwise-or. Return 0 if nothing needs to be refreshed, or any combination of the following.

```
LWT_DIRTY_WIREFRAME
LWT_DIRTY_HELPTEXT
```

nhandles = **count**(instance, eventinfo)

Returns the number of handles. A "handle" is a component of the tool's wireframe that the user can move independently. If this returns 0, then the `start` callback is used to set the initial handle point.

priority = **handle**(instance, eventinfo, hnum, pos)

Returns the 3D location and priority of handle `hnum`, or 0 if the handle is currently invalid.

hnum = **start**(instance, eventinfo)

Take an initial mouse-down position and return the index of the handle that should be dragged.

hdrag = **adjust**(instance, eventinfo, hnum)

Drag the handle to a new location. Returns the index of the handle that should continue being dragged (typically the same as `hnum`).

domouse = **down**(instance, eventinfo)

Process a mouse-down event. If this function returns 0, handle processing will be done instead of raw mouse event processing.

move(instance, eventinfo)

Process a mouse-move event. This will only be called if the `down` function returned a nonzero value.

up`( instance, eventinfo )`
: Process a mouse-up event. This will only be called if the `down` function returned a nonzero value.

event`( instance, code )`
: Process a general event indicated by one of the following codes.

`LWT_EVENT_DROP`
: The tool has been dropped. The user has clicked on an empty area of the interface, or pressed the spacebar, or selected another tool.

`LWT_EVENT_RESET`
: The user has requested that the tool return to its initial state. Numeric parameters should be reset to their default values.

`LWT_EVENT_ACTIVATE`
: The tool has been activated.

`xpanel = `**panel**`( instance )`
: Create an `LWXP_VIEW` xpanel for the tool instance.

Event Information

Most of the tool functions take an LWToolEvent as an argument.

```
   typedef struct st_LWToolEvent {
      LWDVector posRaw, posSnap;
      LWDVector deltaRaw, deltaSnap;
      LWDVector axis;
      LWDVector ax, ay, az;
      double    pxRaw, pxSnap;
      double    pyRaw, pySnap;
      int       dx, dy;
      int       portAxis;
      int       flags;
   } LWToolEvent;
```

posRaw, **posSnap**
: The event position in 3D space. The snap vector is the raw vector after quantizing to the nearest grid intersection in 3D.

deltaRaw, **deltaSnap**
: The vector from the initial mouse-down event to the current event location. This is just the difference between the initial and current positions.

`axis`

The event axis. All the points under the mouse location are along this axis through `pos`.

`ax`, `ay`, `az`

Screen coordinate system. `ax` points to the right, `ay` points up and `az` points into the screen. Movement by 1.0 along each vector corresponds to approximately one pixel of screen space movement.

`pxRaw`, `pxSnap`
`pyRaw`, `pySnap`

Parametric translation values. These are the mouse offsets converted to values in model space. They provide a method for computing abstract distance measures from left/right and up/down mouse movement roughly scaled to the magnification of the viewport..

`dx`, `dy`

Screen movement in pixels. This is the total raw mouse offset from the starting position.

`portAxis`

The view type. 0, 1 or 2 for orthogonal views, or -1 for perspective views.

`flags`

This contains flag bits assembled using bitwise-or. It can be some combination of the following.

`LWTOOLF_CONSTRAIN`

The action of the tool is constrained. Activated by a standard key or mouse combination.

`LWTOOLF_CONS_X`

`LWTOOLF_CONS_Y`

The direction of constraint for orthogonal moves. Initially neither bit is set, but as the user moves enough to select a primary direction, one or the other will be set.

`LWTOOLF_ALT_BUTTON`

Alternate mouse button event, usually the right button.

`LWTOOLF_MULTICLICK`

Multiple mouse click event.

Draw Access

The `draw` callback is given an LWWireDrawAccess containing a set of drawing functions for rendering the visual representation of the tool in the interface.

```
   typedef struct st_LWWireDrawAccess {
      void  *data;
      void  (*moveTo) (void *, LWFVector, int);
      void  (*lineTo) (void *, LWFVector, int);
      void  (*spline) (void *, LWFVector, LWFVector, LWFVector, int);
      void  (*circle) (void *, double, int);
      int    axis;
      void  (*text)   (void *, const char *, int);
      double pxScale;
   } LWWireDrawAccess;
```

`data`

An opaque pointer to data used by Modeler. Pass this as the first argument to the drawing functions.

`moveTo``( data, pos, line_style )`

Move the drawing point to the new position. Use this to set one endpoint of a line or a spline or the center of a circle. The third argument sets the line style for the drawing functions and can be one of the following.

```
LWWIRE_SOLID
LWWIRE_DASH
```

`lineTo``( data, pos, coord_type )`

Draw a line segment from the current drawing point to the given position. The coordinate type can be one of the following.

`LWWIRE_ABSOLUTE`

Absolute coordinates in model space.

`LWWIRE_RELATIVE`

Relative coordinates in model space. The `pos` argument is an offset from the current drawing point, which is the most recent position specified in a previous call to a drawing function.

`LWWIRE_SCREEN`

Relative coordinates in screen space. A distance of 1.0 in this coordinate system corresponds to about 20 pixels. Tool handles will typically be drawn in screen space, so that they remain the same displayed size regardless of the zoom level of the view.

`spline``( data, LWFVector, LWFVector, LWFVector, coord_type )`

Draw a curve from the current drawing point. The vectors are Bezier control points, with the current drawing point acting as the first of the required four points. When using relative coordinates, each position vector is an offset from the previous one.

`circle``( data, radius, coord_type )`
: Draw a circle centered at the current drawing point.

`axis`
: The view in which you're drawing. This can be 0, 1 or 2 for the *x*, *y* and *z* axis views, or -1 for a perspective view.

`text``( data, textline, justify )`
: Draw a single line of text. The `justify` argument positions the text relative to the current drawing point and can be one of the following.

```
LWWIRE_TEXT_L
LWWIRE_TEXT_C
LWWIRE_TEXT_R
```

`pxScale`
: The approximate size of a pixel in the current view.

History

In LightWave 7.0, the `text` function and the `pxScale` field were added to LWWireDrawAccess, but `LWMESHEDITTOOL_VERSION` was *not* incremented. If your activation accepts a version of 4, use the Product Info global to determine whether these items are available.

Example

The boxes/box4 sample is a simple example of a mesh edit tool. It's described in the Boxes tutorial. The mesh edit tool samples also include capsule, which creates a capsule shaped primitive, superq for making ellipsoidal and toroidal superquadrics, and spikeytool for adding spikes during subdivision.

ObjectLoader

Availability LightWave 6.0
Component Layout, Modeler
Header lwobjimp.h

Object loaders read object files that aren't in LightWave's native object file format.

When an object loader's activation function is called, it should open the object file and try to recognize its contents. LightWave calls all of the installed object loaders in sequence until one of them recognizes the file. Each object loader is therefore responsible for identifying the files it can load. If the file isn't one the loader understands, the loader sets the `result` field of the `local` structure to `LWOBJIM_NOREC` and returns `AFUNC_OK`.

If, on the other hand, the loader understands the object file, it reads the file and submits its contents to the host through the functions provided in the `local` structure.

Handler Activation Function

```
   XCALL_( int ) MyObjImport( long version, GlobalFunc *global,
      LWObjectImport *local, void *serverData );
```

The `local` argument to an object loader's activation function is an LWObjectImport.

```
   typedef struct st_LWObjectImport {
      int          result;
      const char  *filename;
      LWMonitor   *monitor;
      char        *failedBuf;
      int          failedLen;
      void        *data;
      void        (*done)    (void *);
      void        (*layer)   (void *, short int lNum, const char *name);
      void        (*pivot)   (void *, const LWFVector pivot);
      void        (*parent)  (void *, int lNum);
      void        (*lFlags)  (void *, int flags);
      LWPntID     (*point)   (void *, const LWFVector xyz);
      void        (*vmap)    (void *, LWID type, int dim,
                                 const char *name);
      void        (*vmapVal) (void *, LWPntID point, const float *val);
      LWPolID     (*polygon) (void *, LWID type, int flags, int numPts,
                                 const LWPntID *);
      void       (*polTag)  (void *, LWPolID polygon, LWID type,
                             const char *tag);
      void       (*surface) (void *, const char *, const char *, int,
                             void *);
      void       (*vmapPDV) (void *, LWPntID point, LWPolID polygon,
                             const float *val);
   } LWObjectImport;
```

result

Set this to indicate whether the object was loaded successfully. The result codes are

LWOBJIM_OK
: The object was loaded successfully.

LWOBJIM_NOREC
: The loader didn't recognize the file format. This can happen frequently, since all loaders are called in sequence until one of them *doesn't* return this result.

LWOBJIM_BADFILE
: The loader couldn't open the file. If the loader is able to open the file but believes it has found an error in the contents, it should return `IPSTAT_NOREC`.

LWOBJIM_ABORTED
: Use this to indicate that the user cancelled the load operation. This can happen if you use the monitor to indicate the progress of a lengthy image loading operation..

LWOBJIM_FAILED
: An error occurred during loading, for example a memory allocation failed.

filename

The name of the file to load.

monitor

A monitor for displaying the progress of the load to the user. You don't have to use this, but you're encouraged to if your object loading takes an unusual amount of time. This is the same structure as that returned by the monitor global's `create` function.

failedBuf
failedLen

These are used to display an error message to the user when object loading fails. Use `strcpy` or a similar function to copy a single-line error string into `failedBuf`. `failedLen` is the maximum size of this string, and it may be 0.

data
An opaque pointer to data used internally by LightWave during object loading. Pass this as the first argument to the loading functions.

done(data)
Call this when object loading is complete, after setting the `result` field.

layer(data, layernum, layername)
Create a new layer. All of the geometry you load will be put in this layer until you call `layer` again. Even if your object format doesn't support anything that could be interpreted as layers, this needs to be called at least once to initialize a layer that will receive your geometry. The layer name is optional and can be NULL. Layers are ordinarily created in increasing numerical order, starting at 1, but this isn't required.

pivot(data, pivpoint)
Set the pivot point for the current layer. The pivot point is the origin for rotations in the layer.

parent(data, layernum)
Set the parent layer for the current layer. Layer parenting is a mechanism for creating object hierarchies.

lFlags(data, layerflags)
Set flags for the current layer. The only flag currently defined is the low order bit, 1 << 0, which when set signifies that the layer is hidden.

pointID = **point**(data, pos)
Create a point in the current layer. Returns a point ID that can be used later to refer to the point in polygon vertex lists.

vmap(data, type, dim, name)

Create or select a vertex map. If the vmap doesn't exist, this function creates it. Otherwise it selects the vmap for subsequent calls to `vmapVal`. The lwmeshes.h header defines common vmap IDs, but you can create custom vmap types for special purposes.

`LWVMAP_PICK` - selection set
`LWVMAP_WGHT` - weight map
`LWVMAP_MNVW` - subpatch weight map
`LWVMAP_TXUV` - texture UV coordinates
`LWVMAP_MORF` - relative vertex displacement (morph)
`LWVMAP_SPOT` - absolute vertex displacement (morph)
`LWVMAP_RGB`, `LWVMAP_RGBA` - vertex color

The dimension of a vmap is just the number of values per point.

vmapVal(data, point, valarray)
Set the value of the current vmap for the point. The number of elements in the value array should be the same as the dimension of the vmap.

polID = **polygon**(data, type, flags, npoints, point_array)
Create a polygon in the current layer. The type will usually be one of the polygon types defined in lwmeshes.h.

`LWPOLTYPE_FACE` - face
`LWPOLTYPE_CURV` - higher order curve
`LWPOLTYPE_PTCH` - subdivision control cage polygon
`LWPOLTYPE_MBAL` - metaball
`LWPOLTYPE_BONE` - bone

The flags are specific to each type. The point array contains `npoints` point IDs returned by calls to the `point` function.

polTag(data, polygon, type, tag)
Associate a tag string with a polygon. A polygon's surface is set, for example, by passing `LWPTAG_SURF` as the type and the surface name as the tag. Note that you can do this without first having called `surface`.

surface(data, basename, refname, chunk_size, surf_chunk)

Set parameters for a surface. The base name is the name of the surface, while the reference name is the name of a "parent" surface (which can be NULL). Parameters not explicitly defined for the surface will be taken from its reference, or parent, surface. The surface data is passed as the memory image of a LightWave object file `SURF` chunk. See the object file format document for a detailed description of the contents of a `SURF` chunk.

Note that this method of specifying surface parameters doesn't allow you to associate envelopes or image textures with a surface. In the `SURF` chunk, envelopes and images are referenced by index into `ENVL` and `CLIP` chunks, respectively, and object loaders have no way to create these.

vmapPDV(data, point, polygon, valarray)
Like `vmapVal`, but sets the per-polygon, or discontinuous, vmap vector for a polygon vertex.

Example

The vidscape sample loads VideoScape ASCII object files. Several examples of this simple format (the files with `.geo` extensions) are included in the directory. The singular merit of the VideoScape format is that it's easy to write, even by hand, but in particular using languages like LScript, Perl and BASIC. The qbasic program that generated the `ball.geo` object is included. The vidscape sample also demonstrates the use of a MeshDataEdit plug-in to *save* objects in non-LightWave formats.

ObjReplacementHandler ObjReplacementInterface

Availability LightWave 6.0
Component Layout
Header lwobjrep.h

Object replacement handlers are called at each time step to decide whether Layout should use a different object file to represent an object. An object's geometry might be replaced depending on its camera distance (level of detail replacement), or a time (object sequence loading), or some other criterion (previewing versus rendering, for example).

Object replacement can be used in combination with ObjectLoaders to perform procedural object animation. The replacement plug-in might write a brief description file for the parameters of a time step, which the object import server would then convert into a complete mesh during loading.

Handler Activation Function

```
   XCALL_( int ) MyObjReplace( long version, GlobalFunc *global,
      LWObjReplacementHandler *local, void *serverData );
```

The `local` argument to an object replacement's activation function is an LWObjReplacementHandler.

```
   typedef struct st_LWObjReplacementHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      void            (*evaluate) (LWInstance, LWObjReplacementAccess *);
   } LWObjReplacementHandler;
```

The first two members of this structure are standard handler functions. The `context` argument to the `inst->create` function is the LWItemID of the item associated with this instance. An object replacement handler provides an evaluation function in addition to the standard handler functions.

evaluate(instance, access)
: This is where the object replacement happens. The access structure passed to this function contains information about the currently loaded object and the evaluation time. You compare these and

provide a new filename if a different object should be loaded. If the currently loaded geometry can be used for the new frame and time, set the new filename to NULL.

Interface Activation Function

```
XCALL_( int ) MyInterface( long version, GlobalFunc *global,
   LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Object Replacement Access

The access structure is the data passed to the handler's evaluation function. All of the fields of this structure are read-only except for `newFilename`.

```
typedef struct st_LWObjReplacementAccess {
   LWItemID    objectID;
   LWFrame     curFrame, newFrame;
   LWTime      curTime,  newTime;
   int         curType,  newType;
   const char *curFilename;
   const char *newFilename;
} LWObjReplacementAccess;
```

objectID
Item ID of the object.

curFrame, **curTime**
The frame number and time at which the currently loaded object file was most recently evaluated.

newFrame, **newTime**
The evaluation frame and time. If you provide a new filename, this is the time at which that object file will be loaded. Because of network rendering, the new frame and time may not follow the `curFrame` and `curTime` values sequentially.

curType, **newType**
These describe the current geometry and the type needed for the new time. An object replacement handler might ignore the time values and only perform replacements when the types differ. The type can be

LWOBJREP_NONE
: The current geometry for the object is a null object. This value only appears in `curType`.

LWOBJREP_PREVIEW
: The object will be used during previewing and user interaction with the interface.

LWOBJREP_RENDER
: The object will be used during rendering.

curFilename
: The filename of the currently loaded object file. This will be NULL if the `curType` is `LWOBJREP_NONE`.

`newFilename`
: If you want to replace the currently loaded object file, set this to the name of a different file. Set this to NULL if the object file shouldn't be changed. The memory that holds this string must persist after the evaluation function returns.

Example

The objseq sample lets the user select a list of files from a file dialog. It sorts the selected filenames and then replaces the object at frame 1 with the first file, at frame 2 with the second file, and so on.

PixelFilterHandler
PixelFilterInterface

Availability LightWave 6.0

Component Layout, Modeler
Header lwfilter.h

Pixel filters apply image processing effects to individual pixels in the rendered image.

Pixel filters look like image filters at first glance, but they differ in several significant ways. Pixel filters can modify any of the buffers, not just the red, green, blue and alpha values, and they have access to the raytracing functions. They're applied during rendering, before antialiasing and motion blur, so their effects are automatically accumulated by Layout for antialiasing and motion blur purposes.

Unlike image filters, which have access to the entire image and are called once per frame, pixel filters only evaluate, and only have access to, a single pixel sample at a time, and they can be called multiple times per pixel during the rendering of a frame.

Handler Activation Function

```
XCALL_( int ) MyPixelFilter( long version, GlobalFunc *global,
   LWPixelFilterHandler *local, void *serverData );
```

The `local` argument to a pixel filter's activation function is an LWPixelFilterHandler.

```
typedef struct st_LWPixelFilterHandler {
   LWInstanceFuncs *inst;
   LWItemFuncs     *item;
   LWRenderFuncs   *rend;
   void            (*evaluate) (LWInstance, const LWPixelAccess *);
   unsigned int    (*flags)    (LWInstance);
} LWPixelFilterHandler;
```

The first three members of this structure are the standard handler functions. In addition to these, a pixel filter provides an evaluation function and a flags function.

The `context` argument to the `inst->create` function is a pointer to an integer containing context flags. If the `LWFCF_PREPROCESS` flag is set, the instance is being created for an image other than the rendered output, and buffers other than the RGBA of the image won't be available.

A pixel filter can be activated by both Layout and Modeler. When activated by Modeler, the LWItemFuncs pointer in the local data is NULL. Be sure to test for this before filling in the `useItems` and `changeID` fields. Note too that if your pixel filter relies on Layout-only globals, those won't be available when Modeler calls your callbacks.

evaluate`( instance, access )`

> This is where the pixel filter does its work. For each frame, the filter is given access to the red, green, blue and alpha values of each pixel sample, along with any other pixel data requested by the flags function. The access structure, described below, provides pixel information and functions for examining the buffers and writing new values.

flags`( instance )`

> Returns an int that tells the renderer which buffers the pixel filter will examine and/or modify and whether the evaluation function will call one of the raytracing functions in the access structure. The return value contains bitfields combined using bitwise-or. See the image filter page for a list of the buffer codes. In addition to these, the `LWPFF_RAYTRACE` flag indicates that the evaluation function will call the raytracing functions, and the `LWPFF_EVERYPIXEL` flag indicates that the filter should be evaluated for every pixel, despite adaptive sampling settings.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Pixel Access

The pixel access structure passed to the evaluation function contains the pixel coordinates for the sample, functions for getting and setting pixel values, and the raytracing functions. Because the sampling of the output image is adaptive, pixel positions may be evaluated in any order, multiple times, or not at all. The evaluation function must call `setRGBA` for every pixel it evaluates, even if the filter doesn't modify the pixel.

```
   typedef struct st_LWPixelAccess {
      double            sx, sy;
      void             (*getVal)  (int type, int num, float *);
      void             (*setRGBA) (const float[4]);
      void             (*setVal)  (int type, int num, float *);
      LWIlluminateFunc *illuminate;
      LWRayTraceFunc   *rayTrace;
      LWRayCastFunc    *rayCast;
      LWRayShadeFunc   *rayShade;
   } LWPixelAccess;
```

sx, **sy**

Image coordinates of the sample, in pixel units. These will often contain fractional values.

getVal(type, buflen, buf)

Get a pixel value from one of the buffers. If the buffer type is invalid or a type not requested by the flags function, the pixel value returned in `buf` is undefined. See the image filter page for the list of buffer types. `buflen` is the number of contiguous values to return. For most buffers, this number will be 1, but the RGB buffers can be retrieved all at once. With a `type` of `LWBUF_RAW_RED`, for example, the number can be up to 3 to get `RAW_RED`, `RAW_GREEN` and `RAW_BLUE`, and for `LWBUF_RED` it can be up to 4, for the RGBA values.

setRGBA(rgba)

The RGBA (red, green, blue and alpha) output of the pixel filter. This must be called even if the filter doesn't modify the values.

setVal(type, buflen, buf)

Write a value to one of the buffers.

lit = **illuminate**(lightID, position, direction, color)

len = **rayTrace**(position, direction, color)

len = **rayCast**(position, direction)
len = **rayShade**(position, direction, shaderAccess)

These functions trace rays into the scene. See the raytracing functions page for details. You can only use these if the return value of your flags function includes the `LWPFF_RAYTRACE` flag.

Example

The zcomp sample includes a pixel filter that composites the render with an image sequence using the `LWBUF_DEPTH` buffer. zcomp compares the depth at each pixel with the corresponding depth in the image to be composited, and substitutes the image pixel if it's nearer in z order to the camera.

The mandfilt sample turns LightWave into a Mandelbrot set renderer. Unlike most real pixel filters, it simply overwrites the pixel values with its own output, so it should be run in an empty scene. But it does demonstrate how pixel filter output is antialiased and adaptively sampled by LightWave.

ProceduralTextureHandler
ProceduralTextureInterface

Availability LightWave 6.0
Component Layout, Modeler
Header lwtexture.h

Fundamentally, a procedural texture is a function of three variables. In other words, given three numbers *x*, *y* and *z*, a procedural texture calculates and returns a fourth number *t*.

The variables are usually the coordinates of a 3D position, either in world space or in some idealized texture space, and the number returned by the function is used to modulate a rendering parameter, typically one of the surface attributes. LightWave's built-in fractal noise is an example of a procedural texture.

Some procedural textures will also return a *gradient*. Roughly speaking, this is the direction of the texture at the sample point, or the direction in which the value of the texture function increases the fastest. If the texture is being used as a bump map, the renderer can infer a bump normal from the gradient.

If your texture function is analytical, you can compute the gradient from the partial derivative of the function with respect to each axis. You aren't required to form the gradient that way, or at all. If the texture doesn't return a gradient, the renderer will calculate a numerical approximation by calling your texture function at six neighboring points.

Textures can also return a color. This is useful when the texture will be applied to the color channel of a surface or will modulate some other color-valued parameter.

Handler Activation Function

```
   XCALL_( int ) MyTexture( long version, GlobalFunc *global,
      LWTextureHandler *local, void *serverData );
```

The `local` argument to a texture's activation function is an

LWTextureHandler.

```
   typedef struct st_LWTextureHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      LWRenderFuncs   *rend;
      double          (*evaluate) (LWInstance, LWTextureAccess *);
      unsigned int    (*flags)    (LWInstance);
   } LWTextureHandler;
```

The first three members of this structure are the standard handler functions. In addition to these, a procedural texture provides an evaluation function and a flags function.

The `context` argument to the `inst->create` function is the LWTextureID for the texture. LWTextureID is defined in the `lwtxtr.h` header file and is used by the texture functions and texture editor globals.

A procedural texture can be activated by Modeler as well as Layout. When activated by Modeler, the LWItemFuncs pointer in the local data is NULL. Be sure to test for this before filling in the `useItems` and `changeID` fields. Note too that if your texture relies on Layout-only globals, those won't be available when Modeler calls your callbacks.

`txval = **evaluate**( instance, access )`

Returns a texture value, given the information in the access structure, described below.

`flagbits = **flags**( instance )`

Returns an int that tells LightWave about the texture. The return value can be any of the following flags combined using bitwise-or.

`LWTEXF_GRAD`

The texture returns a gradient (the evaluation function sets the value of the `txGrad` member of the access structure). If this flag isn't set, the texture engine ignores `txGrad` and, when necessary, calculates the gradient numerically (by evaluating 6 neighboring points).

`LWTEXF_SLOWPREVIEW`

Set this if the texture evaluates too slowly to be previewed in real time.

`LWTEXF_AXIS`

The texture uses an axis. The texture editor will allow the user to select an axis for the texture, and this selection will be found in the `axis` member of the access structure.

`LWTEXF_AALIAS`
: The texture value is already antialiased. Currently ignored, but it may not be in the future.

`LWTEXF_DISPLACE`
: Use the texture value for displacements. If this flag is set, the texture editor's axis selector is enabled and the displacement occurs along the selected axis. If this flag isn't set, but `LWTEXF_GRAD` is, the texture engine will use the gradient for displacements. If neither flag is set, no displacement will occur.

LWTEXF_HV_SRF
: The texture is appropriate for use as a HyperVoxels surface texture. This basically means that the texture function is continuous and evaluates relatively quickly.

LWTEXF_HV_VOL
: The texture is appropriate for use as a HyperVoxels volume texture. Efficiency is especially important for these textures.

LWTEXF_SELF_COLOR
: The texture returns an RGBA color in addition to a value.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Texture Access

The access structure passed to the evaluation function contains parameters that can affect the texture value. The texture can return a gradient and a color through the `txGrad` and `txRGBA` fields. The other fields are read-only.

```
   typedef struct st_LWTextureAccess {
      double  wPos[3];
      double  tPos[3];
      double  size[3];
      double  amp;
      double  spotSize;
      double  txGrad[3];
      int     axis;
      int     flags;
      double  octaves;
      double  txRGBA[4];
   } LWTextureAccess;
```

`wPos`
: The world coordinate position of the sample to be textured.

`tPos`
: The position of the sample in texture coordinates.

`size`
: The size of the texture. The size value is used to scale the texture spatially. The interpretation is up to the texture, but typically this is the size of a texture cell or the distance between repeating elements.

`amp`
: The amplitude of the texture. This value is typically used to scale the magnitude or strength of the texture.

`spotSize`
: The approximate diameter of the sample spot. This is useful when antialiasing the texture.

`txGrad`
: Storage for the texture gradient at the sample. The evaluation function must fill this in when the flags function returns `LWTEXF_GRAD`. Otherwise it can be ignored.

`axis`
: The texture axis selected by the user. Only valid if the flags function set the `LWTEXF_AXIS` or `LWTEXF_DISPLACE` flags.

`flags`
: The access flags provide information about the context in which the evaluation function was called.

  LWTXEF_VECTOR
  : Set when a bump is being evaluated.

  LWTXEF_AXISX
  LWTXEF_AXISY
  LWTXEF_AXISZ
  : Which dimensions are used for evaluation. Currently, all three of these are always set, but in the future, the texture engine might evaluate the texture in 2D only, for example, and it would use these flags to allow the texture to switch to an evaluation optimized for 2D.

  LWTXEF_DISPLACE
  : Set when a displacement is being evaluated.

  LWTXEF_COLOR

Set when a color is being evaluated.

octaves
The number of octaves, or frequencies, that should be used by the texture. This affects the amount of structure the texture generates at different scales. This field is currently only initialized by HyperVoxels.

txRGBA
Storage for the texture color at the sample. The evaluation function must fill this in when the flags function returns `LWTEXF_SELF_COLOR`. Otherwise it can be ignored..

Example

The rapts sample contains 10 procedural textures.

SceneConverter

Availability LightWave 6.0
Component Layout
Header lwscenecv.h

Scene converters load scene files written in formats other than LightWave's native format.

When the user selects a scene file to load, Layout first tries to load it directly as a LightWave format file. If it can't, it passes the filename to each installed scene converter until one of them claims to recognize the file. The scene converter reads the file and rewrites it as a LightWave scene file, passing the name of this temporary file back to Layout. After loading this file, Layout calls the scene converter's `deleteTmp` function to remove it.

Activation Function

```
   XCALL_( int ) MySceneConvert( long version, GlobalFunc *global,
      LWSceneConverter *local, void *serverData );
```

The `local` argument to a scene converter's activation function is an LWSceneConverter.

```
   typedef struct st_LWSceneConverter {
      const char *filename;
      LWError     readFailure;
      const char *tmpScene;
      void       (*deleteTmp) (const char *tmpScene);
   } LWSceneConverter;
```

filename

The name of the non-native scene file. This is the file to be converted.

readFailure

A one-line error message. Set this if you recognize the file format but can't read the file for some reason. If you *don't* recognize the format, leave this and the `tmpScene` and `deleteTmp` fields NULL. This tells Layout to submit the file to the next installed converter.

tmpScene

The filename of the temporary LightWave-format scene file created by the scene converter. If you successfully create this temporary file, you should also provide a valid `deleteTmp` callback. If an error occurs during the conversion, you should remove the partially written temporary file, set `tmpScene` to NULL, and set the `readFailure` field to an error message. Since this tells Layout to stop trying to load the file, you should be careful to distinguish between files you don't recognize and those you do but which contain errors. If you're not sure, leave the `readFailure` field NULL so that other converters have a chance to try to load the file.

deleteTmp(filename)

A function you provide for removing the `tmpScene` file you create. Layout calls this after reading the `tmpScene` file.

Example

Most scene converters will follow the pattern shown in this pseudocode. Note that rather than write our own `deleteTmp` function for removing the temporary LightWave scene file, we just pass back the C runtime `remove` function.

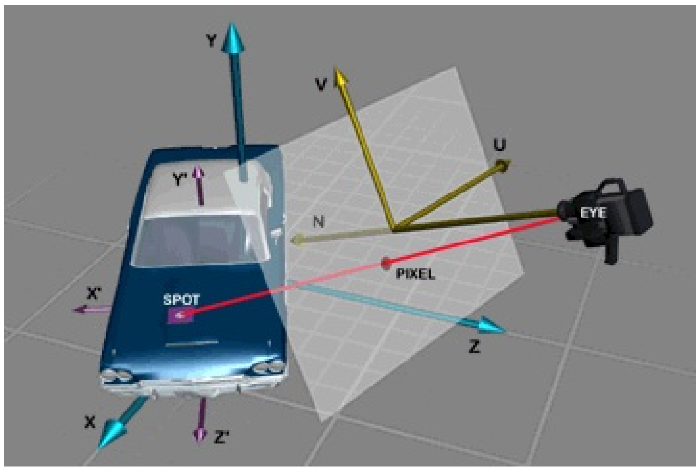
```
#include <lwserver.h>
#include <lwscenecv.h>
#include <stdio.h>
#include <stdlib.h>

XCALL_( int )
MySceneConvert( long version, GlobalFunc *global,
   LWSceneConverter *local, void *serverData )
{
   static char tempfile[ 260 ];
   FILE *ifp, *ofp;

   ifp = fopen( local->filename, "rb" );
   if ( !ifp ) return AFUNC_BADLOCAL;

   ... read some of the file ...

   if ( not our format )
      return AFUNC_OK;

   ofp = fopen( tempfile, "w" );
   if ( !ofp ) {
      fclose( ifp );
      local->readFailure = "Couldn't create temp scene file.";
      return AFUNC_OK;
   }

   ... convert the scene ...

   if ( error while converting ) {
      fclose( ifp );
      fclose( ofp );
      local->readFailure = "Error while converting.";
      return AFUNC_OK;
   }

   /* successful */

   local->tmpScene = tempfile;
   local->deleteTemp = remove;
   return AFUNC_OK;
}

ServerRecord ServerDesc[] = {
   { LWSCENECONVERTER_CLASS, "MySceneConverter", MySceneConvert },
   { NULL }
};
```

ShaderHandler ShaderInterface

Availability LightWave 6.0
Component Layout, Modeler
Header lwshader.h

Shaders set the color and other appearance attributes of each visible spot on a surface.

A *surface*, sometimes called a *material* in other programs, is a collection of attributes that define the appearance of a polygon. The same surface can be applied to multiple objects, and different surfaces can be applied to different polygons on the same object. Shaders are always associated with a surface and affect its appearance during rendering by setting or modifying its attributes. More than one shader can be associated with a surface, and the effects of one shader might in turn be modified by the next shader in line. A shader can also fire rays into the scene that cause other shaders to be evaluated.



For each pixel in the rendered image, the renderer finds the spot in the scene that the camera sees at that pixel. If the spot is on an object, its appearance depends on the suface assigned to the polygon it lies in. The renderer uses the surface settings to calculate a color for the pixel, and if a shader is attached to the surface, its evaluation function is called to either modify the surface settings or perform its own color calculation.

The shader evaluation function is given the surface attributes, the geometry of the spot, the ID of the object and the polygon, the source of the viewing ray, and other contextual information, and it has access to raytracing functions that can tell it even more about the scene.

Handler Activation Function

```
   XCALL_( int ) MyShader( long version, GlobalFunc *global,
      LWShaderHandler *local, void *serverData );
```

The `local` argument to a shader's activation function is an LWShaderHandler.

```
   typedef struct st_LWShaderHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      LWRenderFuncs   *rend;
      void            (*evaluate) (LWInstance, LWShaderAccess *);
      unsigned int    (*flags)    (LWInstance);
   } LWShaderHandler;
```

The first three members of this structure are the standard handler functions. In addition to these, a shader provides an evaluation function and a flags function.

The `context` argument to the `create` function is the LWSurfaceID for the surface. LWSurfaceID is defined in the `lwsurf.h` header file and is used by the surface functions global.

A shader can be activated by Modeler as well as Layout. When activated by Modeler, the LWItemFuncs pointer in the local data is NULL. Be sure to test for this before filling in the `useItems` and `changeID` fields. Note too that if your shader relies on Layout-only globals, those won't be available when Modeler calls your callbacks.

`evaluate``( instance, access )`

> This is where the shader does its work. At each time step in the animation, the evaluation function is called for each pixel affected by the shader's surface. The access argument, described below, contains information about the spot to be colored.

`flagbits = `**`flags`**`( instance )`

> Returns an int that tells the renderer which surface attributes the shader will modify and whether it will call the raytracing functions. The flags are bits combined using bitwise-or. They correspond to members of the shader access structure described below. For

efficiency reasons, the renderer may ignore changes to any surface attributes that weren't indicated by the bit flags returned from this function, and it won't provide access to the raytracing functions unless the `LWSHF_RAYTRACE` bit is set. The flags are

LWSHF_NORMAL
LWSHF_COLOR
LWSHF_LUMINOUS
LWSHF_DIFFUSE
LWSHF_SPECULAR
LWSHF_MIRROR
LWSHF_TRANSP
LWSHF_ETA
LWSHF_ROUGH
LWSHF_TRANSLUCENT
LWSHF_RAYTRACE

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers. Inline xpanels will appear in the Shaders tab of the Surface Editor.

Shader Access

The evaluation function is called for every visible spot on a surface and is passed a shader access structure describing the spot to be shaded. The access structure contains some values which are read-only and some which are meant to be modified. The read-only values describe the geometry of the pixel being shaded. The read-write values describe the current attribute settings of the spot and should be modified in place to affect the final look of the spot. Since shaders may be layered, these properties may be altered many more times before final rendering. The access structure also contains raytracing functions that can be called only while rendering.

```
   typedef struct st_LWShaderAccess {
      int               sx, sy;
      double            oPos[3], wPos[3];
      double            gNorm[3];
      double            spotSize;
      double            raySource[3];
      double            rayLength;
      double            cosine;
      double            oXfrm[9], wXfrm[9];
      LWItemID          objID;
      int               polNum;
      double            wNorm[3];
      double            color[3];
      double            luminous;
      double            diffuse;
      double            specular;
      double            mirror;
      double            transparency;
      double            eta;
      double            roughness;
      LWIlluminateFunc *illuminate;
      LWRayTraceFunc   *rayTrace;
      LWRayCastFunc    *rayCast;
      LWRayShadeFunc   *rayShade;
      int               flags;
      int               bounces;
      LWItemID          sourceID;
      double            wNorm0[3];
      double            bumpHeight;
      double            translucency;
      double            colorHL;
      double            colorFL;
      double            addTransparency;
      double            difSharpness;
      LWPntID           verts[4];
      float             weights[4];
      float             vertsWPos[4][3];
      LWPolID           polygon;
      double            replacement_percentage;
      double            replacement_color[3];
      double            reflectionBlur;
      double            refractionBlur;
   } LWShaderAccess;
```

Read-Only Parameters

These fields provide read-only information about the local geometry of the spot and the context of the evaluation.

sx, **sy**

The pixel coordinates at which the spot is visible in the rendered image. This is labeled **PIXEL** in the figure, but note that it won't necessarily be the spot's projection onto the viewplane. When the viewing ray originates on a reflective surface, for example, the pixel coordinates are usually for the source of the ray (the spot's reflection). The pixel coordinate origin (0, 0) is in the upper left corner of the image.

oPos

Spot position in object (Modeler) coordinates (the (**X'**, **Y'**, **Z'**) system in the figure).

`wPos`

Spot position in world coordinates (**X**, **Y**, **Z**). This is the position after transformation and the effects of bones, displacement and morphing.

`gNorm`

Geometric normal in world coordinates. This is the raw polygonal normal at the spot, unperturbed by smoothing or bump mapping.

`wNorm0`

The interpolated normal in world coordinates. This is the same as `gNorm`, but after smoothing.

`spotSize`

Approximate spot diameter, useful for texture antialiasing. The diameter is only approximate because spots in general aren't circular. On a surface viewed on edge, they're long and thin.

`raySource`

Origin of the incoming viewing ray in world coordinates. Labeled **EYE** in the figure, this is often the camera, but it can also, for example, be a point on a reflective surface.

`rayLength`

The distance the viewing ray traveled in free space to reach this spot (ordinarily the distance between `raySource` and `wPos`).

`cosine`

The cosine of the angle between the raw surface normal and a ray pointing from the spot back toward the `raySource`. This is the same as the dot product of `gNorm` and the unit vector `(raySource - wPos)/rayLength`. Low values correspond to high angles and therefore glancing views. This is also a measure of how approximate the spot size is.

`oXfrm`

Object to world transformation matrix. The nine values in this array form a 3 x 3 matrix that describes the rotation and scaling of the object. This is useful primarily for transforming direction vectors (bump gradients, for example) from object to world space.

```
LWDVector ovec, wvec;
wvec[ 0 ] = ovec[ 0 ] * oXfrm[ 0 ]
          + ovec[ 1 ] * oXfrm[ 1 ]
          + ovec[ 2 ] * oXfrm[ 2 ];
```

```
        wvec[ 1 ] = ovec[ 0 ] * oXfrm[ 3 ]
                  + ovec[ 1 ] * oXfrm[ 4 ]
                  + ovec[ 2 ] * oXfrm[ 5 ];
        wvec[ 2 ] = ovec[ 0 ] * oXfrm[ 6 ]
                  + ovec[ 1 ] * oXfrm[ 7 ]
                  + ovec[ 2 ] * oXfrm[ 8 ];
```

`wXfrm`

World to object transformation matrix (the inverse of `oXfrm`).

`objID`

The object being shaded. It's possible for a single shader instance to be shared between multiple objects, so this may be different for each call to the shader's evaluation function. For sample sphere rendering the ID will refer to an object not in the current scene.

`polNum`

An index identifying the polygon that contains the spot. It may represent other sub-object information in non-mesh objects. See also the `polygon` field.

`flags`

Bit fields describing the nature of the call. The `LWSAF_SHADOW` bit tells you when the evaluation function is being called during shadow computations, which you might want to treat differently from "regular" shader evaluation.

`bounces`

The number of times the viewing ray has branched, or bounced, before reaching this spot. This value can be used to limit recursion, particularly the shader's own calls to the raytracing functions.

`sourceID`

The item ID of the source of the viewing ray.

`verts`

The four vertices surrounding the spot, useful for interpolating vertex-based surface data.

`weights`

The weights assigned to the four neighboring vertices.

vertsWPos

The world coordinate position of the neighboring vertices.

polygon

The polygon ID of the polygon containing the spot.

Modifiable Parameters

These parameters are used by the renderer to compute the perceived color at the spot and may be modified by the shader. Almost all of them correspond directly to surface parameters in the user interface, although the values may be represented by different ranges. Unless stated otherwise, the values of these fields nominally range from 0.0 to 1.0, and values outside that range are also valid.

The shader's flags function must have returned the correct flags for the fields the shader will modify, or changes to these fields may be ignored. Prior to LightWave 7.0, to set the perceived color directly a shader would set all of the parameters to zero except for `luminous`, which would be 1.0, and `color`, which would be the output color of the spot. Newer shaders can instead use the `replacement_color` and `replacement_percentage` fields.

`wNorm`
: Surface normal in world coordinates. If you modify this vector, you must renormalize it (make its length equal to 1.0).

`bumpHeight`
: The relative height of peaks in the bump map. Increasing this value makes the bump mapping appear more pronounced. Currently this is used solely as a texture input. If no texture is applied to the bump channel of the surface, it will be 0, and values you write will be ignored.

`color`
: The RGB components of the base color of the spot.

`luminous`
: Luminosity level. Higher values add more of the base color to the final color of the spot. This component is unaffected by lighting and shading.

`diffuse`
: Diffuse reflection level.

`specular`
: Specular reflection level.

`mirror`
: Mirror reflection level.

`transparency`

Transparency level.

`eta`

Index of refraction. In the real world this ranges between 1.0 and about 3.5, depending on the material, but values outside that range are also valid here.

`roughness`

Surface roughness, the inverse of the exponent in the Phong specular highlight formula. The corresponding user parameter is called Glossiness. The roughness is approximately $2^{(10g+2)}$.

`translucency`

Translucency level. This determines how much the brightness of a surface is affected by the brightness of the environment behind it.

`colorHL`

Amount of highlight coloring. Higher values mix more of the base color of the spot into the color of specular highlights, a simple way to simulate the behavior of metallic (nondielectric) surfaces.

`colorFL`

Color filtering amount. This controls how strongly the light passing through a transparent surface is colored by that surface, which affects the color of other surfaces illuminated by this light.

`addTransparency`

Additive transparency. An additively transparent surface adds its own color to the colors of surfaces seen through it. This usually has the effect of lightening the color of the underlying surfaces.

`difSharpness`

Diffuse sharpness level. This controls how the shading varies with the angle of the light. Higher values make the brightness of illuminated areas more uniform and increase the sharpness of the transition between lit and dark areas (the terminator of a planet, for example).

replacement_percentage
replacement_color

Use these together to set a color for the surface that will be unaffected by subsequent shading and lighting calculations. This is typically used by plug-ins that partially replace LightWave's lighting model.

reflectionBlur
refractionBlur

The amount of blurring applied to reflections and refractions.

Rendering Functions

```
lit = illuminate( lightID, position, direction, color )
len = rayTrace( position, direction, color )
len = rayCast( position, direction )
len = rayShade( position, direction, shaderAccess )
```

See the raytracing functions page for a description of these.

History

LightWave 7.0 added the `replacement_percentage`, `replacement color`, `reflectionBlur` and `refractionBlur` fields to LWShaderAccess, but `LWSHADER_VERSION` was *not* incremented. If your shader activation accepts version 4, use the Product Info global to determine whether these fields are available before attempting to read or write them.

Example

The blotch sample is a simple shader that renders a circular blotch of a user-specified position, size and color.

VolumetricHandler

Availability LightWave 6.0
Component Layout
Header lwvolume.h

Volumetric handlers model the attenuation and scattering of light in gases, differences in density in 3D medical imaging data, or the shapes of surfaces too complex to model explicitly with geometry. They do this by participating in LightWave's raytracing mechanism.

For each ray fired into the scene, the volumetric handler calculates a color and opacity for one or more samples. It hands each sample back to LightWave, which integrates all of the samples from all of the volumetrics to produce the final color seen from the source of the ray.

A *sample* represents a segment of the ray over which the color and opacity are constant. Consider a simple cloud pierced by a ray. The handler that draws the cloud isn't interested in the parts of the ray that are outside it, so it creates no samples there. In the simplest case, it may create a single sample that begins at the point where the ray enters the cloud and extends as far as the ray remains inside. If the cloud is somewhat transparent, the color might be a linear combination of the cloud color and the backdrop color, and the opacity will be somewhat less than 1.0.

Handler Activation Function

```
   XCALL_( int ) MyVolumetric( long version, GlobalFunc *global,
      LWVolumetricHandler *local, void *serverData );
```

The `local` argument to a volumetric handler's activation function is an LWVolumetricHandler.

```
   typedef struct st_LWVolumetricHandler {
      LWInstanceFuncs *inst;
      LWItemFuncs     *item;
      LWRenderFuncs   *rend;
      double          (*evaluate) (LWInstance, LWVolumeAccess *);
      unsigned int    (*flags)    (LWInstance);
   } LWVolumetricHandler;
```

The first three members of this structure point to the standard handler functions. In addition to these, a volumetric handler provides an evaluation function and a flags function.

`d = **evaluate**( instance, access )`

Called for each ray fired into the scene. The evaluation function adds zero or more samples to the ray, based on the information in the access structure, described below.

`f = **flags**( instance )`

Returns an int that tells the renderer which effects the handler supplies. The return value contains bitfields combined using bitwise-or.

LWVOLF_SHADOWS
: The evaluation function can be called for shadow rays.

LWVOLF_REFLECTIONS
: Can be evaluated for reflection rays.

LWVOLF_REFRACTIONS
: Can be evaluated for refraction rays.

Interface Activation Function

```
   XCALL_( int ) MyInterface( long version, GlobalFunc *global,
      LWInterface *local, void *serverData );
```

This is the standard interface activation for handlers.

Volumetric Access

This is the structure passed to the handler's evaluation function.

```
   typedef struct st_LWVolumeAccess {
      void             *ray;
      int               flags;
      LWItemID          source;
      double            o[3], dir[3];
      double            rayColor[3];
      double            farClip, nearClip;
      double            oDist, frustum;
      void             (*addSample) (void *ray, LWVolumeSample *smp);
      double           (*getOpacity)(void *ray, double dist,
                                     double opa[3]);
      LWIlluminateFunc *illuminate;
      LWRayTraceFunc   *rayTrace;
      LWRayCastFunc    *rayCast;
      LWRayShadeFunc   *rayShade;
   } LWVolumeAccess;
```

ray

An opaque pointer to LightWave's representation of the ray. Pass this as the first argument to the `addSample` and `getOpacity` functions.

`flags`

Evaluation flags. Some of these allow the evaluation function to streamline its calculations.

LWVEF_OPACITY
: Calculate an opacity value for each sample. When this flag is absent, the opacity calculation can be omitted.

LWVEF_COLOR
: Calculate a color for each sample.

LWVEF_RAYTRACE
: If this flag is absent, the evaluation function is being called during the volumetric pass that occurs before pixel filtering but after normal rendering. Otherwise the evaluation function is being called during "regular" raytracing (reflection, refraction or shadow rays, for example).

`source`

The item from which the ray originated. This can be a light (for shadow rays), a camera, or `LWITEM_NULL` for other sources.

`o, dir`

The origin and direction of the ray. The origin is the position of the source item or of a spot on the surface of the source object.

`far, near`

Far and near clipping distances. These are distances along the ray measured from the origin `o` along the direction `dir`. All sample segments will normally fall between these two.

`rayColor`

The color seen from the origin of the ray, before volumetric effects are applied.

`oDist`

Distance from the origin `o` to the true start of the viewing path. This is non-zero for reflection and refraction rays. If the origin `o` is a spot on the surface of an object, `oDist` is the distance from that spot to the camera. This is good to know if your calculations will be based on the length of the path to the viewer (the camera) and not just on the length of the ray fired from the spot on the object.

`frustum`

Pixel frustum, equal to `2 * tan(0.5 * hfov)/w`, where `hfov` is the horizontal field of view and `w` is the width of the rendered image in pixels. The frustum is a measure of the size of a pixel relative to the ray. (It's the actual size of the pixel at a distance of 1.0.) This quantity plays a role in calculating sample size, or *stride*, during raymarching. A typical calculation of the stride might look like

```
increment = scale_factor * frustum;
stride = dist * increment;
```

`addSample( ray, sample )`

Add a new volume sample to the ray. This is how volumetric handlers submit their contributions to the integration of opacity and color along the ray. The sample structure is described below.

`opacity = getOpacity( ray, dist, opa )`

Returns the currently calculated opacity (vector and scalar) at the specified distance.

```
lit = illuminate( lightID, position, direction, color )
len = rayTrace( position, direction, color )
len = rayCast( position, direction )
len = rayShade( position, direction, shaderAccess )
```

See the raytracing functions page for a description of these.

Volume Sample

A volume sample is a single ray segment with a uniform color and opacity. The distance and stride define the position and size of the sample, and the opacity and color are given as color vectors. By the way, you can create surface samples by setting `stride` to 0 and `dist` to `0.9999 * farClip`.

```
   typedef struct st_LWVolumeSample {
      double  dist;
      double  stride;
      double  opacity[3];
      double  color[3];
   } LWVolumeSample;
```

dist

The starting point of the sample expressed as a distance from the origin of the ray. This should be greater than or equal to `nearClip`.

stride

The length of the sample. `dist + stride` should be less than or equal to `farClip`.

opacity

The red, green and blue components of the opacity of this sample.

color

The color at this sample.

Example

The atmosphere sample is a straightforward implementation of some of the volumetric techniques discussed in chapter 14 (K. Musgrave, L. Gritz, S. Worley) of *Texturing and Modeling*, 2nd ed., Academic Press, 1998. It includes both a fast analytical solution that creates a single sample and a more refined solution that uses raymarching and multiple samples per ray.

Common Commands

These commands are available in both Layout and Modeler. They're handled by a set of editor modules that are common to both programs. Although the editors are represented to the user as separate windows, you can issue most of these commands regardless of whether the editor's window is open.

In the command list that follows, the data types of the arguments are denoted by the initial letter. (The same letters are used in C `printf` formatting.)

number
: An integer.

gfloat
: A floating-point number.

string
: A string, such as a filename or channel name.

he**x**
: A hexadecimal number, such as an item identifier.

Surface Editor

The Surface Editor is a window for setting and changing surface parameters. These commands are a counterpart to the Surface Functions global.

Surf_OpenWindow
Surf_CloseWindow
Surf_SetWindowPos **n**x **n**y

Open and close the Surface Editor window, and set its position. In Layout, `SurfaceEditor` is a synonym for `Surf_OpenWindow` and `Surf_CloseWindow`.

Surf_SetSurf **s**surfname **s**objname

Set the current surface in the Surface Editor. All of the surface commands that modify surface parameters operate on the current surface. The object name is the name returned by the Item Info `name` function in Layout and the State Query `object` function in Modeler.

Surf_SetInt **s**channel **n**value

Set the integer value associated with the surface channel. Use this to add or remove an envelope or a texture (equivalent to pressing the E or T buttons in the interface). The channel name can be any of the strings defined in lwsurf.h. The value is one of the following.

0 - no envelope, no texture
1 - envelope, no texture
2 - texture, no envelope
3 - envelope and texture

Once you've added an envelope, you can create keys for it and manipulate it in other ways using the Animation Envelopes global. Use the Texture Functions global to modify the texture. You can get the envelope and texture IDs you'll need from the Surface Functions global. Getting the IDs also allows you to check whether an envelope or a texture exists before issuing this command.

Surf_SetFloat **s**channel **g**value

Set the value of a float-valued channel.

Surf_SetColor **s**channel **g**red **g**green **g**blue

Set the color of a color-valued channel.

Surf_AddShader **s**shader
Surf_RemShader **s**shader

Add or remove a shader. The argument is the shader's server name, the string in the `name` field of the shader's ServerRecord.

Surf_Rename **s**name

Rename the current surface.

`Surf_Copy` **s**`surfname` **s**`objname`

Copy the settings of the current surface (the one set by `Surf_SetSurf`) to the surface specified in the arguments.

`Surf_SetBakerImage` **s**`filename`

Set the base filename for the image file output of the Surface Baker shader.

Image Editor

The Image Editor is a window for managing LightWave's list of images, image sequences and animation files. See also the Image List global.

IE_OpenWindow
IE_SetWindowPos **n**x **n**y

Open the Image Editor window, and set its position. In Layout, `ImageEditor` is a synonym for `IE_OpenWindow`.

Graph Editor

The Graph Editor is a window for editing parameters that vary with time. These functions of time are called *channels*. The function is assigned a value at specific times, and the (value, time) pair is called a *key*. The value at other times is found by interpolating between keys, or by extrapolating from the first and last key. See the envelope SDK sample to find out exactly how this is done.

The channels available for editing are displayed in a list, called the channel bin. Channels are selected for editing from the bin, and commands can be used both to select channels and to change the contents of the bin. One or more keys in the selected channels can also be selected. Commands typically operate on the selected channels and keys.

The Graph Editor commands complement the functions available through the Animation Envelopes and Channel Info globals.

GE_OpenWindow **n**mode
GE_SetWindowPos **n**x **x**y
GE_SetWindowSize **n**width **n**height

Open the Graph Editor window, and set its position and size. In Layout, `GraphEditor` is a synonym for `GE_OpenWindow`.

GE_ClearBin

Remove all channels from the channel bin.

GE_SetEnv **s**channelname **n**append
GE_SetEnvID **x**channelid **n**append

Add the channel to the channel bin and select it. If `append` is true, the channel bin isn't cleared, and the channel is added to the current selection. Otherwise the added channel replaces the contents of the channel bin.

GE_GetLayoutSel **n**append

Get motion channels for the items selected in Layout and add them to the channel bin. If `append` is true (non-zero), the channel bin isn't cleared, and selected item channels are added to the bin. Otherwise the selected item channels replace the contents of the channel bin.

GE_FilterSelection **s**filter

Remove channels from the channel bin whose names don't match the filter string. (Contrary to what the name implies, this command filters the bin contents, *not* the selection.) The filter is a regular expression. "*.Position.*" leaves only position channels in the list, for example. If the filter can't be parsed, the user will be prompted for a valid filter. Currently, the filter string must be enclosed in double-quotes.

GE_SelectAllCurves

Select all of the channels in the channel list.

GE_SetGroup **x**groupid

Assign the selected channels to a group.

GE_ApplyServer **s**class **s**server
GE_RemoveServer **s**class **n**index

Apply a plug-in to the selected channels, or remove it. The `class` and `server` arguments are the first and second fields of the ServerRecord for the plug-in. The index refers to the list of applied servers of a given class. The first server in each list has an index of 1.

GE_BakeCurves

Create a keyframe at every frame of the selected channels, and make every span's interpolation linear.

GE_SelectAllKeys **n**deselect

Select all of the keys in the selected channels, if `deselect` is false, or deselect all keys if `deselect` is true.

GE_CopySelKeys
GE_PasteKeys **g**frame
Copy the selected keys and paste them at the given frame.

GE_DeleteSelKeys
Delete the selected keys.

GE_MoveKeys **g**deltaframe **g**deltavalue
Shift the selected keys in both time and value. The deltas are added to the time and value, so to shift only one of these, set the other delta to 0.

GE_SnapKeysToFrames
Shift each selected key in time to the nearest integral frame. (Keys may be set at fractional frame times.)

GE_ReduceKeys **n**recursive **g**threshold
Delete neighboring keys with values that differ by less than the threshold. For example, consider consecutive keys A B C D E with values that all lie within the threshold. When `recursive` is false, the first `GE_ReduceKeys` deletes keys B and D, leaving A C E. The second call deletes C, and the third deletes E. When `recursive` is true, a single call to `GE_ReduceKeys` deletes all the keys except A.

GE_LockKeys **n**unlock
Lock the selected keys, if `unlock` is false, or unlock the keys if `unlock` is true. A locked key can still be selected but can't be edited in the interface, a protection against accidental changes.

GE_LeaveFootprints
GE_PickupFootprints
GE_BacktrackFootprints
Apply, remove or revert to footprints. A footprint is a static copy of a selected curve. It's displayed in the Graph Editor window as a reference while the user edits the curve, and the user can undo the changes made after the footprint was created. `GE_LeaveFootprints`

records the current state of the selected curves. `GE_PickupFootprints` removes the footprints. `GE_BacktrackFootprints` reverts the curves to the state recorded in the footprints.

GE_CopyTimeslice **n**fromfootprint **g**frame
GE_PasteTimeslice **g**frame

Copy and paste keys at specific times. If `fromfootprint` is true, the keys are copied from the footprint rather than the curve.

GE_MatchFootprintAtFrame frame

Create a key at the specified frame with the value of the footprint at that frame.

GE_CreateExpression **s**name **s**expression

Create an expression. The name is used to refer to the expression in both the interface and the `GE_AttachExpression` commands. Consult the LightWave user documentation for information about what an expression can contain.

GE_AttachExpression **s**channelname **s**expressionname
GE_AttachExpressionID **x**channelid **s**expressionname

Associate an expression with the channel identified either by name or by channel ID.

GE_LoadExpressions **s**filename
GE_SaveExpressions **s**filename

Store all of the expressions in a file, or retrieve them from a file.

Layout Commands

Layout commands are case-sensitive. Layout's native command mechanism is the `evaluate` function, with the command name and its arguments passed as a single, space-delimited string. Commands issued using the `lookup` and `execute` functions will be translated into `evaluate` strings before being processed, and the command arguments passed to `execute` must all be DynaValues of type `DY_STRING`.

Layout doesn't strip quote marks from arguments. They're treated literally, so they should only be included when they're actually part of the argument. (There aren't any commands whose argument lists require anything other than spaces to delimit the arguments.)

Most Layout command arguments are optional. If the argument isn't supplied, Layout will prompt the user for the argument. In the command list that follows, the data types of the arguments are denoted by the initial letter. (The same letters are used in C `printf` formatting.)

number	An integer.
gfloat	A floating-point number.
string	A string, such as a filename or channel name.
he**x**	A hexadecimal number, such as an item identifier.

The commands are divided into 15 categories, roughly according to what they do and which globals provide the current values of the parameters they set.

Selection

Most item-specific commands operate on the currently selected item. The Interface Info global returns a list of the currently selected items.

SelectItem **x**ID
SelectByName **s**name
AddToSelection **x**ID
RemoveFromSelection **x**ID

Select an item by item ID or by name, or add and remove an item from the list of selected items.

SelectAllObjects
SelectAllBones
SelectAllLights
SelectAllCameras

Select all items of a given type.

EditObjects
EditBones
EditLights
EditCameras

Set the edit mode. This may affect the scope of some commands that depend on the contents of the scene or the item selection.

PreviousItem
NextItem
FirstItem
LastItem

Select items by traversing the item list.

SelectParent
SelectChild
PreviousSibling
NextSibling

Select items by traversing the parenting tree.

Objects

Commands that set object parameters operate on the currently selected object. You can use the functions returned by the Object Info global to get the current values of these parameters.

LoadObject **s**filename
LoadObjectLayer **n**layer **s**filename

Load an object, or an object layer, and add it to the scene.

ReplaceWithObject **s**filename
ReplaceObjectLayer **n**layer **s**filename

Replace the selected object or object layer.

AddNull **s**name
ReplaceWithNull **s**name

Add a null object to the scene, or replace the selected object with a null.

AddPartigon **s**filename

Add a partigon (a particle object) to the scene. The filename is used to save the object.

SaveAllObjects

Save all objects in the scene. This updates (overwrites) the object files and is useful primarily for saving any surface changes that may have been made in Layout.

SaveObject **s**filename

Save the selected object to a different file. Later references to the object in the scene will use the new filename.

SaveTransformed **s**filename

Save a copy of the selected object in its current state, generally after patching, deformations and motions have been applied in the current frame.

SaveEndomorph **s**filename

Save an endomorph of the selected object. This is the geometry of the object after transforming the points using a vertex map of type `MORF` or `SPOT`.

SaveObjectCopy **s**filename
    Save a copy of the selected object.

ClearAllObjects
    Remove all objects from the scene.

SubdivisionOrder **n**order
    Set the subdivision order for subpatches in the selected object. The argument can be any of the values returned by the Object Info `subdivOrder` function.

SubPatchLevel **n**display **n**render
MetaballResolution **n**display **n**render
    Set the patch resolution for subpatches and metaballs.

MorphTarget **x**ID
    Set the selected object's morph target. The argument is the item ID of the target object.

MorphAmount **g**amount
    Set the selected object's morph amount, usually a value between 0.0 and 1.0.

MorphMTSE
MorphSurfaces
    Toggle the selected object's MTSE (Multiple Target Single Envelope) and Morph Surfaces options.

ObjectDissolve **g**dissolve
    Set the selected object's dissolve, usually a value between 0.0 and 1.0.

PolygonSize **g**size
    Set the selected object's polygon size. The default size is 1.0.

UnseenByRays

UnseenByCamera
UnaffectedByFog

Toggle options affecting the participation of the object in certain rendering components.

SelfShadow
CastShadow
ReceiveShadow

Toggle the selected object's raytraced shadow options.

IncludeLight **x**ID
ExcludeLight **x**ID

Include or exclude a light. The argument is the item ID of the light.

ShadowExclusion

A toggle that's on by default and applies to the whole scene. When turned off, shadow calculations will ignore light exclusion.

PolygonEdgeFlags **n**flags
PolygonEdgeColor **g**red **g**green **g**blue

Set the flags and color of the object's polygon edges. See the Object Info `edgeOpts` function for a list of possible flags.

EnableDeformations

Enable or disable deformations for all objects in the scene.

Bones

Commands that set bone parameters operate on the currently selected bone. You can use the functions returned by the Bone Info global to get the current values of these parameters.

AddBone **s**name
AddChildBone **s**name

Add a bone to the scene. The currently selected bone becomes the parent of new child bones.

DrawBones
DrawChildBones

Activates a mode in which bones can be added by clicking and dragging in the interface.

ClearAllBones

Remove all bones from the scene.

SkelegonsToBones

Convert skelegons in the object to scene bones.

BoneActive

Make the bone active or inactive. An inactive bone is ignored during deformation calculations.

BoneFalloffType **n**type

Set the falloff function for the bone. The falloff is proportional to the distance raised to the power -2^{type}. A `type` of 0 is inverse distance, 1 is inverse distance squared, 2 is inverse distance to the fourth power, and so on.

BoneRestPosition **g**X **g**Y **g**Z
BoneRestRotation **g**H **g**P **g**B
BoneRestLength **g**length
BoneStrength **g**strength

Set bone parameters.

BoneStrengthMultiply

When set, the strength will be multiplied by the rest length.

RecordRestPosition

Take the bone's current position to be its rest position.

BoneWeightMapName **s**name
BoneWeightMapOnly
BoneNormalization

Set a weight map for a bone. If the Weight Map Only option is toggled on, the normalization toggle is also turned on by default. With normalization on, the displacement of a point is divided by the sum of the weights.

BoneLimitedRange
BoneMinRange **g**distance
BoneMaxRange **g**distance
    Toggle limited range, and set the minimum and maximum extents of the range.

BoneJointComp
BoneJointCompParent
BoneJointCompAmounts **g**self **g**parent
    Toggle joint compensation and the inclusion of the parent in the calculations, and set the compensation amounts. Both joint compensation and muscle flexing are deformation adjustments designed to preserve volume, and they can optionally take the parent into account when the adjustment is calculated.

BoneMuscleFlex
BoneMuscleFlexParent
BoneMuscleFlexAmounts **g**self **g**parent
    Toggle muscle flexing and the inclusion of the parent in the calculations, and set the flex amounts.

Lights

Commands that set light parameters operate on the currently selected light. You can use the functions returned by the Light Info global to get the current values of these parameters.

AddDistantLight **s**name
AddPointLight **s**name
AddSpotlight **s**name
AddLinearLight **s**name
AddAreaLight **s**name
    Add a light of the given type to the scene.

ClearAllLights
    Remove all lights from the scene.

SaveLight
    Save the light's parameters to a file. The user is prompted for the

filename. The light information is stored in the format used for scene files.

DistantLight
PointLight
Spotlight
LinearLight
AreaLight
    Set the type of the light.

LightColor **g**red **g**green **g**blue
LightIntensity **g**intensity
    Set the color and intensity of the selected light.

LightFalloffType **n**type
LightRange **g**distance
    Set the falloff type and range (or nominal distance) for the light. The falloff type affects the interpretation of the range value. These parameters aren't valid for distant lights.

AffectDiffuse
AffectSpecular
AffectCaustics
AffectOpenGL
    Toggle these effects for the light.

LightConeAngle **g**angle
LightEdgeAngle **g**angle
    Set the cone angle and soft edge angle for a spotlight.

LightQuality **n**quality
    Set the quality level for a linear or area light.

ShadowType **n**type
    Set the light's shadow type.

CacheShadowMap
    Toggle the Cache Shadow Map option for a shadow-mapped light.

ShadowMapSize **n**size

Set the dimension of the shadow map for a light. A shadow map is a 2D array of pixels. The amount of memory it uses is proportional to the square of the size.

ShadowMapFuzziness **g**fuzziness

Set the amount of blur or softness for the shadow map.

ShadowMapFitCone
ShadowMapAngle **g**angle

Set the "field of view" for a shadow map. This is the angle subtended by the shadow map at the position of the light. The `FitCone` toggle sets this angle equal to the cone angle of a spotlight.

LensFlare

Toggle a lens flare effect for the light.

FlareIntensity **g**intensity

Set the lens flare intensity.

VolumetricLighting

Toggle volumetrics for the light.

Global Illumination

EnableLensFlares
EnableShadowMaps
EnableVolumetricLights
EnableRadiosity
EnableCaustics

Toggle these features for the whole scene.

AmbientColor **g**red **g**green **g**blue
AmbientIntensity **g**intensity

Set the ambient light color and intensity.

NoiseReduction

Toggle the Shader Noise Reduction feature.

RadiosityType **n**type

Set the radiosity type, which can be one of the following.

0 - Backdrop Only
1 - Monte Carlo
2 - Interpolated

CacheRadiosity
CacheCaustics

Toggle caching of the solutions for radiosity and caustics.

RadiosityIntensity **g**intensity
`RadiosityTolerance **g**tolerance`
`CausticIntensity **g**intensity`

Set the radiosity intensity and tolerance and the caustic intensity.

VolumetricRadiosity

A toggle that's on by default, this controls whether volumetrics are taken into account by radiosity.

Cameras

Commands that set camera parameters operate on the currently selected camera. You can use the functions returned by the Camera Info global to get the current values of these parameters.

AddCamera **s**name

Add a camera to the scene.

ClearAllCameras

Remove all cameras from the scene and restore the default camera.

FrameSize **n**width **n**height
ResolutionMultiplier **g**multiplier

Set the size of the output image. The resolution multiplier makes it easier to scale the image up or down without affecting (or knowing) the base frame size.

PixelAspect **g**aspect

Set the ratio of width to height for a pixel. Aspects less than 1.0

produce pixels that are taller than they are wide.

CameraMask
MaskPosition **n**left **n**top **n**width **n**height
MaskColor **g**red **g**green **g**blue

Enabling the camera mask causes the mask color to be drawn into the image outside the mask rectangle.

LimitedRegion

Toggle limited region rendering. When this is enabled, only the part of the image inside the limited region rectangle will be rendered.

ZoomFactor **g**factor

Set the camera zoom. This is the distance of the camera from the plane of projection. The image on this plane always has a half-height of 1.0.

ApertureHeight **g**height

Set the aperture height of the camera. The focal length is `zoom * ApertureHeight / 2`.

Antialiasing **n**level

Set the antialiasing level.

0 - Off
1 - Low (5 passes)
2 - Medium (9 passes)
3 - High (17 passes)
4 - Extreme (33 passes)

EnhancedAA

Toggles Enhanced antialiasing.

AdaptiveSampling

Toggles adaptive sampling.

AdaptiveThreshold **g**threshold

Sets the threshold at which adjacent pixels are sufficiently different to receive antialiasing attention. Pixels with luminosity differences smaller than this will be skipped during antialiasing when adaptive sampling is enabled.

ParticleBlur
Toggle particle blur.

MotionBlur **n**type
Set the motion blur option.

0 - Off
1 - Normal
2 - Dithered

BlurLength **g**length
Set the blur length as a fraction of the time between successive frames.

Stereoscopic
EyeSeparation **g**separation
Toggle stereoscopic rendering and set the interoccular distance.

DepthOfField
FocalDistance **g**distance
LensFStop **g**fstop
Toggle depth of field rendering and set the focus distance and f-stop.

Items

These commands operate on the currently selected item and work with items of all types. Use the Item Info or Interface Info globals to get the current values of the parameters set by many of these commands.

ClearSelected
Remove the selected items from the scene.

Rename **s**name
Rename the item. This works for all item types except objects, which derive their names from their filenames.

Clone **n**clones

Create one or more clones of the item.

ItemActive **n**active

Make the item active or inactive. An inactive item doesn't participate in rendering calculations. Inactive objects are 100% dissolved, and inactive lights have an intensity of 0. For bones, the command is a synonym for the `BoneActive` command. It currently has no effect on cameras.

ApplyServer **s**class **s**server

Apply a plug-in. The arguments are the first and second fields of the ServerRecord for the plug-in.

RemoveServer **s**class **n**index

Remove a plug-in. The index refers to the list of applied servers of a given class. The first server in each list has an index of 1.

AddEnvelope **s**channel
RemoveEnvelope **s**channel

Create or remove the envelope associated with a parameter in Layout. These commands "press the E button" for envelopes Layout itself maintains. Use them for envelopes *not* created by your plug-in through either the Variant Parameters or Animation Envelopes globals. The `channel` argument is the same as the channel name displayed in the Scene Editor when an item's channel list is expanded.

SchematicPosition **n**X **n**Y

Set the position of the current item within schematic views. The coordinates are relative to the upper left corner of the view pane.

ItemLock **n**locked

Lock or unlock the selected item. A locked item can't be selected by clicking on it in the view.

ItemVisibility **n**visibility

Set the visibility of the selected item. The argument can be any of the values returned by the Interface Info global's `itemVis` function.

ItemColor **n**color

Set the wireframe color of the selected item. The argument is an index into a color table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
⬛	🟦	🟩	🟦	🟥	🟪	🟫	⬜	🟦	🟩	🟦	🟥	🟪	🟨	⬜

Motion

Motion commands operate on the current item. Use the `param` function of the Item Info global, as well as the Animation Envelopes and Channel Info globals, to read motion-related settings.

CreateKey **g**time

Create a key for all nine motion channels (position, rotation and scale). If a key already exists, this changes its settings.

DeleteKey **g**time

Delete all of an item's motion keys at a given time.

AutoKey

Toggle the Auto Key setting. When on, changes in the position, rotation or scale of an item are automatically recorded to a key. When off, a key must be created explicitly.

Position **g**X **g**Y **g**Z
Rotation **g**H **g**P **g**B
Scale **g**X **g**Y **g**Z

Position, rotate or scale the item. This has no effect on keys at the current time unless Auto Key has been turned on.

AddPosition **g**X **g**Y **g**Z
AddRotation **g**H **g**P **g**B
AddScale **g**X **g**Y **g**Z

Increment the motion. Like `Position`, `Rotation` and `Scale`, but the values are relative rather than absolute.

PivotPosition **g**X **g**Y **g**Z
PivotRotation **g**H **g**P **g**B

Set the position and rotation of the pivot (the origin of the item's rotation).

RecordPivotRotation
: Add the current item rotation angles to the pivot rotation, and reset the item rotation angles to (0, 0, 0).

Numeric
: Prepare for numeric entry. The specific behavior depends on which tool is active. For the Move, Rotate and Size tools, this activates the vector edit fields, and values typed after this will be entered there.

EnableXH
EnableYP
EnableZB
: Unlock or lock a channel against changes caused by mouse dragging. Which channel is affected depends on which tool is active. If the Move tool is active, for example, `EnableXH` toggles the enable state of the X channel, and when it's disabled, moving an item by dragging the mouse may affect the Y and Z components of its position, but not the X component.

WorldCoordinateSystem
ParentCoordinateSystem
LocalCoordinateSystem
: These affect the coordinate system in which mouse movement is interpreted for the motion tools. Moving the mouse can move an item along the world, parent or local (item) axes.

LoadMotion **s**filename
SaveMotion **s**filename
: Load or save an item's keyframes in a motion file. The format of the file is similar to the format of the key data in scene files.

`Undo`
: Undo the most recent motion change.

Reset
: Set an item's animation channels to the values they had when the

object was created or loaded. Which channels are affected depends on which tool is currently active. Channels disabled by `EnableXH`, `EnableYP` or `EnableZB` are not reset.

`ParentItem` **x**ID

Set the parent of the current item to the item given by the ID. The item's motion is relative to that of its parent.

ParentInPlace

A global setting. When turned off, setting a parent relationship causes the child item to jump to a new (world coordinate) position, since the child's position setting suddenly becomes relative to that of its parent. When turned on, parenting in place prevents the jump, adjusting the child's position relative to its parent so that its world position doesn't change. This setting can be read as a flag using the Interface Info global.

TargetItem **x**ID

Set the target of the current item to the item given by the ID. The item will rotate so that it points toward the target.

PathAlignLookAhead **g**time

Set the look-ahead interval, in seconds, for motion channels controlled by Align to Path. This is the amount of time by which changes in orientation of the item anticipate changes in the path direction.

EnableIK

A global toggle that enables or disables inverse kinematics calculations for all items.

UnaffectedByIK

Toggle the Unaffected by IK of Descendents option. When on, the item is isolated from the motion effects of inverse kinematics applied to the item's children. The item becomes the base of the IK chain containing its children.

GoalItem **x**ID

Set the IK goal of the current item to the item given by the ID.

FullTimeIK

Toggle Full-time IK for the item. This determines whether IK is recalculated each time the item's goal is moved.

GoalStrength **g**strength

Set the strength, or amount of influence, of an item's IK goal. The goal strength is relative to that of the other goals associated with an IK chain.

MatchGoalOrientation

When on, this toggle causes the item's rotation and scale to match that of its goal.

KeepGoalWithinReach

When on, this toggle constrains the motion of the goal so that it's always "attached" to the item.

HController **n**controller
PController **n**controller
BController **n**controller

Set the controller for each rotation channel. The argument can be any of the values returned by the Item Info `controller` function.

HLimits **g**min **g**max
PLimits **g**min **g**max
BLimits **g**min **g**max

Set minimum and maximum limits for each rotation channel.

`RecordMinAngles`
`RecordMaxAngles`

Use the current rotation of the item as the minimum or maximum limit for all three rotation channels.

`LimitH`
`LimitP`
`LimitB`

Enable or disable rotation limits.

HStiffness **g**stiffness
PStiffness **g**stiffness

BStiffness **g**stiffness

Set the stiffness (resistance to rotation via IK).

Effects

These commands affect settings on the Effects panel. Use the Backdrop Info and Fog Info globals to read these settings.

GradientBackdrop
ZenithColor **g**red **g**green **g**blue
SkyColor **g**red **g**green **g**blue
GroundColor **g**red **g**green **g**blue
NadirColor **g**red **g**green **g**blue

Enable or disable the gradient backdrop, and set its key colors.

FogType **n**type

Set the fog type. The type can be any of the values that are valid for the Fog Info `type` field.

FogMinDistance **g**distance
FogMaxDistance **g**distance

Set the distance extents of the fog.

FogMinAmount **g**amount
FogMaxAmount **g**amount

Set the fog amounts at the minimum and maximum distances.

FogColor **g**red **g**green **g**blue

Set the fog color.

Scenes

These commands involve render options and scene loading and saving. You can read many of the parameters using the Scene Info global.

ClearScene

Clear the scene. All items are removed and all scene parameters are reset to their default values.

LoadScene **s**filename

  Load a scene file.

LoadFromScene **s**filename

  Load the objects, and optionally the lights, from a scene file.

SaveScene
SaveSceneAs **s**filename

  Save the scene.

SaveLWSC1 **s**filename

  Save the scene in version 1 format. This is the scene file format used by versions of LightWave prior to 6.0. Some information may be lost, since the old format doesn't support many of the newer features.

SaveSceneCopy **s**filename

  Save a copy of the scene. Unlike `SaveSceneAs`, `SaveSceneCopy` doesn't rename the scene.

ContentDirectory **s**dirname

  Set the content directory (a file system path). Objects and other files referenced in the scene are listed relative to this path.

FirstFrame **n**frame
LastFrame **n**frame
FrameStep **n**frames

  Set the frame range and step size for rendering.

FramesPerSecond **g**fps

  Set the playback speed of the animation. This is used to convert between frame numbers and elapsed time in seconds.

AutoFrameAdvance

  Toggle the Auto Frame Advance feature. During rendering with this turned off, the user must confirm before each frame is rendered.

EnableVIPER

  Enable or disable the VIPER (Virtual Interactive Preview Render) system. Enabling VIPER allows it to store buffers from the most recently rendered frame, but doesn't open the VIPER window.

RayTraceShadows
RayTraceReflection
RayTraceRefraction
RayTraceTransparency
Toggle raytracing of shadows, reflection, refraction and transparency.

RenderFrame
RenderSelected
RenderScene
Render the current frame, render only the currently selected objects in the frame, or render all frames.

Navigation

These commands are concerned with the user's navigation within Layout's interface, but being able to set the current frame and animate the display is also useful to plug-ins. The current frame is available from the Interface Info global as the current time. Use the `framesPerSecond` field from the Scene Info global to convert this to a frame number.

Some of these commands operate on a single view pane, the one that currently contains the mouse cursor.

GoToFrame **n**frame
PreviousFrame
NextFrame
PreviousKey
NextKey
Shuttle the display to a specific frame, an adjacent frame, or the next or previous frame containing a motion key.

Refresh
RefreshNow
Tell Layout to recompute motions and geometry. After changing the current frame, plug-ins that bake geometry can issue the `RefreshNow` command so that vertex positions are updated immediately. By default, `Refresh` runs during idle time, which won't occur while a plug-

in's code is executing.

Redraw
RedrawNow
Redraw the display. Like `RefreshNow`, `RedrawNow` executes immediately rather than waiting for idle time. But the draw commands by themselves don't cause motions and geometry to be recomputed.

PlayForward
PlayBackward
Pause
Animate the display, or make it stop animating.

PreviewFirstFrame **n**frame
PreviewLastFrame **n**frame
PreviewFrameStep **n**frames
Set the range of frames and the frame step for animating the display. The frame range also affects the range of frames that are accessible to the user through the frame shuttle control.

MakePreview
PlayPreview
FreePreview
LoadPreview
SavePreview
Manage pre-rendered previews. These store the animated display as an image sequence.

ClearAudio
LoadAudio
PlayAudio
Manage Layout's audio track.

FractionalFrames
When this toggle is set, Layout allows the user to move the frame shuttle to non-integer frame numbers.

CenterMouse
Center a view over the current mouse position.

CenterItem

Center a view over the position of the selected item. This is a toggle that causes the view to remain centered on the item while it's being moved.

FitAll
FitSelected

Scale and move a view so that it contains all items or all selected items.

ZoomIn
ZoomOut
ZoomInX2
ZoomOutX2

Scale a view. The `X2` versions scale the view by a factor twice as large.

Display

These commands set display options. Many of these can be read from the Interface Info global. As with some navigation commands, many of these commands operate on the view pane that contains the mouse cursor.

GridSize **g**size
IncreaseGrid
DecreaseGrid

Set the grid size, or scale it in approximately logarithmic steps: 1, 2, 5, 10, 20, 50, and so on. The grid size is the interval in meters between grid lines in the display. The scale of the view is relative to this size, so the entire scene (but not the grid) will *appear to* grow or shrink in the view panes as this number is changed.

DynamicUpdate **n**level

Set the amount of display interactivity. The level can be any of the values that are valid for the Interface Info `dynaUpdate` field.

ShowMotionPaths
ShowHandles

ShowIKChains
ShowCages
ShowSafeAreas
ShowFieldChart
ShowTargetLines

Toggle the drawing of these display components.

BoundingBoxThreshold **n**threshold

Set the minimum number of points at which an object will be drawn initially as a bounding box. Lower numbers improve interactivity, but plug-ins that bake geometry transformations may need to set this above the number of points in the object they're baking. Otherwise only the bounding box vertex positions will be computed by Layout.

TopView
BottomView
BackView
FrontView
RightView
LeftView
XYView
XZView
ZYView
PerspectiveView
LightView
CameraView
SchematicView

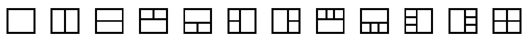
Set the type of view in a view pane. The first six are orthographic views in the direction of the world -Y, +Y, -Z, +Z, -X and +X axes, respectively. `XYView` (X increasing to the right on the display, Y increasing upwards) is a synonym for `BackView`. `XZView` (X right, Y up) is a synonym for `TopView`, and `ZYView` (Z right, Y up) is a synonym for `RightView`. The light and camera views look through the selected light and camera. The schematic view is a diagram of the parenting hierarchy.

ViewLayout **n**layout
PreviousViewLayout
NextViewLayout

SingleView

Select a view layout, the arrangement of up to four view panes. The layout indexes are

0 1 2 3 4 5 6 7 8 9 10 11



`SingleView` is a synonym for `ViewLayout 0`.

SaveViewLayout

Make the current view layout the default.

Layout_SetWindowPos **n**X **n**Y
Scene_SetWindowPos **n**X **n**Y
Item_SetWindowPos **n**X **n**Y

Set the position of Layout's main window and the Scene Editor and Item Properties panels. The arguments are the screen coordinates of the upper left corner.

Layout_SetWindowSize **n**width **n**height
Scene_SetWindowSize **n**width **n**height

Set the dimensions of Layout's main window and the Scene Editor panel.

HideToolbar

Toggle the visibility of the toolbar (the vertical strip of buttons next to the view area).

AddButton **s**command **s**group

Add a button to the toolbar or one of the other menu areas. The `group` is the name of the menu group in which the button will be added.

AlertLevel **n**level

Set the alert level. This controls the method used to display warning, error and informational messages. The level can be any of the values returned by the Interface Info `alertLevel` function.

AutoConfirm **n**on_off

Set the auto-confirm switch. When this is turned on, confirmation dialogs that would otherwise be displayed to the user are suppressed. Plug-ins can use this to issue a large number of commands without inundating the user with requests to confirm each operation. Remember to turn it off again when you no longer need it.

Tools

These enable specific tools. Once a tool is active, the user's mouse movements and button clicking are interpreted as operations of the tool. These are available as commands primarily so that the user can map them to shortcut keys, but a few of the commands (Reset, for example) use the active tool as a context.

```
MoveTool
RotateTool
SizeTool
StretchTool
SquashTool
MovePivotTool
RotatePivotTool
MovePathTool
MorphAmountTool
RestLengthTool
ConeAngleTool
CameraZoomTool
AdjustRegionTool
LightIntensityTool
ChangeTool
```

Panels

These commands open or close panels.

EditServer **s**class **n**index
LastPluginInterface

Display a plug-in's interface. The `class` is the first field of the plug-in's ServerRecord, and the `index` refers to the list of applied servers of a given class. The first server in each list has an index of 1. Use the Item Info global to examine each item's server lists.

HideWindows

Toggle the visibility of open panels.

About

Display the About dialog.

SceneEditor
GraphEditor
SurfaceEditor
ImageEditor
ItemProperties
DisplayOptions
GeneralOptions
FlareOptions
VolumetricLightingOptions
MotionOptions
PreviewOptions
RenderOptions
NetRender
Backdrop
Volumetrics
Compositing
ImageProcessing
AboutOpenGL
VIPER
Presets
Statistics
EditMenus
EditKeys
EditPlugins
MasterPlugins
Generics
CommandHistory
CommandInput **s**command

Open one of the panels, or close the panel if it's open.

Miscellaneous

These commands don't easily fit into any of the other categories.

Model

Switch to Modeler. If Modeler isn't running, it will be started. If the Hub isn't running, the command has no effect.

Synchronize

Synchronize objects in the scene with Modeler (reload objects that have been modified in Modeler).

AddPlugins **s**filename

Install the plug-ins found in the file.

SaveCommandList

Save a list of these commands as a text file. The list will also contain all of the installed LayoutGeneric class plug-ins, as well as commands for some of the editors.

Quit

Quit Layout.

Modeler Commands

Modeler commands are *not* case-sensitive. Modeler's native command mechanism is the `lookup` and `execute` pair of functions, which translate the command name into a code and use an array of DynaValues to pass the arguments. Commands issued using the `evaluate` function will be converted into `lookup` and `execute` calls. (See the CommandSequence document for definitions of these functions.)

Modeler's `evaluate` function treats double quote marks as delimiters, not literal characters. Use them when a string argument contains spaces. You can insert a quote mark as a literal character in a string argument by preceeding it with the backslash (`\`) escape character. If you need a literal backslash, use two in a row. (This can produce some odd-looking code if you're generating `evaluate` strings using `sprintf`. To generate a literal quote character that won't be removed by either the C compiler or Modeler's command processor, the `sprintf` string needs to contain *three* backslashes, followed by the quote character: `\\\"`.)

Some Modeler command arguments are optional. If they occur at the end of the argument list, they can simply be omitted. If they're in the middle, they can be replaced by placeholders, a `DY_NULL` DynaValue for `execute` or an asterisk (`*`) in an `evaluate` string. Either way, Modeler will supply default values.

In the command list that follows, optional arguments are placed inside square brackets. The types of the arguments are denoted by the initial letter.

number
: A single integer or floating-point number. These are passed to `execute` as DynaValues of type `DY_INTEGER` or `DY_FLOAT`.

vector
: A triple of numbers. In `evaluate` strings, vectors are delimited by angle brackets (`<` and `>`). If one or two of the numbers is omitted, the last number present is repeated, so `<0>` and `<0 0 0>` are equivalent. For `execute`, vectors are passed as `DY_VINT`s or

`DY_VFLOAT`s.

string

A string, such as a filename or surface name. Passed to `execute` as `DY_STRING`s.

keyword

A string containing one of several options. Valid keywords are listed in the definition of the command.

flags

A string in which each character represents a toggle.

The commands are divided into six broad categories here, but this is just to get the list under control. The categories don't have any programming significance.

General

CLOSE
CLOSE_ALL

Close the current object workspace, or all object workspaces.

`EXIT`

Exit Modeler.

`NEW`

Create a new, empty, unnamed object.

UNDO
REDO

These move back and forth in Modeler's undo buffer.

DELETE
CUT
COPY
PASTE

The delete command removes the selected geometry without placing it in Modeler's clipboard, unlike cutting, but deletes can still be undone.

LOAD **s**filename
SAVE **s**filename

Load and save object files.

REVERT **s**filename

Reload an existing object file.

SETOBJECT **s**name [**n**index]

Set the current object by name, filename or index.

SETLAYER **s**layers
SETALAYER **s**layers
SETBLAYER **s**layers

Set the current foreground (or active) and background layers. `SETALAYER` is a synonym for `SETLAYER`. The `layers` argument is a string containing one or more layer numbers separated by spaces. Layers are numbered sequentially, starting at 1.

`SETLAYERNAME` **`s`**`name`

Set the name of the current layer.

`SETPIVOT` **`v`**`pos`

Set the pivot point for the current layer. The pivot point is the origin for rotations in Layout.

`SURFACE` **`s`**`name`

Set the current default surface. Geometry created after this is called will be assigned this surface.

`SELECTVMAP` **`k`**`type` **`s`**`name`

Set the current vertex map of a given type. The type can be `MORF` (relative morph), `SPOT` (absolute morph), `WGHT` (weight), `MNVW` (subpatch weight) or `TXUV` (texture UV).

`CMDSEQ` **`s`**`name [`**`s`**`arg]`

Activate another command sequence plug-in, identified by its internal name, the string in the plug-in's ServerRecord `name` field. The argument string is placed in the `argument` field of the LWModCommand structure passed to the plug-in's activation function.

`MESHEDIT` **`s`**`name`

Activate a MeshDataEdit class plug-in, identified by its internal name.

`PLUGIN` **`s`**`filename [`**`s`**`class` **`s`**`name` **`s`**`username]`

Install the plug-ins contained in a .p file.

Selection

Commands are applied to the current *selection*, a subset of the geometry data residing in Modeler when a command is issued. The selection is made up of elements from the current layers, and within those layers, is defined by your choice of EltOpSelect mode for each command.

For `OPSEL_USER` and `OPSEL_DIRECT` modes, you can change which elements are selected using the `SEL_POINT` and `SEL_POLYGON` family of commands. (To individually select points and polygons by ID, you'll need to use a mesh edit with the special `OPSEL_MODIFY` mode.)

`SEL_POINT` **k**`action [condition ...]`

Modify point selection. If it isn't already, the Point tab in Modeler's interface will be selected after this command is issued. The action can be either `SET` or `CLEAR`. If there is no condition, the action will apply to all points. Otherwise, the points specified by the condition will be added to the selection for `SET` and removed from the selection for `CLEAR`. The possible conditions with their additional arguments are:

`"VOLUME"` **v**`lo` **v**`hi`
: Points within the volume.

`"CONNECT"`
: Points connected to already selected ones. Only works with `SET`.

`"NPEQ"` **n**`pols`
: Points belonging to exactly pols polygons.

`"NPLT"` **n**`pols`
: Points belonging to less than pols polygons.

`"NPGT"` **n**`pols`
: points belonging to more than npol polygons.

`SEL_POLYGON` **k**`action [condition ...]`

Modify polygon selection. Like `SEL_POINT`, with the following conditions:

`"VOLEXCL"` **v**`lo,` **v**`hi`
: Polygons entirely within the volume.

`"VOLINCL"` **v**`lo,` **v**`hi`
: Polygons at least partly within the volume.

`"CONNECT"`
: Polygons connected to already selected ones. Only works with `SET`.

`"NVEQ"` **n**`verts`
: Polygons with exactly verts vertices.

`"NVLT"` **n**`verts`

Polygons with less than verts vertices.

`"NVGT" `**`n`**`verts`

Polygons with more than verts vertices.

`"SURFACE" `**`s`**`surface`

Polygons having the given surface.

`"FACE"`

Face polygons only.

`"CURVE"`

Curve polygons only.

`"NONPLANAR" [`**`n`**`limit]`

Polygons less planar than the given limit. If limit is omitted, the user's default limit is used.

`SEL_INVERT`

Invert the selection. Selected data becomes unselected and vice versa.

`SEL_HIDE `**`k`**`state`

Hide data from view. The `state` can be `SELECTED` (hide the selected data, the default) or `UNSELECTED`.

`SEL_UNHIDE`

Unhide all hidden data.

`INVERT_HIDE`

Invert the hiding of data. Hidden data becomes unhidden and vice versa.

Point Transforms

The flex and deform transformations use similar region data to define the scope of their operations. These are set globally and then applied to all transformations of a given type.

`FIXEDFLEX `**`k`**`axis `**`n`**`start `**`n`**`end [`**`f`**`ease]`

Set the flex functions to operate on a fixed range along an axis. Ease flags can be `"i"` (ease-in), `"o"` (ease-out), or `"io"`.

`AUTOFLEX `**`k`**`axis `**`k`**`polarity [`**`f`**`ease]`

Set the flex functions to operate on an automatic range along an axis of the given polarity, which can be `"+"` or `"-"`.

`DEFORMREGION `**`v`**`radius [`**`v`**`center `**`k`**`axis]`

Set the area of effect for the deformation tools. If axis is omitted, the effect is bounded in all directions by the given radius. If an axis is specified, the effect is unbounded along that axis.

MOVE **v**offset
SHEAR **v**offset
MAGNET **v**offset

Translate points by the given offset. Shear translates along the flex axis. Magnet translates in the deform region.

ROTATE **n**angle **k**axis [**v**center]
TWIST **n**angle **k**axis [**v**center]
VORTEX **n**angle **k**axis [**v**center]

Rotate points along the given axis by the angle given in degrees. Twist uses the flex axis, and vortex uses the deform region.

SCALE **v**factor [**v**center]
TAPER **v**factor [**v**center]
POLE **v**factor [**v**center]

Scale points by the given factors around the given center. Taper uses the flex axis, and pole uses the deform region.

`BEND` **`n`**`angle` **`n`**`direction [`**`v`**`center]`
`BEND2` **`n`**`angle` **`n`**`direction [`**`v`**`center]`

Bend points by the given bend angle in the direction around the optional center. These commands use the current flex axis. In new code, use `BEND2`, which always interprets angle values as degrees.

`JITTER` **`v`**`radius [`**`k`**`type` **`v`**`center]`

Randomly translate points using different displacement functions. The jitter type can be `UNIFORM`, `GAUSSIAN`, `NORMAL` or `RADIAL`. `UNIFORM` is the default and moves points randomly along all three axes within a box of the given size. `GAUSSIAN` distributes the offsets in a bell curve around the start point. `NORMAL` shifts the points in and out along the local surface normal. `RADIAL` shifts points toward or away from the center.

`SMOOTH [`**`n`**`iterations` **`n`**`strength]`

Apply a smoothing function to attempt to remove kinks in polygons connecting affected points.

`QUANTIZE` **`v`**`size`

Snap all points to a 3D grid defined by the size vector.

`MERGEPOINTS [`**`n`**`mindist]`

Merge points lying within a certain minimum distance of each other. If no distance is given, it is computed heuristically.

Object Creation

`MAKEBOX` **v**`lowcorner` **v**`highcorner [`**v**`segments]`

Make a box with the given extent and and number of segments.

`MAKEBALL` **v**`radius` **n**`sides` **n**`segments [`**v**`center]`

Make a globe-style sphere.

`MAKETESBALL` **v**`radius` **n**`level [`**v**`center]`
`MAKETESBALL2` **v**`radius` **n**`segments [`**v**`center]`

Make a tesselated sphere. A level 0 tesball is an icosahedron. A level *n* tesball divides the edges of the icosahedron into 2^n segments. The newer `MAKETESBALL2` comand allows the number of segments along the edges to be any number, not just powers of 2.

`MAKEDISC` **v**`radius` **n**`top` **n**`bottom` **k**`axis` **n**`sides [`**n**`segments` **v**`center]`

Make a disc.

`MAKECONE` **v**`radius` **n**`top` **n**`bottom` **k**`axis` **n**`sides [`**n**`segments` **v**`center]`

Make a cone. The top is the pointy end.

`MAKETEXT` **s**`text` **n**`font [`**k**`cornertype` **n**`kern` **n**`scale` **k**`axis` **v**`pos]`
`MAKETEXT2` **s**`text` **n**`font [`**k**`cornertype` **n**`kern` **n**`scale` **k**`axis` **v**`pos` **k**`alignment]`

Generate text using the given font index. The font index begins at 1 for `MAKETEXT` and 0 (in agreement with the Font List global's `index` function) for `MAKETEXT2`. The corner type can be either `SHARP` or `BUFFERED`. The kern is an additional distance to put between characters (normally 0). The scale sets the approximate height of the character cell in meters. The axis defines the plane in which the text will lie. For `MAKETEXT`, text is always left-aligned with the position argument, while for `MAKETEXT2`, the alignment can be `LEFT`, `CENTER` or `RIGHT`.

Replication

`LATHE` **k**`axis` **n**`sides [`**v**`center` **n**`endangle` **n**`startangle` **n**`offset]`

Spin a template around an axis.

`EXTRUDE` **k**`axis` **n**`extent [`**n**`segments]`

Sweep a template along an axis.

`MIRROR` **k**`axis` **n**`plane`

Copy selected data, flipping it through a plane.

PATHCLONE **s**filename [**n**step **n**start **n**end]
PATHEXTRUDE **s**filename [**n**step **n**start **n**end]

Load a motion file and clone or extrude the selected data along the path in the file. Clone creates copies of the selected geometry at intervals along the path, while extrude creates a single continuous object by connecting the copies.

RAILCLONE **n**segments [**k**divs **f**flags **n**strength]
RAILEXTRUDE **n**segments [**k**divs **f**flags **n**strength]

Clone or extrude selected data along one or more rails (curves) in background layers. If segments is 0, the number of segments is computed automatically. Otherwise the number of segments is fixed, and the intervals between segments depend on whether `divs` is `KNOTS` (the default) or `LENGTHS`. For `KNOTS`, an equal number of segments is placed between each knot, or curve vertex, while for `LENGTHS`, the segments are spaced at equal intervals along the curve. The flags can be `"o"` (oriented), `"s"` (scaled) or `"os"`.

Tools

AXISDRILL **s**operation **k**axis [**s**surface]
SOLIDDRILL **s**operation [**s**surface]

The drill commands slice the foreground geometry using a 2D template or 3D shape in the background layer. The operation can be `CORE`, `TUNNEL`, `SLICE` or `STENCIL`.

BOOLEAN **s**operation

Booleans combine geometry in the foreground and background to create new shapes. The geometry involved must form fully enclosed volumes, which Modeler treats as solids when performing booleans. The operation can be `UNION`, `SUBTRACT`, `INTERSECT` or `ADD`.

BEVEL **n**inset **n**shift

Create a beveled edge around each selected polygon. The polygon's edges are moved inward (toward the polygon's center) by the inset amount, and the polygon is offset in the direction of its normal by the shift amount. The gap between the polygon's new and old vertices is filled by new polygons that form the beveled edge.

SHAPEBEVEL pattern

Perform multiple bevels using a single command. The patten for a shapebevel is either a string containing pairs of inset/shift values, or a

`DY_CUSTOM` DynaValue with the `val[0]` field set to the number of pairs, and the `val[1]` field cast to a pointer to an array of doubles holding the pairs.

SMOOTHSHIFT **n**offset [**n**maxangle]

Extrude part of a mesh. Vertices are moved in the average ("smoothed") direction of the shared polygons' normals. The mesh breaks at edges that form angles greater than `maxangle`, and the shift direction isn't averaged across those edges.

SMSCALE **n**offset

Move and scale part of a mesh. Vertices are moved as they are with `SMOOTHSHIFT`, but no new polygons are created.

Polygons

FLIP

Flip the sidedness (reverse the surface normals) of faces and the directions of curves.

TRIPLE

Convert polygons into triangles by subdividing.

FREEZECURVES

Convert curves into polygons.

REMOVEPOLS

Delete polygons, leaving behind their vertices as points.

UNIFYPOLS

Delete duplicate polygons.

ALIGNPOLS

Roughly speaking, flip polygons so that they all face in the "same" direction to form a coherent mesh. Incoherent meshes can result from the use of the `UNIFYPOLS` command, or when the geometry is created in another program that ignores the sidedness of polygons. It isn't always possible for `ALIGNPOLS` to infer the correct sidedness for all polygons, however.

CHANGESURFACE **s**surface

Set the surface of selected polygons.

CHANGEPART **s**part

Set the part tag for selected polygons.

SUBDIVIDE **k**mode [**n**maxangle]

Split triangles into four smaller triangles and quads into four smaller quads. The mode can be `FLAT`, `SMOOTH`, or `METAFORM`. With flat subdivision, the new polygons retain the parent's normal. A smooth subdivide creates polygons with normals that interpolate the parent's normal and those of its neighbors. Modeler won't try to interpolate normals across edges that form angles larger than `maxangle`. Metaform, unlike smooth mode, moves the original vertices to approximate the continuity (roundness) of a higher-order patch.

FRACSUBDIVIDE **k**mode **n**fractal [**n**maxangle]

Like `SUBDIVIDE`, but applies a fractal displacement to each new vertex as a function of its position.

TOGGLECCSTART
TOGGLECCEND

These affect the interpretation of the first and last points in curves. When toggled "on," the first (or last) point in a curve becomes a continuity control point. It and the curve segment connected to it are no longer part of the curve, but it can be used to affect the shape of the first (or last) segment that is still part of the curve.

TOGGLEPATCHES

Toggling this "on" converts the selected geometry into a subpatch control cage.

UNWELD

Create multiple copies of the selected points so that none are shared by two polygons at once.

The following eight commands are only valid with EltOpSelect modes of `OPSEL_USER` or `OPSEL_DIRECT`. Just as in the user interface, these operations require explicit selection of the elements they will operate on.

MAKE4PATCH **n**perpendicular **n**parallel

Create a quad mesh from three or four curves. The `perpendicular` and `parallel` values determine the number of segments that will be created in directions perpendicular and parallel to the last-selected curve. The curves must share vertices to form a closed area.

SKINPOLS

Create a triangle mesh that connects two or more polygons and encloses the volume between them. This is sometimes called lofting. The original polygons aren't required to have the same number of vertices.

MORPHPOLS **n**segments

Create a triangle mesh that connects exactly two polygons with the same number of vertices. The new mesh is divided into a number of segments along the line connecting the two original polygons.

MERGEPOLS

Merge selected polygons into a single polygon. Each selected polygon must share at least one edge with another.

WELDPOINTS

Merge selected points into a single point. The position of the resulting point is the same as that of the last-selected point before the weld.

WELDAVERAGE

Weld selected points into a single point located at the average position of the welded points.

SPLITPOLS

Divide a polygon into two smaller polygons. The new edge is created between selected points.

SMOOTHCURVES

Smooth a composite of two curves at their join point.

Object File Examples

This page is a supplement to the LightWave LWO2 object file format specification. It illustrates the most common elements of LightWave object files using a unit cube embellished in various ways. The discussion assumes you have access to the spec, but reading it isn't a prerequisite. In fact, you may want to read through these examples before tackling the reference information in the spec.

File contents are presented as both hex dumps and outlines. Although the outlines are much easier to read, the hex dumps are important because they're unambiguous. They contain the actual bytes of the file, written as 2-digit hexadecimal numbers, with 16 per line. Many of the files themselves can be found in the same directory as this page.

The source code samples in the LightWave plug-in SDK include a standalone LWO2 reader.

The Basic Cube

The first example is a simple unit cube centered on the origin, with default surface settings and a single layer. The file is 348 bytes in length. A hex dump of the entire file looks like this.

```
46 4F 52 4D 00 00 01 54 4C 57 4F 32 54 41 47 53   FORM   LWO2TAGS
00 00 00 08 44 65 66 61 75 6C 74 00 4C 41 59 52       Default LAYR
00 00 00 12 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 50 4E 54 53 00 00 00 60 BF 00         PNTS
00 00 BF 00 00 00 BF 00 00 00 3F 00 00 00 BF 00
```

```
00 00 BF 00 00 00 3F 00 00 00 BF 00 00 00 3F 00
00 00 BF 00 00 00 BF 00 00 00 3F 00 00 00 BF 00
00 00 3F 00 00 00 BF 00 00 00 3F 00 00 00 3F 00
00 00 BF 00 00 00 3F 00 00 00 3F 00 00 00 3F 00
00 00 BF 00 00 00 3F 00 00 00 3F 00 00 00 42 42                 BB
4F 58 00 00 00 18 BF 00 00 00 BF 00 00 00 BF 00   OX
00 00 3F 00 00 00 3F 00 00 00 3F 00 00 00 50 4F                 PO
4C 53 00 00 00 40 46 41 43 45 00 04 00 00 00 01   LS    FACE
00 02 00 03 00 04 00 00 00 04 00 05 00 01 00 04
00 01 00 05 00 06 00 02 00 04 00 03 00 02 00 06
00 07 00 04 00 00 00 03 00 07 00 04 00 04 00 04
00 07 00 06 00 05 50 54 41 47 00 00 00 1C 53 55         PTAG    SU
52 46 00 00 00 00 00 01 00 00 00 02 00 00 00 03   RF
00 00 00 04 00 00 00 05 00 00 53 55 52 46 00 00             SURF
00 2A 44 65 66 61 75 6C 74 00 00 00 43 4F 4C 52     Default   COLR
00 0E 3F 48 C8 C9 3F 48 C8 C9 3F 48 C8 C9 00 00
44 49 46 46 00 06 3F 80 00 00 00 00               DIFF
```

Here's the same file written in outline form.

```
FORM 340 LWO2
   TAGS 8
      "Default"
   LAYR 18
      0
      0
      0.0  0.0  0.0
      ""
   PNTS 96
      -0.5  -0.5  -0.5
       0.5  -0.5  -0.5
       0.5  -0.5   0.5
      -0.5  -0.5   0.5
      -0.5   0.5  -0.5
       0.5   0.5  -0.5
       0.5   0.5   0.5
      -0.5   0.5   0.5
   BBOX 24
      -0.5  -0.5  -0.5
       0.5   0.5   0.5
   POLS 64
      FACE
      4  0  1  2  3
      4  0  4  5  1
      4  1  5  6  2
      4  3  2  6  7
      4  0  3  7  4
      4  4  7  6  5
   PTAG 28
      SURF
      0  0
      1  0
      2  0
      3  0
      4  0
      5  0
   SURF 42
      "Default"
      ""
      COLR 14
         0.78431  0.78431  0.78431
         0
      DIFF 6
         1.0
         0
```

LightWave object files use the IFF syntax described in the EA-IFF85 document. Data is stored in a collection of chunks. Each chunk begins with a 4-byte chunk ID and the size of the chunk in bytes, and this is followed by the chunk contents.

```
   FORM 340 LWO2
```

Formally, a LightWave object file is a single IFF `FORM` chunk of type `LWO2`. The first 4 bytes are the characters 'F', 'O', 'R', 'M', and this is followed by a 4-byte integer containing the chunk size (the size of the file minus 8) and the `FORM` type (the characters 'L', 'W', 'O', '2'). As with all numbers in `LWO2` files, the chunk size is always written in big-endian (Motorola, network) byte order.

```
      TAGS 8
         "Default"
```

The `TAGS` chunk contains an array of strings. Whenever something is identified by name in the file, the ID is often a 0-based index into the `TAGS` array. The only named element in this file is its single surface, named "Default".

```
      LAYR 18
         0
         0
         0.0  0.0  0.0
         ""
```

The layer header signals the start of a new layer. All geometry elements that appear in the file after this and before the next `LAYR` chunk belong to this layer. The layer header contains an index, a flags word, the pivot point of the layer, the layer's name, and the index of the parent layer. This is the first (and only) layer, so its index is 0 and the optional parent index is omitted. The bits in the flags word are also 0, and the layer hasn't been given a name.

The pivot point is the origin for rotations in this layer and is expressed in world coordinates. Pivots typically differ from (0, 0, 0) when layers and layer parenting are used to create an object hierarchy.

```
      PNTS 96
      -0.5  -0.5  -0.5
       0.5  -0.5  -0.5
       0.5  -0.5   0.5
      -0.5  -0.5   0.5
      -0.5   0.5  -0.5
       0.5   0.5  -0.5
       0.5   0.5   0.5
      -0.5   0.5   0.5
```

The PNTS chunk contains triples of floating-point numbers, the coordinates of a list of points. The numbers are written as IEEE 32-bit floats in network byte order. The IEEE float format is the standard bit pattern used by almost all CPUs and corresponds to the internal representation of the C language `float` type. In other words, this isn't some bizarre proprietary encoding. You can process these using simple `fread` and `fwrite` calls (but don't forget to correct the byte order if necessary).

```
   BBOX 24
      -0.5  -0.5  -0.5
       0.5   0.5   0.5
```

The bounding box for the layer, just so that readers don't have to scan the PNTS chunk to find the extents.

```
   POLS 64
      FACE
      4  0  1  2  3
      4  0  4  5  1
      4  1  5  6  2
      4  3  2  6  7
      4  0  3  7  4
      4  4  7  6  5
```

The POLS chunk contains a list of polygons. A "polygon" in this context is anything that can be described using an ordered list of vertices. A POLS of type FACE contains ordinary polygons, but the POLS type can also be CURV, PTCH, MBAL or BONE, for example.

The high 6 bits of the vertex count for each polygon are reserved for flags, which in effect limits the number of vertices per polygon to 1023. Don't forget to mask the high bits when reading the vertex count. The flags are currently only defined for CURVs.

The point indexes following the vertex count refer to the points defined in the most recent PNTS chunk. Each index can be a 2-byte or a 4-byte integer. If the high order (first) byte of the index is *not* 0xFF, the index is 2 bytes long. This allows values up to 65279 to be stored in 2 bytes. If the high order byte *is* 0xFF, the index is 4 bytes long and its value is in the low three bytes (`index & 0x00FFFFFF`). The maximum value for 4-byte indexes is 16,777,215 (2^{24} - 1). Objects with more than 2^{24} vertices can be stored using multiple pairs of PNTS and POLS chunks.

The cube has 6 square faces each defined by 4 vertices. LightWave polygons are single-sided by default (double-sidedness is a possible surface property). The vertices are listed in clockwise order as viewed from the visible side, starting with a convex vertex. (The normal is defined as the cross product of the first and last edges.)

```
PTAG 28
   SURF
   0  0
   1  0
   2  0
   3  0
   4  0
   5  0
```

The PTAG chunk associates tags with polygons. In this case, it identifies which surface is assigned to each polygon. The first number in each pair is a 0-based index into the most recent POLS chunk, and the second is a 0-based index into the TAGS chunk.

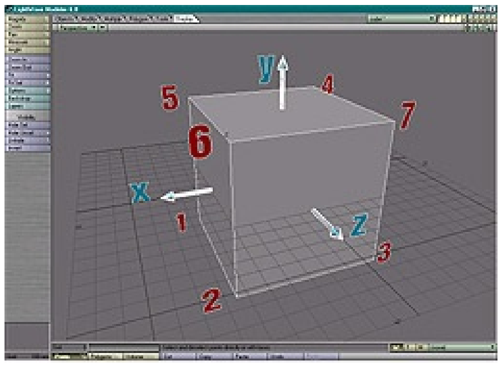
```
SURF 42
   "Default"
   ""
   COLR 14
      0.78431  0.78431  0.78431
      0
   DIFF 6
      1.0
      0
```

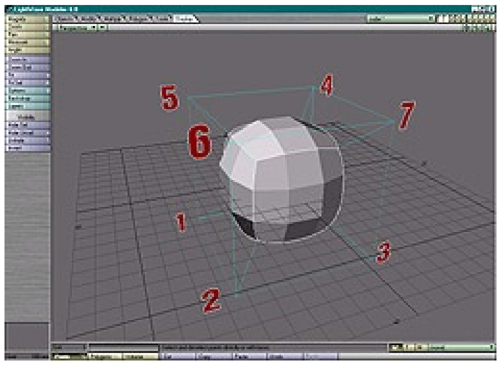
The description of each surface is stored in a SURF chunk. The only items guaranteed to be in a SURF chunk are the names of the surface and of its parent. The parent name is often empty, but if it's not, any surface parameters not defined in the SURF can be inherited from the parent's SURF. When there's no parent, undefined parameters are assigned default values. ("Default" is just the default *name*. If you aren't concerned about confusing people, you're free to give non-default values to a surface with this name.)

Following the name fields is a collection of subchunks, each of which defines a property of the surface. Like IFF chunks, SURF subchunks start with a 4-byte ID followed by a chunk size, but the size is 2 bytes in length rather than 4. Although subchunks tend to be quite small, SURFs may contain a large number of them, as we'll see later.

The 0 at the end of the COLR and DIFF subchunks indicates that these surface attributes are not enveloped (don't vary over time). We'll change that later, too.

Subpatched Cube





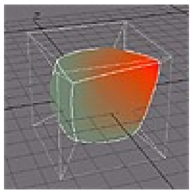
Loading the original cube (left) and activating subdivision patches turns the cube into a control cage for the patches (right). (The numbers in the figure are the point indexes. The hidden corner is point 0.) In the object file, the only difference between these two objects is the polygon type ID in the POLS chunk. For the subpatch version of the cube, the ID is PTCH rather than FACE.

```
POLS 64
   PTCH
   ...
```

In all other respects the files are identical. The geometry that results from subdivision is determined interactively by the user through settings in LightWave. The method used to generate the patches is proprietary, but it produces results similar to other subdivision surface methods. The LightWave plug-in API includes functions for reading the subpatch geometry. Subpatches can also be frozen, after which they are ordinary polygons that can be saved explicitly as FACEs.

Vertex Maps

VMAP chunks associate vectors with the points in the most recent PNTS chunk.



The vectors can contain texture coordinates, weights, colors, or anything else that it makes sense to assign to a vertex. A subpatch weight map (type MNVW), for example, can be used to alter the shape of subpatch geometry by "pulling" it toward control cage vertices with higher weight values. MNVW VMAPs have a dimension of 1, meaning that they contain a single value (the weight) per vertex.

```
   56 4D 41 50 00 00 00 12 4D 4E 56 57 00 01 62 61   VMAP    MNVW  ba
   73 65 00 00 00 07 3F 49 C6 6E                     se

      VMAP 18
         MNVW
         1
         "base"
         7  0.78818
```

The image illustrates the effect of this VMAP, prosaically named "base," on our subpatched cube. A single non-zero weight has been assigned to vertex 7.

Meatballs?

In addition to FACEs and PTCHs, POLS can also store curves, bones and metaballs (sometimes spooneristically referred to as meatballs).

The CURV type holds the vertices of Catmull-Rom splines. The low two flag bits of the vertex count indicate whether the endpoints are part of the curve or just continuity control points. Curves are currently ignored by the renderer, so their use is limited to modeling. BONE polygons are line segments created in Modeler that can be converted to bones in Layout. MBALs are single-point polygons. The points are associated with a VMAP of type MBAL that contains the radius of influence of each metaball.

Envelopes

The potential complexity of surface information becomes apparent when we start adding envelopes and textures to the definitions of surface parameters. In our first example of this, envelopes are added to the color and luminosity channels of the Default surface of our basic cube. Four ENVL chunks are added to the file (three for the color channel).

```
   45 4E 56 4C 00 00 00 70 00 01 4E 41 4D 45 00 08   ENVL      NAME
   43 6F 6C 6F 72 2E 52 00 54 59 50 45 00 02 04 0A   Color.R TYPE
```

```
50 52 45 20 00 02 00 01 4B 45 59 20 00 08 00 00   PRE     KEY
00 00 3F 48 C8 C9 53 50 41 4E 00 10 54 43 42 20         SPAN  TCB
00 00 00 00 00 00 00 00 00 00 00 00 4B 45 59 20               KEY
00 08 3F 80 00 00 3F 80 00 00 53 50 41 4E 00 10             SPAN
54 43 42 20 00 00 00 00 00 00 00 00 00 00 00 00   TCB
50 4F 53 54 00 02 00 01                           POST
```

```
   ENVL 112
      1
      NAME 8    Color.R
      TYPE 2    0x040A
      PRE  2    1
      KEY  8    0.0  0.78431
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    1.0  1.0
      SPAN 16   TCB   0.0  0.0  0.0
      POST 2    1
```

Note that the PRE, KEY, and TCB IDs include a trailing space.

The envelopes for the red, green and blue components of the color channel are written to separate, contiguous ENVL chunks. The type code contains flags indicating that the envelope is represented to the user as a percentage and that this is the first component of a (color) vector. The pre and post behavior codes control what happens outside the range of the keys, and in this case they're set to keep the value constant.

This envelope contains two keys. Each KEY subchunk contains a time in seconds and a value, and the corresponding SPAN subchunk stores the interpolation parameters and identifies the type of interpolation between the key and its predecessor. The parameters for TCB curves are the tension, continuity and bias, all 0.0 in this example.

The envelopes for the green and blue channels are very similar.

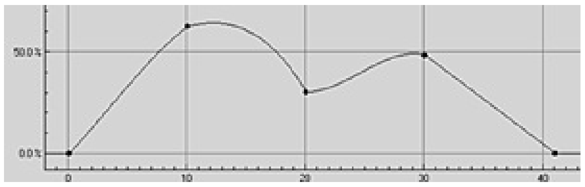
```
45 4E 56 4C 00 00 00 70 00 02 4E 41 4D 45 00 08   ENVL      NAME
43 6F 6C 6F 72 2E 47 00 54 59 50 45 00 02 04 0B   Color.G TYPE
50 52 45 20 00 02 00 01 4B 45 59 20 00 08 00 00   PRE     KEY
00 00 3F 48 C8 C9 53 50 41 4E 00 10 54 43 42 20         SPAN  TCB
00 00 00 00 00 00 00 00 00 00 00 00 4B 45 59 20               KEY
00 08 3F 80 00 00 3F 00 00 00 53 50 41 4E 00 10             SPAN
54 43 42 20 00 00 00 00 00 00 00 00 00 00 00 00   TCB
50 4F 53 54 00 02 00 01                           POST
```

```
   ENVL 112
      2
      NAME 8    Color.G
      TYPE 2    0x040B
      PRE  2    1
      KEY  8    0.0  0.78431
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    1.0  0.5
      SPAN 16   TCB   0.0  0.0  0.0
      POST 2    1

45 4E 56 4C 00 00 00 70 00 03 4E 41 4D 45 00 08   ENVL      NAME
43 6F 6C 6F 72 2E 42 00 54 59 50 45 00 02 04 0C   Color.B TYPE
50 52 45 20 00 02 00 01 4B 45 59 20 00 08 00 00   PRE     KEY
00 00 3F 48 C8 C9 53 50 41 4E 00 10 54 43 42 20         SPAN  TCB
00 00 00 00 00 00 00 00 00 00 00 00 4B 45 59 20               KEY
00 08 3F 80 00 00 00 00 00 00 53 50 41 4E 00 10             SPAN
54 43 42 20 00 00 00 00 00 00 00 00 00 00 00 00   TCB
50 4F 53 54 00 02 00 01                           POST

   ENVL 112
      3
      NAME 8    Color.B
      TYPE 2    0x040C
      PRE  2    1
      KEY  8    0.0  0.78431
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    1.0  0.0
      SPAN 16   TCB   0.0  0.0  0.0
      POST 2    1
```

The envelope for the luminosity channel contains 5 keys.

```
45 4E 56 4C 00 00 00 D0 00 04 4E 41 4D 45 00 0C   ENVL      NAME
4C 75 6D 69 6E 6F 73 69 74 79 00 00 54 59 50 45   Luminosity  TYPE
00 02 04 00 50 52 45 20 00 02 00 01 4B 45 59 20       PRE     KEY
00 08 00 00 00 00 00 00 00 00 53 50 41 4E 00 10             SPAN
54 43 42 20 00 00 00 00 00 00 00 00 00 00 00 00   TCB
4B 45 59 20 00 08 3E AA AA AB 3F 20 00 00 53 50   KEY           SP
41 4E 00 0C 42 45 5A 49 3F 5F 0B 6D 3E 1A E6 07   AN  BEZI
4B 45 59 20 00 08 3F 2A AA AB 3E 9A E6 07 53 50   KEY           SP
41 4E 00 10 54 43 42 20 BF 00 00 00 BF 80 00 00   AN  TCB
3F 80 00 00 4B 45 59 20 00 08 3F 80 00 00 3E F7       KEY
A6 F5 53 50 41 4E 00 10 54 43 42 20 00 00 00 00     SPAN  TCB
00 00 00 00 00 00 00 00 4B 45 59 20 00 08 3F AE           KEY
EE EF 00 00 00 00 53 50 41 4E 00 04 4C 49 4E 45         SPAN  LINE
50 4F 53 54 00 02 00 01                           POST

   ENVL 208
      4
      NAME 12   Luminosity
      TYPE 2    0x0400
      PRE  2    1
      KEY  8    0.0  0.0
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    0.33333  0.625
      SPAN 12   BEZI  0.87127  0.15127
      KEY  8    0.66667  0.28804
      SPAN 16   TCB   -0.5  -1.0  1.0
      KEY  8    1.0  0.4837
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    1.36667  0.0
      SPAN 4    LINE
      POST 2    1
```

The span between the first and second keys is a Bezier curve requiring two parameters. The third key's tension, continuity and bias are non-zero. The span between the fourth and fifth keys has been set to Linear, which requires no interpolation parameters.



These envelopes are referenced by index in the SURF chunk.

```
53 55 52 46 00 00 00 36 44 65 66 61 75 6C 74 00   SURF    Default
00 00 43 4F 4C 52 00 0E 3F 48 C8 C9 3F 48 C8 C9     COLR
3F 48 C8 C9 00 01 4C 55 4D 49 00 06 00 00 00 00         LUMI
00 04 44 49 46 46 00 06 3F 80 00 00 00 00           DIFF

   SURF 54
      "Default"
      ""
      COLR 14    0.78431  0.78431  0.78431  1
      LUMI 6     0.0  4
      DIFF 6     1.0  0
```

The COLR subchunk includes a reference to ENVL 1 (which implies ENVL 2 and 3 as well), and the LUMI subchunk refers to ENVL 4. As we've seen, the 0 in DIFF means that it doesn't have an envelope.

Textures

In the next example, an image is planar mapped onto the north (+Z) face of our basic cube, which is assigned a new surface called "Screen".

```
54 41 47 53 00 00 00 10 44 65 66 61 75 6C 74 00   TAGS    Default
53 63 72 65 65 6E 00 00                           Screen

   TAGS 16
      "Default"
      "Screen"
```

The new surface's name is written into TAGS. Note that because the length of the name (including the 0 byte that terminates the string) is odd, a pad byte is appended so that the next chunk starts on an even byte. All strings in object files are written this way.

```
50 54 41 47 00 00 00 1C 53 55 52 46 00 00 00 00   PTAG    SURF
00 01 00 00 00 02 00 00 00 03 00 01 00 04 00 00
00 05 00 00

   PTAG 28
      SURF
      0  0
      1  0
      2  0
      3  1
      4  0
      5  0
```

The PTAG shows that polygon 3 is assigned the Screen surface, while the others still have the surface named Default.

```
43 4C 49 50 00 00 00 1E 00 00 00 01 53 54 49 4C   CLIP        STIL
00 14 49 6D 61 67 65 73 2F 74 65 73 74 62 61 72     Images/testbar
73 2E 69 66 66 00                                 s.iff

  CLIP 30
     1
     STIL 20
        "Images/testbars.iff"
```

Information about the mapped image is stored in a CLIP chunk. This one happens to be a still, but it could also be an image sequence (ISEQ) or an animation (ANIM), and it could be modified by one or more image processing operators. This filename is relative to the current content directory, and it's written in a platform-neutral format.

```
53 55 52 46 00 00 01 48 53 63 72 65 65 6E 00 00   SURF    Screen
00 00 43 4F 4C 52 00 0E 3F 48 C8 C9 3F 48 C8 C9     COLR
3F 48 C8 C9 00 00 44 49 46 46 00 06 3F 80 00 00         DIFF
00 00 53 50 45 43 00 06 00 00 00 00 00 00 42 4C     SPEC        BL
4F 4B 01 0C 49 4D 41 50 00 32 80 00 43 48 41 4E   OK  IMAP    CHAN
00 04 43 4F 4C 52 4F 50 41 43 00 08 00 00 3F 80     COLROPAC
00 00 00 00 45 4E 41 42 00 02 00 01 4E 45 47 41       ENAB    NEGA
00 02 00 00 41 58 49 53 00 02 00 01 54 4D 41 50       AXIS    TMAP
00 68 43 4E 54 52 00 0E 00 00 00 00 00 00 00 00     CNTR
00 00 00 00 00 00 53 49 5A 45 00 0E 3F 80 00 00         SIZE
3F 80 00 00 3F 80 00 00 00 00 52 4F 54 41 00 0E             ROTA
00 00 00 00 00 00 00 00 00 00 00 00 00 00 46 41                 FA
4C 4C 00 10 00 00 00 00 00 00 00 00 00 00 00 00   LL
00 00 00 00 4F 52 45 46 00 08 28 6E 6F 6E 65 29       OREF  (none)
00 00 43 53 59 53 00 02 00 00 50 52 4F 4A 00 02     CSYS    PROJ
00 00 41 58 49 53 00 02 00 02 49 4D 41 47 00 02     AXIS    IMAG
00 01 57 52 41 50 00 04 00 01 00 01 57 52 50 57     WRAP      WRPW
00 06 3F 80 00 00 00 00 57 52 50 48 00 06 3F 80           WRPH
00 00 00 00 41 41 53 54 00 06 00 01 3F 80 00 00       AAST
50 49 58 42 00 02 00 01 53 54 43 4B 00 06 00 00   PIXB    STCK
00 00 00 00 54 41 4D 50 00 06 3F 80 00 00 00 00       TAMP

  SURF 328
     "Screen"
     ""
     COLR 14   0.78431  0.78431  0.78431  0
     DIFF 6    1.0  0
     SPEC 6    0.0  0
     BLOK 268
        IMAP 50
           "\x80"
           CHAN 4    COLR
           OPAC 8    0  1.0  0
```

```
      ENAB 2    1
      NEGA 2    0
      AXIS 2    1
   TMAP 104
      CNTR 14   0.0  0.0  0.0
         0
      SIZE 14   1.0  1.0  1.0  0
      ROTA 14   0.0  0.0  0.0  0
      FALL 16   0  0.0  0.0  0.0  0
      OREF 8    "(none)"
      CSYS 2    0
   PROJ 2    0
   AXIS 2    2
   IMAG 2    1
   WRAP 4    1  1
   WRPW 6    1.0  0
   WRPH 6    1.0  0
   AAST 6    1  1.0
   PIXB 2    1
   STCK 6    0.0  0
   TAMP 6    1.0  0
```

Texture layers are stored in `BLOK`s inside the `SURF` chunk. A `BLOK` begins with a header subchunk that identifies the texture type of the layer. For `IMAP` (image map) and `PROC` (procedural) layer types, the `BLOK` also contains a `TMAP` that describes the mapping from world or object space to texture space. And the `BLOK` contains other subchunks specific to the layer type.

The first field of the `BLOK` header (the `IMAP` subchunk) is called an *ordinal string*. When multiple textures are applied to a surface channel, the ordinal string determines the order in which they're evaluated. Object readers can sort `BLOK`s by using `strcmp` to compare the ordinal strings. Writers can generate ordinal strings with the following function.

```
void make_ord( int nbloks, int index, unsigned char *ord )
{
   int i, d;

   for ( i = 8, d = 16; i < 128; i *= 2 )
      if ( i >= nbloks ) break;
      d /= 2;
   }
   ord[ 0 ] = 128 + index * d;
   ord[ 1 ] = 0;
}
```

`nbloks` is the total number of `BLOK`s, and `index` is a number between 0 and `nbloks - 1`. This works for `nbloks <= 128`. In the unlikely event that you need to apply more than 128 texture layers to a single surface channel, you can extend the method of this function to create ordinal strings with two or more characters.

(You probably will have to do this if you're generating new ordinals to fit with ones LightWave has made. With enough fooling around, users can cause LightWave to generate ordinal strings that are fairly long. They're valid, just longer than strictly necessary, unlike the ones generated by our `make_ord` function.)

The rest of the BLOK header identifies which surface channel the texture layer modifies, the layer's opacity, whether it's enabled, whether its output is inverted, and what the displacement axis is. The subchunks following the TMAP are specific to IMAP layers. The AXIS subchunk in the IMAP header is only used if the texture is applied as a displacement map. The AXIS in the body of the BLOK is the one that determines the image mapping plane. The IMAG subchunk contains a CLIP index that identifies the image.

If we add a procedural texture layer that uses the built-in Turbulence function, the BLOK for the new layer looks like the following.

```
42 4C 4F 4B 00 DE 50 52 4F 43 00 32 90 00 43 48   BLOK  PROC 2  CH
41 4E 00 04 43 4F 4C 52 4F 50 41 43 00 08 00 00   AN  COLROPAC
3F 80 00 00 00 00 45 4E 41 42 00 02 00 01 4E 45         ENAB    NE
47 41 00 02 00 00 41 58 49 53 00 02 00 01 54 4D   GA    AXIS    TM
41 50 00 68 43 4E 54 52 00 0E 00 00 00 00 00 00   AP  CNTR
00 00 00 00 00 00 00 00 53 49 5A 45 00 0E 3F 80           SIZE
00 00 3F 80 00 00 3F 80 00 00 00 00 52 4F 54 41               ROTA
00 0E 00 00 00 00 00 00 00 00 00 00 00 00 00 00
46 41 4C 4C 00 10 00 00 00 00 00 00 00 00 00 00   FALL
00 00 00 00 00 00 4F 52 45 46 00 08 28 6E 6F 6E         OREF  (non
65 29 00 00 43 53 59 53 00 02 00 00 41 58 49 53   e)  CSYS    AXIS
00 02 00 02 56 41 4C 55 00 0C 3F 4C CC CD 3F 4C       VALU
CC CD 3F 4C CC CD 46 55 4E 43 00 18 54 75 72 62         FUNC  Turb
75 6C 65 6E 63 65 00 00 00 00 00 03 00 00 00 00   ulence
3F 00 00 00
```

```
      BLOK 222
         PROC 50
            "\x90"
            CHAN 4    COLR
            OPAC 8    0  1.0  0
            ENAB 2    1
            NEGA 2    0
            AXIS 2    1
         TMAP 104
            CNTR 14   0.0  0.0  0.0
               0
            SIZE 14   1.0  1.0  1.0  0
            ROTA 14   0.0  0.0  0.0  0
            FALL 16   0  0.0  0.0  0.0  0
            OREF 8    "(none)"
            CSYS 2    0
         AXIS 2    2
         VALU 12   0.8  0.8  0.8
         FUNC 24   "Turbulence"  3  0.0  0.5
```

Note the similarities to the image map layer. The BLOK header begins with PROC, and the ordinal string (`"\x90"`) puts this texture after the image map (`"\x80"`), but otherwise the header is the same as the IMAP header, and we also have a TMAP with the same contents. The FUNC subchunk names the procedural and lists its parameters, in this case the number of frequencies or octaves, the contrast level, and the small power.

You might also notice that the structure of a BLOK closely follows the layout of the Texture Editor interface. The header corresponds to the items above the first divider in the editor, the TMAP to the stuff below the second divider, and the other subchunks to the type-specific settings in between.

UV Mapping

The next example uses UV mapping to paint an image onto one of the cube faces, equivalent to the earlier planar mapping example.

UV mapped textures use VMAPs of type TXUV to hold the U and V texture coordinates. TXUV VMAPs have a dimension of 2.

```
   56 4D 41 50 00 00 00 3A 54 58 55 56 00 02 55 56   VMAP    TXUV  UV
   20 54 65 78 74 75 72 65 00 00 00 02 00 00 00 00    Texture
   00 00 00 00 00 03 3F 80 00 00 00 00 00 00 00 06
   00 00 00 00 3F 80 00 00 00 07 3F 80 00 00 3F 80
   00 00

      VMAP 58
         TXUV
         2
         "UV Texture"
         2  0.0  0.0
         3  1.0  0.0
         6  0.0  1.0
         7  1.0  1.0

   53 55 52 46 00 00 01 5C 55 56 45 78 61 6D 70 6C   SURF    UVExampl
   65 00 00 00 43 4F 4C 52 00 0E 3F 48 C8 C9 3F 48   e   COLR
   C8 C9 3F 48 C8 C9 00 00 44 49 46 46 00 06 3F 80           DIFF
   00 00 00 00 53 50 45 43 00 06 00 00 00 00 00 00       SPEC
   42 4C 4F 4B 01 1E 49 4D 41 50 00 32 80 00 43 48   BLOK  IMAP    CH
   41 4E 00 04 43 4F 4C 52 4F 50 41 43 00 08 00 00   AN  COLROPAC
   3F 80 00 00 00 00 45 4E 41 42 00 02 00 01 4E 45         ENAB    NE
   47 41 00 02 00 00 41 58 49 53 00 02 00 01 54 4D   GA    AXIS    TM
   41 50 00 68 43 4E 54 52 00 0E 00 00 00 00 00 00   AP  CNTR
   00 00 00 00 00 00 00 00 53 49 5A 45 00 0E 3F 80           SIZE
   00 00 3F 80 00 00 3F 80 00 00 00 00 52 4F 54 41               ROTA
   00 0E 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   46 41 4C 4C 00 10 00 00 00 00 00 00 00 00 00 00   FALL
   00 00 00 00 00 00 4F 52 45 46 00 08 28 6E 6F 6E         OREF  (non
   65 29 00 00 43 53 59 53 00 02 00 00 50 52 4F 4A   e)  CSYS    PROJ
   00 02 00 05 41 58 49 53 00 02 00 02 49 4D 41 47       AXIS    IMAG
   00 02 00 01 57 52 41 50 00 04 00 01 00 01 57 52       WRAP      WR
```

```
50 57 00 06 3F 80 00 00 00 00 57 52 50 48 00 06   PW        WRPH
3F 80 00 00 00 00 56 4D 41 50 00 0C 55 56 20 54         VMAP  UV T
65 78 74 75 72 65 00 00 41 41 53 54 00 06 00 01   exture  AAST
3F 80 00 00 50 49 58 42 00 02 00 01 53 54 43 4B       PIXB    STCK
00 06 00 00 00 00 00 00 54 41 4D 50 00 06 3F 80           TAMP
00 00 00 00
```

```
   SURF 348
      "UVExample"
      ""
      COLR 14   0.78431  0.78431  0.78431  0
      DIFF 6    1.0  0
      SPEC 6    0.0  0
      BLOK 286
         IMAP 50
            "\x80"
            CHAN 4    COLR
            OPAC 8    0  1.0  0
            ENAB 2    1
            NEGA 2    0
            AXIS 2    1
         TMAP 104
            CNTR 14   0.0  0.0  0.0
               0
            SIZE 14   1.0  1.0  1.0  0
            ROTA 14   0.0  0.0  0.0  0
            FALL 16   0  0.0  0.0  0.0  0
            OREF 8    "(none)"
            CSYS 2    0
         PROJ 2    5
         AXIS 2    2
         IMAG 2    1
         WRAP 4    1  1
         WRPW 6    1.0  0
         WRPH 6    1.0  0
         VMAP 12   "UV Texture"
         AAST 6    1  1.0
         PIXB 2    1
         STCK 6    0.0  0
         TAMP 6    1.0  0
```

The surface description is nearly identical to the planar mapping case. Although most of it will be ignored, we still have a complete TMAP subchunk. The value in PROJ (projection) has changed from 0 (planar) to 5 (UV), and a VMAP subchunk identifies the TXUV VMAP by name.

Discontinuous UVs

When the UV mapping is topologically equivalent to a cylinder or a sphere, a seam is formed where the edges of the map meet. Problems arise when the renderer needs to interpolate between points on opposite sides of this UV international date line. The seam is a *discontinuity*, a place where the mapping instantly jumps from one value to another.

To deal with this, a second set of UV coordinates is assigned to points of the object near the seam. This creates an area of overlap that allows the coordinate interpolation to be calculated correctly. These secondary UVs are used to render only those polygons on which the seam lies.

The following example wraps a single image around four faces of the basic cube.

```
56 4D 41 50 00 00 00 62 54 58 55 56 00 02 55 56   VMAP    TXUV  UV
20 54 65 78 74 75 72 65 00 00 00 00 3E 00 00 00    Texture
00 00 00 00 00 01 3E C0 00 00 00 00 00 00 00 02
3E 00 00 00 3F 80 00 00 00 03 3E C0 00 00 3F 80
00 00 00 04 3F 20 00 00 00 00 00 00 00 05 3F 60
00 00 00 00 00 00 00 06 3F 20 00 00 3F 80 00 00
00 07 3F 60 00 00 3F 80 00 00

   VMAP 98
      TXUV
      2
      "UV Texture"
      0  0.125  0.0
      1  0.375  0.0
      2  0.125  1.0
      3  0.375  1.0
      4  0.625  0.0
      5  0.875  0.0
      6  0.625  1.0
      7  0.875  1.0
```

The secondary UV coordinates are stored in a VMAD chunk.

```
56 4D 41 44 00 00 00 2A 54 58 55 56 00 02 55 56   VMAD    TXUV  UV
20 54 65 78 74 75 72 65 00 00 00 05 00 04 BE 00    Texture
00 00 00 00 00 00 00 07 00 04 BE 00 00 00 3F 80
00 00

   VMAD 42
      TXUV
      2
      "UV Texture"
      5  4  -0.125  0.0
      7  4  -0.125  1.0
```

Each entry contains both a point and a polygon index. The seam in this case falls in the middle of polygon 4, and the VMAD says that when rendering any part of this polygon, the VMAP values for points 5 and 7 should be replaced with the ones in the VMAD for those points. Other polygons that share those points are unaffected by this replacement.

VMADs were added to the file format with version 6.5 of LightWave. Although they will be used most often for UV mapping, they can be used to supplement other kinds of vertex mapping. They can also be applied without a corresponding VMAP.

Plug-ins

Object files can store instances of several kinds of plug-ins. The plug-in data is stored in different places, depending on the plug-in class. Channel modifiers are stored in CHAN subchunks inside an ENVL chunk. Here, the NoisyChannel plug-in has been applied to the red channel of a surface.

```
45 4E 56 4C 00 00 00 BA 00 01 4E 41 4D 45 00 08   ENVL      NAME
43 6F 6C 6F 72 2E 52 00 54 59 50 45 00 02 04 0A   Color.R TYPE
50 52 45 20 00 02 00 01 4B 45 59 20 00 08 00 00   PRE     KEY
00 00 3F 48 C8 C9 53 50 41 4E 00 10 54 43 42 20         SPAN  TCB
00 00 00 00 00 00 00 00 00 00 00 00 4B 45 59 20               KEY
00 08 3F 22 22 22 3F 8E 9B D3 53 50 41 4E 00 10             SPAN
54 43 42 20 00 00 00 00 00 00 00 00 00 00 00 00   TCB
4B 45 59 20 00 08 3F 8C CC CD 3F 22 C8 59 53 50   KEY            SP
41 4E 00 10 54 43 42 20 00 00 00 00 00 00 00 00   AN  TCB
00 00 00 00 50 4F 53 54 00 02 00 01 43 48 41 4E       POST    CHAN
00 20 4E 6F 69 73 79 43 68 61 6E 6E 65 6C 00 00     NoisyChannel
00 00 00 00 00 00 3F 80 00 00 3F 80 00 00 00 00
00 00
```

```
   ENVL 186
      1
      NAME 8    Color.R
      TYPE 2    0x040A
      PRE  2    1
      KEY  8    0.0  0.78431
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    0.63333  1.1141
      SPAN 16   TCB   0.0  0.0  0.0
      KEY  8    1.1  0.63587
      SPAN 16   TCB   0.0  0.0  0.0
      POST 2    1
      CHAN 32
         "NoisyChannel"
         0
         0.0  1.0  1.0  0.0
```

The value following the name is a flags word. If the first bit is set, the plug-in is disabled. The data that follows the flags belongs to the plug-in, and unless the author has documented it, it can only be interpreted by the plug-in. Except for size, which must be even and can't exceed 65536 bytes, including the name, the file format places no restrictions on what plug-ins can write here.

Shader information is stored inside a BLOK of type SHDR.

```
53 55 52 46 00 00 00 72 44 65 66 61 75 6C 74 00   SURF    Default
00 00 43 4F 4C 52 00 0E 3F 71 BE 8C 3F 48 C8 C9     COLR
3F 48 C8 C9 00 01 44 49 46 46 00 06 3F 80 00 00         DIFF
00 00 42 4C 4F 4B 00 42 53 48 44 52 00 0A 80 00     BLOK  SHDR
45 4E 41 42 00 02 00 01 46 55 4E 43 00 2C 44 65   ENAB    FUNC  De
6D 6F 5F 42 6C 6F 74 63 68 00 3E CC CC CD 00 00   mo_Blotch
00 00 3F 4C CC CD 00 00 00 00 00 00 00 00 00 00
```

```
   00 00 3F 80 00 00 3F 00 00 00

      SURF 114
         "Default"
         ""
         COLR 14   0.78431  0.78431  0.78431  0
         DIFF 6    1.0  0
         SPEC 6    0.0  0
         BLOK 66
            SHDR 10
               "\x80"
               ENAB 2    1
            FUNC 44
               "Demo_Blotch"
               0.4  0.0  0.8  0.0  0.0  0.0  1.0  0.5
```

The SHDR header contains an ENAB subchunk that determines whether the shader is enabled. The FUNC subchunk holds the plug-in name and its data.

"EA IFF 85" Standard for Interchange Format Files

Document Date:	January 14, 1985
From:	Jerry Morrison, Electronic Arts
Status of Standard:	Released and in use

1. Introduction

Standards are Good for Software Developers

As home computer hardware evolves to better and better media machines, the demand increases for higher quality, more detailed data. Data development gets more expensive, requires more expertise and better tools, and has to be shared across projects. Think about several ports of a product on one CD-ROM with 500M Bytes of common data!

Development tools need standard interchange file formats. Imagine scanning in images of "player" shapes, moving them to a paint program for editing, then incorporating them into a game. Or writing a theme song with a Macintosh score editor and incorporating it into an Amiga game. The data must at times be transformed, clipped, filled out, and moved across machine kinds. Media projects will depend on data transfer from graphic, music, sound effect, animation, and script tools.

Standards are Good for Software Users

Customers should be able to move their own data between independently developed software products. And they should be able to buy data libraries usable across many such products. The types of data objects to exchange are open-ended and include plain and formatted text, raster and structured graphics, fonts, music, sound effects, musical instrument descriptions, and animation.

The problem with expedient file formats typically memory dumps is that they're too provincial. By designing data for one particular use (e.g. a screen snapshot), they preclude future expansion (would you like a full page picture? a multi-page document?). In neglecting the possibility that other programs might read their data, they fail to save contextual information (how many bit planes? what resolution?). Ignoring that other programs might create such files, they're intolerant of extra data (texture palette for a picture editor), missing data (no color map), or minor variations (smaller image). In practice, a filed representation should rarely mirror an in-memory representation. The former should be designed for longevity; the latter to optimize the manipulations of a particular program. The same filed data will be read into different memory formats by different programs.

The IFF philosophy: "A little behind-the-scenes conversion when programs read and write files is far better than NxM explicit conversion utilities for highly specialized formats."

So we need some standardization for data interchange among development tools and products. The more developers that adopt a standard, the better for all of us and our customers.

Here is "EA IFF 1985"

Here is our offering: Electronic Arts' IFF standard for Interchange File Format. The full name is "EA IFF 1985". Alternatives and justifications are included for certain choices. Public domain subroutine packages and utility programs are available to make it easy to write and use IFF-compatible programs.

Part 1 introduces the standard. Part 2 presents its requirements and background. Parts 3, 4, and 5 define the primitive data types, FORMS, and LISTS, respectively, and how to define new high level types. Part 6 specifies the top level file structure. Appendix A is included for quick reference and Appendix B names the committee responsible for this standard.

References

American National Standard Additional Control Codes for Use with ASCII, ANSI standard 3.64-1979 for an 8-bit character set. See also ISO standard 2022 and ISO/DIS standard 6429.2.

Amiga™ is a trademark of Commodore-Amiga, Inc.

C, A Reference Manual, Samuel P. Harbison and Guy L. Steele Jr., Tartan Laboratories. Prentice-Hall, Englewood Cliffs, NJ, 1984.

Compiler Construction, An Advanced Course, edited by F. L. Bauer and J. Eickel (Springer-Verlag, 1976). This book is one of many sources for information on recursive descent parsing.

DIF Technical Specification © 1981 by Software Arts, Inc. DIF™ is the format for spreadsheet data interchange developed by Software Arts, Inc. DIF™ is a trademark of Software Arts, Inc.

Electronic Arts™ is a trademark of Electronic Arts.

"FTXT" IFF Formatted Text, from Electronic Arts. IFF supplement document for a text format.

Inside Macintosh © 1982, 1983, 1984, 1985 Apple Computer, Inc., a programmer's reference manual.
Apple® is a trademark of Apple Computer, Inc.
Macintosh™ is a trademark licensed to Apple Computer, Inc.

"ILBM" IFF Interleaved Bitmap, from Electronic Arts. IFF supplement document for a raster image format.

M68000 16/32-Bit Microprocessor Programmer's Reference Manual © 1984, 1982, 1980, 1979 by Motorola, Inc.

PostScript Language Manual © 1984 Adobe Systems Incorporated.
PostScript™ is a trademark of Adobe Systems, Inc.
Times and Helvetica® are trademarks of Allied Corporation.

InterScript: A Proposal for a Standard for the Interchange of Editable Documents © 1984 Xerox Corporation.
Introduction to InterScript © 1985 Xerox Corporation.

---

2. Background for Designers

Part 2 is about the background, requirements, and goals for the standard. It's geared for people who want to design new types of IFF objects. People just interested in using the standard may wish to skip this part.

What Do We Need?

A standard should be long on prescription and short on overhead. It should give lots of rules for designing programs and data files for synergy. But neither the programs nor the files should cost too much more than the expedient variety. While we're looking to a future with CD-ROMs and perpendicular recording, the standard must work well on floppy disks.

For program portability, simplicity, and efficiency, formats should be designed with more than one implementation style in mind. (In practice, pure stream I/O is adequate although random access makes it easier to write files.) It ought to be possible to read one of many objects in a file without scanning all the preceding data. Some programs need to read and play out their data in real time, so we need good compromises between generality and efficiency.

As much as we need standards, they can't hold up product schedules. So we also need a kind of decentralized extensibility where any software developer can define and refine new object types without some "standards authority" in the loop. Developers must be able to extend existing formats in a forward- and backward-compatible way. A central repository for design information and example programs can help us take full advantage of the standard.

For convenience, data formats should heed the restrictions of various processors and environments. E.g. word-alignment greatly helps 68000 access at insignificant cost to 8088 programs.

Other goals include the ability to share common elements over a list of objects and the ability to construct composite objects containing other data objects with structural information like directories.

And finally, *"Simple things should be simple and complex things should be possible."* --Alan Kay.

Think Ahead

Let's think ahead and build programs that read and write files for each other and for programs yet to be designed. Build data formats to last for future computers so long as the overhead is acceptable. This extends the usefulness and life of today's programs and data.

To maximize interconnectivity, the standard file structure and the specific object formats must all be general and extensible. Think ahead when designing an object. It should serve many purposes and allow many programs to store and read back all the information they need; even squeeze in custom data. Then a programmer can store the available data and is encouraged to include fixed contextual details. Recipient programs can read the needed parts, skip unrecognized stuff, default missing data, and use the stored context to help transform the data as needed.

Scope

IFF addresses these needs by defining a standard file structure, some initial data object types, ways to define new types, and rules for accessing these files. We can accomplish a great deal by writing programs according to this standard, but don't expect direct compatibility with existing software. We'll need conversion programs to bridge the gap from the old world.

IFF is geared for computers that readily process information in 8-bit bytes. It assumes a "physical layer" of data storage and transmission that reliably maintains "files" as strings of 8-bit bytes. The standard treats a "file" as a container of data bytes and is independent of how to find a file and whether it has a byte count.

This standard does not by itself implement a clipboard for cutting and pasting data between programs. A clipboard needs software to mediate access, to maintain a "contents version number" so programs can detect updates, and to manage the data in "virtual memory".

Data Abstraction

The basic problem is how to represent information in a way that's program-independent, compiler- independent, machine-independent, and device-independent.

The computer science approach is "data abstraction", also known as "objects", "actors", and "abstract data types". A data abstraction has a "concrete representation" (its storage format), an "abstract representation" (its capabilities and uses), and access procedures that isolate all the calling software from the concrete representation. Only the access procedures touch the data storage. Hiding mutable details behind an interface is called "information hiding". What data abstraction does is abstract from details of implementing the object, namely the selected storage representation and algorithms for manipulating it.

The power of this approach is modularity. By adjusting the access procedures we can extend and restructure the data without impacting the interface or its callers. Conversely, we can extend and restructure the interface and callers without making existing data obsolete. It's great for interchange!

But we seem to need the opposite: fixed file formats for all programs to access. Actually, we could file data abstractions ("filed objects") by storing the data and access procedures together. We'd have to encode the access procedures in a standard machine-independent programming language la PostScript. Even still, the interface can't evolve freely since we can't update all copies of the access procedures. So we'll have to design our abstract representations for limited evolution and occasional revolution (conversion).

In any case, today's microcomputers can't practically store data abstractions. They can do the next best thing: store arbitrary types of data in "data chunks", each with a type identifier and a length count. The type identifier is a reference by name to the access procedures (any local implementation). The length count enables storage-level object operations like "copy" and "skip to next" independent of object type.

Chunk writing is straightforward. Chunk reading requires a trivial parser to scan each chunk and dispatch to the proper access/conversion procedure. Reading chunks nested inside other chunks requires recursion, but no lookahead or backup.

That's the main idea of IFF. There are, of course, a few other details.

Previous Work

Where our needs are similar, we borrow from existing standards.

Our basic need to move data between independently developed programs is similar to that addressed by the Apple Macintosh desk scrap or "clipboard" [Inside Macintosh chapter "Scrap Manager"]. The Scrap Manager works closely with the Resource Manager, a handy filer and swapper for data objects (text strings, dialog window templates, pictures, fonts) including types yet to be designed [Inside Macintosh chapter "Resource Manager"]. The Resource Manager is a kin to Smalltalk's object swapper.

We will probably write a Macintosh desk accessory that converts IFF files to and from the Macintosh clipboard for quick and easy interchange with programs like MacPaint and Resource Mover.

Macintosh uses a simple and elegant scheme of 4-character "identifiers" to identify resource types, clipboard format types, file types, and file creator programs. Alternatives are unique ID numbers assigned by a central authority or by hierarchical authorities, unique ID numbers generated by algorithm, other fixed length character strings, and variable length strings. Character string identifiers double as readable signposts in data files and programs. The choice of 4 characters is a good tradeoff between storage space, fetch/compare/store time, and name space size. We'll honor Apple's designers by adopting this scheme.

"PICT" is a good example of a standard structured graphics format (including raster images) and its many uses [Inside Macintosh chapter "QuickDraw"]. Macintosh provides QuickDraw routines in ROM to create, manipulate, and display PICTs. Any application can create a PICT by simply asking QuickDraw to record a sequence of drawing commands. Since it's just as easy to ask QuickDraw to render a PICT to a screen or a printer, it's very effective to pass them between programs, say from an illustrator to a word processor. An important feature is the ability to store "comments" in a PICT which QuickDraw will ignore. Actually, it passes them to your optional custom "comment handler".

PostScript, Adobe's print file standard, is a more general way to represent any print image (which is a specification for putting marks on paper) [PostScript Language Manual]. In fact, PostScript is a full-fledged programming language. To interpret a PostScript program is to render a document on a raster output device. The language is defined in layers: a lexical layer of identifiers, constants, and operators; a layer of reverse polish semantics including scope rules and a way to define new subroutines; and a printing-specific layer of built-in identifiers and operators for rendering graphic images. It is clearly a powerful (Turing equivalent) image definition language. PICT and a subset of PostScript are candidates for structured graphics standards.

A PostScript document can be printed on any raster output device (including a display) but cannot generally be edited. That's because the original flexibility and constraints have been discarded. Besides, a PostScript program may use arbitrary computation to supply parameters like placement and size to each operator. A QuickDraw PICT, in comparison, is a more restricted format of graphic primitives parameterized by constants. So a PICT can be edited at the level of the primitives, e.g. move or thicken a line. It cannot be edited at the higher level of, say, the bar chart data which generated the picture.

PostScript has another limitation: Not all kinds of data amount to marks on paper. A musical instrument description is one example. PostScript is just not geared for such uses.

"DIF" is another example of data being stored in a general format usable by future programs [DIF Technical Specification]. DIF is a format for spreadsheet data interchange. DIF and PostScript are both expressed in plain ASCII text files. This is very handy for printing, debugging, experimenting, and transmitting across modems. It can have substantial cost in compaction and read/write work, depending on use. We won't store IFF files this way but we could define an ASCII alternate representation with a converter program.

InterScript is Xerox' standard for interchange of editable documents [Introduction to InterScript]. It approaches a harder problem: How to represent editable word processor documents that may contain formatted text, pictures, cross-references like figure numbers, and even highly specialized objects like mathematical equations? InterScript aims to define one standard representation for each kind of information. Each InterScript-compatible editor is supposed to preserve the objects it doesn't understand

and even maintain nested cross-references. So a simple word processor would let you edit the text of a fancy document without discarding the equations or disrupting the equation numbers.

Our task is similarly to store high level information and preserve as much content as practical while moving it between programs. But we need to span a larger universe of data types and cannot expect to centrally define them all. Fortunately, we don't need to make programs preserve information that they don't understand. And for better or worse, we don't have to tackle general-purpose cross-references yet.

---

3. Primitive Data Types

Atomic components such as integers and characters that are interpretable directly by the CPU are specified in one format for all processors. We chose a format that's most convenient for the Motorola MC68000 processor [M68000 16/32-Bit Microprocessor Programmer's Reference Manual].

N.B.: Part 3 dictates the format for "primitive" data types where and only where used in the overall file structure and standard kinds of chunks (Cf. Chunks). The number of such occurrences will be small enough that the costs of conversion, storage, and management of processor- specific files would far exceed the costs of conversion during I/O by "foreign" programs. A particular data chunk may be specified with a different format for its internal primitive types or with processor- or environment- specific variants if necessary to optimize local usage. Since that hurts data interchange, it's not recommended. (Cf. Designing New Data Sections, in Part 4.)

Alignment

All data objects larger than a byte are aligned on even byte addresses relative to the start of the file. This may require padding. Pad bytes are to be written as zeros, but don't count on that when reading.

This means that every odd-length "chunk" (see below) must be padded so that the next one will fall on an even boundary. Also, designers of structures to be stored in chunks should include pad fields where needed to align every field larger than a byte. Zeros should be stored in all the pad bytes.

Justification: Even-alignment causes a little extra work for files that are used only on certain processors but allows 68000 programs to construct and scan the data in memory and do block I/O. You just add an occasional pad field to data structures that you're going to block read/write or else stream read/write an extra byte. And the same source code works on all processors. Unspecified alignment, on the other hand, would force 68000 programs to (dis)assemble word and long-word data one byte at a time. Pretty cumbersome in a high level language. And if you don't conditionally compile that out for other processors, you won't gain anything.

Numbers

Numeric types supported are two's complement binary integers in the format used by the MC68000 processor high byte first, high word first the reverse of 8088 and 6502 format. They could potentially include signed and unsigned 8, 16, and 32 bit integers but the standard only uses the following:

```
UBYTE    8 bits unsigned
WORD    16 bits signed
UWORD   16 bits unsigned
LONG    32 bits signed
```

The actual type definitions depend on the CPU and the compiler. In this document, we'll express data type definitions in the C programming language. [See C, A Reference Manual.] In 68000 Lattice C:

```
typedef unsigned char   UBYTE;  /*  8 bits unsigned */
typedef short           WORD;   /* 16 bits signed   */
typedef unsigned short  UWORD;  /* 16 bits unsigned */
typedef long            LONG;   /* 32 bits signed   */
```

Characters

The following character set is assumed wherever characters are used, e.g. in text strings, IDs, and TEXT chunks (see below).

Characters are encoded in 8-bit ASCII. Characters in the range NUL (hex 0) through DEL (hex 7F) are well defined by the 7-bit ASCII standard. IFF uses the graphic group ' ' (SP, hex 20) through '~' (hex 7E).

Most of the control character group hex 01 through hex 1F have no standard meaning in IFF. The control character LF (hex 0A) is defined as a "newline" character. It denotes an intentional line break, that is, a paragraph or line terminator. (There is no way to store an automatic line break. That is strictly a function of the margins in the environment the text is placed.) The control character ESC (hex 1B) is a reserved escape character under the rules of ANSI standard 3.64-1979 American National Standard Additional Control Codes for Use with ASCII, ISO standard 2022, and ISO/DIS standard 6429.2.

Characters in the range hex 7F through hex FF are not globally defined in IFF. They are best left reserved for future standardization. But note that the FORM type FTXT (formatted text) defines the meaning of these characters within FTXT forms. In particular, character values hex 7F through hex 9F are control codes while characters hex A0 through hex FF are extended graphic characters like 'é', as per the ISO and ANSI standards cited above. [See the supplementary document "FTXT" IFF Formatted Text.]

Dates

A "creation date" is defined as the date and time a stream of data bytes was created. (Some systems call this a "last modified date".) Editing some data changes its creation date. Moving the data between volumes or machines does not.

The IFF standard date format will be one of those used in MS-DOS, Macintosh, or Amiga DOS (probably a 32-bit unsigned number of seconds since a reference point). *Issue:* Investigate these three.

Type IDs

A "type ID", "property name", "FORM type", or any other IFF identifier is a 32-bit value: the concatenation of four ASCII characters in the range ' ' (SP, hex 20) through '~' (hex 7E). Spaces (hex 20) should not precede printing characters; trailing spaces are ok. Control characters are forbidden.

```
typedef CHAR ID[4];
```

IDs are compared using a simple 32-bit case-dependent equality test.

Data section type IDs (aka FORM types) are restriced IDs. (Cf. Data Sections.) Since they may be stored in filename extensions (Cf. Single Purpose Files) lower case letters and punctuation marks are forbidden. Trailing spaces are ok.

Carefully choose those four characters when you pick a new ID. Make them mnemonic so programmers can look at an interchange format file and figure out what kind of data it contains. The name space makes it possible for developers scattered around the globe to generate ID values with minimal collisions so long as they choose specific names like "MUS4" instead of general ones like "TYPE" and "FILE". EA will "register" new FORM type IDs and format descriptions as they're devised, but collisions will be improbable so there will be no pressure on this "clearinghouse" process. Appendix A has a list of currently defined IDs.

Sometimes it's necessary to make data format changes that aren't backward compatible. Since IDs are used to denote data formats in IFF, new IDs are chosen to denote revised formats. Since programs won't read chunks whose IDs they don't recognize (see Chunks, below), the new IDs keep old programs from stumbling over new data. The conventional way to chose a "revision" ID is to increment the last character if it's a digit or else change the last character to a digit. E.g. first and second revisions of the ID "XY" would be "XY1" and "XY2". Revisions of "CMAP" would be "CMA1" and "CMA2".

Chunks

Chunks are the building blocks in the IFF structure. The form expressed as a C typedef is:

```
typedef struct {
   ID      ckID;
   LONG    ckSize; /* sizeof(ckData) */
   UBYTE   ckData[/* ckSize */];
} Chunk;
```

We can diagram an example chunk a "CMAP" chunk containing 12 data bytes like this:

ckID:	'CMAP'
ckSize:	12
ckData:	0, 0, 0, 32 0, 0, 64, 0 0, 0, 64, 0 (12 bytes)

The fixed header part means "Here's a type ckID chunk with ckSize bytes of data."

The ckID identifies the format and purpose of the chunk. As a rule, a program must recognize ckID to interpret ckData. It should skip over all unrecognized chunks. The ckID also serves as a format version number as long as we pick new IDs to identify new formats of ckData (see above).

The following ckIDs are universally reserved to identify chunks with particular IFF meanings: "`LIST`", "`FORM`", "`PROP`", "`CAT `", and "`    `". The special ID "`    `" (4 spaces) is a ckID for "filler" chunks, that is, chunks that fill space but have no meaningful contents. The IDs "`LIS1`" through "`LIS9`", "`FOR1`" through "`FOR9`", and "`CAT1`" through "`CAT9`" are reserved for future "version number" variations. All IFF-compatible software must account for these 23 chunk IDs. Appendix A has a list of predefined IDs.

The ckSize is a logical block size how many data bytes are in ckData. If ckData is an odd number of bytes long, a 0 pad byte follows which is not included in ckSize. (Cf. Alignment.) A chunk's total physical size is ckSize rounded up to an even number plus the size of the header. So the smallest chunk is 8 bytes long with ckSize = 0. For the sake of following chunks, programs must respect every chunk's ckSize as a virtual end-of-file for reading its ckData even if that data is malformed, e.g. if nested contents are truncated.

We can describe the syntax of a chunk as a regular expression with "#" representing the ckSize, i.e. the length of the following {braced} bytes. The "[0]" represents a sometimes needed pad byte. (The regular expressions in this document are collected in Appendix A along with an explanation of notation.)

```
Chunk ::= ID #{ UBYTE* } [0]
```

One chunk output technique is to stream write a chunk header, stream write the chunk contents, then random access back to the header to fill in the size. Another technique is to make a preliminary pass over the data to compute the size, then write it out all at once.

Strings, String Chunks, and String Properties

In a string of ASCII text, `LF` denotes a forced line break (paragraph or line terminator). Other control characters are not used. (Cf. Characters.)

The ckID for a chunk that contains a string of plain, unformatted text is "`TEXT`". As a practical matter, a text string should probably not be longer than 32767 bytes. The standard allows up to 231 - 1 bytes.

When used as a data property (see below), a text string chunk may be 0 to 255 characters long. Such a string is readily converted to a C string or a Pascal `STRING[255]`. The ckID of a property must be the property name, not "`TEXT`".

When used as a part of a chunk or data property, restricted C string format is normally used. That means 0 to 255 characters followed by a `NUL` byte (ASCII value 0).

Data Properties

Data properties specify attributes for following (non-property) chunks. A data property essentially says "identifier = value", for example "XY = (10, 200)", telling something about following chunks. Properties may only appear inside data sections ("`FORM`" chunks, cf. Data Sections) and property sections ("`PROP`" chunks, cf. Group `PROP`).

The form of a data property is a special case of Chunk. The ckID is a property name as well as a property type. The ckSize should be small since data properties are intended to be accumulated in RAM when reading a file. (256 bytes is a reasonable upper bound.) Syntactically:

```
    Property ::= Chunk
```

When designing a data object, use properties to describe context information like the size of an image, even if they don't vary in your program. Other programs will need this information.

Think of property settings as assignments to variables in a programming language. Multiple assignments are redundant and local assignments temporarily override global assignments. The order of assignments doesn't matter as long as they precede the affected chunks. (Cf. LISTs, CATs, and Shared Properties.)

Each object type (FORM type) is a local name space for property IDs. Think of a "CMAP" property in a "FORM ILBM" as the qualified ID "ILBM.CMAP". Property IDs specified when an object type is designed (and therefore known to all clients) are called "standard" while specialized ones added later are "nonstandard".

Links

Issue: A standard mechanism for "links" or "cross references" is very desirable for things like combining images and sounds into animations. Perhaps we'll define "link" chunks within FORMs that refer to other FORMs or to specific chunks within the same and other FORMs. This needs further work. EA IFF 1985 has no standard link mechanism.

For now, it may suffice to read a list of, say, musical instruments, and then just refer to them within a musical score by index number.

File References

Issue: We may need a standard form for references to other files. A "file ref" could name a directory and a file in the same type of operating system as the ref's originator. Following the reference would expect the file to be on some mounted volume. In a network environment, a file ref could name a server, too.

Issue: How can we express operating-system independent file refs?

Issue: What about a means to reference a portion of another file? Would this be a "file ref" plus a reference to a "link" within the target file?

---

4. Data Sections

The first thing we need of a file is to check: Does it contain IFF data and, if so, does it contain the kind of data we're looking for? So we come to the notion of a "data section".

A "data section" or IFF "FORM" is one self-contained "data object" that might be stored in a file by itself. It is one high level data object such as a picture or a sound effect. The IFF structure "FORM" makes it self-identifying. It could be a composite object like a musical score with nested musical instrument descriptions.

Group FORM

A data section is a chunk with ckID "FORM" and this arrangement:

```
FORM       ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
FormType   ::= ID
LocalChunk ::= Property | Chunk
```

The ID "FORM" is a syntactic keyword like "struct" in C. Think of a "struct ILBM" containing a field "CMAP". If you see "FORM" you'll know to expect a FORM type ID (the structure name, "ILBM" in this example) and a particular contents arrangement or "syntax" (local chunks, FORMs, LISTs, and CATs). (LISTs and CATs are discussed in part 5, below.) A "FORM ILBM", in particular, might contain a local chunk "CMAP", an "ILBM.CMAP" (to use a qualified name).

So the chunk ID "FORM" indicates a data section. It implies that the chunk contains an ID and some number of nested chunks. In reading a FORM, like any other chunk, programs must respect its ckSize as a virtual end-of-file for reading its contents, even if they're truncated.

The FormType (or FORM type) is a restricted ID that may not contain lower case letters or punctuation characters. (Cf. Type IDs. Cf. Single Purpose Files.)

The type-specific information in a FORM is composed of its "local chunks": data properties and other chunks. Each FORM type is a local name space for local chunk IDs. So "CMAP" local chunks in other FORM types may be unrelated to "ILBM.CMAP". More than that, each FORM type defines semantic scope. If you know what a FORM ILBM is, you'll know what an ILBM.CMAP is.

Local chunks defined when the FORM type is designed (and therefore known to all clients of this type) are called "standard" while specialized ones added later are "nonstandard".

Among the local chunks, property chunks give settings for various details like text font while the other chunks supply the essential information. This distinction is not clear cut. A property setting cancelled by a later setting of the same property has effect only on data chunks in between. E.g. in the sequence:

```
   prop1 = x  (propN = value)*  prop1 = y
```

where the propNs are not prop1, the setting prop1 = x has no effect.

The following universal chunk IDs are reserved inside any FORM: "LIST", "FORM", "PROP", "CAT ", "   ", "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9". (Cf. Chunks. Cf. Group LIST. Cf. Group PROP.) For clarity, these universal chunk names may not be FORM type IDs, either.

Part 5, below, talks about grouping FORMs into LISTs and CATs. They let you group a bunch of FORMs but don't impose any particular meaning or constraints on the grouping. Read on.

Composite FORMS

A FORM chunk inside a FORM is a full-fledged data section. This means you can build a composite object like a multi-frame animation sequence from available picture FORMs and sound effect FORMs. You can insert additional chunks with information like frame rate and frame count.

Using composite FORMs, you leverage on existing programs that create and edit the component FORMs. Those editors may even look into your composite object to copy out its type of component, although it'll be the rare program that's fancy enough to do that. Such editors are not allowed to replace their component objects within your composite object. That's because the IFF standard lets you specify consistency requirements for the composite FORM such as maintaining a count or a directory of the components. Only programs that are written to uphold the rules of your FORM type should create or modify such FORMs.

Therefore, in designing a program that creates composite objects, you are strongly requested to provide a facility for your users to import and export the nested FORMs. Import and export could move the data through a clipboard or a file.

Here are several existing FORM types and rules for defining new ones.

FTXT

An FTXT data section contains text with character formatting information like fonts and faces. It has no paragraph or document formatting information like margins and page headers. FORM FTXT is well matched to the text representation in Amiga's Intuition environment. See the supplemental document "FTXT" IFF Formatted Text.

ILBM

"ILBM" is an InterLeaved BitMap image with color map; a machine-independent format for raster images. FORM ILBM is the standard image file format for the Commodore-Amiga computer and is useful in other environments, too. See the supplemental document "ILBM" IFF Interleaved Bitmap.

PICS

The data chunk inside a "PICS" data section has ID "PICT" and holds a QuickDraw picture. Issue: Allow more than one PICT in a PICS? See Inside Macintosh chapter "QuickDraw" for details on PICTs and how to create and display them on the Macintosh computer.

The only standard property for PICS is "XY", an optional property that indicates the position of the PICT relative to "the big picture". The contents of an XY is a QuickDraw Point.

Note: PICT may be limited to Macintosh use, in which case there'll be another format for structured graphics in other environments.

Other Macintosh Resource Types

Some other Macintosh resource types could be adopted for use within IFF files; perhaps MWRT, ICN, ICN#, and STR#.

Issue: Consider the candidates and reserve some more IDs.

Designing New Data Sections

Supplemental documents will define additional object types. A supplement needs to specify the object's purpose, its FORM type ID, the IDs and formats of standard local chunks, and rules for generating and interpreting the data. It's a good idea to supply typedefs and an example source program that accesses the new object. See "ILBM" IFF Interleaved Bitmap for a good example.

Anyone can pick a new FORM type ID but should reserve it with Electronic Arts at their earliest convenience. [Issue: EA contact person? Hand this off to another organization?] While decentralized format definitions and extensions are possible in IFF, our preference is to get design consensus by committee, implement a program to read and write it, perhaps tune the format, and then publish the format with example code. Some organization should remain in charge of answering questions and coordinating extensions to the format.

If it becomes necessary to revise the design of some data section, its FORM type ID will serve as a version number (Cf. Type IDs). E.g. a revised "VDEO" data section could be called "VDE1". But try to get by with compatible revisions within the existing FORM type.

In a new FORM type, the rules for primitive data types and word-alignment (Cf. Primitive Data Types) may be overriden for the contents of its local chunks but not for the chunk structure itself if your documentation spells out the deviations. If machine-specific type variants are needed, e.g. to store vast numbers of integers in reverse bit order, then outline the conversion algorithm and indicate the variant inside each file, perhaps via different FORM types. Needless to say, variations should be minimized.

In designing a FORM type, encapsulate all the data that other programs will need to interpret your files. E.g. a raster graphics image should specify the image size even if your program always uses 320 x 200 pixels x 3 bitplanes. Receiving programs are then empowered to append or clip the image rectangle, to add or drop bitplanes, etc. This enables a lot more compatibility.

Separate the central data (like musical notes) from more specialized information (like note beams) so simpler programs can extract the central parts during read-in. Leave room for expansion so other programs can squeeze in new kinds of information (like lyrics). And remember to keep the property chunks manageably short let's say 2 256 bytes.

When designing a data object, try to strike a good tradeoff between a super-general format and a highly-specialized one. Fit the details to at least one particular need, for example a raster image might as well store pixels in the current machine's scan order. But add the kind of generality that makes it usable with foreseeable hardware and software. E.g. use a whole byte for each red, green, and blue color value even if this year's computer has only 4-bit video DACs. Think ahead and help other programs so long as the overhead is acceptable. E.g. run compress a raster by scan line rather than as a unit so future programs can swap images by scan line to and from secondary storage.

Try to design a general purpose "least common multiple" format that encompasses the needs of many programs without getting too complicated. Let's coalesce our uses around a few such formats widely separated in the vast design space. Two factors make this flexibility and simplicity practical. First, file storage space is getting very plentiful, so compaction is not a priority. Second, nearly any locally-performed data conversion work during file reading and writing will be cheap compared to the I/O time.

It must be ok to copy a LIST or FORM or CAT intact, e.g. to incorporate it into a composite FORM. So any kind of internal references within a FORM must be relative references. They could be relative to the start of the containing FORM, relative from the referencing chunk, or a sequence number into a collection.

With composite FORMS, you leverage on existing programs that create and edit the components. If you write a program that creates composite objects, please provide a facility for your users to import and export the nested FORMS. The import and export functions may move data through a separate file or a clipboard.

Finally, don't forget to specify all implied rules in detail.

5. LISTs, CATs, and Shared Properties

Data often needs to be grouped together like a list of icons. Sometimes a trick like arranging little images into a big raster works, but generally they'll need to be structured as a first class group. The objects "LIST" and "CAT" are IFF-universal mechanisms for this purpose.

Property settings sometimes need to be shared over a list of similar objects. E.g. a list of icons may share one color map. LIST provides a means called "PROP" to do this. One purpose of a LIST is to define the scope of a PROP. A "CAT", on the other hand, is simply a concatenation of objects.

Simpler programs may skip LISTs and PROPs altogether and just handle FORMs and CATs. All "fully-conforming" IFF programs also know about "CAT ", "LIST", and "PROP". Any program that reads a FORM inside a LIST must process shared PROPs to correctly interpret that FORM.

Group CAT

A CAT is just an untyped group of data objects.

Structurally, a CAT is a chunk with chunk ID "CAT " containing a "contents type" ID followed by the nested objects. The ckSize of each contained chunk is essentially a relative pointer to the next one.

```
   CAT          ::= "CAT " #{ ContentsType (FORM | LIST | CAT)* }
   ContentsType ::= ID  -- a hint or an "abstract data type" ID
```

In reading a CAT, like any other chunk, programs must respect it's ckSize as a virtual end-of-file for reading the nested objects even if they're malformed or truncated.

The "contents type" following the CAT's ckSize indicates what kind of FORMs are inside. So a CAT of ILBMs would store "ILBM" there. It's just a hint. It may be used to store an "abstract data type". A CAT could just have blank contents ID ("   ") if it contains more than one kind of FORM.

CAT defines only the format of the group. The group's meaning is open to interpretation. This is like a list in LISP: the structure of cells is predefined but the meaning of the contents as, say, an association list depends on use. If you need a group with an enforced meaning (an "abstract data type" or Smalltalk "subclass"), some consistency constraints, or additional data chunks, use a composite FORM instead (Cf. Composite FORMS).

Since a CAT just means a concatenation of objects, CATS are rarely nested. Programs should really merge CATS rather than nest them.

Group LIST

A LIST defines a group very much like CAT but it also gives a scope for PROPS (see below). And unlike CATS, LISTS should not be merged without understanding their contents.

Structurally, a LIST is a chunk with ckID "LIST" containing a "contents type" ID, optional shared properties, and the nested contents (FORMS, LISTS, and CATS), in that order. The ckSize of each contained chunk is a relative pointer to the next one. A LIST is not an arbitrary linked list the cells are simply concatenated.

```
    LIST         ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
    ContentsType ::= ID
```

Group PROP

PROP chunks may appear in LISTS (not in FORMS or CATS). They supply shared properties for the FORMS in that LIST. This ability to elevate some property settings to shared status for a list of forms is useful for both indirection and compaction. E.g. a list of images with the same size and colors can share one "size" property and one "color map" property. Individual FORMS can override the shared settings.

The contents of a PROP is like a FORM with no data chunks:

```
   PROP ::= "PROP" #{ FormType Property* }
```

It means, "Here are the shared properties for FORM type <<FormType>>."

A LIST may have at most one PROP of a FORM type, and all the PROPS must appear before any of the FORMS or nested LISTS and CATS. You can have subsequences of FORMS sharing properties by making each subsequence a

LIST.

Scoping: Think of property settings as variable bindings in nested blocks of a programming language. Where in C you could write:

```
TEXT_FONT text_font = Courier;  /* program's global default */

File(); {
   TEXT_FONT text_font = TimesRoman;    /* shared setting       */
   {
      TEXT_FONT text_font = Helvetica;  /* local setting        */
      Print("Hello ");                  /* uses font Helvetica  */
   }

   {
      Print("there.");                  /* uses font TimesRoman */
   }
}
```

An IFF file could contain:

```
LIST {
   PROP TEXT {
      FONT {TimesRoman}       /* shared setting       */
   }

   FORM TEXT {
      FONT {Helvetica}        /* local setting        */
      CHRS {Hello }           /* uses font Helvetica  */
   }

   FORM TEXT {
      CHRS {there.}           /* uses font TimesRoman */
   }
}
```

The shared property assignments selectively override the reader's global defaults, but only for FORMs within the group. A FORM's own property assignments selectively override the global and group-supplied values. So when reading an IFF file, keep property settings on a stack. They're designed to be small enough to hold in main memory.

Shared properties are semantically equivalent to copying those properties into each of the nested FORMs right after their FORM type IDs.

Properties for LIST

Optional "properties for LIST" store the origin of the list's contents in a PROP chunk for the fake FORM type "LIST". They are the properties originating program "OPGM", processor family "OCPU", computer type "OCMP", computer serial number or network address "OSN ", and user name "UNAM". In our imperfect world, these could be called upon to distinguish between unintended variations of a data format or to work around bugs in particular originating/receiving program pairs. Issue: Specify the format of these properties.

A creation date could also be stored in a property but let's ask that file creating, editing, and transporting programs maintain the correct date in the local file system. Programs that move files between machine types are expected to copy across the creation dates.

---

6. Standard File Structure

File Structure Overview

An IFF file is just a single chunk of type FORM, LIST, or CAT. Therefore an IFF file can be recognized by its first 4 bytes: "FORM", "LIST", or "CAT ". Any file contents after the chunk's end are to be ignored.

Since an IFF file can be a group of objects, programs that read/write single objects can communicate to an extent with programs that read/write groups. You're encouraged to write programs that handle all the objects in a LIST or CAT. A graphics editor, for example, could process a list of pictures as a multiple page document, one page at a time.

Programs should enforce IFF's syntactic rules when reading and writing files. This ensures robust data transfer. The public domain IFF reader/writer subroutine package does this for you. A utility program "IFFCheck" is available that scans an IFF file and checks it for conformance to IFF's syntactic rules. IFFCheck also prints an outline of the chunks in the file, showing the ckID and ckSize of each. This is quite handy when building IFF programs. Example programs are also available to show details of reading and writing IFF files.

A merge program "IFFJoin" will be available that logically appends IFF files into a single CAT group. It "unwraps" each input file that is a CAT so that the combined file isn't nested CATs.

If we need to revise the IFF standard, the three anchoring IDs will be used as "version numbers". That's why IDs "FOR1" through "FOR9", "LIS1" through "LIS9", and "CAT1" through "CAT9" are reserved.

IFF formats are designed for reasonable performance with floppy disks. We achieve considerable simplicity in the formats and programs by relying on the host file system rather than defining universal grouping structures like directories for LIST contents. On huge storage systems, IFF files could be leaf nodes in a file structure like a B-tree. Let's hope the host file system implements that for us!

Thre are two kinds of IFF files: single purpose files and scrap files. They differ in the interpretation of multiple data objects and in the file's external type.

Single Purpose Files

A single purpose IFF file is for normal "document" and "archive" storage. This is in contrast with "scrap files" (see below) and temporary backing storage (non-interchange files).

The external file type (or filename extension, depending on the host file system) indicates the file's contents. It's generally the FORM type of the data contained, hence the restrictions on FORM type IDs.

Programmers and users may pick an "intended use" type as the filename extension to make it easy to filter for the relevant files in a filename requestor. This is actually a "subclass" or "subtype" that conveniently separates files of the same FORM type that have different uses. Programs cannot demand conformity to its expected subtypes without overly restricting data interchange since they cannot know about the subtypes to be used by future programs that users will want to exchange data with.

Issue: How to generate 3-letter MS-DOS extensions from 4-letter FORM type IDs?

Most single purpose files will be a single FORM (perhaps a composite FORM like a musical score containing nested FORMs like musical instrument descriptions). If it's a LIST or a CAT, programs should skip over unrecognized objects to read the recognized ones or the first recognized one. Then a program that can read a single purpose file can read something out of a "scrap file", too.

Scrap Files

A "scrap file" is for maximum interconnectivity in getting data between programs; the core of a clipboard function. Scrap files may have type "`IFF `" or filename extension "`.IFF`".

A scrap file is typically a `CAT` containing alternate representations of the same basic information. Include as many alternatives as you can readily generate. This redundancy improves interconnectivity in situations where we can't make all programs read and write super-general formats. [Inside Macintosh chapter "Scrap Manager".] E.g. a graphically-annotated musical score might be supplemented by a stripped down 4-voice melody and by a text (the lyrics).

The originating program should write the alternate representations in order of "preference": most preferred (most comprehensive) type to least preferred (least comprehensive) type. A receiving program should either use the first appearing type that it understands or search for its own "preferred" type.

A scrap file should have at most one alternative of any type. (A `LIST` of same type objects is ok as one of the alternatives.) But don't count on this when reading; ignore extra sections of a type. Then a program that reads scrap files can read something out of single purpose files.

Rules for Reader Programs

Here are some notes on building programs that read IFF files. If you use the standard IFF reader module "`IFFR.C`", many of these rules and details will be automatically handled. (See "Support Software" in Appendix A.) We recommend that you start from the example program "`ShowILBM.C`". You should also read up on recursive descent parsers. [See, for example, Compiler Construction, An Advanced Course.]

- The standard is very flexible so many programs can exchange data. This implies a program has to scan the file and react to what's actually there in whatever order it appears. An IFF reader program is a parser.

- For interchange to really work, programs must be willing to do some conversion during read-in. If the data isn't exactly what you expect, say, the raster is smaller than those created by your program, then adjust it. Similarly, your program could crop a large picture, add or drop bitplanes, and create/discard a mask plane. The program should give up gracefully on data that it can't convert.
- If it doesn't start with "FORM", "LIST", or "CAT ", it's not an IFF-85 file.
- For any chunk you encounter, you must recognize its type ID to understand its contents.
- For any FORM chunk you encounter, you must recognize its FORM type ID to understand the contained "local chunks". Even if you don't recognize the FORM type, you can still scan it for nested FORMs, LISTs, and CATs of interest.
- Don't forget to skip the pad byte after every odd-length chunk.
- Chunk types LIST, FORM, PROP, and CAT are generic groups. They always contain a subtype ID followed by chunks.
- Readers ought to handle a CAT of FORMs in a file. You may treat the FORMs like document pages to sequence through or just use the first FORM.
- Simpler IFF readers completely skip LISTs. "Fully IFF-conforming" readers are those that handle LISTs, even if just to read the first FORM from a file. If you do look into a LIST, you must process shared properties (in PROP chunks) properly. The idea is to get the correct data or none at all.
- The nicest readers are willing to look into unrecognized FORMs for nested FORM types that they do recognize. For example, a musical score may contain nested instrument descriptions and an animation file may contain still pictures.

Note to programmers: Processing PROP chunks is not simple! You'll need some background in interpreters with stack frames. If this is foreign to you, build programs that read/write only one FORM per file. For the more intrepid programmers, the next paragraph summarizes how to process LISTs and PROPs. See the general IFF reader module "IFFR.C" and the example program "ShowILBM.C" for details.

Allocate a stack frame for every LIST and FORM you encounter and initialize it by copying the stack frame of the parent LIST or FORM. At the top level, you'll need a stack frame initialized to your program's global defaults.

While reading each LIST or FORM, store all encountered properties into the current stack frame. In the example ShowILBM, each stack frame has a place for a bitmap header property ILBM.BMHD and a color map property ILBM.CMAP. When you finally get to the ILBM's BODY chunk, use the property settings accumulated in the current stack frame.

An alternate implementation would just remember PROPs encountered, forgetting each on reaching the end of its scope (the end of the containing LIST). When a FORM XXXX is encountered, scan the chunks in all remembered PROPs XXXX, in order, as if they appeared before the chunks actually in the FORM XXXX. This gets trickier if you read FORMs inside of FORMs.

Rules for Writer Programs

Here are some notes on building programs that write IFF files, which is much easier than reading them. If you use the standard IFF writer module "IFFW.C" (see "Support Software" in Appendix A), many of these rules and details will automatically be enforced. See the example program "Raw2ILBM.C".

- An IFF file is a single FORM, LIST, or CAT chunk.
- Any IFF-85 file must start with the 4 characters "FORM", "LIST", or "CAT ", followed by a LONG ckSize. There should be no data after the chunk end.
- Chunk types LIST, FORM, PROP, and CAT are generic. They always contain a subtype ID followed by chunks. These three IDs are universally reserved, as are "LIS1" through "LIS9", "FOR1" through "FOR9", "CAT1" through "CAT9", and "   ".
- Don't forget to write a 0 pad byte after each odd-length chunk.
- Four techniques for writing an IFF group: (1) build the data in a file mapped into virtual memory, (2) build the data in memory blocks and use block I/O, (3) stream write the data piecemeal and (don't forget!) random access back to set the group length count, and (4) make a preliminary pass to compute the length count then stream write the data.
- Do not try to edit a file that you don't know how to create. Programs may look into a file and copy out nested FORMs of types that they recognize, but don't edit and replace the nested FORMs and don't add or remove them. That could make the containing structure inconsistent.

You may write a new file containing items you copied (or copied and modified) from another IFF file, but don't copy structural parts you don't understand.

- You must adhere to the syntax descriptions in Appendex A. E.g. PROPs may only appear inside LISTs.

Appendix A. Reference

Type Definitions

The following C typedefs describe standard IFF structures. Declarations to use in practice will vary with the CPU and compiler. For example, 68000 Lattice C produces efficient comparison code if we define ID as a "LONG". A macro "MakeID" builds these IDs at compile time.

```
/* Standard IFF types, expressed in 68000 Lattice C.    */

typedef unsigned char  UBYTE;   /*  8 bits unsigned     */
typedef short          WORD;    /* 16 bits signed       */
typedef unsigned short UWORD;   /* 16 bits unsigned     */
typedef long           LONG;    /* 32 bits signed       */

typedef char ID[4];      /* 4 chars in ' ' through '~'   */

typedef struct {
    ID      ckID;
    LONG    ckSize; /* sizeof(ckData)       */
    UBYTE   ckData[/* ckSize */];
    } Chunk;

/* ID typedef and builder for 68000 Lattice C. */
typedef LONG ID;         /* 4 chars in ' ' through '~'   */
#define MakeID(a,b,c,d) ( (a)<<24 | (b)<<16 | (c)<<8 | (d) )

/* Globally reserved IDs. */
#define ID_FORM   MakeID('F','O','R','M')
#define ID_LIST   MakeID('L','I','S','T')
#define ID_PROP   MakeID('P','R','O','P')
#define ID_CAT    MakeID('C','A','T',' ')
#define ID_FILLER MakeID(' ',' ',' ',' ')
```

Syntax Definitions

Here's a collection of the syntax definitions in this document.

```
Chunk         ::= ID #{ UBYTE* } [0]

Property      := Chunk

FORM          ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
```

```
    FormType     := ID
    LocalChunk   := Property | Chunk

    CAT          ::= "CAT " #{ ContentsType (FORM | LIST | CAT)* }
    ContentsType ::= ID  -- a hint or an "abstract data type" ID

    LIST         ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
    PROP         ::= "PROP" #{ FormType Property* }
```

In this extended regular expression notation, the token "#" represents a ckSize LONG count of the following {braced} data bytes. Literal items are shown in "quotes", [square bracketed items] are optional, and "*" means 0 or more instances. A sometimes-needed pad byte is shown as "[0]".

Defined Chunk IDs

This is a table of currently defined chunk IDs. We may also borrow some Macintosh IDs and data formats.

- Group chunk IDs
  FORM, LIST, PROP, CAT.
- Future revision group chunk IDs
  FOR1 - FOR9, LIS1 - LIS9, CAT1 - CAT9.
- FORM type IDs
  (The above group chunk IDs may not be used for FORM type IDs.)
  (Lower case letters and punctuation marks are forbidden in FORM type IDs.)
  8SVX 8-bit sampled sound voice, ANBM animated bitmap, FNTR raster font, FNTV vector font, FTXT formatted text, GSCR general-use musical score, ILBM interleaved raster bitmap image, PDEF Deluxe Print page definition, PICS Macintosh picture, PLBM (obsolete), USCR Uhuru Sound Software musical score, UVOX Uhuru Sound Software Macintosh voice, SMUS simple musical score, VDEO Deluxe Video Construction Set video.
- Data chunk IDs
  "   ", TEXT, PICT.
- PROP LIST property IDs
  OPGM, OCPU, OCMP, OSN, UNAM.

Support Software

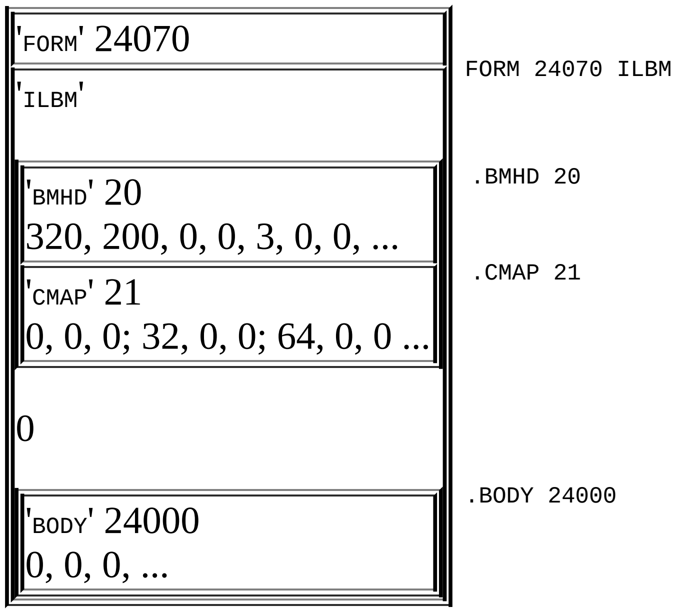
These public domain C source programs are available for use in building IFF-compatible programs:

- `IFF.H`, `IFFR.C`, `IFFW.C`
  IFF reader and writer package. These modules handle many of the details of reliably reading and writing IFF files.
- `IFFCheck.C`
  This handy utility program scans an IFF file, checks that the contents are well formed, and prints an outline of the chunks.
- `PACKER.H`, `Packer.C`, `UnPacker.C`
  Run encoder and decoder used for ILBM files.
- `ILBM.H`, `ILBMR.C`, `ILBMW.C`
  Reader and writer support routines for raster image `FORM ILBM`. `ILBMR` calls `IFFR` and `UnPacker`. `ILBMW` calls `IFFW` and `Packer`.
- `ShowILBM.C`
  Example caller of `IFFR` and `ILBMR` modules. This Commodore-Amiga program reads and displays a `FORM ILBM`.
- `Raw2ILBM.C`
  Example `ILBM` writer program. As a demonstration, it reads a raw raster image file and writes the image as a `FORM ILBM` file.
- `ILBM2Raw.C`
  Example `ILBM` reader program. Reads a `FORM ILBM` file and writes it into a raw raster image.
- `REMALLOC.H`, `Remalloc.c`
  Memory allocation routines used in these examples.
- `INTUALL.H`
  generic "include almost everything" include-file with the sequence of includes correctly specified.
- `READPICT.H`, `ReadPict.c`
  given an `ILBM` file, read it into a bitmap and a color map
- `PUTPICT.H`, `PutPict.c`
  given a bitmap and a color map, save it as an `ILBM` file.
- `GIO.H`, `Gio.c`
  generic I/O speedup package. Attempts to speed disk I/O by buffering writes and reads.
- `giocall.c`
  sample call to gio.
- `ilbmdump.c`
  reads in `ILBM` file, prints out ascii representation for including in C files.
- `bmprintc.c`

prints out a C-language representation of data for a bitmap.

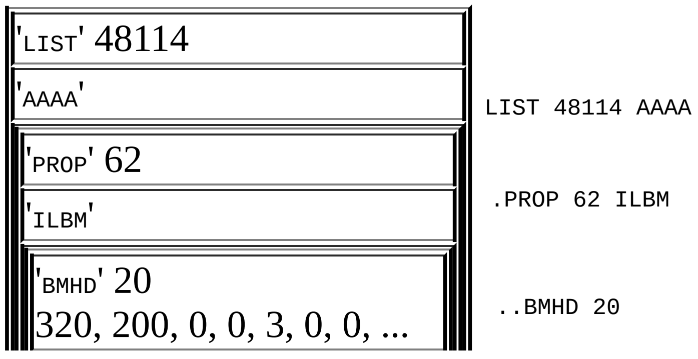
Example Diagrams

Here's a box diagram for an example IFF file, a raster image `FORM ILBM`. This `FORM` contains a bitmap header property chunk `BMHD`, a color map property chunk `CMAP`, and a raster data chunk `BODY`. This particular raster is 320 x 200 pixels x 3 bit planes uncompressed. The "`0`" after the `CMAP` chunk represents a zero pad byte; included since the `CMAP` chunk has an odd length. The text to the right of the diagram shows the outline that would be printed by the `IFFCheck` utility program for this particular file.

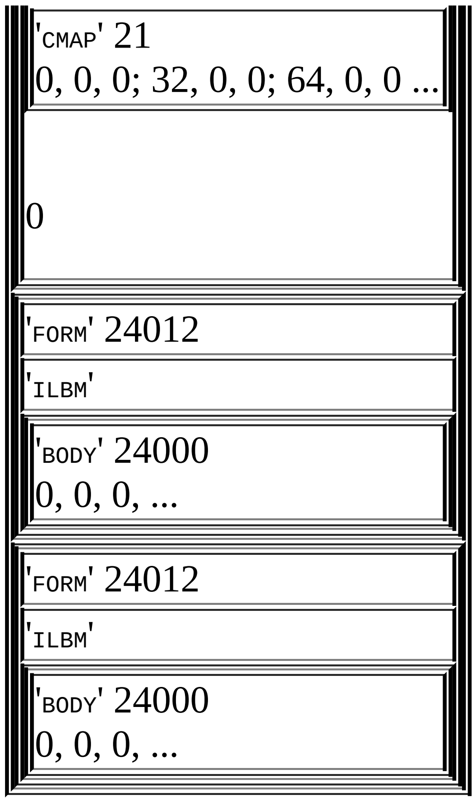


```
FORM 24070 ILBM

 .BMHD 20

 .CMAP 21

.BODY 24000
```

This second diagram shows a `LIST` of two `FORMS ILBM` sharing a common `BMHD` property and a common `CMAP` property. Again, the text on the right is an outline a la `IFFCheck`.



```
LIST 48114 AAAA

 .PROP 62 ILBM

  ..BMHD 20
```



```
..CMAP 21

.FORM 24012 ILBM

..BODY 24000

.FORM 24012 ILBM

..BODY 24000
```

Appendix B. Standards Committee

The following people contributed to the design of this IFF standard:

- Bob "Kodiak" Burns, Commodore-Amiga
- R. J. Mical, Commodore-Amiga
- Jerry Morrison, Electronic Arts
- Greg Riker, Electronic Arts
- Steve Shaw, Electronic Arts
- Barry Walsh, Commodore-Amiga

Flexible Precision Images

December 3, 2000

This document describes the FPBM (Flexible Precision Buffer Map) file format for images and animations introduced with LightWave 6.0.

Introduction

An *image* is a rectangular array of values. The image data generated by LightWave includes not only the red, green and blue levels of each pixel in the rendered image, but also values at each pixel for the alpha level, z-depth, shading, reflectivity, surface normal, 2D motion, and other buffers used during rendering. Most of these quantities are represented internally as floating-point numbers, and all may change over time. Existing image and animation file formats are inadequate for storing all of this information, which is the motivation for the new FPBM format.

The data channels in an FPBM are called *layer*s, and each layer can store values as 8-bit or 16-bit integers or as 32-bit floating-point numbers. A set of these layers for a given animation time is called a *frame*. An FPBM containing a single frame is a still image, and one containing a time sequence of frames is an animation.

(The descriptions here of the animation features of FPBM should be considered preliminary. They haven't been implemented in LightWave yet.)

Chunks

The FPBM format is based on the metaformat for binary files described in "EA IFF 85 Standard for Interchange Format Files." (See also ILBM, an earlier IFF image format.) The basic structural element in an IFF file is the *chunk*. A chunk consists of a four-byte ID tag, a four-byte chunk size, and *size* bytes of data. If the size is odd, the chunk is followed by a 0 pad byte, so that the next chunk begins on an even byte boundary. (The pad byte isn't counted in the size.)

A chunk ID is a sequence of 4 bytes containing 7-bit ASCII values, usually upper-case printable characters, used to identify the chunk's data. ID tags can be interpreted as unsigned integers for comparison purposes. They're typically constructed using macros like the following.

```
   #define CKID_(a,b,c,d) (((a)<<24)|((b)<<16)|((c)<<8)|(d))
   #define ID_FORM CKID_('F','O','R','M')
   #define ID_FPBM CKID_('F','P','B','M')
   ...
```

FPBM files start with the four bytes "`FORM`" followed by a four-byte integer giving the length of the file (minus 8) and the four byte ID "`FPBM`". The remainder of the file is a collection of chunks containing layer data.

To be read, IFF files must be *parsed*. FPBM files are pretty uniform, but in general the order in which chunks can occur in an IFF file isn't fixed. You may encounter chunks or layer types that aren't defined here, which you should be prepared to skip gracefully if you don't understand them. You can do this by using the chunk size to seek to the next chunk. And you may encounter chunk sizes that differ from those implied here. Readers must respect the chunk size. Missing data should be given default values, and extra data, which the reader presumably doesn't understand, should be skipped.

Data Basics

The data in an FPBM will be described in this document using C language conventions. Chunks will be represented as structures, and the values within each structure will be defined as the C basic types *short* or *float*. As used here, a short is a signed, two's complement, 16-bit integer, and a float is a 32-bit IEEE floating-point number. All data in an FPBM is written in big-endian (Motorola, Internet) byte order. Programs running in environments (primarily Microsoft Windows) that use a different byte order must swap bytes after reading and before writing.

File Structure

Structurally, FPBMs are quite simple.

```
FORM formsize FPBM
FPHD 28 FPHeader
for each frame
   FLEX 2 numLayers
   for each layer
      LYHD 20 LayerHeader
      LAYR datasize data
```

The header is followed by one or more frames. Each frame begins with a layer count, and this is followed by the layers. Each layer begins with a header describing the data it contains.

The following sections describe the FPHD, FLEX and LYHD chunks.

FPHD - Flexible Precision Header

The FPHeader contains information that applies globally to all of the frames in the file. It appears first in the file, after the FORM prefix and before the first frame.

```
typedef struct st_FPHeader {
   short width;
   short height;
   short numLayers;
   short numFrames;
   short numBuffers;
   short flags;
   short srcBytesPerLayerPixel;
   short pad2;
   float pixelAspect;
   float pixelWidth;
   float framesPerSecond;
} FPHeader;
```

width, height

Pixel dimensions of the image.

numLayers

The maximum number of layers per frame. Some layers may not be stored for all frames, since their contents may not differ from frame to frame.

numFrames

Number of frames. For still images, this will be 1.

numBuffers

Number of animation buffers. This affects the interpretation of delta encoded layer data. When the file is written for single buffered playback, the number of buffers is 1 and the deltas are relative to the previous frame. For double buffered playback, the deltas are relative to the frame "two frames back," since the new frame is drawn over the contents of the back buffer. This field is ignored for still images.

flags

One or more of the following flag values, combined using bitwise-or.

Source_Int (0 << 0)
Source_FP (1 << 0)

The "natural" representation of the data, or the way the data was stored before being written to the file, either integer (the default) or floating-point. This can differ from the way the data is actually stored in the file (specified for each layer in the LayerHeader `flags` field). Readers may wish to restore the data to its original representation using this information. It can also be used to indicate the precision of the source data.

InterlaceFlag (1 << 1)

Scanlines should be interlaced (field rendered) for playback. (The actual interlace state is stored for each layer in the LayerHeader.)

srcBytesPerLayerPixel

Use this in combination with the `Source_Int` and `Source_FP` flags to determine the "natural" data type for the data in the file, or the type in which the data was stored before it was written to the file. The most common values for this field are 1, 2 and 4. The actual size of a pixel in each layer is in the LayerHeader's `bytesPerLayerPixel` field.

pad2

Reserved for future use.

pixelAspect

Pixel aspect ratio expressed as width divided by height.

pixelWidth

Pixel width in millimeters. This fixes the size of the image for print. To calculate horizontal and vertical DPI (dots per inch) from this value,

```
hdpi = 25.4 / pixelWidth
vdpi = 25.4 / (pixelWidth * pixelAspect)
```

framesPerSecond

Number of frames per second for animations. Writers may set this to 0.0 for still images.

FLEX - Frame Header

The FrameHeader appears at the start of each frame. This chunk may grow in the future to include other information.

```
typedef struct st_FrameHeader {
   short numLayers;
} FrameHeader;
```

numLayers

Number of layers in this frame.

LYHD - Layer Header

The LayerHeader appears at the start of each layer to describe the layer's contents.

```
typedef struct st_LayerHeader {
   short flags;
   short layerType;
   short bytesPerLayerPixel;
   short compression;
   float blackPoint;
   float whitePoint;
   float gamma;
} LayerHeader;
```

flags

One or more of the following flags, combined using bitwise-or.

Layer_Int (0 << 0)
Layer_FP (1 << 0)
    Data in the layer is integer (the default) or floating-point.

Layer_Interlace (1 << 1)
    Scanlines are interlaced (field rendered).

Layer_EvenField (0 << 2)
Layer_OddField (1 << 2)
    Field dominance for interlaced layers. This indicates which field is displayed first in time.

layerType

The data channel contained in the layer. Possible values include

Layer_MONO 0
    Monochrome (grayscale) image channel.

Layer_RED 1
Layer_GREEN 2
Layer_BLUE 3
Layer_ALPHA 4
    Color and alpha channels.

Layer_OBJECT 5
    Object ID.

Layer_SURFACE 6
    Surface or material ID.

Layer_COVERAGE 7
    Object transparency/antialiasing.

Layer_ZDEPTH 8
Layer_WDEPTH 9
    The Z depth is the distance from the camera to the nearest object

visible in a pixel. Strictly speaking, this is the perpendicular distance from the plane defined by the camera's position and view vector. The W depth buffer contains the inverse of Z.

Layer_GEOMETRY 10
: The values in this buffer are the dot-products of the surface normals with the eye vector (or the cosine of the angle of the surfaces to the eye). They reveal something about the underlying shape of the objects in the image. Where the value is 1.0, the surface is facing directly toward the camera, and where it's 0, the surface is edge-on to the camera.

Layer_SHADOW 11
: Indicates where shadows are falling in the final image. It may also be thought of as an illumination map, showing what parts of the image are visible to the lights in the scene.

Layer_SHADING 12
: A picture of the diffuse shading and specular highlights applied to the objects in the scene. This is a component of the rendering calculations that depends solely on the angle of incidence of the lights on a surface. It doesn't include the effects of explicit shadow calculations.

Layer_DFSHADING 13
Layer_SPSHADING 14
: Like the `Layer_SHADING` buffer, but these store the amount of diffuse and specular shading (highlighting) separately, rather than adding them together.

Layer_TEXTUREU 15
Layer_TEXTUREV 16
Layer_TEXTUREW 17
: Texture coordinates.

Layer_NORMALX 18
Layer_NORMALY 19
Layer_NORMALZ 20
: Normal vector. This is the geometric normal of the object

surface visible in each pixel.

Layer_REFLECT 21
Reflection.

Layer_MOTIONX 22
Layer_MOTIONY 23
Support for 2D vector-based motion blur. These buffers contain the pixel distance moved by the item visible in each pixel. The amount of movement depends on the camera exposure time and includes the effects of the camera's motion.

bytesPerLayerPixel
Number of bytes per pixel, usually 1, 2 or 4.

compression
One of the following compression codes.

NoCompression 0
Data is uncompressed.

HorizontalRLE 1
VerticalRLE 3
Run-length encoding (RLE). The horizontal type is identical to the `byteRun1` RLE encoding used in ILBM and the output of the Macintosh `PackBits` function. The vertical type compresses along columns rather than rows. The compressor treats the data as a sequence of bytes, regardless of the data type of the layer's values.

HorizontalDelta 2
VerticalDelta 4
Delta encoding for animation. Only the parts of the image that differ from a previous frame are written.

The RLE and delta methods are described in more detail below.

blackPoint, whitePoint
The nominal minimum and maximum buffer levels. These define the

dynamic range of the data in the layer. Typical values for RGB layers are 0.0 and 1.0.

gamma

Linearity of the data. This and the black and white points are used primarily to encode RGB levels for different display devices. The default is 1.0.

Layer Data

The data for a layer is written in a `LAYR` chunk that immediately follows the layer's LayerHeader. The data is a rectangular array of values. The origin is the top left corner, and before compression, values are stored from left to right, and rows from top to bottom. No padding is added to the end of any row.

When the compression type is `NoCompression`, this is also how the layer is written in the file. The number of bytes in one row is

```
rowbytes = LayerHeader.bytesPerLayerPixel * FPHeader.width;
```

The number of rows is `FPHeader.Height`, and the total number of bytes of layer data (and the `LAYR` chunk size) is

```
layerbytes = rowbytes * FPHeader.height;
```

RLE Compression

The following psuedocode illustrates how RLE-compressed bytes are unpacked.

```
loop
   read the next source byte into n
   if n >= 0
      copy the next n + 1 bytes literally
   else if n < 0
      replicate the next byte -n + 1 times
until the row or column is full
```

The unpacker reads from the source (compressed data in a `LAYR` chunk) and writes to a destination (a memory buffer). For horizontal RLE, the destination pointer is incremented by 1 for each decoded byte, while for vertical RLE, the destination pointer is incremented by `rowbytes` bytes.

Each row (or column) is separately packed. In other words, runs never cross rows (or columns).

In the inverse routine (the packer), it's best to encode a 2 byte repeat run as a replicate run except when preceded and followed by a literal run, in which case it's best to merge the three into one literal run. Always encode 3 byte repeats as replicate runs.

Delta Compression

The delta compression method uses RLE, but it adds a mechanism for skipping bytes that haven't changed. This is used when storing animation frames. The skipped bytes retain the values stored there by a previous frame.

```
loop
   read the next source byte into nc
   if nc < 0
      skip ahead -nc columns
   else
      for i = 0 to nc
         read the next source byte into nr
         if nr < 0
            skip ahead -nr rows
         else
            unpack rle encoded span of size nr + 1
until the layer is full
```

Example Code

The `unpackRLE` function decodes RLE compressed data. **`psrc`** points to the source pointer. The function advances the source pointer as it decodes the compressed bytes. **`dst`** is the destination buffer where decoded bytes are written. **`size`** is the RLE span, or the number of destination bytes that should be produced. This is typically `rowbytes` for horizontal RLE and `FPHeader.Height` for vertical RLE. **`step`** is the number of bytes that the destination pointer should be moved after each decoded byte is written, typically 1 for horizontal and `rowbytes` for vertical. The function returns `TRUE` if it succeeds and `FALSE` otherwise.

```
int unpackRLE( char **psrc, char *dst, int size, int step )
{
   int c, n;
   char *src = *psrc;

   while ( size > 0 ) {
      n = *src++;
      if ( n >= 0 ) {
         ++n;
         size -= n;
         if ( size < 0 ) return FALSE;
         while ( n-- ) {
            *dst = *src++;
            dst += step;
         }
      }
      else {
         n = -n + 1;
         size -= n;
         if ( size < 0 ) return FALSE;
         c = *src++;
         while ( n-- ) {
            *dst = c;
            dst += step;
         }
      }
   }
   *psrc = src;
   return TRUE;
}
```

The `packRLE` function reads uncompressed bytes from the source buffer and writes encoded bytes to the destination. It returns the number of bytes written to the destination (the packed size of the source bytes).

```
#define DUMP    0
#define RUN     1
#define MINRUN  3
#define MAXRUN  128
#define MAXDUMP 128

int packRLE( char *src, char *dst, int size, int step )
{
   char c, lastc;
   int
      mode = DUMP,
      rstart = 0,
      putsize = 0,
      sp = 1,
      i;

   lastc = *src;
   size--;

   while ( size > 0 ) {
      c = *( src + sp * step );
      sp++;
      size--;

      switch ( mode ) {
         case DUMP:
            if ( sp > MAXDUMP ) {
               *dst++ = sp - 2;
               for ( i = 0; i < sp - 1; i++ )
                  *dst++ = *( src + i * step );
               putsize += sp;
               src += ( sp - 1 ) * step;
               sp = 1;
               rstart = 0;
               break;
            }

            if ( c == lastc ) {
               if (( sp - rstart ) >= MINRUN ) {
                  if ( rstart > 0 ) {
                     *dst++ = rstart - 1;
                     for ( i = 0; i < rstart; i++ )
                        *dst++ = *( src + i * step );
                     putsize += rstart + 1;
                  }
                  mode = RUN;
               }
               else if ( rstart == 0 ) mode = RUN;
            }
            else rstart = sp - 1;
            break;

         case RUN:
            if (( c != lastc ) || ( sp - rstart > MAXRUN )) {
               *dst++ = rstart + 2 - sp;
               *dst++ = lastc;
               putsize += 2;
               src += ( sp - 1 ) * step;
               sp = 1;
               rstart = 0;
               mode = DUMP;
            }
      }
      lastc = c;
   }

   switch ( mode ) {
      case DUMP:
         *dst++ = sp - 1;
         for ( i = 0; i < sp; i++ )
            *dst++ = *( src + i * step );
         putsize += sp + 1;
         break;

      case RUN:
         *dst++ = rstart + 1 - sp;
         *dst   = lastc;
         putsize += 2;
   }

   return putsize;
}
```

The `unpackDelta` function decodes delta-compressed data. After skipping to a part of the layer containing changes, it calls `unpackRLE`.

```
int unpackDelta( char *src, char *dst, int size, int vstep,
   int hstep )
{
   int n, nn;

   while ( size > 0 ) {
      n = *src++;
      --size;
      if ( n < 0 )
         dst += -n * vstep;
      else {
         for ( ; n >= 0; n-- ) {
            nn = *src++;
            --size;
            if ( nn < 0 )
               nn = -nn;
            else {
               ++nn;
               if ( !unpackRLE( &src, dst, nn, hstep ))
                  return FALSE;
            }
            dst += nn * hstep;
         }
      }
   }

   return TRUE;
}
```

"ILBM" IFF Interleaved Bitmap

Document Date: January 17, 1986
From: Jerry Morrison, Electronic Arts
Status of Standard: Released and in use

1. Introduction

"EA IFF 85" is Electronic Arts' standard for interchange format files. "ILBM" is a format for a 2 dimensional raster graphics image, specifically an InterLeaved bitplane BitMap image with color map. An ILBM is an IFF "data section" or "FORM type", which can be an IFF file or a part of one. (See the IFF reference.)

[Ed.: Editorial remarks (the text appearing between "[Ed.:" and "End ed.]" brackets) have been inserted at certain points in this document to update information about the way ILBM is currently implemented. *EW*. **End ed.]**

An ILBM is an archival representation designed for three uses. First, a standalone image that specifies exactly how to display itself (resolution, size, color map, etc.). Second, an image intended to be merged into a bigger picture which has its own depth, color map, and so on. And third, an empty image with a color map selection or "palette" for a paint program. ILBM is also intended as a building block for composite IFF FORMS like "animation sequence" and "structured graphics". Some uses of ILBM will be to preserve as much information as possible across disparate environments. Other uses will be to store data for a single program or highly cooperative programs while maintaining subtle details. So we're trying to accomplish a lot with this one format.

This memo is the IFF supplement for FORM ILBM. Section 2 defines the purpose and format of property chunks bitmap header BMHD, color map CMAP, hotspot GRAB, destination merge data DEST, sprite information SPRT, and Commodore Amiga viewport mode CAMG. Section 3 defines the standard data chunk BODY. These are the "standard" chunks. Section 4 defines the nonstandard color range data chunk CRNG. Additional specialized chunks like texture pattern can be added later. The ILBM syntax is summarized in Appendix A as a regular expression and in Appendix B as a box diagram. Appendix C explains the optional run encoding scheme. Appendix D names the committee responsible for this FORM ILBM standard.

Details of the raster layout are given in part 3, "Standard Data Chunk". Some elements are based on the Commodore Amiga hardware but generalized for use on other computers. An alternative to ILBM would be appropriate for computers with true color data in each pixel, though the wealth of available ILBM images makes import and export important. [**Ed.:** A standard for 24-bit RGB, 8-bit grayscale and 32-bit RGBA is described in Appendix E. **End ed.**]

Reference:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.

Amiga® is a registered trademark of Commodore-Amiga, Inc.
Electronic Arts™ is a trademark of Electronic Arts.
Macintosh™ is a trademark licensed to Apple Computer, Inc.
MacPaint™ is a trademark of Apple Computer, Inc.

2. Standard Properties

ILBM has several property chunks that act on the main data chunk. The required property BMHD and any optional properties must appear before any BODY chunk. (Since an ILBM has only one BODY chunk, any following properties would be superfluous.) Any of these properties may be shared over a LIST of FORMS ILBM by putting them in a PROP ILBM. (See the EA IFF 85 document.)

[**Ed.:** BMHD is the only essential property chunk for ILBMS used with LightWave. For broader support of different image types, you may also want to support CMAP and possibly CAMG. The other property chunks (and CAMG) have Amiga-specific semantics. It's also safe to assume that you'll never encounter PROPS. **End ed.**]

BMHD

The required property BMHD holds a BitMapHeader as defined in the following documentation. It describes the dimensions of the image, the encoding used, and other data necessary to understand the BODY chunk to follow.

```
typedef UBYTE Masking;  /* Choice of masking technique. */

#define mskNone   0
#define mskHasMask   1
#define mskHasTransparentColor   2
#define mskLasso  3

typedef UBYTE Compression;    /* Choice of compression algorithm
   applied to the rows of all source and mask planes.  "cmpByteRun1"
   is the byte run encoding described in Appendix C.  Do not compress
   across rows! */
#define cmpNone   0
#define cmpByteRun1  1

typedef struct {
   UWORD w, h;             /* raster width & height in pixels      */
   WORD  x, y;             /* pixel position for this image        */
   UBYTE nPlanes;          /* # source bitplanes                   */
   Masking masking;
   Compression compression;
   UBYTE pad1;             /* unused; ignore on read, write as 0   */
   UWORD transparentColor; /* transparent "color number" (sort of) */
   UBYTE xAspect, yAspect; /* pixel aspect, a ratio width : height */
   WORD  pageWidth, pageHeight;  /* source "page" size in pixels   */
} BitMapHeader;
```

Fields are filed in the order shown. The UBYTE fields are byte-packed (the C compiler must not add pad bytes to the structure).

The fields `w` and `h` indicate the size of the image rectangle in pixels. Each row of the image is stored in an integral number of 16 bit words. The number of words per row is `words=((w+15)/16)` or `Ceiling(w/16)`. The fields `x` and `y` indicate the desired position of this image within the destination picture. Some reader programs may ignore `x` and `y`. A safe default for writing an ILBM is `(x, y) = (0, 0)`.

The number of source bitplanes in the BODY chunk (see below) is stored in `nPlanes`. An ILBM with a CMAP but no BODY and `nPlanes = 0` is the recommended way to store a color map.

Note: Color numbers are color map index values formed by pixels in the destination bitmap, which may be deeper than `nPlanes` if a DEST chunk calls for merging the image into a deeper image.

The field `masking` indicates what kind of masking is to be used for this image. The value `mskNone` designates an opaque rectangular image. The value `mskHasMask` means that a mask plane is interleaved with the bitplanes in the `BODY` chunk (see below). [**Ed.:** These are usually the only masking options you'll encounter. **End ed.**] The value `mskHasTransparentColor` indicates that pixels in the source planes matching `transparentColor` are to be considered "transparent". (Actually, `transparentColor` isn't a "color number" since it's matched with numbers formed by the source bitmap rather than the possibly deeper destination bitmap. Note that having a transparent color implies ignoring one of the color registers. See `CMAP`, below.) The value `mskLasso` indicates the reader may construct a mask by lassoing the image as in MacPaint. To do this, put a 1 pixel border of `transparentColor` around the image rectangle. Then do a seed fill from this border. Filled pixels are to be transparent.

Issue: Include in an appendix an algorithm for converting a transparent color to a mask plane, and maybe a lasso algorithm.

A code indicating the kind of data compression used is stored in `compression`. Beware that using data compression makes your data unreadable by programs that don't implement the matching decompression algorithm. So we'll employ as few compression encodings as possible. The run encoding `byteRun1` is documented in Appendix C, below.

The field `pad1` is a pad byte reserved for future use. It must be set to 0 for consistency.

The `transparentColor` specifies which bit pattern means "transparent". This only applies if masking is `mskHasTransparentColor` or `mskLasso` (see above). Otherwise, `transparentColor` should be 0 (see above).

The pixel aspect ratio is stored as a ratio in the two fields `xAspect` and `yAspect`. This may be used by programs to compensate for different aspects or to help interpret the fields `w`, `h`, `x`, `y`, `pageWidth`, and `pageHeight`, which are in units of pixels. The fraction `xAspect/yAspect` represents a pixel's width/height. It's recommended that your programs store proper fractions in BitMapHeaders, but aspect ratios can always be correctly compared with the test

```
   xAspect * yDesiredAspect = yAspect * xDesiredAspect
```

Typical values for aspect ratio are width : height = 10 : 11 (Amiga 320 x 200 display) and 1 : 1 (Macintosh).

The size in pixels of the source "page" (any raster device) is stored in `pageWidth` and `pageHeight`, e.g. (320, 200) for a low resolution Amiga display. This information might be used to scale an image or to automatically set the display format to suit the image. Note that the image can be larger than the page.

CMAP

The optional (but encouraged) property `CMAP` stores color map data as triplets of red, green, and blue intensity values. The `n` color map entries ("color registers") are stored in the order 0 through `n-1`, totaling `3n` bytes. Thus `n` is the `ckSize/3`. Normally, `n` would equal $2^{nPlanes}$.

A `CMAP` chunk contains a ColorMap array as defined below. Note that these typedefs assume a C compiler that implements packed arrays of 3-byte elements.

```
   typedef struct {
      UBYTE red, green, blue;       /* color intensities 0..255 */
   } ColorRegister;                 /* size = 3 bytes */

   typedef ColorRegister ColorMap[n];  /* size = 3n bytes */
```

The color components `red`, `green`, and `blue` represent fractional intensity values expressed in 256ths in the range 0 through 255 (e.g., 24/256). White is (255, 255, 255--i.e., hex 0xFF, 0xFF, 0xFF) and black is (0, 0, 0). If your machine has less color resolution, use the higher order color bits when displaying by simply shifting the `CMAP` R, G, and B values to the right. When writing a `CMAP`, storage of less than 8 bits each of R, G, and B was previously accomplished by left justifying the significant bits within the stored bytes (i.e., a 4-bit per gun value of 0xF, 0xF, 0xF was stored as 0xF0, 0xF0, 0xF0). This provided correct color values when the `ILBM` was redisplayed on the same hardware since the zeros were shifted back out.

However, if color values stored by the above method were used as-is when redisplaying on hardware with more color resolution, diminished color could result. For example, a value of (0xF0, 0xF0, 0xF0) would be pure white on 4-bit-per-gun hardware (i.e., 0xF, 0xF, 0xF), but not quite white (0xF0, 0xF0, 0xF0) on 8-bit-per-gun hardware.

Therefore, when storing CMAP values, it is now suggested that you store full 8 bit values for R, G, and B which correctly scale your color values for eight bits. For 4-bit RGB values, this can be as simple as duplicating the 4-bit values in both the upper and lower parts of the bytes--i.e., store (0x1, 0x7, 0xF) as (0x11, 0x77, 0xFF). This will provide a more correct color rendition if the image is displayed on a device with 8 bits per gun.

When reading in a CMAP for 8-bit-per-gun display or manipulation, you may want to assume that any CMAP which has 0 values for the low bits of all guns for all registers was stored shifted rather than scaled, and provide your own scaling. Use defaults if the color map is absent or has fewer color registers than you need. Ignore any extra color registers.

The example type Color4 represents the format of a color register in working memory of an Amiga computer, which has 4 bit video DACs. (The ":4" tells the C compiler to pack the field into 4 bits.)

```
typedef struct {
   unsigned pad1 :4, red :4, green :4, blue :4;
} Color4;   /* Amiga RAM format. Not filed. */
```

Remember that every chunk must be padded to an even length, so a color map with an odd number of entries would be followed by a 0 byte, not included in the ckSize.

[**Ed:** Information on storing 8-bit grayscale, 24-bit color, and 32-bit color plus alpha ILBMs can be found in Appendix E. **End ed.**]

GRAB

The optional property GRAB locates a "handle" or "hotspot" of the image relative to its upper left corner, e.g. when used as a mouse cursor or a "paint brush". A GRAB chunk contains a Point2D.

```
typedef struct {
   WORD x, y;  /* relative coordinates (pixels) */
} Point2D;
```

DEST

The optional property "DEST" is a way to say how to scatter zero or more source bitplanes into a deeper destination image. Some readers may ignore DEST.

The contents of a DEST chunk is a DestMerge structure:

```
typedef struct {
   UBYTE depth;      /* # bitplanes in the original source  */
   UBYTE pad1;       /* unused; for consistency put 0 here  */
   UWORD planePick;  /* how to map source planes into destination */
   UWORD planeOnOff; /* default bitplane data for planePick */
   UWORD planeMask;  /* selects which bitplanes to store into */
} DestMerge;
```

The low order depth number of bits in planePick, planeOnOff, and planeMask correspond one-to-one with destination bitplanes. Bit 0 with bitplane 0, etc. (Any higher order bits should be ignored.) "1" bits in planePick mean "put the next source bitplane into this bitplane", so the number of "1" bits should equal nPlanes. "0" bits mean "put the corresponding bit from planeOnOff into this bitplane". Bits in planeMask gate writing to the destination bitplane: "1" bits mean "write to this bitplane" while "0" bits mean "leave this bitplane alone". The normal case (with no DEST property) is equivalent to planePick = planeMask = $2^{nPlanes}$ - 1.

Remember that color numbers are formed by pixels in the destination bitmap (depth planes deep) not in the source bitmap (nPlanes planes deep).

SPRT

The presence of an "SPRT" chunk indicates that this image is intended as a sprite. It's up to the reader program to actually make it a sprite, if even possible, and to use or overrule the sprite precedence data inside the SPRT chunk:

```
typedef UWORD SpritePrecedence;
/* relative precedence, 0 is the highest */
```

Precedence 0 is the highest, denoting a sprite that is foremost.

Creating a sprite may imply other setup. E.g. a 2 plane Amiga sprite would have transparentColor = 0. Color registers 1, 2, and 3 in the CMAP would be stored into the correct hardware color registers for the hardware sprite number used, while CMAP color register 0 would be ignored.

CAMG

A CAMG chunk is specifically for the Commodore Amiga computer. All Amiga-based reader and writer software should deal with CAMG. A CAMG chunk contains a single long word (length = 4) which specifies the Amiga display mode of the picture.

[**Ed.:** The Amiga has built-in support for interpreting the bits in a CAMG. Most of them are only meaningful on an Amiga, but two bits in the low word directly affect the interpretation of the data in the BODY chunk. Readers that attempt to support all ILBMS should test for these bits so that they can correctly translate the BODY. The bits are

```
#define CAMG_HAM 0x800   /* hold and modify */
#define CAMG_EHB 0x80    /* extra halfbrite */
```

HAM (hold-and-modify) mode allows the Amiga to display 12-bit and 18-bit RGB images using only 6 or 8 bits per pixel. HAM images store pixel values in the BODY chunk as codes that are divided into a mode in the high two bits and data in the other bits. The mode bits have the following interpretation.

00 - data bits are an index into the CMAP palette
01 - data bits contain the blue level
10 - data bits contain the red level
11 - data bits contain the green level

Unless a pixel is color-mapped (mode 00), only one of its three RGB levels is given in its code. The other two are assumed to be the same as those for the pixel to its left. If the pixel is the first one (the leftmost) in a scanline, the hold color is assumed to be (0, 0, 0). The number of data bits is 4 for standard HAM and 6 for HAM8, and the corresponding BitMapHeader `nPlanes` value will normally be 6 or 8.

It is possible for the mode to be a single bit. `nPlanes` will then be either 5 or 7. The single bit is the low bit, while the high bit is assumed to be 0, implying that only the blue level can be modified. For obvious reasons, this is rarely if ever encountered.

As described in the `CMAP` section, the data bits should be precision-extended when the levels are decoded to 24-bit. Regardless of the number of data bits, the maximum level should translate to 255 at 8 bits per RGB channel.

The iff SDK sample, which reads and writes IFF ILBM images, includes an `unHam` function that shows how the `BODY` data for a HAM image can be translated into more conventional 24-bit RGB.

Extra-Halfbrite is another Amiga variant, now quite rare. EHBs are 64-color pictures with 32-color palettes. Colors 32 to 63 are "half-bright" versions of colors 0 to 31, computed by bit shifting the RGB levels right by one. The easiest way to read EHB images is to extend the color table to include colors 32 to 63 and then interpret the `BODY` data as you would for any other indexed color image. **End ed.**]

---

3. Standard Data Chunk

Raster Layout

Raster scan proceeds left-to-right (increasing X) across scan lines, then top-to-bottom (increasing Y) down columns of scan lines. The coordinate system is in units of pixels, where (0,0) is the upper left corner.

The raster is typically organized as bitplanes in memory. The corresponding bits from each plane, taken together, make up an index into the color map which gives a color value for that pixel. The first bitplane, plane 0, is the low order bit of these color indexes.

A scan line is made of one "row" from each bitplane. A row is one plane's bits for one scan line, but padded out to a word (2 byte) boundary (not necessarily the first word boundary). Within each row, successive bytes are displayed in order and the most significant bit of each byte is displayed first.

[**Ed:** A conventional indexed color display stores the value of a pixel in a single byte (below, left). For a pixel at (*x*, *y*), the memory offset from the start of an image *w* pixels wide is just *wy* + *x* (ignoring any scanline padding), and the value stored there is an index into a table of RGB color records. In an `ILBM`, the bits of a given pixel aren't contiguous in memory. They are instead stored in separate *bitplanes*, each of which contains a single bit from a given pixel (below, right).

```
                                  plane 0:  10000011b
                                  plane 1:  00110010b
                                  plane 2:  10001001b
        pixel:  00111010b         plane 3:  11010100b
                                  plane 4:  01010111b
                                  plane 5:  10011010b
```

To retrieve a pixel value (naïvely), you must read bytes at different addresses (six of them in the above example), mask off all but one bit from each of them, and string the bits together. For the pixel at (*x*, *y*), the byte offset into each bitplane is (*wy* + *x*) / 8, and the bit is 7 - (*x* mod 8). Bitplane *n* contains the *n*-th bit of the pixel value. **End ed.**]

A "mask" is an optional "plane" of data the same size (w, h) as a bitplane. It tells how to "cut out" part of the image when painting it onto another image."One" bits in the mask mean "copy the corresponding pixel to the destination" while "zero" mask bits mean "leave this destination pixel alone". In other words, "zero" bits designate transparent pixels.

The rows of the different bitplanes and mask are interleaved in the file (see below). This localizes all the information pertinent to each scan line. It makes it much easier to transform the data while reading it to adjust the image size or depth. It also makes it possible to scroll a big image by swapping rows directly from the file without random-accessing to all the bitplanes.

BODY

The source raster is stored in a `BODY` chunk. This one chunk holds all bitplanes and the optional mask, interleaved by row.

The BitMapHeader, in a `BMHD` property chunk, specifies the raster's dimensions `w`, `h`, and `nPlanes`. It also holds the `masking` field which indicates if there is a mask plane and the `compression` field which indicates the compression algorithm used. This information is needed to interpret the `BODY` chunk, so the `BMHD` chunk must appear first. While reading an `ILBM`'s `BODY`, a program may convert the image to another size by filling (with `transparentColor`) or clipping.

The `BODY`'s content is a concatenation of scan lines. Each scan line is a concatenation of one row of data from each plane in order 0 through `nPlanes-1` followed by one row from the mask (if `masking = hasMask`). If the BitMapHeader field `compression` is `cmpNone`, all `h` rows are exactly `(w+15)/16` words wide. Otherwise, every row is compressed according to the specified algorithm and their stored widths depend on the data compression.

Reader programs that require fewer bitplanes than appear in a particular `ILBM` file can combine planes or drop the high-order (later) planes. Similarly, they may add bitplanes and/or discard the mask plane.

Do *not* compress across rows and don't forget to compress the mask just like the bitplanes. Remember to pad any `BODY` chunk that contains an odd number of bytes and skip the pad when reading.

---

4. Nonstandard Data Chunks

The following data chunks were defined after various programs began using `FORM ILBM` so they are "nonstandard" chunks.

CRNG

A `CRNG` chunk contains "color register range" information. It's used by Electronic Arts' Deluxe Paint program to identify a contiguous range of color registers for a "shade range" and color cycling. There can be zero or more `CRNG` chunks in an `ILBM`, but all should appear before the `BODY` chunk. Deluxe Paint normally writes 4 `CRNG` chunks in an `ILBM` when the user asks it to "Save Picture".

```
typedef struct {
   WORD  pad1;       /* reserved for future use; store 0 here    */
   WORD  rate;       /* color cycle rate                         */
   WORD  flags;      /* see below                                */
   UBYTE low, high;  /* lower and upper color registers selected */
} CRange;
```

The bits of the flags word are interpreted as follows: if the low bit is set then the cycle is "active", and if this bit is clear it is not active. Normally, color cycling is done so that colors move to the next higher position in the cycle, with the color in the high slot moving around to the low slot. If the second bit of the `flags` word is set, the cycle moves in the opposite direction. As usual, the other bits of the `flags` word are reserved for future expansion. Here are the masks to test these bits:

```
   #define RNG_ACTIVE 1
   #define RNG_REVERSE 2
```

The fields `low` and `high` indicate the range of color registers (color numbers) selected by this CRange.

The field `rate` determines the speed at which the colors will step when color cycling is on. The units are such that a rate of 60 steps per second is represented as 2^{14} = 16384. Slower rates can be obtained by linear scaling: for 30 steps/second, rate = 8192; for 1 step/second, rate = 16384 / 60, or 273.

> *Warning!* One popular paint package always sets the `RNG_ACTIVE` bit, but uses a rate of 36 (decimal) to indicate cycling is not active.

CCRT

Commodore's Graphicraft program uses a similar chunk `CCRT` (for Color Cyling Range and Timing). This chunk contains a CycleInfo structure.

```
   typedef struct {
      WORD  direction;     /*  0 = don't cycle, 1 = cycle forwards,  */
                           /* -1 = cycle backwards                   */
      UBYTE start, end;    /* lower, upper color registers selected  */
      LONG  seconds;       /* # seconds between changing colors plus */
      LONG  microseconds;  /* # microseconds between changing colors */
      WORD  pad;           /* reserved for future use; store 0 here  */
   } CycleInfo;
```

This is very similar to a `CRNG` chunk. A program would probably only use one of these two methods of expressing color cycle data. New programs should use `CRNG`. You could write out both if you want to communicate this information to both Deluxe Paint and Graphicraft.

Appendix A. ILBM Regular Expression

Here's a regular expression summary of the FORM ILBM syntax. This could be an IFF file or a part of one.

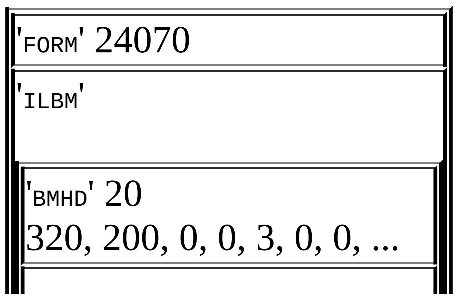
```
ILBM ::= "FORM" #{   "ILBM" BMHD [CMAP] [GRAB] [DEST] [SPRT] [CAMG]
                     CRNG* CCRT* [BODY]   }

BMHD ::= "BMHD" #{   BitMapHeader   }
CMAP ::= "CMAP" #{   (red green blue)* } [0]
GRAB ::= "GRAB" #{   Point2D  }
DEST ::= "DEST" #{   DestMerge   }
SPRT ::= "SPRT" #{   SpritePrecendence }
CAMG ::= "CAMG" #{   LONG  }

CRNG ::= "CRNG" #{   CRange   }
CCRT ::= "CCRT" #{   CycleInfo   }
BODY ::= "BODY" #{   UBYTE*   } [0]
```

The token "#" represents a ckSize LONG count of the following {braced} data bytes. E.g. a BMHD's "#" should equal sizeof(BitMapHeader). Literal strings are shown in "quotes", [square bracket items] are optional, and "*" means 0 or more repetitions. A sometimes-needed pad byte is shown as "[0]".

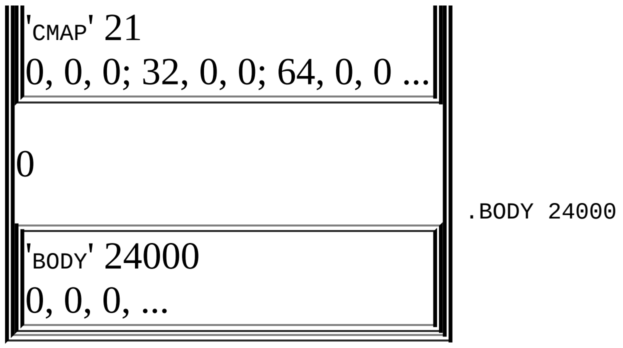
The property chunks (BMHD, CMAP, GRAB, DEST, SPRT, and CAMG) and any CRNG and CCRT data chunks may actually be in any order but all must appear before the BODY chunk since ILBM readers usually stop as soon as they read the BODY. If any of the 6 property chunks are missing, default values are inherited from any shared properties (if the ILBM appears inside an IFF LIST with PROPs) or from the reader program's defaults. If any property appears more than once, the last occurrence before the BODY is the one that counts since that's the one that modifies the BODY.

Appendix B. ILBM Box Diagram

Here's a box diagram for a simple example: an uncompressed image 320 x 200 pixels x 3 bitplanes. The text to the right of the diagram shows the outline that would be printed by the IFFCheck utility program for this particular file.



```
FORM 24070 ILBM

 .BMHD 20

 .CMAP 21
```



The "0" after the CMAP chunk is a pad byte.

Appendix C. ByteRun1 Run Encoding

The run encoding scheme `byteRun1` is best described by psuedo code for the decoder Unpacker (called UnPackBits in the Macintosh toolbox):

```
UnPacker:
   LOOP until produced the desired number of bytes
      Read the next source byte into n
      SELECT n FROM
         [0..127] => copy the next n+1 bytes literally
         [-1..-127]  => replicate the next byte -n+1 times
         -128  => noop
         ENDCASE;
      ENDLOOP;
```

In the inverse routine Packer, it's best to encode a 2 byte repeat run as a replicate run except when preceded and followed by a literal run, in which case it's best to merge the three into one literal run. Always encode 3 byte repeats as replicate runs.

Remember that each row of each scan line of a raster is separately packed.

[**Ed:** Some versions of Adobe Photoshop incorrectly use the `n=128` no-op as a repeat code, which breaks strictly conforming readers. To read Photoshop ILBMs, allow the use of `n=128` as a repeat. This is pretty safe, since no known program writes real no-ops into their ILBMs. The reason `n=128` is a no-op is historical: the Mac Packbits buffer was only 128 bytes, and a repeat code of 128 generates 129 bytes. **End ed.**]

Appendix D. Standards Committee

The following people contributed to the design of this FORM ILBM standard:

Bob "Kodiak" Burns, Commodore-Amiga
R. J. Mical, Commodore-Amiga
Jerry Morrison, Electronic Arts
Greg Riker, Electronic Arts
Steve Shaw, Electronic Arts
Dan Silva, Electronic Arts
Barry Walsh, Commodore-Amiga

Appendix E. IFF Hints

Hints on ILBM files from Jerry Morrison, Oct 1988. How to avoid some pitfalls when reading ILBM files:

- Don't ignore the BitMapHeader `masking` field. A bitmap with a mask (such as a partially transparent DPaint brush or a DPaint picture with a stencil) will read as garbage if you don't de-interleave the mask.
- Don't assume all images are compressed. Narrow images aren't usually run-compressed since that would actually make them longer.
- Don't assume a particular image size. You may encounter overscan pictures and PAL pictures.

Different hardware display devices have different color resolutions:

Device	***R:G:B bits***	***maxColor***
Mac SE	1	1
IBM EGA	2:2:2	3
Atari ST	3:3:3	7
Amiga	4:4:4	15
CD-I	5:5:5	31
IBM VGA	6:6:6	63
Mac II	8:8:8	255

An `ILBM` `CMAP` defines 8 bits of Red, Green and Blue (i.e., 8:8:8 bits of R:G:B). When displaying on hardware which has less color resolution, just take the high order bits. For example, to convert `ILBM`'s 8-bit Red to the Amiga's 4-bit Red, right shift the data by 4 bits (`R4 := R8 >> 4`).

To convert hardware colors to `ILBM` colors, the `ILBM` specification says just set the high bits (`R8 := R4 << 4`). But you can transmit higher contrast to foreign display devices by scaling the data [0..maxColor] to the full range [0..255]. In other words, R8 := (Rn x 255) / maxColor. (Example #1: EGA color 1:2:3 scales to 85:170:255. Example #2: Amiga 15:7:0 scales to 255:119:0). This makes a big difference where macColor is less than 15. In the extreme case, Mac SE white (1) should be converted to ILBM white (255), not to ILBM gray (128).

CGA and EGA subtleties

IBM EGA colors in 350 scan line mode are 2:2:2 bits of R:G:B, stored in memory as xxR'G'B'RGB. That's 3 low-order bits followed by 3 high-order bits.

IBM CGA colors are 4 bits stored in a byte as xxxxIRGB. (EGA colors in 200 scan line modes are the same as CGA colors, but stored in memory as xxxIxRGB.) That's 3 high-order bits (one for each of R, G, and B) plus one low-order "Intensity" bit for all 3 components R, G, and B. Exception: IBM monitors show IRGB = 0110 as brown, which is really the EGA color R:G:B = 2:1:0, not dark yellow 2:2:0.

24-bit ILBMs

When storing deep images as ILBMS (e.g., images with 8 bits each of R, G, and B), the bits for each pixel represent an absolute RGB value for that pixel rather than an index into a limited color map. The order for saving the bits is critical since a deep ILBM would not contain the usual CMAP of RGB values (such a CMAP would be too large and redundant).

To interpret these "deep" ILBMS, it is necessary to have a standard order in which the bits of the R, G, and B values will be stored. A number of different orderings have already been used in deep ILBMS and a default has been chosen from them.

The following bit ordering has been chosen as the default bit ordering for deep ILBMS.

```
Default standard deep ILBM bit ordering:
saved first ------------------------------------------------> saved last
R0 R1 R2 R3 R4 R5 R6 R7 G0 G1 G2 G3 G4 G5 G6 G7 B0 B1 B2 B3 B4 B5 B6 B7
```

[**Ed.:** Recall from Section 3 that the bits representing the value at a given pixel are divided into separate bitplanes. A 24-bit RGB image uses 24 bitplanes. Also recall that images are stored in the BODY one complete scanline at a time, so one row from each of the 24 bitplanes is written before moving to the next scanline. For each scanline, the red bitplane rows are stored first, followed by green and blue. The first plane holds the least significant bit of the red value for each pixel, and the last holds the most significant bit of the blue value.

8-bit Grayscale

The original standard doesn't prescribe the form of an 8-bit grayscale image, but we can infer one from the convention for 24-bit color. Grayscale images also lack a CMAP, and their bitplanes are saved in least to most significant bit order.

```
Grayscale ILBM bit ordering:
saved first -----> last
I0 I1 I2 I3 I4 I5 I6 I7
```

Some programs fail to recognize 8-bit ILBMs with no color table. For maximum portability, ILBM writers can include a CMAP containing 256 entries, with the RGB levels ranging from (0, 0, 0) for the first entry to (255, 255, 255) for the last. Strictly speaking, this creates an indexed color image in which all of the colors happen to be shades of gray, but this distinction may not make any difference in practice.

32-bit RGB plus Alpha

A more recent (and much less widely supported) extension of the standard is the 32-bit RGBA. This adds an 8-bit grayscale alpha image to the red, green and blue stored in 24-bit ILBMs. The alpha bitplanes are stored after the R, G and B planes for each scanline.

```
32-bit RGBA ILBM bit ordering:
saved first ------------------------------------> last
R0 ... R7 G0 ... G7 B0 ... B7 A0 A1 A2 A3 A4 A5 A6 A7
```

End ed.]

Object Files

November 9, 2001

This document describes the LWO2 file format for 3D objects used by LightWave. The LWO2 format is new for LightWave 6.0. Also see the Object File Examples supplement.

Introduction

The data in LightWave 3D object files comprise the points, polygons and surfaces that describe the geometry and appearance of an object. "Polygons" here means any of several geometric elements (faces, curves or patches, for example) defined by an ordered list of points, and "surfaces" refers to the collection of attributes, sometimes called materials, that define the visual surface properties of polygons.

Object files can contain multiple layers, or parts, and each part can be a single connected mesh or several disjoint meshes. They may also contain one or more surface definitions with no points or polygons at all. Surface definitions can include references to other files (images, for example), plug-ins, and envelopes containing parameter values that vary over time.

This document outlines the object file format and provides a detailed reference for each of the components. The component descriptions include both a regular expression defining the syntax and a discussion of the contents. See also the Examples supplement, a more conversational introduction to the format that includes annotated listings of file contents as well as several sample files.

Data Types

The atomic, or lowest-level, types used in object files are listed below. All of these are written in a byte order variously called big-endian, Motorola, or network order, with the most significant byte written first. The shorthand names (**I2**, **F4**, etc.) will be used throughout this document.

ID Tag

ID4

An ID tag is a sequence of 4 bytes containing 7-bit ASCII values, usually upper-case printable characters. These tags are used to identify the data that follows. `FORM`, `SURF`, `POLS`, and `LWO2` are all examples of ID tags. ID tags can be interpreted as unsigned integers for comparison purposes.

Signed Integer

I1, I2, I4

Unsigned Integer

U1, U2, U4

Integers can be signed or unsigned and 1, 2 or 4 bytes in length. Signed integers are two's complement.

Float

F4

4-byte IEEE floating-point values.

String

S0

Names or other character strings are written as a series of ASCII character values followed by a zero (or null) byte. If the length of the string including the null terminating byte is odd, an extra null is added so that the data that follows will begin on an even byte boundary.

Several useful composite datatypes are built from these fundamental types.

Variable-length Index

`VX ::= index[U2] | (index + 0xFF000000)[U4]`

This is an index into an array of items (points or polygons), or a collection of items each uniquely identified by an integer (clips or envelopes). A VX is written as a variable length 2- or 4-byte element. If the index value is less than 65,280 (0xFF00), then the index is written as an unsigned two-byte integer. Otherwise the index is written as an unsigned four byte integer with bits 24-31 set. When reading an index, if the first byte encountered is 255 (0xFF), then the four-byte form is being used and the first byte should be discarded or masked out.

Color

`COL12 ::= red[F4], green[F4], blue[F4]`

A color is written as a triple of floats representing the levels of red, green and blue. The nominal level range is [0.0, 1.0], but values outside this range are also possible.

Coordinate

`VEC12 ::= X[F4], Y[F4], Z[F4]`

3D coordinates are written as an XYZ vector in floating point format. The values are distances along the X, Y, and Z axes.

Percentage

`FP4 ::= fraction[F4]`

Percentages are written as floats, with 1.0 representing 100%.

Angle

`ANG4 ::= radians[F4]`

Angles are specified as floating point values in radians.

Filename

`FNAM0 ::= name[S0]`

Filenames are written as strings in a platform-neutral format. For absolute (fully qualified) paths, the first node represents a disk or similar storage device, and its name is separated from the rest of the path by a colon. Other nodes in the path are separated by forward slashes. `disk:path/file` is an absolute path, and `path/subpath/file` is a relative path.

Chunks

The object file format is derived from the metaformat for binary files described in "EA IFF 85 Standard for Interchange Format Files." The basic structural element in an IFF file is the *chunk*. A chunk consists of an ID tag, a size, and *size* bytes of data. If the size is odd, the chunk is followed by a 0 pad byte, so that the next chunk begins on an even byte boundary. (The pad byte isn't counted in the size.)

CHUNK ::= tag[ID4], length[U4], data[...], pad[U1] ?

Within some chunks, object files use *subchunks*, which are just like chunks except that the size is a 2-byte integer.

SUB-CHUNK ::= tag[ID4], length[U2], data[...], pad[U1] ?

In this document, chunks will be written as a chunk ID followed by a data description inside curly brackets: **`ID-tag { data }`**. Given this notation, we can say formally that an object file is a `FORM` chunk of type `LWO2`.

file ::= FORM { 'LWO2'[ID4], data[CHUNK] * }

Informally, object files start with the four bytes "`FORM`" followed by a four-byte integer giving the length of the file (minus 8) and the four byte ID "`LWO2`". The remainder of the data is a collection of chunks, some of which will contain subchunks.

To be read, IFF files must be *parsed*. The order in which chunks can occur in a file isn't fixed. Some chunks, however, contain data that depends on the contents of other chunks, and this fixes a *relative* order for the chunks involved. Chunks and subchunks also depend on context for their meaning. The `CHAN` subchunk in an envelope chunk isn't the same thing as the `CHAN` subchunk in a surface block. And you may encounter chunks that aren't defined here, which you should be prepared to skip gracefully if you don't understand them. You can do this by using the chunk size to seek to the next chunk.

The following is a list of the defined chunks that can be found in an object file. Full descriptions of the contents of `ENVL`, `CLIP` and `SURF` chunks are deferred to sections that follow the chunk list and comprise the remainder of this document.

Layer

```
LAYR { number[U2], flags[U2], pivot[VEC12], name[S0], parent[U2] ? }
```

Signals the start of a new layer. All the data chunks which follow will be included in this layer until another layer chunk is encountered. If data is encountered before a layer chunk, it goes into an arbitrary layer. If the least significant bit of `flags` is set, the layer is hidden. The parent index indicates the default parent for this layer and can be -1 or missing to indicate no parent.

Point List

```
PNTS { point-location[VEC12] * }
```

Lists (*x*, *y*, *z*) coordinate triples for a set of points. The number of points in the chunk is just the chunk size divided by 12. The `PNTS` chunk must precede the `POLS`, `VMAP` and `VMAD` chunks that refer to it. These chunks list points using a 0-based index into `PNTS`.

The LightWave coordinate system is left-handed, with +X to the right or east, +Y upward, and +Z forward or north. Object files don't contain explicit units, but by convention the unit is meters. Coordinates in `PNTS` are relative to the pivot point of the layer.

Vertex Mapping

```
VMAP { type[ID4], dimension[U2], name[S0],
( vert[VX], value[F4] # dimension )* }
```

Associates a set of floating-point vectors with a set of points. `VMAP`s begin with a type, a dimension (vector length) and a name. These are followed by a list of vertex/vector pairs. The vertex is given as an index into the most recent `PNTS` chunk, in VX format. The vector contains `dimension` floating-point values. There can be any number of these chunks, but they should all have different types or names.

Some common type codes are

PICK
: Selection set. This is a VMAP of dimension 0 that marks points for quick selection by name during modeling. It has no effect on the geometry of the object.

WGHT
: Weight maps have a dimension of 1 and are generally used to alter the influence of deformers such as bones. Weights can be positive or negative, and the default weight for unmapped vertices is 0.0.

MNVW
: Subpatch weight maps affect the shape of geometry created by subdivision patching.

TXUV
: UV texture maps have a dimension of 2.

`RGB`, `RGBA`
: Color maps, with a dimension of 3 or 4.

MORF
: These contain vertex displacement deltas.

SPOT
: These contain absolute vertex displacements (alternative vertex positions).

Other widely used map types will almost certainly appear in the future.

Polygon List

```
POLS { type[ID4], ( numvert+flags[U2], vert[VX] # numvert )* }
```

A list of polygons for the current layer. Possible polygon types include:

FACE
: "Regular" polygons, the most common.

CURV
: Catmull-Rom splines. These are used during modeling and are currently ignored by the renderer.

PTCH
: Subdivision patches. The POLS chunk contains the definition

of the control cage polygons, and the patch is created by subdividing these polygons. The renderable geometry that results from subdivision is determined interactively by the user through settings within LightWave. The subdivision method is undocumented.

MBAL

Metaballs. These are single-point polygons. The points are associated with a `VMAP` of type `MBAL` that contains the radius of influence of each metaball. The renderable polygonal surface constructed from a set of metaballs is inferred as an isosurface on a scalar field derived from the sum of the influences of all of the metaball points.

BONE

Line segments representing the object's skeleton. These are converted to bones for deformation during rendering.

Each polygon is defined by a vertex count followed by a list of indexes into the most recent `PNTS` chunk. The maximum number of vertices is 1023. The 6 high-order bits of the vertex count are flag bits with different meanings for each polygon type. (Currently only two flags are defined: the low two bits are continuity control point toggles for `CURV` polygons. Other flags may be defined in the future.) When reading `POLS`, remember to mask out the `flags` to obtain `numverts`.

When writing `POLS`, the vertex list for each polygon should begin at a convex vertex and proceed clockwise as seen from the visible side of the polygon. LightWave polygons are single-sided (although double-sidedness is a possible surface property), and the normal is defined as the cross product of the first and last edges.

Tag Strings

`TAGS { tag-string[S0] * }`

Lists the tag strings that can be associated with polygons by the `PTAG` chunk.

Polygon Tag Mapping

`PTAG { type[ID4], ( poly[VX], tag[U2] )* }`

Associates tags of a given type with polygons in the most recent `POLS` chunk. The most common polygon tag types are

SURF
: The surface assigned to the polygon. The actual surface attributes are found by matching the name in the TAGS chunk with the name in a SURF chunk.

PART
: The part the polygon belongs to. Parts are named groups of polygons analogous to point selection sets (but a polygon can belong to only one part).

SMGP
: The smoothing group the polygon belongs to. Shading is only interpolated within a smoothing group, not across groups.

The polygon is identified by an index into the previous POLS chunk, and the tag is given by an index into the previous TAGS chunk. Not all polygons will have a value for every tag type. The behavior for polygons lacking a given tag depends on the type.

Discontinuous Vertex Mapping

```
VMAD { type[ID4], dimension[U2], name[S0],
( vert[VX], poly[VX], value[F4] # dimension )* }
```

(Introduced with LightWave 6.5.) Associates a set of floating-point vectors with the vertices of specific polygons. VMADs are similar to VMAPs, but they assign vectors to polygon vertices rather than points. For a given mapping, a VMAP always assigns only one vector to a point, while a VMAD can assign as many vectors to a point as there are polygons sharing the point.

The motivation for VMADs is the problem of seams in UV texture mapping. If a UV map is topologically equivalent to a cylinder or a sphere, a seam is formed where the opposite edges of the map meet. Interpolation of UV coordinates across this discontinuity is aesthetically and mathematically incorrect. The VMAD substitutes an equivalent mapping that interpolates correctly. It only needs to do this for polygons in which the seam lies.

VMAD chunks are paired with VMAPs of the same name, if they exist. The vector values in the VMAD will then replace those in the corresponding VMAP, but only for calculations involving the specified polygons. When the same points are used for calculations on polygons not specified in the VMAD, the VMAP values are used.

VMADs need not be associated with a VMAP. They can also be used simply to define a (discontinuous) per-polygon mapping. But not all mapping types are valid for VMADs, since for some types it makes no sense for points to have more than one map value. TXUV, RGB, RGBA and WGHT types are supported for VMADs, for example, while MORF and SPOT are not. VMADs of unsupported types are preserved but never evaluated.

Envelope Definition

```
ENVL { index[VX], attributes[SUB-CHUNK] * }
```

An array of keys. Each ENVL chunk defines the value of a single parameter channel as a function of time. The index is used to identify this envelope uniquely and can have any non-zero value less than 0x1000000. Following the index is a collection of subchunks that describe the envelope. These are documented below, in the Envelope Subchunks section.

Image or Image Sequence

```
CLIP { index[U4], attributes[SUB-CHUNK] * }
```

Describes an image or a sequence of images. Surface definitions specify images by referring to CLIP chunks. The term "clip" is used to describe these because they can be numbered sequences or animations as well as stills. The index identifies this clip uniquely and may be any non-zero value less than 0x1000000. The filename and any image processing modifiers follow as a variable list of subchunks, which are documented below in the Clip Subchunks section.

Surface Definition

```
SURF { name[S0], source[S0], attributes[SUB-CHUNK] * }
```

Describes the shading attributes of a surface. The name uniquely identifies the surface. This is the string that's stored in TAGS and referenced by tag index in PTAG. If the source name is non-null, then this surface is derived from, or composed with, the source surface.

The base attributes of the source surface can be overridden by this surface, and texture blocks can be added to the source surface. The material attributes follow as a variable list of subchunks documented below in the Surface Subchunks section.

Bounding Box

`BBOX { min[VEC12], max[VEC12] }`

Store the bounding box for the vertex data in a layer. Optional. The `min` and `max` vectors are the lower and upper corners of the bounding box.

Description Line

`DESC { description-line[S0] }`

Store an object description. Optional. This should be a simple line of upper and lowercase characters, punctuation and spaces which describes the contents of the object file. There should be no control characters in this text string and it should generally be kept short.

Commentary Text

`TEXT { comment[S0] }`

Store comments about the object. Optional. The text is just like the `DESC` chunk, but it can be about any subject, it may contain newline characters and it does not need to be particularly short.

Thumbnail Icon Image

`ICON { encoding[U2], width[U2], data[U1] * }`

An iconic or thumbnail image for the object which can be used when viewing the file in a browser. Currently the only suported `encoding` is 0, meaning uncompressed RGB byte triples. The `width` is the number of pixels in each row of the image, and the height (number of rows) is `(chunkSize - 4)/width`. This chunk is optional.

Envelope Subchunks

The `ENVL` chunk contains a series of subchunks describing the keyframes, intervals and global attributes of a single envelope. Note that the `PRE`, `KEY` and `TCB` IDs each include a trailing space when written in the file.

Envelope Type

`TYPE { user-format[U1], type[U1] }`

The type subchunk records the format in which the envelope is displayed to the user and a type code that identifies the components of certain predefined envelope triples. The user format has no effect on the actual values, only the way they're presented in LightWave's interface.

`02`	Float
`03`	Distance
`04`	Percent
`05`	Angle

The predefined envelope types include the following.

`01, 02, 03`	Position: X, Y, Z
`04, 05, 06`	Rotation: Heading, Pitch, Bank
`07, 08, 09`	Scale: X, Y, Z
`0A, 0B, 0C`	Color: R, G, B
`0D, 0E, 0F`	Falloff: X, Y, Z

Pre-Behavior

`PRE { type[U2] }`

The pre-behavior for an envelope defines the signal value for times before the first key. The type code selects one of several predefined behaviors.

0 - Reset
: Sets the value to 0.0.

1 - Constant
: Sets the value to the value at the nearest key.

2 - Repeat
: Repeats the interval between the first and last keys (the primary interval).

3 - Oscillate
: Like Repeat, but alternating copies of the primary interval are time-reversed.

4 - Offset Repeat

Like Repeat, but offset by the difference between the values of the first and last keys.

5 - Linear

Linearly extrapolates the value based on the tangent at the nearest key.

Post-Behavior

```
POST { type[U2] }
```

The post-behavior determines the signal value for times after the last key. The type codes are the same as for pre-behaviors.

Keyframe Time and Value

```
KEY { time[F4], value[F4] }
```

The value of the envelope at the specified time in seconds. The signal value between keyframes is interpolated. The time of a keyframe isn't restricted to integer frames.

Interval Interpolation

```
SPAN { type[ID4], parameters[F4] * }
```

Defines the interpolation between the most recent `KEY` chunk and the `KEY` immediately before it in time. The `type` identifies the interpolation algorithm and can be `STEP`, `LINE`, `TCB` (Kochanek-Bartels), `HERM` (Hermite), `BEZI` (1D Bezier) or `BEZ2` (2D Bezier). Different parameters are stored for each of these.

Plug-in Channel Modifiers

```
CHAN { server-name[S0], flags[U2], data[...] }
```

Channel modifiers can be associated with an envelope. Each channel chunk contains the name of the plug-in and some flag bits. Only the first flag bit is defined; if set, the plug-in is disabled. The data that follows this, if any, is owned by the plug-in.

Channel Name

```
NAME { channel-name[S0] }
```

An optional name for the envelope. LightWave itself ignores the names of surface envelopes, but plug-ins can browse the envelope database by name.

The source code in the sample/envelope directory of the LightWave plug-in SDK demonstrates interpolation and extrapolation of envelopes and shows how the contents of the SPAN subchunks define TCB, Bezier and Hermite curves.

Clip Subchunks

The CLIP chunk contains a series of subchunks describing a single, possibly time-varying image. The first subchunk has to be one of the source chunks: STIL, ISEQ, ANIM, XREF or STCC.

Still Image

```
STIL { name[FNAM0] }
```

The source is a single still image referenced by a filename in neutral path format.

Image Sequence

```
ISEQ { num-digits[U1], flags[U1], offset[I2], reserved[U2], start[I2],
end[I2], prefix[FNAM0], suffix[S0] }
```

The source is a numbered sequence of still image files. Each filename contains a fixed number of decimal digits that specify a frame number, along with a prefix (the part before the frame number, which includes the path) and a suffix (the part after the number, typically a PC-style extension that identifies the file format). The prefix and suffix are the same for all files in the sequence.

The flags include bits for looping and interlace. The offset is added to the current frame number to obtain the digits of the filename for the current frame. The start and end values define the range of frames in the sequence.

Plug-in Animation

```
ANIM { filename[FNAM0], server-name[S0], flags[U2], data[...] }
```

This chunk indicates that the source imagery comes from a plug-in animation loader. The loader is defined by the server name, a flags value, and the server's data.

Reference (Clone)

```
XREF { index[U4], string[S0] }
```

The source is a copy, or instance, of another clip, given by the index. The string is a unique name for this instance of the clip.

Color-cycling Still
`STCC { lo[I2], hi[I2], name[FNAM0] }`

A still image with color-cycling is a source defined by a neutral-format name and cycling parameters. `lo` and `hi` are indexes into the image's color table. Within this range, the color table entries are shifted over time to cycle the colors in the image. If `lo` is less than `hi`, the colors cycle forward, and if `hi` is less than `lo`, they go backwards.

Except for the `TIME` subchunk, the subchunks after the source subchunk modify the source image and are applied as filters layered on top of the source image.

Time
`TIME { start-time[FP4], duration[FP4], frame-rate[FP4] }`

Defines source times for an animated clip.

Contrast
`CONT { contrast-delta[FP4], envelope[VX] }`

RGB levels are altered in proportion to their distance from 0.5. Positive deltas move the levels toward one of the extremes (0.0 or 1.0), while negative deltas move them toward 0.5. The default is 0.

Brightness
`BRIT { brightness-delta[FP4], envelope[VX] }`

The delta is added to the RGB levels. The default is 0.

Saturation
`SATR { saturation-delta[FP4], envelope[VX] }`

The saturation of an RGB color is defined as `(max - min)/max`, where `max` and `min` are the maximum and minimum of the three RGB levels. This is a measure of the intensity or purity of a color. Positive deltas turn up the saturation by increasing the `max` component and decreasing the `min` one, and negative deltas have the opposite effect. The default is 0.

Hue

```
HUE { hue-rotation[FP4], envelope[VX] }
```

The hue of an RGB color is an angle defined as

r is max: 1/3 (g - b)/(r - min)
g is max: 1/3 (b - r)/(g - min) + 1/3
b is max: 1/3 (r - g)/(b - min) + 2/3

with values shifted into the [0, 1] interval when necessary. The levels between 0 and 1 correspond to angles between 0 and 360 degrees. The hue delta rotates the hue. The default is 0.

Gamma Correction

```
GAMM { gamma[F4], envelope[VX] }
```

Gamma correction alters the distribution of light and dark in an image by raising the RGB levels to a small power. By convention, the gamma is stored as the inverse of this power. A gamma of 0.0 forces all RGB levels to 0.0. The default is 1.0.

Negative

```
NEGA { enable[U2] }
```

If non-zero, the RGB values are inverted, (1.0 - r, 1.0 - g, 1.0 - b), to form a negative of the image.

Plug-in Image Filters

```
IFLT { server-name[S0], flags[U2], data[...] }
```

Plug-in image filters can be used to pre-filter an image before rendering. The filter has to be able to exist outside of the special environment of rendering in order to work here (it can't depend on functions or data that are only available during rendering). Filters are given by a server name, an enable flag, and data bytes that belong to the plug-in.

Plug-in Pixel Filters

```
PFLT { server-name[S0], flags[U2], data[...] }
```

Pixel filters may also be used as clip modifiers, and they are stored and used in a way that is exactly like image filters.

Surface Sub-chunks

The subchunks found in `SURF` chunks can be divided into two types. Basic surface parameters are stored in simple subchunks with no nested subchunks, while texture and shader data is stored in surface blocks containing nested subchunks.

Basic Surface Parameters

The following surface subchunks define the base characteristics of a surface. These are the "start" values for the surface, prior to texturing and plug-in shading, and correspond to the options on the main window of the LightWave Surface Editor. Even if textures and shaders completely obscure the base appearance of the surface in final rendering, these settings are still used for previewing and real-time rendering.

Base Color

```
COLR { base-color[COL12], envelope[VX] }
```

The base color of the surface, which is the color that lies under all the other texturing attributes.

Base Shading Values

```
DIFF, LUMI, SPEC, REFL, TRAN, TRNL { intensity[FP4], envelope[VX] }
```

The base level of the surface's diffuse, luminosity, specular, reflection, transparency, or translucency settings. Except for diffuse, if any of these subchunks is absent for a surface, a value of zero is assumed. The default diffuse value is 1.0.

Specular Glossiness

```
GLOS { glossiness[FP4], envelope[VX] }
```

Glossiness controls the falloff of specular highlights. The intensity of a specular highlight is calculated as $\cos^n a$, where a is the angle between the reflection and view vectors. The power n is the specular exponent. The `GLOS` chunk stores a glossiness g as a floating point fraction related to n by: $n = 2^{(10g + 2)}$. A glossiness of 20% (0.2) gives a specular exponent of 2^4, or 16, equivalent to the "Low" glossiness preset in versions of LightWave prior to 6.0. Likewise 40% is 64 or "Medium," 60% is 256 or "High," and 80% is 1024 or "Maximum."

The GLOS subchunk is only meaningful when the specularity in SPEC is non-zero. If GLOS is missing, a value of 40% is assumed.

Diffuse Sharpness

```
SHRP { sharpness[FP4], envelope[VX] }
```

Diffuse sharpness models non-Lambertian surfaces. The sharpness refers to the transition from lit to unlit portions of the surface, where the difference in diffuse shading is most obvious. For a sharpness of 0.0, diffuse shading of a sphere produces a linear gradient. A sharpness of 50% (0.5) corresponds to the fixed "Sharp Terminator" switch in versions of LightWave prior to 6.0. It produces planet-like shading on a sphere, with a brightly lit day side and a rapid falloff near the day/night line (the terminator). 100% sharpness is more like the Moon, with no falloff until just before the terminator.

Bump Intensity

```
BUMP { strength[FP4], envelope[VX] }
```

Bump strength scales the height of the bumps in the gradient calculation. Higher values have the effect of increasing the contrast of the bump shading. The default value is 1.0.

Polygon Sidedness

```
SIDE { sidedness[U2] }
```

The sidedness of a polygon can be 1 for front-only, or 3 for front and back. If missing, single-sided polygons are assumed.

Max Smoothing Angle

```
SMAN { max-smoothing-angle[ANG4] }
```

The maximum angle between adjacent polygons that will be smooth shaded. Shading across edges at higher angles won't be interpolated (the polygons will appear to meet at a sharp seam). If this chunk is missing, or if the value is <= 0, then the polygons are not smoothed.

Reflection Options

```
RFOP { reflection-options[U2] }
```

Reflection options is a numeric code that describes how reflections are handled for this surface and is only meaningful if the reflectivity in REFL is non-zero.

0 - Backdrop Only
: Only the backdrop is reflected.

1 - Raytracing + Backdrop
: Objects in the scene are reflected when raytracing is enabled. Rays that don't intercept an object are assigned the backdrop color.

2 - Spherical Map
: If an image is provided in an RIMG subchunk, the image is reflected as if it were spherically wrapped around the scene.

3 - Raytracing + Spherical Map
: Objects in the scene are reflected when raytracing is enabled. Rays that don't intercept an object are assigned a color from the image map.

If there is no RFOP subchunk, a value of 0 is assumed.

Reflection Map Image

```
RIMG { image[VX] }
```

A surface reflects this image as if it were spherically wrapped around the scene. The RIMG is only used if the reflection options in RFOP are set to use an image and the reflectivity of the surface in REFL is non-zero. The image is the index of a CLIP chunk, or zero to indicate no image.

Reflection Map Image Seam Angle

```
RSAN { seam-angle[ANG4], envelope[VX] }
```

This angle is the heading angle of the reflection map seam. If missing, a value of zero is assumed.

Reflection Blurring

```
RBLR { blur-percentage[FP4], envelope[VX] }
```

The amount of blurring of reflections. The default is zero.

Refractive Index

```
RIND { refractive-index[F4], envelope[VX] }
```

The surface's index of refraction. This is used to bend refraction rays when raytraced refraction is enabled in the scene. The value is the ratio of the speed of light in a vacuum to the speed of light in the material (always >= 1.0 in the real world). The default is 1.0.

Transparency Options

```
TROP { transparency-options[U2] }
```

The transparency options are the same as the reflection options in `RFOP`, but for refraction.

Refraction Map Image

```
TIMG { image[VX] }
```

Like `RIMG`, but for refraction.

Refraction Blurring

```
TBLR { blur-percentage[FP4], envelope[VX] }
```

The amount of refraction blurring. The default is zero.

Color Highlights

```
CLRH { color-highlights[FP4], envelope[VX] }
```

Specular highlights are ordinarily the color of the incident light. Color highlights models the behavior of dialectric and conducting materials, in which the color of the specular highlight tends to be closer to the color of the material. A higher color highlight value blends more of the surface color and less of the incident light color.

Color Filter

```
CLRF { color-filter[FP4], envelope[VX] }
```

The color filter percentage determines the amount by which rays passing through a transparent surface are tinted by the color of the surface.

Additive Transparency

```
ADTR { additive[FP4], envelope[VX] }
```

Additive transparency is a simple rendering trick that works independently of the mechanism associated with the `TRAN` and related settings. The color of the surface is added to the color of the scene elements behind it in a proportion controlled by the additive value.

Glow Effect

`GLOW { type[U2], intensity[F4], intensity-envelope[VX], size[F4], size-envelope[VX] }`

The glow effect causes a surface to spread and affect neighboring areas of the image. The type can be 0 for Hastings glow, and 1 for image convolution. The size and intensity define how large and how strong the effect is.

You may also encounter glow information written in a GVAL subchunk containing only the intensity and its envelope (the subchunk length is 6).

Render Outlines

`LINE { flags[U2], ( size[F4], size-envelope[VX], ( color[COL12], color-envelope[VX] )? )? }`

The line effect draws the surface as a wireframe of the polygon edges. Currently the only flag defined is an enable switch in the low bit. The size is the thickness of the lines in pixels, and the color, if not given, is the base color of the surface. Note that you may encounter LINE subchunks with no color information (these will have a subchunk length of 8 bytes) and possibly without size information (subchunk length 2).

Alpha Mode

`ALPH { mode[U2], value[FP4] }`

The alpha mode defines the alpha channel output options for the surface.

0 - Unaffected by Surface
: The surface has no effect on the alpha channel when rendered.

1 - Constant Value
: The alpha channel will be written with the constant value following the mode in the subchunk.

2 - Surface Opacity
: The alpha value is derived from surface opacity, which is the default if the ALPH chunk is missing.

3 - Shadow Density
: The alpha value comes from the shadow density.

Vertex Color Map

`VCOL { intensity[FP4], envelope[VX], vmap-type[ID4], name[S0] }`

The vertex color map subchunk identifies an RGB or RGBA VMAP that will be used to color the surface.

Surface Blocks

A surface may contain any number of *blocks* which hold texture layers or shaders. Each block is defined by a subchunk with the following format.

BLOK { header[SUB-CHUNK], attributes[SUB-CHUNK] * }

Since this regular expression hides much of the structure of a block, it may be helpful to visualize a typical texture block in outline form.

- block
  - header
    - ordinal string
    - channel
    - enable flag
    - opacity...
  - texture mapping
    - center
    - size...
  - other attributes...

The first subchunk is the header. The subchunk ID specifies the block type, and the subchunks within the header subchunk define properties that are common to all block types. The ordinal string defines the sorting order of the block relative to other blocks. The header is followed by other subchunks specific to each type. For some texture layers, one of these will be a texture mapping subchunk that defines the mapping from object to texture space. All of these components are explained in the following sections.

Ordinal Strings

Each BLOK represents a texture layer applied to one of the surface channels, or a shader plug-in applied to the surface. If more than one layer is applied to a channel, or more than one shader is applied to the surface, we need to know the evaluation order of the layers or shaders, or in what order they are "stacked." The ordinal string defines this order.

Readers can simply compare ordinal strings using the C `strcmp` function to sort the `BLOK`s into the correct order. Writers of `LWO2` files need to generate valid ordinal strings that put the texture layers and shaders in the right order. See the Object Examples supplement for an example function that generates ordinal strings.

To understand how LightWave uses these, imagine that instead of strings, it used floating-point fractions as the ordinals. Whenever LightWave needed to insert a new block between two existing blocks, it would find the new ordinal for the inserted block as the average of the other two, so that a block inserted between ordinals 0.5 and 0.6 would have an ordinal of 0.55.

But floating-point ordinals would limit the number of insertions to the (fixed) number of bits used to represent the mantissa. Ordinal strings are infinite-precision fractions written in base 255, using the ASCII values 1 to 255 as the digits (0 isn't used, since it's the special character that marks the end of the string).

Ordinals can't end on a 1, since that would prevent arbitrary insertion of other blocks. A trailing 1 in this system is like a trailing 0 in decimal, which can lead to situations like this,

```
   0.5     "\x80"
   0.50    "\x80\x01"
```

where there's no daylight between the two ordinals for inserting another block.

Block Headers

Every block contains a header subchunk.

block-header { ordinal[S0], block-attributes[SUB-CHUNK] * }

The ID of the header subchunk identifies the block type and can be one of the following.

`IMAP` - an image map texture
`PROC` - a procedural texture
`GRAD` - a gradient texture
`SHDR` - a shader plug-in

The header contains an ordinal string (described above) and subchunks that are common to all block types.

Channel

```
CHAN { texture-channel[ID4] }
```

This is required in all texture layer blocks and can have a value of `COLR`, `DIFF`, `LUMI`, `SPEC`, `GLOS`, `REFL`, `TRAN`, `RIND`, `TRNL`, or `BUMP`, The texture layer is applied to the corresponding surface attribute. If present in a shader block, this value is ignored.

Enable State

```
ENAB { enable[U2] }
```

True if the texture layer or shader should be evaluated during rendering. If `ENAB` is missing, the block is assumed to be enabled.

Opacity

```
OPAC { type[U2], opacity[FP4], envelope[VX] }
```

Opacity is valid only for texture layers. It specifies how opaque the layer is with respect to the layers before it (beneath it) on the same channel, or how the layer is combined with the previous layers. The types can be

0 - Normal
1 - Subtractive
2 - Difference
3 - Multiply
4 - Divide
5 - Alpha
6 - Texture Displacement
7 - Additive

Alpha opacity uses the current layer as an alpha channel. The previous layers are visible where the current layer is white and transparent where the current layer is black. Texture Displacement distorts the underlying layers. If OPAC is missing, 100% Additive opacity is assumed.

Displacement Axis

`AXIS { displacement-axis[U2] }`

For displacement mapping, defines the plane from which displacements will occur. The value is 0, 1 or 2 for the X, Y or Z axis.

Texture Mapping

Image map and procedural textures employ the TMAP subchunk to define the mapping they use to get from object or world coordinate space to texture space.

TMAP { attributes[SUB-CHUNK] * }

The TMAP subchunk contains a set of attribute chunks which describe the different aspects of this mapping.

Position, Orientation and Size

`CNTR, SIZE, ROTA { vector[VEC12], envelope[VX] }`

These subchunks each consist of a vector for the texture's size, center and rotation. The size and center are normal positional vectors in meters, and the rotation is a vector of heading, pitch and bank in radians. If missing, the center and rotation are assumed to be zero. The size should always be specified if it si to be used for any given mapping.

Reference Object

`OREF { object-name[S0] }`

Specifies a reference object for the texture. The reference object is given by name, and the scene position, rotation and scale of the object are combined with the previous chunks to compute the texture mapping. If the object name is "(none)" or OREF is missing, no reference object is used.

Falloff

```
FALL { type[U2], vector[VEC12], envelope[VX] }
```

Texture effects may fall off with distance from the texture center if this subchunk is present. The vector represents a rate per unit distance along each axis. The type can be

0 - Cubic
: Falloff is linear along all three axes independently.

1 - Spherical
: Falloff is proportional to the Euclidean distance from the center.

2 - Linear X
3 - Linear Y
4 - Linear Z
: Falloff is linear only along the specified axis. The other two vector components are ignored.

Coordinate System

```
CSYS { type[U2] }
```

The coordinate system can be 0 for object coordinates (the default if the chunk is missing) or 1 for world coordinates.

Image Maps

Texture blocks with a header type of `IMAP` are image maps. These use an image to modulate one of the surface channels. In addition to the basic parameters listed below, the block may also contain a `TMAP` chunk.

Projection Mode

```
PROJ { projection-mode[U2] }
```

The projection defines how 2D coordinates in the image are transformed into 3D coordinates in the scene. In the following list of projections, image coordinates are called *r* (horizontal) and *s* (vertical).

0 - Planar
: The image is projected on a plane along the major axis (specified in the `AXIS` subchunk). *r* and *s* map to the other

two axes.

1 - Cylindrical

The image is wrapped cylindrically around the major axis. *r* maps to longitude (angle around the major axis).

2 - Spherical

The image is wrapped spherically around the major axis. *r* and *s* map to longitude and latitude.

3 - Cubic

Like Planar, but projected along all three axes. The dominant axis of the geometric normal selects the projection axis for a given surface spot.

4 - Front Projection

The image is projected on the current camera's viewplane. *r* and *s* map to points on the viewplane.

5 - UV

r and *s* map to points (*u*, *v*) defined for the geometry using a vertex map (identified in the BLOK's VMAP subchunk).

Major Axis

```
AXIS { texture-axis[U2] }
```

The major axis used for planar, cylindrical and spherical projections. The value is 0, 1 or 2 for the X, Y or Z axis.

Image Map

```
IMAG { texture-image[VX] }
```

The CLIP index of the mapped image.

Image Wrap Options

```
WRAP { width-wrap[U2], height-wrap[U2] }
```

Specifies how the color of the texture is derived for areas outside the image.

0 - Reset

Areas outside the image are assumed to be black. The ultimate effect of this depends on the opacity settings. For an additive texture layer on the color channel, the final color will come from the preceding layers or from the base color of the surface.

1 - Repeat
The image is repeated or tiled.

2 - Mirror
Like repeat, but alternate tiles are mirror-reversed.

3 - Edge
The color is taken from the image's nearest edge pixel.

If no wrap options are specified, 1 is assumed.

Image Wrap Amount

```
WRPW, WRPH { cycles[FP4], envelope[VX] }
```

For cylindrical and spherical projections, these parameters control how many times the image repeats over each full interval.

UV Vertex Map

```
VMAP { txuv-map-name[S0] }
```

For UV projection, which depends on texture coordinates at each vertex, this selects the name of the TXUV vertex map that contains those coordinates.

Antialiasing Strength

```
AAST { flags[U2], antialising-strength[FP4] }
```

The low bit of the flags word is an enable flag for texture antialiasing. The antialiasing strength is proportional to the width of the sample filter, so larger values sample a larger area of the image.

Pixel Blending

```
PIXB { flags[U2] }
```

Pixel blending enlarges the sample filter when it would otherwise be smaller than a single image map pixel. If the low-order flag bit is set, then pixel blending is enabled.

Sticky Projection

```
STCK { on-off[U2], time[FP4] }
```

The "sticky" or fixed projection time for front projection image maps. When on, front projections will be fixed at the given time.

Texture Amplitude

```
TAMP { amplitude[FP4], envelope[VX] }
```

Appears in image texture layers applied to the bump channel. Texture amplitude scales the bump height derived from the pixel values. The default is 1.0.

Procedural Textures

Texture blocks of type PROC are procedural textures that modulate the value of a surface channel algorithmically.

Axis

```
AXIS { axis[U2] }
```

If the procedural has an axis, it may be defined with this chunk using a value of 0, 1 or 2.

Basic Value

```
VALU { value[FP4] # (1, 3) }
```

Procedurals are often modulations between the current channel value and another value, given here. This may be a scalar or a vector.

Algorithm and Parameters

```
FUNC { algorithm-name[S0], data[...] }
```

The FUNC subchunk names the procedural and stores its parameters. The name will often map to a plug-in name. The variable-length data following the name belongs to the procedural.

Gradient Textures

Texture blocks of type GRAD are gradient textures that modify a surface channel by mapping an input parameter through an arbitrary transfer function. Gradients are represented to the user as a line containing keys. Each key is a color, and the gradient function is an interpolation of the keys in RGB space. The input parameter selects a point on the line, and the output of the texture is the value of the gradient at that point.

Parameter Name

```
PNAM { parameter[S0] }
```

The input parameter. Possible values include

"Previous Layer"
"Bump"
"Slope"
"Incidence Angle"
"Light Incidence"
"Distance to Camera"
"Distance to Object"
"X Distance to Object"
"Y Distance to Object"
"Z Distance to Object"
"Weight Map"

Item Name

`INAM { item-name[S0] }`

The name of a scene item. This is used when the input parameter is derived from a property of an item in the scene.

Gradient Range

`GRST, GREN { input-range[FP4] }`

The start and end of the input range. These values only affect the display of the gradient in the user interface. They don't affect rendering.

Repeat Mode

`GRPT { repeat-mode[U2] }`

The repeat mode. This is currently undefined.

Key Values

`FKEY { ( input[FP4], output[FP4] # 4 )* }`

The transfer function is defined by an array of keys, each with an input value and an RGBA output vector. Given an input value, the gradient can be evaluated by selecting the keys whose positions bracket the value and interpolating between their outputs. If the input value is lower than the first key or higher than the last key, the gradient value is the value of the closest key.

Key Parameters

```
IKEY { interpolation[U2] * }
```

An array of integers defining the interpolation for the span preceding each key. Possible values include

0 - Linear
1 - Spline
2 - Step

Shaders

Shaders are BLOK subchunks with a header type of SHDR. They are applied to a surface after all basic channels and texture layers are evaluated, and in the order specified by the ordinal sequence. The only header chunk they support is ENAB and they need only one data chunk to describe them.

Shader Algorithm

```
FUNC { algorithm-name[S0], data[...] }
```

Just like a procedural texture layer, a shader is defined by an algorithm name (often a plug-in), followed by data owned by the shader.

Chunk Index

AAST	Image Map Antialiasing Strength
ADTR	Surface Additive Transparency
ALPH	Surface Alpha Mode
ANIM	Clip Animation
AXIS	Displacement Axis
AXIS	Image Map Major Axis
AXIS	Procedural Texture Axis
BBOX	Bounding Box
BLOK	Surface Block
BRIT	Clip Brightness
BUMP	Surface Bump Intensity
CHAN	Channel Plug-in

`CHAN`	Texture Layer Channel
`CLIP`	Image, Image Sequence
`CLRF`	Surface Color Filter
`CLRH`	Surface Color Highlights
`CNTR`	Texture Center
`COLR`	Surface Base Color
`CONT`	Clip Contrast
`CSYS`	Texture Coordinate System
`DESC`	Description Line
`DIFF`	Surface Diffuse
`ENAB`	Surface Block Enable
`ENVL`	Envelope
`FALL`	Texture Falloff
`FKEY`	Gradient Key Values
`FORM`	IFF Format File
`FUNC`	Procedural Texture Algorithm
`FUNC`	Surface Shader Algorithm
`GAMM`	Clip Gamma Correction
`GLOS`	Surface Specular Glossiness
`GLOW`	Surface Glow Effect
`GREN`	Gradient End
`GRPT`	Gradient Repeat Mode
`GRST`	Gradient Start
`HUE `	Clip Hue
`ICON`	Thumbnail Icon Image
`IFLT`	Clip Image Filter
`IKEY`	Gradient Key Parameters
`IMAG`	Image Map Image
`INAM`	Gradient Item Name
`ISEQ`	Clip Image Sequence
`KEY `	Keyframe Time and Value

LAYR Layer
LINE Surface Render Outlines
LUMI Surface Luminosity

NAME Envelope Channel Name
NEGA Clip Negative

OPAC Texture Layer Opacity
OREF Texture Reference Object
PFLT Clip Pixel Filter
PIXB Image Map Pixel Blending
PNAM Gradient Parameter Name
PNTS Point List
POLS Polygon List
POST Envelope Post-Behavior
PRE Envelope Pre-Behavior
PROJ Image Map Projection Mode
PTAG Polygon Tag Mapping

RBLR Reflection Blurring
REFL Surface Reflectivity
RFOP Surface Reflection Options
RIMG Surface Reflection Map Image
RIND Surface Refractive Index
ROTA Texture Rotation
RSAN Surface Reflection Map Image Seam Angle

SPAN Envelope Interval Interpolation
SATR Clip Saturation
SHRP Surface Diffuse Sharpness
SIDE Surface Polygon Sidedness
SIZE Texture Size
SMAN Surface Max Smoothing Angle
SPEC Surface Specularity
STCC Clip Color-cycling Still
STCK Sticky Projection
STIL Clip Still Image
SURF Surface Definition

TAGS	Tag Strings
TAMP	Image Map Texture Amplitude
TBLR	Refraction Blurring
TEXT	Commentary Text
TIME	Clip Time
TIMG	Surface Refraction Map Image
TMAP	Texture Mapping
TRAN	Surface Transparency
TRNL	Surface Translucency
TROP	Surface Transparency Options
TYPE	Envelope Type
VALU	Procedural Texture Value
VCOL	Surface Vertex Color Map
VMAD	Discontinuous Vertex Map
VMAP	Vertex Map
VMAP	Image Map UV Vertex Map
WRAP	Image Map Wrap Options
WRPW	Image Map Width Wrap Amount
WRPH	Image Map Height Wrap Amount
XREF	Clip Reference (Clone)

Scene Files

November 9, 2001

This document describes the LWSC version 3 file format for 3D scenes used by LightWave 6.0 and later. At this point, it's an incomplete rough draft that's missing descriptions of most of the keywords. But the introductory information will allow you to parse the file at least, and the semantics of most of the keywords can be deduced.

If you've worked with version 1 of the format (version 2 was an unreleased interim format), version 3 will seem quite familiar. Scene files are still text files containing keyword-value pairs. The most important difference is in the way keyframe data is stored, but obviously there are many others comprising features not available in LightWave prior to 6.0.

Item Numbers

When a scene file needs to refer to specific items to establish item relationships (parenting, for example), it uses item numbers. Items are numbered in the order in which they appear in the file, starting with 0.

Item numbers can be written in one of two ways, depending on which keyword they're used with. In general, if the type of the item (object, bone, light, camera) can be determined from the keyword alone, the item number will simply be the ordinal, written as a decimal integer. When the keyword can be used with items of more than one type, the item number is an unsigned integer written as an 8-digit hexadecimal string, the format produced by the C-language `"%8X"` print format specifier, and the high bits identify the item type.

The first hex digit (most significant 4 bits) of the hex item number string identifies the item type.

1 - Object
2 - Light
3 - Camera

4 - Bone

The other digits make up the item number, except in the case of bones. For bones, the next 3 digits (bits 16-27) are the bone number and the last 4 digits (bits 0-15) are the object number for the object the bone belongs to. Some examples:

`10000000` - the first object
`20000000` - the first light
`4024000A` - the 37th bone (24 hex) in the 11th object (0A hex)

Blocks

Information in a scene file is organized into blocks, the ASCII text analog of the chunks described in the IFF specification. Each block consists of an identifier or name followed by some data. The format of the data is determined by the block name. Block names resemble C-style identifiers. In particular, they never contain spaces or other non-alphanumeric characters.

A single-line block is delimited by the newline that terminates the line. Multiline blocks are delimited by curly braces (the **{** and **}** characters, ASCII codes 123 and 125). The name of a multiline block follows the opening curly brace on the same line. The curly brace and the name are separated by a single space. The data follows on one or more subsequent lines. Each line of data is indented using two spaces. The closing brace is on a line by itself and is not indented.

Individual data elements are separated from each other by a single space. String data elements are enclosed in double quotes and may contain spaces.

Blocks can be nested. In other words, the data of a block can include other blocks. A block that contains nested blocks is always a multiline block. At each nesting level, the indention of the data is incremented by two additional spaces.

```
SingleLineBlock data
{ MultiLineBlock
  data
  { NestedMultiLineBlock
    data
    }
  }
```

Envelopes

An envelope defines a function of time. For any animation time, an envelope's parameters can be combined to generate a value at that time. Envelopes are used to store position coordinates, rotation angles, scale factors, camera zoom, light intensity, texture parameters, and anything else that can vary over time.

The envelope function is a piecewise polynomial curve. The function is tabulated at specific points, called *keys*. The curve segment between two adjacent keys is called a *span*, and values on the span are calculated by interpolating between the keys. The interpolation can be linear, cubic, or stepped, and it can be different for each span. The value of the function before the first key and after the last key is calculated by extrapolation.

In scene files, an envelope is stored in a block named `Envelope` that contains one or more nested `Key` blocks and one `Behaviors` block.

```
  { Envelope
    nkeys
    Key value time spantype p1 p2 p3 p4 p5 p6
    Key ...
    Behaviors pre post
  }
```

The `nkeys` value is an integer, the number of `Key` blocks in the envelope. Envelopes must contain at least one `Key` block. The contents of a `Key` block are as follows.

value
: The key value, a floating-point number. The units and limits of the value depend on what parameter the envelope represents.

time
: The time in seconds, a float. This can be negative, zero or positive. Keys are listed in the envelope in increasing time order.

spantype
: The curve type, an integer. This determines the kind of interpolation that will be performed on the span between this key and the previous

key, and also indicates what interpolation parameters are stored for the key.

0 - TCB (Kochanek-Bartels)
1 - Hermite
2 - 1D Bezier (obsolete, equivalent to Hermite)
3 - Linear
4 - Stepped
5 - 2D Bezier

p1...p6
Curve parameters. The data depends on the span type.

TCB, Hermite, 1D Bezier
The first three parameters are tension, continuity and bias. The fourth and fifth parameters are the incoming and outgoing tangents. The sixth parameter is ignored and should be 0. Use the first three to evaluate TCB spans, and the other two to evaluate Hermite spans.

2D Bezier
The first two parameters are the incoming time and value, and the second two are the outgoing time and value.

The `Behaviors` block contains two integers.

pre, **post**
Pre- and post-behaviors. These determine how the envelope is extrapolated at times before the first key and after the last one.

0 - Reset
Sets the value to 0.0.

1 - Constant
Sets the value to the value at the nearest key.

2 - Repeat
Repeats the interval between the first and last keys (the primary interval).

3 - Oscillate
Like Repeat, but alternating copies of the primary interval are time-reversed.

4 - Offset Repeat
Like Repeat, but offset by the difference between the values of the first and last keys.
5 - Linear
Linearly extrapolates the value based on the tangent at the nearest key.

The source code in the sample/envelope directory of the LightWave plug-in SDK demonstrates how envelopes are evaluated.

Scene

FirstFrame **n**first
LastFrame **n**last
FrameStep **n**step
The frame range and step size for rendering. In the simplest case, the first frame and frame step are 1, and the last frame is the number of frames to be rendered.

PreviewFirstFrame **n**first
PreviewLastFrame **n**last
PreviewFrameStep **n**step
The frame range and step size for previewing. These may be unrelated to the values for rendering. They also control the visible ranges of certain interface elements, for example the frame slider in the main window.

CurrentFrame **n**frame
The frame displayed in the interface when the scene is loaded.

FramesPerSecond **g**frames
This controls the duration of each frame.

Objects

LoadObjectLayer **n**layer **s**filename
Begins a group of statements about an object. The layer number is the index recorded in the LAYR chunk of the object file.

ShowObject **n**visibility **n**color

Determines how the object is displayed in the interface. The visibility codes are

0 - hidden
1 - bounding box
2 - vertices only
3 - wireframe
4 - front face wireframe
5 - shaded solid
6 - textured shaded solid

The color used to draw bounding boxes, vertices and wireframes can be one of the following.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
⬛	🟦	🟩	🟦	🟥	🟪	🟫	⬜	🟦	🟩	🟦	🟥	🟪	🟨	⬜

The default visibility and color are stored in the config file. If they haven't been altered by the user, they are textured shaded solid (6) and cyan (3) in LightWave 6.5.

ObjectMotion
NumChannels **n**channels
Channel **n**index
{ Envelope ...

The `ObjectMotion` keyword signals the start of the motion information for the object. Motions are stored in envelopes, one for each motion channel. There are 9 standard channels, numbered from 0 to 8.

0, 1, 2 - (x, y, z) position
3, 4, 5 - (heading, pitch, bank) rotation
6, 7, 8 - (sx, sy, sz) scale factors along each axis

The values of all of these are relative to the object's parent, if it has one.

UseBonesFrom 1

.

Plugin **s**class **n**listpos **s**name
EndPlugin
Lists an object plug-in. The class can be

MorphAmount
MorphTarget
MorphSurfaces
MTSEMorphing
.

DisplacementMaps
{ TextureBlock ...
.

ClipMaps
{ TextureBlock ...
.

ObjectDissolve
.

DistanceDissolve
.

MaxDissolveDistance
.

AffectedByFog
.

UnseenByRays
.

UnseenByCamera
.

ShadowOptions
.

ExcludeLight

.

PolygonEdgeFlags

.

PolygonEdgeThickness

.

PolygonEdgeColor

.

PolygonEdgesZScale

.

EdgeNominalDistance

.

Animation Envelopes

Availability LightWave 6.0
Component Layout, Modeler
Header lwenvel.h

A key is a structure that holds the value of an animation parameter at a specific time. An envelope is an array of keys, along with methods for interpolation (tweening) and extrapolation (what happens to the parameter value before the first key and after the last one). The Animation Envelopes global returns functions that allow you to create and manage envelopes and their keys, including a function to display an interface to the user for editing envelopes.

Other global mechanisms are built on top of envelopes. A channel contains the continuous value of a parameter as a function of time, and this is based on both the underlying envelope and on external effects, including plug-ins (channel and item motion classes, for example) that can alter channel values. And the Variant Parameters global defines a data type used by XPanel envelope controls.

See also the Motions section of the Layout commands page, as well as the commands supported by the Graph and Surface Editors.

Global Call

```
LWEnvelopeFuncs *envfunc;
envfunc = global( LWENVELOPEFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWEnvelopeFuncs.

```
typedef struct st_LWEnvelopeFuncs {
   LWEnvelopeID    (*create)      (LWChanGroupID, const char *, int);
   void            (*destroy)     (LWEnvelopeID);
   LWChanGroupID   (*createGroup) (LWChanGroupID parent, const char *);
   void            (*destroyGroup)(LWChanGroupID);
   LWError         (*copy)        (LWEnvelopeID to, LWEnvelopeID from);
   LWError         (*load)        (LWEnvelopeID, LWLoadState *);
   LWError         (*save)        (LWEnvelopeID, LWSaveState *);
   double          (*evaluate)    (LWEnvelopeID, LWTime);
   int             (*edit)        (LWChanGroupID, LWEnvelopeID,
                                     int flags, void *data);
   int             (*envAge)      (LWEnvelopeID);
   LWEnvKeyframeID (*createKey)   (LWEnvelopeID, LWTime, double value);
```

```
      void            (*destroyKey)  (LWEnvelopeID, LWEnvKeyframeID);
      LWEnvKeyframeID (*findKey)     (LWEnvelopeID, LWTime);
      LWEnvKeyframeID (*nextKey)     (LWEnvelopeID, LWEnvKeyframeID);
      LWEnvKeyframeID (*prevKey)     (LWEnvelopeID, LWEnvKeyframeID);
      int             (*keySet)      (LWEnvelopeID, LWEnvKeyframeID,
                                        LWKeyTag, void *value);
      int             (*keyGet)      (LWEnvelopeID, LWEnvKeyframeID,
                                        LWKeyTag, void *value);
      int             (*setEnvEvent) (LWEnvelopeID, LWEnvEventFunc,
                                        void *data);
      int             (*egSet)       (LWEnvelopeID, LWChanGroupID,
                                        int tag, void *value);
      int             (*egGet)       (LWEnvelopeID, LWChanGroupID,
                                        int tag, void *value);
   } LWEnvelopeFuncs;
```

env = **create**(group, name, type)

Create a new envelope. The type defines how the envelope's values are interpreted and displayed to the user in the graph editor. It can be one of the following.

LWET_FLOAT
LWET_DISTANCE
LWET_PERCENT
LWET_ANGLE

destroy(env)

Destroy an envelope created using `create`.

group = **createGroup**(parent, name)

Create a new envelope group. An envelope group is just a way to organize related envelopes.

destroyGroup(group)

Destroy an envelope group created using `createGroup`.

error = **copy**(to, from)

Copy an envelope. This is meant to be called from within a handler's `copy` callback.

error = **load**(env, loadstate)

Load an envelope. This is meant to be called from within a handler's `load` callback.

error = **save**(env, savestate)

Save an envelope. This is meant to be called from within a handler's `save` callback.

value = **evaluate**(env, time)

Returns the interpolated value of the envelope.

`result = `**`edit`**`( group, env, flags, data )`

Open the graph editor window and allow the user to edit the envelope. The flags and data arguments are currently unused.

`age = `**`envAge`**`( env )`

Returns an integer containing the number of times the envelope has been changed.

`key = `**`createKey`**`( env, time, value )`

Create a new key in an envelope.

`destroyKey``( env, key )`

Delete a key in an envelope.

`key = `**`findKey`**`( env, time )`

Returns the key for a given time.

`key = `**`nextKey`**`( env, key )`

Returns the next key in the envelope.

`key = `**`prevKey`**`( env, key )`

Returns the previous key in the envelope.

`result = `**`keySet`**`( env, key, tag, value )`

Set a value associated with a key. This can be the value of the key itself, the shape of the key, or one of the interpolation parameters. The result is true (non-zero) if the function succeeds and false (0) if it fails. The tag describing the value can be one of the following.

LWKEY_VALUE
: The value of the key.

LWKEY_SHAPE
: The curve type, an integer corresponding to the options in the graph editor:
 0 - TCB (Kochanek-Bartels)
 1 - Hermite
 2 - 1D Bezier (obsolete, equivalent to Hermit)
 3 - Linear
 4 - Stepped
 5 - 2D Bezier

LWKEY_TENSION
LWKEY_CONTINUITY
LWKEY_BIAS
: The Kochanek-Bartels blending parameters.

LWKEY_PARAM_0

LWKEY_PARAM_1
LWKEY_PARAM_2
LWKEY_PARAM_3

The curve parameters. These are the Hermite coefficients for Hermite curves, and the incoming and outgoing tangents for 2D Bezier curves.

`result = keyGet( env, key, tag, value )`

Get a value associated with a key. The result is true (non-zero) if the function succeeds and false (0) if it fails. The tags are the same as those for `keySet`, along with `LWKEY_TIME`, the time of the key.

`result = setEnvEvent( env, event_func, data )`

Set a callback for an envelope. Whenever the envelope is modified, your `event_func` function will be called with `data` as its first argument. Currently the result is false (0) if `event_func` is NULL and true (non-zero) otherwise.

When you no longer need it, you must unhook your event callback by calling `setEnvEvent` again with a NULL `event_func` argument. (But if your callback has already been called for an `LWEEVNT_DESTROY` event, don't try to unhook it, since at that point the envelope no longer exists.) The `data` argument should be the same as it was in the original call. This argument is used to uniquely identify the owner of a callback, which is necessary because more than one event callback can be set for a given envelope. For the same reason, `data` should not be NULL.

`result = egSet( env, group, tag, value )`

Set a value associated with the envelope. The result is true (non-zero) if the function succeeds and false (0) if it fails.

LWENVTAG_VISIBLE

Invisible envelopes won't appear in the graph editor. You can use these to store internal variables. The value for this tag is an integer containing true (1) or false (0).

LWENVTAG_PREBEHAVE
LWENVTAG_POSTBEHAVE

Pre- and post-behavior setting, an integer corresponding to the options in the graph editor:

0 - Reset

1 - Constant
2 - Repeat
3 - Oscillate
4 - Offset Repeat
5 - Linear

`result = **egGet**( env, group, tag, value )`

Get a value associated with an envelope. In addition to the value types defined for `egSet`, you can use `LWENVTAG_KEYCOUNT` to get the number of keys defined for the envelope. The result is true (non-zero) if the function succeeds and false (0) if it fails.

Event Callback

The `setEnvEvent` function lets you set a callback that LightWave will call whenever an envelope is modified. The callback looks like this.

```
   typedef int (*LWEnvEventFunc) (void *data, LWEnvelopeID env,
      LWEnvEvent event, void *eventData);
```

`data` is what you passed as the third argument to the `setEnvEvent` function. The `eventData` depends on the event, which can be one of the following.

```
   LWEEVNT_DESTROY
   LWEEVNT_KEY_INSERT
   LWEEVNT_KEY_DELETE
   LWEEVNT_KEY_VALUE
   LWEEVNT_KEY_TIME
```

For the `KEY` events, the `eventData` is the LWKeyframeID. For the `DESTROY` event, the `eventData` is currently undefined and the LWEnvelopeID is invalid. When your callback is called for a `DESTROY` event, the envelope has already been destroyed, and you should ensure that you invalidate any of your own references to the envelope.

Example

The envelope sample shows how envelopes are interpolated. It also uses the the envelope global functions to create and examine the envelope to be interpolated.

The following code fragment finds a key for the red level of the first light at 5 seconds. If the light doesn't have a color envelope, we add it using the

`AddEnvelope` command, and if there's no key at 5 seconds, we create it. The key value (the red level) is set to 0.75.

In order to do this, we need to find the item ID for the first light, the channel group for that light, the red channel in the channel group, the underlying envelope for the red channel, and the key in that envelope at 5 seconds, if it exists. In addition to the envelope global, we use the channel info, item info and message globals.

```
#include <lwserver.h>
#include <lwenvel.h>
#include <lwhost.h>

LWEnvelopeFuncs *envf;
LWChannelInfo *chinfo;
LWItemInfo *iteminfo;
LWMessageFuncs *msgf;
LWItemID id;
LWChanGroupID group;
LWEnvelopeID envred;
LWEnvKeyframeID key;
char buf[ 128 ];
double val;

chinfo = global( LWCHANNELINFO_GLOBAL, GFUSE_TRANSIENT );
envf = global( LWENVELOPEFUNCS_GLOBAL, GFUSE_TRANSIENT );
iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
msg = global( LWMESSAGEFUNCS_GLOBAL, GFUSE_TRANSIENT );

if ( !chinfo || !envf || !iteminfo || !msgf )
   return AFUNC_BADGLOBAL;

id = iteminfo->first( LWI_LIGHT, NULL );
group = iteminfo->chanGroup( id );

envred = findEnv( group, "Color.R" );
if ( !envred ) {
   sprintf( buf, "SelectItem %x", id );
   local->evaluate( local->data, buf );
   local->evaluate( local->data, "AddEnvelope Color.R" );
   envred = findEnv( group, "Color.R" );
}
if ( !envred ) {
   msgf->info( "Couldn't create an envelope for",
      iteminfo->name( id ));
   return AFUNC_OK;
}

val = 0.75;
key = envf->findKey( envred, 5.0 );
if ( !key )
   key = envf->createKey( envred, 5.0, val );
if ( key )
   envf->keySet( envred, key, LWKEY_VALUE, &val );
else {
   sprintf( buf, "%s.Color.R", iteminfo->name( id ));
   msg->info( "Couldn't create a key in", buf );
}
```

Our `findEnv` function simply loops through the channels in a channel group searching for a given channel name. If a match is found, it returns the envelope ID for the channel.

```
LWEnvelopeID findEnv( LWChanGroupID group, char *name )
{
   LWChannelID chan;

   chan = chinfo->nextChannel( group, NULL );
   while ( chan ) {
      if ( !strcmp( chinfo->channelName( chan ), name ))
         return chinfo->channelEnvelope( chan );
      chan = chinfo->nextChannel( group, chan );
   }
   return NULL;
}
```

Backdrop Info

Availability LightWave 6.0
Component Layout
Header lwrender.h

The backdrop info global returns a function that evaluates the color of the backdrop in a specific direction at a given time, as well as the type, colors and squeeze values for the default solid backdrop. The parameters are read-only, but you can set them using commands.

Global Call

```
LWBackdropInfo *bkdropinfo;
bkdropinfo = global( LWBACKDROPINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWBackdropInfo.

```
typedef struct st_LWBackdropInfo {
   void     (*backdrop) (LWTime, const double ray[3], double color[3]);
   int       type;
   void     (*color)    (LWTime, double zenith[3], double sky[3],
                           double ground[3], double nadir[3]);
   void     (*squeeze)  (LWTime, double *sky, double *ground);
} LWBackdropInfo;
```

backdrop(time, ray, color)
: Sets the `color` argument to the RGB levels of the backdrop color in the `ray` direction at the specified time. Several effects can cause this color to differ entirely from the one implied by the other members of the LWBackdropInfo.

type
: `LWBACK_SOLID` (the default backdrop is a single uniform color) or `LWBACK_GRADIENT` (the default backdrop is a gradient derived from the zenith, sky, ground and nadir colors).

color(time, zenith, sky, ground, nadir)
: The arrays are filled with the RGB levels for each of the four gradient nodes.

squeeze(time, sky, ground)

The squeeze amount is stored in the `sky` and `ground` arguments. A squeeze of 1.0 produces a linear interpolation between the horizon and the pole, while higher amounts cause the color to vary more quickly near the horizon.

Example

This code fragment shows how to obtain the backdrop color in a given direction.

```
#include <lwserver.h>
#include <lwrender.h>

LWBackDropInfo *bkdropinfo;
double ray[ 3 ], color[ 3 ], dx, dy, dz, d;
LWTime t;

bkdropinfo = global( LWBACKDROPINFO_GLOBAL, GFUSE_TRANSIENT );
if ( !bkdropinfo ) return AFUNC_BADGLOBAL;
...

/* normalize the direction ray */
d = sqrt( dx * dx + dy * dy + dz * dz );

if ( d > 0 ) {
   ray[ 0 ] = dx / d;
   ray[ 1 ] = dy / d;
   ray[ 2 ] = dz / d;
   bkdropinfo->backdrop( t, ray, color );
   ...
```

Bone Info

Availability LightWave 6.0
Component Layout
Headerlwrender.h

The bone info global returns functions for getting bone-specific information about any of the bones in a scene. Use the item info global to get the bone list and for generic item information. The data returned by these functions is read-only, but you can use commands to set many of the parameters.

Global Call

```
LWBoneInfo *boneinfo;
boneinfo = global( LWBONEINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWBoneInfo.

```
typedef struct st_LWBoneInfo {
   unsigned int (*flags)     (LWItemID);
   void         (*restParam) (LWItemID, LWItemParam, LWDVector vec);
   double       (*restLength)(LWItemID);
   void         (*limits)    (LWItemID, double *inner, double *outer);
   const char * (*weightMap) (LWItemID);
   double       (*strength)  (LWItemID);
   int          (*falloff)   (LWItemID);
   void         (*jointComp) (LWItemID, double *self, double *parent);
   void         (*muscleFlex)(LWItemID, double *self, double *parent);
} LWBoneInfo;
```

boneflags = **flags**(bone)
: Returns a set of flag bits combined using bitwise-or. The flags are

  LWBONEF_ACTIVE
  : The bone is active.

  LWBONEF_LIMITED_RANGE
  : The bone has a limited range.

  LWBONEF_SCALE_STRENGTH
  : The strength of the bone is scaled by the rest length.

  LWBONEF_WEIGHT_MAP_ONLY
  : Deformation will be based solely on the weight map.

  LWBONEF_WEIGHT_NORM

The weight normalization option is turned on. The relative strength of each weight map value is scaled so that the total for all values is 1.0.

LWBONEF_JOINT_COMP
LWBONEF_JOINT_COMP_PAR

Joint compensation is enabled for the bone. This can also account for the rotation of the bone's parent.

LWBONEF_MUSCLE_FLEX
LWBONEF_MUSCLE_FLEX_PAR

Muscle flexing is enabled for the bone. Like joint compensation, this is a volume preserving adjustment to the deformation caused by the bone and can include the effect of the bone's parent.

restParam(bone, param_type, vector)

Gets vector parameters for the rest position of a given bone. Parameters of the animated bone can be read from the normal item info functions. See the item info parameter list for the values that can be passed in the `param_type` argument.

length = **restLength**(bone)

Returns the rest length of the bone.

limits(bone, inner_limit, outer_limit)

For limited range bones, this gets the inner and outer limit radii for the bone.

name = **weightMap**(bone)

Returns the name of the weight map for the bone. The weight map is a vertex map of type `LWVMAP_WGHT`. The object info and scene objects globals provide functions for reading the values in a vmap.

bone_strength = **strength**(bone)

Returns the bone strength setting.

type = **falloff**(bone)

Returns the falloff as an index into an options list. In general, the falloff function is the distance raised to the power -2^{type}. A `type` of 0 is inverse distance, 1 is inverse distance squared, 2 is inverse distance to the fourth power, and so on.

jointComp(bone, self, parent)
Fills in `self` and `parent` with the joint compensation amount.

muscleFlex(bone, self, parent)
Fills in `self` and `parent` with the muscle flexing amount.

History

In LightWave 7.0, the server name for this global (`LWBONEINFO_GLOBAL`) was incremented from "LW Bone Info 2" to "LW Bone Info 3". The following functions and flags were added.

```
strength
falloff
jointComp
muscleFlex

LWBONEF_JOINT_COMP
LWBONEF_JOINT_COMP_PAR
LWBONEF_MUSCLE_FLEX
LWBONEF_MUSCLE_FLEX_PAR
```

Example

This code fragment collects information about the bones in the scene.

```
#include <lwserver.h>
#include <lwrender.h>

LWItemInfo *iteminfo;
LWBoneInfo *boneinfo;
LWItemID object, bone;
unsigned int flags;
LWDVector pos;
double restlen;

iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
boneinfo = global( LWBONEINFO_GLOBAL, GFUSE_TRANSIENT );
if ( !iteminfo || !boneinfo ) return AFUNC_BADGLOBAL;

object = iteminfo->first( LWI_OBJECT, NULL );
while ( object ) {
   bone = iteminfo->first( LWI_BONE, object );
   while ( bone ) {
      flags = boneinfo->flags( bone );
      boneinfo->restParam( bone, LWIP_POSITION, pos );
      restlen = boneinfo->restLength( bone );
      ...

      bone = iteminfo->next( bone );
   }
```

```
    object = iteminfo->next( object );
}
```

Camera Info

Availability LightWave 6.0
Component Layout
Header lwrender.h

The camera info global returns functions for getting camera-specific information about any of the cameras in a scene. Use the item info global to get the camera list and for generic item information. The information returned by these functions is read-only, but you can set camera parameters using commands.

Global Call

```
   LWCameraInfo *caminfo;
   caminfo = global( LWCAMERAINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWCameraInfo.

```
   typedef struct st_LWCameraInfo {
      double       (*zoomFactor)    (LWItemID, LWTime);
      double       (*focalLength)   (LWItemID, LWTime);
      double       (*focalDistance) (LWItemID, LWTime);
      double       (*fStop)         (LWItemID, LWTime);
      double       (*blurLength)    (LWItemID, LWTime);
      void         (*fovAngles)     (LWItemID, LWTime, double *hfov,
                                       double *vfov);
      unsigned int (*flags)         (LWItemID);
      void         (*resolution)    (LWItemID, int *w, int *h);
      double       (*pixelAspect)   (LWItemID, LWTime);
      double       (*separation)    (LWItemID, LWTime);
      void         (*regionLimits)  (LWItemID, int *x0, int *y0, int *x1,
                                       int *y1);
      void         (*maskLimits)    (LWItemID, int *x0, int *y0, int *x1,
                                       int *y1);
      void         (*maskColor)     (LWItemID, LWDVector color);
   } LWCameraInfo;
```

`zoom = zoomFactor( camera, time )`
: Returns the zoom factor.

flen = **focalLength**(camera, time)
: Returns the focal length in millimeters.

fdist = **focalDistance**(camera, time)
: Returns the distance from the camera at which objects are in focus.

fstop = **fStop**(camera, time)
Returns the f-stop number.

blurlen = **blurLength**(camera, time)
Returns the blur length as a fraction of the frame time.

fovAngles(camera, time, hfov, vfov)
Gets the `hfov` and `vfov` (horizontal and vertical field of view) angles, expressed in radians.

f = **flags**(camera);
Returns flags describing the camera, combined using bitwise-or.

```
LWCAMF_STEREO
LWCAMF_LIMITED_REGION
LWCAMF_MASK
```

resolution(camera, width, height)
Gets the image size in pixels for the images rendered by the camera.

aspect = **pixelAspect**(camera, time)
Returns the pixel aspect ratio of images rendered by the camera, expressed as width/height. Values greater than 1.0 mean short wide pixels and values less than 1.0 mean tall thin pixels.

sep = **separation**(camera, time)
Returns the interocular distance (eye separation) for stereoscopic rendering, in meters.

regionLimits(camera, x0, y0, x1, y1)
Gets the limited region rectangle for the camera.

maskLimits(camera, x0, y0, x1, y1)
Gets the mask rectangle for the camera.

maskColor(camera, color)
Gets the color that will be rendered in areas of the image outside the mask rectangle.

Example

This code fragment collects information about the first camera.

```
#include <lwserver.h>
#include <lwrender.h>

LWItemInfo *iteminfo;
LWCameraInfo *caminfo;
LWItemID id;
LWTime t = 3.0;          /* seconds */
double zoom, flen, fdist, fstop, blen, hfov, vfov;

iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
caminfo  = global( LWCAMERAINFO_GLOBAL, GFUSE_TRANSIENT );

if ( iteminfo && caminfo ) {
   id = iteminfo->first( LWI_CAMERA, NULL );
   zoom  = caminfo->zoomFactor( id, t );
   flen  = caminfo->focalLength( id, t );
   fdist = caminfo->focalDistance( id, t );
   fstop = caminfo->fStop( id, t );
   blen  = caminfo->blurLength( id, t );
   fovAngles( id, t, &hfov, &vfov );
}
```

Channel Info

Availability LightWave 6.0
Component Layout, Modeler
Header lwenvel.h

A channel is an animation parameter that varies as a function of time. In contrast to envelopes, which are arrays of keys, channels may include the effects of other plug-ins and calculations. The channel info global gives you access to Layout's list of grouped channels. A channel's underlying envelope data may also be read (see the Animation Envelopes page for more information).

Global Call

```
   LWChannelInfo *chaninfo;
   chaninfo = global( LWCHANNELINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global call returns a pointer to an LWChannelInfo.

```
   typedef struct st_LWChannelInfo {
      LWChanGroupID (*nextGroup)       (LWChanGroupID parent,
                                          LWChanGroupID group);
      LWChannelID   (*nextChannel)     (LWChanGroupID, LWChannelID);
      const char *  (*groupName)       (LWChanGroupID);
      const char *  (*channelName)     (LWChannelID);
      LWChanGroupID (*groupParent)     (LWChanGroupID);
      LWChanGroupID (*channelParent)   (LWChannelID);
      int           (*channelType)     (LWChannelID);
      double        (*channelEvaluate) (LWChannelID, LWTime);
      const LWEnvelopeID (*channelEnvelope) (LWChannelID);
      int           (*setChannelEvent) (LWChannelID,
                                          LWChanEventFunc, void *);
      const char *  (*server)          (LWChannelID, const char *class,
                                          int index);
      unsigned int  (*serverFlags)     (LWChannelID, const char *class,
                                          int index );
      LWInstance    (*serverInstance)  (LWChannelID, const char *class,
                                          int index );
      int           (*serverApply)     (LWChannelID, const char *class,
                                          const char *name, int flags );
      void          (*serverRemove)    (LWChannelID, const char *class,
                                          const char *name, LWInstance);
   } LWChannelInfo;
```

`group = nextGroup( parent_group, prev_group )`

Returns the next channel group in the group list. If the parent is NULL, this returns groups from the root of the channel tree, and if the previous group is NULL, it returns the first group.

`channel = `**`nextChannel`**`( group, prev_channel )`

Returns the next channel in the group. If the previous channel is NULL, this returns the first channel in the group.

`gname = `**`groupName`**`( group )`

Returns the name of the channel group.

`cname = `**`channelName`**`( channel )`

Returns the name of the channel.

`parent = `**`groupParent`**`( group )`

Returns the parent group of a channel group.

`parent = `**`channelParent`**`( channel )`

Returns the group a channel belongs to.

`type = `**`channelType`**`( channel )`

Returns the value type of the channel, which determines how the value is interpreted and displayed to the user. It can be one of the following.

```
LWET_FLOAT
LWET_DISTANCE
LWET_PERCENT
LWET_ANGLE
```

`value = `**`channelEvaluate`**`( channel, time )`

Returns the value of the channel at the specified time.

`envelope = `**`channelEnvelope`**`( channel )`

Returns the underlying envelope for a channel. The envelope can be examined and modified using the Animation Envelopes global.

`result = `**`setChannelEvent`**`( channel, event_func, data )`

Set a callback for a channel. Whenever the channel's underlying envelope is modified, your `event_func` function will be called with `data` as its first argument. The result is true (non-zero) if the function succeeds and false (0) if it fails. The callback receives the same event codes as the envelope global's `setEnvEvent` function, plus `LWCEVNT_VALUE`.

`servname = `**`server`**`( channel, class, index )`

Returns the name of a plug-in applied to the channel. The `class` argument is the class string, which will often be `LWCHANNEL_HCLASS` but may be others. The index specifies the "slot," or position in the server list, and counts from 1.

`flags = `**`serverFlags`**`( channel, class, index )`

Returns flags for the plug-in applied to the channel. This is the channel-specific version of the Item Info `serverFlags` function.

`instance = **serverInstance**( channel, class, index )`

Returns an opaque pointer to the plug-in's instance data. This is the LWInstance created and used by the plug-in's LWInstanceFuncs callbacks.

`index = **serverApply**( channel, class, name, flags )`

Apply the plug-in to the channel. Returns the server list index, or 0 if it fails. The `name` is the server name, the string in the `name` field of the plug-in's ServerRecord. The `flags` can be any combination of those returned by `serverFlags`.

`**serverRemove**( channel, class, name, instance )`

Remove the plug-in from the channel. The `instance` is the pointer returned by `serverInstance`.

History

In LightWave 7.0, the service name for this global was incremented from "Channel Info" to "Channel Info 2", and the `serverFlags`, `serverInstance`, `serverApply` and `serverRemove` functions were added, along with the `LWCEVNT_VALUE` event code.

Example

The `find_channels` function finds all of the channels belonging to a group and prints the name and type of each channel. It calls itself recursively to examine the subgroups of the parent group.

```
#include <lwserver.h>
#include <lwenvel.h>

static void find_channels( LWChannelInfo *chinfo,
   LWChanGroupID parent, int indent )
{
   LWChanGroupID group;
   LWChannelID chan;

   group = chinfo->nextGroup( parent, NULL );
   while ( group ) {
      printf( "%*s(G) \"%s\"\n", indent, " ",
         chinfo->groupName( group ));
      find_channels( chinfo, group, indent + 2 );
      chan = chinfo->nextChannel( group, NULL );
      while ( chan ) {
         printf( "%*s(C) \"%s\" type %d\n", indent + 2, " ",
            chinfo->channelName( chan ),
            chinfo->channelType( chan ));
         chan = chinfo->nextChannel( group, chan );
      }
      group = chinfo->nextGroup( parent, group );
   }
}
```

Color Picker

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The color picker global returns a function that prompts the user for a color selection. The request displays the color dialog currently installed in LightWave. This may be the default system dialog or a custom ColorPicker plug-in.

The function returned by the color picker global calls the color picker module's activation function directly. Plug-ins calling the function act as the host side of the ColorPicker plug-in class.

Global Call

```
   LWColorActivateFunc *colorpick;
   colorpick = global( LWCOLORACTIVATEFUNC_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWColorActivateFunc.

```
   typedef int LWColorActivateFunc (int version, LWColorPickLocal *);
```

The return value of this function can be any of the values defined for the return values of activation functions. Any value other than `AFUNC_OK` must be handled as an error.

The version is passed as the `version` argument to the color picker's activation function. This should be set to the value defined by the `LWCOLORPICK_VERSION` symbol in `lwdialog.h`. Color picker plug-ins with a different activation version will return `AFUNC_BADVERSION`.

The second argument to this function is a pointer to a structure that is passed as the `local` argument to the color picker's activation function.

The Local Structure

The color picker function passes an LWColorPickLocal as the local argument to the activation function of the color picker plug-in.

```
   typedef void LWHotColorFunc (void *data, float r, float g, float b);

   typedef struct st_LWColorPickLocal {
      int             result;
      const char     *title;
      float           red, green, blue;
      void           *data;
      LWHotColorFunc *hotFunc;
   } LWColorPickLocal;
```

result
: The result of the request. This will be 1 if the user selected a color, 0 if the user cancelled the request, and a negative number to indicate an error.

title
: The title string. This is generally displayed near the top of the color dialog and tells the user the context of the color request.

red, **green**, **blue**
: The initial color. If the user selects a color, these fields will be modified to contain the selected color. The nominal range for RGB levels is 0.0 to 1.0, but they can be outside this range.

data
: A pointer to data that will be passed to your hot color callback. This can point to anything your callback requires, or NULL. The color picker ignores it.

hotFunc`( data, r, g, b )`
: A color callback you supply. The color picker calls this while the user is playing with any of its color selection mechanisms. This allows you to update your own display interactively as the user selects a color. (The "hot" part of the name refers to this dynamic interaction. This isn't an NTSC color gamut test.) The callback should execute quickly enough that it doesn't bog down the interactivity of the color picker with the user.

Example

This code fragment asks the user for a color.

```
   #include <lwserver.h>
   #include <lwhost.h>

   void colorcb( MyDisplayData *data, float r, float g, float b )
   {
      /* redraw my display with the current color */
      ...
```

```
}

LWColorActivateFunc *colorpick;
LWColorPickLocal clrloc;
MyDisplayData myhotdata;
int result;

colorpick = global( LWCOLORACTIVATEFUNC_GLOBAL, GFUSE_TRANSIENT );
if ( !colorpick ) goto NoColorPick;  /* global calls can fail */

clrloc.title   = "Widget Color";
clrloc.red     = current_red;
clrloc.green   = current_green;
clrloc.blue    = current_blue;
clrloc.data    = &myhotdata;
clrloc.hotFunc = colorcb;

result = colorpick( LWCOLORPICK_VERSION, &clrloc );
if ( result == AFUNC_OK && clrloc.result > 0 ) {
   current_red   = clrloc.red;
   current_green = clrloc.green;
   current_blue  = clrloc.blue;
   ...
```

Context Menu Services

Availability LightWave 7.0
Component Layout, Modeler
Header lwpanel.h

Context menu services are a set of functions for creating and displaying context menus over your panels. A context menu is a modal popup window containing a list of options. You typically display one of these when the user right-clicks or shift-clicks an item in your panel.

Global Call

```
   ContextMenuFuncs *cmenuf;
   cmenuf = global( LWCONTEXTMENU_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to a ContextMenuFuncs.

```
   typedef struct st_ContextMenuFuncs {
      LWContextMenuID (*cmenuCreate) (LWPanPopupDesc, void *userdata);
      int             (*cmenuDeploy) (LWContextMenuID, LWPanelID,
                                      int item);
      void            (*cmenuDestroy)(LWContextMenuID);
   } ContextMenuFuncs;
```

menu = **cmenuCreate**(popdesc, userdata)
Create a context menu. The `popdesc` structure, described below, contains your menu parameters and callbacks. The `userdata` is a pointer that you want your callbacks to receive.

selection = **cmenuDeploy**(menu, panel, item)
Display the context menu, typically in response to some user action. A context menu is always displayed in association with a particular `panel`. The `item` is the 0-based index of the menu item that should be selected when the menu is first displayed, and the return value is the index of the item selected by the user. Either of these can be -1 to indicate no selection.

cmenuDestroy(menu)
Free the menu and related resources allocated by `cmenuCreate`.

Popup Descriptor

The `cmenuCreate` function uses an LWPanPopupDesc structure to define the menu. This is the same structure used by Panels custom popup controls, but for that purpose, its details are conveniently hidden from you by the `CUSTPOPUP_CTL` macro.

```
typedef struct st_LWPanPopupDesc {
   LWType  type;
   int     width;
   int    (*countFn)(void *userdata);
   char * (*nameFn) (void *userdata, int item);
} LWPanPopupDesc;
```

type

Set this to `LWT_POPUP`.

width

The width of the menu in pixels.

nitems = **countFn**(userdata)

Your count callback, which returns the number of items in the menu. The `userdata` is whatever you passed as the second argument to `cmenuCreate`.

itemstring = **nameFn**(userdata, item)

Your item name callback, which returns the string that should be displayed for the item.

Example

The following code fragments create and display a simple context menu. First, we'll create a data structure for our menu and define the callbacks.

```
typedef struct st_MyMenuData {
   int    count;
   char **name;
} MyMenuData;

static char *itemname[] = {
   "New", "Load", "Save", "Copy", "Paste", NULL };

MyMenuData menudata = { 5, itemname };

int menuCount( MyMenuData *data )
{
   return data->count;
}

int menuName( MyMenuData *data, int index )
{
   if ( index >= 0 && index < data->count )
      return data->name[ index ];
   return NULL;
}
```

Don't forget to initialize the global.

```
#include <lwpanel.h>

ContextMenuFuncs *cmenuf;

cmenuf = global( LWCONTEXTMENU_GLOBAL, GFUSE_TRANSIENT );
if ( !cmenuf ) return AFUNC_BADGLOBAL;
```

Create the menu. Typically you'll do this when you're creating the associated panel and its controls.

```
LWContextMenuID menu;
LWPanPopupDesc desc;

desc.type    = LWT_POPUP;
desc.width   = 200;
desc.countFn = menuCount;
desc.nameFn  = menuName;

menu = cmenuf->cmenuCreate( &desc, menudata );
if ( !menu ) goto MenuFail;
```

Display the context menu in response to some user action.

```
int select, current;

select = cmenuf->cmenuDeploy( menu, panel, current );
if ( select != -1 ) {
   current = select;
   ...
```

When you're done with it, free the menu.

```
cmenuf->cmenuDestroy( menu );
```

Directory Info

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The function returned by the directory info global gives plug-ins read-only access to LightWave's internal directory list. This tells you where LightWave would look first for a given item. It can be used to set the initial path for a file request.

Global Call

```
   LWDirInfoFunc *dirinfo;
   dirinfo = global( LWDIRINFOFUNC_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWDirInfoFunc.

```
   typedef const char * LWDirInfoFunc (const char *dirtype);
```

The `dirtype` argument identifies which directory path should be returned by the directory info function. It can be any of the predefined file type strings. The paths returned may be relative rather than absolute. In most cases, relative paths will be relative to the content directory. Also, in some cases, the path may be NULL.

Example

This code fragment initializes a path string with the default directory for images.

```
   #include <lwserver.h>
   #include <lwhost.h>

   #define MAXFILESZ 260

   char *imgdir, path[ MAXFILESZ ] = "";
   LWDirInfoFunc *dirinfo;

   dirinfo = global( LWDIRINFOFUNC_GLOBAL, GFUSE_TRANSIENT );
   if ( dirinfo ) {
      imgdir = dirinfo( LWFTYPE_IMAGE );
      if ( imgdir )
         if ( strlen( imgdir ) < MAXFILESZ )
            strcpy( path, imgdir );
```

```
}
```

Dynamic Conversion

Availability LightWave 6.0
Component Modeler
Header lwdyna.h

This global returns a function that converts between DynaValues of different types. This is most often useful for converting between string and numeric values. DynaValues are used by the command system and by the requester API.

Global Call

```
   DynaConvertFunc *dynacvt;
   dynacvt = global( LWDYNACONVERTFUNC_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to a DynaConvertFunc.

```
   typedef int DynaConvertFunc (const DynaValue *from, DynaValue *to,
     const DynaStringHint *hint);
```

`from`
: The value to convert from.

`to`
: Receives the converted value of the `from` argument. Before calling the conversion function, set the `type` field of the `to` argument to the desired type for the conversion.

`hint`
: Specifies a mapping between `DY_STRING` types and certain kinds of numeric values. If the types involved in the conversion don't require a hint, this is ignored and may be NULL.

If the conversion succeeds, the function returns `DYERR_NONE`. Otherwise it returns an error code. Possible error codes include `DYERR_MEMORY`, `DYERR_BADTYPE` and `DYERR_INTERNAL`.

String Hints

If one of the DynaValues is a string and the other is a choice or a bitfield, the conversion uses the `hint` argument as a lookup table for translating between them. The structure used to pass hints looks like this.

```
typedef struct st_DynaStringHint {
   DyChoiceHint   *chc;
   DyBitfieldHint *bits;
} DynaStringHint;
```

There are two kinds of hints in a DynaStringHint structure, only one of which will be used for a given string conversion.

```
typedef struct st_DyChoiceHint {
   const char *item;
   int         value;
} DyChoiceHint;
```

The choice hint is an array of DyChoiceHint used when converting between `DY_STRING` and `DY_CHOICE` types. The `item`/`value` pairs indicate a mapping between choice values and strings. In other words, `hint->chc[i].item` will be converted to `hint->chc[i].value` and vice-versa. The array is terminated with a null `item` string.

```
typedef struct st_DyBitfieldHint {
   char code;
   int  bitval;
} DyBitfieldHint;
```

The bitfield hint is an array of DyBitfieldHint used when converting between an array of characters (`DY_STRING`) and a bitfield (the bits in a `DY_INTEGER`). Bitfields appear as arguments to certain CommandSequence commands. In the string representation, the presence of a given character corresponds to a set bit in a bitfield, and if that character isn't in the string, the bit is clear.

The DyBitfieldHint `code` field contains a character, and the `bitval` field is an integer with a bit pattern. When converting from a `DY_STRING`, if the `code` character (upper or lower case) is present in the string, the `bitval` bits will be bitwise-ORed into the conversion result. When converting from a bitfield, if the `bitval` pattern is present (`value & bitval == bitval`), the `code` character is appended to the string result. The hint list is terminated with a zero `bitval`.

Example

This code fragment converts between the string and the bitfield representations of a set of compass point flags.

```
#include <lwserver.h>
#include <lwdyna.h>

#define FLAG_NORTH (1<<0)
#define FLAG_SOUTH (1<<1)
#define FLAG_EAST  (1<<2)
#define FLAG_WEST  (1<<3)

DyBitfieldHint compass_hint[ 5 ] = {
   'n', FLAG_NORTH,
   's', FLAG_SOUTH,
   'e', FLAG_EAST,
   'w', FLAG_WEST,
   0, 0
};
DynaStringHint hint = { NULL, compass_hint };
DynaValue
   dystr = { DY_STRING },
   dyint = { DY_INTEGER };
DynaConvertFunc *dynacvt;
int result;

dynacvt = global( LWDYNACONVERTFUNC_GLOBAL, GFUSE_TRANSIENT );
if ( !dynacvt ) return AFUNC_BADGLOBAL;

dystr.str.buf = "ns";
result = dynacvt( &dystr, &dyint, &hint );

if ( result == DYERR_NONE ) {
   ...dyint.intv.value should contain FLAG_NORTH | FLAG_SOUTH...
```

Dynamic Monitor

Availability LightWave 6.0
Component Modeler
Header lwdyna.h

The Modeler monitor global returns functions for initializing and displaying a progress dialog in Modeler. See also the monitor global for Layout.

Global Call

```
   DynaMonitorFuncs *monf;
   monf = global( LWDYNAMONITORFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to a DynaMonitorFuncs.

```
   typedef struct st_DynaMonitorFuncs {
      LWMonitor * (*create)  (const char *, const char *);
      void        (*destroy) (LWMonitor *);
   } DynaMonitorFuncs;
```

`mon = create( title, caption )`

Create a monitor. This function returns an LWMonitor structure (described below) containing the actual progress display functions. The title text is ordinarily displayed at the top of the monitor dialog, and the caption text is displayed at the bottom. If `create` returns NULL, your plug-in should continue to run without reporting an error. Monitors are nice to have, but aren't essential.

`destroy( mon )`

Free a monitor obtained from `create`.

LWMonitor

The monitor structure returned by `create` is defined in the `lwmonitor.h` header file.

```
   typedef struct st_LWMonitor {
      void       *data;
      void     (*init) (void *, unsigned int);
      int      (*step) (void *, unsigned int);
      void     (*done) (void *);
   } LWMonitor;
```

data
: An opaque pointer to private data. Pass this as the first argument to all of the monitor functions.

init(data, total)
: Initialize the monitor. The `total` argument is the number of steps in the task to be monitored. It's up to you to decide what constitutes a step.

cancelled = **step**(data, increment)
: Advance the progress display by the fraction `total/increment`. When the sum of the steps reaches `total`, the progress display will indicate to the user that the task has finished. If `step` returns 1, the user has requested that the task be aborted.

done(data)
: Remove the progress display. This should always be called, even if the task doesn't finish.

Example

This code fragment demonstrates the use of a monitor. Macros in `lwmonitor.h` allow you to call the LWMonitor functions without worrying about whether the `create` call succeeded.

```
#include <lwserver.h>
#include <lwdyna.h>

DynaMonitorFuncs *monf;
LWMonitor *mon = NULL;

monf = global( LWDYNAMONITORFUNCS_GLOBAL, GFUSE_TRANSIENT );

if ( monf )
   mon = monf->create( "Hello", "Just fooling around" );

MON_INIT( mon, 100 );
for ( i = 0; i < 100; i += 2 ) {
   ...do something that takes a long time...
   if ( MON_INCR( mon, 2 )) break;
}
MON_DONE( mon );

...
if ( monf && mon )
   monf->destroy( mon );
```

Dynamic Request

Availability LightWave 6.0
Component Layout, Modeler
Header lwdyna.h

The dynamic request global returns functions that provide a simple user interface API. The requester mechanism predates both the Panels and XPanels systems and is available primarily for backward compatibility. Requesters in LightWave 6.0 and later are implemented as non-modal xpanels, so this global can be used as an alternative method of creating those.

Global Call

```
   DynaReqFuncs *reqf;
   reqf = global( LWDYNAREQFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to a DynaReqFuncs.

```
   typedef struct st_DynaReqFuncs {
      DynaRequestID (*create)   (const char *);
      int           (*addCtrl)  (DynaRequestID, const char *,
                       DyReqControlDesc *);
      DynaType      (*ctrlType) (DynaRequestID, int);
      int           (*valueSet) (DynaRequestID, int, DynaValue *);
      int           (*valueGet) (DynaRequestID, int, DynaValue *);
      int           (*post)     (DynaRequestID);
      void          (*destroy)  (DynaRequestID);
      LWXPanelID    (*xpanel)   (DynaRequestID);
   } DynaReqFuncs;
```

`req = create( title )`

Create a new dynamic request dialog. The title is displayed at the top of the dialog window. You can create as many dialogs as you like, but only one can be displayed at a time. The requester ID returned by `create` is passed as the first argument to the other requester functions.

`ctl = addCtrl( req, label, descriptor )`

Add a control to the dialog. Controls are stacked vertically in the requester window in the order in which they're added, with the first control on top. The return value is a control index used to identify the control in calls to `ctrlType`, `valueSet` and `valueGet`. The descriptor, explained below, contains the control type and other information necessary for its display.

`**ctrlType**( req, ctl )`
: Returns the type of a control.

`valueSet``( req, ctl, value )`

Initialize the value of a control. Values are expressed as DynaValues. This function will return an error code if the DynaValue type is incompatible with the type of the control, or if the control doesn't exist.

`valueGet``( req, ctl, value )`

Retrieve the value of a control. This function will return an error code if the DynaValue type is incompatible with the type of the control, or if the control doesn't exist.

`result = `**`post`**`( req )`

Display the requester. This function won't return until the user has dismissed the requester. It returns 1 if the user accepted the inputs and 0 if he or she cancelled them.

`destroy``( req )`

Free the requester and associated resources allocated by `create`.

`xpanel``( req )`

Returns the panel ID for the requester. See the XPanels discussion for more about this.

Control Types

These are the kinds of controls you can create.

DY_STRING

A single line of editable text. If you just want to display static text on the requester, use a `DY_TEXT` control.

DY_INTEGER
DY_FLOAT
DY_DISTANCE

Numeric edit fields. Distance controls display length units and handle unit conversions. Internally the value is a floating-point number of meters.

DY_VINT

DY_VFLOAT
DY_VDIST
Vector edit fields. A vector in this case is an array of three numbers.

DY_BOOLEAN
DY_CHOICE
Boolean controls are like checkboxes. Internally their states are stored as integers with values of either 0 or 1. Choice controls are like radio buttons, an array of mutually exclusive booleans. The internal representation is an integer containing a zero-based index into the choice list.

DY_SURFACE
Lets the user choose one of the surfaces currently in Modeler's internal surface list. The underlying data for a surface control is a string containing the surface name.

DY_FONT
Lets the user choose one of the fonts currently in Modeler's internal font list. The underlying data for a font control is an integer index into the font list.

DY_TEXT
Static (read-only) text. Use this to write things on the requester. If you need a text edit field, use a `DY_STRING` control instead.

DY_LAYERS
Lets the users select one or more layers. These are represented internally by set bits in an integer.

Control Descriptor

The `addCtrl` function is passed a descriptor to tell it what kind of control to create. For most controls, the descriptor is just a type code corresponding to the DynaType of the variable it represents (one of the types in the list above), but for string, choice and static text controls, some additional information is required to create the control.

String

```
typedef struct st_DyReqStringDesc {
   DynaType         type;
   int              width;
} DyReqStringDesc;
```

`width`
: The displayed width of the edit field, in characters.

Choice

```
typedef struct st_DyReqChoiceDesc {
   DynaType         type;
   const char     **items;
   int              vertical;
} DyReqChoiceDesc;
```

`items`
: An array of strings. Each string is the label for a choice. The `valueGet` call will return an index into this array to indicate the selected item.

`vertical`
: A non-zero value will cause the choices to be arranged vertically on the requester. Otherwise, they're arranged horizontally.

Static Text

```
typedef struct st_DyReqTextDesc {
   DynaType         type;
   const char     **text;
} DyReqTextDesc;
```

`text`
: An array of strings. Each string is displayed on its own line in the requester.

The control descriptor is a union that collects the type code and this extra information into a single structure.

```
typedef union un_DyReqControlDesc {
   DynaType         type;
   DyReqStringDesc  string;
   DyReqChoiceDesc  choice;
   DyReqTextDesc    text;
} DyReqControlDesc;
```

Example

This code fragment creates a requester asking for personal information. It relies on functions called `reqAdd`, `reqSet` and `reqGet` to hide some of the details. The source for these functions follows.

```
   #include <lwserver.h>
   #include <lwmodeler.h>

   DynaRequestID req;
   int ctl[ 5 ], ok;

   char name[ 40 ] = "(Your name here)";
   int age = 30, gender = 0;
   double height = 1.8;
   double measure[ 3 ] = { 0.9144, 0.6096, 0.9144 };
   char *glabel[] = { "Female", "Male", NULL };

   reqf = global( LWDYNAREQFUNCS_GLOBAL, GFUSE_TRANSIENT );
   if ( !reqf ) return AFUNC_BADGLOBAL;

   req = reqf->create( "All About Me" );
   if ( !req ) goto ErrorNoReq;

   ctl[ 0 ] = reqAdd( req, DY_STRING, "Name", NULL, 20 );
   ctl[ 1 ] = reqAdd( req, DY_INTEGER, "Age", NULL, 0 );
   ctl[ 2 ] = reqAdd( req, DY_CHOICE, "Gender", glabel, 1 );
   ctl[ 3 ] = reqAdd( req, DY_FLOAT, "Height", NULL, 0 );
   ctl[ 4 ] = reqAdd( req, DY_VDIST, "Measurements", NULL, 0 );

   reqSet( req, ctl[ 0 ], name );
   reqSet( req, ctl[ 1 ], &age );
   reqSet( req, ctl[ 2 ], &gender );
   reqSet( req, ctl[ 3 ], &height );
   reqSet( req, ctl[ 4 ], measure );

   ok = reqf->post( req );

   if ( ok ) {
      reqGet( req, ctl[ 0 ], name, sizeof( name ));
      reqGet( req, ctl[ 1 ], &age, 0 );
      reqGet( req, ctl[ 2 ], &height, 0 );
      reqGet( req, ctl[ 3 ], &gender, 0 );
      reqGet( req, ctl[ 4 ], measure, 0 );
   }

   reqf->destroy( req );
```

The `reqAdd` function creates a requester control. Most control types can be created with only the DynaType and label, but a few require additional information. The `text` argument is an array of strings used to create `DY_CHOICE` and `DY_TEXT` controls. The `extra` argument is the string width in characters for `DY_STRING` controls and the vertical flag for `DY_CHOICE` controls.

```
int reqAdd( DynaRequestID req, DynaType type, char *label,
   char **text, int extra )
{
   DyReqControlDesc desc;

   desc.type = type;

   switch ( type ) {
      case DY_STRING:
         desc.string.width = extra;
         break;
      case DY_CHOICE:
         desc.choice.items = text;
         desc.choice.vertical = extra;
         break;
      case DY_TEXT:
         desc.text.text = text;
         break;
      default:
         break;
   }

   return reqf->addCtrl( req, title, &desc );
}
```

The `reqSet` and `reqGet` functions can set and get the value of any control. The `val` argument points to a variable of an appropriate type for the control.

```
int reqSet( DynaRequestID req, int ctl, void *val )
{
   DynaValue dv;
   int *ivec;
   double *fvec;

   dv.type = reqf->ctrlType( req, ctl );
   switch ( dv.type ) {
      case DY_STRING:
      case DY_SURFACE:
         dv.str.buf = ( char * ) val;
         dv.str.bufLen = 0;
         break;
      case DY_INTEGER:
      case DY_BOOLEAN:
      case DY_FONT:
      case DY_LAYERS:
      case DY_CHOICE:
         dv.intv.value = dv.intv.defVal = *(( int * ) val );
         break;
      case DY_FLOAT:
      case DY_DISTANCE:
         dv.flt.value = dv.flt.defVal = *(( double * ) val );
         break;
      case DY_VINT:
         ivec = ( int * ) val;
         dv.ivec.val[ 0 ] = ivec[ 0 ];
         dv.ivec.val[ 1 ] = ivec[ 1 ];
         dv.ivec.val[ 2 ] = ivec[ 2 ];
         break;
      case DY_VFLOAT:
      case DY_VDIST:
         fvec = ( double * ) val;
         dv.fvec.val[ 0 ] = fvec[ 0 ];
         dv.fvec.val[ 1 ] = fvec[ 1 ];
         dv.fvec.val[ 2 ] = fvec[ 2 ];
         break;
      default:
         return 0;
   }

   return reqf->valueSet( req, ctl, &dv );
}


int reqGet( DynaRequestID req, int ctl, void *val, int len )
```

```
{
   DynaValue dv;
   int *ivec, result;
   double *fvec;

   dv.type = reqf->ctrlType( req, ctl );

   if ( dv.type == DY_STRING || dv.type == DY_SURFACE ) {
      dv.str.bufLen = len;
   }

   result = reqf->valueGet( req, ctl, &dv );

   switch ( dv.type ) {
      case DY_STRING:
      case DY_SURFACE:
         strcpy( val, dv.str.buf );
         break;
      case DY_INTEGER:
      case DY_BOOLEAN:
      case DY_FONT:
      case DY_LAYERS:
      case DY_CHOICE:
         *(( int * ) val ) = dv.intv.value;
         break;
      case DY_FLOAT:
      case DY_DISTANCE:
         *(( double * ) val ) = dv.flt.value;
         break;
      case DY_VINT:
         ivec = ( int * ) val;
         ivec[ 0 ] = dv.ivec.val[ 0 ];
         ivec[ 1 ] = dv.ivec.val[ 1 ];
         ivec[ 2 ] = dv.ivec.val[ 2 ];
         break;
      case DY_VFLOAT:
      case DY_VDIST:
         fvec = ( double * ) val;
         fvec[ 0 ] = dv.fvec.val[ 0 ];
         fvec[ 1 ] = dv.fvec.val[ 1 ];
         fvec[ 2 ] = dv.fvec.val[ 2 ];
         break;
      default:
         break;
   }

   return result;
}
```

File I/O

This page describes the mechanism LightWave provides to move plug-in data into and out of files. The file I/O functions are used by handlers in their `load` and `save` callbacks to store and retrieve instance data in scene and object files. These functions can also be used by any plug-in class to read and write files accessed through the File I/O global.

Loading

Data is loaded from files using the functions in an LWLoadState. The `lwio.h` header file also defines macros for most of these functions. Both the functions and the corresponding macros are listed in the definitions.

```
typedef struct st_LWLoadState {
   int   ioMode;
   void *readData;
   int  (*read)    (void *data, char  *, int len);
   int  (*readI1)  (void *data, char  *, int n);
   int  (*readI2)  (void *data, short *, int n);
   int  (*readI4)  (void *data, long  *, int n);
   int  (*readU1)  (void *data, unsigned char  *, int n);
   int  (*readU2)  (void *data, unsigned short *, int n);
   int  (*readU4)  (void *data, unsigned long  *, int n);
   int  (*readFP)  (void *data, float *, int n);
   int  (*readStr) (void *data, char  *, int max);
   LWID (*findBlk) (void *data, const LWBlockIdent *);
   void (*endBlk)  (void *data);
   int  (*depth)   (void *data);
} LWLoadState;
```

`ioMode`

Indicates whether the file contents will be interpreted as binary (`LWIO_BINARY`) or text (`LWIO_ASCII`). Handler plug-ins that receive the LWLoadState in their instance load function can usually infer from the `ioMode` whether they're reading their data from a scene file or an object file. If the LWLoadState is created by the File I/O global's `openLoad` function, the `ioMode` matches the one passed to `openLoad`. This global supports a third `ioMode`, `LWIO_BINARY_IFF`.

In ASCII mode, all of the read functions are line-buffered, meaning that they won't wrap around to the next line when reading an array of values. If you ask for six numbers and the current line contains only five, the read functions will return five values rather than trying to get the sixth from the following line.

`readData`

An opaque pointer to data used by the LWLoadState functions. Pass this as the first argument to these functions.

`rn = read( readData, buf, n )`

Read raw bytes. In binary mode, `n` bytes are read directly from the file. In ASCII mode, up to `n` bytes of the current line are read from the file, possibly leaving more bytes to be read later (the file pointer isn't moved to the next line until all of the current line is read). The return value is the number of bytes actually read (which may be zero in ASCII mode if the current line is empty), or -1 for end of data.

```
rn = readI1( readData, bytebuf, n )
rn = LWLOAD_I1( ls, bytebuf, n )
```

Read an array of bytes. These are interpreted as numbers rather than text characters. Returns the number of bytes read.

```
rn = readI2( readData, shortbuf, n )
rn = LWLOAD_I2( ls, shortbuf, n )
```

Read an array of two-byte integers. Returns the number of short integers read.

```
rn = readI4( readData, longbuf, n )
rn = LWLOAD_I4( ls, longbuf, n )
```

Read an array of four-byte integers. Returns the number of integers read.

```
rn = readU1( readData, ubytebuf, n )
rn = LWLOAD_U1( ls, ubytebuf, n )
```

Read an array of unsigned bytes. These are interpreted as numbers rather than text characters. Returns the number of bytes read.

```
rn = readU2( readData, ushortbuf, n )
rn = LWLOAD_U2( ls, ushortbuf, n )
```

Read an array of unsigned two-byte integers. Returns the number of short integers read.

```
rn = readU4( readData, ulongbuf, n )
rn = LWLOAD_U4( ls, ulongbuf, n )
```

Read an array of unsigned four-byte integers. Returns the number of integers read.

```
rn = readFP( readData, floatbuf, n )
rn = LWLOAD_FP( ls, floatbuf, n )
```

Read an array of floating point numbers. Returns the number of floats read.

`len = **readStr**( readData, strbuf, maxn )`
`len = **LWLOAD_STR**( ls, strbuf, maxn )`

Read a string. Double quotes used to delimit the string in a text file are removed. Returns the length of the string.

`id = **findBlk**( readData, idlist )`
`id = **LWLOAD_FIND**( ls, idlist )`

Read ahead, looking for the next block. The ID list is a 0-terminated array of LWBlockIdent structures, and the function returns when it finds any one of the blocks in the list. In binary mode, a block is identified by a 4-byte integer constructed using the `LWID_` macro defined in the `lwtypes.h` header file. In ASCII mode, the block ID is a string token. Returns 0 if no blocks in the list were found.

`**endBlk**( readData )`
`**LWLOAD_END**( ls )`

Move the file pointer to the end of the current block. Call this when you've finished reading a block.

`d = **depth**( readData)`
`d = **LWLOAD_DEPTH**( ls )`

Returns the current block nesting level, where 0 means we've entered no blocks.

Saving

Data is saved to files using the functions in an LWSaveState.

```
   typedef struct st_LWSaveState {
      int   ioMode;
      void *writeData;
      void (*write)    (void *data, const char  *, int len);
      void (*writeI1)  (void *data, const char  *, int n);
      void (*writeI2)  (void *data, const short *, int n);
      void (*writeI4)  (void *data, const long  *, int n);
      void (*writeU1)  (void *data, const unsigned char  *, int n);
      void (*writeU2)  (void *data, const unsigned short *, int n);
      void (*writeU4)  (void *data, const unsigned long  *, int n);
      void (*writeFP)  (void *data, const float *, int n);
      void (*writeStr) (void *data, const char  *);
      void (*beginBlk) (void *data, const LWBlockIdent *, int leaf);
      void (*endBlk)   (void *data);
      int  (*depth)    (void *data);
   } LWSaveState;
```

`ioMode`

Indicates whether the file contents will be interpreted as binary (`LWIO_BINARY`) or text (`LWIO_ASCII`). Handler plug-ins that receive the LWSaveState in their instance save function can usually infer from the `ioMode` whether they're writing their data to a scene file or an object file. If the LWSaveState is created by the File I/O global's `openSave` function, the `ioMode` matches the one passed to `openSave`. This global supports a third `ioMode`, `LWIO_BINARY_IFF`.

In ASCII mode, the write functions are line-buffered, meaning that each call to a write function results in a single newline-terminated line in the file.

`writeData`

An opaque pointer to data used by the LWSaveState functions. Pass this as the first argument to these functions.

`write``( writeData, buf, len )`

Write raw bytes. In binary mode, `len` bytes are written directly to the file. In ASCII mode, the `buf` argument is assumed to be a null-terminated string and the length is computed from that. This string is written with a newline at the end.

`writeI1``( writeData, bytebuf, n )`
`LWSAVE_I1``( ss, bytebuf, n )`

Write an array of bytes. The values are treated as numbers rather than text characters.

`writeI2``( writeData, shortbuf, n )`
`LWSAVE_I2``( ss, shortbuf, n )`

Write an array of two-byte integers. In ASCII mode, these `n` numbers are all written to a single line. A newline is written after the numbers unless you're currently inside a leaf block.

`writeI4``( writeData, longbuf, n )`
`LWSAVE_I4``( ss, longbuf, n )`

Write an array of four-byte integers.

`writeU1``( writeData, ubytebuf, n )`
`LWSAVE_U1``( ss, ubytebuf, n )`

Write an array of unsigned bytes. The values are treated as numbers rather than text characters. In text files, each byte is written as a pair of hexadecimal digits.

`writeU2``( writeData, ushortbuf, n )`
`LWSAVE_U2``( ss, ushortbuf, n )`

Write an array of unsigned two-byte integers. In text files, the values are written in hex.

`writeU4``( writeData, ulongbuf, n )`
`LWSAVE_U4``( ss, ulongbuf, n )`

Write an array of unsigned four-byte integers. In text files, the values are written in hex.

writeFP(writeData, floatbuf, n)
LWSAVE_FP(ss, floatbuf, n)

Write an array of floats.

writeStr(writeData, strbuf)
LWSAVE_STR(ss, strbuf)

Write a string. In ASCII mode, the string may be contained in double quote marks (which will be removed when the string is later read by the LWLoadState `readStr` function).

beginBlk(writeData, blockid, leaf)
LWSAVE_BEGIN(ss, blockid, leaf)

Create a new block. `blockid` is an LWBlockIdent that will be used to label the block. The `leaf` flag is true if this block will not contain sub-blocks.

endBlk(writeData)
LWSAVE_END(ss)

End the current block.

d = **depth**(writeData)
d = **LWSAVE_DEPTH**(ss)

Return the current block nesting level, where zero means you've entered no blocks.

Block Identifiers

The LWBlockIdent structure is used to label blocks.

```
typedef struct st_LWBlockIdent {
   LWID        id;
   const char *token;
} LWBlockIdent;
```

id

A four-byte code usually built by the `LWID_` macro defined in lwtypes.h. Used when writing to binary files. This is also the value returned by findBlk for both binary and ASCII files, which makes it useful as the descriminator in a case statement.

token

A plain text label used when writing to ASCII files. This string should contain no spaces.

When creating custom files for your own use, you may use any ID and label you like. Their only purpose is to identify the data that follows them when you read the file back in.

Example

Most of the file I/O functions are straightforward, so this example code concentrates on the use of the block functions to write and read block-structured data.

LightWave scene files use blocks to create keyword-value pairs and to delimit keyframe data. Blocks also appear as the subchunks in each SURF chunk of an object file. Block structure makes the data self-documenting and more human-readable. It also makes your file format extensible without sacrificing backward compatibility. Older readers will automatically skip blocks they don't recognize and can find blocks even if they've been written in a different order.

We'll create a data structure well suited to blocky storage. This structure is borrowed from an astronomy application, where it describes the circumstances of an observer.

```
#include <lwserver.h>
#include <lwio.h>

typedef struct {
   float    timezone;
   char     tzname[ 4 ];
   int      ltim[ 6 ];          /* yr mon day hr min sec */
   float    location[ 2 ];      /* lat lon */
   int      horizon_type;
   float    temperature;
   float    pressure;
   float    elevation;
   float    epoch;
} Observer;
```

We need labels for each of the blocks. These will be used for both saving and loading. The ID arrays are divided into root blocks in the first and subblocks of the horizon block in the second, which is what we'll need when we read this data back in. The `#define`s may seem like an extra step now, but they'll come in handy later.

```
#define ID_TZON  LWID_( 'T','Z','O','N' )
#define ID_TZNM  LWID_( 'T','Z','N','M' )
#define ID_LTIM  LWID_( 'L','T','I','M' )
#define ID_LOCA  LWID_( 'L','O','C','A' )
#define ID_EPOC  LWID_( 'E','P','O','C' )
```

```
#define ID_HRZN  LWID_( 'H','R','Z','N' )
#define ID_TYPE  LWID_( 'T','Y','P','E' )
#define ID_TEMP  LWID_( 'T','E','M','P' )
#define ID_PRES  LWID_( 'P','R','E','S' )
#define ID_ELEV  LWID_( 'E','L','E','V' )

static LWBlockIdent idroot[] = {
   ID_TZON, "TimeZone",
   ID_TZNM, "TimeZoneName",
   ID_LTIM, "LocalTime",
   ID_LOCA, "Location",
   ID_EPOC, "Epoch",
   ID_HRZN, "Horizon",
   0
};

static LWBlockIdent idhrzn[] = {
   ID_TYPE, "Type",
   ID_TEMP, "Temperature",
   ID_PRES, "Pressure",
   ID_ELEV, "Elevation",
   0
};
```

This is the save function. Note that it doesn't care whether the LWSaveState's `ioMode` is `LWIO_ASCII` or `LWIO_BINARY`. It also doesn't care whether the LWSaveState came from a handler's `save` callback or from the file I/O global's `openSave` function.

```
int write_obs( LWSaveState *save, Observer *obs )
{
   LWSAVE_BEGIN( save, &idroot[ 0 ], 1 );
      LWSAVE_FP( save, &obs->timezone, 1 );
   LWSAVE_END( save );

   LWSAVE_BEGIN( save, &idroot[ 1 ], 1 );
      LWSAVE_STR( save, obs->tzname );
   LWSAVE_END( save );

   LWSAVE_BEGIN( save, &idroot[ 2 ], 1 );
      LWSAVE_I4( save, obs->ltim, 6 );
   LWSAVE_END( save );

   LWSAVE_BEGIN( save, &idroot[ 3 ], 1 );
      LWSAVE_FP( save, obs->location, 2 );
   LWSAVE_END( save );

   LWSAVE_BEGIN( save, &idroot[ 4 ], 1 );
      LWSAVE_FP( save, &obs->epoch, 1 );
   LWSAVE_END( save );

   LWSAVE_BEGIN( save, &idroot[ 5 ], 0 );

      LWSAVE_BEGIN( save, &idhrzn[ 0 ], 1 );
         LWSAVE_I4( save, &obs->horizon_type, 1 );
      LWSAVE_END( save );

      LWSAVE_BEGIN( save, &idhrzn[ 1 ], 1 );
         LWSAVE_FP( save, &obs->temperature, 1 );
      LWSAVE_END( save );
      LWSAVE_BEGIN( save, &idhrzn[ 2 ], 1 );
         LWSAVE_FP( save, &obs->pressure, 1 );
      LWSAVE_END( save );

      LWSAVE_BEGIN( save, &idhrzn[ 3 ], 1 );
         LWSAVE_FP(  save, &obs->elevation, 1 );
      LWSAVE_END( save );

   LWSAVE_END( save );

   return 1;
}
```

If the `ioMode` is `LWIO_ASCII`, the output of the `write_obs` function looks like this.

```
TimeZone 4
TimeZoneName "EDT"
LocalTime 2000 4 24 2 5 30
Location 37.75 -122.55
Epoch 2000
{ Horizon
  Type 1
  Temperature 40
  Pressure 30
  Elevation 100
}
```

Each leaf block is a single line containing a keyword (the LWBlockIdent token) and a list of values. Non-leaf blocks are delimited by curly brackets and indented to show the block nesting level.

A hex dump of the same data written to a binary file would look like the following. Each block begins with the 4-byte ID and a 2-byte size field. All of the numbers are in big-endian (Internet, Motorola) byte order.

```
54 5A 4F 4E 00 04   TZON 4
40 80 00 00           4.0
54 5A 4E 4D 00 04   TZNM 4
45 44 54 00           "EDT"
4C 54 49 4D 00 18   LTIM 24
00 00 07 D0           2000
00 00 00 04           4
00 00 00 18           24
00 00 00 02           2
00 00 00 05           5
00 00 00 1E           30
4C 4F 43 41 00 08   LOCA 8
42 17 00 00           37.75
C2 F5 19 9A           -122.55
45 50 4F 43 00 04   EPOC 4
44 FA 00 00           2000.0
48 52 5A 4E 00 28   HRZN 40
54 59 50 45 00 04     TYPE 4
00 00 00 01             1
54 45 4D 50 00 04     TEMP 4
42 20 00 00             40.0
```

```
50 52 45 53 00 04     PRES 4
41 F0 00 00             30.0
45 4C 45 56 00 04     ELEV 4
42 C8 00 00             100.0
```

The function to read this data just searches for blocks in a loop and switches to load each one. The outer `while` loop reads root blocks, and the inner loop reads horizon blocks when the `HRZN` root block is found.

```
int read_obs( LWLoadState *load, Observer *obs )
{
   LWID id;

   while ( id = LWLOAD_FIND( load, idroot )) {
      switch ( id ) {
         case ID_TZON:
            LWLOAD_FP( load, &obs->timezone, 1 );
            break;
         case ID_TZNM:
            LWLOAD_STR( load, obs->tzname, 4 );
            break;
         case ID_LTIM:
            LWLOAD_I4( load, obs->ltim, 6 );
            break;
         case ID_LOCA:
            LWLOAD_FP( load, obs->location, 2 );
            break;
         case ID_EPOC:
            LWLOAD_FP( load, &obs->epoch, 1 );
            break;
         case ID_HRZN:
            while ( id = LWLOAD_FIND( load, idhrzn )) {
               switch ( id ) {
                  case ID_TYPE:
                     LWLOAD_I4( load, &obs->horizon_type, 1 );
                     break;
                  case ID_TEMP:
                     LWLOAD_FP( load, &obs->temperature, 1 );
                     break;
                  case ID_PRES:
                     LWLOAD_FP( load, &obs->pressure, 1 );
                     break;
                  case ID_ELEV:
                     LWLOAD_FP( load, &obs->elevation, 1 );
                     break;
               }
               LWLOAD_END( load );
            }
            break;
      }
      LWLOAD_END( load );
   }

   return 1;
}
```

File Request

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The file request global returns a function that prompts the user for a file selection. The request displays the file dialog currently installed in LightWave. This may be the default system dialog or a custom file dialog plug-in. See the File Request 2 global for a newer interface to the file dialog mechanism.

Global Call

```
   LWFileReqFunc *filereq;
   filereq = global( LWFILEREQFUNC_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWFileReqFunc.

```
   typedef int LWFileReqFunc (const char *hail, char *name, char *path,
      char *fullName, int buflen);
```

hail

The title string. This is generally displayed near the top of the file dialog and tells the user what kind of file is being requested.

name

The initial file name, not including the path. This can be empty, or it can contain a default name. It can also contain file type patterns that on most systems will filter the names displayed in the dialog. Pattern strings for certain types of files can be obtained from the File Type Pattern global. If you construct your own pattern strings, remember that these are platform-specific and may also be locale-specific.

Some systems display different dialogs for loading and saving. If the first character of the name is '<', a load dialog will be displayed, and if it's '>', a save dialog will be displayed. These initial characters won't appear as part of the initial name or file type pattern.

If the user selects a file, the initial name is replaced with the name (not including the path) of the selected file.

`path`

The initial path. Default paths for certain file types can be obtained from the Directory Info global. If you construct your own path string, remember that path lexics depend on the platform. If the user selects a file, the initial path is replaced with the path of the selected file.

`fullName`

The file request returns the selected file name, including the path, in this string. The initial contents are ignored.

`bufLen`

The size in bytes of the `name`, `path` and `fullName` strings. Note that all of them must be at least this size and must be large enough to hold the largest file name string you expect to process (a minimum of 256 bytes is recommended).

Example

This code fragment asks the user for the name of a file to save.

```
#include <lwhost.h>
#define MAXFILESZ 260

static char
   node[ MAXFILESZ ] = "",
   path[ MAXFILESZ ] = "",
   name[ MAXFILESZ ] = "";

LWFileReqFunc *filereq;
LWMessageFuncs *message;
int result;

filereq = global( LWFILEREQFUNC_GLOBAL, GFUSE_TRANSIENT );
message = global( LWMESSAGEFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( !filereq || !message ) return AFUNC_BADGLOBAL;

result = filereq( "Save Widget", node, path, name, MAXFILESZ );
if ( result ) {
   save_widget( widget, name );
   message->info( "The widget has been saved to", node );
}
else
   /* the user cancelled the file dialog */
   ...
```

File Request 2

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The file request 2 global returns a function that prompts the user for a file selection. The request displays the file dialog currently installed in LightWave. This may be the default system dialog or a custom file dialog plug-in. See the File Request global for an alternative interface to the file dialog mechanism.

The primary advantage of this file request global over the original File Request is a smarter and more flexible interface to the file dialog. The dialog is initialized by filling out a structure, rather than through a limited number of function arguments.

Note that in contrast to the original, this global allows plug-ins to call the file request activation function directly. Plug-ins calling this global act as the host side of the FileRequester plug-in class.

Global Call

```
   LWFileActivateFunc *filereq;
   filereq = global( LWFILEACTIVATEFUNC_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWFileActivateFunc.

```
   typedef int LWFileActivateFunc (int version, LWFileReqLocal *);
```

The return value of this function can be any of the values defined for the return values of activation functions. Any value other than `AFUNC_OK` must be handled as an error.

The version is passed as the `version` argument to the file request plug-in's activation function. This should be set to the value defined by the `LWFILEREQ_VERSION` symbol in `lwdialog.h`. File request plug-ins with a different activation version will return `AFUNC_BADVERSION`.

The second argument to this function is a pointer to a structure that is passed as the `local` argument to the file request plug-in's activation function.

The Local Structure

The file request function passes an LWFileReqLocal as the local argument to the activation function of the file request plug-in.

```
typedef struct st_LWFileReqLocal {
   int         reqType;
   int         result;
   const char *title;
   const char *fileType;
   char       *path;
   char       *baseName;
   char       *fullName;
   int         bufLen;
   int        (*pickName) (void);
} LWFileReqLocal;
```

`reqType`
Indicates the type of file request. Possible values are

```
FREQ_LOAD
FREQ_SAVE
FREQ_DIRECTORY
FREQ_MULTILOAD
```

`result`
The result of the request. This will be 1 if the user selected a file, 0 if the user cancelled the request, and a negative number to indicate an error.

`title`
The title string. This is generally displayed near the top of the file dialog and tells the user what kind of file is being requested.

`fileType`
A file type string used to filter the filenames displayed in the dialog. This is one of the type names listed in the document for the File Type Pattern global, rather than a literal filter.

`path`
The initial path. Default paths for certain file types can be obtained from the Directory Info global. If you construct your own path string, remember that path lexics depend on the platform. If the user selects a file, the initial path is replaced with the path of the selected file.

`baseName`
The initial file name, not including the path. This can be empty, or it

can contain a default name. The initial name can also contain wildcards that may be used to filter the names displayed in the dialog. If the user selects a file, the initial name is replaced with the name (not including the path) of the selected file.

`fullName`
: The file request returns the selected file name, including the path, in this string. The initial contents are ignored.

`bufLen`
: The size in bytes of the `baseName`, `path` and `fullName` strings. Note that all of them must be the same size, and should be large enough to hold the largest file name you expect to process (a minimum of 256 bytes is recommended).

`pickName``()`
: A callback function you provide when making `MULTILOAD` requests. This function will be called for each selected file. It returns 0 to continue and any non-zero value to stop processing the files in a multiple file selection. Each time your `pickName` is called, your LWFileReqLocal structure will contain the next name in the list of names selected by the user. Your LWFileReqLocal therefore needs to be declared in a place where it will be visible to your `pickName` function.

Example

This code fragment asks the user for the name of an image file to save.

```
#include <lwserver.h>
#include <lwhost.h>

#define MAXFILESZ 260

static char
   node[ MAXFILESZ ] = "",
   path[ MAXFILESZ ] = "",
   name[ MAXFILESZ ] = "";
static LWFileReqLocal frloc;

LWFileActivateFunc *filereq;
int result;

filereq = global( LWFILEACTIVATEFUNC_GLOBAL, GFUSE_TRANSIENT );
if ( !filereq ) goto NoFileReq;  /* global calls can fail */

frloc.reqType  = FREQ_SAVE;
frloc.title    = "Save Image";
frloc.bufLen   = MAXFILESZ;
frloc.pickName = NULL;
frloc.fileType = "Images";
frloc.path     = path;
frloc.baseName = node;
frloc.fullName = name;

strcpy( frloc.path, "MyImages" );     /* a relative path */
strcpy( frloc.baseName, "foo" );      /* a default name  */

result = filereq( LWFILEREQ_VERSION, &frloc );
if ( result == AFUNC_OK && frloc.result > 0 ) {
   save_image( myimage, frloc.fullName );
   ...
```

File Type Pattern

Availability LightWave 6.0 **Component** Layout, Modeler
Header lwhost.h

This global returns a function that allows plug-ins to retrieve file name pattern strings. These can be used to translate the string in the `fileType` passed to file request plug-ins into a literal filter string.

Global Call

```
LWFileTypeFunc *filetypes;
filetypes = global( LWFILETYPEFUNC_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWFileTypeFunc.

```
typedef const char * LWFileTypeFunc (const char *type);
```

The `type` argument identifies the kind of file you want a pattern string for. It can be any of the following.

LWFTYPE_ANIMATION	LWFTYPE_OBJECT	LWFTYPE_SCENE
LWFTYPE_IMAGE	LWFTYPE_PLUGIN	LWFTYPE_SETTING
LWFTYPE_ENVELOPE	LWFTYPE_PREVIEW	LWFTYPE_SURFACE
LWFTYPE_MOTION	LWFTYPE_PSFONT	LWFTYPE_CONTENT

The association between a type string and a literal filter is usually stored in the configuration file. Pattern strings are platform-specific. Under Windows, the string is a list of wildcard extensions separated by semicolons, e.g. `*.iff;*.tga` for images. On the Mac, the string is a list of 4-character file types, also separated by semicolons. The Unix string uses a regular expression.

Example

This code fragment obtains the pattern string for image files.

```
#include <lwserver.h>
#include <lwhost.h>

char *imgpat;
LWFileTypeFunc *filetypes;

filetypes = global( LWFILETYPEFUNC_GLOBAL, GFUSE_TRANSIENT );
```

```
if ( filetypes )
   imgpat = filetypes( LWFTYPE_IMAGE );
...
```

Fog Info

Availability LightWave 6.0
Component Layout
Header lwrender.h

The fog info global returns information about the fog settings for the scene. The parameters are read-only, but you can set them using commands.

Global Call

```
LWFogInfo *foginfo;
foginfo = global( LWFOGINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWFogInfo.

```
typedef struct st_LWFogInfo {
   int     type;
   int     flags;
   double (*minDist) (LWTime);
   double (*maxDist) (LWTime);
   double (*minAmt)  (LWTime);
   double (*maxAmt)  (LWTime);
   void   (*color)   (LWTime, double col[3]);
} LWFogInfo;
```

`type`
The fog type. Generally this identifies the falloff function used to interpolate the fog level at distances between `minDist` and `maxDist`. Possible types are

```
LWFOG_NONE
LWFOG_LINEAR
LWFOG_NONLINEAR1
LWFOG_NONLINEAR2
```

`flags`
Fog-related flags. Currently the only flag defined for this field is `LWFOGF_BACKGROUND`, which indicates that fog will affect the backdrop.

`amount = minDist( time )`
Returns the distance from the viewer (typically the camera) at which the fog effect is at a minimum.

`amount = maxDist( time )`
Returns the distance at which the fog effect reaches its maximum.

`amount =` **`minAmt`**`( time )`

Returns the minimum amount of fog (the amount at the minimum distance). Fog amounts range from 0.0 to 1.0.

`amount =` **`maxAmt`**`( time )`

Returns the maximum amount of fog.

`color``( time, rgb )`

The color of the fog.

Example

This code fragment uses Fog Info to determine whether a point at distance `d` is in fog at time `t`.

```
#include <lwserver.h>
#include <lwrender.h>

LWFogInfo *foginfo;

foginfo = global( LWFOGINFO_GLOBAL, GFUSE_TRANSIENT );
if ( foginfo ) {
   if ( foginfo->type != LWFOG_NONE ) {
      if ( d >= foginfo->minDist( t )) {
         ...d is in fog
```

Font List

Availability LightWave 6.0
Component Modeler
Header lwmodeler.h

The font list global provides a set of functions for managing Modeler's internal list of fonts.

Global Call

```
   LWFontListFuncs *fontf;
   fontf = global( LWFONTLISTFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWFontListFuncs.

```
   typedef struct st_LWFontListFuncs {
      int          (*count) (void);
      int          (*index) (const char *name);
      const char * (*name)  (int index);
      int          (*load)  (const char *filename);
      void         (*clear) (int index);
   } LWFontListFuncs;
```

`numfonts = count()`

Returns the number of fonts in the list.

`fontindex = index( fontname )`

Returns the list index for a named font, or -1 if a font of that name isn't in the list.

`fontname = name( fontindex )`

Returns the name of a font given its list index, or NULL if the index is less than 0 or greater than `numfonts - 1`.

`fontindex = load( filename )`

Adds the Postscript Type 1 font to the list and returns its list index, or -1 if the font couldn't be loaded. Since the font list is kept in an order that may differ from the order in which fonts are added, a font added by `load` might be inserted anywhere in the list, not just at the end, and fonts with higher indexes will be shifted downward (incrementing their indexes).

Note that only Postscript fonts can be added to the list in this way.

TrueType fonts are made available to Modeler indirectly, through operating system calls, rather than directly by reading their file contents.

clear(fontindex)

Removes the font at the given index from the list. Fonts at higher indexes will be shifted up (their indexes will decrement) to fill the gap left by the removed font.

Example

This code fragment loads the Postscript font Helvetica.

```
#include <lwserver.h>
#include <lwmodeler.h>

LWFontListFuncs *fontf;
int index;

fontf = global( LWFONTLISTFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( !fontf ) return AFUNC_BADGLOBAL;

/* You can get the default fonts directory from the
   Directory Info global. */

index = fontf->load( "Helvetica.pfd" );
```

Global Memory
Global Render Memory

Availability LightWave 6.0
Component Layout
Header lwrender.h

These globals allow plug-ins to allocate and share named chunks of memory. The memory comes from a pool managed by Layout. "Global Render Memory" is used during rendering and is freed automatically when rendering ends. "Global Memory" persists until the scene is cleared.

Global Call

```
   LWGlobalPool *memfunc;
   memfunc = global( LWGLOBALPOOL_RENDER_GLOBAL, GFUSE_TRANSIENT );
   memfunc = global( LWGLOBALPOOL_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWGlobalPool.

```
   typedef struct st_LWGlobalPool {
      LWMemChunk   (*first)  (void);
      LWMemChunk   (*next)   (LWMemChunk);
      const char * (*ID)     (LWMemChunk);
      int          (*size)   (LWMemChunk);
      LWMemChunk   (*find)   (const char *ID);
      LWMemChunk   (*create) (const char *ID, int size);
   } LWGlobalPool;
```

`mem = first()`

Returns the first memory chunk in the pool. This and the `next` function allow you to traverse the entire list of memory chunks in the pool. Use them if you need to search for memory chunks using criteria more complex than just the chunk ID string.

`mem = next( mem )`

Returns the next memory block in the list.

`name = ID( mem )`

Returns the chunk identifier. This is the name string that was passed to `create`.

`bytes = size( mem )`

Returns the size in bytes of a memory chunk.

`mem =` **`find`**`( name )`

Returns the memory chunk with the given ID. Multiple chunks may be created with the same ID, so this returns the first one.

`mem =` **`create`**`( name, size )`

Creates a memory chunk with the given size and ID and returns a pointer to the memory. If you want the name string to uniquely identify the chunk, you should try to `find` a chunk with your ID before using the ID in `create`.

Example

This code fragment allocates a render memory chunk named "my memory".

```
   #include <lwserver.h>
   #include <lwhost.h>
   #define COUNT 100

   static char name[] = "my widget memory";
   LWGlobalPool *memfunc;
   LWMemChunk mem;
   int *p, i;

   memfunc = global( LWGLOBALPOOL_RENDER_GLOBAL, GFUSE_TRANSIENT );
   if ( !memfunc ) goto NoMemFunc;  /* global calls can fail */

   mem = memfunc->find( name );
   if ( !mem )
      mem = memfunc->create( name, COUNT * sizeof( int ));
   if ( !mem )
      goto ErrorNoMem;

   p = ( int * ) mem;
   for ( i = 0; i < COUNT; i++ ) {
      p[ i ] = ...
```

Host Display Info

Availability LightWave 6.0
Component Layout, Modeler
Header lwdisplay.h

The host display global returns platform-specific information about LightWave's interface. This is primarily useful when you want to create a platform-dependent interface for your plug-in.

Global Call

```
HostDisplayInfo *hdi;
hdi = global( LWHOSTDISPLAYINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to a HostDisplayInfo, or NULL if called in a non-interactive (e.g. Screamernet) context.

```
typedef struct st_HostDisplayInfo {
 #ifdef _WIN32
   HANDLE    instance;
   HWND      window;
 #endif
 #ifdef _XGL
   Display  *xsys;
   Window    window;
 #endif
 #ifdef _MACOS
   WindowPtr window;
 #endif
} HostDisplayInfo;
```

The operating system #defines (`_WIN32`, `_XGL`, `_MACOS`) are discussed in the section on plug-in compiling.

`window`
LightWave's main window. This is often used as the parent window of the plug-in's main window.

`_WIN32` **`instance`**
Under Microsoft Windows, this is the instance handle for the LightWave process. You won't need this very often. If you want a handle to a resource (dialog templates, icons, bitmaps) that's stored in your plug-in's .p file, you need to use *your plug-in's* instance handle, not LightWave's. See your Win32 documentation to find out how to

get your instance handle using the `DllMain` function.

`_XGL` **`xsys`**

Under Unix, this is LightWave's window session handle.

Example

This code fragment displays everyone's favorite first message, but using the Win32 `MessageBox` function with LightWave's main window handle as the parent window.

```
#include <lwserver.h>
#include <lwdisplay.h>   /* includes windows.h under Windows */

HostDisplayInfo *hdi;

hdi = global( LWHOSTDISPLAYINFO_GLOBAL, GFUSE_TRANSIENT );

if ( hdi ) {
   MessageBox( hdi->window, "Hello, world!", "My Message", MB_OK );
}
```

Image List

Availability LightWave 6.0
Component Layout, Modeler
Header lwimage.h

This global provides access to LightWave's internal image list. Also see the Image Utility global.

A single image ID can refer to image sequences and animations as well as stills, and "image" is used here to refer to all of these. Most of the functions that return pixel information do so for the current state of the image, which generally depends on the current frame during rendering and on the most recently rendered frame at other times.

Global Call

```
   LWImageList *imglist;
   imglist = global( LWIMAGELIST_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWImageList.

```
   typedef struct st_LWImageList {
      LWImageID     (*first)    (void);
      LWImageID     (*next)     (LWImageID);
      LWImageID     (*load)     (const char *);
      const char *  (*name)     (LWImageID);
      const char *  (*filename) (LWImageID, LWFrame);
      int           (*isColor)  (LWImageID);
      void          (*needAA)   (LWImageID);
      void          (*size)     (LWImageID, int *w, int *h);
      LWBufferValue (*luma)     (LWImageID, int x, int y);
      void          (*RGB)      (LWImageID, int x, int y,
                                   LWBufferValue[3]);
      double        (*lumaSpot) (LWImageID, double x, double y,
                                   double spotSize, int blend);
      void          (*RGBSpot)  (LWImageID, double x, double y,
                                   double spotSize, int blend, double[3]);
      void          (*clear)    (LWImageID);
      LWImageID     (*sceneLoad)(const LWLoadState *);
      void          (*sceneSave)(const LWSaveState *, LWImageID);
      int           (*hasAlpha) (LWImageID);
      LWBufferValue (*alpha)    (LWImageID, int x, int y);
      double        (*alphaSpot)(LWImageID, double x, double y,
                                   double spotSize, int blend);
      LWPixmapID    (*evaluate) (LWImageID, LWTime t);
   } LWImageList;
```

`image = first()`

Returns the first image in the list.

`image = `**`next`**`( prev_image )`

Returns the image after `prev_image` in the list, or NULL if `prev_image` is the last image in the list.

`image = `**`load`**`( filename )`

Add the image to the list and return its ID. Animation files can be loaded with this function, but image sequences cannot.

`iname = `**`name`**`( image )`

Returns the name of the image as it appears to the user.

`fname = `**`filename`**`( image, frame )`

Returns the filename of the image. This is the value that should be stored for later retrieval of the image using `load`.

`iscol = `**`isColor`**`( image )`

True for images with color data and false for grayscale images.

`needAA``( image )`

Called by shaders that want to use the image list `lumaSpot` and `RGBSpot` functions during rendering. This tells Layout to prefilter the image for later spot evaluation. Currently this function can only be called from a shader's `init` function.

`size``( image, width, height )`

Returns the width and height of the image in pixels.

`gray = `**`luma`**`( image, x, y )`

Returns the grayscale value of a pixel. If this is a color image (`isColor` is true), the value returned is the NTSC/PAL luminance, which combines the RGB levels using the weights 0.2989 red, 0.5866 green, 0.1144 blue.

`RGB``( image, x, y, color )`

Returns the red, green and blue values of a pixel.

`gray = `**`lumaSpot`**`( image, x, y, spotsize, blend )`

Returns the grayscale value of a spot on the image. `x` and `y` are the center of the spot in pixels, and the spot size is the diameter of the spot in pixel units. The value is the weighted average of the pixels within the spot. If `blend` is true and the spot size is small, however, the value will be interpolated from neighboring pixels that may be outside the spot. Currently this function can only be called during the spot evaluation function of a shader, and `needAA` must have been called previously from the shader's `init` function.

`RGBSpot( image, x, y, spotsize, blend, color )`
Returns the color values of a spot on the image. Like `lumaSpot`, this function can only be called during the spot evaluation function of a shader.

`clear( image )`
Remove the image from the image list. This has the effect of removing all references to the image in the scene.

`image = sceneLoad( loadstate )`
Read an image reference from a file and add the image to the image list. This is meant to be called by a handler's load callback to retrieve an image that's part of its instance data. The reference will have been written by `sceneSave`.

`sceneSave( savestate, image )`
Write an image reference to a file. This is meant to be called by a handler's save callback to store a reference to the image as part of the handler's instance data.

`hasa = hasAlpha( image )`
True if the image includes an alpha channel.

`a = alpha( image, x, y )`
Returns the alpha value of a pixel.

`a = alphaSpot( image, x, y, spotsize, blend )`
Returns the alpha value of a spot (see `lumaSpot` and `RGBSpot`).

`pixmap = evaluate( image, time )`
Returns a pixmap of the image that can be used with the Image Utility global. This function creates a copy of the image, similar to calling the Image Utility `create` function and then copying the pixels using Image List `RGB` and Image Utility `setPixel`. But the image is *evaluated*, meaning that the frame matching the `time` argument is retrieved for sequences and animations, and any adjustments and filters are applied.

Example

The zcomp sample includes a pixel filter that composites the rendered image with a previously generated image based on the z-depth. It uses the Image List global to manage both the image to be composited and its z-buffer, which is treated as a floating-point grayscale image and read using the `luma` function.

Image Utility

Availability LightWave 6.0
Component Layout, Modeler
Header lwimage.h

This global provides functions for creating and saving still images. Also see the Image List global.

Pixmaps, used by this global and identified by LWPixmapID, differ from the images in the image list, which are identified by LWImageID. Pixmaps are stills, and you can draw on them and save them. Once saved to a file, the image can be loaded into LightWave using the Image List `load` function. The pixmap returned by the Image List `evaluate` function is a *copy* of the image, and drawing on this copy does not change the original image.

Global Call

```
   LWImageUtil *imgutil;
   imgutil = global( LWIMAGEUTIL_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWImageUtil.

```
   typedef struct st_LWImageUtil {
      LWPixmapID   (*create)    (int w, int h, LWImageType);
      void         (*destroy)   (LWPixmapID);
      int          (*save)      (LWPixmapID, int saver, const char *name);
      void         (*setPixel)  (LWPixmapID, int x, int y, void *pix);
      void         (*getPixel)  (LWPixmapID, int x, int y, void *pix);
      void         (*getInfo)   (LWPixmapID, int *w, int *h, int *type);
      LWPixmapID   (*resample)  (LWPixmapID, int w, int h, int mode);
      int          (*saverCount)(void);
      const char * (*saverName) (int saver);
   } LWImageUtil;
```

`image = create( w, h, type )`

Create a new image. The type specifies the organization of the pixel data and may be any of the image I/O pixel types.

`destroy( image )`

Release resources allocated by `create`. The image ID is no longer valid after this is called.

`result = save( image, saver_index, filename )`

Save the image to a file using the specified format. The format is determined by the choice of image saver, which can be one of Layout's built-in image savers or any of the installed ImageSaver class plug-ins. Use the `saverCount` and `saverName` functions to determine what formats are available and which saver index to use.

setPixel(image, x, y, pixel)

Set the value of a pixel in the image. The format of the pixel data depends on the pixel type of the image.

getPixel(image, x, y, pixel)

Get the value of a pixel in the image.

getInfo(image, w, h, type)

Get the width, height and pixel type of an image.

image2 = **resample**(image, w, h, mode)

Create a new image by resizing an existing image. The mode determines how the existing pixels will be resampled and can be one of the following values.

LWISM_SUBSAMPLING
LWISM_MEDIAN
LWISM_SUPERSAMPLING
LWISM_BILINEAR
LWISM_BSPLINE
LWISM_BICUBIC

count = **saverCount**()

Returns the number of available image savers.

name = **saverName**(saver_index)

Returns the name of an image saver.

Example

This example creates a rainbow image, saves it, and loads it into Layout's internal image list using the image list global.

```
#include <lwserver.h>
#include <lwimage.h>
#include <lwhost.h>

LWMessageFuncs *msg;
LWImageUtil *imgutil;
LWImageList *imglist;
LWImageID image;
```

```
   LWPixmapID pixmap;
   int x, y, saver, nsavers;
   unsigned char rgb[ 3 ];
   char *filename = "rainbow.tga";

   imgutil = global( LWIMAGEUTIL_GLOBAL, GFUSE_TRANSIENT );
   imglist = global( LWIMAGELIST_GLOBAL, GFUSE_TRANSIENT );
   msg = global( LWMESSAGEFUNCS_GLOBAL, GFUSE_TRANSIENT );
   if ( !imgutil || !imglist || !msg )
      return AFUNC_BADGLOBAL;

   pixmap = imgutil->create( 256, 20, LWIMTYP_RGB24 );
   if ( !pixmap ) {
      msg->error( "Couldn't create the image.", NULL );
      return AFUNC_OK;
   }

   for ( x = 0; x < 256; x++ )
      for ( y = 0; y < 20; y++ ) {
         hsv2rgb( 359.0f * x / 255.0f, y / 20.0f, 1.0f, rgb );
         imgutil->setPixel( pixmap, x, y, rgb );
      }

   nsavers = imgutil->saverCount();
   for ( saver = 0; saver < nsavers; saver++ )
      if ( !strcmp( "Targa Format (.tga)",
         imgutil->saverName( saver ))) break;

   if ( saver == nsavers )
      msg->error( "Couldn't find the Targa saver.", NULL );
   else
      imgutil->save( pixmap, saver, filename );

   imgutil->destroy( pixmap );

   image = imglist->load( filename );
```

The `hsv2rgb` function looks like this.

```
   void hsv2rgb( float h, float s, float v, char rgb[] )
   {
      float r, g, b, p, q, f, t;
      int i;

      if ( s == 0 ) {
         rgb[ 0 ] = rgb[ 1 ] = rgb[ 2 ] = ( int )( v * 255 );
         return;
      }

      h /= 60.0f;
      i = ( int ) h;
      f = h - i;
      p = v * ( 1.0f - s );
      q = v * ( 1.0f - ( s * f ));
      t = v * ( 1.0f - ( s * ( 1.0f - f )));

      switch ( i ) {
         case 0:  r = v;  g = t;  b = p; break;
         case 1:  r = q;  g = v;  b = p; break;
         case 2:  r = p;  g = v;  b = t; break;
         case 3:  r = p;  g = q;  b = v; break;
         case 4:  r = t;  g = p;  b = v; break;
      case 5:  r = v;  g = p;  b = q; break;
   }

   rgb[ 0 ] = ( int )( r * 255 );
   rgb[ 1 ] = ( int )( g * 255 );
   rgb[ 2 ] = ( int )( b * 255 );
}
```

Info Messages

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

This global provides a set of functions for displaying messages to the user.

Global Call

```
LWMessageFuncs *msgf;
msgf = global( LWMESSAGEFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWMessageFuncs.

```
typedef struct st_LWMessageFuncs {
   void (*info)     (const char *, const char *);
   void (*error)    (const char *, const char *);
   void (*warning)  (const char *, const char *);
   int  (*okCancel) (const char *title, const char *, const char *);
   int  (*yesNo)    (const char *title, const char *, const char *);
   int  (*yesNoCan) (const char *title, const char *, const char *);
   int  (*yesNoAll) (const char *title, const char *, const char *);
} LWMessageFuncs;
```

All of these functions allow you to display one or two lines of text. The second line is optional and may be NULL.

The first three functions are informational only. Depending on the user-defined alert level, the arguments are displayed either in a separate dialog with an OK button, or in a status area on the main window. The remaining functions allow you to ask the user questions, and they differ in the choice of responses available to the user. The possible return values are

	3	**2**	**1**	**0**
`okCancel`	-	-	OK	Cancel
`yesNo`	-	-	Yes	No
`yesNoCan`	-	Yes	No	Cancel
`yesNoAll`	Yes to All	Yes	No	Cancel

Example

This code fragment displays everyone's favorite first message.

```
#include <lwserver.h>
#include <lwhost.h>

LWMessageFuncs *msgf;

msgf = global( LWMESSAGEFUNCS_GLOBAL, GFUSE_TRANSIENT );

if ( msgf ) {
   msgf->info( "Hello, world!", NULL );
}
```

Instance Update

Availability LightWave 6.0
Component Layout, Modeler
Header lwhandler.h

A handler plug-in calls this to synchronize LightWave with changes to the plug-in's instance data. LightWave will refresh its own interface and will usually call the handler's evaluation function in the process.

Global Call

```
LWInstUpdate *instupdate;
instupdate = global( LWINSTUPDATE_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWInstUpdate.

```
typedef void LWInstUpdate (const char *class, LWInstance);
```

The arguments are the plug-in class of your handler and the instance that has changed.

Example

Several of the samples, including blotch, inertia and txchan use the update function.

Interface Info

Availability LightWave 6.0 **Component** Layout
Header lwrender.h

The interface info global returns information about the state of Layout's user interface. The data is read-only, but you can set the parameters using selection, navigation and display commands.

Global Call

```
   LWInterfaceInfo *intinfo;
   intinfo = global( LWINTERFACEINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWInterfaceInfo.

```
   typedef struct st_LWInterfaceInfo {
      LWTime           curTime;
      const LWItemID *selItems;
      unsigned int  (*itemFlags) (LWItemID);
      LWFrame          previewStart, previewEnd, previewStep;
      int              dynaUpdate;
      void          (*schemaPos) (LWItemID, double *x, double *y);
      int           (*itemVis)   (LWItemID);
      unsigned int     displayFlags;
      unsigned int     generalFlags;
      int              boxThreshold;
      int           (*itemColor) (LWItemID);
      int              alertLevel;
      int              autoKeyCreate;
   } LWInterfaceInfo;
```

`curTime`
: The current animation time selected in the Layout interface.

`selItems`
: A NULL-terminated array of item IDs for the items currently selected in the interface.

`flags = itemFlags( item )`
: Returns a set of bit flags for the item. These can be any combination of the following.

```
LWITEMF_SELECTED
LWITEMF_SHOWCHILDREN
LWITEMF_SHOWCHANNELS
LWITEMF_LOCKED
```

`previewStart`, **`previewEnd`**, **`previewStep`**

The range and step size used by the frame slider and by Layout previews. These differ from the range and step for rendering, which are returned by the scene info global.

`dynaUpdate`

Contains the current state of Layout's Dynamic Update setting, which controls how frequently the interface is updated while the user makes changes. Possible values are

```
LWDYNUP_OFF
LWDYNUP_DELAYED
LWDYNUP_INTERACTIVE
```

`schemaPos``( item, x, y )`

The `x` and `y` arguments receive the position of the item in schematic viewports. This and the `SchematicPosition` command can be used by plug-ins to rearrange the schematic views.

`visibility =` **`itemVis`**`( item )`

Returns a code describing how an item is drawn in the interface. For objects, this can be one of the following.

```
LWOVIS_HIDDEN
LWOVIS_BOUNDINGBOX
LWOVIS_VERTICES
LWOVIS_WIREFRAME
LWOVIS_FFWIREFRAME
LWOVIS_SHADED
LWOVIS_TEXTURED
```

Other item types are limited to `LWIVIS_HIDDEN` and `LWIVIS_VISIBLE`.

`displayFlags`

Returns the state of certain display options as bit fields combined using bitwise-or. When set, a bit indicates that the corresponding option is turned on for the display.

```
LWDISPF_MOTIONPATHS
LWDISPF_HANDLES
LWDISPF_IKCHAINS
LWDISPF_CAGES
LWDISPF_SAFEAREAS
LWDISPF_FIELDCHART
```

`generalFlags`

Returns the state of certain interface options as bit fields combined using bitwise-or. When set, a bit indicates that the corresponding option is turned on for the interface.

```
LWGENF_HIDETOOLBAR
LWGENF_RIGHTTOOLBAR
LWGENF_PARENTINPLACE
```

```
LWGENF_FRACTIONALFRAME
LWGENF_KEYSINSLIDER
LWGENF_PLAYEXACTRATE
LWGENF_AUTOKEY
```

`boxThreshold`

The bounding box threshold. Objects with a number of points greater than this threshold are drawn initially as bounding boxes to speed up interaction.

`color_index =` **`itemColor`**`( item )`

Returns an index into the list of colors used to drawing an item's wireframe.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
⬛	🟦	🟩	🟦	🟥	🟪	🟫	⬜	🟦	🟩	🟦	🟥	🟪	🟨	⬜

`alertLevel`

The alert level for information dialogs. This affects how the information is displayed. Possible values are

```
LWALERT_BEGINNER
LWALERT_INTERMEDIATE
LWALERT_EXPERT
```

`autoKeyCreate`

The auto key create mode state, defined as one of these values:

```
LWAKC_OFF
LWAKC_MODIFIED
LWAKC_ALL.
```

History

In LightWave 7.5, the `autoKeyCreate` field was added, along with the `LWGENF_AUTOKEY` generalFlags definition.

Example

This code fragment collects information about the currently selected items.

```
   #include <lwserver.h>
   #include <lwrender.h>

   LWInterfaceInfo *intinfo;
   LWItemInfo *iteminfo;
   LWTime t;
   LWItemID *id;
   int i, f, type;
```

```
intinfo = global( LWINTERFACEINFO_GLOBAL, GFUSE_TRANSIENT );
iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
if ( !intinfo || !iteminfo ) return AFUNC_BADGLOBAL;

t = intinfo->curTime;
id = intinfo->selItems;
for ( i = 0; id[ i ]; i++ ) {
   f = intinfo->itemFlags( id[ i ] );
   type = iteminfo->type( id[ i ] );
   switch ( type ) {
      case LWI_OBJECT:
         ...
```

Item Info

Availability LightWave 6.0 **Component** Layout
Header lwrender.h

The item info global returns functions for traversing a list of the items in a scene and for getting information about any one of them. The information available through this global is common to all item types. Information specific to certain item types is provided through separate global functions.

Global Call

```
   LWItemInfo *iteminfo;
   iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWItemInfo.

```
   typedef struct st_LWItemInfo {
      LWItemID      (*first)       (LWItemType, LWItemID);
      LWItemID      (*next)        (LWItemID);
      LWItemID      (*firstChild)  (LWItemID parent);
      LWItemID      (*nextChild)   (LWItemID parent, LWItemID prevChild);
      LWItemID      (*parent)      (LWItemID);
      LWItemID      (*target)      (LWItemID);
      LWItemID      (*goal)        (LWItemID);
      LWItemType    (*type)        (LWItemID);
      const char *  (*name)        (LWItemID);
      void          (*param)       (LWItemID, LWItemParam, LWTime,
                                     LWDVector);
      unsigned int  (*limits)      (LWItemID, LWItemParam,
                                     LWDVector min, LWDVector max);
      const char *  (*getTag)      (LWItemID, int);
      void          (*setTag)      (LWItemID, int, const char *);
      LWChanGroupID (*chanGroup)   (LWItemID);
      const char *  (*server)      (LWItemID, const char *, int);
      unsigned int  (*serverFlags) (LWItemID, const char *, int);
      void          (*controller)  (LWItemID, LWItemParam, int type[3]);
      unsigned int  (*flags)       (LWItemID);
      LWTime        (*lookAhead)   (LWItemID);
      double        (*goalStrength)(LWItemID);
      void          (*stiffness)   (LWItemID, LWItemParam, LWDVector);
      unsigned int  (*axisLocks)   (LWItemID, LWItemParam);

   } LWItemInfo;
```

`id = first( itemtype, bone_object )`

Returns the ID of the first item of a given type, or `LWITEM_NULL` if there are no items of this type in the scene. Valid item types are

```
LWI_OBJECT
LWI_LIGHT
LWI_CAMERA
LWI_BONE
```

If `itemtype` is `LWI_BONE`, the second argument is the ID of the boned object. Otherwise it should be `LWITEM_NULL`.

`id = `**`next`**`( item )`

Returns the next item of the same type as the `item` argument. If there are no more, this returns `LWITEM_NULL`.

`id = `**`firstChild`**`( parent )`

Returns the first child item of the parent item, or `LWITEM_NULL` if the parent item has no children.

`id = `**`nextChild`**`( parent, child )`

Returns the next child item given a parent item and the previous child, or `LWITEM_NULL` if there are no more children.

`id = `**`parent`**`( item )`

Returns the item's parent, if any, or `LWITEM_NULL`.

`id = `**`target`**`( item )`

Returns the item's target, if any, or `LWITEM_NULL`.

`id = `**`goal`**`( item )`

Returns the item's goal, if any, or `LWITEM_NULL`.

`itemtype = `**`type`**`( item )`

Returns the type of an item.

`itemname = `**`name`**`( item )`

Returns the name of the item as it appears to the user.

`param``( item, param_type, time, vector )`

Returns vector parameters associated with an item. This data is read-only. The `param_type` argument identifies which parameter vector you want. The parameters are

`LWIP_POSITION`

The keyframed position *before* parenting. Equivalently, if the item is parented, this is its position relative to its parent.

`LWIP_W_POSITION`

The keyframed position in world coordinates (*after* parenting).

`LWIP_ROTATION`

The keyframed rotation, in radians (relative to its parent's rotation).

`LWIP_SCALING`

The keyframed scale factors (relative to the parent's scale).

`LWIP_PIVOT`

`LWIP_PIVOT_ROT`

The item's pivot point position and rotation, in its own coordinates. The pivot point is the origin for the item's rotations.

`LWIP_RIGHT`

`LWIP_UP`
`LWIP_FORWARD`

+X, +Y and +Z direction vectors for the item, in world coordinates. Together they form the item's rotation and scale transformation matrix. Since they include scaling, these vectors aren't normalized.

`LWIP_W_RIGHT`

`LWIP_W_UP`
`LWIP_W_FORWARD`

+X, +Y and +Z direction vectors for the world, in item coordinates. In other words, these are the inverse of the previous parameters.

The value is written to the vector array for the given time.

`flags = `**`limits`**`( item, param_type, minvec, maxvec )`

Get upper and lower bounds on vector parameters. These may be limits set by the user on joint angles or ranges of movement. The function returns an integer containing bit flags that indicate which of the three vector components contain limits. The symbols for these bits are

`LWVECF_0`
`LWVECF_1`
`LWVECF_2`

If the bit is set, then the corresponding element of the vector array contains a valid limit. If the bit is 0, the channel is unbounded.

`tag = `**`getTag`**`( item, tagnum )`

Retrieve a tag string associated with an item.The tags are numbered starting at 1. `getTag` returns NULL if the tag number is out of range. Tags strings are stored with the item in the scene file.

`setTag``( item, tagnum, tag )`

Associate a tag string with an item. If `tagnum` is 0, a new tag is created for the item. If `tagnum` is the number of an existing tag, the tag string for that tag is replaced. If `tagnum` is outside these values, the `setTag` call is ignored.

`changroup = `**`chanGroup`**`( item )`

Returns the channel group associated with an item. Use this with the Animation Envelopes and Channel Info globals.

`servname = `**`server`**`( item, class, index )`

Returns the name of a plug-in applied to an item. The class argument is the class name, and the index refers to the position in the server list for that class. The first server in the list has an index of 1. Returns NULL if no plug-in matching the arguments can be found. This function can also be used to query the names of servers that aren't associated with items, such as pixel and image filters and volumetrics, and for those the item ID is ignored.

`flags = `**`serverFlags`**`( item, class, index )`

Returns flags for the plug-in identified by the item, class name and server list index. Currently the possible flags are

```
LWSRVF_DISABLED
LWSRVF_HIDDEN
```

`controller``( item, param_type, hpb_controllers )`

Returns a code indicating which mechanism controls the item's rotation. The third argument is an array of three integers, one each for heading, pitch and bank, that receive a controller code for keyframes, targeting, alignment to a path, or inverse kinematics.

```
LWMOTCTL_KEYFRAMES
LWMOTCTL_TARGETING
LWMOTCTL_ALIGN_TO_PATH
LWMOTCTL_IK
```

`itemflags = `**`flags`**`( item )`

Returns certain item settings as a set of bit flags.

```
LWITEMF_ACTIVE
LWITEMF_UNAFFECT_BY_IK
LWITEMF_FULLTIME_IK
LWITEMF_GOAL_ORIENT
LWITEMF_REACH_GOAL
```

`time = `**`lookAhead`**`( item )`

Returns the look-ahead interval, in seconds, for motion channels controlled by `LWMOTCTL_ALIGN_TO_PATH`. This is the amount of time by which changes in orientation of the item anticipate changes in the path direction.

`strength = `**`goalStrength`**`( item )`

Returns the item's IK goal strength.

`stiffness``( item, param_type, vector )`

Fills `vector` with the item's joint stiffness settings in heading, pitch and bank. Use `LWIP_ROTATION` as the `param_type`.

`flags = `**`axisLocks`**`( item, param_type )`

Returns bits indicating which channels are locked in the UI. See **limits().**

History

LightWave 7.5 added the `axisLocks` function, but `LWITEMINFO_GLOBAL` was *not* incremented. If you ask for "LW Item Info 3", use the Product Info global to determine whether you're running in LightWave 7.0 or later before attempting to call these functions.

Example

This code fragment traverses the object list, collecting names and some parameters.

```
#include <lwserver.h>
#include <lwrender.h>

LWItemInfo *iteminfo;
LWItemID id;
char *name;
LWTime t = 3.0;          /* seconds */
LWDVector rt, up, fd;

iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
if ( iteminfo ) {
   id = iteminfo->first( LWI_OBJECT, NULL );
   while ( id ) {
      name = iteminfo->name( id );
      iteminfo->param( id, LWIP_RIGHT, t, rt );
      iteminfo->param( id, LWIP_UP, t, up );
      iteminfo->param( id, LWIP_FORWARD, t, fd );
      if ( rt[ 0 ] > 0.0 ) { ...

      id = iteminfo->next( id );
   }
}
```

The vectors returned by the `param` function can be used to transform points between item and world coordinates. In the following fragments, `p` is the position of a point in item coordinates and `q` is the same point's position in world coordinates:

```
LWDVector p, q, rt, up, fd, wrt, wup, wfd, wpos, piv;
LWItemID id;
...
iteminfo->param( id, LWIP_RIGHT,      t, rt );
iteminfo->param( id, LWIP_UP,         t, up );
iteminfo->param( id, LWIP_FORWARD,    t, fd );
iteminfo->param( id, LWIP_W_POSITION, t, wpos );
iteminfo->param( id, LWIP_PIVOT,      t, piv );
```

To convert from item to world coordinates, subtract the pivot position (to move the rotation origin to the world origin), multiply by the matrix formed from the direction vectors, and offset the result by the world position of the item.

```
for ( i = 0; i < 3; i++ )
   q[ i ] = ( p[ 0 ] - piv[ 0 ] ) * rt[ i ]
          + ( p[ 1 ] - piv[ 1 ] ) * up[ i ]
          + ( p[ 2 ] - piv[ 2 ] ) * fd[ i ]
          + wpos[ i ];
```

To transform from world to item coordinates, just apply the same procedure in reverse, using the inverse direction vectors.

```
iteminfo->param( id, LWIP_W_RIGHT,   t, wrt );
iteminfo->param( id, LWIP_W_UP,      t, wup );
iteminfo->param( id, LWIP_W_FORWARD, t, wfd );

for ( i = 0; i < 3; i++ )
   p[ i ] = ( q[ 0 ] - wpos[ 0 ] ) * wrt[ i ]
          + ( q[ 1 ] - wpos[ 1 ] ) * wup[ i ]
          + ( q[ 2 ] - wpos[ 2 ] ) * wfd[ i ]
          + piv[ i ];
```

Layout Monitor

Availability LightWave 7.0
Component Layout
Header lwmonitor.h

The Layout monitor global returns functions for initializing and displaying a progress dialog in Layout. This is primarily for showing the progress of lengthy or complex operations in non-handler plug-in classes. Filters and file importer classes have their own monitor mechanisms. See also the monitor global for Modeler.

Global Call

```
   LWLMonFuncs *lmonf;
   lmonf = global( LWLMONFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWLMonFuncs.

```
   typedef struct st_LWLMonFuncs {
      LWLMonID (*create)   (void);
      void     (*setup)    (LWLMonID, char *title, unsigned int flags,
                              const char *histfile);
      void     (*setwinpos)(LWLMonID, int x, int y, int w, int h);
      void     (*init)     (LWLMonID, unsigned int total, const char *);
      int      (*step)     (LWLMonID, unsigned int incr, const char *);
      void     (*done)     (LWLMonID);
      void     (*destroy)  (LWLMonID);
   } LWLMonFuncs;
```

mon = **create**()
: Create a new monitor instance. The monitor returned by `create` is passed as the first argument to the other monitor functions.

setup(mon, title, flags, histfile)
: Configure the monitor. The `title` string is the title of the monitor window. The `histfile` is the filename of a history file where messages displayed to the user will also be written. This can be NULL if you don't want a history file. The flags can be any of the following combined using bitwise-or.

  LMO_NOABORT
  : By default, the user can inform you that your operation should

be stopped. This flag disables the Abort button.

LMO_REVIEW

If this is set, the monitor window remains open after you call `done`. This allows the user to review the messages displayed during the operation.

LMO_HISTAPPEND

By default, the history file is overwritten each time `init` is called. This flag causes new message strings to be appended to the file instead.

LMO_IMMUPD

Enables immediate update of the monitor on every step. The default is to delay updates to avoid incurring too much overhead for rapid step events.

setwinpos(mon, x, y, w, h)

Set the position and size of the monitor window. The dimensions are in pixels. If you don't call this, Layout will select defaults for you.

init(mon, total, message)

Open the monitor window. The `total` is the number of steps in the operation. While `step` is being called, Layout displays your progress to the user as a percentage of this total. The `message` is the first string displayed to the user.

abort = **step**(mon, increment, message)

Advance the progress display by the fraction `total/increment`. When the sum of the steps reaches the total, the progress display will indicate to the user that the task has finished. An increment of 0 is valid and can be used to change the message without changing the progress indication. The `message` can also be NULL, in which case Layout may substitute a generic progress message. If `step` returns 1, the user has requested that the task be aborted.

done(mon)

Tell the monitor that the task has been completed. If the flags passed to `setup` included `LMO_REVIEW`, the monitor window remains open and control won't be returned from `done` until the user closes the window. Otherwise `done` closes the window and control returns immediately.

destroy(mon)

Free the monitor instance and resources allocated by `create`. If it's open, the monitor window will be closed.

Example

This code fragment creates and displays a monitor in Layout. Displaying progress to the user is helpful but not essential, so in most cases failure in some part of the monitor processing shouldn't cause your plug-in to fail.

```
#include <lwserver.h>
#include <lwgeneric.h>
#include <lwmonitor.h>
#include <time.h>

LWLMonFuncs *monf;
LWLMonID mon;
int i, total, step;

/* get the global and a monitor */

monf = global( LWLMONFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( monf ) {
   mon = monf->create();
   if ( mon )
      monf->setup( mon, "Just Testing", LMO_REVIEW, NULL );
}

...
/* perform a lengthy task */

if ( monf && mon ) monf->init( mon, total, "Starting..." );
for ( i = 0; i < total; i += step ) {
   ...do something...
   if ( monf && mon )
      if ( monf->step( mon, step, NULL )) {
         monf->step( mon, 0, "Aborted!" );
         break;
      }
}
if ( monf && mon ) monf->done( mon );

...
/* no longer need the monitor */

if ( monf && mon ) monf->destroy( mon );
```

Light Info

Availability LightWave 6.0 **Component** Layout, Modeler
Header lwrender.h

The light info global returns functions for getting light-specific information about any of the lights in a scene. Use the Item Info global to get the light list and for generic item information. The information returned by these functions is read-only, but you can use commands to set many of the parameters.

Global Call

```
   LWLightInfo *lightinfo;
   lightinfo = global( LWLIGHTINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWLightInfo.

```
   typedef struct st_LWLightInfo {
      void         (*ambient)     (LWTime, LWDVector color);
      int          (*type)        (LWItemID);
      void         (*color)       (LWItemID, LWTime, LWDVector color);
      int          (*shadowType)  (LWItemID);
      void         (*coneAngles)  (LWItemID, LWTime, double *radius,
                                     double *edge);
      unsigned int (*flags)       (LWItemID);
      double       (*range)       (LWItemID, LWTime);
      int          (*falloff)     (LWItemID);
      LWImageID    (*projImage)   (LWItemID);
      int          (*shadMapSize) (LWItemID);
      double       (*shadMapAngle)(LWItemID, LWTime);
      double       (*shadMapFuzz) (LWItemID, LWTime);
      int          (*quality)     (LWItemID);
      void         (*rawColor)    (LWItemID, LWTime, LWDVector color);
      double       (*intensity)   (LWItemID, LWTime);
      void         (*shadowColor)  (LWItemID, LWTime, LWDVector color);

   } LWLightInfo;
```

`ambient( time, color )`

Returns the color of the global ambient light for the scene at the given time. The RGB levels include the effect of the user's intensity setting for the ambient light.

`lighttype = type( light )`

Returns the type of the light as one of the following values.

```
LWLIGHT_DISTANT
LWLIGHT_POINT
LWLIGHT_SPOT
LWLIGHT_LINEAR
LWLIGHT_AREA
```

color(light, time, rgb)

Sets the `rgb` argument to the color of the light (with intensity factored in) at the given time. Use the `rawColor` and `intensity` functions for separate access to these settings.

shadowtype = **shadowType**(light)

Returns the shadow type for the light as one of the following values.

```
LWLSHAD_OFF
LWLSHAD_RAYTRACE
LWLSHAD_MAP
```

coneAngles(light, time, radius, edge)

Returns the cone angles for spotlights. `radius` receives an angle that is half the total light cone angle, and `edge` receives the angular width of the soft edge. Both angles are in radians.

settings = **flags**(light)

Returns flag bits for settings related to the light.

```
LWLFL_LIMITED_RANGE
LWLFL_NO_DIFFUSE
LWLFL_NO_SPECULAR
LWLFL_NO_CAUSTICS
LWLFL_LENS_FLARE
LWLFL_VOLUMETRIC
LWLFL_NO_OPENGL
LWLFL_FIT_CONE
LWLFL_CACHE_SHAD_MAP
```

The `FIT_CONE` flag indicates that the shadow map angle is set to the light's spotlight cone angle.

dist = **range**(light, time)

Returns the range or nominal distance for the light. The interpretation of this value depends on the falloff type. If falloff is linear, the value is the distance at which the intensity of the light falls to 0. For inverse distance falloff types, the value is the distance at which the intensity equals the user's intensity setting for the light. When there's no falloff (the `falloff` function returns `LWLFALL_OFF`, or the `LWLFL_LIMITED_RANGE` flag bit is clear), the return value is undefined.

falloff_type = **falloff**(light)

Returns the falloff type. Falloff scales the intensity of a light as a function of *d* (distance from the light) and *r* (the value returned by the `range` function).

`LWLFALL_OFF`	1 (no falloff)
`LWLFALL_LINEAR`	1 - d / r (or 0 when $d > r$)
`LWFALL_INV_DIST`	r / d
`LWFALL_INV_DIST_2`	$(r / d)^2$

`image = `**`projImage`**`( light )`

Returns the image ID of the projection image. Use the Image List global to get information about the image.

`size = `**`shadMapSize`**`( light )`

The size of the shadow map. Shadow maps are square arrays of pixels, so the amount of memory used by a shadow map is proportional to the square of the size.

`angle = `**`shadMapAngle`**`( light, time )`

The angle subtended by the shadow map, in radians.

`fuzziness = `**`shadMapFuzz`**`( light, time )`

The amount of fuzziness at the edges of shadows in the shadow map.

`index = `**`quality`**`( light )`

The quality level of an extended (linear or area) light source, proportional to the number of sample points on the light.

`rawColor``( light, time, rgb )`
`level = `**`intensity`**`( light, time )`

These return the separate components of the light color returned by the `color` function.

`shadowColor``( light, time, rgb )`

Returns the shadow color for the light in `rgb`.

History

In LightWave 7.5, the `shadowColor` function was added.

Example

This code fragment collects information about the first light.

```
#include <lwserver.h>
#include <lwrender.h>

LWItemInfo *iteminfo;
LWLightInfo *ltinfo;
LWItemID id;
LWTime t = 3.0;          /* seconds */
LWDVector color;
double range, radius, edge;
int lighttype, shadowtype;
unsigned int flags;

iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
ltinfo   = global( LWLIGHTINFO_GLOBAL, GFUSE_TRANSIENT );

if ( iteminfo && ltinfo ) {
   id = iteminfo->first( LWI_LIGHT, NULL );
   lighttype  = ltinfo->type( id );
   shadowtype = ltinfo->shadowType( id );
   flags      = ltinfo->flags( id );
   ltinfo->color( id, t, color );

   if ( type == LWLIGHT_SPOT )
      ltinfo->coneAngles( id, &radius, &edge );
   if ( flags & LWLFL_LIMITED_RANGE )
      range = ltinfo->range( id );
}
```

Locale Info

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The locale info global returns a code indicating the (human) language setting of the system.

Global Call

```
unsigned long locinfo;
locinfo = ( unsigned long ) global( LWLOCALEINFO_GLOBAL,
   GFUSE_TRANSIENT );
```

The global function ordinarily returns a `void *`, so this should be cast to an integer type to get the return value.

The language ID is in the low 16 bits of the return value. The high 16 bits are reserved for future use. The language ID can be extracted using a macro defined in `lwhost.h`.

```
langid = locinfo & LWLOC_LANGID;
```

The language IDs are identical to those defined in the Microsoft Win32 API and exposed in the Microsoft Visual C++ `winnt.h` header file. Bits 7 - 0 define the language group and bits 15 - 8 define the sublanguage. The plug-in SDK header file `lwserver.h` contains symbols for some of the more common language IDs.

```
LANGID_GERMAN      0x0407
LANGID_USENGLISH   0x0409
LANGID_UKENGLISH   0x0809
LANGID_SPANISH     0x040a
LANGID_FRENCH      0x040c
LANGID_ITALIAN     0x0410
LANGID_JAPANESE    0x0411
LANGID_KOREAN      0x0412
LANGID_RUSSIAN     0x0419
LANGID_SWEDISH     0x041D
```

Note that the low order bits for `USENGLISH` and `UKENGLISH` are the same. Win32 defines 9 flavors of English (as well as 16 flavors of both Arabic and Spanish, for example) that are distinguished by sublanguage code.

Your plug-in isn't required to implement localization, but even if you don't provide error messages or panel text in multiple languages, you may still want to localize things like date formats or currency symbols.

Example

The following code fragment selects a greeting string based on the locale.

```
#include <lwserver.h>
#include <lwhost.h>

unsigned long locinfo;
locinfo = ( unsigned long ) global( LWLOCALEINFO_GLOBAL,
   GFUSE_TRANSIENT );

switch ( locinfo & LWLOC_LANGID ) {
   case LANGID_GERMAN:    msg = "Guten Tag";         break;
   case LANGID_USENGLISH:
   case LANGID_UKENGLISH: msg = "Good day";          break;
   case LANGID_SPANISH:   msg = "Buenos dias";       break;
   case LANGID_FRENCH     msg = "Bonjour";           break;
   case LANGID_ITALIAN    msg = "Buon giorno";       break;
   case LANGID_JAPANESE   msg = "Konnichi wa";       break;
   case LANGID_KOREAN     msg = "Annyoung hase yo"; break;
   case LANGID_RUSSIAN    msg = "Zdravstvuite";      break;
   case LANGID_SWEDISH    msg = "God dag";           break;
   ...
```

Comp Info

Availability LightWave 6.0
Component Layout
Header lwrender.h

The compositing info global identifies the images being used as the background, foreground and foreground alpha images. This data structure is read-only.

Global Call

```
LWCompInfo *compinfo;
compinfo = global( LWCOMPINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWCompInfo.

```
typedef struct st_LWCompInfo {
   LWImageID bg;
   LWImageID fg;
   LWImageID fgAlpha;
} LWCompInfo;
```

bg
The background image.

fg
The foreground image.

fgAlpha
The foreground alpha image.

Example

This code fragment retrieves information about the background image.

```
#include <lwserver.h>
#include <lwrender.h>
#include <lwimage.h>

LWCompInfo *compinfo;
LWImageList *imglist;
char *name;
int width, height;

compinfo = global( LWCOMPINFO_GLOBAL, GFUSE_TRANSIENT );
imglist = global( LWIMAGELIST_GLOBAL, GFUSE_TRANSIENT );
if ( !compinfo || !imglist ) goto ErrorBadGlobal;
```

```
name = imglist->name( compinfo.bg );
imglist->size( compinfo.bg, &width, &height );
...
```

Multithreading Utilities

Availability LightWave 6.0
Component Layout, Modeler
Header lwmtutil.h

The multithreading global supplies a mutex (mutual exclusion) mechanism for managing threaded execution of your plug-in. LightWave may invoke your plug-in from multiple threads simultaneously, which has the effect of threading your code. But when doing certain things, for example when reading and writing global data, the threads of your code should be executed one at a time, rather than all at once. The mutex mechanism is a way for the threads of your code to cooperate in waiting for one another.

Think of a mutex as a dressing room, a place where a thread can have some privacy. Any time your plug-in needs to do something synchronously (one thread at a time), you ask to be let into the dressing room by calling `lock`. If another thread (another "you") is already in that dressing room, your thread waits until the other thread is done. Then your thread gets the dressing room, and other threads that want that dressing room must wait for you to finish. When you're finished, you call `unlock`.

The LWMTUtilID returned by the `create` function allows you to use up to 10 separate mutexes. These are numbered from 0 to 9 and are passed as the second argument to `lock` and `unlock`. You might think of these as 10 different dressing rooms.

Multithreading is a complex topic. If you're unfamiliar with it, you're encouraged to seek out a general programming text that discusses the writing of thread-safe code.

Global Call

```
   LWMTUtilFuncs *mtutil;
   mtutil = global( LWMTUTILFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWMTUtilFuncs.

```
   typedef struct st_LWMTUtilFuncs {
      LWMTUtilID (*create) (void);
      void       (*destroy)(LWMTUtilID mtid);
      int        (*lock)   (LWMTUtilID mtid, int mutexID);
      int        (*unlock) (LWMTUtilID mtid, int mutexID);
   } LWMTUtilFuncs;
```

`mtid = create()`

Returns an LWMTUtilID that can be used by the lock and unlock functions. The return value is NULL if `create` fails.

`destroy( mtid )`

Free resources allocated by `create`.

`ok = lock( mtid, index )`

Blocks until the mutex becomes available. Returns true if successful, or false if the lock couldn't be executed for some reason. The index is an integer from 0 to 9 that identifies which of the ten mutexes to lock. If another thread has already called `lock` for this mutex, the calling thread waits until the other thread calls `unlock`.

`ok = unlock( mtid, index )`

Release the mutex. If another thread has been waiting for this mutex, that thread will execute. Returns true if successful, otherwise false.

Example

This code fragment outlines the sequence of steps you'd take to use a mutex.

```
   #include <lwmtutil.h>

   LWMTUtilFuncs *mtutil;
   LWMTUtilID mtid;

   mtutil = global( LWMTUTILFUNCS_GLOBAL, GFUSE_TRANSIENT );
   if ( !mtutil )
      ...global not available, do this some other way...

   /* create the mutex */
   mtid = mtutil->create();
   ...

   /* enclose critical code (code that must run synchronously) in
      matching lock()/unlock() calls */
   if ( mtutil->lock( mtid, 0 )) {
      ...do something that can't be threaded...
      mtutil->unlock( mtid, 0 );
   }
   ...

   /* free the mutex when you no longer need it */
   if ( mtid ) mtutil->destroy( mtid );
```

Object Info

Availability LightWave 6.0 **Component** Layout
Header lwrender.h

The object info global returns functions for getting object-specific information about any of the objects in a scene. Use the Item Info global to get the object list and for generic item information. See also the Scene Objects global. The data returned by the object info functions is read-only, but you can use commands to set many of the parameters.

Global Call

```
   LWObjectInfo *objinfo;
   objinfo = global( LWOBJECTINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWObjectInfo.

```
   typedef struct st_LWObjectInfo {
      const char * (*filename)    (LWItemID);
      int          (*numPoints)   (LWItemID);
      int          (*numPolygons) (LWItemID);
      unsigned int (*shadowOpts)  (LWItemID);
      double       (*dissolve)    (LWItemID, LWTime);
      LWMeshInfoID (*meshInfo)    (LWItemID, int frozen);
      unsigned int (*flags)       (LWItemID);
      double       (*fog)         (LWItemID, LWTime);
      LWTextureID  (*dispMap)     (LWItemID);
      LWTextureID  (*clipMap)     (LWItemID);
      void         (*patchLevel)  (LWItemID, int *display, int *render);
      void         (*metaballRes) (LWItemID, double *display,
                                     double *render);
      LWItemID     (*boneSource)  (LWItemID);
      LWItemID     (*morphTarget) (LWItemID);
      double       (*morphAmount) (LWItemID, LWTime);
      unsigned int (*edgeOpts)    (LWItemID);
      void         (*edgeColor)   (LWItemID, LWTime, LWDVector color);
      int          (*subdivOrder) (LWItemID);
      double       (*polygonSize) (LWItemID, LWTime);
      int          (*excluded)    (LWItemID object, LWItemID light);
      void         (*matteColor)  (LWItemID, LWTime, LWDVector color);
      double       (*thickness)   (LWItemID, LWTime, int type);
      double       (*edgeZScale)  (LWItemID, LWTime);
   } LWObjectInfo;
```

`name = **filename**( object )`
Returns the filename for the object file.

`count = **numPoints**( object )`

Returns the number of points in the object mesh.

`count = **numPolygons**( object )`

Returns the number of polygons in the object mesh.

`sopts = **shadowOpts**( object )`

Returns bits for shadow options.

```
LWOSHAD_SELF
LWOSHAD_CAST
LWOSHAD_RECEIVE
```

`amount = **dissolve**( object, time )`

Returns the object dissolve amount at the given time.

`meshinfo = **meshInfo**( object, frozen )`

Returns a mesh info structure for the object. This is a complete description of the object's geometry. See the Mesh Info page for a detailed discussion of mesh info structures.

If `frozen` is true, the mesh for objects with subpatches and metaballs will contain the geometry that results from subdivision and isosurface calculation. The `pntBasePos` function will return the same point positions that Layout uses for object coordinate texture mapping. These are completely undeformed positions in the case of regular polygons and subpatches, and positions at freezing time for metaballs and partigons. `pntOtherPos` will return the actual world coordinates used by Layout. These should only be considered final if the mesh is obtained after all object transformations have been completed.

`f = **flags**( object )`

Returns the state of certain object settings as bits combined using bitwise-or. Possible flags are

```
LWOBJF_UNSEEN_BY_CAMERA
LWOBJF_UNSEEN_BY_RAYS
LWOBJF_UNAFFECT_BY_FOG
LWOBJF_MORPH_MTSE
LWOBJF_MORPH_SURFACES
```

`foglevel = **fog**( object, time )`

Returns the amount by which the object is affected by fog.

`texture =` **`dispMap`**`( object )`

Returns the texture ID of the displacement image map applied to the object.

`texture =` **`clipMap`**`( object )`

Returns the texture ID of the clip map applied to the object.

`patchLevel``( object, display, render )`

Returns the interface and render patch level for the object's subpatches.

`metaballRes``( object, display, render )`

Returns the interface and render resolution of the object's metaballs.

`boneobj =` **`boneSource`**`( object )`

Returns the object whose bones are being used to deform the given object. (An object can be deformed by the bones of another object.)

`morphobj =` **`morphTarget`**`( object )`

Returns the morph target of the given object.

`amount =` **`morphAmount`**`( object, time )`

Returns the morph amount at a given time. If `flags` returns the `LWOBJF_MORPH_MTSE` bit, Multiple Target/Single Envelope morphing is enabled, and the morph amount includes an index into a chain of morph targets. Assume A's target is B, and B's target is C. Morph amounts between 0.0 and 1.0 morph A to B, while amounts between 1.0 and 2.0 morph A to C. The interpolant is the fractional part of the morph amount, and the index is the integer part.

`options =` **`edgeOpts`**`( object )`

Returns the object's edge rendering options, which can be any of the following combined using bitwise-or.

`LWEDGEF_SILHOUETTE`

`LWEDGEF_UNSHARED`

`LWEDGEF_CREASE`

`LWEDGEF_SURFACE`
`LWEDGEF_OTHER`

Edge lines are drawn in the indicated areas. An unshared edge belongs to only one polygon. A crease is an edge where two polygons meet at an angle exceeding the max smoothing angle of the surface.

A surface edge is where the polygons on either side have different surfaces.

`LWEDGEF_SHRINK_DIST`

The thickness of the lines is proportional to distance from the camera.

`edgeColor( object, time, color )`

The color used to render edges is written in the `color` argument.

`index = subdivOrder( object )`

Returns the subdivision order as a 0-based index into a list of options.

0 - First
1 - After Morphing
2 - After Bones
3 - After Displacement
4 - After Motion
5 - Last

`size = polygonSize( object, time )`

Returns the polygon size setting. This is a scale factor with a default of 1.0.

`state = excluded( object, light )`

Returns true if the light is excluded from the object. Light exclusion is a user setting that prevents the light from affecting the rendering of the object.

`matteColor(object,  time,  color)`

The matte color set for the object is writen into the `color` argument.

`thick = thickness(object,  time,  type)`

Returns the thickness for the specified edge type, where `type` is one of the following:

`LWTHICK_SILHOUETTE`

```
      LWTHICK_UNSHARED
      LWTHICK_CREASE
      LWTHICK_SURFACE
      LWTHICK_OTHER
      LWTHICK_LINE
      LWTHICK_PARTICLE_HEAD
      LWTHICK_PARTICLE_TAIL
```

`zsc =` **`edgeZScale`**`(object,  time)`

Returns the Edge Z Scale setting for edge rendering of `object`.

History

In LightWave 7.5, the following functions and flags were added.

```
matteColor
thickness
edgeZScale

LWOBJF_UNSEEN_BY_ALPHA
LWOBJF_MATTE
LWOBJF_MORPH_SURFACES
```

Example

The scenscan, spreadsheet and unwrap SDK samples use the Object Info global.

The following code fragment collects information about the first object.

```
#include <lwserver.h>
#include <lwrender.h>

LWItemInfo *iteminfo;
LWObjectInfo *objinfo;
LWItemID id;
LWTime t = 3.0;          /* seconds */
char *fname;
int npoints, npols;
unsigned int shopts;
double dissolve;

iteminfo = global( LWITEMINFO_GLOBAL, GFUSE_TRANSIENT );
objinfo  = global( LWOBJECTINFO_GLOBAL, GFUSE_TRANSIENT );

if ( iteminfo && objinfo ) {
   id = iteminfo->first( LWI_OBJECT, NULL );
   if ( id ) {
      fname    = objinfo->filename( id );
      npoints  = objinfo->numPoints( id );
      npols    = objinfo->numPolygons( id );
      shopts   = objinfo->shadowOpts( id );
      dissolve = objinfo->dissolve( id, t );
   }
}
```

Panels

Availability LightWave 6.0
Component Layout, Modeler
Header lwpanel.h

The Panels global supplies a set of routines for creating user interface windows from within plug-ins. Also see the related raster and context menu globals.

LWPanels (or the newer XPanels system) gives you a way to create interfaces for your plug-ins that have the LightWave look and feel, using a single code base for all of the platforms LightWave supports.

Creating a non-trivial user interface is a complex task that demands an understanding of both real-time, event-driven programming and of human factors (the ergonomics of the mind). Good design marries function and aesthetics, while a good implementation seeks a balance between responsiveness and power.

This page can't hope to teach any of that, of course, but it's worth mentioning that there's more to this process than the mere building blocks presented here.

If you've programmed interfaces for Microsoft Windows or Apple MacOS, you're familiar with a design method that uses resource files to define dialogs "ahead of time," or statically. When your code runs, you make an operating system call to load your dialog template and use it to create a dialog. Events are either sent to a single, central callback or are pulled by your code from an event queue.

LWPanels doesn't use dialog templates. Dialogs, or *panels*, are built "on the fly" by calling functions that add and position a panel's controls.. Panels are defined by a sequence of function calls rather than a list of directives in a resource file. And events may be sent to many different callbacks. You tell LWPanels where to send them.

If you're accustomed to designing dialogs in a visual environment, the LWPanels approach may take some getting used to.

Other aspects of a panel's life cycle are very similar to those for Windows or MacOS dialogs. You initialize the values of controls before the panel is displayed, and you can read back those values at any time, but in particular after the panel is closed. Interactive controls like sliders generate events while the user is modifying them, and you can respond to those events by changing the values or the appearance of other controls. You can draw on a panel, and blit bitmaps onto it.

If you *haven't* written an interface in another environment, your first reading of this page is likely to be overwhelming. (In fact, that may be true regardless of your previous experience.) Try looking at some of the SDK samples, particularly the ones mentioned by name at the end of this page, to get an initial idea of what's going on, and then refer back to the documentation for an explanation of anything you don't immediately understand.

Handlers whose interfaces use LWPanels will almost always create and display their panels from within the callback you put in the `options` field of the LWInterface structure. (Don't use the `panel` field of that structure; that's for xpanels.)

Global Call

```
   LWPanelFuncs *panf;
   panf = global( LWPANELFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWPanelFuncs.

```
   typedef struct st_LWPanelFuncs {
      LWPanelID   (*create)      (char *, void *);
      void        (*destroy)     (LWPanelID);
      int         (*open)        (LWPanelID, int flags);
      int         (*handle)      (LWPanelID, int);
      void        (*draw)        (LWPanelID, DrMode);
      void        (*close)       (LWPanelID);
      void        (*get)         (LWPanelID, pTag, void *);
      void        (*set)         (LWPanelID, pTag, void *);
      LWControl * (*addControl)  (LWPanelID, char *type,
                                    LWPanControlDesc *, char *label);
      LWControl * (*nextControl) (LWPanelID, LWControlID);
      DrawFuncs   *drawFuncs;
      void        *user_data;
      GlobalFunc  *globalFun;
   } LWPanelFuncs;
```

`panel = `**`create`**`( title, panf )`

Create a panel. This allocates resources for the panel but doesn't display it.

`destroy``( panel )`

Free the resources allocated by `create` and `addControl`. The panel ID is no longer valid after this is called. Panels should not be destroyed while open.

`result = `**`open`**`( panel, flags )`

Display the panel. The panel and its controls must already be created, positioned, sized, initialized and ready to go. The flags are a combination of the following.

`PANF_BLOCKING`

When this is set, the panel is modal, meaning that it will be the only LightWave window that can receive user input. The `open` function will not return until the panel has been closed. Without this flag, the panel is non-modal and the `open` function returns immediately.

`PANF_CANCEL`

Add a Cancel button at the bottom of the panel.

`PANF_FRAME`

Add operating system-specific decoration to the panel window. This flag currently has no effect.

`PANF_MOUSETRAP`

The panel wants mouse input, which will be passed to panel callbacks.

`PANF_PASSALLKEYS`
: The Enter and Escape keys normally close a panel. This flag allows the panel's keyboard callback to handle them instead.

`PANF_ABORT`
: Changes the label of the Cancel button to "Abort". This should be used with the `PANF_CANCEL` flag.

`PANF_NOBUTT`
: Display no buttons (no Continue or Continue/Cancel) at the bottom of the panel.

`PANF_RESIZE`
: Allow resizing of the panel window. When this is set, the panel will accept calls to the `set` function with `PAN_W` and `PAN_H` tags.

`result = `**`handle`**`( panel, flag )`

Process user input for non-modal panels. When the panel is non-modal (opened without the `PANF_BLOCKING` flag), `open` returns immediately. In order to allow user input processing to occur, the plug-in yields control by calling `handle`. If `flag` is 0, `handle` returns as soon as the event queue is empty. It returns 0 if the panel is still open, or -1 if the user has closed it. If `flag` is `EVNT_BLOCKING`, `handle` won't return until the user closes the panel.

`draw``( panel, drmode )`

Redraw the panel. LWPanels performs its own drawing and then calls your panel draw callback, if you've set one. Any of the drawing modes described later for controls are also valid here, but in most cases you'll use `DR_REFRESH`.

`close``( panel )`

Close a non-modal panel. Typically users will close your panels, so you won't need to call this. A closed panel can be reopened later.

`get``( panel, ptag, value )`
`set``( panel, ptag, value )`

Set and retrieve various panel attributes. The `value` is the panel attribute cast as a void *. Many of the attributes are pointers to callback functions, which are described below. The `ptag` identifies the attribute and can be one of the following.

PAN_X, PAN_Y, PAN_W, PAN_H
: Panel position and size in pixels.

PAN_TITLE
: The panel title passed to `create`.

PAN_PANFUN (get)

The LWPanelFuncs pointer passed to `create`.

PAN_FLAGS (get)
: The flags passed to the `open` function.

PAN_USERDATA
: Your data pointer. This is passed as the second argument to all of the panel callbacks.

PAN_MOUSEX, PAN_MOUSEY
: The position of the mouse at the time of the most recent event, relative to the upper left corner of the panel.

PAN_QUALIFIERS (get)
: An integer containing bit flags that provide additional information about the most recent mouse event. These are the same qualifier bits that are passed to mouse callbacks.

PAN_MOUSEBUTTON, PAN_MOUSEMOVE
: Your mouse event callbacks.

PAN_USERKEYS, PAN_USERKEYUPS
: Your keyboard input callbacks.

PAN_USERDRAW
: Your panel draw callback.

PAN_USERACTIVATE, PAN_USEROPEN, PAN_USERCLOSE
: Callbacks that LWPanels calls when the panel is activated (receives input focus from the operating system), opened and closed, respectively.

PAN_VERSION (get)
: The LWPanels API version. Compare this to `LWPANELS_API_VERSION`, which is defined in lwpanel.h.

PAN_RESULT (set)
: Set this to pass results when closing panels manually.

PAN_HOSTDISPLAY (get)
: A pointer to a HostDisplayInfo for the panel.

PAN_TO_FRONT (set)
: Move the panel to the top of the window z-order.

`control = `**`addControl`**`( panel, type, ctrldesc, label )`

Add a control to a panel. Call this after the panel has been created but before it's opened. In practice, you'll seldom call this function explicitly. For each control type, lwpanel.h supplies a macro that calls `addControl` with the proper arguments for that control. Returns a pointer to an LWControl structure, described below. By default, each control is positioned beneath the previous one, and the panel autosizes to fit all of the controls. Controls can be moved after they're created, but internally they remain in the order in which they're created, which for example affects the drawing order.

`control = **nextControl**( panel, control )`

Enumerate the controls that have been added to a panel. Get the first control in the list by passing NULL as the second argument.

`drawFuncs`

A pointer to a DrawFuncs structure, described below.

`user_data`

A place to store whatever you like.

`globalFun`

Set this to the GlobalFunc passed to your activation function.

Panel Callbacks

The LWPanelFuncs `set` function allows you to install a number of panel callbacks that LWPanels will call when certain events occur. You aren't required to install any, so only use them if you need them. All panel callbacks receive as their second argument the value you set for `PAN_USERDATA`.

panhook(panel, userdata)

This is the form of the callback for `PAN_USERACTIVATE`, `PAN_USEROPEN` and `PAN_USERCLOSE`.

pankey(panel, userdata, key)

The form for `PAN_USERKEYS` and `PAN_USERKEYUPS`. For alphanumeric keys, the key code is just the ASCII code. lwpanel.h defines special codes for other keys.

panmouse(panel, userdata, qualifiers, x, y)

For `PAN_MOUSEBUTTON` and `PAN_MOUSEMOVE`. The `x` and `y` mouse positions are relative to the upper left corner of the panel. The `qualifiers` are bit flags.

```
IQ_CTRL
IQ_SHIFT
IQ_ALT
IQ_CONSTRAIN
IQ_ADJUST
```

```
MOUSE_LEFT
MOUSE_MID
MOUSE_RIGHT
MOUSE_DOWN
```

pandraw(panel, userdata, drawmode)

This is for `PAN_USERDRAW`. The `drawmode` is the same as those used for controls and is described later.

Drawing Functions

The `drawFuncs` member of LWPanelFuncs is a structure containing functions that allow you to draw on your panel. See also the Raster Functions global for creating and efficiently displaying bitmaps. You can call these at any time, but in most cases you'll want to be synchronized with the redrawing done by LWPanels, and for that you should limit drawing to panel and control draw callbacks.

```
typedef struct st_DrawFuncs {
   void (*drawPixel)   (LWPanelID, int color, int x, int y);
   void (*drawRGBPixel)(LWPanelID, int r, int g, int b, int x, int y);
   void (*drawLine)    (LWPanelID, int color, int x1, int y1, int x2,
                          int y2);
   void (*drawBox)     (LWPanelID, int color, int x, int y, int w,
                          int h);
   void (*drawRGBBox)  (LWPanelID, int r, int g, int b, int x, int y,
                          int w, int h);
   void (*drawBorder)  (LWPanelID, int indent, int x, int y, int w,
                          int h);
   int  (*textWidth)   (LWPanelID, char *text);
   void (*drawText)    (LWPanelID, char *text, int color, int x,
                          int y);
   const LWDisplayMetrics *(*dispMetrics)();
} DrawFuncs;
```

drawPixel(panel, color, x, y)
drawRGBPixel(panel, r, g, b, x, y)

Draw a pixel. The coordinates are relative to the upper-left corner of the panel. The color is specified as one of the palette colors defined in lwpanel.h or as levels of red, green and blue between 0 and 255.

drawLine(panel, color, x1, y1, x2, y2)

Draw a line connecting the endpoints.

drawBox(panel, color, x, y, w, h)
drawRGBBox(panel, r, g, b, x, y, w, h)

Draw a solid rectangle.

drawBorder(panel, indent, x, y, w, h)

Draw a rectangular border similar to the ones use to mark the borders of controls. The `indent` is the thickness of the border. If `h` is 0, `drawBorder` creates a horizontal divider.

w = **textWidth**(panel, str)

Returns the pixel width of the character string. Use this and the font height information in the LWDisplayMetrics structure to find the rectangular extent of a line of text.

drawText(panel, str, color, x, y)

Render a line of text.

dmet = **dispMetrics**()

Returns an LWDisplayMetrics structure. Except for the screen size and text height, most of this structure is obsolete.

```
typedef struct st_display_Metrics {
   int  width, height;
   int  pixX, pixY;
   int  maxColors, depth;
   int  textHeight;
   int  textAscent;
} display_Metrics;
#define LWDisplayMetrics display_Metrics
```

width, **height**

The size of the screen, in pixels.

pixX, **pixY**

The pixel aspect ratio. In most cases, `pixX == pixY`, indicating square pixels.

maxColors, **depth**

Palette size and bit depth for indexed color displays. `depth` is 0 for true color displays.

textHeight

The height of the LWPanels font in pixels.

textAscent

Ignore this.

Controls

For each control you add to a panel, the LWPanelFuncs `addControl` function returns a pointer to an LWControl.

```
typedef struct st_LWControl {
   void  (*draw)(LWControlID, DrMode);
   void  (*get) (LWControlID, cTag, LWValue *);
   void  (*set) (LWControlID, cTag, LWValue *);
   void  *priv_data;
} LWControl;
```

draw(control, drawmode)

Draw or redraw the control. LWPanels performs its own drawing and calls your control draw callback, if you've set one for this control. The draw mode is one of the following.

DR_RENDER
: Draw the control normally.

DR_GHOST
: Draw the control with a disabled or ghosted appearance.

DR_ERASE
: Erase the control.

DR_REFRESH
: Redraw the control in its current state (normal, ghosted or erased).

get(control, ctag, param)
set(control, ctag, param)

Get and set control attributes, including the value of the control. The `param` is a pointer to an LWValue. The `ctag` identifies the attribute and is one of the following.

CTL_VALUE
: The value of the control.

CTL_LABEL
: The control label passed to `addControl`.

CTL_X, CTL_Y, CTL_W, CTL_H
: The rectangular extent of the control, in pixels. This may include some padding used for control spacing and alignment. `X` and `Y` are relative to the upper left corner of the panel.

CTL_HOTX, CTL_HOTY, CTL_HOTW, CTL_HOTH
: The extent of the control's "hot" area. Generally this excludes the label and any padding but may include the border

decoration.

CTL_LABELWIDTH
: The width of the label in pixels. Because of padding, this may differ from `w - HOTW`.

CTL_MOUSEX, CTL_MOUSEY
: The position of the mouse at the time of the most recent control event, relative to the upper left corner of the panel..

CTL_FLAGS
: Flags marking the current state of the control. `CTLF_DISABLE` indicates the control is disabled, or read-only, but still visible. Disabled controls can still trigger callback events, so that, for example, you can display a message explaining why the control's functionality is unavailable. `CTLF_INVISIBLE` indicates that the control has been erased. `CTLF_GHOST` is a synonym for `CTLF_DISABLED`. These flags will affect the draw mode passed to your control draw callbacks.

CTL_USERDATA
: Your data pointer for this control. This is passed as the second argument to the control callbacks. **Note:** For some control types, calling `set` with this tag (or calling the `CON_SETEVENT` macro) has important side effects. Even if your userdata is NULL for those controls, you'll want to explicitly set it before opening the panel.

CTL_USERDRAW
: Your control draw callback.

CTL_USEREVENT
: Your control event callback.

CTL_PANEL, CTL_PANFUN (get)
: The panel and LWPanelFuncs.

CTL_RANGEMIN, CTL_RANGEMAX
: Slider limits.

CTL_ACTIVATE (set)
: Set the input focus to this control. Only valid for edit fields.

priv_data
: An opaque pointer to data used internally by LWPanels.

Control Callbacks

The LWControl `set` function allows you to install callbacks for custom drawing related to the control and for responding to control events. These callbacks receive as their second argument the value you set for `CTL_USERDATA`.

ctlevent(control, userdata)
    This is the form of the callback for `CTL_USEREVENT`. The context of a control event will vary depending on the type of control. Sliders generate events as the user moves them, and button controls generate an event when they're pressed.

ctldraw(control, userdata, drawmode)
    This is for `CTL_USERDRAW`. The `drawmode` is one of the modes listed for the `draw` function. You can draw anywhere on the panel from within this callback.

In addition to these, several control types have type-specific callbacks. These are described in the following sections.

Macros

The lwpanel.h header file defines a set of macros that hide some of the complexity of using the LWPanels system. These macros are an integral part of the LWPanels API.

The macros require that your source declare and initialize a few variables.

```
   static LWPanControlDesc desc;
   static LWValue
      ival    = { LWT_INTEGER },
      ivecval = { LWT_VINT },
      fval    = { LWT_FLOAT },
      fvecval = { LWT_VFLOAT },
      sval    = { LWT_STRING };
```

These are used as temporary variables while setting up arguments to the panel functions.

Panel Life Cycle

Three macros are provided for creating, displaying and destroying a panel.

PAN_CREATE(pf, title)
    Calls the LWPanelFuncs `create` function.

PAN_POST(pf, pan)

Calls the LWPanelFuncs `open` function with the flags set to `PANF_BLOCKING | PANF_CANCEL | PANF_FRAME`.

PAN_KILL(pf, pan)

Calls the LWPanelFuncs `destroy` function.

Panel Attributes

These macros call the LWPanelFuncs `get` and `set` functions for specific attributes. An advantage of using them is that you can use constants in your source code. The macro takes care of stuffing the value into an appropriate variable and passing a pointer to that variable. The first two arguments to all of these functions are the LWPanelFuncs pointer and the panel.

x = **PAN_GETX**(pf, pan)
y = **PAN_GETY**(pf, pan)
w = **PAN_GETW**(pf, pan)
h = **PAN_GETH**(pf, pan)

Get the position and size of the panel in pixels. `x` and `y` are relative to the upper left corner of the screen.

PAN_SETW(pf, pan, w)
PAN_SETH(pf, pan, h)
MOVE_PAN(pf, pan, x, y)

Set the position and size of the panel. Note that the panel automatically adjusts its size as controls are added to it.

PAN_SETDATA(pf, pan, userdata)
PAN_SETDRAW(pf, pan, drawFn)
PAN_SETKEYS(pf, pan, keyFn)

Set the `PAN_USERDATA` and the panel draw and key event callbacks.

version = **PAN_GETVERSION**(pf, pan)

Returns the version of LWPanels.

Creating Controls

After creating a panel and before displaying it, your code calls these macros to populate the panel with the elements of your interface. The first three arguments to all of these macros are

- the LWPanelFuncs returned by the global call
- the LWPanelID returned by the `create` call (or the `PAN_CREATE` macro)
- a string that will be used to label the control

All of these macros ultimately call the LWPanelFuncs `addControl` function and return a pointer to LWControl for the newly created control.

Edit fields

c = **INT_CTL**(pf, pan, label)
c = **INTRO_CTL**(pf, pan, label)
c = **IVEC_CTL**(pf, pan, label)
c = **IVECRO_CTL**(pf, pan, label)
    Integer edit fields. `RO` is read-only, and `VEC` is a group of three fields.

c = **FLOAT_CTL**(pf, pan, label)
c = **FLOATRO_CTL**(pf, pan, label)
c = **FVEC_CTL**(pf, pan, label)
c = **FVECRO_CTL**(pf, pan, label)
c = **DIST_CTL**(pf, pan, label)
c = **DVEC_CTL**(pf, pan, label)
    Floating point edit fields. The distance controls handle unit inputs and conversions.

c = **STR_CTL**(pf, pan, label, cw)
c = **STRRO_CTL**(pf, pan, label, cw)
    String edit fields. The `cw` argument is the number of characters that should be visible, or the width of the control in characters. The string itself can be longer or shorter than this. If the fixed width of the integer and floating point controls isn't suitable, you can of course use the string controls and do the numeric conversion yourself.

Buttons

c = **BUTTON_CTL**(pf, pan, label)

c = **WBUTTON_CTL**(pf, pan, label, w)

Simple buttons. In order for these to do anything, you'll need to set an event callback that responds when the button is pressed. The `WBUTTON` version accepts a width in pixels.

c = **BOOL_CTL**(pf, pan, label)
c = **BOOLBUTTON_CTL**(pf, pan, label)
c = **WBOOLBUTTON_CTL**(pf, pan, label, w)

The first of these displays a checkmark, while the other two are buttons that are displayed in selected or unselected states. The underlying value is an integer set to 0 or 1.

Sliders and mouse feedback

c = **SLIDER_CTL**(pf, pan, label, w, min, max)
c = **UNSLIDER_CTL**(pf, pan, label, w, min, max)

This kind of slider is a thumb button that moves in or along a horizontal track. It has an associated integer edit field. The `UN` version is "unbounded," meaning that values outside the `min`, `max` range can be entered in the edit field. The width is in pixels.

c = **HSLIDER_CTL**(pf, pan, label, w, min, max)
c = **VSLIDER_CTL**(pf, pan, label, h, min, max)

Horizontal and vertical sliders, without an associated edit field. The width or height is in pixels.

c = **MINISLIDER_CTL**(pf, pan, label, w, min, max)
c = **PERCENT_CTL**(pf, pan, label)
c = **ANGLE_CTL**(pf, pan, label)

A minislider is a small button that captures mouse drag but doesn't move. It has an associated integer edit field. The `w` argument is the visible width of the edit field in pixels. The `PERCENT` control is a minislider with a floating point edit field that displays the percent (%) character after the number. `ANGLE` also has a floating point edit field. The value is displayed to the user in degrees, but plug-ins get and set it in radians.

c = **DRAGBUT_CTL**(pf, pan, label, w, h)
c = **VDRAGBUT_CTL**(pf, pan, label)

c = **HDRAGBUT_CTL**(pf, pan, label)

Drag buttons are minisliders with no associated edit field.

c = **AREA_CTL**(pf, pan, label, w, h)
c = **DRAGAREA_CTL**(pf, pan, label, w, h)

These create a rectangle with a border. AREA controls generate mouse click events, and DRAGAREA controls generate both click and drag events. For AREA, the CTL_MOUSEX and CTL_MOUSEY values contain click coordinates relative to the upper left corner of the control. For DRAGAREA, MOUSEX and MOUSEY are relative to the upper left corner of the panel. Use the GETV_IVEC macro in the DRAGAREA event callback to get an array of three integers containing control-relative mouse coordinates.

Multiple choice

c = **HCHOICE_CTL**(pf, pan, label, choices)
c = **VCHOICE_CTL**(pf, pan, label, choices)

An array of mutually exclusive boolean buttons.

c = **TABCHOICE_CTL**(pf, pan, label, choices)

Similar to HCHOICE and VCHOICE, but drawn to look like file folder tabs. These are generally used to switch between several sets of controls occupying the same space, like flipping to different tabbed notebook pages. You're responsible for erasing and drawing the appropriate sets of controls affected by tabbing.

c = **POPUP_CTL**(pf, pan, label, choices)
c = **WPOPUP_CTL**(pf, pan, label, choices, w)
c = **POPDOWN_CTL**(pf, pan, label, choices)

These create scrolling popup menus. The choices argument is a NULL-terminated string array, and the value of the control is a 0-based index into this array. The W version lets you set the width in pixels. POPUPs display the label to the left of the button and the current selection on the button face, and when opened, the position of the menu window is shifted so that the current selection is under the mouse cursor. POPDOWNs display the label on the button face and always open with the first menu item under the cursor.

c = **CUSTPOPUP_CTL**(pf, pan, label, w, nameFn, countFn)

Like `WPOPUP`, but uses callbacks to fill in the menu rather than a static string array. The menu can therefore be different each time the user opens it. The value of the control is a 0-based index into the current list of menu items. The callbacks are

```
int count( void *userdata )
char *name( void *userdata, int index )
```

The `userdata` is the `CTL_USERDATA` for the control. If you have more than one custom popup that uses the same callbacks, you can use this to distinguish between them. `count` returns the number of menu items, and `name` returns an item, given a 0-based index.

c = **ITEM_CTL**(pf, pan, label, globalFn, itemtype)
c = **WITEM_CTL**(pf, pan, label, globalFn, itemtype, w)
c = **PIKITEM_CTL**(pf, pan, label, globalFn, itemtype, w)

These are `POPUP`s that display a list of scene items. The `globalFn` is the GlobalFunc passed to your activation function. If you set the LWPanelFuncs `globalFun` field, you can get it from there. The control value is an LWItemID cast as an int. In addition to the item types listed on the Item Info page, lwpanel.h defines `LWI_IMAGE`, for a list of images, and `LWI_ANY`, for a list of items of all types. Remember to include lwrender.h. `PIKITEM` behaves like `POPDOWN`.

c = **CHANNEL_CTL**(pf, pan, label, w, h);

Displays a tree containing the channels currently in the scene. The control value is an LWChannelID cast as an int. You'll need lwenvel.h and the Channel Info global to set and make use of this value.

c = **LISTBOX_CTL**(pf, pan, label, w, ch, nameFn, countFn)

A listbox is a static rectangle containing a menu with a scrollbar. `ch` is the height of the menu area of the listbox in text lines (the number of visible menu items). The callbacks are of the same form as those for `CUSTPOPUP`. The value of the control is a 0-based index into the current list. After the control has been created, you must call the `CON_SETEVENT` macro, or the control `set` function with the `CTL_USERDATA` tag, even if your userdata is NULL. `LISTBOX` controls rely on this to perform some internal initialization.

c = **MULTILISTBOX_CTL**(pf, pan, label, w, ch, nameFn, countFn, columnFn)

Like a listbox, but divides the text of each item into multiple columns. The `countFn` callback is the same as those for `LISTBOX` and `CUSTPOPUP`. The other two are of the following form.

```
char *mname( void *userdata, int index, int column )
int colwidth( void *userdata, int index )
```

The name callback returns a string, given 0-based indexes for the list position and column. The column callback returns the width of each column in pixels. You can have up to ten columns. Return 0 when the column index is greater than or equal to the number of columns you want.

After the control has been created, you must call the `CON_SETEVENT` macro, or the control `set` function with the `CTL_USERDATA` tag, even if your userdata is NULL. `MULTILISTBOX` controls rely on this to perform some internal initialization.

c = **TREE_CTL**(pf, pan, label, w, h, infoFn, countFn, childFn)

A tree control is like a listbox, but with the menu items organized hierarchically. Child nodes of the tree can be revealed, hidden and sometimes moved by the user. The value of a tree control is a pointer to a node in your tree data, cast as an int. Trees don't prescribe the internal form of your tree data, but you have to be able to answer the questions about that data asked by the callbacks, which look like this.

void *child(void *userdata, void *node, int i)

Returns a pointer to the i-th child of the node. `node` is a pointer returned by a previous call to this callback, or NULL for the root.

```
int count( void *userdata, void *node )
```

Returns the number of child nodes for a given node.

char *info(void *userdata, void *node, int *flags)

Returns the name of the node. This is what is displayed in the tree control. If `flags` is non-zero, store the `flags` value in your node data, and if it's 0, retrieve it from your data and put it into `flags`.

void move(void *userdata, void *node, void *parent, int i)

Called when the user moves a node. The node becomes the `i`-th child of the `parent` node. This isn't in the `TREE_CTL` macro's argument list. `TREE_CTL` sets this to NULL, which prevents the user from moving your nodes. If you want to allow the user to move your nodes, use the code in the body of the `TREE_CTL` macro to create the control, putting your move callback in `desc.tree.moveFn`.

Color

c = **RGB_CTL**(pf, pan, label)
c = **MINIRGB_CTL**(pf, pan, label)
c = **MINIHSV_CTL**(pf, pan, label)
c = **RGBVEC_CTL**(pf, pan, label)

RGB (and HSV) levels for all of these controls are integers in the range 0 to 255. You can offer your users more sophisticated color selection by creating a button that calls the current colorpicker.

Files and directories

c = **FILE_CTL**(pf, pan, label, cw)
c = **LOAD_CTL**(pf, pan, label, cw)
c = **SAVE_CTL**(pf, pan, label, cw)
c = **DIR_CTL**(pf, pan, label, cw)

These combine a string edit field with a button that opens the file dialog. `cw` is the width of the edit field in characters. `FILE` is an older control type preserved for backward compatibility.

c = **FILEBUTTON_CTL**(pf, pan, label, w)
c = **LOADBUTTON_CTL**(pf, pan, label, w)
c = **SAVEBUTTON_CTL**(pf, pan, label, w)
c = **DIRBUTTON_CTL**(pf, pan, label, w)

Just the button for opening the file dialog. The label appears inside the button.

Drawing

c = **TEXT_CTL**(pf, pan, label, strings)

Use this to put static lines of text on the panel.

c = **BORDER_CTL**(pf, pan, label, w, h)
For drawing borders. If `h` is 0, the border is a horizontal divider.

c = **CANVAS_CTL**(pf, pan, label, w, h)
A bordered rectangle for convenient drawing. The width and height don't include the border, so the rectangle `(0, 0, w-1, h-1)` (relative to the control's `HOTX` and `HOTY`) lies inside the border.

c = **OPENGL_CTL**(pf, pan, label, width, height)
This creates and initializes an OpenGL window. LWPanels takes care of the platform specific setup for the window. You can draw in this window using standard OpenGL function calls during your event and draw callbacks for the control.

XPanels

c = **XPANEL_CTL**(pf, pan, label, xpanel)
This creates an xpanel window on your panel. Anything you can put into an xpanel can be put into an xpanel control.

Control Values

These macros call the LWControl `get` and `set` functions with the `CTL_VALUE` attribute, which is how you initialize and read back the values of your controls.

```
SET_STR( ctl, buf, buflen )        GET_STR( ctl, buf, buflen )
SET_INT( ctl, n )                  GET_INT( ctl, n )
SET_FLOAT( ctl, f )                GET_FLOAT( ctl, f )
SET_IVEC( ctl, x, y, z )           GET_IVEC( ctl, x, y, z )
SETV_IVEC( ctl, nv )               GETV_IVEC( ctl, nv )
SET_FVEC( ctl, x, y, z )           GET_FVEC( ctl, x, y, z )
SETV_FVEC( ctl, fv )               GETV_FVEC( ctl, fv )
```

Control Attributes

These macros get and set other control attributes.

CON_X(ctl)
CON_Y(ctl)

CON_W(ctl)
CON_H(ctl)
Returns the position and size of the control.

CON_HOTX(ctl)
CON_HOTY(ctl)
CON_HOTW(ctl)
CON_HOTH(ctl)
Returns the position and size of the control's "hot" rectangle.

CON_LW(ctl)
Returns the label width.

CON_PAN(ctl)
CON_PANFUN(ctl)
Returns the panel the control belongs to and the LWPanelFuncs pointer.

CON_MOUSEX(ctl)
CON_MOUSEY(ctl)
Returns mouse coordinates for the most recent mouse event.

MOVE_CON(ctl, x, y)
Set the positon of the control relative to the upper left corner of the panel.

CON_SETEVENT(ctl, eventFn, userdata)
Set the control's event function and CTL_USERDATA. **Note:** For some control types, calling this macro (or the control's `set` function with the CTL_USERDATA tag) has important side effects. Even if your userdata is NULL for those controls, you'll want to explicitly set it before opening the panel.

ERASE_CON(ctl)
REDRAW_CON(ctl)
GHOST_CON(ctl)
RENDER_CON(ctl)
UNGHOST_CON(ctl)
Draw or redraw the control.

ACTIVATE_CON(ctl)

Activate the control (give the control the input focus). This is only valid for edit field controls.

History

In LightWave 7.0, `LWPANELS_API_VERSION` was incremented to 19 and the `PANF_NOBUTT` and `PANF_RESIZE` flags were added.

Example

At least ten of the SDK samples use the LWPanels system. The panctl sample exercises LWPanels by creating an instance of every supported control type. It also demonstrates event handling for some of the interactive controls. The hello sample is the simplest example. It creates a panel with a single string control. The complete life cycle of this panel is repeated in the following code fragment.

```
#include <lwserver.h>
#include <lwpanel.h>

LWPanelFuncs *panf;
LWPanelID panel;
LWControl *ctl;
LWPanControlDesc desc;
LWValue sval = { LWT_STRING };
char edit[ 80 ] = "This is an edit field.";

panf = global( LWPANELFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( !panf ) return AFUNC_BADGLOBAL;

panel = PAN_CREATE( panf, "Hello World!" );
if ( !panel ) return AFUNC_BADGLOBAL;
ctl = STR_CTL( panf, panel, "Edit Me", 40 );
SET_STR( ctl, edit, sizeof( edit ));

if ( panf->open( panel, PANF_BLOCKING | PANF_CANCEL ))
   GET_STR( ctl, edit, sizeof( edit ));

PAN_KILL( panf, panel );
```

Particle Services

Availability LightWave 6.0
Component Layout
Header lwprtcl.h

The particles global returns functions that allow you to create particle systems and to read and write particle data. Particles are typically used by volumetric renderers to define the extent and local density of a volume. LightWave's Hypervoxels, for example, uses them this way.

The host side of the particles global manages a database of particle systems. The global supplies methods for adding, deleting and reading particle data in the database. Having such a database allows one plug-in to create particle systems that others can later use.

Global Call

```
   LWPSysFuncs *psysf;
   psysf = global( LWPSYSFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWPSysFuncs.

```
   typedef struct st_LWPSysFuncs {
      LWPSysID   (*create)      (int flags, int type);
      int        (*destroy)     (LWPSysID);
      int        (*init)        (LWPSysID, int np);
      void       (*cleanup)     (LWPSysID);
      void       (*load)        (LWPSysID, LWLoadState *);
      void       (*save)        (LWPSysID, LWSaveState *);
      int        (*getPCount)   (LWPSysID);
      void       (*attach)      (LWPSysID, LWItemID);
      void       (*detach)      (LWPSysID, LWItemID);
      LWPSysID * (*getPSys)     (LWItemID);
      LWPSBufID  (*addBuf)      (LWPSysID, LWPSBufDesc);
      LWPSBufID  (*getBufID)    (LWPSysID, int bufFlag);
      void       (*setBufData)  (LWPSBufID, void *data);
      void       (*getBufData)  (LWPSBufID, void *data);
      int        (*addParticle) (LWPSysID);
      void       (*setParticle) (LWPSBufID, int index, void *data);
      void       (*getParticle) (LWPSBufID, int index, void *data);
      void       (*remParticle) (LWPSysID, int index);
   } LWPSysFuncs;
```

You can allocate, initialize and read particle data either individually (one particle at a time) or all at once. Which approach you take will depend primarily on where the data comes from and how you use it.

`psys =` **`create`**`( flags, type )`

Create a particle system. The flags indicate which buffers should be allocated for the particles and can be any of the following, combined using bitwise-or.

`LWPSB_POS`	position	`float[3]`
`LWPSB_SIZ`	size	`float`
`LWPSB_SCL`	scale	`float[3]`
`LWPSB_ROT`	rotation	`float[3]`
`LWPSB_VEL`	velocity	`float[3]`
`LWPSB_AGE`	age	`float`
`LWPSB_FCE`	force	`float`
`LWPSB_PRS`	pressure	`float`
`LWPSB_TMP`	temperature	`float`
`LWPSB_MAS`	mass	`float`
`LWPSB_LNK`	link to particle (for trails)	`int`
`LWPSB_ID`	ID (unique index for the particle)	`int`
`LWPSB_ENB`	enable state (dead/alive/limbo)	`char`
`LWPSB_RGBA`	display color and alpha	`char[4]`
`LWPSB_CAGE`	time since last collision	`float`

The particle type can be either `LWPST_PRTCL` (single points) or `LWPST_TRAIL` (line segments). `LWPST_TRAIL` particle systems should include an `LWPSB_LNK` buffer for the second point in each trail.

`result =` **`destroy`**`( psys )`

Free the particle system.

`error =` **`init`**`( psys, nparticles )`

Allocate memory for the particles. This is equivalent to calling the `addParticle` function `nparticles` times.

`cleanup``( psys )`

Frees the memory allocated by `init` and `addParticle`.

`load``( psys, loadstate )`

Read the particle system data from a file. This will typically be used by handler load callbacks.

`save( psys, savestate )`
: Write the particle system data to a file. This will typically be used by handler save callbacks.

`count = getPCount( psys )`
: Returns the number of particles.

`attach( psys, item )`
: Associate a particle system with an item in the scene, usually an object. More than one particle system can be attached to an item, and more than one item can share the same particle system. Attaching a particle system to an item makes it possible for others, Hypervoxels in particular, to use the `getPSys` function to find it.

`detach( psys, item )`
: Remove the association between a particle system and an item.

`psys_array = getPSys( item )`
: Returns a NULL-terminated array of particle system IDs that have been associated with the item by the `attach` function.

`psbuf = addBuf( psys, bufdesc )`
: Add a custom per-particle buffer. Call this before any calls to `init` or `addParticle`. (The predefined buffer types should be added when `create` is called.) The structure used to define the buffer is described below. The buffer ID returned by this function can be used with the functions that get and set buffer data.

`psbuf = getBufID( psys, bufbit )`
: Returns a buffer ID for one of the predefined buffers. This is used with the functions that get and set buffer data. The second argument is one of the buffer flags passed to `create`.

`setBufData( psbuf, data )`
: Set the buffer values for all particles. The data is an array of the appropriate type for the buffer, with a number of entries equal to the number of particles. Use `setParticle` to set the buffer data for one particle at a time.

`getBufData( psbuf, data )`
: Get the buffer values for all particles. Use `getParticle` to get the buffer data for one particle at a time.

`index = addParticle( psys )`

Add a particle.

`setParticle``( psbuf, index, data )`
: Set the buffer value for a particle. Particles are numbered from 0 to `getPCount` - 1 in the order in which they're added.

`getParticle``( psbuf, index, data )`
: Get the buffer value for a particle.

`remParticle``( psys, index )`
: Remove a particle.

Particle Buffers

The `addBuf` function uses a buffer descriptor to define the buffer to be added.

```
   typedef struct st_LWPSBufDesc {
      const char *name;
      int         dataType;
      int         dataSize;
   } LWPSBufDesc;
```

name
: A string that names the buffer. In the future, this may allow users or plug-ins to refer to the buffer by name.

dataType
: The data type of the data in the buffer.

```
LWPSBT_FLOAT
LWPSBT_INT
LWPSBT_CHAR
```

dataSize
: The number of values per particle in the buffer (and not the number of bytes).

Example

The particle sample is a displacement handler that demonstrates the use of the particle system global. Its operation is similar to that of the HVParticle displacement handler Hypervoxels adds to objects that lack particle systems.

Preview Functions

Availability LightWave 6.0
Component Layout, Modeler
Header lwpreview.h

The Preview Functions global is the plug-in API for LightWave's VIPER (Versatile Interactive Preview Render) window. VIPER allows users to preview the effects of shader, environment, volumetric, pixel filter and image filter plug-ins. It uses image buffers from the most recent test render to generate a reduced-size rendering of the scene, and it can composite this with your plug-in's output while the user changes your parameters.

The previewer is an extension of your plug-in's interface. The API supplies functions that let you subscribe (install), set the context (tell VIPER that your interface is the active one), open the VIPER window, give VIPER your rendering callbacks, and get the prerendered image and information about the camera.

This is a *low-level* API, and you may never need to use it. Beginning with LightWave 7.0, VIPER can preview your plug-in automatically, without your intervention. It switches contexts transparently and calls your regular handler callbacks to render the preview. The rendering of the preview occurs whenever you call the Instance Update global.

Global Call

```
   LWPreviewFuncs *pvf;
   pvf = global( LWPREVIEWFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWPreviewFuncs.

```
   typedef struct st_LWPreviewFuncs {
      PvContextID (*subscribe)  (char *title, void *userData,
                                    closeFunc *);
      void        (*unsubscribe)(PvContextID);
      void        (*open)       (PvContextID);
      void        (*close)      (void);
      int         (*isOpen)     (void);
      void        (*setContext) (PvContextID);
      void        (*setClick)   (PvContextID, ClickFunc *);
      void        (*setRender)  (PvContextID, void *renderData,
                                    InitFunc *, CleanupFunc *,
                                EvaluateFunc *);
      void       (*setOptions) (PvContextID, char **list,
                                OptionsFunc *, int selection);
      void       (*startRender)(PvContextID);
      void       (*stopRender) (PvContextID);
      void       (*getPixel)   (PvSample *pixel);
      LWImageID  (*getBitmap)  (int *width, int *height);
      LWItemID   (*getCamera)  (double pos[3], double rot[3],
                                double *zoomFactor);
      void       (*getView)    (int *width, int *height,
                                double *pixelAspect);
      void       (*setPreset)  (PvContextID, presetFunc *);
   } LWPreviewFuncs;
```

preview = **subscribe**(title, userdata, close_func)

Obtain a PvContextID that identifies your plug-in instance to the preview system. Generally you'll call this from your interface activation function. The title is the string that should appear in the preview window's title bar when the previewer is set to your context. The user data is passed to your click, options and close callbacks.

unsubscribe(context)

Invalidate your context ID and free resources allocated by `subscribe`.

open(context)

Open the preview window.

close()

Close the preview window.

isopen = **isOpen**()

Returns TRUE if the preview window is open.

setContext(context)

Set the preview window's context. After this, the interface and rendering callbacks associated with the context ID will be the ones the previewer calls when responding to the user and generating an image.

setClick(context, click_func)

Set the callback that will be called when the user clicks on the preview image.

setRender(context, render_data, init_func, cleanup_func, eval_func)

Set the callbacks that will be called when the previewer renders its display. The render data will be passed to each of the callbacks. Typically, it's your instance data, and the callbacks call your standard handler functions.

setOptions(context, list, options_func, selection)
Set the options that will appear in the Options popup when your context is the active one. This includes a NULL-terminated array of strings, a callback that's called when an option is selected by the user, and the index of the option that should initially appear selected.

startRender(context)
Force the previewer to render an image.

stopRender(context)
Interrupt any rendering being done by the previewer.

getPixel(pixel)
Get information about a pixel in the previewer's prerendered buffers. Fill in the `x` and `y` fields of the PvSample structure for the position of the pixel in the preview image. The previewer will return information about the pixel in the other fields.

image = **getBitmap**(width, height)
Returns an image ID for the previewer's RGBA buffers. You can use this with the Image List global to query the image. The previewer writes the image dimensions in the `width` and `height` arguments.

camera = **getCamera**(pos, rot, zoom)
Get information about the state of the camera at the time the previewer's buffers were generated. You can get more detailed information from the Item Info and Camera Info globals, but it may not match the image in the previewer, since the user may have moved the camera or changed its settings after the previewer image was created.

getView(width, height, pixel_aspect)
Get information about the size and pixel aspect of the previewer image.

setPreset(context, preset_func)
  Set the callback that will be called when the user wants to create a shelf preset for your plug-in's settings.

Pixel Sample

The `getPixel` function and the click and evaluate callbacks store information about a pixel in a PvSample.

```
   typedef struct st_PvSample {
      int         x, y;
      float       rgbaz[5];
      LWMicropol  mp;
   } PvSample;
```

x, y
  The pixel coordinates.

rgbaz
  The red, green, blue, alpha and depth value at the pixel.

mp
  An LWMicropol structure describing the geometry visible in the pixel. See the explanation of this structure on the Texture Functions page.

  The previewer can only fill in the fields for which it knows the values. These include `gNorm`, `wNorm` (the same as `gNorm` in this case), `oPos`, `wPos`, `oAxis`, `wAxis` and the `verts` and `weights` arrays. If the display and render subdivision levels differ, the point IDs in the `verts` array may not be valid (the previewer has the render mesh, but the plug-in may have the display mesh).

Callbacks

The previewer uses callbacks both for rendering and to allow your plug-in to respond to user actions.

Interface

```
   typedef int  clickFunc(int count, void *userData, PvSample *pixel);
   typedef void optionsFunc(int option, void *userData);
   typedef void presetFunc(void *userData, LWImageID image);
   typedef void closeFunc(void *userData);
```

The click callback tells you that the user has clicked on the preview image and gives you information about the pixel. The options callback is called when the user has selected an option from the custom Options pop-up on the preview window. The first argument is an index into the array of strings you passed to `setOptions`. The close callback is called when the user closes the preview window.

The preset callback tells you that the user wants to add a preset to the shelf for your plug-in's settings. The image is the same one returned by `getBitmap`. Use the Shelf Functions global to add the preset.

The user data for all of these is the pointer you passed to `subscribe`.

Rendering

```
   typedef int  initFunc(void *renderData, int manual);
   typedef void cleanupFunc(void *renderData);
   typedef int  evaluateFunc(void *renderData, int w, int h,
                   PvSample *pixel);
```

Your preview init function should perform the same kinds of operations that your handler `init` and `newTime` callbacks perform, and your preview cleanup is analogous to your handler `cleanup`. The second argument to the init callback is TRUE if the user explicitly requested the render (by clicking on a button in the previewer's window).

The evaluate callback receives the width and height of the preview image and a PvSample for the pixel to be evaluated. Your evaluate writes new values in the `rgbaz` field of the PvSample. The PvSample's LWMicropol is read-only (writing to it has no effect).

History

The `setPreset` function was added in LightWave 7.0, but `LWPREVIEWFUNCS_GLOBAL` was *not* incremented. Before calling `setPreset`, you can use the Product Info global to determine whether you're running in a version of LightWave prior to 7.0.

Product Info

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The product info global returns the product (LightWave or Inspire), its major and minor version numbers, and its build number. Build numbers can be used to distinguish between bug-fix revisions for which the major and minor version numbers weren't incremented, which can happen in particular when the revision only affects one platform. See also the compatibility discussion.

Global Call

```
unsigned long prodinfo;
prodinfo = ( unsigned long ) global( LWPRODUCTINFO_GLOBAL,
   GFUSE_TRANSIENT );
```

The global function ordinarily returns a `void *`, so this should be cast to an integer type to get the return value.

The product ID is in the low four bits of the return value. The build number is in bits 15 - 4, the minor version number is in bits 19 - 16, and the major revision number is in bits 23 - 20. All of these components can be extracted using macros defined in `lwhost.h`.

```
product = prodinfo & LWINF_PRODUCT;
major = LWINF_GETMAJOR( prodinfo );
minor = LWINF_GETMINOR( prodinfo );
build = LWINF_GETBUILD( prodinfo );
```

Currently, the product can be `LWINF_PRODLWAV` (LightWave), `LWINF_PRODINSP3D` (Inspire), or `LWINF_PRODOTHER`.

Raster Services

Availability LightWave 6.0
Component Layout, Modeler
Header lwpanel.h

Raster services are a set of functions for manipulating bitmaps (rasters) used as interface elements in your panels. A raster isn't visible until you call `blitPanel` to draw it on a panel.

Global Call

```
   LWRasterFuncs *rastf;
   rastf = global( LWRASTERFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWRasterFuncs.

```
   typedef struct st_LWRasterFuncs {
      void       (*drawPixel)   (LWRasterID, int color, int x, int y);
      void       (*drawRGBPixel)(LWRasterID, int r, int g, int b,
                                    int x, int y);
      void       (*drawLine)    (LWRasterID, int color, int x, int y,
                                    int x2, int y2);
      void       (*drawBox)     (LWRasterID, int color, int x, int y,
                                    int w, int h);
      void       (*drawRGBBox)  (LWRasterID, int r, int g, int b,
                                    int x, int y, int w, int h);
      void       (*eraseBox)    (LWRasterID, int x, int y, int w,
                                    int h);
      void       (*drawBorder)  (LWRasterID, int indent, int x, int y,
                                    int w, int h);
      void       (*drawText)    (LWRasterID, char *, int color, int x,
                                    int y);
      LWRasterID (*create)      (int w, int h, int flags);
      void       (*destroy)     (LWRasterID);
      void       (*blitPanel)   (LWRasterID, int x, int y, LWPanelID,
                                    int x, int y, int w, int h);
   } LWRasterFuncs;
```

drawPixel(raster, color, x, y)
drawRGBPixel(raster, r, g, b, x, y)
    Set the color of a pixel. The color is either one of the predefined colors in lwpanel.h or 8-bit levels of red, green and blue.

drawLine(raster, color, x, y, x2, y2)
    Draw a line from (`x`, `y`) to (`x2`, `y2`) inclusive.

drawBox(raster, color, x, y, w, h)
drawRGBBox(raster, r, g, b, x, y, w, h)
eraseBox(raster, x, y, w, h)

Draw a filled box. The color for `eraseBox` is the panel's background color.

drawBorder(raster, indent, x, y, w, h)

Draw a LightWave-style rectangular border. The `indent` controls the border thickness.

drawText(raster, text, color, x, y)

Draw a line of text. The coordinates specify the upper left corner of the first character cell. You can get information about the pixel dimensions of the text using the LWDisplayMetrics functions returned by the Panels global.

raster = **create**(w, h, flags)

Create a raster. No flags are currently defined, so `flags` should be 0.

destroy(raster)

Free a raster.

blitPanel(raster, srcx, srcy, panel, dstx, dsty, w, h)

Transfer a raster image, or part of one, to the surface of a panel. `srcx` and `srcy` are the upper left corner of the source rectangle, while `dstx` and `dsty` are the upper left corner of the destination, relative to the upper left corner of the panel window.

Example

The binview SDK sample makes extensive use of the raster functions to create and display its own fixed-pitch font glyphs. It also uses a raster to display an icon.

Scene Info

Availability LightWave 6.0 **Component** Layout
Header lwrender.h

The scene info global returns information about the current scene. This information is read-only and reflects the state of the scene at the time the global function is called. You can set these parameters using commands.

Global Call

```
LWSceneInfo *sceneinfo;
sceneinfo = global( LWSCENEINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWSceneInfo.

```
typedef struct st_LWSceneInfo {
   const char *name;
   const char *filename;
   int         numPoints;
   int         numPolygons;
   int         renderType;
   int         renderOpts;
   LWFrame     frameStart;
   LWFrame     frameEnd;
   LWFrame     frameStep;
   double      framesPerSecond;
   int         frameWidth;
   int         frameHeight;
   double      pixelAspect;
   int         minSamplesPerPixel;
   int         maxSamplesPerPixel;
   int         limitedRegion[4];
   int         recursionDepth;
   LWItemID   (*renderCamera) (LWTime);
   int         numThreads;
   const char *animFilename;
   const char *RGBPrefix;
   const char *alphaPrefix;
} LWSceneInfo;
```

`name`
User's name for the scene.

`filename`
Filename of the scene file.

`numPoints`, **`numPolygons`**

Total number of points and polygons for all the objects in the scene.

`renderType`
: The render type, which can be one of the following.

```
LWRTYPE_WIRE
LWRTYPE_QUICK
LWRTYPE_REALISTIC
```

`renderOpts`
: This is a combination of bits for different rendering options. The bit flags are

```
LWROPT_SHADOWTRACE
LWROPT_REFLECTTRACE
LWROPT_REFRACTTRACE
LWROPT_FIELDS
LWROPT_EVENFIELDS
LWROPT_MOTIONBLUR
LWROPT_DEPTHOFFIELD
LWROPT_LIMITEDREGION
LWROPT_PARTICLEBLUR
LWROPT_ENHANCEDAA
LWROPT_SAVEANIM
LWROPT_SAVERGB
LWROPT_SAVEALPHA
```

`LWROPT_EVENFIELDS` is set only if field rendering is on and the first line of the output image is from the field that comes first in time.

`frameStart`, **`frameEnd`**
: The numbers of the first and last frame defined for the scene. These are the rendering limits, not to be confused with the limits set by the user for previews (which you can get from the interface info global).

`frameStep`
: The step size, in frames, during rendering (the user setting for the Frame Step).

`framesPerSecond`
: Number of frames per playback second. This will ordinarily be 24 for film, 30 for NTSC video and 25 for PAL video. Note that this is the number of frames, not fields.

`frameWidth`, **`frameHeight`**
: Rendered image size in pixels.

`pixelAspect`
: The aspect ratio of the pixels in the image, expressed as width/height. Values greater than 1.0 mean short wide pixels and values less than 1.0 mean tall thin pixels.

minSamplesPerPixel, **maxSamplesPerPixel**

Limits on the number of samples per pixel in the final image. Because of different rendering techniques and adaptive sampling it is impossible to compute a precise number of antialiasing samples at any pixel, but this gives a range for the current rendering options.

limitedRegion

The extents of the limited region area, in pixels. The extents are given in the order x0, y0, x1, y1.

recursionDepth

The maximum recursion depth for raytracing.

camID = **renderCamera**(time)

Returns the item ID of the camera that will render the frame at the specified time.

numThreads

The number of threads of execution that will be used during rendering.

animFilename

The name of the current animation file.

RGBPrefix

The current RGB file saving prefix

AlphaPrefix

The current RGB file saving prefix

Example

This code fragment calculates the running time and frame aspect.

```
#include <lwserver.h>
#include <lwrender.h>

LWSceneInfo *lwsi;
double duration, frameAspect;

lwsi = global( LWSCENEINFO_GLOBAL, GFUSE_TRANSIENT );

if ( lwsi ) {
   duration = ( lwsi->frameEnd - lwsi->frameStart + 1 )
      / lwsi->framesPerSecond;
   frameAspect = lwsi->pixelAspect * lwsi->frameWidth
      / lwsi->frameHeight;
}
```

Scene Objects

Availability LightWave 6.0
Component Layout, Modeler
Header lwmeshes.h

The scene objects global gives plug-ins access to the internal representation of each object file loaded into the current scene in Layout or the object database in Modeler. Functions are provided for reading object geometry and the vertex map list.

Although vertex maps are stored in object files, LightWave's internal vmap list isn't object-specific, and this has several consequences. If a vmap of the same name and type is stored in two different object files, LightWave creates a single entry in the vmap list when both files are loaded. The vmap is shared by the two objects. If both objects are then removed, the vmap is *not* removed from the list. And the only way to determine whether a vmap affects a given object is to test all of its vertices, using the Mesh Info `pntVGet` function, to see whether any of them is mapped.

Global Call

```
   LWObjectFuncs *objfunc;
   objfunc = global( LWOBJECTFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWObjectFuncs.

```
   typedef struct st_LWObjectFuncs {
      int           (*numObjects)  (void);
      const char * (*filename)    (int obj);
      int           (*maxLayers)   (int obj);
      int           (*layerExists) (int obj, int lnum);
      void          (*pivotPoint)  (int obj, int lnum, LWFVector pos);
      LWMeshInfo * (*layerMesh)   (int obj, int lnum);
      int           (*numVMaps)    (LWID);
      const char * (*vmapName)    (LWID, int index);
      int           (*vmapDim)     (LWID, int index);
      LWID          (*vmapType)    (int index);
      const char * (*layerName)   (int obj, int lnum);
      int           (*layerVis)    (int obj, int lnum);
      const char * (*userName)    (int obj);
      const char * (*refName)     (int obj);
   } LWObjectFuncs;
```

```
count = numObjects()
```

Returns a count of the number of objects in the object database. This is the number of unique object files that have been loaded, which in general will be different from the number of animateable object items (clones and null objects, for example, aren't counted).

`name = **filename**( object_index )`

Returns the filename for the object. Objects in the database are indexed by integers ranging from 0 to `count` - 1. Except during rendering, the index associated with a given object can change at any time as objects are added to or removed from the object database by the user.

`maxlayer = **maxLayers**( object_index )`

Returns a value one greater than the highest indexed, existing layer for the object. This is just the layer count if all layers with indexes between 0 and `maxlayer` - 1 exist.

`exists = **layerExists**( object_index, layer_index )`

True if the layer exists.

`**pivotPoint**( object_index, layer_index, pos )`

Get the pivot point for the object layer.

`mesh = **layerMesh**( object_index, layer_index )`

Returns a mesh info structure for the object layer. These are described on the Mesh Info page. For object layers with subpatches, the mesh returned by this function does *not* include geometry that would be created by subdivision unless the subpatches have been frozen.

`vmap_count = **numVMaps**( vmtype );`

Returns a count of the number of vertex maps of a given type, or the total of all types in the scene if `vmtype` is 0. Vmap type codes are an extensible set of four-character identifiers. The `lwmeshes.h` header defines some of the common vmap IDs.

`LWVMAP_PICK` - selection set
`LWVMAP_WGHT` - weight map
`LWVMAP_MNVW` - SubPatch weight map
`LWVMAP_TXUV` - texture UV coordinates
`LWVMAP_MORF` - relative vertex displacement
`LWVMAP_SPOT` - absolute vertex displacement

`vmap_name = **vmapName**( vmtype, vmindex );`

Returns the name of a vmap. The index ranges from 0 to `vmap_count` -

1.

`dimensions = **vmapDim**( vmtype, vmindex );`

Returns the number of dimensions, or values per vertex, of a vmap. Vmaps are typically 2D or 3D (two or three coordinate values per vertex), but they can have any number of dimensions, including 0.

vmtype = **vmapType**(vmindex)

Returns the LWID for the vmap. Call `numVMaps` with a `vmtype` of 0 to find the upper bound on `vmindex`.

lname = **layerName**(objnum, lnum)

Returns the name assigned to the layer, or NULL if the layer is unnamed.

isvis = **layerVis**(objnum, lnum)

Returns a boolean indicating whether the layer is marked as visible.

`name = **userName**( object_index )`

Returns the name of the object as seen by the user. This is typically the base filename without the path or extension, or "Unnamed N" for unsaved objects. These are not guaranteed to be unique.

`name = **refName**( object_index )`

Returns an internal reference name for this object. The reference name is guaranteed to be unique and unchanging for the lifetime of the object. This is useful in Modeler as an argument to commands requiring a filename, since some objects in Modeler may not have been saved yet and therefore have no filename.

Example

The inertia sample uses this global to display a list of vertex maps to the user. The vmap values are used in the displacement evaluation to scale the lag.

Shelf Functions

Availability LightWave 6.0
Component Layout, Modeler
Header lwshelf.h

The shelf is a window where users can store and retrieve presets for your plug-in. The shelf API supplies functions that let you subscribe (connect), set the context (tell the shelf that your interface is the active one), open the shelf window, and add, load and save presets.

For some plug-in classes, you don't need to call this global in order to gain access to the shelf's preset management. Beginning with LightWave 7.0, the user can load and save presets for your plug-in through the VIPER (Versatile Interactive Preview Render) interface. Presets can be added to the shelf or loaded into your plug-in through calls to your regular handler load and save callbacks.

Global Call

```
LWShelfFuncs *shelff;
shelff = global( LWSHELFFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWShelfFuncs.

```
typedef struct st_LWSHELFFUNCS {
   LWShelfCltID (*subscribe)(char *name, char *subName,
                             void *userData, int flags,
                             LWShelfLoadOkFunc *,
                             LWShelfLoadFunc *,
                             LWShelfSaveFunc *);
   void (*unsubscribe)   (LWShelfCltID);
   void (*open)          (LWShelfCltID);
   int  (*isOpen)        (LWShelfCltID clt);
   void (*close)         (LWShelfCltID);
   void (*setContext)    (LWShelfCltID);
   int  (*addPreset)     (LWShelfCltID, LWPixmapID img,
                            LWShelfParmList parms);
   void (*load)          (LWShelfCltID, char *filename,
                            int prompt_user);
   void (*save)          (LWShelfCltID, char *filename,
                            LWImageID thumimg, LWShelfParmList);
   int  (*addNamedPreset)(LWShelfCltID clt, LWPixmapID img,
                            LWShelfParmList parms, const char *name,
                            const char *comment );
} LWShelfFuncs;
```

client = **subscribe**(class, server, userdata, flags, loadok, load, save)

Initialize the interaction between a plug-in instance and the shelf. The client ID returned by this function is passed as the first argument to all of the others. The user data, typically the instance data for handlers, will be passed as the first argument to the three callbacks. The flags argument is a set of flag bits combined using bitwise-or. They indicate what kind of preset loading and saving the plug-in supports and can be any combination of the following.

SHLF_BIN
: Saves to and loads from binary files.

SHLF_ASC
: Saves to and loads from ASCII (text) files

SHLF_SEP
: Saves to and loads from separate (non-LightWave) files.

unsubscribe(client)

Conclude your instance's use of the shelf. You should call this before your instance is destroyed.

open(client)

Open the preset shelf window and set the context to your plug-in. The window will display only the presets for your plug-in.

open = **isOpen**(client)

Returns true if the shelf window is open.

close(client)

Close the shelf window.

setContext(client)

Set the shelf context.

index = **addPreset**(client, img, params)

Add a preset to the shelf. The `img` is the preset's thumbnail in the shelf window, created using the Image Utility functions. The `params` are passed as a NULL-terminated array of strings (tags) that you can use in your shelf load callback to tell which parameters in this preset should be loaded. In the simplest case, `params` will be NULL.

`addPreset` is implemented as a call to `addNamedPreset`, which is passed a default name generated by the shelf system. New code should call `addNamedPreset` directly.

load(client, filename, prompt_user)
save(client, filename, thumb_img, params)

Load or save a preset in an external file. Presets are ordinarily stored in a file managed by the shelf system, but you can use these functions to load and save presets in files you name. For loading, if `prompt_user` is true and the preset contains a parameter list (`params` was non-NULL when `save` was called for the preset), the user is prompted for input.

index = **addNamedPreset**(client, img, params, name, comment)

Add a named preset to the shelf. Like `addPreset`, but you can also specify a name and a comment that will help the user identify the preset.

Callbacks

The last three arguments to the `subscribe` function are callbacks that the shelf calls when a preset is to be loaded or saved. The shelf system calls these to actually load and save the preset. For all three callbacks, the first argument is the user data you passed to `subscribe`.

```
   typedef int  LWShelfLoadOkFunc (void *userdata);
   typedef void LWShelfLoadFunc   (void *userdata, const LWLoadState *,
                                      const char *filename,
                                      LWShelfParmList);
   typedef void LWShelfSaveFunc   (void *userdata, const LWSaveState *,
                                      const char *filename );
```

LWShelfLoadOkFunc

Returns a code that tells the shelf system whether to load the preset and whether the user should be prompted first. The code can be one of the following.

SHLC_NOWAY
: Do not load. Your plug-in should tell the user why.

SHLC_DFLT
: Display the default confirmation dialog.

SHLC_FORCE
: Load without consulting the user.

Your plug-in can display its own confirmation dialog and then return `NOWAY` or `FORCE` as appropriate, but you'll lose the ability to use parameter lists to selectively load parameters from your presets.

LWShelfLoadFunc

Load the preset. The shelf calls this when the user double-clicks on a preset thumbnail in the shelf window, or when you call the `load` function to load a preset from a file. In the first case, the `filename` argument will be NULL, and you'll read the preset's parameters by calling the LWLoadState functions. Typically, handlers pass this job on to their handler load callback. In the second case, the LWLoadState will be NULL, and you'll read the preset from the file named in `filename`. You can still pass this on to your handler's load callback by creating an LWLoadState for the preset file using the File I/O global.

The parameter list is an array of strings. It contains a subset of the parameter list you passed to `addPreset`, `addNamedPreset` or `save`. The default load confirmation dialog presents the list to the user and allows the user to select parameters. Only selected parameters are passed to your load callback. You can use the selections to load only portions of a preset.

LWShelfSaveFunc

Save the preset. The shelf calls this when you call `addPreset`, `addNamedPreset` or `save`. When adding a preset to the shelf, the `filename` argument will be NULL, and you'll store the preset's parameters by calling the LWSaveState functions. Typically, handlers pass this job on to their handler save callback. When saving a preset to a file, the LWSaveState will be NULL, and you'll write the preset to the file named in `filename`. You can still pass this on to your handler's save callback by creating an LWSaveState for the preset file using the File I/O global.

The parameter list is an array of strings representing parameters or groups of parameters in your preset data. When the preset is reloaded, the user can select from this list of named parameters, and the user's selection will be passed to the load callback.

Example

The shelf SDK sample is a Master plug-in that demonstrates the shelf functions. (This is *all* it does, in fact.) The plug-in's "data" is merely a color. The interface lets you add color presets to the shelf or save them in separate files. You can then load them back into the plug-in's handler instance.

State Query

Availability LightWave 6.0
Component Modeler
Header lwmodeler.h

This global provides a set of functions that return information about the current modeling environment.

Global Call

```
   LWStateQueryFuncs *query;
   query = global( LWSTATEQUERYFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWStateQueryFuncs.

```
   typedef struct st_LWStateQueryFuncs {
      int          (*numLayers) (void);
      unsigned int (*layerMask) (EltOpLayer);
      const char * (*surface)   (void);
      unsigned int (*bbox)      (EltOpLayer, double *minmax);
      const char * (*layerList) (EltOpLayer, const char *);
      const char * (*object)    (void);
      int          (*mode)      (int);
      const char * (*vmap)      (int, LWID *);
   } LWStateQueryFuncs;
```

`nlayers = `**`numLayers`**`()`

Returns the number of data layers for the current object.

`mask = `**`layerMask`**`( oplayer )`

Returns bits for the data layers included in the EltOpLayer selection. If bit *i* of the mask is set, then layer *i* + 1 of the current object belongs to the set defined by the `oplayer` argument. This function is provided primarily for backward compatibility. New code should use the `layerList` function, which is designed for multiple objects and an unlimited number of layers.

`surfname = `**`surface`**`()`

Returns the name of the current default surface.

`npoints = `**`bbox`**`( oplayer, box )`

Returns the number of points in the layer selection. If `box` isn't NULL, it is an array of six doubles that will receive the bounding box of the points in the layer selection, in the order (x0, x1, y0, y1, z0, z1).

`layers = `**`layerList`**`( oplayer, objname )`

Returns a string containing layer numbers for the given EltOpLayer and object. The layer numbers in the string are separated by spaces, with the highest numbered layer listed first. The object name is its filename, or NULL for the current object.

`objname = `**`object`**`()`

Returns the filename of the current object. If the geometry in the current layers hasn't been saved to a file yet, this returns the reference name (the name that would be returned by the Object Functions `refName` function). If no object has been loaded into Modeler, this returns NULL.

`m = `**`mode`**`( setting )`

Returns the state of a user interface setting. The setting codes are

LWM_MODE_SELECTION
: Returns the selection mode (points, polygons, volume) as an integer.

LWM_MODE_SYMMETRY
: Returns the state of the symmetry toggle.

`vmapname = `**`vmap`**`( index, lwid )`

Returns the name of the currently selected vertex map, and stores the `LWID` of the vmap in the second argument. The index can be one of the following.

```
LWM_VMAP_WEIGHT
LWM_VMAP_TEXTURE
LWM_VMAP_MORPH
```

Example

This code fragment exercises the query functions.

```
#include <lwserver.h>
#include <lwmodeler.h>

LWStateQueryFuncs *query;
double box[ 6 ];
char *surfname, *layers, *objname;
int nlayers, npoints;

query = global( LWSTATEQUERYFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( !query ) return AFUNC_BADGLOBAL;

nlayers  = query->numLayers();
npoints  = query->bbox( OPLYR_PRIMARY, box );
surfname = query->surface();
```

```
objname  = query->object();
layers   = query->layerList( OPLYR_FG, objname );
```

Surface Editor

Availability LightWave 6.0
Component Layout, Modeler
Header lwsurfed.h

The surface editor global allows you to control the surface editing interface.

Global Call

```
   LWSurfEdFuncs *surfedf;
   surfedf = global( LWSURFEDFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWSurfEdFuncs.

```
   typedef struct st_LWSurfEdFuncs {
      void (*open)       (int);
      void (*close)      (void);
      int  (*isOpen)     (void);
      void (*setSurface) (LWSurfaceID);
      void (*setPosition)(int x, int y);
      void (*getPosition)(int *x, int *y, int *w, int *h);
   } LWSurfEdFuncs;
```

open(int)
: Open the surface editor window.

close()
: Close the window.

state = **isOpen**()
: True if the editor window is open.

setSurface(surfid)
: Set the current surface in the editor.

setPosition(x, y)
: Set the window's position relative to the upper left corner of the screen.

getPosition(x, y, w, h)
: Get the window's position and size in pixels.

Surface Functions

Availability LightWave 6.0
Component Layout, Modeler
Header lwsurf.h

This global allows you to get information about surfaces and surface parameters.

Global Call

```
   LWSurfaceFuncs *surff;
   surff = global( LWSURFACEFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWSurfaceFuncs.

```
   typedef struct st_LWSurfaceFuncs {
      LWSurfaceID   (*create)      (const char *objName,
                                      const char *surfName);
      LWSurfaceID   (*first)       (void);
      LWSurfaceID   (*next)        (LWSurfaceID);
      LWSurfaceID * (*byName)      (const char *name, const char *obj);
      LWSurfaceID * (*byObject)    (const char *name);
      const char *  (*name)        (LWSurfaceID);
      const char *  (*sceneObject) (LWSurfaceID);
      int           (*getInt)      (LWSurfaceID, const char *channel);
      double *      (*getFlt)      (LWSurfaceID, const char *channel);
      LWEnvelopeID  (*getEnv)      (LWSurfaceID, const char *channel);
      LWTextureID   (*getTex)      (LWSurfaceID, const char *channel);
      LWImageID     (*getImg)      (LWSurfaceID, const char *channel);
      LWChanGroupID (*chanGrp)     (LWSurfaceID);
      const char *  (*getColorVMap)(LWSurfaceID surf);
      void          (*setColorVMap)(LWSurfaceID surf,
                                      const char *vmapName, int type);
   } LWSurfaceFuncs;
```

`surf = create( objname, surfname )`
: Create a new surface. The object name is the filename, which you can get from the Object Info `filename` function and Modeler's State Query `object` function, given the object's item ID.

`surf = first()`
: Returns the ID of the first surface in the surfaces list.

`surf = next( surf )`
: Returns the ID of the next surface in the surfaces list (the one following the argument).

`surfarray = byName( surfname, objname )`

Returns the IDs of the (possibly many) surfaces with a given name. Different objects can have surfaces of the same name. If `objname` is NULL, the array will contain every surface ID named `surfname`, regardless of which object it belongs to. The array of surface IDs is terminated by an ID of NULL.

`surfarray = `**`byObject`**`( objname )`

Returns the surfaces belonging to the object. The object name is the filename.

`surfname = `**`name`**`( surf )`

Returns the name of a surface.

`scenename = `**`sceneObject`**`( surf )`

Returns the filename of the object to which the surface belongs.

`val = `**`getInt`**`( surf, channel )`

Returns the value of the surface parameter (evaluates the channel) at the current time. Use this function for integer parameters and `getFlt` for floating-point parameters. The `channel` is one of the channel names listed in lwsurf.h.

`valarray = `**`getFlt`**`( surf, channel )`

Returns the value of the surface parameter. The return value in most cases points to one double, but for colors, it points to three.

`envelope = `**`getEnv`**`( surf, channel )`

Returns the envelope ID for the surface parameter. This can be used with the Animation Envelopes global.

`texture = `**`getTex`**`( surf, channel )`

Returns a texture ID for the surface parameter that can be used with the Texture Functions global.

`image = `**`getImg`**`( surf, channel )`

Returns the image associated with the surface parameter. This function is limited to use with surface channels that refer directly to images, e.g. `SURF_RIMG` and `SURF_TIMG` (reflection and refraction maps). Images that are part of textures have to be obtained through the Texture Functions global.

`group = `**`chanGrp`**`( surf )`

Returns the channel group for the surface. This is the parent group for envelopes associated with the surface's parameters. It can be used with the Channel Info global. Note: because of a bug, this field may be NULL in some builds of LightWave 6.

`name = `**`getColorVMap`**`( surf )`

Returns the name of the vertex color map for the surface.

`setColorVMap``( surf, vmapname, type )`

Set the surface's vertex color map. The type can be `LWVMAP_RGB` (the vmap has a dimension of 3 and contains red, green and blue levels) or `LWVMAP_RGBA` (dimension of 4, with RGB and alpha levels).

Example

The scenscan SDK sample includes a `getObjectSurfs` function that collects surface information for all of an object's surfaces.

For some parameters, you'll want to consult the object file format specification, since the form of the data returned by the `get` functions is in some cases the same as its binary image in the object file. This code fragment reads and interprets the reflection options.

```
#include <lwserver.h>
#include <lwsurf.h>

LWSurfaceFuncs *surff;
LWSurfaceID surfid;
LWImageID rimg;
double refl, rsan, *dval;
int rfop;

... assume surff and surfid have been initialized ...

dval = surff->getFlt( surfid, SURF_REFL );        // reflectivity
refl = *dval;
if ( refl > 0.0f ) {
   rfop = surff->getInt( surfid, SURF_RFOP );     // options
   switch ( rfop ) {
      case 0:  /* backdrop only */       ... break;
      case 1:  /* raytrace + backdrop */ ... break;
      case 2:  /* spherical map */       ... break;
      case 3:  /* raytrace + map */      ... break;
   }
   if ( rfop == 2 || rfop == 3 ) {
      rimg = surff->getImg( surfid, SURF_RIMG );  // image map
      dval = surff->getFlt( surfid, SURF_RSAN );  // seam angle
      rsan = *dval;
   }
}
```

System ID

Availability LightWave 6.0
Component Layout, Modeler
Header lwhost.h

The system ID global returns the hardware key serial number. Each unique serial number represents a LightWave license, so a plug-in installed on a particular machine can use this value to lock that installation to the LightWave license. But note that your plug-in can be called in contexts in which the host doesn't require a hardware key, so exercise care when relying on the value returned by this function.

Global Call

```
   unsigned long sysid;
   sysid = ( unsigned long ) global( LWSYSTEMID_GLOBAL,
      GFUSE_TRANSIENT );
```

The global function ordinarily returns a `void *`, so this should be cast to an integer type to get the return value.

The serial number is in the low 28 bits of the return value. The high four bits indicate whether the plug-in is running in an interactive Layout or Modeler session, or in a non-interactive Screamernet session. The two components of the number can be extracted using macros defined in `lwhost.h`.

```
   context  = sysid & LWSYS_TYPEBITS;
   serialno = sysid & LWSYS_SERIALBITS;
```

The `context` can be `LWSYS_LAYOUT`, `LWSYS_MODELER`, or `LWSYS_SCREAMERNET`.

If the `context` is `LWSYS_SCREAMERNET`, the serial number will either be the node number, or 0 if no node number is available. The serial number can also be 0 for products other than LightWave that don't require a hardware key, e.g. Inspire, regardless of the context. (The product info global allows you to identify the product your plug-in is running in.)

Texture Editor

Availability LightWave 6.0
Component Layout, Modeler
Header lwtxtred.h

This global provides access to a user interface for editing textures. If you use XPanels with vparms that can be textured, the interaction with the texture editor is handled for you, and you don't need this global. But if your interface is built with classic Panels or OS-specific elements, you can use this global to provide your users with the standard texture interface.

Global Call

```
   LWTxtrEdFuncs *txedf;
   txedf = global( LWTXTREDFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWTxtrEdFuncs.

```
   typedef struct st_LWTxtrEdFuncs {
      LWTECltID   (*subscribe)   (char *title, int flags, void *userData,
                                   LW_TxtrRemoveFunc *,
                                   LW_TxtrAutoSizeFunc *,
                                   LW_TxtrEventFunc *);
      void        (*unsubscribe) (LWTECltID);
      void        (*open)        (LWTECltID, LWTextureID, char *title);
      void        (*setTexture)  (LWTECltID, LWTextureID, char *title);
      void        (*setPosition) (LWTECltID, int, int);
      void        (*close)       (LWTECltID);
      int         (*isOpen)      (LWTECltID);
      int         (*refresh)     (LWTECltID);
      LWTLayerID  (*currentLayer)(LWTECltID);
      int         (*selectAdd)   (LWTECltID, LWTextureID);
      int         (*selectRem)   (LWTECltID, LWTextureID);
      int         (*selectClr)   (LWTECltID);
      LWTextureID (*selectFirst) (LWTECltID);
      LWTextureID (*selectNext)  (LWTECltID, LWTextureID);
      void        (*setGradientAutoSize)(LWTECltID,
                                   LW_GradAutoSizeFunc *);
   } LWTxtrEdFuncs;
```

`client = subscribe( title, flags, data, txremove, txautosz, txevent )`

Returns an identifier that plug-ins use in later calls to the texture editor functions. The callbacks are optional and are called when the user removes or autosizes a texture, or does anything with it in the editor. The `data` argument is passed to these callbacks; its contents are up to you, and it can be NULL. The flags determine what the user can do in the editor and can be one or more of the following.

`TEF_USEBTN`
: Add use/remove buttons at the bottom of the pane.

`TEF_OPACITY`
: Add layer opacity settings.

`TEF_BLEND`
: Add blend options to the layer global settings.

`TEF_TYPE`
: Add layer type control on the top of the pane.

`TEF_LAYERS`
: Add layer list pane on the left side of the pane.

`TEF_ALL`
: All of the above flags. This is the standard configuration for the texture editor.

`**unsubscribe**( client )`

Free resources allocated by `subscribe`. This call invalidates the client ID. You'll need to call `subscribe` again before calling the texture editor functions.

`**open**( client, texture, title )`

Open the texture editor window.

`**setTexture**( client, texture, title )`

Initialize the texture editor with the texture to be edited.

`**setPosition**( client, x, y )`

Move the editor window. The coordinates are for the upper left corner of the window.

`**close**( client )`

Close the texture editor window.

`isopen = **isOpen**( client )`

True if the editor window is currently open.

`result = **refresh**( client )`

Redraw the editor window.

`tlayer = **currentLayer**( client )`

Returns the texture layer currently being edited.

```
ok = selectAdd( client, texture )
ok = selectRem( client, texture )
ok = selectClr( client )
```

Add a texture to a multiselection, remove a texture from a multiselection, or clear the selection list.

`texture = **selectFirst**( client )`

`next = **selectNext**( client, texture )`

Enumerate the selected textures.

`**setGradientAutoSize**( client, gsizecb )`

Set a callback for autosize requests from gradient texture layers.

Callbacks

The callbacks passed to `subscribe` and `setGradientAutoSize` allow you to react to user actions in the editor.

```
   typedef void LW_TxtrRemoveFunc (LWTextureID, void *userData);
   typedef int  LW_TxtrAutoSizeFunc (LWTextureID, void *userData,
      double bbox[3][2]);
   typedef int  LW_GradAutoSizeFunc (LWTxtrParamDesc *param,
      int paramNb, void *userData);
   typedef int  LW_TxtrEventFunc (LWTextureID, void *userData,
      int eventCode);
```

The remove callback is called when a texture is removed. The texture autosize callback is called when the user has requested that the texture size be set automatically. The bounding box array should be set to the default size of the texture. The gradient autosize callback is called for automatic sizing of gradient layers. The size should be set in the parameter description. See the Texture Functions global for a description of the LWTxtrParamDesc structure.

The event callback is called when the texture settings are modified by the user. This gives you a chance to update thumbnails or other aspects of your interface that depend on the texture settings. The event code can be one of the following.

TXEV_ALTER
: A texture setting has changed.

TXEV_TRACK
: A texture setting is being changed (a slider is being manipulated, for example).

TXEV_DELETE
: A texture layer has been deleted.

Example

The txchan and atmosphere samples use Texture Editor functions.

Texture Functions

Availability LightWave 6.0 **Component** Layout, Modeler
Header lwtxtr.h

The Texture Functions global gives plug-ins access to LightWave's texture engine. A plug-in can create and use textures to modulate its parameters, and it can read and modify the settings of existing textures.

Global Call

```
LWTextureFuncs *txfunc;
txfunc = global( LWTEXTUREFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWTextureFuncs.

```
typedef struct st_LWTextureFuncs {
   LWTxtrContextID (*contextCreate)(LWTxtrParamFuncs);
   void          (*contextDestroy) (LWTxtrContextID);
   void          (*contextAddParam)(LWTxtrContextID,
                                      LWTxtrParamDesc);
   LWTextureID   (*create)       (int returnType, const char *name,
                                    LWTxtrContextID, void *userdata);
   void          (*destroy)      (LWTextureID);
   void          (*copy)         (LWTextureID to, LWTextureID from);
   void          (*newtime)      (LWTextureID, LWTime, LWFrame);
   void          (*cleanup)      (LWTextureID);
   void          (*load)         (LWTextureID, const LWLoadState *);
   void          (*save)         (LWTextureID, const LWSaveState *);
   double        (*evaluate)     (LWTextureID, LWMicropolID, double *);
   void          (*setEnvGroup)  (LWTextureID, LWChanGroupID);
   LWTLayerID    (*firstLayer)   (LWTextureID);
   LWTLayerID    (*lastLayer)    (LWTextureID);
   LWTLayerID    (*nextLayer)    (LWTextureID, LWTLayerID);
   LWTLayerID    (*layerAdd)     (LWTextureID, int type);
   void          (*layerSetType) (LWTLayerID, int type);
   int           (*layerType)    (LWTLayerID);
   double        (*layerEvaluate)(LWTLayerID, LWMicropolID, double *);
   LWChanGroupID (*layerEnvGrp)  (LWTLayerID);
   int           (*setParam)     (LWTLayerID, int tag, void *data);
   int           (*getParam)     (LWTLayerID, int tag, void *data);
   void          (*evaluateUV)   (LWTLayerID, int wAxis, int oAxis,
                                    double oPos[3], double wPos[3],
                                    double uv[2]);
   double        (*noise)        (double p[3]);
   void *        (*userData)     (LWTextureID);

   LWChanGroupID (*envGroup)     (LWTextureID);
   LWTextureID   (*texture)      (LWTLayerID);
   const char *  (*name)         (LWTextureID);
   int           (*type)         (LWTextureID);
   LWTxtrContextID (*context)    (LWTextureID);
   } LWTextureFuncs;
```

It's helpful to divide these functions into three categories according to whether they deal with contexts, handler calls, or texture settings. Plug-ins that use textures to modify their own parameters will mostly use functions in the first two groups, since typically the texture settings are supplied by the user through the Texture Editor. The last group is more often useful when plug-ins want to query or modify existing textures.

Contexts

Some texture layer types use additional parameters to modify the texture value. Currently this is a gradient thing. The texture context is used to populate and support the Input Parameter menu for gradient layers in the Texture Editor.

`context = `**`contextCreate`**`( paramfuncs )`

> Create a texture context. The Texture Editor uses the callbacks in the `paramfuncs` argument to get the value of the parameters.

`contextDestroy``( context )`

> Free resources allocated by `contextCreate`.

`contextAddParam``( context, param )`

> Add a parameter to the context. For gradient layers, this parameter will be added to the Input Parameter menu.

Handler Calls

The functions in this group call the texture's handler callbacks. See the document for the procedural texture plug-in class for more information about the other side of these calls. In most cases, you'll call these from within your own plug-in's handler callbacks. In all cases, however, these functions must be called in proper handler order. The `newtime` function, for example, must be called before calling `evaluate`.

`texture = `**`create`**`( returntype, name, context, userdata )`

> Create a texture. The LWTextureID returned by this function is a container that will hold one or more texture layers. The final value of the texture is a combination of the values calculated for each layer.

The data type of the texture value can be one of the following.

```
TRT_VECTOR
TRT_COLOR
TRT_PERCENT
TRT_SCALAR
TRT_DISPLACEMENT
```

The `name` is used to identify the texture in the user interface. The `context` is a context ID returned by `contextCreate`, or NULL if you don't want to add any input parameters for the texture. The `userdata` is any data you'd like to associate with the texture. You can retrieve it using the `userdata` function.

destroy(texture)

Free the texture.

copy(to, from)

Copy a texture.

newtime(texture, time, frame)

Prepare the texture to be evaluated at a new render time. This allows the texture to do time-dependent precalculations.

cleanup(texture)

Call this when calculations using the texture are completed, typically after the last frame has been rendered.

load(texture, loadstate)

Read the texture from a file.

save(texture, savestate)

Write the texture to a file.

alpha = **evaluate**(texture, micropol, value)

Evaluate the texture. You must initialize the LWMicropol structure, described later. The result is returned in `value`.

Texture Data

These functions are used to get and set the data that defines a texture.

setEnvGroup(texture, changroup)

Set the channel group to be used by the texture. Envelopes created for parameters in the texture's layers will belong to this group.

`tlayer = `**`firstLayer`**`( texture )`

`tlayer = `**`lastLayer`**`( texture )`
`tlayer = `**`nextLayer`**`( texture, tlayer )`

Enumerate the texture's layers. The layer ID returned by these functions can be passed to functions that return the layer's data.

`tlayer = `**`layerAdd`**`( texture, type )`

Add a texture layer. The type is one of the following.

`TLT_IMAGE`

An image map.

`TLT_PROC`

A procedural texture.

`TLT_GRAD`

A gradient.

`layerSetType``( tlayer, type )`

Change the layer type.

`type = `**`layerType`**`( tlayer )`

Returns the layer type.

`alpha = `**`layerEvaluate`**`( tlayer, micropol, value )`

Evaluate the layer. Like the `evaluate` function, but it calculates the texture value only for the specified layer.

`changroup = `**`layerEnvGroup`**`( tlayer )`

Returns the channel group for the layer's enveloped parameters.

`result = `**`setParam`**`( tlayer, tag, value )`
`result = `**`getParam`**`( tlayer, tag, value )`

Set or get a layer parameter. The tag identifies the parameter and its data type.

`TXTAG_POSI (double [3])`

`TXTAG_ROTA (double [3])`
`TXTAG_SIZE (double [3])`

The origin, rotation and scale of the texture layer.

`TXTAG_FALL (double [3])`

Falloff, an amount per unit distance.

`TXTAG_PROJ (int *)`

Projection type for image maps, which can be one of the following.

```
TXPRJ_PLANAR
TXPRJ_CYLINDRICAL
TXPRJ_SPHERICAL
TXPRJ_CUBIC
TXPRJ_FRONT
TXPRJ_UVMAP
```

`TXTAG_AXIS (int *)`
: The texture axis, for image map projections that require one.

`TXTAG_WWRP (double *)`

`TXTAG_HWRP (double *)`
: Width and height wrap amount. Some projection types use this to tile the texture image.

`TXTAG_COORD (int *)`
: The coordinate system. This is 1 for world coordinates (the texture doesn't move with the object), and 0 for object coordinates.

`TXTAG_IMAGE (LWImageID *)`
: The image for image mapped layers. The value is a pointer to an image ID, typically obtained from the Image List `load` function.

`TXTAG_VMAP (VMapID *)`
: The vertex map used, for example, by UV mapped and weight mapped textures. The value is the opaque pointer returned by the MeshEditOp `pointVSet` function or the mesh info `pntVLookup` function.

`TXTAG_ROBJ (LWItemID *)`
: The reference object, from which the texture origin, rotation and scale will be taken. The item ID is typically obtained from the Item Info `first` and `next` functions.

`TXTAG_OPAC (double *)`
: Layer opacity.

`TXTAG_AA (int *)`
: Boolean, whether texture antialiasing is enabled for the layer.

`TXTAG_AAVAL (double *)`
: Antialiasing threshold. The texture value will only be antialiased if it differs from adjacent values by an amount greater than this threshold.

`TXTAG_PIXBLEND (int *)`
: Boolean, whether pixel blending is enabled. Pixel blending is a form of antialiasing that's active in regions where the texture resolution is lower than the output resolution.

`TXTAG_WREPEAT (int *)`

`TXTAG_HREPEAT (int *)`
: Width and height repeat behavior.

```
TXRPT_RESET
TXRPT_REPEAT
TXRPT_MIRROR
TXRPT_EDGE
```

`TXTAG_ACTIVE (int *)`

Boolean, whether texture layer is active.

`TXTAG_INVERT (int *)`

Boolean, whether texture layer is inverted.

`TXTAG_BLEND (int *)`

Texture blending mode:

```
TXBLN_NORMAL
TXBLN_SUBTRACT
TXBLN_DIFFERENCE
TXBLN_MULTIPLY
TXBLN_DIVIDE
TXBLN_ALPHA
TXBLN_DISPLACE
TXBLN_ADD
```

`evaluateUV``( tlayer, waxis, oaxis, opos, wpos, uv )`

For texture layers that use one of the implicit image mapping projections (planar, cubic, cylindrical, spherical), returns the UV coordinates for a given position. If the texture uses an explicit UV mapping, the UV coordinates can be obtained directly from the vertex map through mesh info or MeshEditOp functions.

The `w` arguments are in world coordinates, and the `o` arguments are in object coordinates. The axis arguments are the dominant axis for cubic mapping and can be 0, 1 or 2 for the X, Y or Z axis. This is usually chosen as the polygon normal component that's largest in absolute value. For projections other than cubic, these arguments are ignored. The position arguments specify the position for which the UV should be returned.

`value =` **`noise`**`( pos )`

Easy access to a noise function.

`data =` **`userData`**`( texture )`

Returns whatever was passed as the user data argument to the `create` function.

`changroup =` **`envGroup`**`( texture )`

Returns the channel group for the texture.

`id =` **`texture`**`( layer )`

Returns the texture ID, given any layer in the texture.

`tname = **name**( texture )`

`datatype = **type**( texture )`
`ctxt = **context**( texture )`

> These return information about the texture. The information is the same as that supplied in the first three arguments to the `create` function.

Parameter Callbacks

The argument to the `contextCreate` function is an LWTxtrParamFuncs, which contains callbacks for evaluating the input parameters. These callbacks are functions in your plug-in that determine the value of the parameter.

```
   typedef struct st_LWTxtrParamFuncs {
      double     (*paramEvaluate)(LWTxtrParamDesc *, int paramnum,
                                  LWMicropol *, gParamData);
      gParamData (*paramTime)    (void *userData, LWTxtrParamDesc *,
                                  int paramnum, LWTime, LWFrame);
      void       (*paramCleanup) (LWTxtrParamDesc *, int paramnum,
                                  gParamData);
   } LWTxtrParamFuncs;
```

`value = **paramEvaluate**( pdesc, pindex, micropol, pdata )`

> Returns the value of the parameter. The `pdesc` is the parameter description you passed to `contextAddParam` for this parameter. The `pindex` is an integer identifying the parameter by the order in which it was created. It's 1 for the parameter created by your first call to the `contextAddParam` function, 2 for the second parameter, and so on. The 0 index is reserved for the Previous Layer parameter, which always exists. The `micropol` is the micropolygon passed to the texture. The `pdata` argument is the user data you returned from your `paramTime` callback.

`pdata = **paramTime**( userdata, pdesc, pindex, time, frame )`

> The init function for the parameter. This is called before `paramEvaluate` so that you can perform precalculations for your parameter. The `userdata` is the same as that returned by the `userdata` function.

`**paramCleanup**( pdesc, pindex, pdata )`

> The cleanup function for the parameter. This allows you to free any resources allocated in your `paramTime`.

Parameter Descriptor

The second argument to `contextAddParam` is a description of the parameter contained in an LWTxtrParamDesc structure. This structure is also passed to your parameter callbacks.

```
   typedef struct st_LWTxtrParamDesc{
      char     *name;
      double    start;
      double    end;
      int       type;
      int       flags;
      int       itemType;
      LWItemID  itemID;
      char     *itemName;
   } LWTxtrParamDesc;
```

`name`

The name of the parameter as it should appear in the user interface.

`start`, `end`

The nominal limits of the parameter's value. These form the endpoints of a gradient.

`type`

The data type of the parameter, which can be one of the following.

```
LWIPT_FLOAT
LWIPT_DISTANCE
LWIPT_PERCENT
LWIPT_ANGLE
```

`flags`

Parameter flags.

`LWGF_FIXED_MIN`

The minimum parameter value is fixed.

`LWGF_FIXED_MAX`

The maximum value is fixed.

`LWGF_FIXED_START`

The start value is fixed.

`LWGF_FIXED_END`

The end value is fixed.

`itemType`, `itemID`, `itemName`

If the parameter depends on a scene item, these fields describe the item. The type can be one of the following. (If the parameter doesn't use an item, the type should be `LWGI_NONE`.)

```
LWGI_NONE
LWGI_OBJECT
LWGI_LIGHT
```

```
LWGI_CAMERA
LWGI_BONE
LWGI_VMAP
```

Micropolygon Descriptor

The micropolygon provides the geometry information used to evaluate a texture. You need to initialize one of these before calling `evaluate` or `layerEvaluate`. You also receive one of these in your parameter callbacks.

```
typedef struct st_LWMicropol {
   double           oPos[3];
   double           wPos[3];
   double           oScl[3];
   double           gNorm[3];
   double           wNorm[3];
   double           ray[3];
   double           bumpHeight;
   double           txVal;
   double           spotSize;
   double           raySource[3];
   double           rayLength;
   double           cosine;
   double           oXfrm[9];
   double           wXfrm[9];
   LWItemID         objID;
   LWItemID         srfID;
   LWPntID          verts[4];
   float            weights[4];
   float            vertsWPos[4][3];
   int              polNum;
   int              oAxis;
   int              wAxis;
   int              context;
   LWIlluminateFunc *illuminate;
   LWRayTraceFunc   *rayTrace;
   LWRayCastFunc    *rayCast;
   LWRayShadeFunc   *rayShade;
   void             *userData;
   LWPolID           polygon;
} LWMicropol;
```

Almost all of the micropolygon fields correspond to fields of the same name in LWShaderAccess. See the shader page for descriptions of those fields.

oScl
: Texture scale in object coordinates.

ray
: The direction of the incoming viewing ray.

txVal

The initial value that will be modified by the texture.

srfID
The ID of the surface associated with the texture. (Note: This is incorrectly typed as an LWItemID. Just cast the LWSurfaceID to LWItemID when setting this field.)

context
This will be `TCC_ANY` in most cases. The other two are used when the texture needs to be evaluated in two separate steps, which is unusual.

`TCC_ANY`
All layers will be evaluated.
`TCC_OBJECT`
Only object coordinate layers will be evaluated.
`TCC_WORLD`
Only world coordinate layers will be evaluated.

History

In LightWave 7.5, the server name for this global (`LWTEXTUREFUNCS_GLOBAL`) was incremented from "Texture Functions 2" to "Texture Functions 3", and the `TXTAG_ACTIVE`, `TXTAG_INVERT` and `TXTAG_BLEND` tags were added to LWTextureFuncs.

Example

The txchan sample contains motion, channel, image filter and environment plug-ins, all of which use a texture to modulate their data. The texture layers are defined by the user and evaluated through the Texture Functions global.

The following code fragment demonstrates how to extract UV values for image maps associated with a surface.

```
#include <lwserver.h>
#include <lwsurf.h>
#include <lwtxtr.h>

LWSurfaceFuncs *surff;
LWTextureFuncs *txtrf;
LWSurfaceID surf;
LWTextureID tex;
LWTLayerID tlayer;
int type;
```

As always, you need to get the globals before you can use them.

```
surff = global( LWSURFACEFUNCS_GLOBAL, GFUSE_TRANSIENT );
txtrf = global( LWTEXTUREFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

Each surface has many channels (Color, Diffuse, Luminous, Specular, etc.), and each channel can be textured. If a channel is textured, the texture can have many layers. It's at the level of the texture layer that you want to look for UVs.

```
tex = surff->getTex( surf, SURF_COLR );
if ( tex ) {
   tlayer = txtrf->firstLayer( tex );
   while ( tlayer ) {
      type = txtrf->layerType( tlayer );
      if ( type == TLT_IMAGE ) {
```

Now you have an image texture layer. You can ask what the projection is.

```
         int proj;
         txtrf->getParam( tlayer, TXTAG_PROJ, &proj );
         if ( proj == TXPRJ_UVMAP ) {
```

If the projection type is UV, get the vmap.

```
            void *vmap;
            txtrf->getParam( tlayer, TXTAG_VMAP, &vmap );
```

Use this with the mesh edit `pointVSet` and `pointVEval` functions to get the UVs. (You can also use the mesh info `pntVSet`, `pntVGet` and `pntVPGet` functions.)

```
            edit->pointVSet( edit->state, vmap, 0, NULL );
            for each point
               edit->pointVEval( edit->state, pntID, polID, uv );
         }
         else {
```

If the projection is *not* UV, use `evaluateUV`.

```
            for each point
               txtrf->evaluateUV( tlayer, wAxis, oAxis, oPos, wPos,
                  uv );
         }
      }
      tlayer = txtrf->layerNext( tlayer );
   }
}
```

Time Info

Availability LightWave 6.5
Component Layout
Component lwrender.h

The time info global returns the time of the frame currently being rendered.

Global Call

```
LWTimeInfo *timeinfo;
timeinfo = global( LWTIMEINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWTimeInfo.

```
typedef struct st_LWTimeInfo {
   LWTime  time;
   LWFrame frame;
} LWTimeInfo;
```

`time`
The time in seconds of the frame currently being rendered.

`frame`
The frame number for the current frame.

Variant Parameters

Availability LightWave 6.0
Component Layout, Modeler
Header lwvparm.h

A variant parameter, or vparm, is a double-precision variable or 3-vector that can vary as a function of time. Vparms are used as containers for the values of XPanel controls that can be enveloped or textured (any control with "-env" in its type name). That's the rationale for the existence of vparms, but you're free to use them for other things as well.

The variant parameters global supplies the implementation of the vparm data type.

Global Call

```
   LWVParmFuncs *vparmf;
   vparmf = global( LWVPARMFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWVParmFuncs.

```
   typedef struct st_LWVParmFuncs {
      LWVParmID   (*create)  (int envType, int texType);
      void        (*destroy) (LWVParmID);
      void        (*setup)   (LWVParmID,
                                 const char *channelName,
                                 LWChanGroupID group,
                                 LWTxtrContextID gc,
                                 LWVP_EventFunc *eventFunc,
                                 const char *pluginName,
                                 void *userData);
      LWError     (*copy)    (LWVParmID to, LWVParmID from);
      LWError     (*load)    (LWVParmID, const LWLoadState *load);
      LWError     (*save)    (LWVParmID, const LWSaveState *save);
      double      (*getVal)  (LWVParmID, LWTime t,
                                 LWMicropolID mp, double *result);
      int         (*setVal)  (LWVParmID, double *value);
      int         (*getState)(LWVParmID);
      void        (*setState)(LWVParmID, int state);
      void        (*editEnv) (LWVParmID);
      void        (*editTex) (LWVParmID);
      void        (*initMP)  (LWMicropolID mp);
      void        (*getEnv)  (LWVParmID, LWEnvelopeID envlist[3]);
      LWTextureID (*getTex)  (LWVParmID);
   } LWVParmFuncs;
```

`vparm = create( envtype, txtype )`

Create a new vparm. The envelope type can be one of the following.

LWVP_FLOAT

A floating-point number.

LWVP_PERCENT

A floating-point number with a nominal range of 0.0 to 1.0. This value will be represented to the user as a percentage.

LWVP_DIST

A floating-point distance. In meters internally, it may appear in the interface with a variety of units.

LWVP_ANGLE

An angle. In radians internally but in degrees for users.

LWVP_COLOR

A floating-point color vector.

Each of these types corresponds to an XPanel control type. To create a 3-vector vparm, add `LWVPF_VECTOR` to one of the first four types (the `LWVP_COLOR` type code already includes the `LWVPF_VECTOR` bit).

The texture type corresponds to the return type you would specify in the Texture Functions `create` function and can be one of the following.

```
LWVPDT_NOTXTR
LWVPDT_VECTOR
LWVPDT_COLOR
LWVPDT_PERCENT
LWVPDT_SCALAR
LWVPDT_DISPLACEMENT
```

destroy`( vparm )`

Free a vparm.

setup`( vparm, name, cgroup, txcontext, eventfunc, plugname, userdata )`

Initialize a vparm. This must be called for every vparm you create. The `name` is the name of the envelope (the base name for vectors), and the `cgroup` is the channel group in which the envelopes are created. The event callback is described below. The plug-in name (the name in your ServerRecord's `name` field) and user data are used by LightWave to identify the owner of a vparm. The user data is also passed to the event callback.

`error =` **copy**`( vpto, vpfrom )`

Copy a vparm. This will primarily be called by handler copy callbacks.

`error =` **load**`( vparm, loadstate )`

Read a vparm from a file. This is meant to be called by handler load callbacks, but it might also be called by plug-ins using the file I/O global to read a file containing vparm data.

`error = `**`save`**`( vparm, savestate )`

Write a vparm to a file. This is meant to be called by handler save callbacks, but might also be used to save the vparm to a file created through the file I/O global.

`result = `**`getVal`**`( vparm, time, micropol, value )`

Get the value of a vparm. The `micropol` is used by textures. If it is NULL, the texture's contribution to the value is ignored. See the Texture Functions global for a description of the LWMicropol structure. The `value` argument should always point to storage for three doubles, whether or not the vparm is a vector. If the vparm is textured, `getVal` returns the texture opacity.

`result = `**`setVal`**`( vparm, value )`

Set the value of a vparm. If the value is enveloped, calling this has no effect. Returns the number of elements processed (0, 1, or 3).

`state = `**`getState`**`( vparm )`

Returns a set of state bits. If the `LWVPSF_ENV` bit is set, an envelope exists for the vparm, and if the `LWVPSF_TEX` bit is set, the vparm has a texture.

`setState``( vparm, state )`

Create or destroy the vparm's underlying envelope or texture. The `state` argument uses the same state bits as `getState`. If the bit is clear (0), the envelope or texture is destroyed, and if the bit is set (1), an envelope or texture is created for the vparm using the information previously specified in `setup`. Never call this for a vparm associated with an XPanel control. The XPanel system takes care of creating and destroying vparm envelopes and textures automatically.

`editEnv``( vparm )`

Open the graph editor for the vparm. This does nothing if the vparm isn't enveloped. You won't need to call this for a vparm associated with an XPanel control, since the control will give the user a way to call the graph editor without your help.

`editTex``( vparm )`

Open the texture editor for the vparm. This has no effect if no texture has been created for the vparm. You won't need to call this for vparms used with XPanel controls.

initMP(micropol)

Initialize a micropolygon. The transformation matrices are set to the identity matrix. Most other fields are set to 0.

getEnv(vparm, envarray)

Gets the envelope IDs of the vparm's envelopes. The first element of the array will contain the single ID for non-vector vparms. If no envelopes exist for the vparm, the array elements will be NULL.

texture = **getTex**(vparm)

Returns the texture ID for the vparm's texture.

Event Callback

The event callback you pass to `setup` is used to inform you of changes to the underlying envelopes and texture of your vparm, and in one case to ask you for data needed by the texture.

```
   typedef int LWVP_EventFunc( LWVParmID vp, void *userdata,
      en_lwvpec eventcode, void *eventdata );
```

The `userdata` is whatever you passed as the last argument to `setup`. The `eventcode` will be one of the following.

LWVPEC_TXTRACK
: Generated as the texture changes.

LWVPEC_TXUPDATE
: Generated after the texture has changed.

LWVPEC_TXAUTOSIZE
: Request for texture autosize. For this event, `eventdata` is an array of six doubles (*x* low, *x* high, *y* low, *y* high, *z* low, *z* high) that you're being asked to initialize for the default size of the texture.

LWVPEC_ENVTRACK
: Generated as the envelope changes.

LWVPEC_ENVUPDATE
: Generated after the envelope has changed.

LWVPEC_ENVNEW
: An envelope has been created.

LWVPEC_ENVOLD
: An envelope is being destroyed.

LWVPEC_TEXNEW
: A texture has been created.

LWVPEC_TEXOLD
: The texture is being destroyed.

Currently, for all events other than `TXAUTOSIZE`, the `eventdata` will be NULL and can be ignored.

Example

Several of the SDK samples (blotch, inertia, mandfilt and rapts) use vparms as part of their XPanel interfaces. It's not a coincidence that all of these are handlers, since handlers are more likely to need time-dependent parameters.

Viewport Info

Availability LightWave 7.5 **Component** Layout
Header lwrender.h

The viewport info global returns information about the state of Layout's OpenGL viewports. The data is read-only, but you can set the parameters using navigation and display commands.

Global Call

```
   LWViewportInfo *vpinfo;
   vpinfo = global( LWVIEWPORTINFO_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWViewportInfo.

```
   typedef struct st_LWViewportInfo {
        int                numViewports;
        int              (*type)  (int);
        unsigned int     (*flags) (int);
        void             (*pos)   (int, LWDVector);
        void             (*xfrm)  (int, double mat[9]);
        void             (*clip)  (int, double *hither, double *yon);
        void             (*rect)  (int, int *left, int *top, int *width, int *height);
   } LWViewportInfo;
```

`numViewports`

The number of viewports visible in the Layout interface.

`view = `**`type`**`( i )`

The type of view in the i^{th} viewport. It will be one of the following:

```
LVVIEWT_NONE
LVVIEWT_TOP
LVVIEWT_BOTTOM
LVVIEWT_BACK
LVVIEWT_FRONT
LVVIEWT_RIGHT
LVVIEWT_LEFT
LVVIEWT_PERSPECTIVE
LVVIEWT_LIGHT
LVVIEWT_CAMERA
LVVIEWT_SCHEMATIC
```

`flags = `**`flags`**`( i )`

Returns a set of bit flags for the i th viewport. These can be any combination of the following.

```
LWVIEWF_CENTER
LWVIEWF_WEIGHTSHADE
LWVIEWF_XRAY
```

`pos``(i, spot)`

Fills the `spot` vector with the viewing position of the i th viewport.

`xfrm``( i, mat[9] )`

Fills `mat` with a 3x3 transformation from world coordinates to viewport cordinates for the i th viewport.

`clip``( i, &hither, &yon )`

Fills `hither` and `yon` with the near and far Z clipping distances for the i th viewport.

`rect``( i, &left, &top, &width, &height )`

Fills `left`, `top`, `width` and `height` with pixel coordinates of the i th viewport.

XPanels

Availability LightWave 6.0
Component Layout, Modeler
Header lwxpanel.h

This global is a new addition to LightWave's system of platform-independent user interface components. To understand its place in that system and how it differs from Panels, it may help to know something about the history of user interfaces in LightWave.

XPanels is the third generation of LightWave interface APIs. The first was the requester API, which in its original Modeler incarnation predates the plug-in architecture itself. Modeler requesters were always modal. You gave Modeler a list of controls and their initial values, Modeler formatted and displayed the dialog, and after the user closed the dialog, you read out the control values. The original requester API is now just another way of creating an xpanel, and you can use it in Layout as well as Modeler.

Panels is the second generation, and it's pretty bohemian, like Algol was to Fortran. It introduced callbacks so that you can respond to the user interactively, and it comes with a list of 50 or so predefined control types and some drawing functions. You can set the size of a panel and the positions of its controls, and you can decorate your panels and tell them to send you a steady stream of mouse and keyboard events. As with requesters, controls are defined using a sequence of function calls.

XPanels is a return to the Modeler requester philosophy of automation. You give up some control over the appearance and behavior of your controls, but in exchange, things that can require a lot of code with panels, like adding an E (envelope) button to a numeric edit field, along with all of the functionality that implies, are very easy to do with xpanels. And xpanels can be embedded in LightWave's own windows. Unlike the previous interface models, the controls on an xpanel are defined primarily through arrays of static data.

Panels continues to support some control types, such as the OpenGL control, that have no xpanel equivalents, so it's still your best option in some cases. But you aren't forced to choose, since Panels also allows you to create xpanel controls--xpanels embedded within your panel.

Global Call

```
LWXPanelFuncs *xpanf;
xpanf = global( LWXPANELFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWXPanelFuncs.

```
typedef struct st_LWXPanelFuncs {
   int             version;
   LWXPDrawFuncs  *drawf;
   LWXPanelID     (*create)     (int type, LWXPanelControl *);
   void           (*destroy)    (LWXPanelID);
   void           (*describe)   (LWXPanelID, LWXPanelDataDesc *,
                                   LWXPanelGetFunc *,
                                   LWXPanelSetFunc *);
   void           (*hint)       (LWXPanelID, unsigned long,
                                   LWXPanelHint *);
   void *         (*getData)    (LWXPanelID, unsigned long);
   void           (*setData)    (LWXPanelID, unsigned long, void *);
   void *         (*formGet)    (LWXPanelID, unsigned long);
   void           (*formSet)    (LWXPanelID, unsigned long, void *);
   void           (*viewInst)   (LWXPanelID, void *);
   void           (*viewRefresh)(LWXPanelID);
   int            (*post)       (LWXPanelID);
   int            (*open)       (LWXPanelID);
} LWXPanelFuncs;
```

A *panel* is a window or a dialog box containing buttons, edit fields, dropdown lists, image rectangles, and other user interface elements collectively called *controls*. You create panels to allow users to enter and change settings that affect the operation of your plug-in.

The XPanels system distinguishes between two kinds of panels, called *forms* and *views*. They differ primarily in the method used to move data values into and out of controls.

A view panel presents a "view" of a *data instance*, which is a pointer to some data owned by your plug-in, typically a handler's instance data. You pass this pointer to XPanels in the `viewInst` function. The callbacks you pass to the `describe` function will be called by XPanels to update this data.

A form panel isn't passed a data instance pointer, so you have greater freedom in choosing how to represent your data within your plug-in. XPanels interacts with your data through the `formGet` and `formSet` functions and your change notification callback.

Because they don't store a data instance pointer, forms have no way to distinguish between multiple instances of the same dataset, making it more difficult to use the same panel with different data. The significance of this may grow as XPanels evolves. In the future, for example, users may be able to multiselect every instance of your plug-in and edit the parameters for all of them simultaneously.

`version`

> The version number of the interface. Certain control types or hints may not be available in older interfaces.

`drawf`

> Drawing functions, which are explained in detail below. Custom drawing on xpanels is limited to contexts in which an LWXPDrArea region is defined.

`panel =` **`create`**`( type, controls )`

> Create a new panel. The type can be `LWXP_VIEW` or `LWXP_FORM`. The `controls` argument is an array of LWXPanelControl structures describing the controls on the panel. It must have at least panel lifetime, meaning that the `controls` variable must be valid as long as the panel exists. The easiest way to ensure this is to declare the array static. See below for more about the controls list.

`destroy``( panel )`

> Free the panel and related resources allocated by `create`.

`describe``( panel, datadesc, getfunc, setfunc )`

> Define the data values for a panel. The `datadesc` argument is a list of value IDs and their types. Like the `controls` list passed to `create`, `datadesc` must have at least panel lifetime. For view panels, the values are updated interactively using your `getfunc` and `setfunc` callbacks. If the panel is a form, the `getfunc` and `setfunc` arguments are ignored and should be NULL.

`hint``( panel, id, hints )`

> Apply a hint array to the panel or one of its controls. This may be called multiple times, but only while the panel is closed. The hint array is applied to the panel if the ID is 0. Otherwise it's applied to the control with the corresponding ID.

`data =` **`getData`**`( panel, id )`

Returns the user data for the panel (`id == 0`) or for one of its controls, set by a previous call to `setData`.

`**setData**( panel, id, data )`
: Set the user data for a panel (`id == 0`) or one of its controls. The `data` pointer is passed to several of the callbacks. It's also what `getData` returns.

`value = **formGet**( panel, vid )`
: Read the value of a control on a form panel. Returns NULL if the value is undefined.

`**formSet**( panel, vid, value )`
: Write a value to a control on a form panel.

`**viewInst**( panel, instance )`
: Set the instance data pointer for a view panel. This is what is passed to the get and set callbacks you designate when you call the `describe` function.

`**viewRefresh**( panel )`
: Refreshes a view panel by reloading control values from the instance data. Use this when the instance data has been modified by something other than the user's interaction with your interface. When called from a form, `viewRefresh` redraws the controls.

`result = **post**( panel )`
: Display the panel for modal interaction. *Modal* means that the user must close the panel before continuing to work in LightWave. This function won't return until the panel has been closed. It returns 1 if the user dismisses the panel with the "OK" button and 0 if the user presses "Cancel" (or if the panel couldn't be displayed). See the description of the `XpDLOGTYPE` macro for more information about the buttons, like "OK" and "Cancel," that are automatically added to modal panels.

`result = **open**( panel )`
: Display the panel for non-modal interaction. This function returns immediately and the panel stays open until the user closes it. Returns 1 if the panel was opened successfully, otherwise it returns 0. Handlers generally won't need to call `open` (or `post`), since LightWave takes care of opening the panel you return in the `panel` field of the LWInterface structure.

Describing Controls and Data

Each of the controls on a panel is described by an LWXPanelControl structure. An array of these is passed as the second argument to the `create` function when the panel is created.

```
   typedef struct st_LWXPanelControl {
      unsigned long   cid;
      const char     *label;
      const char     *ctrlclass;
   } LWXPanelControl;
```

The second argument to the `describe` function is an array of LWXPanelDataDesc.

```
   typedef struct st_LWXPanelDataDesc {
      unsigned long   vid;
      const char     *name;
      const char     *datatype;
   } LWXPanelDataDesc;
```

The first member of both structures is an identifier. It can be any positive integer greater than or equal to 0x8000. Values less than 0x8000 are reserved for use by XPanels. The second member is a human-readable description. For controls, this string is used as a label that appears next to the control on the panel. The third member identifies the class of the control or the data type of the value.

In most cases, your control array and data description array will be parallel, and the value IDs in the data description will match the control IDs in the control array. But some control types don't have an associated value, and you might declare some values to XPanels that aren't directly associated with any control.

Control Types

The following definitions list the control class (the string that goes in the `ctrlclass` field of the LWXPanelControl), the data type of the control's value (the string that goes in the `datatype` field of the LWXPanelDataDesc), and the C equivalent (the kind of variable your plug-in stores the value in). The definitions will often refer to hints and callbacks, both of which are described in later sections.

"**string**" : "string" : char[]

A text edit field.

"**integer**" : "integer" : int
An integer edit field.

"**float**" : "float" : double
An edit field for floating-point numbers. Variations on the floating-point control type are summarized in the following table. Controls that include envelope (E) and texture (T) buttons use variant parameters as their underlying data type.

"float"	`"float"`	`double`
"float3"	`"float3"`	`double[3]`
"float-env"	`"float-env"`	`double[]` (vparm)
"float3-env"	`"float3-env"`	`double[][3]` (vparm)
"distance"	`"distance"`	`double`
"distance3"	`"distance3"`	`double[3]`
"distance-env"	`"distance-env"`	`double[]` (vparm)
"distance3-env"	`"distance3-env"`	`double[][3]` (vparm)
"percent"	`"percent"`	`double`
"percent3"	`"percent3"`	`double[3]`
"percent-env"	`"percent-env"`	`double[]` (vparm)
"percent3-env"	`"percent3-env"`	`double[][3]` (vparm)
"angle"	`"angle"`	`double`
"angle3"	`"angle3"`	`double[3]`
"angle-env"	`"angle-env"`	`double[]` (vparm)
"angle3-env"	`"angle3-env"`	`double[][3]` (vparm)
"color"	`"color"`	`double[3]`
"color-env"	`"color-env"`	`double[][3]` (vparm)
"time"	`"time"`	`double`

Distance controls display units; their internal values are in meters. Percent controls display percentages; their internal values are the percent values divided by 100 (100% is 1.0). Angle controls display angles in degrees and store them in radians. Times are stored as elapsed seconds, but are displayed in the units selected by the user elsewhere in LightWave's interface.

"**vButton**"
A button. XPanels calls the control's button click callback whenever

the user presses the button.

"**iBoolean**" : "integer" : int
A checkbox.

"**iSlider**" : "integer" : int
"**iSliderText**" : "integer" : int
A slider (a thumb button that can be dragged within a horizontal track). Use the `XpMIN` and `XpMAX` hints to define the range of the slider. The `Text` version includes an integer edit field.

"**iChoice**" : "integer" : int
An array of radio buttons (mutually exclusive boolean buttons). The value is a 0-based index into the list of choices. To initialize the choices, use the `XpSTRLIST` hint.

"**axis**" : "axis" : 0, 1, 2
Radio buttons for selecting X, Y or Z.

"**iPopChoice**" : "integer" : int
A scrolling popup menu. The value is a 0-based index. You can initialize a popup with the `XpSTRLIST` hint and a static array of strings, or the `XpPOPFUNCS` hint and popup item count and name callbacks.

"**vPopCmd**"
A popup menu that works like a multiple-choice button. Your popup command callback receives a 0-based index.

"**sPresetText**" : "string" : char[]
Combines a string edit field and a popup menu. The user can type into the edit field or fill the field with the value associated with one of the presets named in the menu.

"**sFileName**" : "string" : char[]
Combines a string edit field and a button that opens the file dialog. Use the `XpXREQCFG` macro to set the mode (load or save), the title string and the file type filter.

"**iChoTransform**" : "integer" : int

"**iPopTransform**" : "integer" : int

Like iChoice and iPopChoice controls, but the value isn't the menu index. It is instead the value of an array element referenced by the index. The array contains values that are more useful to your plug-in than the 0, 1, 2... indexes into the menu. For the `i`-th menu item, the value is `map[i]`, where `map[]` is an integer array associated with the control using the `XpCHOXFORM` or `XpPOPXFORM` hints.

"**dThumbnail**"

A rectangular area that you can draw on using the drawing functions. Thumbnails generate mouse events. The user can click, double-click, or drag out a rectangular selection.

"**sInfo**" : "string" : char[]

A read-only text field.

Hints

The `hint` function accepts as one of its arguments an array of panel-building commands, called *hints*. These are packaged as macros defined in the `lwxpanel.h` header file. The macros are used as initializers for the hint array passed to `hint`. For example,

```
   static LWXPanelHint hints[] = {
      XpLABEL( 0, "My Panel Title" ),
      XpSTRLIST( MY_POP, choices_array ),
      ...
```

There are currently over 50 hint macros.

XpH(x)

Cast the value to an LWXPanelHint. This is just a way to get values of different types into a single array.

XpEND

Terminate the hint array. Some macros are also followed by a sublist of hints which is terminated by `XpEND`.

XpCALL(hint_array)

Insert another hint array into this one.

XpLABEL(id, label)

Set the label string for a control or group. An ID of 0 sets the label for the panel.

XpVECLABEL(ctlID, labels)

Set the labels for each of the three elements of a vector control (integer3, float3, etc.). `labels` is an array of three strings.

XpSTRLIST(ctlID, strlist)

Set the string array for a popup control.

XpCLASS(id, class)

Change the class of a control. Hints previously applied to this control may be lost if they're incompatible with the new class.

XpADD(ctlID, class, valID)

Create a control. You can also create controls using the LWXPanelControl array passed to the `create` function and the LWXPanelDataDesc array passed to `describe`.

XpDELETE(id)

Remove a control from the panel. This doesn't remove the control's value from the data description.

XpCTRLCFG(ctlID, options)

Set class-specific option flags for a control. Currently this is only used by dThumbnail controls, which can be configured using the following bit flags combined using bitwise-or.

```
THUM_SML    THUM_SQ      THUM_LAND    THUM_EURO
THUM_MED    THUM_NTSC    THUM_FULL
THUM_LRG    THUM_35MM    THUM_WIDE
THUM_XLG    THUM_PORT    THUM_ANAW
```

XpXREQCFG(ctlID, mode, title, filter)

Set options for the file dialog opened by an sFileName control. The `mode` is one of the following.

```
LWXPREQ_LOAD
LWXPREQ_SAVE
LWXPREQ_DIR
```

The `title` is the string that will appear in the title bar of the dialog.

The `filter` is a string that identifies the file type. It's the same string you'd pass to the File Type global.

XpFOCUS(id)
Identify the control that should receive focus when the panel is first displayed. An ID of 0 indicates the panel should use a default setting, usually the first editable text field.

XpVALUE(ctlID, valID)
Associate a control ID with a value ID. Ordinarily, a control and its value are implicitly bound together by having the same ID in the LWXPanelControl and LWXPanelDataDesc arrays passed to `create` and `describe`, but a control and value pair aren't required to share the same ID.

XpRESTORE(id)
XpRESTORE_()
XpRESTORE_ON
XpRESTORE_OFF
Define what happens to control values when the user presses the Cancel button in a modal `LWXP_VIEW` panel. By default, values are reset to what they were when the panel was opened. The `ON` and `OFF` macros set this behavior for all values, and the other two toggle the behavior for a single value or for a list of values.

XpLINK_(valID)
XpLINK(valID, link)
Link a value to another value, or to a list of values. Links create a dependency relationship. If the `link` value changes, then the dependent `valID` value is also updated.

XpENABLE_(valID)
XpENABLE_MAP_(valID, map)
XpENABLEMSG_(valID, msg)
XpENABLEMSG_MAP_(valID, map, msg)
Link the enable state of the controls in the trailing ID list to the `valID` value. The trailing ID list actually contains value IDs, and the control IDs are inferred from this. If the `valID` value is true (non-zero), the dependent controls are enabled, otherwise they're disabled. If you

supply a map (an integer array containing boolean values), the enable state will be based on `map[value]` rather than on the value directly. If you supply a message string, it will be displayed whenever the user tries to manipulate a disabled control. The message typically explains why the control is disabled: "Sprinkles is only valid when the flavor is vanilla."

XpINTXFORM(ctlID, count, ilist)
XpPOPXFORM(ctlID, count, ilist)
XpCHOXFORM(ctlID, count, ilist)
Set the transform map for the control. Transform control classes such as iPopTrans and iChoTrans provide a mechanism for mapping the control's underlying integer value into a more useful value for the client. `ilist` is an array of 32-bit values that XPanels will substitute for the base value of the control. When the base value is 0, XPanels uses `ilist[0]`, and so on.

XpMIN(ctlID, min)
Set the minimum of a control's range. `min` is always an integer, as are the other range parameters in the following hints.

XpMAX(ctlID, max)
Set the maximum of a control's range.

XpSTEP(ctlID, step)
Set the increment of a range adjustment.

XpRANGE(ctlID, min, max, step)
Set all three range parameters.

XpHARDMINMAX(ctlID, bmin, bmax)
Enforce the range limits in the edit field component of a slider control. `bmin` and `bmax` are booleans that set or clear this for each end of the range. If the control is enveloped, the envelope values may still fall outside the limits.

XpCLRMINMAX(ctlID, bmin, bmax)
Remove a minimum or maximum set by `XpMIN`, `XpMAX` or `XpRANGE`.

XpSENS(ctlID, delta)

Set the sensitivity of a minislider. The delta scales the pixel to minislider translation. Larger deltas cause the minislider value to change more slowly.

XpTRACK(ctlID, track)

Indicate whether the control should generate intermediate (tracking) events. If the boolean track flag is true (non-zero), events will be generated for the control as long as the user manipulates it on the interface. In particular, components such as sliders and text fields will generate a constant stream of events to inform the client of user actions. If this is false, events are only generated after the user has set a new value.

XpIMMUPD_(ctlID)
XpIMMUPD(ctlID, dep)

Create an "immediate update" dependency relationship for a control, such as a slider, that can generate `LWXPEVENT_TRACK` events. Whenever the control is being modified by the user (and is generating tracking events), the dependent controls will be updated.

XpDESTROYNOTIFY(func)

Set the callback that XPanels will call after the panel has been destroyed. The argument points to an LWXPanelDestroyNotifyFunc.

XpCHGNOTIFY(func)

Set the callback that XPanels will call whenever a value is modified. The argument points to an LWXPanelChangeNotifyFunc.

XpBUTNOTIFY(ctlID, func)

Set the callback that XPanels will call when the button control is pressed. The argument points to an LWXPanelBtnClickFunc.

XpPOPCMDFUNC(ctlID, func)

Set the callback that XPanels will call when an item is selected in a popup control. The argument points to an LWXPanelPopCmdFunc.

XpPOPFUNCS(ctlID, countfunc, namefunc)

Set the callbacks that XPanels will call when the popup choice control is opened. The arguments point to an LWXPanelPopCntFunc and an LWXPanelPopNameFunc. The callbacks are an alternative to providing a static array of strings for the popup.

XpDRAWCBFUNC(ctlID, func)
Set the callback that XPanels will call when a drawable control needs to be drawn. The argument points to an LWXPanelControlDrawFunc. Currently this is restricted to use with dThumbnail controls.

XpZOOMCBFUNC(ctlID, func, rect)
Set the callback that XPanels will call when the user clicks and drags the mouse within the control. The function argument points to an LWXPanelControlZoomFunc. `rect` is a boolean that determines whether mouse dragging is reported. Currently this is only for dThumbnail controls.

XpALIAS_(aliasID)
Create a new alias group, or add controls, values or other aliases to an existing alias group. An alias group is just an ID with its own list of IDs. The groups created using the `ALIAS` tag define a logical grouping of IDs but otherwise do not have any default behaviour. Instead, alias group IDs are generally used as arguments to other panel hints.

XpUNALIAS_(aliasID)
Remove the IDs in the trailing list from the alias group. If the ID list is empty (this hint is followed immediately by `XpEND`), all the elements of the group are removed and the group is disbanded.

XpGROUP_(groupID)
Create a new group, or add the trailing list of IDs to an existing group. The trailing list may contain the IDs of controls, aliases, other groups, stacks or tabs.

XpORDER_(groupID)
Reorder the IDs in a group. If the group ID is 0, this reorders IDs for the panel.

XpSTACK_(groupID, valID)
XpSTACK_MAP_(groupID, valID, map)

Define a control stack. Only one control at a time is visible in a stack (the rest are "hidden underneath"). Which control is visible is determined by the `valID` value. The value must be of integer type. A value of 0 selects the first control in the group. If you supply a map (an integer array), the visible control depends on `map[value]` rather than the value directly. A value of -1 means that no control from this group should be visible.

XpTABS_(groupID)

Create a tab window. Additional tab windows will be created for the IDs in the trailing list. The text displayed on the tab is taken from the label assigned to the ID using the `XpLABEL` hint.

XpCTRLFRONT(ctlID)

Identify the tab that should be displayed on top of other tabs. By default, the first tab is on top.

XpORIENT(ctlID, orientation)

Set the control's orientation, where 0 is horizontal and 1 is vertical. The default is horizontal. Not all controls can be oriented.

XpLEFT(groupID)
XpLEFT_()

Attempt to left align the first control in a group or the trailing list. If possible, subsequent controls are moved up beside the left-aligned control. This hint is ignored for some control classes (but see the `XpNARROW` hint), and space constraints can also affect whether it can be applied.

XpNARROW(groupID)
XpNARROW_()

The client may override the default "move up" rules by indicating that the controls in the group or the trailing list are narrow.

XpDIVADD(id)
XpDIVADD_(id)

This suggests the panel should draw a divider after the control or group specified by the ID list. The hint has one fixed argument so that the client can specify one of the special divider locations, intended mostly for inline panels: 0 for the top of the panel, -1 for the bottom.

XpDIVREM(id)
XpDIVREM_(id)

This indicates that the panel should not place a divider after the control or group. The panel performs some built-in guesswork on divider placement. Typically, they're put at "transition points," examples of which include points between a standalone control and a group, between separate groups, and between a group and a tab group. It won't always be easy to determine the abstract location of a divider solely from its appearance. For example, a divider may appear to follow a control but actually follow the group to which the control belongs.

XpBORDER(type)

Suggests the kind of border the panel should be drawn with. Possible values are

```
LWXPBDR_NONE
LWXPBDR_UP
LWXPBDR_DOWN
```

XpDLOGTYPE(type)

Modal panels include buttons that allow the user to accept or cancel their input. This macro specifies which buttons will be added. The numbers in the following list are the values returned by the `post` function when the user dismisses the panel by pressing the corresponding button.

`LWXPDLG_OKCANCEL` - OK 1, Cancel 0 (the default)
`LWXPDLG_DONE` - Done 1
`LWXPDLG_OKONLY` - OK 1
`LWXPDLG_YESNO` - Yes 1, No 0
`LWXPDLG_YESNOALL` - Yes 1, No 0, Yes To All 2, Cancel 3
`LWXPDLG_YESNOCAN` - Yes 1, No 0, Cancel 3

Callbacks

Xpanels and its controls rely on callbacks to communicate with your plug-in. Except for the data get and set callbacks, you tell XPanels about them using hints.

Data get/set

```
typedef void *LWXPanelGetFunc (void *inst, unsigned long vid);
typedef int   LWXPanelSetFunc (void *inst, unsigned long vid,
   void *value);
```

Your get and set functions are passed as the third and fourth arguments to the `describe` function when you're creating an `LWXP_VIEW` panel. XPanels calls the get function to get a value from you. The function returns a pointer to the value. XPanels calls the set function to give you a value when it has changed. The set callback returns a refresh code indicating what, if any, redrawing should take place on the panel. The code can be one of the following.

```
LWXPRC_NONE
LWXPRC_DFLT
LWXPRC_DRAW
LWXPRC_FULL
```

Destroy event

```
typedef void LWXPanelDestroyNotifyFunc (void *);
```

This is called after the panel has been destroyed. Provide one of these if you need to do any panel-specific cleanup. Use the `XpDESTROYNOTIFY` hint to set the callback.

Value change event

```
typedef void LWXPanelChangeNotifyFunc (LWXPanelID pan,
   unsigned long cid, unsigned long vid, int event_type);
```

This is called when user interaction is changing the value of a control, and when your panel is receiving or losing the input focus. Use the `XpCHGNOTIFY` hint to set the callback. The event type is one of the following.

```
LWXPEVENT_TRACK
LWXPEVENT_VALUE
LWXPEVENT_HIT
LWXPEVENT_FOCUS
LWXPEVENT_LOSEFOCUS
```

Typically, `LWXP_FORM` panels use the `TRACK` and `VALUE` events to dynamically follow changes in the values of their controls, while `LWXP_VIEW` panels rely instead on their data set callback.

Button click event

```
typedef void LWXPanelBtnClickFunc (LWXPanelID pan, int cid);
```

Called when the user clicks a button control. Use the `XpBUTNOTIFY` hint to set the callback.

Popup command

```
typedef void LWXPanelPopCmdFunc (LWXPanelID pan, int cid, int cmd);
```

Called when the user selects an item from a popup menu. Use the `XpPOPCMDFUNC` hint to set the callback.

Popup choice

```
typedef int         LWXPanelPopCntFunc  (void *userdata);
typedef const char *LWXPanelPopNameFunc (void *userdata, int idx);
```

Called when a popup is displayed. The count callback returns the number of items in the menu. The name callback is then called for each item and returns the string to be displayed for the item. Use the `XpPOPFUNCS` hint to set these callbacks.

Draw event

```
typedef void LWXPanelControlDrawFunc (LWXPanelID pan,
   unsigned long cid, LWXPDrAreaID *reg, int w, int h);
```

Called when a drawable control (a dThumbnail) needs to be redrawn. Use the `XpDRAWCBFUNC` hint to set the callback.

Zoom event

```
typedef void LWXPanelControlZoomFunc (LWXPanelID pan,
   unsigned long cid, int x, int y, int *region, int clickcount );
```

Called when the user clicks and drags the mouse within the control (a dThumbnail). Use the `XpZOOMCBFUNC` hint to set the callback.

Drawing Functions

Drawing is limited to contexts in which a valid LWXPDrAreaID is available, currently the draw callback of a thumbnail control.

```
typedef struct st_LWXPDrawFuncs {
   void (*drawPixel)    (LWXPDrAreaID, int c, int x, int y );
   void (*drawRGBPixel) (LWXPDrAreaID, int r, int g, int b, int x,
                           int y );
   void (*drawLine)     (LWXPDrAreaID, int c, int x, int y, int x2,
                           int y2 );
   void (*drawBox)      (LWXPDrAreaID, int c, int x, int y, int w,
                           int h );
   void (*drawRGBBox)   (LWXPDrAreaID, int r, int g, int b, int x,
                           int y, int w, int h );
   void (*drawBorder)   (LWXPDrAreaID, int indent, int x, int y,
                           int w, int h );
   int  (*textWidth)    (LWXPDrAreaID, char *s );
   int  (*textHeight)   (LWXPDrAreaID, char *s );
   void (*drawText)     (LWXPDrAreaID, char *s, int c, int x, int y );
} LWXPDrawFuncs;
```

drawPixel(drawid, color, x, y)
drawRGBPixel(drawid, r, g, b, x, y)

Draw a pixel. The coordinates are relative to the upper-left corner of the drawing area. The color is specified as one of the palette colors defined in lwpanel.h or as levels of red, green and blue between 0 and 255.

drawLine(drawid, color, x1, y1, x2, y2)

Draw a line connecting the endpoints.

drawBox(drawid, color, x, y, w, h)
drawRGBBox(drawid, r, g, b, x, y, w, h)

Draw a solid rectangle.

drawBorder(drawid, indent, x, y, w, h)

Draw a rectangular border similar to the ones use to mark the borders of controls. The `indent` is the thickness of the border. If `h` is 0, `drawBorder` creates a horizontal divider.

tw = **textWidth**(drawid, string)
th = **textHeight**(drawid, string)

The pixel width and height of the text in the font used by panels.

drawText(drawid, string, color, x, y)

Render a line of text.

Example

Many of the SDK samples (including blotch, box, hotvideo, kepler, mandfilt, NoisyChan, specular, and txchan) use XPanels for their interfaces. xpanchan, xpanlgen and xpanxtreme were written specifically to demonstrate XPanels features.

Plug-in Classes

LightWave plug-ins are divided into categories called *classes*. For those familiar with the formal classes supported by object-oriented programming languages, this term may have unintended implications. The class of a plug-in simply defines what kind of plug-in it is, what it does, and how it interacts with LightWave.

Class Index

AnimLoaderHandler	FrameBufferHandler	MeshDataEdit
AnimSaverHandler	Global	MeshEditTool
ChannelHandler	ImageFilterHandler	ObjectLoader
ColorPicker	ImageLoader	ObjReplacementHandler
CommandSequence	ImageSaver	PixelFilterHandler
CustomObjHandler	ItemMotionHandler	ProceduralTextureHandler
DisplacementHandler	LayoutGeneric	SceneConverter
EnvironmentHandler	LayoutTool	ShaderHandler
FileRequester	MasterHandler	VolumetricHandler

Thumbnail Descriptions

AnimLoaderHandler
AnimSaverHandler
: Load and save files containing animation streams, e.g. MPEG, QuickTime and AVI files.

ChannelHandler
: Channel handlers can be applied to any animation parameter that can vary over time. They modify or replace the value of the parameter.

ColorPicker
: Provide a user interface for selecting colors.

CommandSequence
: Modeler plug-ins that can do almost anything the user can do through the interface.

CustomObjHandler
: Layout plug-ins that add custom drawing to an object in Layout's interface. These are often used to add visual feedback for the parameters controlled by null objects, but they can be used with any object.

DisplacementHandler
: Deform an object during animation by moving its points.

EnvironmentHandler
: Replace the render backdrop, for procedural sky and ground rendering, for example.

FileRequester
: Provide a user interface for selecting files (an alternative to the host system's default file dialog).

FrameBufferHandler
: Provide a display or a device interface for rendered frames.

Global
: Provide services to other plug-ins. See the globals section for information about LightWave's built-in globals.

ImageFilterHandler
: Image post-processing of rendered frames.

ImageLoader
ImageSaver
: Load and save files containing still images. Support for a number of image file formats is provided through plug-ins of these classes.

ItemMotionHandler
: Animate the position, size and scale of an item.

LayoutGeneric
: Miscellaneous utilities, Layout commands.

LayoutTool
: Interactive custom tools in Layout.

MasterHandler
: These receive event notifications from Layout and can control the behavior of other plug-ins based on those events.

MeshDataEdit
: Modeler plug-ins that create and modify geometry at the point and polygon level.

MeshEditTool
: Mesh editing with full user interactivity.

ObjectLoader
: Load the 3D geometry data in non-LightWave object files.

ObjReplacementHandler
: Replace the geometry of an object during animation.

PixelFilterHandler
: Modify or replace the value of each pixel sample during rendering.

ProceduralTextureHandler
: Procedural textures are just functions, like fractal noise, useful for adding detail to the appearance of an object or for modulating an item's motion.

SceneConverter
: Loads the animation data in non-LightWave scene files.

ShaderHandler
: Modifies the appearance of an object's surface.

VolumetricHandler
: Creates volumetric rendering effects, including transmission and scattering through transparent media, and hypertexturing.

Globals

Globals are services that any plug-in can request by calling the *global function*. This is passed as the second argument to every plug-in's activation function. Global services allow you to construct platform-independent user interfaces for your plug-ins and to query and modify LightWave's internal state.

The Global Function

The second argument to every activation function is a GlobalFunc.

```
   typedef void * GlobalFunc (const char *serviceName, int useMode);
```

The service name is a string that tells the global function which global service is being requested. It can be any of the globals supplied with LightWave and listed in the table above, or a custom global provided by a Global class plug-in.

The SDK header files define symbolic names for the service name strings of globals supplied with LightWave. The string for the current version of the Scene Info global, for example, is "LW Scene Info 2", but rather than use this string literal in your source code, you can use the symbolic name `LWSCENEINFO_GLOBAL`, defined in `lwrender.h`.

```
   LWSceneInfo *sceneinfo;
   sceneinfo = global( LWSCENEINFO_GLOBAL, GFUSE_TRANSIENT );
```

Using the symbolic name helps to ensure that your code is always synchronized with the headers you're currently compiling with. They're also a little easier to remember, since in most cases the symbolic name is the same as the type of the data object returned by the global function.

The use mode tells LightWave whether to *lock* the code module that supplies the global service. It can be `GFUSE_ACQUIRE` or `GFUSE_TRANSIENT`.

During routine housekeeping, LightWave may free memory by unloading plug-ins that haven't been called recently, and this can include Global class plug-ins. If you've requested a service provided by a Global class plug-in, the data and function pointers returned by the global function would become invalid if the Global plug-in were allowed to disappear from memory. The `GFUSE_ACQUIRE` mode locks the module, ensuring that it won't be unloaded from memory before you use it.

Globals obtained with `GFUSE_ACQUIRE` should be unlocked when they're no longer needed. You unlock them by calling the global function with a use mode of `GFUSE_RELEASE`.

```
   global( LWSCENEINFO_GLOBAL, GFUSE_RELEASE );
```

Failing to unlock a global usually isn't fatal, but it prevents LightWave from optimizing its use of memory. And locking a module doesn't prevent other plug-ins from using it.

Use the `GFUSE_TRANSIENT` mode when the global doesn't have to be locked. `GFUSE_TRANSIENT` is safe to use for the global services built into LightWave and when the services provided by Global class plug-ins are used immediately.

If the call to the global function succeeds, the return value is data, typically a pointer to a structure, that's specific to the requested global service. The documentation of the globals supplied with LightWave describes what each global returns. If the global call fails, a possibility that callers should always be prepared for, the return value is NULL.

Macros, Constants and Enum Members

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Symbol	Header
ABS()	lwmath.h
ACTIVATE_CON()	lwpanel.h
AFUNC_BADAPP	lwserver.h
AFUNC_BADGLOBAL	
AFUNC_BADLOCAL	
AFUNC_BADVERSION	
AFUNC_OK	
ANGLE_CTL()	lwpanel.h
AREA_CTL()	
BOOL_CTL()	
BOOLBUTTON_CTL()	
BORDER_CTL()	
BUTTON_CTL()	
CANVAS_CTL()	
CHANNEL_CTL()	
CLAMP()	lwmath.h
COLOR_BG	lwpanel.h
COLOR_BLACK	
COLOR_DK_GREY	
COLOR_DK_YELLOW	
COLOR_LT_GREY	
COLOR_LT_YELLOW	
COLOR_MD_GREY	
COLOR_WHITE	
CON_H()	
CON_HOTH()	
CON_HOTW()	
CON_HOTX()	
CON_HOTY()	
CON_LW()	
CON_MOUSEX()	
CON_MOUSEY()	
CON_PAN()	
CON_PANFUN()	
CON_SETEVENT()	
CON_W()	
CON_X()	
CON_Y()	
CSERR_ARGCOUNT	lwcmdseq.h
CSERR_ARGTYPE	
CSERR_ARGVALUE	
CSERR_BADSEL	
CSERR_IO	
CSERR_MEMORY	
CSERR_NONE	
CSERR_OPFAILURE	
CSERR_USERABORT	
CTL_ACTIVATE	lwpanel.h

CTL_FLAGS	
CTL_H	
CTL_HOTH	
CTL_HOTW	
CTL_HOTX	
CTL_HOTY	
CTL_LABEL	
CTL_LABELWIDTH	
CTL_MOUSEX	
CTL_MOUSEY	
CTL_PANEL	
CTL_PANFUN	
CTL_RANGEMAX	
CTL_RANGEMIN	
CTL_USERDATA	
CTL_USERDRAW	
CTL_USEREVENT	
CTL_VALUE	
CTL_W	
CTL_X	
CTL_Y	
CTLF_DISABLE	
CTLF_GHOST	
CTLF_INVISIBLE	
CUSTPOPUP_CTL()	
DEGREES()	lwmath.h
DIR_CTL()	lwpanel.h
DIRBUTTON_CTL()	
DIST_CTL()	
DR_ERASE	
DR_GHOST	
DR_REFRESH	
DR_RENDER	
DRAGAREA_CTL()	
DRAGBUT_CTL()	
DVEC_CTL()	
DY__LAST	lwdyna.h
DY_BOOLEAN	
DY_CHOICE	
DY_CUSTOM	
DY_DISTANCE	
DY_FLOAT	
DY_FONT	
DY_INTEGER	
DY_LAYERS	
DY_NULL	
DY_STRING	
DY_SURFACE	
DY_TEXT	
DY_VDIST	
DY_VFLOAT	
DY_VINT	
DYERR_BADCTRLID	
DYERR_BADSEQ	
DYERR_BADTYPE	
DYERR_INTERNAL	
DYERR_MEMORY	
DYERR_NONE	
DYERR_TOOMANYCTRL	
EDCOUNT_ALL	lwmeshedt.h
EDCOUNT_DELETE	
EDCOUNT_SELECT	

EDDF_DELETE	
EDDF_SELECT	
EDERR_BADARGS	
EDERR_BADLAYER	
EDERR_BADSURF	
EDERR_BADVMAP	
EDERR_NOMEMORY	
EDERR_NONE	
EDERR_USERABORT	
EDPF_CCEND	
EDPF_CCSTART	
EDSELM_CLEARCURRENT	
EDSELM_FORCEPOLS	
EDSELM_FORCEVRTS	
EDSELM_SELECTNEW	
EHMODE_PREVIEW	lwenviron.h
EHMODE_REAL	
ERASE_CON()	lwpanel.h
EVNT_ALL	
EVNT_BLOCKING	
FGETFLT()	lwxpanel.h
FGETINT()	
FILE_CTL()	lwpanel.h
FILEBUTTON_CTL()	
FLOAT_CTL()	
FLOATRO_CTL()	
FREQ_DIRECTORY	lwdialog.h
FREQ_LOAD	
FREQ_MULTILOAD	
FREQ_SAVE	
FSETFLT()	lwxpanel.h
FSETINT()	
FVEC_CTL()	lwpanel.h
FVECRO_CTL()	
GET_FLOAT()	
GET_FVEC()	
GET_INT()	
GET_IVEC()	
GET_STR()	
GETV_FVEC()	
GETV_IVEC()	
GFUSE_ACQUIRE	lwserver.h
GFUSE_RELEASE	
GFUSE_TRANSIENT	
GHOST_CON()	lwpanel.h
HALFPI	lwmath.h
HCHOICE_CTL()	lwpanel.h
HDRAGBUT_CTL()	
HSLIDER_CTL()	
IMG_GREY8	lwimageio.h
IMG_INDEX8	
IMG_RGB24	
IMGF_REVERSE	
INT_CTL()	lwpanel.h
INTRO_CTL()	
IPSTAT_ABORT	lwimageio.h
IPSTAT_BADFILE	
IPSTAT_FAILED	
IPSTAT_NOREC	
IPSTAT_OK	

IQ_ADJUST	lwpanel.h
IQ_ALT	
IQ_CONSTRAIN	
IQ_CTRL	
IQ_SHIFT	
ITEM_CTL()	
IVEC_CTL()	
IVECRO_CTL()	

LANGID_FRENCH	lwserver.h
LANGID_GERMAN	
LANGID_ITALIAN	
LANGID_JAPANESE	
LANGID_KOREAN	
LANGID_RUSSIAN	
LANGID_SPANISH	
LANGID_SWEDISH	
LANGID_UKENGLISH	
LANGID_USENGLISH	
LCLICK()	lwpanel.h
LISTBOX_CTL()	
LMO_HISTAPPEND	lwmonitor.h
LMO_IMMUPD	
LMO_NOABORT	
LMO_REVIEW	
LMON_DFLT()	
LMON_DONE()	
LMON_INCR()	
LMON_INIT()	
LMON_KILL()	
LMON_MSG()	
LMON_MSGI()	
LMON_MSGS()	
LMON_NEW()	
LMON_STEP()	
LMON_WPOS()	
LOAD_CTL()	lwpanel.h
LOADBUTTON_CTL()	
LWALERT_BEGINNER	lwrender.h
LWALERT_EXPERT	
LWALERT_INTERMEDIATE	
LWANIMLOADER_HCLASS	lwanimlod.h
LWANIMLOADER_ICLASS	
LWANIMLOADER_VERSION	
LWANIMSAVER_HCLASS	lwanimsav.h
LWANIMSAVER_ICLASS	
LWANIMSAVER_VERSION	
LWAST_FLOAT	
LWAST_UBYTE	
LWBACK_GRADIENT	lwrender.h
LWBACK_SOLID	
LWBACKDROPINFO_GLOBAL	
LWBONEF_ACTIVE	
LWBONEF_JOINT_COMP	
LWBONEF_JOINT_COMP_PAR	
LWBONEF_LIMITED_RANGE	
LWBONEF_MUSCLE_FLEX	
LWBONEF_MUSCLE_FLEX_PAR	
LWBONEF_SCALE_STRENGTH	
LWBONEF_WEIGHT_MAP_ONLY	
LWBONEF_WEIGHT_NORM	
LWBONEINFO_GLOBAL	
LWBUF_ALPHA	lwfilter.h
LWBUF_BLUE	

LWBUF_DEPTH	
LWBUF_DIFFSHADE	
LWBUF_DIFFUSE	
LWBUF_GEOMETRY	
LWBUF_GREEN	
LWBUF_LUMINOUS	
LWBUF_MIRROR	
LWBUF_MOTION_X	
LWBUF_MOTION_Y	
LWBUF_RAW_BLUE	
LWBUF_RAW_GREEN	
LWBUF_RAW_RED	
LWBUF_RED	
LWBUF_REFL_BLUE	
LWBUF_REFL_GREEN	
LWBUF_REFL_RED	
LWBUF_SHADING	
LWBUF_SHADOW	
LWBUF_SPECIAL	
LWBUF_SPECSHADE	
LWBUF_SPECULAR	
LWBUF_TRANS	
LWCAMERAINFO_GLOBAL	lwrender.h
LWCAMF_LIMITED_REGION	
LWCAMF_MASK	
LWCAMF_STEREO	
LWCEVNT_VALUE	lwenvel.h
LWCHANNEL_HCLASS	lwchannel.h
LWCHANNEL_ICLASS	
LWCHANNEL_VERSION	
LWCHANNELINFO_GLOBAL	lwenvel.h
LWCOF_SCHEMA_OK	lwcustobj.h
LWCOFL_SELECTED	
LWCOLORACTIVATEFUNC_GLOBAL	lwhost.h
LWCOLORPICK_CLASS	lwdialog.h
LWCOLORPICK_VERSION	
LWCOMPINFO_GLOBAL	lwrender.h
LWCONTEXTMENU_GLOBAL	lwpanel.h
LWCSYS_ICON	lwcustobj.h
LWCSYS_OBJECT	
LWCSYS_WORLD	
LWCUSTOMOBJ_HCLASS	
LWCUSTOMOBJ_ICLASS	
LWCUSTOMOBJ_VERSION	
LWDIRINFOFUNC_GLOBAL	lwhost.h
LWDISPF_CAGES	lwrender.h
LWDISPF_FIELDCHART	
LWDISPF_HANDLES	
LWDISPF_IKCHAINS	
LWDISPF_MOTIONPATHS	
LWDISPF_SAFEAREAS	
LWDISPLACEMENT_HCLASS	lwdisplce.h
LWDISPLACEMENT_ICLASS	
LWDISPLACEMENT_VERSION	
LWDisplayMetrics	lwpanel.h
LWDK_ALT	
LWDK_CHAR()	
LWDK_CTRL	
LWDK_DELETE	
LWDK_END	
LWDK_F1	
LWDK_F10	
LWDK_F11	
LWDK_F12	

LWDK_F2	
LWDK_F3	
LWDK_F4	
LWDK_F5	
LWDK_F6	
LWDK_F7	
LWDK_F8	
LWDK_F9	
LWDK_HELP	
LWDK_HOME	
LWDK_PAD_0	
LWDK_PAD_1	
LWDK_PAD_2	
LWDK_PAD_3	
LWDK_PAD_4	
LWDK_PAD_5	
LWDK_PAD_6	
LWDK_PAD_7	
LWDK_PAD_8	
LWDK_PAD_9	
LWDK_PAD_CPAREN	
LWDK_PAD_DASH	
LWDK_PAD_DECIMAL	
LWDK_PAD_ENTER	
LWDK_PAD_OPAREN	
LWDK_PAD_PLUS	
LWDK_PAD_SLASH	
LWDK_PAD_STAR	
LWDK_PAGEDOWN	
LWDK_PAGEUP	
LWDK_RETURN	
LWDK_SC_DOWN	
LWDK_SC_LEFT	
LWDK_SC_RIGHT	
LWDK_SC_UP	
LWDK_SHIFT	
LWDK_TOP_0	
LWDK_TOP_1	
LWDK_TOP_2	
LWDK_TOP_3	
LWDK_TOP_4	
LWDK_TOP_5	
LWDK_TOP_6	
LWDK_TOP_7	
LWDK_TOP_8	
LWDK_TOP_9	
LWDMF_BEFOREBONES	lwdisplce.h
LWDMF_WORLD	
LWDYNACONVERTFUNC_GLOBAL	lwdyna.h
LWDYNAMONITORFUNCS_GLOBAL	
LWDYNAREQFUNCS_GLOBAL	
LWDYNUP_DELAYED	lwrender.h
LWDYNUP_INTERACTIVE	
LWDYNUP_OFF	
LWEDGEF_CREASE	
LWEDGEF_OTHER	
LWEDGEF_SHRINK_DIST	
LWEDGEF_SILHOUETTE	
LWEDGEF_SURFACE	
LWEDGEF_UNSHARED	
LWEEVNT_DESTROY	lwenvel.h
LWEEVNT_KEY_DELETE	
LWEEVNT_KEY_INSERT	
LWEEVNT_KEY_TIME	

LWEEVNT_KEY_VALUE	
LWENF_TRANSPARENT	lwenviron.h
LWENVELOPEFUNCS_GLOBAL	lwenvel.h
LWENVIRONMENT_HCLASS	lwenviron.h
LWENVIRONMENT_ICLASS	
LWENVIRONMENT_VERSION	
LWENVTAG_KEYCOUNT	lwenvel.h
LWENVTAG_POSTBEHAVE	
LWENVTAG_PREBEHAVE	
LWENVTAG_VISIBLE	
LWET_ANGLE	
LWET_DISTANCE	
LWET_FLOAT	
LWET_PERCENT	
LWEVNT_COMMAND	lwmaster.h
LWEVNT_NOTHING	
LWEVNT_SELECT	
LWEVNT_TIME	
LWFBT_FLOAT	lwframbuf.h
LWFBT_UBYTE	
LWFCF_PREPROCESS	lwfilter.h
LWFILEACTIVATEFUNC_GLOBAL	lwhost.h
LWFILEIOFUNCS_GLOBAL	lwio.h
LWFILEREQ_CLASS	lwdialog.h
LWFILEREQ_VERSION	
LWFILEREQFUNC_GLOBAL	lwhost.h
LWFILETYPEFUNC_GLOBAL	
LWFOG_LINEAR	lwrender.h
LWFOG_NONE	
LWFOG_NONLINEAR1	
LWFOG_NONLINEAR2	
LWFOGF_BACKGROUND	
LWFOGINFO_GLOBAL	
LWFONTLISTFUNCS_GLOBAL	lwmodeler.h
LWFRAMEBUFFER_HCLASS	lwframbuf.h
LWFRAMEBUFFER_ICLASS	
LWFRAMEBUFFER_VERSION	
LWFTYPE_ANIMATION	lwhost.h
LWFTYPE_ENVELOPE	
LWFTYPE_IMAGE	
LWFTYPE_MOTION	
LWFTYPE_OBJECT	
LWFTYPE_PLUGIN	
LWFTYPE_PREVIEW	
LWFTYPE_PSFONT	
LWFTYPE_SCENE	
LWFTYPE_SETTING	
LWFTYPE_SURFACE	
LWGENF_FRACTIONALFRAME	lwrender.h
LWGENF_HIDETOOLBAR	
LWGENF_KEYSINSLIDER	
LWGENF_PARENTINPLACE	
LWGENF_PLAYEXACTRATE	
LWGENF_RIGHTTOOLBAR	
LWGF_AUTOSIZE	lwtxtr.h
LWGF_FIXED_END	
LWGF_FIXED_MAX	
LWGF_FIXED_MIN	
LWGF_FIXED_START	
LWGI_BONE	
LWGI_CAMERA	
LWGI_LIGHT	
LWGI_NONE	
LWGI_OBJECT	

LWGI_VMAP	
LWGLOBALPOOL_GLOBAL	lwrender.h
LWGLOBALPOOL_RENDER_GLOBAL	
LWGLOBALSERVICE_CLASS	lwglobsrv.h
LWGLOBALSERVICE_VERSION	
LWHOSTDISPLAYINFO_GLOBAL	lwdisplay.h
LWI_ANY	lwpanel.h
LWI_BONE	lwrender.h
LWI_CAMERA	
LWI_IMAGE	lwpanel.h
LWI_LIGHT	lwrender.h
LWI_OBJECT	
LWID_()	lwtypes.h
LWIMAGEFILTER_HCLASS	lwfilter.h
LWIMAGEFILTER_ICLASS	
LWIMAGEFILTER_VERSION	
LWIMAGELIST_GLOBAL	lwimage.h
LWIMAGELOADER_CLASS	lwimageio.h
LWIMAGELOADER_VERSION	
LWIMAGESAVER_CLASS	
LWIMAGESAVER_VERSION	
LWIMAGEUTIL_GLOBAL	lwimage.h
LWIMF_AFTERIK	lwmotion.h
LWIMPAR_ASPECT	lwimageio.h
LWIMPAR_BLACKPOINT	
LWIMPAR_FRAMESPERSECOND	
LWIMPAR_GAMMA	
LWIMPAR_NUMCOLS	
LWIMPAR_PIXELWIDTH	
LWIMPAR_WHITEPOINT	
LWIMTYP_GREY8	
LWIMTYP_GREYFP	
LWIMTYP_INDEX8	
LWIMTYP_RGB24	
LWIMTYP_RGBA32	
LWIMTYP_RGBAFP	
LWIMTYP_RGBFP	
LWIMTYP_SPECIAL	
LWINF_BUILD	lwhost.h
LWINF_GETBUILD()	
LWINF_GETMAJOR()	
LWINF_GETMINOR()	
LWINF_MAJORREV	
LWINF_MINORREV	
LWINF_PRODINSP3D	
LWINF_PRODLWAV	
LWINF_PRODOTHER	
LWINF_PRODUCT	
LWINF_RESERVED	
LWINIT_PREVIEW	lwrender.h
LWINIT_RENDER	
LWINSTUPDATE_GLOBAL	lwhandler.h
LWINTERFACE_VERSION	
LWINTERFACEINFO_GLOBAL	lwrender.h
LWIO_ASCII	lwio.h
LWIO_BINARY	
LWIO_BINARY_IFF	
LWIO_OBJECT	
LWIO_SCENE	
LWIP_ASPECT()	lwimageio.h
LWIP_DONE()	
LWIP_FORWARD	lwrender.h
LWIP_NUMCOLORS()	lwimageio.h
LWIP_PIVOT	lwrender.h

LWIP_PIVOT_ROT	
LWIP_POSITION	
LWIP_RIGHT	
LWIP_ROTATION	
LWIP_SCALING	
LWIP_SENDLINE()	lwimageio.h
LWIP_SETMAP()	
LWIP_SETPARAM()	
LWIP_SETSIZE()	
LWIP_UP	lwrender.h
LWIP_W_FORWARD	
LWIP_W_POSITION	
LWIP_W_RIGHT	
LWIP_W_UP	
LWIPT_ANGLE	lwtxtr.h
LWIPT_DISTANCE	
LWIPT_FLOAT	
LWIPT_PERCENT	
LWISM_BICUBIC	lwimage.h
LWISM_BILINEAR	
LWISM_BSPLINE	
LWISM_MEDIAN	
LWISM_SUBSAMPLING	
LWISM_SUPERSAMPLING	
LWITEM_ALL	lwrender.h
LWITEM_CAUSTICS	
LWITEM_NULL	
LWITEM_RADIOSITY	
LWITEMF_ACTIVE	
LWITEMF_FULLTIME_IK	
LWITEMF_GOAL_ORIENT	
LWITEMF_LOCKED	
LWITEMF_REACH_GOAL	
LWITEMF_SELECTED	
LWITEMF_SHOWCHANNELS	
LWITEMF_SHOWCHILDREN	
LWITEMF_UNAFFECT_BY_IK	
LWITEMINFO_GLOBAL	
LWITEMMOTION_HCLASS	lwmotion.h
LWITEMMOTION_ICLASS	
LWITEMMOTION_VERSION	
LWIVIS_HIDDEN	lwrender.h
LWIVIS_VISIBLE	
LWKEY_BIAS	lwenvel.h
LWKEY_CONTINUITY	
LWKEY_PARAM_0	
LWKEY_PARAM_1	
LWKEY_PARAM_2	
LWKEY_PARAM_3	
LWKEY_SHAPE	
LWKEY_TENSION	
LWKEY_TIME	
LWKEY_VALUE	
LWLAYOUTGENERIC_CLASS	lwgeneric.h
LWLAYOUTGENERIC_VERSION	
LWLAYOUTTOOL_CLASS	lwlaytool.h
LWLAYOUTTOOL_VERSION	
LWLFALL_INV_DIST	lwrender.h
LWLFALL_INV_DIST_2	
LWLFALL_LINEAR	
LWLFALL_OFF	
LWLFL_CACHE_SHAD_MAP	
LWLFL_FIT_CONE	
LWLFL_LENS_FLARE	

LWLFL_LIMITED_RANGE	
LWLFL_NO_CAUSTICS	
LWLFL_NO_DIFFUSE	
LWLFL_NO_OPENGL	
LWLFL_NO_SPECULAR	
LWLFL_VOLUMETRIC	
LWLIGHT_AREA	
LWLIGHT_DISTANT	
LWLIGHT_LINEAR	
LWLIGHT_POINT	
LWLIGHT_SPOT	
LWLIGHTINFO_GLOBAL	
LWLMONFUNCS_GLOBAL	lwmonitor.h
LWLOAD_DEPTH()	lwio.h
LWLOAD_END()	
LWLOAD_FIND()	
LWLOAD_FP()	
LWLOAD_I1()	
LWLOAD_I2()	
LWLOAD_I4()	
LWLOAD_ID()	
LWLOAD_STR()	
LWLOAD_U1()	
LWLOAD_U2()	
LWLOAD_U4()	
LWLOC_LANGID	lwhost.h
LWLOC_RESERVED	
LWLOCALEINFO_GLOBAL	
LWLPAT_DASH	lwcustobj.h
LWLPAT_DOT	
LWLPAT_LONGDOT	
LWLPAT_SOLID	
LWLSHAD_MAP	lwrender.h
LWLSHAD_OFF	
LWLSHAD_RAYTRACE	
LWM_MODE_SELECTION	lwmodeler.h
LWM_MODE_SYMMETRY	
LWM_VMAP_MORPH	
LWM_VMAP_TEXTURE	
LWM_VMAP_WEIGHT	
LWMAST_LAYOUT	lwmaster.h
LWMAST_SCENE	
LWMASTER_HCLASS	
LWMASTER_ICLASS	
LWMASTER_VERSION	
LWMATH_H	lwmath.h
LWMESHEDIT_CLASS	lwmeshedt.h
LWMESHEDIT_VERSION	
LWMESHEDITTOOL_CLASS	lwmodtool.h
LWMESHEDITTOOL_VERSION	
LWMESSAGEFUNCS_GLOBAL	lwhost.h
LWMODCOMMAND_CLASS	lwcmdseq.h
LWMODCOMMAND_VERSION	
LWMOTCTL_ALIGN_TO_PATH	lwrender.h
LWMOTCTL_IK	
LWMOTCTL_KEYFRAMES	
LWMOTCTL_TARGETING	
LWMTUTILFUNCS_GLOBAL	lwmtutil.h
LWOBJECTFUNCS_GLOBAL	lwmeshes.h
LWOBJECTIMPORT_CLASS	lwobjimp.h
LWOBJECTIMPORT_VERSION	
LWOBJECTINFO_GLOBAL	lwrender.h
LWOBJF_MORPH_MTSE	
LWOBJF_MORPH_SURFACES	

LWOBJF_UNAFFECT_BY_FOG	
LWOBJF_UNSEEN_BY_CAMERA	
LWOBJF_UNSEEN_BY_RAYS	
LWOBJIM_ABORTED	lwobjimp.h
LWOBJIM_BADFILE	
LWOBJIM_FAILED	
LWOBJIM_NOREC	
LWOBJIM_OK	
LWOBJREP_NONE	lwobjrep.h
LWOBJREP_PREVIEW	
LWOBJREP_RENDER	
LWOBJREPLACEMENT_HCLASS	
LWOBJREPLACEMENT_ICLASS	
LWOBJREPLACEMENT_VERSION	
LWOSHAD_CAST	lwrender.h
LWOSHAD_RECEIVE	
LWOSHAD_SELF	
LWOVIS_BOUNDINGBOX	
LWOVIS_FFWIREFRAME	
LWOVIS_HIDDEN	
LWOVIS_SHADED	
LWOVIS_TEXTURED	
LWOVIS_VERTICES	
LWOVIS_WIREFRAME	
LWP_0_DLOG	lwpanel.h
LWP_0_DOIT	
LWP_0_DRAG	
LWP_0_STAT	
LWP_1_BG	
LWP_1_DLOG	
LWP_1_DOIT	
LWP_1_DRAG	
LWP_1_STAT	
LWP_2_BG	
LWP_2_DLOG	
LWP_2_DOIT	
LWP_2_DRAG	
LWP_2_STAT	
LWP_3_BG	
LWP_3_DLOG	
LWP_3_DOIT	
LWP_3_DRAG	
LWP_3_STAT	
LWP_4_DLOG	
LWP_4_DOIT	
LWP_4_DRAG	
LWP_4_STAT	
LWP_5_DLOG	
LWP_5_DOIT	
LWP_5_DRAG	
LWP_5_STAT	
LWP_AQUA1	
LWP_AQUA2	
LWP_AQUA3	
LWP_AQUA4	
LWP_AQUA5	
LWP_AQUA6	
LWP_BG	
LWP_BLACK	
LWP_CANARY1	
LWP_CANARY2	
LWP_CANARY3	
LWP_CANARY4	
LWP_CANARY5	

LWP_CANARY6	
LWP_EDIT_BG	
LWP_EDIT_DIS	
LWP_EDIT_IMG	
LWP_EDIT_SEL	
LWP_GRAY1	
LWP_GRAY2	
LWP_GRAY3	
LWP_GRAY4	
LWP_GRAY5	
LWP_GRAY6	
LWP_GRAY7	
LWP_GRAY8	
LWP_HEADLINE	
LWP_HILIGHT	
LWP_INFO_BG	
LWP_INFO_DIS	
LWP_INFO_IMG	
LWP_INVERT	
LWP_LAVENDER1	
LWP_LAVENDER2	
LWP_LAVENDER3	
LWP_LAVENDER4	
LWP_LAVENDER5	
LWP_LAVENDER6	
LWP_LOWERED	
LWP_NONE	
LWP_NORMAL	
LWP_OLIVE1	
LWP_OLIVE2	
LWP_OLIVE3	
LWP_OLIVE4	
LWP_OLIVE5	
LWP_OLIVE6	
LWP_RAISED	
LWP_SHADOW	
LWP_WHITE	
LWPANEL_H	
LWPANELFUNCS_GLOBAL	
LWPANELS_API_VERSION	
LWPFF_BEFOREVOLUME	lwfilter.h
LWPFF_RAYTRACE	
LWPIXELFILTER_HCLASS	
LWPIXELFILTER_ICLASS	
LWPIXELFILTER_VERSION	
LWPOLTYPE_BONE	lwmeshes.h
LWPOLTYPE_CURV	
LWPOLTYPE_FACE	
LWPOLTYPE_MBAL	
LWPOLTYPE_PTCH	
LWPREVIEW_H	lwpreview.h
LWPREVIEWFUNCS_GLOBAL	
LWPRODUCTINFO_GLOBAL	lwhost.h
LWPSB_AGE	lwprtcl.h
LWPSB_CAGE	
LWPSB_ENB	
LWPSB_FCE	
LWPSB_ID	
LWPSB_LNK	
LWPSB_MAS	
LWPSB_POS	
LWPSB_PRS	
LWPSB_RGBA	
LWPSB_ROT	

LWPSB_SCL	
LWPSB_SIZ	
LWPSB_TMP	
LWPSB_VEL	
LWPSBT_CHAR	
LWPSBT_FLOAT	
LWPSBT_INT	
LWPST_ALIVE	
LWPST_DEAD	
LWPST_LIMBO	
LWPST_PRTCL	
LWPST_TRAIL	
LWPSYSFUNCS_GLOBAL	
LWPTAG_PART	lwmeshes.h
LWPTAG_SURF	
LWPTAG_TXUV	
LWRASTERFUNCS_GLOBAL	lwpanel.h
LWROPT_DEPTHOFFIELD	lwrender.h
LWROPT_EVENFIELDS	
LWROPT_FIELDS	
LWROPT_LIMITEDREGION	
LWROPT_MOTIONBLUR	
LWROPT_PARTICLEBLUR	
LWROPT_REFLECTTRACE	
LWROPT_REFRACTTRACE	
LWROPT_SHADOWTRACE	
LWRTYPE_QUICK	
LWRTYPE_REALISTIC	
LWRTYPE_WIRE	
LWSAF_SHADOW	lwshader.h
LWSAVE_BEGIN()	lwio.h
LWSAVE_DEPTH()	
LWSAVE_END()	
LWSAVE_FP()	
LWSAVE_I1()	
LWSAVE_I2()	
LWSAVE_I4()	
LWSAVE_ID()	
LWSAVE_STR()	
LWSAVE_U1()	
LWSAVE_U2()	
LWSAVE_U4()	
LWSCENECONVERTER_CLASS	lwscenecv.h
LWSCENECONVERTER_VERSION	
LWSCENEINFO_GLOBAL	lwrender.h
LWSHADER_HCLASS	lwshader.h
LWSHADER_ICLASS	
LWSHADER_VERSION	
LWSHELF_H	lwshelf.h
LWSHELFFUNCS_GLOBAL	
LWSHF_COLOR	lwshader.h
LWSHF_DIFFUSE	
LWSHF_ETA	
LWSHF_LUMINOUS	
LWSHF_MIRROR	
LWSHF_NORMAL	
LWSHF_RAYTRACE	
LWSHF_ROUGH	
LWSHF_SPECULAR	
LWSHF_TRANSLUCENT	
LWSHF_TRANSP	
LWSRVF_DISABLED	lwrender.h
LWSRVF_HIDDEN	
LWSTATEQUERYFUNCS_GLOBAL	lwmodeler.h

LWSURFACEFUNCS_GLOBAL	lwsurf.h
LWSURFEDFUNCS_GLOBAL	lwsurfed.h
LWSYS_LAYOUT	lwhost.h
LWSYS_MODELER	
LWSYS_SCREAMERNET	
LWSYS_SERIALBITS	
LWSYS_TYPEBITS	
LWSYSTEMID_GLOBAL	
LWT__LAST	lwpanel.h
LWT_AREA	
LWT_BOOLEAN	
LWT_CHOICE	
LWT_CUSTOM	
LWT_DIRTY_HELPTEXT	lwtool.h
LWT_DIRTY_WIREFRAME	
LWT_DISTANCE	lwpanel.h
LWT_EVENT_ACTIVATE	lwtool.h
LWT_EVENT_DROP	
LWT_EVENT_RESET	
LWT_FLOAT	lwpanel.h
LWT_FONT	
LWT_INTEGER	
LWT_LAYERS	
LWT_LWITEM	
LWT_MLIST	
LWT_NULL	
LWT_PERCENT	
LWT_POPUP	
LWT_RANGE	
LWT_STRING	
LWT_SURFACE	
LWT_TEST_ACCEPT	lwmodtool.h
LWT_TEST_CLONE	
LWT_TEST_NOTHING	
LWT_TEST_REJECT	
LWT_TEST_UPDATE	
LWT_TEXT	lwpanel.h
LWT_TREE	
LWT_VDIST	
LWT_VFLOAT	
LWT_VINT	
LWT_XPANEL	
LWTEXF_AALIAS	lwtexture.h
LWTEXF_AXIS	
LWTEXF_DISPLACE	
LWTEXF_GRAD	
LWTEXF_HV_SRF	
LWTEXF_HV_VOL	
LWTEXF_SELF_COLOR	
LWTEXF_SLOWPREVIEW	
LWTEXTURE_HCLASS	
LWTEXTURE_ICLASS	
LWTEXTURE_VERSION	
LWTEXTUREFUNCS_GLOBAL	lwtxtr.h
LWTIMEINFO_GLOBAL	lwrender.h
LWTOOLF_ALT_BUTTON	lwtool.h
LWTOOLF_CONS_X	
LWTOOLF_CONS_Y	
LWTOOLF_CONSTRAIN	
LWTOOLF_MULTICLICK	
LWTXEF_AXISX	lwtexture.h
LWTXEF_AXISY	
LWTXEF_AXISZ	
LWTXEF_COLOR	

LWTXEF_DISPLACE	
LWTXEF_VECTOR	
LWTXTREDFUNCS_GLOBAL	lwtxtred.h
LWVECF_0	lwrender.h
LWVECF_1	
LWVECF_2	
LWVEF_COLOR	lwvolume.h
LWVEF_OPACITY	
LWVEF_RAYTRACE	
LWVIEW_CAMERA	lwcustobj.h
LWVIEW_LIGHT	
LWVIEW_PERSP	
LWVIEW_SCHEMA	
LWVIEW_XY	
LWVIEW_XZ	
LWVIEW_ZY	
LWVMAP_MNVW	lwmeshes.h
LWVMAP_MORF	
LWVMAP_PICK	
LWVMAP_RGB	
LWVMAP_RGBA	
LWVMAP_SPOT	
LWVMAP_TXUV	
LWVMAP_WGHT	
LWVOLF_REFLECTIONS	lwvolume.h
LWVOLF_REFRACTIONS	
LWVOLF_SHADOWS	
LWVOLUMETRIC_HCLASS	
LWVOLUMETRIC_ICLASS	
LWVOLUMETRIC_VERSION	
LWVP_ANGLE	lwvparm.h
LWVP_COLOR	
LWVP_DIST	
LWVP_FLOAT	
LWVP_PERCENT	
LWVPARMFUNCS_GLOBAL	
LWVPDT_COLOR	
LWVPDT_DISPLACEMENT	
LWVPDT_NOTXTR	
LWVPDT_PERCENT	
LWVPDT_SCALAR	
LWVPDT_VECTOR	
LWVPEC_ENVNEW	
LWVPEC_ENVOLD	
LWVPEC_ENVTRACK	
LWVPEC_ENVUPDATE	
LWVPEC_TEXNEW	
LWVPEC_TEXOLD	
LWVPEC_TXAUTOSIZE	
LWVPEC_TXTRACK	
LWVPEC_TXUPDATE	
LWVPF_VECTOR	
LWVPSF_ENV	
LWVPSF_TEX	
LWWIRE_ABSOLUTE	lwtool.h
LWWIRE_DASH	
LWWIRE_RELATIVE	
LWWIRE_SCREEN	
LWWIRE_SOLID	
LWWIRE_TEXT_C	
LWWIRE_TEXT_L	
LWWIRE_TEXT_R	
LWXP_FORM	lwxpanel.h
LWXP_VIEW	

LWXPANELFUNCS_GLOBAL	
LWXPBDR_DOWN	
LWXPBDR_NONE	
LWXPBDR_UP	
LWXPDLG_DONE	
LWXPDLG_OKCANCEL	
LWXPDLG_OKONLY	
LWXPDLG_YESNO	
LWXPDLG_YESNOALL	
LWXPDLG_YESNOCAN	
LWXPEVENT_FOCUS	
LWXPEVENT_HIT	
LWXPEVENT_LOSEFOCUS	
LWXPEVENT_TRACK	
LWXPEVENT_VALUE	
LWXPRC_DFLT	
LWXPRC_DRAW	
LWXPRC_FULL	
LWXPRC_NONE	
LWXPREQ_DIR	
LWXPREQ_LOAD	
LWXPREQ_SAVE	

MAX()	lwmath.h
MCLICK()	lwpanel.h
MIN()	lwmath.h
MINIHSV_CTL()	lwpanel.h
MINIRGB_CTL()	
MINISLIDER_CTL()	
MOD_MACHINE	lwmodule.h
MOD_SYSSYNC	
MOD_SYSVER	
MON_DONE()	lwmonitor.h
MON_INCR()	
MON_INIT()	
MON_STEP()	
MOUSE_DOWN	lwpanel.h
MOUSE_LEFT	
MOUSE_MID	
MOUSE_RIGHT	
MOVE_CON()	
MOVE_PAN()	
MULTILIST_CTL()	
MX_HCHOICE	
MX_POPUP	
MX_VCHOICE	

NODE_FLAG_EXPND	
NODE_FLAG_WRITE	
NULL	lwtypes.h

OPENGL_CTL()	lwpanel.h
OPLYR_ALL	lwmodeler.h
OPLYR_BG	
OPLYR_EMPTY	
OPLYR_FG	
OPLYR_NONEMPTY	
OPLYR_PRIMARY	
OPLYR_SELECT	
OPSEL_DIRECT	
OPSEL_GLOBAL	
OPSEL_MODIFY	lwmeshedt.h
OPSEL_USER	lwmodeler.h

PALETTE_CTL()	lwpanel.h
PALETTESIZE()	
PAN_CREATE()	
PAN_FLAGS	
PAN_GETH()	
PAN_GETVERSION()	
PAN_GETW()	
PAN_GETX()	
PAN_GETY()	
PAN_H	
PAN_HOSTDISPLAY	
PAN_KILL()	
PAN_LWINSTANCE	
PAN_MOUSEBUTTON	
PAN_MOUSEMOVE	
PAN_MOUSEX	
PAN_MOUSEY	
PAN_PANFUN	
PAN_POST()	
PAN_QUALIFIERS	
PAN_RESULT	
PAN_SETDATA()	
PAN_SETDRAW()	
PAN_SETH()	
PAN_SETKEYS()	
PAN_SETW()	
PAN_TITLE	
PAN_TO_FRONT	
PAN_USERACTIVATE	
PAN_USERCLOSE	
PAN_USERDATA	
PAN_USERDRAW	
PAN_USERKEYS	
PAN_USERKEYUPS	
PAN_USEROPEN	
PAN_VERSION	
PAN_W	
PAN_X	
PAN_Y	
PANEL_SERVICES_NAME	
PANF_ABORT	
PANF_BLOCKING	
PANF_CANCEL	
PANF_FRAME	
PANF_MOUSETRAP	
PANF_NOBUTT	
PANF_PASSALLKEYS	
PANF_RESIZE	
PERCENT_CTL()	
PI	lwmath.h
PIKITEM_CTL()	lwpanel.h
POPDOWN_CTL()	
POPUP_CTL()	

RADIANS()	lwmath.h
RCLICK()	lwpanel.h
REDRAW_CON()	
RENDER_CON()	
RGB_()	
RGB_CTL()	
RGBVEC_CTL()	

SAVE_CTL()	
SAVEBUTTON_CTL()	

ServerUserName	lwserver.h
SET_FLOAT()	lwpanel.h
SET_FVEC()	
SET_INT()	
SET_IVEC()	
SET_STR()	
SETV_FVEC()	
SETV_IVEC()	
SHLC_DFLT	lwshelf.h
SHLC_FORCE	
SHLC_NOWAY	
SHLF_ASC	
SHLF_BIN	
SHLF_SEP	
SLIDER_CTL()	lwpanel.h
SRVTAG_BUTTONNAME	lwserver.h
SRVTAG_CMDGROUP	
SRVTAG_DESCRIPTION	
SRVTAG_ENABLE	
SRVTAG_MENU	
SRVTAG_USERNAME	
STR_CTL()	lwpanel.h
STRRO_CTL()	
SURF_ADTR	lwsurf.h
SURF_ALPH	
SURF_AVAL	
SURF_BUF1	
SURF_BUF2	
SURF_BUF3	
SURF_BUF4	
SURF_BUMP	
SURF_CLRF	
SURF_CLRH	
SURF_COLR	
SURF_DIFF	
SURF_GLOS	
SURF_GLOW	
SURF_GVAL	
SURF_LCOL	
SURF_LINE	
SURF_LSIZ	
SURF_LUMI	
SURF_RBLR	
SURF_REFL	
SURF_RFOP	
SURF_RIMG	
SURF_RIND	
SURF_RSAN	
SURF_SHRP	
SURF_SIDE	
SURF_SMAN	
SURF_SPEC	
SURF_TBLR	
SURF_TIMG	
SURF_TRAN	
SURF_TRNL	
SURF_TROP	
SURF_TSAN	
SURF_VCOL	
SWAP()	lwmath.h
SYSTEM_Ic()	lwpanel.h
TABCHOICE_CTL()	
TCC_ANY	lwtxtr.h

TCC_OBJECT	
TCC_WORLD	
TEF_ALL	lwtxtred.h
TEF_BLEND	
TEF_LAYERS	
TEF_OPACITY	
TEF_TYPE	
TEF_USEBTN	
TEXT_CTL()	lwpanel.h
THUM_35MM	lwxpanel.h
THUM_ANAW	
THUM_EURO	
THUM_FULL	
THUM_LAND	
THUM_LRG	
THUM_MED	
THUM_NTSC	
THUM_PORT	
THUM_SML	
THUM_SQ	
THUM_WIDE	
THUM_XLG	
TLT_GRAD	lwtxtr.h
TLT_IMAGE	
TLT_PROC	
TREE_CTL()	lwpanel.h
TRT_COLOR	lwtxtr.h
TRT_DISPLACEMENT	
TRT_PERCENT	
TRT_SCALAR	
TRT_VECTOR	
TRUECOLOR()	lwpanel.h
TWOPI	lwmath.h
TXEV_ALTER	lwtxtred.h
TXEV_DELETE	
TXEV_TRACK	
TXPRJ_CUBIC	lwtxtr.h
TXPRJ_CYLINDRICAL	
TXPRJ_FRONT	
TXPRJ_PLANAR	
TXPRJ_SPHERICAL	
TXPRJ_UVMAP	
TXRPT_EDGE	
TXRPT_MIRROR	
TXRPT_REPEAT	
TXRPT_RESET	
TXTAG_AA	
TXTAG_AAVAL	
TXTAG_AXIS	
TXTAG_COORD	
TXTAG_FALL	
TXTAG_HREPEAT	
TXTAG_HWRP	
TXTAG_IMAGE	
TXTAG_OPAC	
TXTAG_PIXBLEND	
TXTAG_POSI	
TXTAG_PROJ	
TXTAG_ROBJ	
TXTAG_ROTA	
TXTAG_SIZE	
TXTAG_VMAP	
TXTAG_WREPEAT	
TXTAG_WWRP	

UNGHOST_CON()	lwpanel.h
UNSLIDER_CTL()	

VADD()	lwmath.h
VADD3()	
VADDS()	
VADDS3()	
VCHOICE_CTL()	lwpanel.h
VCLR()	lwmath.h
VCPY()	
VCROSS()	
VDOT()	
VDRAGBUT_CTL()	lwpanel.h
VLEN()	lwmath.h
VMUL3()	
VSCL()	
VSCL3()	
VSET()	
VSLIDER_CTL()	lwpanel.h
VSUB()	lwmath.h
VSUB3()	

WBOOLBUTTON_CTL()	lwpanel.h
WBUTTON_CTL()	
WITEM_CTL()	
WPOPUP_CTL()	

XCALL_()	lwserver.h
XCALL_INIT	
XpADD()	lwxpanel.h
XpALIAS_()	
XPANEL_CTL()	lwpanel.h
XpBORDER()	lwxpanel.h
XpBUTNOTIFY()	
XpCALL()	
XpCHGNOTIFY()	
XpCHOXFORM()	
XpCLASS()	
XpCLRMINMAX()	
XpCTRLCFG()	
XpCTRLFRONT()	
XpDELETE()	
XpDESTROYNOTIFY()	
XpDIVADD()	
XpDIVADD_()	
XpDIVREM()	
XpDIVREM_()	
XpDLOGTYPE()	
XpDRAWCBFUNC()	
XpENABLE()	
XpENABLE_()	
XpENABLE_MAP_()	
XpENABLEMSG_()	
XpENABLEMSG_MAP_()	
XpEND	
XpFOCUS()	
XpGROUP_()	
XpH()	
XpHARDMINMAX()	
XpIMMUPD()	
XpIMMUPD_()	
XpINTXFORM()	
XpLABEL()	

```
XpLEFT()
XpLEFT_()
XpLINK()
XpLINK_()
XpMAX()
XpMIN()
XpNARROW()
XpNARROW_()
XpNEST()
XpORDER_()
XpORIENT()
XpPOPCMDFUNC()
XpPOPFUNCS()
XpPOPXFORM()
XpRANGE()
XpRESTORE()
XpRESTORE_()
XpRESTORE_OFF
XpRESTORE_ON
XpSENS()
XpSTACK_()
XpSTACK_MAP_()
XpSTEP()
XpSTRLIST()
XpSUBCALL()
XpTABS_()
XPTAG_ADD
XPTAG_ALIAS
XPTAG_AUTORESTORE
XPTAG_BORDER
XPTAG_BUTNOTIFY
XPTAG_CALL
XPTAG_CHGNOTIFY
XPTAG_CLASS
XPTAG_CLRMINMAX
XPTAG_CTRLCFG
XPTAG_CTRLFRONT
XPTAG_DELETE
XPTAG_DESTROYNOTIFY
XPTAG_DIVADD
XPTAG_DIVREM
XPTAG_DLOGTYPE
XPTAG_DRAWCBFUNC
XPTAG_ENABLE
XPTAG_END
XPTAG_FOCUS
XPTAG_GROUP
XPTAG_HARDMINMAX
XPTAG_IMMUPD
XPTAG_INTXFORM
XPTAG_LABEL
XPTAG_LEFT
XPTAG_LINK
XPTAG_MAX
XPTAG_MIN
XPTAG_NARROW
XPTAG_NEST
XPTAG_NULL
XPTAG_ORDER
XPTAG_ORIENT
XPTAG_POPCMDFUNC
XPTAG_POPFUNCS
XPTAG_RANGE
XPTAG_SENS
```

XPTAG_STACK
XPTAG_STEP
XPTAG_STRLIST
XPTAG_TABS
XPTAG_TRACK
XPTAG_UNALIAS
XPTAG_VALUE
XPTAG_VECLABEL
XPTAG_XREQCFG
XPTAG_ZOOMCBFUNC
XpTRACK()
XpUNALIAS_()
XpVALUE()
XpVECLABEL()
XpXREQCFG()
XpZOOMCBFUNC()

Structure Members

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Symbol	Member of	Header
a	LWPixelRGBA32	lwimageio.h
a	LWPixelRGBAFP	
activate	ServerRecord	lwserver.h
addBuf	LWPSysFuncs	lwprtcl.h
addControl	LWPanelFuncs	lwpanel.h
addCtrl	DynaReqFuncs	lwdyna.h
addCurve	MeshEditOp	lwmeshedt.h
addFace	MeshEditOp	
addIPnt	MeshEditOp	
addNamedPreset	LWShelfFuncs	lwshelf.h
addParticle	LWPSysFuncs	lwprtcl.h
addPatch	MeshEditOp	lwmeshedt.h
addPoint	MeshEditOp	
addPoly	MeshEditOp	
addPreset	LWShelfFuncs	lwshelf.h
addQuad	MeshEditOp	lwmeshedt.h
addSample	LWVolumeAccess	lwvolume.h
addTransparency	LWShaderAccess	lwshader.h
addTri	MeshEditOp	lwmeshedt.h
adjust	LWLayoutToolFuncs	lwlaytool.h
adjust	LWToolFuncs	lwtool.h
alertLevel	LWInterfaceInfo	lwrender.h
alpha	LWImageList	lwimage.h
alphaSpot	LWImageList	
ambient	LWLightInfo	lwrender.h
amp	LWTextureAccess	lwtexture.h
area	LWPanControlDesc	lwpanel.h
argument	LWModCommand	lwcmdseq.h
aspect	LWAnimLoaderHandler	lwanimlod.h
attach	LWPSysFuncs	lwprtcl.h
ax	LWToolEvent	lwtool.h
axis	LWTextureAccess	lwtexture.h
axis	LWToolEvent	lwtool.h
axis	LWWireDrawAccess	
ay	LWToolEvent	
az	LWToolEvent	
b	LWPixelRGB24	lwimageio.h
b	LWPixelRGBA32	
b	LWPixelRGBAFP	
b	LWPixelRGBFP	
backdrop	LWBackdropInfo	lwrender.h
baseName	LWFileReqLocal	lwdialog.h
bbox	LWStateQueryFuncs	lwmodeler.h
begin	LWAnimFrameAccess	lwanimlod.h
begin	LWAnimSaverHandler	lwanimsav.h
begin	LWFrameBufferHandler	lwframbuf.h
begin	LWImageLoaderLocal	lwimageio.h
beginBlk	LWSaveState	lwio.h
bg	LWCompInfo	lwrender.h

bits	DynaStringHint	lwdyna.h
bitval	DyBitfieldHint	
blitPanel	LWRasterFuncs	lwpanel.h
blue	LWColorPickLocal	lwdialog.h
blurLength	LWCameraInfo	lwrender.h
boneSource	LWObjectInfo	
bounces	LWShaderAccess	lwshader.h
boxThreshold	LWInterfaceInfo	lwrender.h
buf	DyValString	lwdyna.h
buf	LWValString	lwpanel.h
bufLen	DyValString	lwdyna.h
bufLen	LWFileReqLocal	lwdialog.h
bufLen	LWValString	lwpanel.h
build	LWMeshEditTool	lwmodtool.h
bumpHeight	LWMicropol	lwtxtr.h
bumpHeight	LWShaderAccess	lwshader.h
byName	LWSurfaceFuncs	lwsurf.h
byObject	LWSurfaceFuncs	
chan	LWChannelAccess	lwchannel.h
changeID	LWItemFuncs	lwrender.h
chanGroup	LWItemInfo	
chanGrp	LWSurfaceFuncs	lwsurf.h
channelEnvelope	LWChannelInfo	lwenvel.h
channelEvaluate	LWChannelInfo	
channelName	LWChannelAccess	lwchannel.h
channelName	LWChannelInfo	lwenvel.h
channelParent	LWChannelInfo	
channelType	LWChannelInfo	
chc	DynaStringHint	lwdyna.h
choice	DyReqControlDesc	
choice	LWPanControlDesc	lwpanel.h
cid	LWXPanelControl	lwxpanel.h
circle	LWCustomObjAccess	lwcustobj.h
circle	LWWireDrawAccess	lwtool.h
className	ServerRecord	lwserver.h
cleanup	LWPSysFuncs	lwprtcl.h
cleanup	LWRenderFuncs	lwrender.h
cleanup	LWTextureFuncs	lwtxtr.h
clear	LWFontListFuncs	lwmodeler.h
clear	LWImageList	lwimage.h
clipMap	LWObjectInfo	lwrender.h
close	LWAnimSaverHandler	lwanimsav.h
close	LWFrameBufferHandler	lwframbuf.h
close	LWPanelFuncs	lwpanel.h
close	LWPreviewFuncs	lwpreview.h
close	LWShelfFuncs	lwshelf.h
close	LWSurfEdFuncs	lwsurfed.h
close	LWTxtrEdFuncs	lwtxtred.h
closeLoad	LWFileIOFuncs	lwio.h
closeSave	LWFileIOFuncs	
cmenuCreate	ContextMenuFuncs	lwpanel.h
cmenuDeploy	ContextMenuFuncs	
cmenuDestroy	ContextMenuFuncs	
code	DyBitfieldHint	lwdyna.h
color	LWBackdropInfo	lwrender.h
color	LWEnvironmentAccess	lwenviron.h
color	LWFogInfo	lwrender.h
color	LWLightInfo	
color	LWShaderAccess	lwshader.h
color	LWVolumeSample	lwvolume.h
colorFL	LWShaderAccess	lwshader.h
colorHL	LWShaderAccess	
colRect	LWEnvironmentAccess	lwenviron.h

colWidth	LWPanMultiListBoxDesc	lwpanel.h
command	LWInterface	lwhandler.h
coneAngles	LWLightInfo	lwrender.h
context	LWMicropol	lwtxtr.h
context	LWTextureFuncs	
contextAddParam	LWTextureFuncs	
contextCreate	LWTextureFuncs	
contextDestroy	LWTextureFuncs	
controller	LWItemInfo	lwrender.h
copy	LWEnvelopeFuncs	lwenvel.h
copy	LWInstanceFuncs	lwhandler.h
copy	LWTextureFuncs	lwtxtr.h
copy	LWVParmFuncs	lwvparm.h
cosine	LWMicropol	lwtxtr.h
cosine	LWShaderAccess	lwshader.h
count	LWFontListFuncs	lwmodeler.h
count	LWLayoutToolFuncs	lwlaytool.h
count	LWPanLWItemDesc	lwpanel.h
count	LWToolFuncs	lwtool.h
countFn	LWPanListBoxDesc	lwpanel.h
countFn	LWPanMultiListBoxDesc	
countFn	LWPanPopupDesc	
countFn	LWPanTreeDesc	
create	DynaMonitorFuncs	lwdyna.h
create	DynaReqFuncs	
create	LWEnvelopeFuncs	lwenvel.h
create	LWGlobalPool	lwrender.h
create	LWImageUtil	lwimage.h
create	LWInstanceFuncs	lwhandler.h
create	LWLMonFuncs	lwmonitor.h
create	LWMTUtilFuncs	lwmtutil.h
create	LWPanelFuncs	lwpanel.h
create	LWPSysFuncs	lwprtcl.h
create	LWRasterFuncs	lwpanel.h
create	LWSurfaceFuncs	lwsurf.h
create	LWTextureFuncs	lwtxtr.h
create	LWVParmFuncs	lwvparm.h
create	LWXPanelFuncs	lwxpanel.h
createGroup	LWEnvelopeFuncs	lwenvel.h
createKey	LWEnvelopeFuncs	
ctrlclass	LWXPanelControl	lwxpanel.h
ctrlType	DynaReqFuncs	lwdyna.h
curFilename	LWObjReplacementAccess	lwobjrep.h
curFrame	LWObjReplacementAccess	
currentLayer	LWTxtrEdFuncs	lwtxtred.h
curTime	LWInterfaceInfo	lwrender.h
curTime	LWObjReplacementAccess	lwobjrep.h
curType	LWObjReplacementAccess	
curve	EDBoundCv	lwmeshedt.h
cust	DynaValue	lwdyna.h
cust	LWValue	lwpanel.h
data	LWColorPickLocal	lwdialog.h
data	LWGlobalService	lwglobsrv.h
data	LWLayoutGeneric	lwgeneric.h
data	LWMasterAccess	lwmaster.h
data	LWModCommand	lwcmdseq.h
data	LWMonitor	lwmonitor.h
data	LWObjectImport	lwobjimp.h
data	LWWireDrawAccess	lwtool.h
dataSize	LWPSBufDesc	lwprtcl.h
dataType	LWPSBufDesc	
datatype	LWXPanelDataDesc	lwxpanel.h
defVal	DyValFloat	lwdyna.h

defVal	DyValFVector	
defVal	DyValInt	
defVal	DyValIVector	
defVal	LWValFloat	lwpanel.h
defVal	LWValFVector	
defVal	LWValInt	
defVal	LWValIVector	
deleteTmp	LWSceneConverter	lwscenecv.h
deltaRaw	LWToolEvent	lwtool.h
deltaSnap	LWToolEvent	
depth	display_Metrics	lwpanel.h
depth	LWLoadState	lwio.h
depth	LWSaveState	
descln	LWInstanceFuncs	lwhandler.h
describe	LWXPanelFuncs	lwxpanel.h
destroy	DynaMonitorFuncs	lwdyna.h
destroy	DynaReqFuncs	
destroy	LWEnvelopeFuncs	lwenvel.h
destroy	LWImageUtil	lwimage.h
destroy	LWInstanceFuncs	lwhandler.h
destroy	LWLMonFuncs	lwmonitor.h
destroy	LWMeshInfo	lwmeshes.h
destroy	LWMTUtilFuncs	lwmtutil.h
destroy	LWPanelFuncs	lwpanel.h
destroy	LWPSysFuncs	lwprtcl.h
destroy	LWRasterFuncs	lwpanel.h
destroy	LWTextureFuncs	lwtxtr.h
destroy	LWVParmFuncs	lwvparm.h
destroy	LWXPanelFuncs	lwxpanel.h
destroyGroup	LWEnvelopeFuncs	lwenvel.h
destroyKey	LWEnvelopeFuncs	
detach	LWPSysFuncs	lwprtcl.h
diffuse	LWShaderAccess	lwshader.h
difSharpness	LWShaderAccess	
dir	LWEnvironmentAccess	lwenviron.h
dir	LWVolumeAccess	lwvolume.h
dirty	LWLayoutToolFuncs	lwlaytool.h
dirty	LWToolFuncs	lwtool.h
dispData	LWCustomObjAccess	lwcustobj.h
displayFlags	LWInterfaceInfo	lwrender.h
dispMap	LWObjectInfo	
dispMetrics	DrawFuncs	lwpanel.h
dissolve	LWObjectInfo	lwrender.h
dist	LWVolumeSample	lwvolume.h
done	LWAnimFrameAccess	lwanimlod.h
done	LWImageLoaderLocal	lwimageio.h
done	LWImageProtocol	
done	LWLayoutToolFuncs	lwlaytool.h
done	LWLMonFuncs	lwmonitor.h
done	LWMonitor	
done	LWObjectImport	lwobjimp.h
done	LWToolFuncs	lwtool.h
done	MeshEditOp	lwmeshedt.h
down	LWLayoutToolFuncs	lwlaytool.h
down	LWToolFuncs	lwtool.h
draw	LWControl	lwpanel.h
draw	LWLayoutToolFuncs	lwlaytool.h
draw	LWPanelFuncs	lwpanel.h
draw	LWToolFuncs	lwtool.h
drawBorder	DrawFuncs	lwpanel.h
drawBorder	LWRasterFuncs	
drawBorder	LWXPDrawFuncs	lwxpanel.h
drawBox	DrawFuncs	lwpanel.h
drawBox	LWRasterFuncs	

drawBox	LWXPDrawFuncs	lwxpanel.h
drawf	LWXPanelFuncs	
drawFuncs	LWPanelFuncs	lwpanel.h
drawLine	DrawFuncs	
drawLine	LWRasterFuncs	
drawLine	LWXPDrawFuncs	lwxpanel.h
drawPixel	DrawFuncs	lwpanel.h
drawPixel	LWRasterFuncs	
drawPixel	LWXPDrawFuncs	lwxpanel.h
drawRGBBox	DrawFuncs	lwpanel.h
drawRGBBox	LWRasterFuncs	
drawRGBBox	LWXPDrawFuncs	lwxpanel.h
drawRGBPixel	DrawFuncs	lwpanel.h
drawRGBPixel	LWRasterFuncs	
drawRGBPixel	LWXPDrawFuncs	lwxpanel.h
drawText	DrawFuncs	lwpanel.h
drawText	LWRasterFuncs	
drawText	LWXPDrawFuncs	lwxpanel.h
dx	LWToolEvent	lwtool.h
dy	LWToolEvent	
dynaUpdate	LWInterfaceInfo	lwrender.h
edgeColor	LWObjectInfo	
edgeOpts	LWObjectInfo	
edit	LWEnvelopeFuncs	lwenvel.h
editBegin	LWModCommand	lwcmdseq.h
editEnv	LWVParmFuncs	lwvparm.h
editTex	LWVParmFuncs	
egGet	LWEnvelopeFuncs	lwenvel.h
egSet	LWEnvelopeFuncs	
end	EDBoundCv	lwmeshedt.h
end	LWFilterAccess	lwfilter.h
end	LWMeshEditTool	lwmodtool.h
end	LWTxtrParamDesc	lwtxtr.h
endBlk	LWLoadState	lwio.h
endBlk	LWSaveState	
envAge	LWEnvelopeFuncs	lwenvel.h
envGroup	LWTextureFuncs	lwtxtr.h
eraseBox	LWRasterFuncs	lwpanel.h
error	LWMessageFuncs	lwhost.h
eta	LWShaderAccess	lwshader.h
evaluate	LWAnimLoaderHandler	lwanimlod.h
evaluate	LWChannelHandler	lwchannel.h
evaluate	LWCustomObjHandler	lwcustobj.h
evaluate	LWDisplacementHandler	lwdisplce.h
evaluate	LWEnvelopeFuncs	lwenvel.h
evaluate	LWEnvironmentHandler	lwenviron.h
evaluate	LWImageList	lwimage.h
evaluate	LWItemMotionHandler	lwmotion.h
evaluate	LWLayoutGeneric	lwgeneric.h
evaluate	LWMasterAccess	lwmaster.h
evaluate	LWModCommand	lwcmdseq.h
evaluate	LWObjReplacementHandler	lwobjrep.h
evaluate	LWPixelFilterHandler	lwfilter.h
evaluate	LWShaderHandler	lwshader.h
evaluate	LWTextureFuncs	lwtxtr.h
evaluate	LWTextureHandler	lwtexture.h
evaluate	LWVolumetricHandler	lwvolume.h
evaluateUV	LWTextureFuncs	lwtxtr.h
event	LWLayoutToolFuncs	lwlaytool.h
event	LWMasterHandler	lwmaster.h
event	LWToolFuncs	lwtool.h
eventCode	LWMasterAccess	lwmaster.h
eventData	LWMasterAccess	

excluded	LWObjectInfo	lwrender.h
execute	LWLayoutGeneric	lwgeneric.h
execute	LWMasterAccess	lwmaster.h
execute	LWModCommand	lwcmdseq.h
failedBuf	LWObjectImport	lwobjimp.h
failedLen	LWObjectImport	
falloff	LWBoneInfo	lwrender.h
falloff	LWLightInfo	
farClip	LWVolumeAccess	lwvolume.h
fg	LWCompInfo	lwrender.h
fgAlpha	LWCompInfo	
filename	LWImageList	lwimage.h
filename	LWImageLoaderLocal	lwimageio.h
filename	LWImageSaverLocal	
filename	LWObjectFuncs	lwmeshes.h
filename	LWObjectImport	lwobjimp.h
filename	LWObjectInfo	lwrender.h
filename	LWSceneConverter	lwscenecv.h
filename	LWSceneInfo	lwrender.h
fileType	LWFileReqLocal	lwdialog.h
find	LWGlobalPool	lwrender.h
findBlk	LWLoadState	lwio.h
findKey	LWEnvelopeFuncs	lwenvel.h
first	LWGlobalPool	lwrender.h
first	LWImageList	lwimage.h
first	LWItemInfo	lwrender.h
first	LWSurfaceFuncs	lwsurf.h
firstChild	LWItemInfo	lwrender.h
firstLayer	LWTextureFuncs	lwtxtr.h
flags	EDPointInfo	lwmeshedt.h
flags	EDPolygonInfo	
flags	LWBoneInfo	lwrender.h
flags	LWCameraInfo	
flags	LWChannelHandler	lwchannel.h
flags	LWCustomObjAccess	lwcustobj.h
flags	LWCustomObjHandler	
flags	LWDisplacementHandler	lwdisplce.h
flags	LWEnvironmentHandler	lwenviron.h
flags	LWFogInfo	lwrender.h
flags	LWImageFilterHandler	lwfilter.h
flags	LWItemInfo	lwrender.h
flags	LWItemMotionHandler	lwmotion.h
flags	LWLightInfo	lwrender.h
flags	LWMasterHandler	lwmaster.h
flags	LWObjectInfo	lwrender.h
flags	LWPixelFilterHandler	lwfilter.h
flags	LWShaderAccess	lwshader.h
flags	LWShaderHandler	
flags	LWTextureAccess	lwtexture.h
flags	LWTextureHandler	
flags	LWToolEvent	lwtool.h
flags	LWTxtrParamDesc	lwtxtr.h
flags	LWVolumeAccess	lwvolume.h
flags	LWVolumetricHandler	
flt	DynaValue	lwdyna.h
flt	LWValue	lwpanel.h
focalDistance	LWCameraInfo	lwrender.h
focalLength	LWCameraInfo	
fog	LWObjectInfo	
formGet	LWXPanelFuncs	lwxpanel.h
formSet	LWXPanelFuncs	
fovAngles	LWCameraInfo	lwrender.h
frame	LWChannelAccess	lwchannel.h

frame	LWFilterAccess	lwfilter.h
frame	LWItemMotionAccess	lwmotion.h
frame	LWTimeInfo	lwrender.h
frameCount	LWAnimLoaderHandler	lwanimlod.h
frameEnd	LWSceneInfo	lwrender.h
frameHeight	LWSceneInfo	
frameRate	LWAnimLoaderHandler	lwanimlod.h
framesPerSecond	LWSceneInfo	lwrender.h
frameStart	LWSceneInfo	
frameStep	LWSceneInfo	
frameWidth	LWSceneInfo	
frustum	LWVolumeAccess	lwvolume.h
fStop	LWCameraInfo	lwrender.h
fullName	LWFileReqLocal	lwdialog.h
fvec	DynaValue	lwdyna.h
fvec	LWValue	lwpanel.h
g	LWPixelRGB24	lwimageio.h
g	LWPixelRGBA32	
g	LWPixelRGBAFP	
g	LWPixelRGBFP	
generalFlags	LWInterfaceInfo	lwrender.h
get	LWControl	lwpanel.h
get	LWPanelFuncs	
getBitmap	LWPreviewFuncs	lwpreview.h
getBufData	LWPSysFuncs	lwprtcl.h
getBufID	LWPSysFuncs	
getCamera	LWPreviewFuncs	lwpreview.h
getChannel	LWChannelAccess	lwchannel.h
getColorVMap	LWSurfaceFuncs	lwsurf.h
getData	LWXPanelFuncs	lwxpanel.h
getEnv	LWSurfaceFuncs	lwsurf.h
getEnv	LWVParmFuncs	lwvparm.h
getFlt	LWSurfaceFuncs	lwsurf.h
getImg	LWSurfaceFuncs	
getInfo	LWImageUtil	lwimage.h
getInt	LWSurfaceFuncs	lwsurf.h
getLine	LWFilterAccess	lwfilter.h
getOpacity	LWVolumeAccess	lwvolume.h
getParam	LWItemMotionAccess	lwmotion.h
getParam	LWTextureFuncs	lwtxtr.h
getParticle	LWPSysFuncs	lwprtcl.h
getPCount	LWPSysFuncs	
getPixel	LWImageUtil	lwimage.h
getPixel	LWPreviewFuncs	lwpreview.h
getPosition	LWSurfEdFuncs	lwsurfed.h
getPSys	LWPSysFuncs	lwprtcl.h
getState	LWVParmFuncs	lwvparm.h
getTag	LWItemInfo	lwrender.h
getTex	LWSurfaceFuncs	lwsurf.h
getTex	LWVParmFuncs	lwvparm.h
getVal	LWPixelAccess	lwfilter.h
getVal	LWVParmFuncs	lwvparm.h
getView	LWPreviewFuncs	lwpreview.h
global	LWPanLWItemDesc	lwpanel.h
globalFun	LWPanelFuncs	
gNorm	LWMicropol	lwtxtr.h
gNorm	LWShaderAccess	lwshader.h
goal	LWItemInfo	lwrender.h
goalStrength	LWItemInfo	
green	LWColorPickLocal	lwdialog.h
groupName	LWChannelInfo	lwenvel.h
groupParent	LWChannelInfo	

h	LWEnvironmentAccess	lwenviron.h
handle	LWLayoutToolFuncs	lwlaytool.h
handle	LWPanelFuncs	lwpanel.h
handle	LWToolFuncs	lwtool.h
hasAlpha	LWImageList	lwimage.h
height	display_Metrics	lwpanel.h
height	LWFilterAccess	lwfilter.h
height	LWPanAreaDesc	lwpanel.h
height	LWPanTreeDesc	
height	LWPanXPanDesc	
help	LWLayoutToolFuncs	lwlaytool.h
help	LWToolFuncs	lwtool.h
hint	LWXPanelFuncs	lwxpanel.h
hotFunc	LWColorPickLocal	lwdialog.h
id	LWBlockIdent	lwio.h
ID	LWGlobalPool	lwrender.h
id	LWGlobalService	lwglobsrv.h
illuminate	LWMicropol	lwtxtr.h
illuminate	LWPixelAccess	lwfilter.h
illuminate	LWShaderAccess	lwshader.h
illuminate	LWVolumeAccess	lwvolume.h
index	LWFontListFuncs	lwmodeler.h
info	LWDisplacementAccess	lwdisplce.h
info	LWMessageFuncs	lwhost.h
infoFn	LWPanTreeDesc	lwpanel.h
init	LWLMonFuncs	lwmonitor.h
init	LWMonitor	
init	LWPSysFuncs	lwprtcl.h
init	LWRenderFuncs	lwrender.h
initMP	LWVParmFuncs	lwvparm.h
initUV	MeshEditOp	lwmeshedt.h
inst	LWAnimLoaderHandler	lwanimlod.h
inst	LWAnimSaverHandler	lwanimsav.h
inst	LWChannelHandler	lwchannel.h
inst	LWCustomObjHandler	lwcustobj.h
inst	LWDisplacementHandler	lwdisplce.h
inst	LWEnvironmentHandler	lwenviron.h
inst	LWFrameBufferHandler	lwframbuf.h
inst	LWHandler	lwhandler.h
inst	LWImageFilterHandler	lwfilter.h
inst	LWInterface	lwhandler.h
inst	LWItemHandler	lwrender.h
inst	LWItemMotionHandler	lwmotion.h
inst	LWMasterHandler	lwmaster.h
inst	LWObjReplacementHandler	lwobjrep.h
inst	LWPixelFilterHandler	lwfilter.h
inst	LWRenderHandler	lwrender.h
inst	LWShaderHandler	lwshader.h
inst	LWTextureHandler	lwtexture.h
inst	LWVolumetricHandler	lwvolume.h
instance	HostDisplayInfo	lwdisplay.h
instance	LWLayoutTool	lwlaytool.h
instance	LWMeshEditTool	lwmodtool.h
instance	LWTool	lwtool.h
intensity	LWLightInfo	lwrender.h
intv	DynaValue	lwdyna.h
intv	LWValue	lwpanel.h
ioMode	LWLoadState	lwio.h
ioMode	LWSaveState	
isColor	LWImageList	lwimage.h
isOpen	LWPreviewFuncs	lwpreview.h
isOpen	LWShelfFuncs	lwshelf.h
isOpen	LWSurfEdFuncs	lwsurfed.h

isOpen	LWTxtrEdFuncs	lwtxtred.h
item	DyChoiceHint	lwdyna.h
item	LWAnimSaverHandler	lwanimsav.h
item	LWChannelHandler	lwchannel.h
item	LWCustomObjHandler	lwcustobj.h
item	LWDisplacementHandler	lwdisplce.h
item	LWEnvironmentHandler	lwenviron.h
item	LWFrameBufferHandler	lwframbuf.h
item	LWImageFilterHandler	lwfilter.h
item	LWItemHandler	lwrender.h
item	LWItemMotionAccess	lwmotion.h
item	LWItemMotionHandler	
item	LWMasterHandler	lwmaster.h
item	LWObjReplacementHandler	lwobjrep.h
item	LWPixelFilterHandler	lwfilter.h
item	LWRenderHandler	lwrender.h
item	LWShaderHandler	lwshader.h
item	LWTextureHandler	lwtexture.h
item	LWVolumetricHandler	lwvolume.h
itemColor	LWInterfaceInfo	lwrender.h
itemFlags	LWInterfaceInfo	
itemID	LWTxtrParamDesc	lwtxtr.h
itemName	LWTxtrParamDesc	
items	DyReqChoiceDesc	lwdyna.h
items	LWPanChoiceDesc	lwpanel.h
itemType	LWPanLWItemDesc	
itemType	LWTxtrParamDesc	lwtxtr.h
itemVis	LWInterfaceInfo	lwrender.h
ivec	DynaValue	lwdyna.h
ivec	LWValue	lwpanel.h
jointComp	LWBoneInfo	lwrender.h
keyGet	LWEnvelopeFuncs	lwenvel.h
keySet	LWEnvelopeFuncs	
label	LWXPanelControl	lwxpanel.h
lastLayer	LWTextureFuncs	lwtxtr.h
layer	EDPointInfo	lwmeshedt.h
layer	EDPolygonInfo	
layer	LWObjectImport	lwobjimp.h
layerAdd	LWTextureFuncs	lwtxtr.h
layerEnvGroup	LWTextureFuncs	
layerEvaluate	LWTextureFuncs	
layerExists	LWObjectFuncs	lwmeshes.h
layerList	LWStateQueryFuncs	lwmodeler.h
layerMask	LWStateQueryFuncs	
layerMesh	LWObjectFuncs	lwmeshes.h
layerName	LWObjectFuncs	
layerNum	MeshEditOp	lwmeshedt.h
layerSetType	LWTextureFuncs	lwtxtr.h
layerType	LWTextureFuncs	
layerVis	LWObjectFuncs	lwmeshes.h
leafFn	LWPanTreeDesc	lwpanel.h
lFlags	LWObjectImport	lwobjimp.h
limitedRegion	LWSceneInfo	lwrender.h
limits	LWBoneInfo	
limits	LWItemInfo	
line	LWCustomObjAccess	lwcustobj.h
lineTo	LWWireDrawAccess	lwtool.h
list	LWPanLWItemDesc	lwpanel.h
listbox	LWPanControlDesc	
load	LWEnvelopeFuncs	lwenvel.h
load	LWFontListFuncs	lwmodeler.h

load	LWImageList	lwimage.h
load	LWInstanceFuncs	lwhandler.h
load	LWPSysFuncs	lwprtcl.h
load	LWShelfFuncs	lwshelf.h
load	LWTextureFuncs	lwtxtr.h
load	LWVParmFuncs	lwvparm.h
loadScene	LWLayoutGeneric	lwgeneric.h
lock	LWMTUtilFuncs	lwmtutil.h
lookAhead	LWItemInfo	lwrender.h
lookup	LWLayoutGeneric	lwgeneric.h
lookup	LWMasterAccess	lwmaster.h
lookup	LWModCommand	lwcmdseq.h
luma	LWImageList	lwimage.h
lumaSpot	LWImageList	
luminous	LWShaderAccess	lwshader.h
lwitem	LWPanControlDesc	lwpanel.h
maskColor	LWCameraInfo	lwrender.h
maskLimits	LWCameraInfo	
max	LWPanRangeDesc	lwpanel.h
maxAmt	LWFogInfo	lwrender.h
maxColors	display_Metrics	lwpanel.h
maxDist	LWFogInfo	lwrender.h
maxLayers	LWObjectFuncs	lwmeshes.h
maxSamplesPerPixel	LWSceneInfo	lwrender.h
meshInfo	LWObjectInfo	
metaballRes	LWObjectInfo	
min	LWPanRangeDesc	lwpanel.h
minAmt	LWFogInfo	lwrender.h
minDist	LWFogInfo	
minSamplesPerPixel	LWSceneInfo	
mirror	LWShaderAccess	lwshader.h
mode	LWEnvironmentAccess	lwenviron.h
mode	LWStateQueryFuncs	lwmodeler.h
monitor	LWFilterAccess	lwfilter.h
monitor	LWImageLoaderLocal	lwimageio.h
monitor	LWImageSaverLocal	
monitor	LWObjectImport	lwobjimp.h
morphAmount	LWObjectInfo	lwrender.h
morphTarget	LWObjectInfo	
move	LWLayoutToolFuncs	lwlaytool.h
move	LWToolFuncs	lwtool.h
moveFn	LWPanTreeDesc	lwpanel.h
moveTo	LWWireDrawAccess	lwtool.h
mp	PvSample	lwpreview.h
multiList	LWPanControlDesc	lwpanel.h
muscleFlex	LWBoneInfo	lwrender.h
name	LWFontListFuncs	lwmodeler.h
name	LWImageList	lwimage.h
name	LWItemInfo	lwrender.h
name	LWPSBufDesc	lwprtcl.h
name	LWSceneInfo	lwrender.h
name	LWSurfaceFuncs	lwsurf.h
name	LWTextureFuncs	lwtxtr.h
name	LWTxtrParamDesc	
name	LWXPanelDataDesc	lwxpanel.h
name	ServerRecord	lwserver.h
nameFn	LWPanListBoxDesc	lwpanel.h
nameFn	LWPanMultiListBoxDesc	
nameFn	LWPanPopupDesc	
nearClip	LWVolumeAccess	lwvolume.h
needAA	LWImageList	lwimage.h
newFilename	LWObjReplacementAccess	lwobjrep.h

newFrame	LWObjReplacementAccess	
newTime	LWObjReplacementAccess	
newTime	LWRenderFuncs	lwrender.h
newtime	LWTextureFuncs	lwtxtr.h
newType	LWObjReplacementAccess	lwobjrep.h
next	LWGlobalPool	lwrender.h
next	LWImageList	lwimage.h
next	LWItemInfo	lwrender.h
next	LWSurfaceFuncs	lwsurf.h
nextChannel	LWChannelInfo	lwenvel.h
nextChild	LWItemInfo	lwrender.h
nextControl	LWPanelFuncs	lwpanel.h
nextGroup	LWChannelInfo	lwenvel.h
nextKey	LWEnvelopeFuncs	
nextLayer	LWTextureFuncs	lwtxtr.h
noise	LWTextureFuncs	
numLayers	LWStateQueryFuncs	lwmodeler.h
numObjects	LWObjectFuncs	lwmeshes.h
numPnts	EDPolygonInfo	lwmeshedt.h
numPoints	LWMeshInfo	lwmeshes.h
numPoints	LWObjectInfo	lwrender.h
numPoints	LWSceneInfo	
numPolygons	LWMeshInfo	lwmeshes.h
numPolygons	LWObjectInfo	lwrender.h
numPolygons	LWSceneInfo	
numThreads	LWSceneInfo	
numVMaps	LWObjectFuncs	lwmeshes.h
o	LWVolumeAccess	lwvolume.h
oAxis	LWMicropol	lwtxtr.h
object	LWStateQueryFuncs	lwmodeler.h
objectID	LWObjReplacementAccess	lwobjrep.h
objID	LWMicropol	lwtxtr.h
objID	LWShaderAccess	lwshader.h
octaves	LWTextureAccess	lwtexture.h
oDist	LWVolumeAccess	lwvolume.h
okCancel	LWMessageFuncs	lwhost.h
opacity	LWVolumeSample	lwvolume.h
open	LWAnimSaverHandler	lwanimsav.h
open	LWFrameBufferHandler	lwframbuf.h
open	LWPanelFuncs	lwpanel.h
open	LWPreviewFuncs	lwpreview.h
open	LWShelfFuncs	lwshelf.h
open	LWSurfEdFuncs	lwsurfed.h
open	LWTxtrEdFuncs	lwtxtred.h
open	LWXPanelFuncs	lwxpanel.h
openLoad	LWFileIOFuncs	lwio.h
openSave	LWFileIOFuncs	
oPos	LWDisplacementAccess	lwdisplce.h
oPos	LWMicropol	lwtxtr.h
oPos	LWShaderAccess	lwshader.h
options	LWInterface	lwhandler.h
oScl	LWMicropol	lwtxtr.h
oXfrm	LWMicropol	
oXfrm	LWShaderAccess	lwshader.h
p	LWEnvironmentAccess	lwenviron.h
panel	LWInterface	lwhandler.h
panel	LWLayoutToolFuncs	lwlaytool.h
panel	LWToolFuncs	lwtool.h
param	LWItemInfo	lwrender.h
paramCleanup	LWTxtrParamFuncs	lwtxtr.h
paramEvaluate	LWTxtrParamFuncs	
paramTime	LWTxtrParamFuncs	

parent	LWItemInfo	lwrender.h
parent	LWObjectImport	lwobjimp.h
patchLevel	LWObjectInfo	lwrender.h
path	LWFileReqLocal	lwdialog.h
pause	LWFrameBufferHandler	lwframbuf.h
pickName	LWFileReqLocal	lwdialog.h
pivot	LWObjectImport	lwobjimp.h
pivotPoint	LWObjectFuncs	lwmeshes.h
pixelAspect	LWCameraInfo	lwrender.h
pixelAspect	LWSceneInfo	
pixX	display_Metrics	lwpanel.h
pixY	display_Metrics	
pnt	EDPointInfo	lwmeshedt.h
pntBasePos	LWMeshInfo	lwmeshes.h
pntMove	MeshEditOp	lwmeshedt.h
pntOtherPos	LWMeshInfo	lwmeshes.h
pntSelect	MeshEditOp	lwmeshedt.h
pntVGet	LWMeshInfo	lwmeshes.h
pntVIDGet	LWMeshInfo	
pntVLookup	LWMeshInfo	
pntVMap	MeshEditOp	lwmeshedt.h
pntVPGet	LWMeshInfo	lwmeshes.h
pntVPIDGet	LWMeshInfo	
pntVPMap	MeshEditOp	lwmeshedt.h
pntVSelect	LWMeshInfo	lwmeshes.h
point	LWCustomObjAccess	lwcustobj.h
point	LWDisplacementAccess	lwdisplce.h
point	LWObjectImport	lwobjimp.h
pointCount	MeshEditOp	lwmeshedt.h
pointInfo	MeshEditOp	
points	EDPolygonInfo	
pointScan	MeshEditOp	
pointVEval	MeshEditOp	
pointVGet	MeshEditOp	
pointVPGet	MeshEditOp	
pointVSet	MeshEditOp	
pol	EDPolygonInfo	
polFlag	MeshEditOp	
polFlags	LWMeshInfo	lwmeshes.h
polNum	LWMicropol	lwtxtr.h
polNum	LWShaderAccess	lwshader.h
polPnts	MeshEditOp	lwmeshedt.h
polSelect	MeshEditOp	
polSize	LWMeshInfo	lwmeshes.h
polSurf	MeshEditOp	lwmeshedt.h
polTag	LWMeshInfo	lwmeshes.h
polTag	LWObjectImport	lwobjimp.h
polTag	MeshEditOp	lwmeshedt.h
polType	LWMeshInfo	lwmeshes.h
polVertex	LWMeshInfo	
polyCount	MeshEditOp	lwmeshedt.h
polygon	LWMicropol	lwtxtr.h
polygon	LWObjectImport	lwobjimp.h
polygon	LWShaderAccess	lwshader.h
polygonSize	LWObjectInfo	lwrender.h
polyInfo	MeshEditOp	lwmeshedt.h
polyNormal	MeshEditOp	
polyScan	MeshEditOp	
polyTag	MeshEditOp	
popup	LWPanControlDesc	lwpanel.h
portAxis	LWToolEvent	lwtool.h
position	EDPointInfo	lwmeshedt.h
posRaw	LWToolEvent	lwtool.h
posSnap	LWToolEvent	

post	DynaReqFuncs	lwdyna.h
post	LWXPanelFuncs	lwxpanel.h
previewEnd	LWInterfaceInfo	lwrender.h
previewStart	LWInterfaceInfo	
previewStep	LWInterfaceInfo	
prevKey	LWEnvelopeFuncs	lwenvel.h
priv	LWInstanceFuncs	lwhandler.h
priv	LWMeshInfo	lwmeshes.h
priv_data	LWAnimFrameAccess	lwanimlod.h
priv_data	LWControl	lwpanel.h
priv_data	LWImageLoaderLocal	lwimageio.h
priv_data	LWImageProtocol	
priv_data	LWImageSaverLocal	
process	LWImageFilterHandler	lwfilter.h
projImage	LWLightInfo	lwrender.h
ptr	LWValPointer	lwpanel.h
ptr	LWValue	
pxRaw	LWToolEvent	lwtool.h
pxScale	LWWireDrawAccess	
pxSnap	LWToolEvent	
pyRaw	LWToolEvent	
pySnap	LWToolEvent	
quad	LWCustomObjAccess	lwcustobj.h
quality	LWLightInfo	lwrender.h
r	LWPixelRGB24	lwimageio.h
r	LWPixelRGBA32	
r	LWPixelRGBAFP	
r	LWPixelRGBFP	
range	LWLightInfo	lwrender.h
range	LWPanControlDesc	lwpanel.h
rawColor	LWLightInfo	lwrender.h
ray	LWMicropol	lwtxtr.h
ray	LWVolumeAccess	lwvolume.h
rayCast	LWMicropol	lwtxtr.h
rayCast	LWPixelAccess	lwfilter.h
rayCast	LWShaderAccess	lwshader.h
rayCast	LWVolumeAccess	lwvolume.h
rayColor	LWVolumeAccess	
rayLength	LWMicropol	lwtxtr.h
rayLength	LWShaderAccess	lwshader.h
rayShade	LWMicropol	lwtxtr.h
rayShade	LWPixelAccess	lwfilter.h
rayShade	LWShaderAccess	lwshader.h
rayShade	LWVolumeAccess	lwvolume.h
raySource	LWMicropol	lwtxtr.h
raySource	LWShaderAccess	lwshader.h
rayTrace	LWMicropol	lwtxtr.h
rayTrace	LWPixelAccess	lwfilter.h
rayTrace	LWShaderAccess	lwshader.h
rayTrace	LWVolumeAccess	lwvolume.h
read	LWLoadState	lwio.h
readData	LWLoadState	
readFailure	LWSceneConverter	lwscenecv.h
readFP	LWLoadState	lwio.h
readI1	LWLoadState	
readI2	LWLoadState	
readI4	LWLoadState	
readID	LWLoadState	
readStr	LWLoadState	
readU1	LWLoadState	
readU2	LWLoadState	
readU4	LWLoadState	

recursionDepth	LWSceneInfo	lwrender.h
red	LWColorPickLocal	lwdialog.h
reflectionBlur	LWShaderAccess	lwshader.h
refName	LWObjectFuncs	lwmeshes.h
refractionBlur	LWShaderAccess	lwshader.h
refresh	LWTxtrEdFuncs	lwtxtred.h
regionLimits	LWCameraInfo	lwrender.h
remParticle	LWPSysFuncs	lwprtcl.h
remPoint	MeshEditOp	lwmeshedt.h
remPoly	MeshEditOp	
rend	LWCustomObjHandler	lwcustobj.h
rend	LWDisplacementHandler	lwdisplce.h
rend	LWEnvironmentHandler	lwenviron.h
rend	LWPixelFilterHandler	lwfilter.h
rend	LWRenderHandler	lwrender.h
rend	LWShaderHandler	lwshader.h
rend	LWTextureHandler	lwtexture.h
rend	LWVolumetricHandler	lwvolume.h
renderCamera	LWSceneInfo	lwrender.h
renderOpts	LWSceneInfo	
renderType	LWSceneInfo	
replacement_color	LWShaderAccess	lwshader.h
replacement_percentage	LWShaderAccess	
reqType	LWFileReqLocal	lwdialog.h
resample	LWImageUtil	lwimage.h
resolution	LWCameraInfo	lwrender.h
restLength	LWBoneInfo	
restParam	LWBoneInfo	
result	LWColorPickLocal	lwdialog.h
result	LWFileReqLocal	
result	LWImageLoaderLocal	lwimageio.h
result	LWImageSaverLocal	
result	LWObjectImport	lwobjimp.h
RGB	LWImageList	lwimage.h
rgbaz	PvSample	lwpreview.h
RGBSpot	LWImageList	lwimage.h
roughness	LWShaderAccess	lwshader.h
save	LWEnvelopeFuncs	lwenvel.h
save	LWImageUtil	lwimage.h
save	LWInstanceFuncs	lwhandler.h
save	LWPSysFuncs	lwprtcl.h
save	LWShelfFuncs	lwshelf.h
save	LWTextureFuncs	lwtxtr.h
save	LWVParmFuncs	lwvparm.h
saverCount	LWImageUtil	lwimage.h
saverName	LWImageUtil	
saveScene	LWLayoutGeneric	lwgeneric.h
scanPoints	LWMeshInfo	lwmeshes.h
scanPolys	LWMeshInfo	
sceneLoad	LWImageList	lwimage.h
sceneObject	LWSurfaceFuncs	lwsurf.h
sceneSave	LWImageList	lwimage.h
schemaPos	LWInterfaceInfo	lwrender.h
selectAdd	LWTxtrEdFuncs	lwtxtred.h
selectClr	LWTxtrEdFuncs	
selectFirst	LWTxtrEdFuncs	
selectNext	LWTxtrEdFuncs	
selectRem	LWTxtrEdFuncs	
selItems	LWInterfaceInfo	lwrender.h
sendData	LWImageSaverLocal	lwimageio.h
sendLine	LWImageProtocol	
separation	LWCameraInfo	lwrender.h
server	LWChannelInfo	lwenvel.h

server	LWItemInfo	lwrender.h
serverApply	LWChannelInfo	lwenvel.h
serverDefs	ModuleDescriptor	lwmodule.h
serverFlags	LWChannelInfo	lwenvel.h
serverFlags	LWItemInfo	lwrender.h
serverInstance	LWChannelInfo	lwenvel.h
serverRemove	LWChannelInfo	
set	LWControl	lwpanel.h
set	LWPanelFuncs	
setAlpha	LWFilterAccess	lwfilter.h
setBufData	LWPSysFuncs	lwprtcl.h
setChannel	LWChannelAccess	lwchannel.h
setChannelEvent	LWChannelInfo	lwenvel.h
setClick	LWPreviewFuncs	lwpreview.h
setColor	LWCustomObjAccess	lwcustobj.h
setColorVMap	LWSurfaceFuncs	lwsurf.h
setContext	LWPreviewFuncs	lwpreview.h
setContext	LWShelfFuncs	lwshelf.h
setData	LWXPanelFuncs	lwxpanel.h
setEnvEvent	LWEnvelopeFuncs	lwenvel.h
setEnvGroup	LWTextureFuncs	lwtxtr.h
setGradientAutoSize	LWTxtrEdFuncs	lwtxtred.h
setMap	LWImageProtocol	lwimageio.h
setOptions	LWPreviewFuncs	lwpreview.h
setParam	LWImageProtocol	lwimageio.h
setParam	LWItemMotionAccess	lwmotion.h
setParam	LWTextureFuncs	lwtxtr.h
setParticle	LWPSysFuncs	lwprtcl.h
setPattern	LWCustomObjAccess	lwcustobj.h
setPixel	LWImageUtil	lwimage.h
setPosition	LWSurfEdFuncs	lwsurfed.h
setPosition	LWTxtrEdFuncs	lwtxtred.h
setPreset	LWPreviewFuncs	lwpreview.h
setRender	LWPreviewFuncs	
setRGB	LWFilterAccess	lwfilter.h
setRGBA	LWPixelAccess	
setSize	LWImageProtocol	lwimageio.h
setState	LWVParmFuncs	lwvparm.h
setSurface	LWSurfEdFuncs	lwsurfed.h
setTag	LWItemInfo	lwrender.h
setTexture	LWCustomObjAccess	lwcustobj.h
setTexture	LWTxtrEdFuncs	lwtxtred.h
setup	LWLMonFuncs	lwmonitor.h
setup	LWVParmFuncs	lwvparm.h
setUVs	LWCustomObjAccess	lwcustobj.h
setVal	LWPixelAccess	lwfilter.h
setVal	LWVParmFuncs	lwvparm.h
setwinpos	LWLMonFuncs	lwmonitor.h
shadMapAngle	LWLightInfo	lwrender.h
shadMapFuzz	LWLightInfo	
shadMapSize	LWLightInfo	
shadowOpts	LWObjectInfo	
shadowType	LWLightInfo	
shutdown	ModuleDescriptor	lwmodule.h
size	LWGlobalPool	lwrender.h
size	LWImageList	lwimage.h
size	LWTextureAccess	lwtexture.h
source	LWDisplacementAccess	lwdisplce.h
source	LWVolumeAccess	lwvolume.h
sourceID	LWShaderAccess	lwshader.h
specular	LWShaderAccess	
spline	LWWireDrawAccess	lwtool.h
spotSize	LWMicropol	lwtxtr.h
spotSize	LWShaderAccess	lwshader.h

spotSize	LWTextureAccess	lwtexture.h
squeeze	LWBackdropInfo	lwrender.h
srfID	LWMicropol	lwtxtr.h
start	EDBoundCv	lwmeshedt.h
start	LWFilterAccess	lwfilter.h
start	LWLayoutToolFuncs	lwlaytool.h
start	LWToolFuncs	lwtool.h
start	LWTxtrParamDesc	lwtxtr.h
startRender	LWPreviewFuncs	lwpreview.h
startup	ModuleDescriptor	lwmodule.h
state	MeshEditOp	lwmeshedt.h
step	LWLMonFuncs	lwmonitor.h
step	LWMonitor	
stiffness	LWItemInfo	lwrender.h
stopRender	LWPreviewFuncs	lwpreview.h
str	DynaValue	lwdyna.h
str	LWValue	lwpanel.h
strength	LWBoneInfo	lwrender.h
stride	LWVolumeSample	lwvolume.h
string	ServerTagInfo	lwserver.h
string	DyReqControlDesc	lwdyna.h
string	LWPanControlDesc	lwpanel.h
subdivOrder	LWObjectInfo	lwrender.h
subscribe	LWPreviewFuncs	lwpreview.h
subscribe	LWShelfFuncs	lwshelf.h
subscribe	LWTxtrEdFuncs	lwtxtred.h
surface	EDPolygonInfo	lwmeshedt.h
surface	LWObjectImport	lwobjimp.h
surface	LWStateQueryFuncs	lwmodeler.h
sx	LWPixelAccess	lwfilter.h
sx	LWShaderAccess	lwshader.h
sy	LWPixelAccess	lwfilter.h
sy	LWShaderAccess	lwshader.h
sysMachine	ModuleDescriptor	lwmodule.h
sysSync	ModuleDescriptor	
sysVersion	ModuleDescriptor	
tag	ServerTagInfo	lwserver.h
tagInfo	ServerRecord	
target	LWItemInfo	lwrender.h
test	LWMeshEditTool	lwmodtool.h
text	DyReqTextDesc	lwdyna.h
text	LWCustomObjAccess	lwcustobj.h
text	LWPanTextDesc	lwpanel.h
text	LWWireDrawAccess	lwtool.h
text	DyReqControlDesc	lwdyna.h
text	LWPanControlDesc	lwpanel.h
textAscent	display_Metrics	
textHeight	display_Metrics	
textHeight	LWXPDrawFuncs	lwxpanel.h
texture	LWTextureFuncs	lwtxtr.h
textWidth	DrawFuncs	lwpanel.h
textWidth	LWXPDrawFuncs	lwxpanel.h
time	LWChannelAccess	lwchannel.h
time	LWItemMotionAccess	lwmotion.h
time	LWTimeInfo	lwrender.h
title	LWColorPickLocal	lwdialog.h
title	LWFileReqLocal	
tmpScene	LWSceneConverter	lwscenecv.h
token	LWBlockIdent	lwio.h
tool	LWLayoutTool	lwlaytool.h
tool	LWMeshEditTool	lwmodtool.h
tool	LWTool	lwtool.h
top	LWPanListBoxDesc	lwpanel.h

top	LWPanMultiListBoxDesc	
tPos	LWTextureAccess	lwtexture.h
translucency	LWShaderAccess	lwshader.h
transparency	LWShaderAccess	
tree	LWPanControlDesc	lwpanel.h
triangle	LWCustomObjAccess	lwcustobj.h
txGrad	LWTextureAccess	lwtexture.h
txRGBA	LWTextureAccess	
txVal	LWMicropol	lwtxtr.h
type	DyReqChoiceDesc	lwdyna.h
type	DyReqStringDesc	
type	DyReqTextDesc	
type	DyValCustom	
type	DyValFloat	
type	DyValFVector	
type	DyValInt	
type	DyValIVector	
type	DyValString	
type	EDPolygonInfo	lwmeshedt.h
type	LWAnimSaverHandler	lwanimsav.h
type	LWBackdropInfo	lwrender.h
type	LWFogInfo	
type	LWFrameBufferHandler	lwframbuf.h
type	LWImageProtocol	lwimageio.h
type	LWImageSaverLocal	
type	LWItemInfo	lwrender.h
type	LWLightInfo	
type	LWMasterHandler	lwmaster.h
type	LWPanAreaDesc	lwpanel.h
type	LWPanChoiceDesc	
type	LWPanListBoxDesc	
type	LWPanLWItemDesc	
type	LWPanMultiListBoxDesc	
type	LWPanPopupDesc	
type	LWPanRangeDesc	
type	LWPanStringDesc	
type	LWPanTextDesc	
type	LWPanTreeDesc	
type	LWPanXPanDesc	
type	LWTextureFuncs	lwtxtr.h
type	LWTxtrParamDesc	
type	LWValCustom	lwpanel.h
type	LWValFloat	
type	LWValFVector	
type	LWValInt	
type	LWValIVector	
type	LWValPointer	
type	LWValString	
type	DynaValue	lwdyna.h
type	DyReqControlDesc	
type	LWPanControlDesc	lwpanel.h
type	LWValue	
unlock	LWMTUtilFuncs	lwmtutil.h
unsubscribe	LWPreviewFuncs	lwpreview.h
unsubscribe	LWShelfFuncs	lwshelf.h
unsubscribe	LWTxtrEdFuncs	lwtxtred.h
up	LWLayoutToolFuncs	lwlaytool.h
up	LWToolFuncs	lwtool.h
useItems	LWItemFuncs	lwrender.h
user_data	LWPanelFuncs	lwpanel.h
userData	EDPointInfo	lwmeshedt.h
userData	EDPolygonInfo	
userData	LWMicropol	lwtxtr.h

userData	LWTextureFuncs	
userName	LWObjectFuncs	lwmeshes.h
val	DyValCustom	lwdyna.h
val	DyValFVector	
val	DyValIVector	
val	LWValCustom	lwpanel.h
val	LWValFVector	
val	LWValIVector	
value	DyChoiceHint	lwdyna.h
value	DyValFloat	
value	DyValInt	
value	LWChannelAccess	lwchannel.h
value	LWValFloat	lwpanel.h
value	LWValInt	
valueGet	DynaReqFuncs	lwdyna.h
valueSet	DynaReqFuncs	
version	LWXPanelFuncs	lwxpanel.h
vertical	DyReqChoiceDesc	lwdyna.h
vertical	LWPanChoiceDesc	lwpanel.h
verts	LWMicropol	lwtxtr.h
verts	LWShaderAccess	lwshader.h
vertsWPos	LWMicropol	lwtxtr.h
vertsWPos	LWShaderAccess	lwshader.h
vid	LWXPanelDataDesc	lwxpanel.h
view	LWCustomObjAccess	lwcustobj.h
viewDir	LWCustomObjAccess	
viewInst	LWXPanelFuncs	lwxpanel.h
viewPos	LWCustomObjAccess	lwcustobj.h
viewRefresh	LWXPanelFuncs	lwxpanel.h
visItems	LWPanListBoxDesc	lwpanel.h
visItems	LWPanMultiListBoxDesc	
vmap	LWObjectImport	lwobjimp.h
vmap	LWStateQueryFuncs	lwmodeler.h
vmapDim	LWObjectFuncs	lwmeshes.h
vmapName	LWObjectFuncs	
vmapPDV	LWObjectImport	lwobjimp.h
vmapType	LWObjectFuncs	lwmeshes.h
vmapVal	LWObjectImport	lwobjimp.h
vmapVec	EDPointInfo	lwmeshedt.h
warning	LWMessageFuncs	lwhost.h
wAxis	LWMicropol	lwtxtr.h
weightMap	LWBoneInfo	lwrender.h
weights	LWMicropol	lwtxtr.h
weights	LWShaderAccess	lwshader.h
width	display_Metrics	lwpanel.h
width	DyReqStringDesc	lwdyna.h
width	LWFilterAccess	lwfilter.h
width	LWPanAreaDesc	lwpanel.h
width	LWPanListBoxDesc	
width	LWPanLWItemDesc	
width	LWPanMultiListBoxDesc	
width	LWPanPopupDesc	
width	LWPanRangeDesc	
width	LWPanStringDesc	
width	LWPanTreeDesc	
width	LWPanXPanDesc	
window	HostDisplayInfo	lwdisplay.h
wNorm	LWMicropol	lwtxtr.h
wNorm	LWShaderAccess	lwshader.h
wNorm0	LWShaderAccess	
wPos	LWMicropol	lwtxtr.h
wPos	LWShaderAccess	lwshader.h

wPos	LWTextureAccess	lwtexture.h
write	LWAnimSaverHandler	lwanimsav.h
write	LWFrameBufferHandler	lwframbuf.h
write	LWSaveState	lwio.h
writeData	LWSaveState	
writeFP	LWSaveState	
writeI1	LWSaveState	
writeI2	LWSaveState	
writeI4	LWSaveState	
writeID	LWSaveState	
writeStr	LWSaveState	
writeU1	LWSaveState	
writeU2	LWSaveState	
writeU4	LWSaveState	
wXfrm	LWMicropol	lwtxtr.h
wXfrm	LWShaderAccess	lwshader.h
x	PvSample	lwpreview.h
xpan	LWPanXPanDesc	lwpanel.h
xpanel	DynaReqFuncs	lwdyna.h
xpanel	LWPanControlDesc	lwpanel.h
xsys	HostDisplayInfo	lwdisplay.h
y	PvSample	lwpreview.h
yesNo	LWMessageFuncs	lwhost.h
yesNoAll	LWMessageFuncs	
yesNoCan	LWMessageFuncs	
zoomFactor	LWCameraInfo	lwrender.h

Structures and Typedefs

Symbol	Header
ActivateFunc	lwserver.h
cleanupFunc	lwpreview.h
clickFunc	
closeFunc	
ControlDesc	lwpanel.h
cTag	
display_Metrics	
DrawFuncs	
DrMode	
DyBitfieldHint	lwdyna.h
DyChoiceHint	
DynaConvertFunc	
DynaMonitorFuncs	
DynaReqFuncs	
DynaRequestID	
DynaStringHint	
DynaType	
DynaValue	
DyReqChoiceDesc	
DyReqControlDesc	
DyReqStringDesc	
DyReqTextDesc	
DyValCustom	
DyValFloat	
DyValFVector	
DyValInt	
DyValIVector	
DyValString	
EDBoundCv	lwmeshedt.h
EDError	
EDPointInfo	
EDPointScanFunc	
EDPolygonInfo	
EDPolyScanFunc	
EDStateRef	
EltOpLayer	lwmodeler.h
EltOpSelect	
evaluateFunc	lwpreview.h
GlobalFunc	lwserver.h
gParamData	lwtxtr.h
HostDisplayInfo	lwdisplay.h
ImageValue	lwimageio.h
initFunc	lwpreview.h
InputMode	lwpanel.h
LW_TxtrAutoSizeFunc	lwtxtred.h
LW_TxtrEventFunc	
LW_TxtrRemoveFunc	
LWAnimFrameAccess	lwanimlod.h
LWAnimLoaderHandler	
LWAnimSaverHandler	lwanimsav.h
LWBackdropInfo	lwrender.h
LWBlockIdent	lwio.h

LWBoneInfo	lwrender.h
LWBRCltID	lwvbshelf.h
LWBRFileThumFunc	
LWBRFuncs	
LWBRLoadSettings	
LWBRPreMultiFunc	
LWBRPreThumFunc	
LWBRShelfClosed	
LWBRStrList	
LWBufferValue	lwtypes.h
LWCameraInfo	lwrender.h
LWChanEventFunc	lwenvel.h
LWChanGroupID	
LWChannelAccess	lwchannel.h
LWChannelHandler	
LWChannelID	
LWChannelID	lwenvel.h
LWChannelInfo	
LWColorActivateFunc	lwhost.h
LWColorPickLocal	lwdialog.h
LWCommandCode	lwtypes.h
LWCompInfo	lwrender.h
LWControl	lwpanel.h
LWControlID	
LWCtlDrawHook	
LWCtlEventHook	
LWCustomObjAccess	lwcustobj.h
LWCustomObjHandler	
LWDirInfoFunc	lwhost.h
LWDisplacementAccess	lwdisplce.h
LWDisplacementHandler	
LWDualKey	lwpanel.h
LWDVector	lwtypes.h
LWEnvelopeFuncs	lwenvel.h
LWEnvelopeID	
LWEnvEvent	
LWEnvEventFunc	
LWEnvironmentAccess	lwenviron.h
LWEnvironmentHandler	
LWEnvironmentMode	
LWEnvKeyframeID	lwenvel.h
LWEnvTag	
LWError	lwtypes.h
LWFileActivateFunc	lwhost.h
LWFileIOFuncs	lwio.h
LWFileReqFunc	lwhost.h
LWFileReqLocal	lwdialog.h
LWFileTypeFunc	lwhost.h
LWFilterAccess	lwfilter.h
LWFilterContext	
LWFogInfo	lwrender.h
LWFontListFuncs	lwmodeler.h
LWFrame	lwtypes.h
LWFrameBufferHandler	lwframbuf.h
LWFVector	lwtypes.h
LWGlobalPool	lwrender.h
LWGlobalService	lwglobsrv.h
LWHandler	lwhandler.h
LWHotColorFunc	lwdialog.h
LWID	lwtypes.h
LWIlluminateFunc	lwrender.h
LWImageFilterHandler	lwfilter.h
LWImageID	lwtypes.h
LWImageList	lwimage.h

LWImageLoaderLocal	lwimageio.h
LWImageParam	
LWImageProtocol	
LWImageProtocolID	
LWImageSaverLocal	
LWImageType	
LWImageUtil	lwimage.h
LWInstance	lwtypes.h
LWInstanceFuncs	lwhandler.h
LWInstUpdate	
LWInterface	
LWInterfaceInfo	lwrender.h
LWItemFuncs	
LWItemHandler	
LWItemID	
LWItemInfo	
LWItemMotionAccess	lwmotion.h
LWItemMotionHandler	
LWItemParam	lwrender.h
LWItemType	
LWKeyTag	lwenvel.h
LWLayoutGeneric	lwgeneric.h
LWLightInfo	lwrender.h
LWLoadState	lwio.h
LWMasterAccess	lwmaster.h
LWMasterHandler	
LWMemChunk	lwrender.h
LWMeshEditTool	lwmodtool.h
LWMeshInfo	lwmeshes.h
LWMeshInfoID	
LWMessageFuncs	lwhost.h
LWMicropol	lwtxtr.h
LWMicropolID	
LWModCommand	lwcmdseq.h
LWMonitor	lwmonitor.h
LWMTUtilFuncs	lwmtutil.h
LWMTUtilID	
LWObjectFuncs	lwmeshes.h
LWObjectImport	lwobjimp.h
LWObjectInfo	lwrender.h
LWObjReplacementAccess	lwobjrep.h
LWObjReplacementHandler	
LWPanAreaDesc	lwpanel.h
LWPanChoiceDesc	
LWPanControlDesc	
LWPanDrawHook	
LWPanelFuncs	
LWPanelID	
LWPanHook	
LWPanKeyHook	
LWPanListBoxDesc	
LWPanLWItemDesc	
LWPanMouseHook	
LWPanMultiListBoxDesc	
LWPanPopupDesc	
LWPanRangeDesc	
LWPanStringDesc	
LWPanTextDesc	
LWPanTreeDesc	
LWPanXPanDesc	
LWPixelAccess	lwfilter.h
LWPixelFilterHandler	
LWPixelID	lwimageio.h
LWPixelRGB24	

LWPixelRGBA32	
LWPixelRGBAFP	
LWPixelRGBFP	
LWPntID	lwmeshes.h
LWPntScanFunc	
LWPolID	
LWPolScanFunc	
LWPreviewFuncs	lwpreview.h
LWPSBufDesc	lwprtcl.h
LWPSBufID	
LWPSTFuncs	lwvbshelf.h
LWPSTID	
LWPSysFuncs	lwprtcl.h
LWPSysID	
LWRasterFuncs	lwpanel.h
LWRasterID	
LWRayCastFunc	lwrender.h
LWRayShadeFunc	
LWRayTraceFunc	
LWRenderFuncs	
LWRenderHandler	
LWSaveState	lwio.h
LWSceneConverter	lwscenecv.h
LWSceneInfo	lwrender.h
LWShaderAccess	lwshader.h
LWShaderHandler	
LWShelfCltID	lwshelf.h
LWShelfParmList	
LWShelfLoadOkFunc	
LWShelfLoadFunc	
LWShelfSaveFunc	
LWShelfFuncs	
LWStateQueryFuncs	lwmodeler.h
LWSurfaceFuncs	lwsurf.h
LWSurfaceID	
LWSurfEdFuncs	lwsurfed.h
LWTECltID	lwtxtred.h
LWTextureAccess	lwtexture.h
LWTextureFuncs	lwtxtr.h
LWTextureHandler	lwtexture.h
LWTextureID	lwtxtr.h
LWTime	lwtypes.h
LWTLayerID	lwtxtr.h
LWTool	lwtool.h
LWToolEvent	
LWToolFuncs	
LWTxtrContextID	lwtxtr.h
LWTxtrEdFuncs	lwtxtred.h
LWTxtrParamDesc	lwtxtr.h
LWTxtrParamFuncs	
LWType	lwpanel.h
LWValCustom	
LWValFloat	
LWValFVector	
LWValInt	
LWValIVector	
LWValString	
LWValue	
LWVolumeAccess	lwvolume.h
LWVolumeSample	
LWVolumetricHandler	
LWVParmFuncs	lwenvel.h
LWVParmID	
LWWireDrawAccess	lwtool.h

LWXPanelBtnClickFunc	lwxpanel.h
LWXPanelChangeNotifyFunc	
LWXPanelControl	
LWXPanelControlDrawFunc	
LWXPanelControlZoomFunc	
LWXPanelDataDesc	
LWXPanelDestroyNotifyFunc	
LWXPanelFuncs	
LWXPanelGetFunc	
LWXPanelHint	
LWXPanelID	
LWXPanelPopCmdFunc	
LWXPanelPopCntFunc	
LWXPanelPopNameFunc	
LWXPanelSetFunc	
LWXPDrAreaID	
LWXPDrawFuncs	
MeshEditBegin	lwmeshedt.h
MeshEditOp	
ModuleDescriptor	lwmodule.h
MxType	lwpanel.h
optionsFunc	lwpreview.h
pTag	lwpanel.h
PvContextID	lwpreview.h
PvSample	
ServerRecord	lwserver.h
ServerUserName	

6.0B Changes

May 11, 2000

This is a list of the changes in the LightWave 6.0B patch that affect the SDK. In most cases, the changes are additions to global services that won't affect the operation of existing code. In some cases, however, you will have to recompile, and in a few, it may be necessary to rewrite a small amount of your code that was written prior to the release of the patch.

`lwenvel.h`

- Added `server` function to the Channel Info global.

`lwfilter.h`

- Added `LWBUF_MOTION_X` and `LWBUF_MOTION_Y` for vector blur feature. The buffers will be filled in with the image coordinate movement at each pixel during the time the shutter was open. For example, if part of an object has moved 11 pixels right and 6 pixels down since the previous frame, and the blur length is 50%, the motion buffers will contain 5.5 and 3.0 in that part of the frame. Camera motion is also taken into account. This information can be used by filters to perform a Photoshop-style vector motion blur.

`lwgeneric.h`

- Generic plug-ins are now first activated with a version number of 4. If that doesn't work, the `saveScene` function is replaced with one that saves old format scenes, and version numbers 3, 2, and 1 are tried.

`lwhost.h`

- Added new dialog types to Message Functions global, incremented the service name. (3/23)
- The new Locale Info global has been implemented (but the old Language ID global is still supported as well).

`lwimage.h`

- Added `sceneLoad` and `sceneSave` to Image List global, now "Image List 2".
- Changed LWImageID to LWPixmapID in Image Utility functions.

`lwmaster.h`

- Added `LWMAST_LAYOUT` type for masters that survive scene clearing.

`lwmodeler.h`

- `mode` and `vmap` functions added to the state query global, incremented the service name.

`lwpanel.h`

- fixed `PAN_SETDATA`, `PAN_SETDRAW`, `PAN_SETKEYS`; touched up `PAN_SETH`, `MOVE_PAN`; added `PAN_SETW`.

`lwpreview.h`

- Changed `close` to accept void arg.

`lwprtcl.h`

- New API.
- Removed obsolete functions `setDraw`, `setMesh`, `remParticle`.

`lwrender.h`

- Only items matching the current edit mode are now eligible to have their `LWITEMF_SELECTED` bit set by the Interface Info `itemFlags` function.
- Added `LWIP_PIVOT_ROT` to let plug-ins get pivot rotation.
- If the Item Info `param` function is called while the user is dragging items in a viewport and the time argument matches the current time, the function can now return temporary, non-keyframed data. One benefit of this change is that expressions can react to interactively moved items even if Auto Key is turned off.
- changed flag name to `LWOBJF_UNSEEN_BY_CAMERA`.
- A `server` function has been added to the Item Info global. It takes an item ID, a class name, and an index (which starts with one, as in the tag functions). The return value is the name of whatever server is applied in the specified "slot" (or NULL if there isn't one). Naturally the ID is ignored for classes like volumetrics, filters, etc.

- Added `flags` and `fog` to Object Info.
- Corrected "LW Comp Info" to "LW Compositing Info".
- Added `renderCamera` to LWSceneInfo structure.
- Added `type`, `color`, and `squeeze` to LWBackdropInfo structure.
- A function to get the ID of the camera used for rendering has been added to the Scene Info global. It accepts a time argument in order to be ready for automated camera switching in the future.
- Added `serverFlags` and `controller` to Item Info.
- The first frame, last frame, and step size used by the frame slider and by Layout previews can now be obtained through the Interface Info global.
- The vectors returned by the Item Info `param` function now take pivot rotation into account.
- Attempts by plug-ins to look up information about their items while in the process of being created due to cloning should no longer fail.

`lwshader.h`

- Added `polygonID` to LWShaderAccess.
- Added `vertsWPos` to LWShaderAccess.

`lwshelf.h`

- New API.
- Removed closed notification function typedef.
- Removed closed notification from shelf subscription.
- Added `isOpen`.

`lwsurf.h`

- The `byName` function now has an additional object name argument. If the object name is set to NULL, the function will return all the surfaces that share the same name, behaving as before.

`lwtxtr.h`

- `layerEnvGrp` renamed `layerEnvGroup`.
- Input parameter type defines added.
- `polygonID` added to LWMicropol structure.
- `setParam` and `getParam` functions added.
- Added `evaluateUV`, which sets the UV values given the 3D position of

the point in world and local coordinates, plus the world and local dominant axis. This does the coordinate transfromation and projection from 3D to 2D.
- Added new tags to set and get images and vmaps.
- Added `vertsWPos` to LWMicropol.
- Added `setEnvGroup` and `envGroup` functions.
- Added `layerType` function.
- Added tags for getting and setting opacity, reference object, repeat options, pixel blending, AA, AA strength.

`lwtxtred.h`

- `setParam` and `getParam` functions removed (put into lwtxtr.h).
- Added `refresh`, which refreshes the editor if some parameters have been changed without user interaction. The clientID should be NULL if trying to update surface textures (= default client). That's what texture guide is doing.
- Added `currentLayer`, which gets the currently selected layer in the texture editor. If client is NULL, uses surface's texture editor.

`lwvolume.h`

- Removed adaptive flag.
- The `flags` function of each volumetric plug-in is now called to determine whether to include it in reflection, refraction, or shadow computations.
- The `LWVEF_RAYTRACE` bit in the LWVolumeAccess structure is now set when evaluating volumetric plug-ins for a ray.

`lwxpanel.h`

- Added focus event code.
- Added refresh codes.
- Added `XpXREQCFG`, `XpBORDER`, `XpDLGTYPE` hints.

Commands: Layout

- The usage string for the `AddPlugins` command now shows that it accepts a filename argument.
- The `AddToSelection` and `RemoveFromSelection` commands have been added to allow plug-ins and scripts to create and manage multiple

selections. They accept a hexadecimal item ID as an argument, and the usual selection rules apply (for example, if only one item of the current type is selected, it can't be deselected).

- Other new commands include `SubdivisionOrder` and `ResolutionMultiplier`, which work just like the identically named scene file lines.
- A `RemoveServer` command has been added. Like `ApplyServer`, it takes a class name argument, but it differs in that it takes an index rather than a server name (in case there are multiple instances of the same server). Consistent with the Item Info `server` function, these indices count from one. Item-specific plug-ins like motion and displacement handlers are removed from the current item.

Commands: Modeler

- The `SaveCommandList` command in Layout has been duplicated in Modeler.
- A new batch command `maketesball2` has been added which takes the new segments parameter for tesselated spheres. The old `maketesball` command still works using the old level parameter. The segments parameter directly sets the number of segments along the the edges between the 12 polyhedral vertices. Power of two segment values are equivalent to the old level values: 1 = level 0, 2 = level 1, 4 = level 2, 8 = level 3, 16 = level 4, 32 = level 5, etc.
- A new `maketext2` batch command has been added which accepts alignment settings. It also takes fonts as zero-based indices, fixing a long lived bug.
- Added more arguments to the `bevel` batch command to bring it more in line with the options of the Bevel tool.
- There are new plug-in commands for selecting the current object and for setting the vmap for the current morph, texture or weight map.

Commands: Surfaces

- The `Surf_SetSurf` command now has an additional object name argument. This specifies in which object library the surface should be set (there can be multiple surfaces with the same name in different objects).
- `Surf_RemoveShader` command now works this way: Removes all shaders that use the specified name, and if name is NULL, removes all

shaders.

Commands: Envelopes

- Added `Env_ApplyServer` <classname> <servername>.
- Added `Env_RemoveServer` <clasname> <index>.

Sample Code

- modlib: Added object name argument to `mgGetSurfacesByName` in surface.c.
- rapts: Changed `layerEnvGrp` to `layerEnvGroup`.

Miscellaneous

- A new Plug-in Options panel has been added. It contains the controls for generic and master plug-ins that used to be on the General Options panel, as well as an Add Plug-ins button and some UI ranch space for the masters.
- There is now a dedicated options button which calls `ServerInterface` for the current animation saver. This is intended to be the official way to edit compression settings, etc. Saver plug-ins whose interfaces are in their `load` or `open` functions should be rewritten.
- The DirectoryType config file entry for images is now used as the default location for image loading, replacing the ImagesDirectory line. Several other old config file lines have also been discontinued in favor of new DirectoryType entries.
- Envelopes for displacement and clip map channels are now added to the envelope groups of the objects they belong to. This makes them appear in the right place in the Graph Editor hierarchy.

6.5 Changes

November 4, 2000

This is a list of the changes in the LightWave 6.5 patch that affect the SDK. In most cases, the changes won't require you to rewrite or recompile your existing code. The most significant exception is for variant parameters, which have been updated to support parameters modulated by textures.

`lwenvel.h`

- Definitions of variant parameters have been moved to a new header, lwvparm.h.

`lwfilter.h`

- New buffers (`LWBUF_REFL_RED`, `LWBUF_REFL_GREEN` and `LWBUF_REFL_BLUE`) have been added to the list of buffers available to image filters and pixel filters. These contain the RGB levels for reflections calculated by raytracing and environment mapping.

`lwimage.h`

- `LWIMAGELIST_GLOBAL` (Image List) has been incremented to "LW Image List 3".
- Added LWImageList functions: `hasAlpha`, `alpha`, `alphaSpot`, `evaluate`.

`lwio.h`

- Added File I/O type `LWIO_BINARY_IFF`. This type supports 4-byte chunk sizes for block reading and writing of IFF files.

`lwmeshedt.h`

- `LWMESHEDIT_VERSION` (MeshDataEdit) has been incremented to 4.
- Added MeshEditOp functions `pointVPGet`, `pointVEval`, `pntVPMap`. These are for per-polygon vertex mapping.
- The second argument to `initUV` is now float * rather than float[2] so that `initUV` can be used to initialize per-polygon UVs.

`lwmeshes.h`

- Added vertex map types for color vmaps: `LWVMAP_RGB` and `LWVMAP_RGBA`.
- Added LWMeshInfo function `pntVPGet` for per-polygon vertex mapping.

`lwobjimp.h`

- `LWOBJECTIMPORT_VERSION` (ObjectLoader) has been incremented to 3.
- Added LWObjectImport function `vmapPDV` for per-polygon vertex mapping.
- The second argument to `pivot` and the third argument to `vmapVal` are now const.

`lwrender.h`

- New `illuminate` special item types `LWITEM_RADIOSITY` and `LWITEM_CAUSTICS` allow callers of `illuminate` to get information about global illumination at a spot.
- `LWOBJECTINFO_GLOBAL` (Object Info) has been incremented to "LW Object Info 3".
- The texture IDs of the displacement and clip maps of each object can now be obtained using the new `dispMap` and `clipMap` Object Info functions.
- `LWINTERFACEINFO_GLOBAL` (Interface Info) has been incremented to "LW Interface Info 2"
- New Interface Info function `schemaPos` returns the positions of items in schematic viewports. (Use the new `SchematicPosition` command to set them.)
- New Interface Info field `dynaUpdate` contains Layout's Dynamic Update setting, which can be `LWDYNUP_OFF`, `LWDYNUP_DELAYED` or `LWDYNUP_INTERACTIVE`.
- Added a Time Info global that returns the frame and time for the image currently being rendered.

`lwserver.h`

- ServerUserName has been replaced by ServerTagInfo, which among other things allows plug-ins to specify how they should be added to LightWave's menu system.

`lwshelf.h`

- New LWShelfFuncs function `addNamedPreset`.

`lwsurf.h`

- `LWSURFACEFUNCS_GLOBAL` (Surface Functions) incremented to "Surface Functions 2".
- Added surface channel `SURF_VCOL` for color vertex maps.
- Added Surface Functions routines `getColorVMap` and `setColorVMap`.

`lwtexture.h`

- `LWTEXTURE_VERSION` (ProceduralTextureHandler) has been incremented to 5.
- Texture class plug-ins can now return RGBA levels, in support of which added LWTextureAccess field `txRGBA[4]`, LWTextureHandler flag `LWTEXF_SELF_COLOR` and LWTextureAccess flag `LWTXEF_COLOR`.

`lwtxtr.h`

- New LWTxtrParamDesc flag `LWGF_AUTOSIZE` for gradient textures that can automatically adjust their min and max.

`lwtxtred.h`

- `LWTXTREDFUNCS_GLOBAL` (Texture Editor) has been incremented to "Texture Editor 2".
- New callback type `LW_GradAutoSizeFunc` for gradient autosizing.
- New Texture Editor functions for supporting multiselection (`selectAdd`, `selectRem`, `selectClr`, `selectFirst`, `selectNext`) and automatic ranges for gradient textures (`setGradientAutoSize`).

`lwvparm.h`

- New header (variant parameters were formerly defined in lwenvel.h).
- Added `LWVPDT` texture data types.
- Added arguments (texture context, event callback, plug-in name and userdata) to the LWVParmFuncs `create` function.
- Added a micropolygon argument to `getVal` for texture evaluation.
- New functions `getState`, `setState`, `editTex`, `initMP`, `getEnv` and `getTex`. `getState` and `setState` replace the `hasEnv` and `setEnvState` functions, which have been removed. The new functions support textures in addition to envelopes.

lwxpanel.h

- Added `XpENABLE` hint (XPanels).

Commands: Layout

- `SaveSceneCopy` and `SaveObjectCopy` save a copy of the scene or the object in a file with a new name without affecting the current name in Layout.
- `PreviousSibling` and `NextSibling` allow easy navigation among a set of items that share the same parent.
- `AddEnvelope` and `RemoveEnvelope` allow plug-ins and scripts to "press the E button" for a Layout parameter. Each command takes a channel name as its argument (the same names that appear when channels are expanded in the Scene Editor).
- By passing `AutoConfirm` an argument greater than zero, plug-ins and scripts can use this command to automatically respond affirmatively to all confirmation requesters before commands are issued to clear items, etc. This feature should be used with care, and it should be turned off by passing it a zero before control is returned to the user.
- `EditPlugins` opens the new dialog for displaying, adding, and deleting servers.
- `SaveObject` and `SaveTransformed` now accept a filename argument.
- `CommandInput` now accepts a command argument.
- `MaskPosition` allows plug-ins and scripts to set exact pixel values for the left, top, width, and height of the camera mask. One potential use is to create scene files for portions of a frame which can be rendered separately and then perfectly recombined by summing the resulting images (assuming the mask color is black).
- `SchematicPosition` sets the X and Y of the current item's schematic node. This and the `schemaPos` function added to the Interface Info global allow plug-ins to arrange items in Schematic viewports.
- `Antialiasing` takes an argument from 0 to 4, as in scene files.
- Other new commands:

```
TopView             CacheRadiosity          RayTraceShadows
BottomView          CacheCaustics           RayTraceReflection
BackView            EnableVolumetricLights  RayTraceRefraction
FrontView           RadiosityTolerance      MorphTarget
RightView           EnhancedAA              IncludeLight
LeftView            FogType                 ExcludeLight
SchematicView       FogMinDistance          PolygonEdgeColor
CenterItem          FogMaxDistance          CacheShadowMap
FitAll              FogMinAmount            MaskColor
FitSelected         FogMaxAmount            LightIntensityTool
```

```
ShowSafeAreas        FogColor                 EnableVIPER
ShowFieldChart       AddPartigon
```

Commands: Modeler

- The `revert` command reloads an existing object file.

Miscellaneous

- When items are moved, rotated, or scaled, using either the mouse or the numeric fields, the appropriate position, rotation, or scale commands are now added to the command history, which in turn generates events for master plug-ins.
- The `newTime` callbacks of shaders, volumetrics, and pixel filters are now called only after all geometry has been finalized. This fixes a problem that occurs when plug-ins of one of those classes depend on the results of displacement plug-ins.
- There can now be any number of custom object plug-ins per object, and they are added using a server pane rather than a popup.
- RGB and alpha image savers are now recorded in scene files by name rather than by index, preventing problems caused by different machines having different saver lists. Image saver indices in existing scene files will continue to work as before.
- When the `newTime` functions of shaders, environment plug-ins, volumetric plug-ins and lights, pixel filters, custom object and displacement plug-ins are called, inverse kinematics have already been computed. Therefore, the Item Info `param` function now simply returns existing vectors in such cases (assuming the time argument matches the current time), allowing those plug-ins to properly account for IK.
- When a subpatch object is being deformed and its Subdivision Order is later than the deformation, the interior patch points are no longer deformed. In particular, the `evaluate` functions of displacement plug-ins will no longer be called for the patch points.

6.5B Changes

May 8, 2001

This is a list of the changes in the LightWave 6.5B patch that affect the SDK. As with previous changes, these in most cases won't require you to rewrite or recompile your existing code.

Structures associated with the CustomObjHandler class and the Particle Services, Object Info and Interface Info globals were changed, but the custom object version number and the global name strings weren't incremented. This was discovered too close to the release of 6.5B to be remedied. In all cases, the changes involve members added to the ends of structures, so they have no effect on existing plug-ins. New plug-ins that need to distinguish between the old and new structures can use the program build numbers returned by the Product Info global.

`lwcustobj.h`

- A `text` function was added to LWCustomObjAccess (Custom Objects).
- Added `LWVIEW_SCHEMA` to the codes for the `view` field of the LWCustomObjAccess structure.
- Added `LWCOF_SCHEMA_OK` to the flags that can be returned from the LWCustomObjHandler `flags` callback

`lwhost.h`

- Added preprocessor symbols for some of the file type strings used with the File Type and Directory Info globals.

`lwmaster.h`

- Added `LWEVNT_TIME` to the event codes that can be passed to the Master class `event` callback. This event notice is sent whenever the frame slider is moved, which includes playing the scene, but not playing back a preview.

`lwmath.h`

- Bracketed `MIN`, `MAX` and `ABS` macros within preprocessor conditionals so that they won't conflict with versions of the macros that might be defined elsewhere.

`lwmeshes.h`

- Incremented `LWOBJECTFUNCS_GLOBAL` (Scene Objects) to "Scene Objects 3"
- Added `userName` and `refName` functions to the LWObjectFuncs structure. These return, respectively, the name of the object as seen by the user, and an unambiguous internal name for the object that can be used to refer to it in commands.

`lwpanel.h`

- Added LWValPointer to the data types in the LWValue union used by Panels. Besides being more type-friendly, this change is in anticipation of 64-bit operating systems on which ints and pointers may not be the same size.
- LWValPointer used in the definitions of the `CON_PAN`, `CON_PANFUN` and `CON_SETEVENT` macros.

`lwprtcl.h`

- The `LWPSB_ENB` buffer (Particle Services) now encodes three states (`LWPST_DEAD`, `LWPST_ALIVE`, `LWPST_LIMBO`) instead of two (on/off).
- Added `LWPSB_CAGE` (collision age) buffer containing the time since the last collision.
- Added `remParticle` function to LWPSysFuncs.

`lwrender.h`

- Incremented `LWSCENEINFO_GLOBAL` (Scene Info) to "LW Scene Info 3".
- Added `numThreads` to LWSceneInfo.
- Added `LWROPT_PARTICLEBLUR` option for the LWSceneInfo `renderOpts` function.
- Added `patchLevel` and `metaballRes` functions to LWObjectInfo (Object Info).
- Added `LWLFL_NO_OPENGL` to the flags for LWLightInfo (Light Info).
- Incremented `LWCAMERAINFO_GLOBAL` (Camera Info) to "LW Camera Info 2".
- Added `flags`, `resolution`, `pixelAspect`, `separation`, `regionLimits`, `maskLimits` and `maskColor` functions to LWCameraInfo.

- Added `LWCAMF_STEREO`, `LWCAMF_LIMITED_REGION` and `LWCAMF_MASK` camera info flags.
- Added `itemVis` function and `displayFlags` and `generalFlags` fields to the LWInterfaceInfo (Interface Info) structure.

`lwserver.h`

- Added `LANGID_KOREAN` definition.

`lwtxtr.h`

- Added `itemName` to LWTxtrParamDesc (Texture Functions)

`lwxpanel.h`

- Added `XPTAG_NULL` define, which replaces `NULL` in the Xp macros for XPanels.

Commands: Layout

- The arguments to the new `AddPosition`, `AddRotation` and `AddScale` are relative rather than absolute. These are useful for multiple selections and should also improve the reusability of scripts generated by macro recorders.
- An `EditServer` command has been added which opens a plug-in's interface. The syntax is the same as the `RemoveServer` command, with class name and index arguments. If the index isn't specified, the last plug-in in the list is used, making it easy for a script to add a plug-in and immediately open its interface without having to know its index.
- The `AutoConfirm` command now recognizes an argument value of -1, which will automatically respond negatively to all Yes/No or OK/Cancel dialogs.
- The `AddEnvelope` and `RemoveEnvelope` commands now operate on all three parts of a color envelope as a unit (".R", ".G", and ".B" suffixes are ignored).
- Other new commands:

```
ShadowExclusion        GradientBackdrop      ParentInPlace
NoiseReduction         ZenithColor           FractionalFrames
RadiosityIntensity     SkyColor              PolygonEdgeFlags
CausticIntensity       GroundColor           BoneFalloffType
VolumetricRadiosity    NadirColor            ShadowMapSize
DynamicUpdate
```

Commands: Modeler

- The new `meshedit` command allows plug-ins to execute MeshDataEdit class plug-ins.

Commands: Common

- The `Surf_SetSurf` command was using the display name rather than the filename to identify objects. This has been corrected.

Miscellaneous

- The mesh info `pntBasePos` and `pntOtherPos` functions obtained from the Object Info global now provide better information for frozen meshes. The revised `pntBasePos` returns the same point positions that Layout uses for object coordinate texture mapping. These are completely undeformed positions in the case of regular polygons and subpatches, and positions at freezing time for metaballs and partigons. `pntOtherPos` now returns the actual world coordinates used by Layout. The new behavior of these functions should remove the need to use "helper" displacement plug-ins to gather vertex positions.
- An Alert Level popup has been added to the General Options panel. At the Intermediate and Advanced levels, messages displayed using the message global's `info`, `warn` and `error` functions may appear in the status line rather than in separate dialog windows.
- The `changeID` handler callback is now called when all lights or all cameras are cleared at once.
- The `illuminate` raytracing function was returning 0 if the position was partially shadowed, which can happen when the light is a linear or area light. The return value is now 1.0 in such cases, and the partial shadowing is accounted for in the returned color.
- The `source` member of the LWVolumeAccess structure is now set to the camera ID for directly viewed volumetrics, to the light ID for shadow rays, and to `LWITEM_NULL` for other types of rays.
- The Z buffer given to image filters no longer has unfilled holes when unenhanced antialiasing is used with adaptive sampling and without motion blur or depth of field.
- Previously, if an object replacement plug-in tried to replace an object that had been cloned with an object of the same filename, the change

was ignored, since two objects with the same name weren't allowed to have different geometry. This has been fixed by freeing and replacing all instances of the object in such cases, just as manual object replacement does.

- Objects loaded by object import plug-ins no longer cause a crash when the scene is later saved.
- Custom object plug-ins were unable to draw points unless one of the other drawing functions had been called first. This has been fixed.
- Custom objects using the `LWCSYS_ICON` mode are no longer affected by object scaling.
- When the Item Info `param` function was called during a custom object plug-in's `evaluate` function, and FSPE was turned on, the interactive (non-keyframed) position of the item being queried could be forgotten. This has been fixed (but it's generally more efficient for plug-ins to call `param` during their newTime functions).
- When an object is cloned, the clones are now named earlier in the copying process so that custom object and displacement plug-ins can look up the names from inside their `create` functions.
- The viewports are now updated whenever a custom object, displacement or item motion plug-in is added, edited, or removed.
- Keyframe shifting and scaling now work by looping through all members of the envelope group of each affected item, including envelopes added by plug-ins.
- Manual operation of the envelope (E) button for color settings is now properly recorded in the command history.
- Item selections made by clicking in the viewports or the Scene Editor or by using the current item popups are now recorded in the command history so that master plug-ins are notified when these selection changes occur.
- Master plug-ins whose `flags` functions return `LWMAST_LAYOUT` are no longer cleared with the scene.
- The MeshEditOp `pntVMap` function no longer crashes when passed polygons created during the same mesh edit.
- The MeshEditOp polygon creation functions no longer crash when passed null vertex pointers. They instead return an error.

7.0 Changes

September 24, 2001

This is a list of the changes in the LightWave 7.0 release that affect the SDK. The changes include a new Layout tool class and two new globals, as well as extensive enhancement of many of the info globals defined in `lwrender.h`. Layout supports 51 new commands of its own plus another 46 that it now propagates to the various editors.

One change and one bug fix were made to the SDK for the LightWave 7.0B patch. These have been marked here with "[**B**]".

Structures associated with the ShaderHandler class and the Preview Functions and Item Info globals were changed, but the shader version number and the global name strings weren't incremented. In all cases, the changes involve members added to the ends of structures, so they have no effect on existing plug-ins. New plug-ins can use the Product Info global to verify that they're running in at least LightWave 7.0 before they use the new structure members.

`lwcustobj.h`

- `LWCUSTOMOBJ_VERSION` (CustomObjHandler) has been incremented.
- A `quad` function for drawing solid quad polygons has been added to LWCustomObjAccess.
- Quads can be image-mapped using new `setTexture` and `setUVs` functions. `setTexture` accepts a square image in `GL_RGBA` format, and `setUVs` sets the UV coordinates at the corners of a quad.
- The `text` function accepts a new justification argument, 0 for left, 1 for center, 2 for right.
- The color argument to `setColor` now takes a fourth array element for the color's alpha level.
- Since custom objects can draw semi-transparent primitives, `viewPos` and `viewDir` vectors have been added so that plug-ins can depth-sort their primitives.

`lwdisplay.h`

- The header now defines `_WIN32_WINNT`.

`lwdisplce.h`

- `LWDISPLACEMENT_VERSION` (DisplacementHandler) has been incremented because of changes to LWMeshInfo.

`lwenvel.h`

- `LWCHANNELINFO_GLOBAL` (Channel Info) has been incremented to "Channel Info 2".
- `serverApply` and `serverRemove` functions have been added to LWChannelInfo. Also added were `serverFlags` and `serverInstance` functions that return the flags and instance data of a channel handler.
- A new event code, `LWCEVNT_VALUE`, has been added to signal changes to channels.

`lwlaytool.h`

- This is a new header file containing the definition of a new plug-in class, LayoutTool.

`lwmaster.h`

- An `LWEVNT_SELECT` event code has been added (MasterHandler).

`lwmeshes.h`

- A `polFlags` function has been added to LWMeshInfo.
- `pntVIDGet` and `pntVPIDGet` functions have been added to LWMeshInfo. These are like `pntVGet` and `pntVPGet`, but they don't require a preceeding call to `pntVSelect`, making them more reliable when called from threaded code.
- Because of LWMeshInfo changes, `LWOBJECTFUNCS_GLOBAL` (Scene Objects) has been incremented to "Scene Objects 4".

`lwmonitor.h`

- A new global (Layout Monitor) has been created to provide a standard monitor for Layout. The server name is `LWLMONFUNCS_GLOBAL`.

`lwpanel.h`

- A new global (Context Menu) has been created to provide context menus in Panels. The server name is `LWCONTEXTMENU_GLOBAL`.
- `LWPANELS_API_VERSION` (Panels) has been incremented. Panels using the new `PANF_NOBUTT` flag will not have Continue or Cancel buttons. `PANF_RESIZE` panels can be resized.

`lwpreview.h`

- A `setPreset` function has been added to LWPreviewFuncs (Preview Functions).

`lwrender.h`

- Due to an error, `LWSCENEINFO_GLOBAL` (Scene Info) was defined as "LW Scene Info 2" in the public version of `lwrender.h` released with LightWave 6.5B. It should have been "LW Scene Info 3". The correct definition has been restored in the LightWave 7.0 headers.
- Four functions (`flags`, `lookAhead`, `goalStrength` and `stiffness`) have been added to LWItemInfo (Item Info).
- `LWOBJECTINFO_GLOBAL` (Object Info) has been incremented to "LW Object Info 4".
- Eight new functions (`boneSource`, `morphTarget`, `morphAmount`, `edgeOpts`, `edgeColor`, `subdivOrder`, `polygonSize` and `excluded`) have been added to LWObjectInfo. The `excluded` function returns TRUE if the object has been excluded from the given light.
- The LWObjectInfo `flags` function returns new flags `LWOBJF_UNAFFECT_BY_FOG`, `LWOBJF_MORPH_MTSE` and `LWOBJF_MORPH_SURFACES`.
- `LWBONEINFO_GLOBAL` (Bone Info) has been incremented to "LW Bone Info 3".
- Four new functions (`strength`, `falloff`, `jointComp` and `muscleFlex`) have been added to LWBoneInfo.
- The LWBoneInfo `flags` function returns new flags `LWBONEF_JOINT_COMP`, `LWBONEF_JOINT_COMP_PAR`, `LWBONEF_MUSCLE_FLEX` and `LWBONEF_MUSCLE_FLEX_PAR`.
- `LWLIGHTINFO_GLOBAL` (Light Info) has been incremented to "LW Light Info 3".
- Eight new functions (`falloff`, `projImage`, `shadMapSize`, `shadMapAngle`, `shadMapFuzz`, `quality`, `rawColor`, `intensity`) have been added to LWLightInfo.
- The `coneAngles` function now includes a time argument (since the angles can be enveloped) and returns values in radians rather than

degrees.
- A time argument has been added to the LWLightInfo `range` function.
- The LWLightInfo `flags` function returns new flags `LWLFL_FIT_CONE` and `LWLFL_CACHE_SHAD_MAP`.
- `LWINTERFACEINFO_GLOBAL` (Interface Info) has been incremented to "LW Interface Info 3".
- The LWInterfaceInfo `itemColor` function returns the color used to draw the item in the interface. The new `boxThreshold` field contains the bounding box threshold, and `alertLevel` contains `LWALERT_BEGINNER`, `LWALERT_INTERMEDIATE` or `LWALERT_EXPERT`.

`lwshader.h`

- `replacement_percentage` and `replacement_color` fields have been added to LWShaderAccess (ShaderHandler). Use these to set the surface color when your shader calculates lighting and you therefore don't want LightWave to overwrite your color.
- [**B**] `reflectionBlur` and `refractionBlur`. fields have been added to the shader access structure.

`lwsurf.h`

- Two new channel definitions for reflection and refraction blurring have been added (Surface Functions).

`lwtool.h`

- A `text` function and `pxScale` (pixel scale) field have been added to LWWireDrawAccess.

`lwtxtr.h`

- `LWTEXTUREFUNCS_GLOBAL` (Texture Functions) has been incremented to "Texture Functions 2", and four new functions have been added to LWTextureFuncs. `texture` returns a texture ID, given the ID of a texture layer. `name`, `type` and `context` return information about a texture.

`lwvparm.h`

- Due to an error, the public version of `lwvparm.h` (Variant Parameters) released with LightWave 6.5 and 6.5B contained an incorrect definition of the LWVParmFuncs `getEnv` function. This definition showed `getEnv` taking a single argument and returning a single envelope ID. The correct definition shows `getEnv` returning void. It writes up to three envelope IDs in an array passed as the second argument. This definition has been restored in the current header.

Commands: Layout

- The file written by `SaveCommandList` now includes Graph Editor, Image Editor and Surface Editor commands, and Layout now dispatches those commands to their respective components.
- The `Refresh` command now defers its work until the system is idle, for improved interactivity. A `RefreshNow` command has been added for cases in which synchronous updates are desired. The new `Redraw` and `RedrawNow` commands are similar except that they don't cause motions and geometry to be recomputed.
- The `PluginOptions` command has been replaced by `MasterPlugins`, to reflect the fact that the name of the panel it invokes has changed in the interface.
- Other new commands:

```
RecentScenes           ItemActive             PolygonSize
ReplaceObjectLayer     ItemLock               UnaffectedByFog
SquashTool             ItemVisibility         BoneStrengthMultiply
ShowTargetLines        ItemColor              BoneMinRange
BoundingBoxThreshold   RadiosityType          BoneMaxRange
VIPER                  RecentContentDirs      BoneJointComp
Presets                AlertLevel             BoneJointCompParent
EnableDeformations     AddButton              BoneJointCompAmounts
MatchGoalOrientation   Generics               BoneMuscleFlex
KeepGoalWithinReach    ClearAudio             BoneMuscleFlexParent
LimitH                 LoadAudio              BoneMuscleFlexAmounts
LimitP                 PlayAudio              LightQuality
LimitB                 RayTraceTransparency   ShadowMapFuzziness
HStiffness             LoadObjectLayer        ShadowMapFitCone
PStiffness             MorphMTSE              ShadowMapAngle
BStiffness             MorphSurfaces
```

Commands: Modeler

- The new `setcontent` command sets the content directory.

Commands: Common

- New (or newly available) Graph Editor commands include:

```
GE_OpenWindow         GE_DeleteSelKeys        GE_MoveKeys
GE_SetWindowPos       GE_ReduceKeys           GE_CreateExpression
GE_SetWindowSize      GE_LockKeys             GE_AttachExpression
```

```
GE_ApplyServer        GE_CopySelKeys          GE_AttachExpressionID
GE_RemoveServer       GE_PasteKeys            GE_SetGroup
GE_GetLayoutSel       GE_LeaveFootprints      GE_SetEnv
GE_SelectAllCurves    GE_PickupFootprints     GE_SetEnvID
GE_FilterSelection    GE_BacktrackFootprints  GE_LoadExpressions
GE_BakeCurves         GE_CopyTimeslice        GE_SaveExpressions
GE_SnapKeysToFrames   GE_PasteTimeslice       GE_ClearBin
GE_SelectAllKeys      GE_MatchFootprintAtFrame
```

- The newly available Image Editor commands are:

```
IE_OpenWindow
IE_SetClipTable
IE_SetWindowPos
```

- The surface system supports a new `Surf_SetBakerImage` command.

Miscellaneous

- When the user changes the item order, plug-ins are now notified of the new IDs only after the item data have actually been rearranged. Globals that take item IDs as arguments should now behave correctly when called from within a handler's `changeID` callback.
- In LWSN, master plug-ins now receive a `LoadScene` command event at the end of scene loading. Since the issuing of commands is not supported in LWSN, the LWMasterAccess structure contains dummy functions in this situation.
- Attempts by plug-ins to trace rays during Wireframe or Quickshade renders were causing crashes. This has been fixed (the ray tracing functions now always return -1 in those modes).
- The Directory Info global now returns the correct result for "Content" in Modeler.
- Low and high angles are now returned by the Item Info `limits` function even if the corresponding limits are turned off.
- A condition that could cause mesh info queries to fail after an object is cleared from the scene has been fixed.
- All enabled environment plug-ins are now evaluated (although this can be a waste of time since each one will overwrite the results of the previous one).
- The `PreviousItem` and `NextItem` commands now skip over locked items.
- When Adaptive Sampling is on and Antialiasing is off, the `minSamplesPerPixel` from the Scene Info global is now zero, since some pixels may be totally skipped.
- The Item Info `controller` function now returns correct values.
- Button names specified by generic plug-ins are now used if present.

- The envelope commands are now safe to use when there is no current item.
- Object serial numbers are now computed during rather than after cloning so that plug-ins can get the final object names as they are created.
- Since they may draw semi-transparent primitives, custom objects are now processed after all other 3D elements (including the grid).
- [**B**] During scene file loading, newline characters after plug-in server names were not being read before the plug-ins loaded their data. This has been fixed.

7.5 Changes

May 1, 2002

This is a list of the changes in the LightWave 7.5 release that affect the SDK. The changes include a new Layout tool class and a new globals, as well as enhancement of many of the info globals defined in `lwrender.h`.

Structures associated with the Item Info, Object Info, Light Info, Scene Info, and Interface Info, globals were changed, but the shader version number and the global name strings weren't incremented. In all cases, the changes involve members added to the ends of structures, so they have no effect on existing plug-ins. New plug-ins can use the Product Info global to verify that they're running in at least LightWave 7.0 before they use the new structure members.

`lwcustobj.h`

- A bit definitions for the `flags` function return value has been added. The `LWCOF_VIEWPORT_INDEX` flag tells layout to use the viewport number instead of its type in the LWCustomObjAccess view element
- `The LWCOF_NO_DEPTH_BUFFER` flag prevent textured quads from being obscured by OpenGL drawing that is done in front of them.
- The `text` function justification argument values are now defined.

`lwfilter.h`

- A new flag tor the PixelFiltel, `LWPFF_EVERYPIXEL`, has been added. It tells layout to call this filter for every pixel, despite adaptive sampling settings.

`lwhost.h`

- A formal define for the content directory string passed to the Directory Info global, `LWFTYPE_CONTENT`, has been added.

`lwenvel.h`

- A new event code, `LWCEVNT_TRACK`, has been added to signal temporary changes to channels done during interactive editing. Caution should be used when handling these events, ad they will be plentiful, and ambitious callbacks could result in significant performance degradation.
- A new event code, `LWCEVNT_CREATE`, has been added to signal the addition of a channel to a group.
- A new event code, `LWCEVNT_RENAME`, has been added to signal than a channel group has been renamed.

`lwlaytool.h`

- The new plug-in class, LayoutTool actually works now.

`lwmaster.h`

- An `LWEVNT_RENDER_DONE` event code has been added (MasterHandler).

`lwmodtool.h`

- Per-vertex and per-polygon custom allocation available to MeshEdits is now available to MeshEditTools via the bit definitions in the return values of the `Test` function. These bits are conveniently defined by the `LWT_VMEM()` and `LWT_PMEM()` preprocessor macros, which take the memory size as arguments.

`lwtxtr.h`

- The `LWTEXTUREFUNCS_GLOBAL` global (Texture Functions) has incremented to "Texture Functions 3" due to the new tags supported by `getParam` and `setParam`.
- New tag definitions were added: `TXTAG_ACTIVE`, `TXTAG_INVERT` and `TXTAG_BLEND`.
- The possible blend modes have been enumerated:

```
typedef enum LWTextureBlendMode {
    TXBLN_NORMAL=0,
    TXBLN_SUBTRACT,
    TXBLN_DIFFERENCE,
    TXBLN_MULTIPLY,
    TXBLN_DIVIDE,
    TXBLN_ALPHA,
    TXBLN_DISPLACE,
    TXBLN_ADD  }
```

`lwserver.h`

- New server tag `SRVTAG_SELECTCMD` has been added to generate a command upon selection of the item to which the plugin is applied.

`lwrender.h`

- A function to retrieve the state of the UI motion locks (`axisLocks`) has been added to LWItemInfo (Item Info).
- Three new functions (`matteColor`, `thickness`, and `edgeZScale`) have been added to LWObjectInfo. The `thickness` function uses one of the following types:

```
LWTHICK_SILHOUETTE
LWTHICK_UNSHARED
LWTHICK_CREASE
LWTHICK_SURFACE
LWTHICK_OTHER
LWTHICK_LINE
LWTHICK_PARTICLE_HEAD
LWTHICK_PARTICLE_TAIL
```

- The LWObjectInfo `flags` function returns new flags `LWOBJF_MATTE`, and `LWOBJF_UNSEEN_BY_ALPHA`.
- Three new functions (`animFilename`, `RGBPrefix`, and `alphaPrefix`) have been added to LWSceneInfo.
- A new function, `shadowColor`, has been added to LWLightInfo.
- Four new bit definitions are available in the LWSceneInfo `renderOpts`:

`LWROPT_ENHANCEDAA`, `LWROPT_SAVEANIM`, `LWROPT_SAVERGB`, and `LWROPT_SAVEALPHA`..

- The LWInterfaceInfo structure now contains `autoKeyCreate`, with values defined as `LWAKC_OFF`, `LWAKC_MODIFIED`, and `LWAKC_ALL`. The `generalFlags` can now reveal the AutoKey state in the `LWGENF_AUTOKEY` bit.
- A new global (Viewport Info) has been added to facilitate custom drawing and tool handle processing. The server name is `LWVIEWPORTINFO_GLOBAL`.

Miscellaneous

- The Item Info `first` function now returns `LWITEM_NULL` when called with `LWI_BONE` type and `LWITEM_NULL` as the id of the parent object, instead of returning the id of the first bone in the scene..

Articles

This is an area of the documentation where I'd like to put tutorials, informal discussions and FAQ answers, most of which I'm hoping will be contributed by other plug-in authors as time goes on. I've included a page from my web site as an example, but the pages here don't have to be elaborate or technical. They just have to try to be helpful.

Commands

Commands are the natural counterparts of globals. While globals are used primarily to read information about LightWave's state, commands are used to change its state by loading and saving files, setting parameters, and performing operations. Most commands available to plug-ins are parallels of actions users can take through LightWave's interface.

With one exception described below, commands can only be issued from plug-ins of specific classes: LayoutGeneric and MasterHandler in Layout, and CommandSequence in Modeler. Layout and Modeler have command sets unique to each of them, but they also support a common set of commands that operate on components they share, such as the surface and graph editors and the image display.

- Modeler
- Layout
- Common

Two Methods

There are two ways to issue commands, the "lookup/execute" method and the "evaluate" method. Although both methods are available in both Layout and Modeler, the first is native to Modeler and the second to Layout, and there are minor differences in the way they're implemented in each program.

cmdcode = **lookup**(data, cmdname)
result = **execute**(data, cmdcode, argc, argv, [opsel], cmdresult)

> The `lookup` function returns an integer code corresponding to the command name. The command is issued by passing the command code to the `execute` function. Command codes are constant for a given session, so `lookup` only needs to be called once per command, after which the codes can be cached and then used in any number of calls to `execute`. The command's arguments are passed in an array of DynaValues.

Modeler's version of `execute` takes an additional `opsel` argument that determines which geometry will be affected by the command. It can be any one of the EltOpSelect codes except `OPSEL_MODIFY`. Modeler's `execute` returns 0 (`CSERR_NONE`) if it succeeds, or an error code if it fails. Layout's `execute` returns 1 if it succeeds and 0 if it fails.

result = **evaluate**(data, cmdstring)

The `evaluate` function uses a single string to issue the command. The command's name and its arguments are delimited by spaces.

The evaluate method usually requires you to write less code, particularly if you wrap it in a function that builds the command string. The following does this for a LayoutGeneric.

```
static int lwcommand( LWLayoutGeneric *local, const char *cmdname,
   const char *fmt, ... )
{
   static char cmd[ 512 ], arg[ 512 ];

   if ( fmt ) {
      va_list ap;
      va_start( ap, fmt );
      vsprintf( arg, fmt, ap );
      va_end( ap );
      sprintf( cmd, "%s %s", cmdname, arg );
      return local->evaluate( local->data, cmd );
   }
   else
      return local->evaluate( local->data, cmdname );
}
```

The `fmt` argument is a `printf` format string, and the variable number of arguments that follow it correspond to the arguments you'd pass to `printf`.

In Modeler, however, the lookup/execute method has a couple of advantages. It runs slightly faster, since Modeler doesn't have to perform the lookup or the "unstringizing" of the arguments itself, and it allows you to specify a selection criterion.

Layout Command Global

Layout makes available a global that allows plug-ins of any class, not just generics and masters, to issue commands. Currently, this global isn't a first-class citizen of the plug-in API. It isn't prototyped in the SDK headers, and it doesn't have its own document page. This is primarily because there are lots of ways it could be used unsafely, some of which are difficult to anticipate.

You can use this global by adding something like the following to your code.

```
#define LWCOMMANDFUNC_GLOBAL "LW Command Interface"
typedef int ( *LWCommandFunc )( const char *cmd );

LWCommandFunc *evaluate;
evaluate = global( LWCOMMANDFUNC_GLOBAL, GFUSE_TRANSIENT );
```

Note that if this global, or something like it, is eventually elevated to first-class status, its prototype will likely be different from the above.

A common use for this global is the application of dependent plug-ins. The HyperVoxels volumetric, for example, uses it from within its interface to apply or remove its related custom object plug-in.

But be careful. Test your use of this global thoroughly. Issuing commands at the wrong time or in the wrong context can easily cause a crash. In particular, never issue commands during rendering, and don't remove yourself, or the item you're applied to.

Common Elements

This page discusses the components that are common to all plug-ins. These are the structural components that form the bridge between LightWave and your plug-ins. They have funny names and do possibly unfamiliar things, so we need to introduce some terminology.

The *host* is the program, Layout or Modeler, for example, that runs your plug-ins.

A plug-in *module* is a file, usually with a `.p` extension, that contains one or more LightWave plug-ins. Any number of plug-ins can be compiled together into a single module. It's common for an image loader and an image saver to be together in the same file, for example.

Every plug-in file needs a *server description* that lists the plug-ins in the file, and every plug-in needs a special entry point function, its *activation function*. Both of these are defined in the `lwserver.h` header file. Each plug-in file also contains initialization and cleanup functions called `Startup` and `Shutdown`.

Server Description

The server description lists what your plug-in file contains. It's the first thing the host examines when it loads your module. The list appears in your source code as an array of ServerRecords.

```
typedef struct st_ServerRecord {
   const char     *className;
   const char     *name;
   ActivateFunc   *activate;
   ServerTagInfo  *tagInfo;
} ServerRecord;
```

`className`

A string containing the class of the plug-in. The class identifies what kind of plug-in this is. The header files for classes contain `#defines` for each class name. These are also listed in the documentation for each class.

`name`

A string containing the name by which LightWave will uniquely identify your plug-in. This is the name LightWave uses internally and saves in scene and object files. It's also the name displayed to the user if the plug-in doesn't supply at least one user name. The name must be a string of ASCII characters in the range 33 to 127 (note that this excludes spaces). Case is significant.

Although this allows punctuation and other special characters to appear in the name, you're strongly encouraged to limit names to those that would be legal identifiers in the C language. C identifiers contain letters, numbers and the underscore character (ASCII 0x5F). Image saver names, which by convention end with the default filename extension in parentheses, are an exception to this rule.

The use of non-alphanumeric initial characters to force your plug-ins to appear first, or together, in lexicographically sorted lists is discouraged. This practice may interfere with LightWave's internal name processing and may conflict with conventions that evolve in the future.

Each class has its own name space, so plug-ins of different classes can have the same name. Although you'll probably want to avoid giving unrelated plug-ins the same name, you *must* use the same name for the interface class associated with a handler. This is how the host matches a handler with its interface.

`activate`

The activation function. See below.

`tagInfo`

An array of tag strings that describe the plug-in. Among other things, this is where you list the name that will be displayed to your users in LightWave's interface.

Server Tags

The ServerRecord's `tagInfo` field is an array of ServerTagInfo structures.

```
   typedef struct st_ServerTagInfo {
      const char   *string;
      unsigned int  tag;
   } ServerTagInfo;
```

Each tag contains two codes combined using bitwise-or. The high word is the tag type, and the low word is the language ID. Not all of the tags are supported yet. Currently defined tag types include the following.

`SRVTAG_USERNAME`
: The name displayed to the user in LightWave's interface. Multiple user names for different locales can be provided by combining this type code with different language IDs. LightWave attempts to select the name that's most appropriate for the locale of the user's machine. Unlike the internal server name, there are no restrictions on what the string may contain.

Japanese strings should be encoded as JIS on Windows and EUC on Unix.

`SRVTAG_BUTTONNAME`
: The string that will appear on a button or in a popup list used to invoke your plug-in. This is usually an abbreviated version of your user name.

`SRVTAG_CMDGROUP`
: The LightWave interface organizes commands, including plug-ins, into command groups. The command group you specifiy determines the heading under which users will find your plug-in on menu and key customization dialogs. The command group can be a predefined group, or a new group created simply by listing its name.

`SRVTAG_SELECTCMD`

: The string in this tag will be executed as a command when an item with this plug-in applied is selected in Layout. This is useful for activating special tools for certain custom objects, among other things.

In general, the predefined group names are lowercase versions of the group names displayed in the interface. When using one of these groups, the language ID should be 0. Predefined group names are automatically translated to the locale of the user's machine. The following is a partial list of the predefined command groups.

Both	***Layout***	***Modeler***
	bones	
	cameras	
display	effects	create
file	items	construct
preferences	lights	edit
windows	motion	mappings
selection	objects	modify
additional	previews	polygons
	rendering	texture
	time	

`SRVTAG_MENU`

For plug-ins that can be activated as commands or tools (all Modeler classes, plus generics in Layout), the menu string specifies the location of the plug-in's node in LightWave's menu system. Like command groups, the menu string can refer to predefined or custom nodes. They can also specify a "path" resembling a filename, with optional root menu nodes followed by a colon and other nodes separated by forward slashes, and the nodes can be a mix of predefined and custom. The path

```
"tools/objects/Quadrics"
```

for example, would create a (custom) "Quadrics" popup on the (predefined) "Tools" tab, while

```
"polygon/Metaballs"
```

would create a "Metaballs" group. In general, the menu tag path has the form

```
"[menu:]tab[/group[/group...]]"
```

and the menu info tag can contain many of these strings separated by commas. The string

```
"multiply/replicate,LMB:Ultra Studio"
```

would place the command or tool into the standard location in the main menu and into a custom group in the left mouse button popup. It's even possible to place commands into the bottom command bar in Modeler, but this isn't recommended, since the screen real estate there is limited.

The predefined menu hierarchy hadn't been finalized at the time this document was last updated.

`SRVTAG_DESCRIPTION`

A line of text describing the plug-in. This might be displayed in the interface as hint text or as a description next to the user name in customization dialogs.

`SRVTAG_ENABLE`

A string defining the conditions under which the plug-in should be active. This is currently used for Modeler tools and commands to determine the enable state of the plug-in's button. Possible values include

`"pnt"` - active points
`"pol"` - active polygons
`"spnt"` - directly selected points
`"spol"` - directly selected polygons

Compound conditions, which would combine these into boolean expressions, aren't supported yet but may be in the future.

The language ID is a code indicating the language for the name string. The language IDs are identical to those defined in the Microsoft Win32 API and exposed in the Microsoft Visual C++ `winnt.h` header file. Bits 7 - 0 define the language group and bits 15 - 8 define the sublanguage. lwserver.h contains symbols for some of the more common language IDs.

0x0407	LANGID_GERMAN
0x0409	LANGID_USENGLISH
0x0809	LANGID_UKENGLISH
0x040a	LANGID_SPANISH
0x040c	LANGID_FRENCH
0x0410	LANGID_ITALIAN
0x0411	LANGID_JAPANESE
0x0412	LANGID_KOREAN
0x0419	LANGID_RUSSIAN
0x041D	LANGID_SWEDISH

Activation Function

The activation function is the entry point for the service provided by your plug-in. For some plug-in classes, this may be the only function the host calls in your plug-in (other than the startup and shutdown functions). For others, the activation function is where the host finds out about the plug-in's callback functions.

```
XCALL_( int )
MyActivate( long version, GlobalFunc *global, void *local,
   void *serverData );
```

`version`
: A class-specific version number. As development of LightWave continues, the interaction between the host and a given plug-in class is sometimes redefined. This number tells you, among other things, what version of the local data the host has passed. See the compatibility discussion for more information on using this value. In most cases, though, you'll test this value against the version number defined in the header file for your plug-in's class and return `AFUNC_BADVERSION` if they don't match.

`global`
: A function that gives your plug-in access to services provided by the host and by Global class plug-ins. See the pages about the global function and Global plug-ins for more information.

`local`
: Class-specific data. Each plug-in class receives different data through this argument. The documentation for each class, in fact, is primarily concerned with describing the class's local argument. For handler classes, this points to a structure that the plug-in needs to fill. The host gets pointers to other functions in your plug-in this way.

`serverData`
: The data pointer returned by the startup function. Unless you replaced the default startup function, you should ignore this argument. In particular, don't try to dereference the pointer, since on most systems it contains an invalid (although non-NULL) address.

The activation function returns a code that tells the host whether the plug-in was activated successfully.

AFUNC_BADVERSION

The version argument differs from what your plug-in supports. In some cases the host will try again with a lower version number.

AFUNC_BADGLOBAL

A call to the global function failed.

AFUNC_BADLOCAL

Your plug-in doesn't like something in the local data.

AFUNC_BADAPP

The host is a program you don't support.

AFUNC_OK

Return this when none of the other values is appropriate.

Startup and Shutdown

These two optional entry points allow the module to initialize itself when it is first loaded and to clean itself up before being unloaded.

```
   void *Startup( void );
   void Shutdown( void *serverData );
```

Most plug-in files don't require module-level initialization and cleanup. They use the empty startup and shutdown functions supplied by the SDK linker library.

The startup function is called when the plug-in is first loaded by the host. The return value is the data passed to the activation and shutdown functions as the `serverData` argument. Returning NULL from the startup function indicates failure, so even if a module has no real server data, it should still return something. The module's shutdown function is called just before the host unloads the module. Any allocated server data should be freed at this point.

Calling Convention

Functions in the plug-in are called directly by LightWave, and this is a potentially funky thing in some systems since they may be different environments. The `lwserver.h` header file defines an `XCALL_` macro that establishes the calling convention for each platform and compiler. `XCALL_` is applied to anything that preceeds the function name in definitions.

```
XCALL_( static const char * ) DescLn( LWInstance instance )
{ ...
```

All functions in your plug-in that can be called by LightWave need the `XCALL_` treatment, with the exception of the startup and shutdown functions.

Compatibility

This page discusses four different varieties of plug-in compatibility.

- *Backward* compatibility is the ability to use the same code, including the latest SDK headers and source, with both current and older versions of LightWave.
- *Forward* compatibility deals with writing code that won't break in future versions of LightWave.
- Compatibility *across platforms* allows you to use one code base to support more than one operating system or CPU type.
- *Product* compatibility means being able to use the same code in products that may be derived from LightWave or share some of its plug-in API (application programming interface).

Of these, backward compatibility is likely to be the greatest concern. Forward compatibility and compatibility across platforms are largely automatic for the plug-in author, as long as he or she writes to the specification in this documentation and uses the facilities provided by the SDK rather than platform-specific code. The requirements for product compatibility are difficult to predict at the moment, since no concrete examples of the need for it exist yet, but globals for determining the product and version number are available, and we'll introduce them here.

This discussion necessarily delves into some grubby details, so it'll be easier to follow if you're already familiar with the SDK. But if you're not, links to the information you need are provided.

SDK Versions

The plug-in SDK continues to evolve. Changes to it will appear with each release of LightWave. The SDK itself isn't versioned, however. Each class and global has its own version number. Your plug-in remains both forward and backward compatible with SDK changes by using, for classes, the version number passed as the first argument to your activation function, and for globals, increments embedded in the global's service name string.

The SDK defines symbolic names for the versions of each class. The version number for shaders, for example, is `LWSHADER_VERSION`, which as of this writing is defined as 4. Your shader's activation function will usually compare this to the version number passed as the first argument to the function and return `AFUNC_BADVERSION` if the two numbers don't match.

```
XCALL_( int )
MyActivate( long version, GlobalFunc *global, LWShaderHandler *local,
   void *serverData );
{
   if ( version != LWSHADER_VERSION )
      return AFUNC_BADVERSION;
```

This ensures that the LWShaderHandler being passed to you in the `local` argument is the same as the LWShaderHandler in your copy of the lwshader.h SDK header file. The lwshader.h header contains the line

```
#define LWSHADER_VERSION 4
```

as well as the definition of the structures used by shaders, including LWShaderHandler and LWShaderAccess. When you compile your plug-in using this header, the compiler renders the version checking code in your activation function as

```
if ( version != 4 ) ...
```

You test for version 4 because that's the version of the shader API defined in your copy of the header, and the version of the shader-related structures compiled into your plug-in.

LightWave will call your activation function with every version of LWShaderHandler it supports, until it runs out of versions or one of the calls succeeds. Forward compatibility is therefore automatic, as long as LightWave continues to support version 4 of LWShaderHandler.

But what happens when you update your copy of the SDK headers? `LWSHADER_VERSION` may have been incremented, yet you want to continue to support LightWave 6.x, which itself supports shaders no later than version 4.

First, your activation function must accept a range of versions. LightWave starts by calling your activation function with the highest version it supports, then with successively lower versions. (The exception is the interface activation for handlers, which for historical reasons starts at 1 and counts up.) Your plug-in is therefore activated with the highest version supported by both the plug-in and the LightWave it's running in.

```
XCALL_( int )
MyActivate( long version, GlobalFunc *global, LWShaderHandler *local,
   void *serverData );
{
   if ( version > LWSHADER_VERSION || version < 4 )
      return AFUNC_BADVERSION;
```

Then you have to decide how to handle the different versions of LWShaderHandler and other shader-related structures in the rest of your code. In many cases, changes to the API of a class are incremental. Existing structure members are retained, and new members are appended, making it possible to use the most recent versions of the data structures with previous versions of the API. You just need to remember not to use new members when you've been activated with an older version number.

The documentation includes a history of the changes made to the headers with each revision of LightWave. Look for this in both the changes lists and in sections labeled "History" on the pages for each class. Using this information, you can see how the current data structures differ from those in previous versions.

Globals work in a similar way. The SDK headers define symbolic names for the strings you pass to the global function. `LWMESSAGEFUNCS_GLOBAL`, for example, is the symbolic name of the messages global.

```
LWMessageFuncs *msgf;
msgf = global( LWMESSAGEFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( msgf ) { ...
```

By using the symbolic name, you ensure that the LWMessageFuncs structure returned by the global function is the same as the LWMessageFuncs defined in your copy of the lwhost.h SDK header.

The name strings underlying the symbols often contain trailing numbers or other incrementing characters. As of this writing, the string for the messages global, for example, is "Info Messages 2". If the LWMessageFuncs structure changes in the future, the new string will most likely be "Info Messages 3", but LightWave will continue to support "Info Messages 2". Again, your plug-in's forward compatibility with future versions of LightWave is automatic.

For backwards compatibility, you can request earlier versions of globals when the most recent version, or the version defined in your copy of the headers, isn't available. As with class-related structures, the data structures for globals in many cases evolve in backward-compatible ways. The LWMessageFuncs structure for the original "Info Messages" global looks like this.

```
   typedef struct st_LWMessageFuncs {
      void (*info)     (const char *, const char *);
      void (*error)    (const char *, const char *);
      void (*warning)  (const char *, const char *);
   } LWMessageFuncs;
```

The "Info Messages 2" global adds several functions to the end of the structure.

```
   typedef struct st_LWMessageFuncs {
      void (*info)     (const char *, const char *);
      void (*error)    (const char *, const char *);
      void (*warning)  (const char *, const char *);
      int  (*okCancel) (const char *ttl, const char *, const char *);
      int  (*yesNo)    (const char *ttl, const char *, const char *);
      int  (*yesNoCan) (const char *ttl, const char *, const char *);
      int  (*yesNoAll) (const char *ttl, const char *, const char *);
   } LWMessageFuncs;
```

The version 2 definition is backward-compatible with the pointer returned from a request for the original "Info Messages" global, as long as you remember not to use the fields added for "Info Messages 2".

Before LightWave 6.0

The revision of the plug-in API for LightWave 6.0 was the most substantial since the plug-in architecture was first introduced in 1995. Prior to 6.0, API revisions were incremental and had very little effect on existing source code. New globals were made available, and new members were appended to existing class structures. Plug-in authors could take advantage of the new features without changing much of their code and without sacrificing backward compatibility with older versions of LightWave.

This is also true within versions from 6.0 onward. But there is a great divide at 6.0, and the most difficult backward compatibility challenge involves bridging this divide. (This is a problem only for *new source code*. The situation for existing binaries is quite a bit simpler. With a few exceptions, plug-in binaries built with the pre-6.0 SDK will run in LightWave 6.0 but won't have access to any of the new features. Binaries built with the current SDK will *not* run in versions of LightWave prior to 6.0.)

With 6.0, the API doubled in size, and in keeping with the complete overhaul of LightWave itself, many familiar API structures were renamed, rearranged or removed. It's no longer possible to write plug-ins to both the current and 5.x APIs using the same code base and a single set of SDK headers. In some cases it simply won't make sense to continue to work within the limitations of the 5.x SDK, particularly as time passes and older versions of LightWave fade from view. But with that caveat in mind, it's still possible to write a single plug-in that uses new SDK features yet runs in LightWave 5.x.

One way to do this is to segregate any code that requires a particular API version. Your activation functions might look like the following.

```
XCALL_( static int )
Activate( long version, GlobalFunc *global, void *local,
   void *serverData )
{
   unsigned long prodinfo, major;

   prodinfo = ( unsigned long ) global( LWPRODUCTINFO_GLOBAL,
      GFUSE_TRANSIENT );
   major = LWINF_GETMAJOR( prodinfo );
   if ( major < 6 )
      return Activate5( version, global, local, serverData );
   else
      return Activate6( version, global, local, serverData );
}
```

The Product Info global is used here to distinguish between major versions of LightWave. `Activate5` is the activation function you would have written, had this been an exclusively 5.x plug-in, and `Activate6` is the 6.0 version. `Activate` covers both functions and is the activation function that should be listed in the ServerRecord. `Activate5` and `Activate6` reside in two different .c files, each of which includes the appropriate version of the headers and contains the version-sensitive callbacks. The code that doesn't depend on the API version can be called from both of these files.

In the 6 SDK, the ServerRecord structure was extended to include a `tagInfo` member, and the value of the `sysVersion` member of the ModuleDescriptor structure was incremented. (These structures are defined in lwserver.h and lwmodule.h in the 6.0 SDK, and in splug.h and serv_w.c in the 5.x SDK.) The change in `sysVersion` prevents plug-ins linked with the 6 SDK library from being loaded by earlier versions of LightWave that don't know about `tagInfo`. Your 5.x/6 hybrid plug-in must therefore be linked with the 5.x SDK library, and it can't include `tagInfo` in its ServerRecords.

Product Info and System ID

The Product Info global returns the identity of the host (e.g. LightWave), its major and minor version numbers, and the build number. The System ID global tells you which specific program you're running in (e.g., Layout, Modeler or Screamernet). If your plug-in uses features that only appear in certain versions of LightWave, or in LightWave but not in other programs that may share the LightWave SDK, or in Layout but not Modeler, you can use these two globals so that you can either bracket the affected code or fail gracefully.

Note that this is only necessary when the class version number or the global service name aren't sufficient by themselves to ensure compatibility, as in the 5.x example above. The need also arises in cases where the LightWave programmers, proving they are only human, forget to increment a version number when they make a change to a header.

Returning to our shader example, LightWave 7.0 added four fields to the LWShaderAccess structure without a corresponding change to `LWSHADER_VERSION`. As the History section of the shader page points out, you'll need to use the Product Info global to ensure that you're running in at least LightWave 7.0 before you try to read or write those new LWShaderAccess fields.

The least convenient but most reliable version indicator is the build number. You may occasionally need this in order to identify minor patches that retain the same major and minor version numbers. Older versions of a given LightWave component will always have smaller build numbers, so that you can reliably use inequalities to test whether the current program is *at least as* old or new as a specific build. The build numbers for LightWave Layout and Modeler are displayed in their About boxes.

Platforms

LightWave is available on more than one operating system. You can build a version of your plug-in for each of these operating systems and platforms without the use of any platform-specific source code. LightWave supports this by providing services for file I/O and user interface construction (panels, xpanels, requesters, messages, file dialogs, color dialogs, previews, presets and monitors, for example) that hide details specific to each platform. You just need to recompile.

The SDK requires you to define certain preprocessor symbols to distinguish between platforms. The lwdisplay.h header uses these, for example, to selectively compile different versions of the structure returned by the Host Display Info global. Under Windows, your plug-in receives an HWND for the LightWave component's main window, while on the Mac, it receives a WindowPtr. Which one you get will depend on whether you define `_WIN32` or `_MACOS` when you compile.

Although platform independence is usually a good thing, the SDK by no means requires it. Plug-ins aren't exotic objects on any platform. They're shared libraries on the Mac and DLLs in Windows, and they can do everything that other dynamically linked code can do on those platforms.

Under Windows, your plug-in can include an entry point function, usually called `DllMain`. The entry point is a function Windows calls when LightWave loads your plug-in (or when any process or thread links to a DLL). In order to gain access to resources (dialog box templates, bitmaps, icons, version data) you've linked into your .p file, you need to know your module handle, which you receive as the first argument to your entry point function.

```
#ifdef _WIN32
static HINSTANCE hdll;

BOOL WINAPI DllMain( HINSTANCE hInstance, ULONG reason,
   LPVOID reserved )
{
   if ( reason == DLL_PROCESS_ATTACH )
      hdll = hInstance;
   return TRUE;
}
#endif /* _WIN32 */
```

Later, you can pass `hdll` as the first argument to Win32 functions like

`LoadIcon` and `DialogBox`.

Parting Tips

- Always check the activation version (the first argument to the activation function).
- Always check the value returned by the global function. It may be NULL if the global you've requested doesn't exist in the LightWave that's running you.
- You can use the `lookup` function to see whether a command is available.
- Write to the spec. Don't make assumptions about undocumented LightWave internals. Don't try to dereference opaque pointers or read past the ends of buffers, and don't rely on buffers being contiguous or persistent, or in the same form internally as they are in the SDK.
- When possible, follow LightWave conventions.
- Use the Product Info and System ID globals to find out what program is calling you.
- Use platform-independent services for file I/O and user interfaces.

Compiling LightWave Plug-ins

LightWave plug-ins on all platforms are ordinary operating system objects (they're DLLs under Windows, for example), so building them is pretty straightforward. You'll need to define a couple of preprocessor symbols and export one variable name, and you'll need to compile and link with a bit of code supplied with the SDK. LightWave plug-ins are ordinarily given a ".p" filename extension, although this isn't required.

Preprocessor Symbols

The SDK header files rely on preprocessor symbols to identify the operating system and CPU of the host system. These are currently used to define system-specific versions of the `XCALL_` macro and the `HostDisplayInfo` structure (don't worry if you don't know what those are yet), and they determine what is stored in the system identification fields of the module descriptor. The operating system defines are

```
   _WIN32   /* Microsoft Windows */
   _MACOS   /* Macintosh */
   _XGL     /* Unix */
```

and the CPU defines are.

```
   _X86_    /* Intel and Intel-compatible */
   _ALPHA_  /* Alpha AXP */
   _PPC_    /* PowerPC */
   _MIPS_   /* MIPS */
```

You need one symbol from the first list and one from the second, and you need to pass them to the compiler as preprocessor defines.

Exported Variable

The linker needs to export the symbol `_mod_descrip`. LightWave looks for this module descriptor data structure by name when it attempts to load a plug-in. Symbol export is handled differently in different development environments, but it's often a linker command line option.

SDK Library

The SDK ships with source code files defining the `_mod_descrip` structure, default `Startup` and `Shutdown` functions, and a default names array. Each plug-in you create must include `servmain.c` and must be linked with `server.lib`.

Before compiling any plug-ins, you'll need to build `server.lib` for your system. Create a statically linked library containing `servdesc.c`, `username.c`, `startup.c`, and `shutdown.c`, all of which you should find in the `SDK/source` directory. Name the library `server.lib`. Define the operating system and CPU preprocessor symbols described previously when compiling the library source.

You may want to create more than one version of `server.lib` with different compiler settings (debug and release versions, for instance).

Debugging Notes

Don't forget that before you can run or debug your plug-in for the first time, you need to add it to the plug-in list in LightWave.

In general, you must quit and restart LightWave each time you rebuild your plug-in and want to run it. Your plug-in is cached in memory for the life of a LightWave session, so LightWave won't see the changes to your plug-in until it quits and restarts.

LightWave Modeler supports a debug command line switch. When started with the -d switch, Modeler adds an "Unload Plug-ins" command. (This appears in the Modeler/Plug-ins menu with the default configuration, but may not appear in custom menu configurations.) Activating this command forces Modeler to unload all unlocked plug-in modules, so that when you execute your plug-in again it will be reloaded from disk.

Beginning in LightWave 6.5, Modeler's -d switch can take an argument (-d*filename*) which tells it where to write debug information. This can be useful for figuring out why plug-ins aren't loading, or for looking at trace information generated by XPanels.

Microsoft Windows

Plug-in modules under Windows are Win32 dynamic link libraries (DLLs). You don't need to create an import library (.lib) or an export file

(.exp) for plug-in DLLs, but you will need to export `_mod_descrip`. One way to do this is to include a module definition (.def) file containing an `EXPORT _mod_descrip` directive. You can use the default `source\serv.def` file provided with the SDK for this.

Win32 DLLs have a standard entry point function named `DllMain`. You don't need to provide a `DllMain` for your LightWave plug-ins unless, for example, the user interface is built with Windows interface components that require the DLL's instance handle. (But consider building your interface using the platform-independent components provided with the plug-in SDK.)

The alignment of structure members in your DLL must match LightWave's.

Data type	***Address must be...***
`char`	any
`short (16-bit)`	even
`int, long (32-bit)`	divisible by 4
`float`	divisible by 4
`double`	divisible by 8
`structures`	aligned for the largest member
`unions`	aligned for the first member

The recipes for specific compilers discuss what, if anything, you need to do to ensure that your plug-in's data is properly aligned.

If you decide to use makefiles to build your plug-ins, they should contain lines resembling the following:

```
   LWSDK_FLAGS = -D_X86_ -D_WIN32

   .c.obj:
      $(CC) $(CFLAGS) $(LWSDK_FLAGS) $*.c $(LWSDK_SRC)servmain.c

   .obj.p:
      $(LINKER) -dll -out:$@ -def:$(LWSDK_INCL)serv.def $*.obj \
       $(LWSDK_LIB)server.lib $(OTHER_LIBS)
```

In other words, define the symbols `_X86_` (or `_ALPHA_`) and `_WIN32`, include `servmain.c` in the list of source code files, include the module definition file `serv.def` so that `_mod_descrip` is exported, and link with `server.lib`.

Microsoft Visual C++

To build an MSVC version of the SDK library,

- Create a new project workspace, or insert a new project into an existing workspace. The project type should be "Static Library." Name the project "`server`".
- Settings dialog, C/C++ tab, add `_X86_` (or `_ALPHA_`) and `_WIN32` to the preprocessor definitions field.
- In the field for additional include directories, type the path to the plug-in SDK `include` directory.
- Add `servdesc.c`, `username.c`, `startup.c`, and `shutdown.c` to the project. These are located in the `SDK\source` directory.
- Build `server.lib`.

To create a plug-in,

- Create a new project workspace, or insert a new project into an existing workspace. The project type should be "Dynamic-Link Library."
- Settings dialog, C/C++ tab, add `_X86_` (or `_ALPHA_`) and `_WIN32` to the preprocessor definitions field.
- In the field for additional include directories, type the path to the plug-in SDK `include` directory.
- Add your source files to the project. Also add `servmain.c`, `server.lib` and `serv.def`.

Accept the default settings for the calling convention (__cdecl), alignment (8 byte) and runtime library (multithreaded or multithreaded debug, for the release and debug versions, respectively). If you've built both debug and release versions of `server.lib` (and this is recommended), make sure you list the appropriate one for the debug and release versions of your plug-in.

You're ready to build your plug-in. To debug it,

- Settings dialog, Debug tab, enter the full path to `lightwav.exe` or `modeler.exe`, as appropriate, in the field labeled "Executable for debug session."
- Set the working directory to the directory containing the LightWave executables.

- Build a debug version of your plug-in.

Hit F5 to begin debugging. The debugger will warn you that the LightWave executable doesn't contain any debugging information, but that's okay. Your plug-in does have this information, which the debugger will find as soon as your plug-in is started by LightWave.

Borland C++ 4.52

Information provided by Michal Koc.

Before creating any plug-ins, you'll need to build a Borland version of the SDK library.

- Create a new project
- Set the Target Type to Static Library (for .dll) [.lib]
- In Standard Libraries, mark Static
- Set the Platform to Win32
- Set the Target Model to GUI
- Add nodes with the files `servdesc.c`, `username.c`, `startup.c`, and `shutdown.c`
- Options/Project/Directories, set the include and lib paths
- Options/Project/Compiler/Defines, add `_X86_` and `_WIN32`
- Options/Project/Compiler/Code Generation, set fastthis
- Options/Project/32-bit Compiler, set Processor and Data alignment to 8 bytes
- Compile

To build a plug-in,

- Create a new project
- Set the Target Type to Dynamic Library [.dll]
- In Standard Libraries, mark Dynamic
- Set the Platform to Win32
- Set the Target Model to GUI
- Add nodes with `servmain.c`, `server.lib`, and your source files; remove unused nodes
- Options/Project/Directories, set include and lib paths
- Options/Project/Compiler/Defines, add `_X86_` and `_WIN32`
- Options/Project/Compiler/Code Generation, set fastthis
- Options/Project/32-bit Compiler, set Processor and Data aligment to 8 bytes

- Compile

GNU gcc/Mingw32
Information provided by Dan Maas

You can build LightWave plug-ins with Win32 GNU distributions. The procedure given here was developed and tested with the Mingw32 distribution.

Before creating any plug-ins, you'll need to build a GNU version of the SDK library. (Some of the command lines below wrap to a second line here, but they should be entered on a single line.)

- `cd` to the SDK `source` directory.
- Compile the library sources.

```
gcc -c -D_WIN32 -D_X86_ -O6 -I$(LWSDK_INCL) servmain.c servdesc.c username.c
startup.c shutdown.c
```

- Assemble the library.

```
ar r libserver.a servdesc.o username.o startup.o shutdown.o
```

To build a plug-in,

- Compile your source files.

```
gcc -c -D_WIN32 -D_X86_ -O6 -I$(LWSDK_INCL) myplug.c
```

- Link to the SDK library and generate a DLL.

```
dllwrap -o myplug.p --export-all --dllname myplug.p myplug.o
$(LWSDK_LIB)servmain.o $(LWSDK_LIB)libserver.a
```

An equivalent makefile would look like this:

```
LWSDK_CFLAGS = -D_WIN32 -D_X86_ -O6

%.o: %.c
gcc $(LWSDK_CFLAGS) -I$(LWSDK_INCL) -c $<

myplug.p: myplug.o
   dllwrap -o $@ --export-all --dllname $@ \
   myplug.o $(LWSDK_LIB)servmain.o $(LWSDK_LIB)libserver.a
```

Watcom C++ 10.0a

(On June 30, 1999, Sybase, Inc., sent an "end of life" letter to registered owners of Watcom C/C++ announcing that version 11.0 of the compiler would be its last. Watcom is therefore unlikely to play a role in future LightWave plug-in development. This section remains useful, however, as an illustration of the incompatibilities you may encounter with some compilers.)

In Watcom terminology, plug-ins are NT DLLs, so that should be your target type. `server.lib` should also be built as an NT object.

To build a plug-in,

- Add the SDK include directory to the include path [**-i**]
- Disable stack depth checking [**-s**]
- Add macro definitions [**-d**] `_X86_` and `_WIN32`
- Change char default to signed [**-j**]
- Use 8-byte structure alignment [**-zp8**]
- Use the 32-bit flat memory model [**-mf**] (the default)
- Use the stack-based calling convention [**-5s**]
- Export [**exp**] `_mod_descrip`

There's an important mismatch in calling conventions that apparently can't be solved with a compiler switch or a pragma. When using the stack-based calling convention, which plug-ins must, Watcom 10.0a expects functions that return floating-point numbers to put them in specific registers, while LightWave's code leaves them at the top of the FPU stack. You'll encounter this whenever a plug-in compiled with Watcom needs to call a plug-in SDK function that returns a double.

What happens can be illustrated with a little assembly-ish pseudocode. Given

```
   double routine( void );
   double result;
   result = routine();
```

the different ways the function call is handled are

```
   Microsoft:  call routine
               fstp result          ; pop ST(0) into result

   Watcom:     call routine
               mov result,   eax    ; move edx:eax into result
```

```
                mov result+4, edx
```

When compiled in Microsoft Visual C++, `routine` leaves its return value at the top of the FPU stack, which is popped into `result`. In Watcom 10.0a, `routine` leaves its return value in the register pair `edx:eax`, which is then `mov`ed into `result`.

The workaround for this involves adding a bit of inline assembly language to each source file that contains a call to a LightWave function returning a double. At the beginning of each such file, put

```
static double fac;      /* floating point accumulator */
extern void sdk_fstp( void );
#pragma aux sdk_fstp = "fstp fac"
#define SDK_DBLRTN( x ) \
   sdk_fstp();          \
   x = fac;
```

This uses Watcom inline assembly to load the contents of ST(0) into a fixed memory location, from which the value can be copied. Calls that would look like this

```
result = objInfo->dissolve( objID, t );
```

must be changed to

```
objInfo->dissolve( objID, t );
SDK_DBLRTN( result );
```

The SDK function is called without assigning its return value. The assembly instruction `fstp fac` is inserted after the call to retrieve the return value, then `fac` is copied into `result`.

Macintosh

Plug-in modules on the Mac are PowerPC shared libraries of type `shlb`.

Metrowerks CodeWarrior

Before compiling any plug-ins, you'll need to build a CodeWarrior version of the SDK library.

- Create a new, empty project.
- Target Settings/Linker, choose Mac OS PPC Linker.

- PPC Target/Project Type, choose Library.
- PPC Target, name the file `server.lib`.
- PPC Processor, set alignment to PowerPC (the default).
- Define the `_MACOS` and `_PPC_` preprocessor symbols.
- Access Paths/Systems Paths, add the SDK include directory.
- Add `servdesc.c`, `username.c`, `startup.c`, and `shutdown.c`. These are located in the SDK `source` directory.
- Build.

To create a plug-in project,

- Create a new, empty project.
- Target Settings/Linker, choose Mac OS PPC Linker.
- PPC Target/Project Type, choose Shared Library.
- Define the `_MACOS` and `_PPC_` preprocessor symbols.
- Access Paths/Systems Paths, add the SDK include directory.
- PPC PEF/Export Symbols, choose the method you'll use to export `_mod_descrip`. See the CodeWarrior shared library documentation for an explanation of the available methods.
- C/C++ Language, turn on Relaxed Pointer Type Rules and Enums Always Int.
- Add your source code files to the project, along with `servmain.c`.
- Add `server.lib` and `MSL Runtime-PPC.Lib` to the project. If the plug-in calls Mac Toolbox routines, you'll also need to add `InterfaceLib` (or `CarbonLib` if you're compiling for OS X), but consider building your interface using the platform-independent components provided with the plug-in SDK instead.

You can globally define `_MACOS` and `_PPC_` by putting them in a .h file and using that file as the C/C++ Language Prefix File (in Language Settings). To export `_mod_descrip`, you can create a text file containing simply "_mod_descrip". Call the file `server.exp`. In PPC PEF/Export Symbols, choose the "Use .exp file" method, and then add `server.exp` to the project.

In later versions of CodeWarrior, you may find that the compiler won't accept any sort of conversion between char * and const char *. To work around this, you can add the following pragma:

```
#pragma old_argmatch on
```

Building plug-ins for OS X isn't much different from building them for previous operating systems on the Mac, but you must link with `CarbonLib` instead of `InterfaceLib`, and your code must be Carbon compliant. See the Carbon Porting Guide (an Adobe Acrobat PDF) at Apple's Developer website.

CodeWarrior up to version 6.2 cannot debug on OS X. Debugging in MacOS 9 is sufficient in most cases, but if you need to debug in X, you can use **gdb** in a terminal window.

As of this writing, LightWave for OS X is a CFM application. When a CFM app is run, OS X actually runs a mach-o wrapper app that loads the program. The wrapper is called `LaunchCFMApp`, located in

```
/System/Library/Frameworks/Carbon.framework/Versions/A/Support/
LaunchCFMApp
```

To begin debugging, use gdb to run `LaunchCFMApp`.

```
localhost% gdb /System.../LaunchCFMApp
```

In gdb, type `run` followed by the name and path for the LightWave component in which your plug-in runs.

Macintosh Programmer's Workshop

Information provided by Mark Nutter

Before building any plugins, you'll need to build an MPW version of the SDK library:

- In the MPW worksheet, select "Set Directory..." from the Directory menu and set the working directory to the `source` directory of the SDK.
- Select "Create Build Commands..." from the Build menu.
- Enter `server.lib` as the Program Name.
- Select "Static Library" as the Program Type.
- Check "PowerPC Only" as the Target.
- Add `servdesc.c`, `startup.c`, `shutdown.c` and `username.c` as the Source Files. These are in the SDK `source` directory.
- Click the Include Search Paths button and Add the SDK `include` directory.

- Click the PowerPC Options button and enter `-d _MACOS -d _PPC_` in the "C Options" field.
- Click the Create Make button to generate a makefile for `server.lib`.
- Select Build (Cmd-B) from the Build menu. The program name should come up as `server.lib` by default. Click OK to build `server.lib`. You will get a warning that "serverData" is not used within function "Shutdown", which you can ignore.

To build a plugin:

- Select "Set Directory..." from the Directory menu and set the working directory to the directory containing your plug-in's source files.
- Select "Create Build Commands..." from the Build menu.
- Enter your plug-in's name as the Program Name.
- Select "Shared Library" as the Program Type.
- Check "PowerPC Only" as the Target.
- Add the source files for your plug-in. Also add `servmain.c` (in the SDK `source` directory) and the `server.lib` you created.
- Click the Include Search Paths button and Add the SDK `include` directory.
- Click the PowerPC Options button and enter `-d _MACOS -d _PPC_ -typecheck relaxed` in the "C Options" field.
- Click the Exported Symbols button and enter `_mod_descrip` in the Export Symbols box.
- Click the CreateMake button to create your makefile.
- Build (Cmd-B).

Unix

Plug-in modules under Unix are shared object modules, or DSO files. When compiling, remember to define both `_XGL` and the preprocessor symbol for your CPU. `_mod_descrip` must be exported from the DSO, and `servmain.o` must be among the objects passed to the linker. The link line should include any other libraries that the plug-in would need as a stand-alone program.

```
.o.p:
   ld -shared -exported_symbol _mod_descrip -L$(SDK_LIB) \
    $(SDK_LIB)servmain.o $*.o -o $@ -lserver.lib $(OTHER_LIBS)
```

DynaValues and LWValues

The DynaValue data type defined in lwdyna.h is a union of containers for values of various base types. DynaValues are used as function arguments, with commands and requesters, for example, when the function must accept values of multiple types. LWValues are a variation on DynaValues used by Panels and defined in lwpanel.h.

The DynaValue typedef looks like this.

```
   typedef union un_DynaValue {
      DynaType       type;
      DyValString    str;
      DyValInt       intv;
      DyValFloat     flt;
      DyValIVector   ivec;
      DyValFVector   fvec;
      DyValCustom    cust;
   } DynaValue;
```

The elements of this union each support a different base type, and each base type can underlie several DynaTypes. (Two DynaTypes, `DY_NULL` and `DY_TEXT`, have no underlying value.)

Strings

```
   typedef struct st_DyValString {           DY_STRING
      DynaType    type;                      DY_SURFACE
      char       *buf;
      int         bufLen;
   } DyValString;
```

`bufLen` is the size of `buf` in characters. For the `DY_SURFACE` type, `buf` contains the surface name.

Integers

```
   typedef struct st_DyValInt {              DY_INTEGER
      DynaType    type;                      DY_BOOLEAN
      int         value;                     DY_CHOICE
      int         defVal;                    DY_FONT
   } DyValInt;                               DY_LAYERS
```

The value in `defVal` is used by requesters to reset control values. The value for `DY_CHOICE` and `DY_FONT` is a 0-based index into a list. The value for `DY_LAYERS` is a set of bitfields corresponding to layer numbers.

Floating-Point Numbers

```
   typedef struct st_DyValFloat {            DY_FLOAT
      DynaType   type;                       DY_DISTANCE
      double     value;
      double     defVal;
   } DyValFloat;
```

`DY_DIST` values are distances in meters.

Integer Vectors

```
   typedef struct st_DyValIVector {          DY_VINT
      DynaType   type;
      int        val[3];
      int        defVal;
   } DyValIVector;
```

Floating-Point Vectors

```
   typedef struct st_DyValFVector {          DY_VFLOAT
      DynaType   type;                       DY_VDIST
      double     val[3];
      double     defVal;
   } DyValFVector;
```

Custom Values

```
   typedef struct st_DyValCustom {           DY_CUSTOM
      DynaType   type;
      int        val[4];
   } DyValCustom;
```

`DY_CUSTOM` is used to encode values that don't fit one of the standard types. The interpretation of the values in the `val` array will depend on the context in which this type is used.

The Dynamic Conversion global provides a facility for converting between DynaValues of different types.

LWValues

Panels adds a generic pointer type to the list of types, but in all other respects, LWValues differ from DynaValues in name only. Structurally they're equivalent.

```
typedef union un_LWValue {
   LWType        type;
   LWValString   str;
   LWValInt      intv;
   LWValFloat    flt;
   LWValIVector  ivec;
   LWValFVector  fvec;
   LWValPointer  ptr;
   LWValCustom   cust;
} LWValue;
```

Although lwpanel.h defines others, only five LWTypes are needed to describe the values of all LWPanels controls.

```
LWT_STRING
LWT_INTEGER
LWT_FLOAT
LWT_VINT
LWT_VFLOAT
```

File Formats

This section documents the file formats that are native to LightWave. The binary formats are all based on the EA IFF 85 metaformat, which defines a syntax for binary data storage.

Handlers

Formally, handlers are plug-in classes that manage persistent instance data through callbacks identified in the activation function. Handlers don't simply run and exit. They tell LightWave in their activation functions where they can be reached, and then they hang around, waiting for LightWave to call them.

Instance Functions

An *instance* is a block of data you create to describe a specific invocation of your handler plug-in. An ItemMotionHandler, for example, can be invoked for any number of items in the scene, and might even be invoked more than once for a given item, but for each slot it occupies, the plug-in will create and use an instance specifically for that item slot. The instance data is where the plug-in settings for that invocation are stored, and every one of the plug-in's callbacks receives this data as one of its arguments.

The instance callbacks are where your plug-in creates, destroys, copies, loads, saves and describes each instance data block. They're collected together in an LWInstanceFuncs structure which is part of the local data passed to your activation function. Your activation function needs to fill in this structure to tell LightWave where your instance callbacks are.

```
typedef struct st_LWInstanceFuncs {
   void         *priv;
   LWInstance   (*create)  (void *priv, void *context, LWError *);
   void         (*destroy) (LWInstance);
   LWError      (*copy)    (LWInstance, LWInstance from);
   LWError      (*load)    (LWInstance, const LWLoadState *);
   LWError      (*save)    (LWInstance, const LWSaveState *);
   const char * (*descln)  (LWInstance);
} LWInstanceFuncs;
```

`priv`

Passed as the first argument to `create`. Set this to point to data you'd like your `create` function to have.

`instance = `**`create`**`( priv, context, error )`

Create an instance. Called, for example, when the user selects your plug-in on the interface and when Layout loads a scene or an object file that refers to your plug-in. Typically, you'll use `malloc` to allocate memory for a data structure, fill in some of the structure's fields with default values, and return the pointer to this structure. `priv` is the same as the `priv` field of the LWInstanceFuncs structure and contains whatever your activation function put there. The `context` varies depending on the plug-in class, but this is often an item ID for the item this instance will be associated with. If you can't create an instance, set `error` to an error message string and return NULL.

destroy`( instance )`
: Destroy an instance. Called, for example, when the user deselects your plug-in and when the scene is cleared. Typically you'll free any memory and resources obtained in `create` when this instance was created.

copy`( dest, source )`
: Copy the contents of the `source` instance to `dest`. If your instance data contains pointers, you may have to allocate memory for the pointer fields in `dest`.

load`( instance, loadstate )`
: Read instance data from a file. LightWave provides an LWLoadState containing functions used to read the data. See the File I/O page.

save`( instance, savestate )`
: Write instance data to a file using the LWSaveState functions. See the File I/O page.

descln`( instance )`
: Provide a human-readable description of the instance data. This is a single string displayed to the user on the LightWave interface and should be short enough to fit there. It can contain anything, but typically it contains shorthand descriptions of the most important settings. This serves as a reminder to the user, who would otherwise have to open your plug-in's interface to check these settings.

Item Handler Extensions

Handler classes that work on items in the scene provide a pair of callbacks that allow them to manage dependencies on other items.

```
typedef struct st_LWItemFuncs {
   const LWItemID *  (*useItems) (LWInstance);
   void              (*changeID) (LWInstance, const LWItemID *);
} LWItemFuncs;
```

`idlist = **useItems**( instance )`

Returns an array of items this instance depends on. If your plug-in's behavior is based on the parameters of other items (such as the positions of objects), you'll want to be re-evaluated after those parameters change, and you use this function to inform Layout of that. The array is terminated by `LWITEM_NULL`. The function can return NULL if the instance doesn't use any items. It can also return `LWITEM_ALL` to indicate that it wants to be evaluated after any change occurs.

`**changeID**( instance, idlist )`

Notification about a change in item IDs. This function is called if the IDs of items are going to change for any reason. The null-terminated item array passed to this function is of the form "old-1, new-1, old-2, new-2, ..." where the old ID is the value that is changing and the new ID is its new value. Clients should be careful to renumber each item only once.

The `changeID` callback may also be called when an item's data, such as the geometry of an object, has changed, and when called for this reason, the old and new item IDs will be the same.

Handlers with an LWItemFuncs in their local data are ordinarily called by Layout, but some handler classes (currently image and pixel filters, shaders and textures) can also be called by Modeler for previewing. When called by Modeler, the LWItemFuncs pointer will be NULL. *Handlers must test the value of the LWItemFuncs pointer before attempting to fill in the `useItems` and `changeID` fields.*

Render Handler Extensions

Certain handlers involved directly in rendering also provide callbacks for the start and end of a render session and the start of a new sampling pass.

```
   typedef struct st_LWRenderFuncs {
      LWError (*init)    (LWInstance, int);
      void    (*cleanup) (LWInstance);
      LWError (*newTime) (LWInstance, LWFrame, LWTime);
   } LWRenderFuncs;
```

`errormsg = **init**( instance, mode )`

Prepare the instance for a new rendering session. This is called before the first frame of a rendering session is begun. The mode will be either `LWINIT_PREVIEW` or `LWINIT_RENDER`. Returns a string containing an error message if an error occurs, otherwise returns NULL.

`**cleanup**( instance )`
: Called after the last frame of a rendering session is completed.

`errormsg = **newTime**( instance, frame, time )`
: Called at the start of a new sampling pass. This may be called more than once for the same frame but for slightly different times. Returns an error message string or NULL.

Example

This activation function is for an environment handler. The local data includes both the item and render extensions. All of the names on the right side of the equals sign are functions your plug-in provides.

```
   XCALL_( static int )
   Handler( long version, GlobalFunc *global,
      LWEnvironmentHandler *local, void *serverData )
   {
      if ( version != LWENVIRONMENT_VERSION )
         return AFUNC_BADVERSION;

      local->inst->create   = Create;
      local->inst->destroy  = Destroy;
      local->inst->copy     = Copy;
      local->inst->load     = Load;
      local->inst->save     = Save;
      local->inst->descln   = Descln;

      if ( local->item ) {
         local->item->useItems = UseItems;
         local->item->changeID = ChangeID;
      }

      local->rend->init     = Init;
      local->rend->cleanup  = Cleanup;
      local->rend->newTime  = NewTime;

      local->evaluate = Evaluate;
      local->flags    = Flags;

      return AFUNC_OK;
   }
```

The Interface Class

Each handler class has an associated interface class with its own activation function. As mentioned in the server description documentation, LightWave matches handlers with their interfaces by finding matching name strings in the ServerRecord array for a plug-in file.

The interface activation function receives the instance data as the first field of an LWInterface structure. This is the pointer returned by the handler's `create` function. The interface activation fills in the other fields of the LWInterface to tell LightWave how the plug-in wants its interface to be presented to the user.

```
typedef struct st_LWInterface {
   LWInstance  inst;
   LWXPanelID  panel;
   LWError    (*options) (LWInstance);
   LWError    (*command) (LWInstance, const char *);
} LWInterface;
```

`inst`

An instance returned by the handler's `create` function. This is read-only.

`panel`

An xpanel containing the controls for a user interface. The xpanel is created by calling the functions returned by the XPanels global. If this is NULL, LightWave will use the `options` callback instead. Some classes receive real estate on LightWave's panels for displaying their non-modal xpanel interfaces, so for those classes, the plug-in's interface is "always on," rather than being explicitly invoked by the user.

`error = `**`options`**`( instance )`

A callback that typically displays a modal panel. This is equivalent to the way the interface activation function itself worked in versions of LightWave prior to 6.0. LightWave calls this whenever the user explicitly opens the interface for the associated handler, often by double-clicking on the handler's name in a list. If `options` is NULL, LightWave uses the `panel` to display the plug-in's interface. (Exactly one of the `panel` and `options` fields should be non-NULL.)

`error = `**`command`**`( instance, cmdstring )`

A callback that processes batch commands. This isn't used in LightWave 6.0 and can safely be set to NULL.

Example

The avisave sample plug-in uses the `options` function to display the standard Windows codec selection dialog. This is also the approach to take if you're going to display a classic panels interface.

```
   XCALL_( int )
   Interface( long version, GlobalFunc *global, LWInterface *local,
      void *serverData )
   {
      if ( version != LWINTERFACE_VERSION )
         return AFUNC_BADVERSION;

      local->panel   = NULL;
      local->options = Options;
      local->command = NULL;

      return AFUNC_OK;
   }
```

`Options` uses the Win32 `ICCompressorChoose` function to display the dialog. The window handle of Layout's main window, obtained from the Host Display Info global, is passed as the parent window for the dialog.

```
   XCALL_( static LWError )
   Options( SampleAVI *avi )
   {
      ...
      result = ICCompressorChoose( hdi->window,
         ICMF_CHOOSE_ALLCOMPRESSORS, NULL, NULL, &cv,
         "SampleAVI Options" );
      ...
      return NULL;
   }
```

The NoisyChan sample creates an `LWXP_VIEW` xpanel that draws its controls in the lower right corner of Layout's graph editor window.

```
   XCALL_( static int )
   NoisyChannel_UI( long version, GlobalFunc *global, LWInterface *UI,
      void *serverData )
   {
      if ( version != LWINTERFACE_VERSION )
         return AFUNC_BADVERSION;

      GGlobal = global;

      UI->panel   = NoisyXPanel( global, UI->inst );
      UI->options = NULL;
      UI->command = NULL;

      return AFUNC_OK;
   }
```

The `NoisyXPanel` function creates the xpanel.

```
   LWXPanelID NoisyXPanel( GlobalFunc *global, NoisyData *dat )
   {
      LWXPanelFuncs *lwxpf = NULL;
      LWXPanelID     panID = NULL;
      static LWXPanelHint hint[] = {
         XpLABEL(0,"Noisy Channel"),
         XpEND
      };

      lwxpf = (LWXPanelFuncs*)(*global)( LWXPANELFUNCS_GLOBAL,
         GFUSE_TRANSIENT );
      if ( lwxpf ) {
         panID = (*lwxpf->create)( LWXP_VIEW, ctrl_list );
         if ( panID ) {
            (*lwxpf->hint)    ( panID, 0, hint );
            (*lwxpf->describe)( panID, data_descrip, NoiseData_get,
               NoiseData_set );
            (*lwxpf->viewInst)( panID, dat );
            (*lwxpf->setData) ( panID, 0, dat);
         }
      }
      return panID;
   }
```

Image I/O

This page describes the mechanism LightWave uses to move images to and from external files. The mechanism is defined in the lwimageio.h header file. The LWImageProtocol structure and the pixel type codes are used by the ImageLoader, ImageSaver, and AnimLoaderHandler classes.

The image protocol is used somewhat differently depending on which direction the image data is flowing. Loader plug-ins simply call the image protocol functions, but saver plug-ins *provide* these functions as callbacks that LightWave calls. Savers fill in the image protocol structure in their activation functions.

Because of the dual nature of the image protocol structure, there are places in the definitions where it's convenient to refer to the *source* and the *destination* of an image transfer. For loaders, the source is the plug-in and the destination is LightWave. For savers, the source is LightWave and the destination is the plug-in.

Image Protocol

Image data is transferred using calls to the functions in an LWImageProtocol. The `lwimageio.h` header file defines macros that loaders can use to slightly simplify calls to these functions. Both the functions and the corresponding macros are listed in the definitions.

```
typedef struct st_LWImageProtocol {
   int   type;
   void *priv_data;
   int  (*done)     (void *, int);
   void (*setSize)  (void *, int w, int h);
   void (*setParam) (void *, LWImageParam, int, float);
   int  (*sendLine) (void *, int, const LWPixelID);
   void (*setMap)   (void *, int, const unsigned char[3]);
} LWImageProtocol, *LWImageProtocolID;
```

type
: The pixel type code, described below. This identifies the kind of data that will be sent in `sendLine`.

priv_data

The first argument to the protocol functions. This is a pointer to data owned by the destination. Loaders just need to pass this along to the protocol functions (the macros hide this from you). Savers set this field to point to anything they like, typically a structure that holds data needed to process the save.

result = **done**(priv_data, error)
result = **LWIP_DONE**(protocol, error)

Called when there's no more image data to send. The incoming error code and the outgoing result can be any of the result codes defined below.

setSize(priv_data, width, height)
LWIP_SETSIZE(protocol, width, height)

Set the pixel dimensions of the image. The width and height are the number of pixels in a scanline and the number of scanlines, respectively. This is called before the first call to `sendLine`.

setParam(priv_data, paramid, intparam, floatparam)
LWIP_SETPARAM(protocol, paramid, intparam, floatparam)

Set other image parameters. In most cases, only one of the two parameter arguments will be used, while the other should be set to 0 by sources and ignored by destinations. The parameter ID can be one of the following.

`LWIMPAR_ASPECT` (float)

The pixel aspect ratio, defined as width/height. This will most often be 1.0 (square pixels, the default), but D1 NTSC images, for example, use a pixel aspect of 0.9, meaning that each pixel is 0.9 times as wide as it is high.

`LWIMPAR_NUMCOLS` (int)

The number of entries in the color table of an indexed-color image (an image of type `LWIMTYP_INDEX8`). Valid values are between 2 and 256.

`LWIMPAR_PIXELWIDTH` (float)

The physical size of a pixel in millimeters. Savers can combine this with the pixel aspect to record a DPI setting for file formats that support it.

`LWIMPAR_FRAMESPERSECOND` (float)

The playback rate in frames per second.

`LWIMPAR_BLACKPOINT` (float)

The black point of the image. The black point and white point define a nominal minimum and maximum intensity for an image. These are used, for example, when displaying the image on a device with limited dynamic range.

`LWIMPAR_WHITEPOINT` (float)

The white point of the image.

`LWIMPAR_GAMMA` (float)

The nonlinearity of the intensity encoding in the image.

The only parameter that loaders are required to set is the `LWIMPAR_NUMCOLS` value for `LWIMTYP_INDEX8` images.

result = **sendLine**(priv_data, y, scanline_pixels)
result = **LWIP_SENDLINE**(protocol, y, scanline_pixels)

Send one scanline from the image. Loaders must call `setSize` before the first call to `sendLine`. Scanlines are numbered from the top of the image, starting at 0. Loaders don't
have to send scanlines in a particular order, but savers will receive scanlines in top to bottom
order (or bottom to top if they specified the `IMGF_REVERSE` flag in their `sendData` call). A scanline begins with the leftmost pixel. The structure of the pixel data depends on the pixel type. Returns `IPSTAT_OK` or an error code.

setMap(priv_data, index, rgb)
LWIP_SETMAP(protocol, index, rgb)

Set the color of an entry in the color table of an indexed-color image. Loaders need to call `setParam` with a `LWIMPAR_NUMCOLS` parameter before calling `setMap` for the first time, but `setMap` may be called any time after that and before the first `sendLine`. The index identifies the color table entry, which is numbered from 0 to `numcolors-1`.

Pixel Data

The structure of the data in a scanline will vary, depending on the pixel type. Each scanline is an array of either unsigned bytes or floats. Bytes can contain any unsigned value between 0 and 255. The nominal range for float values is 0.0 to 1.0, but values outside that range may also appear.

Each pixel's data is contiguous--the scanline contains all of the channel values for the first pixel, followed by the values for the second, and so on. The `lwimageio.h` header file defines structures for many of the pixel types. You can use these to cast the `void *` argument in `sendLine` to a pointer of the appropriate type for the pixel data.

For each pixel type, the data is organized as follows.

`LWIMTYP_RGB24`
: Each scanline is an array of `unsigned char` in `RGBRGB...` order. The corresponding typedef is `LWPixelRGB24`.

`LWIMTYP_GREY8`
: Each scanline is an array of `unsigned char`, with one byte per grayscale pixel.

`LWIMTYP_INDEX8`
: Each scanline is an array of `unsigned char`, with one byte per pixel containing color map indexes.

`LWIMTYP_GREYFP`
: Each scanline is an array of `float`, with one float per pixel.

`LWIMTYP_RGBFP`
: Each scanline is an array of `float` in `RGBRGB...` order. The corresponding typedef is `LWPixelRGBFP`.

`LWIMTYP_RGBA32`
: Each scanline is an array of `unsigned char` in `RGBARGBA...` order and contains both RGB color and alpha channel values. The corresponding typedef is `LWPixelRGBA32`.

`LWIMTYP_RGBAFP`
: Each scanline is an array of `float` in `RGBARGBA...` order and contains both RGB color and alpha channel values. The corresponding typedef is `LWPixelRGBAFP`.

Error Handling

There are two ways that sources and destinations can notify each other of an error. The destination can return error codes from the `sendLine` and `done` functions, and the source can pass an error code to the destination's `done` function.

If a loader encounters an error while reading a file, it should stop sending data to LightWave and call `done`, setting the error argument to `IPSTAT_FAILED`.

If a saver's `done` callback is called with a non-zero error argument, the saver should perform whatever cleanup it thinks is appropriate, which may include removing the partially saved file, and return the same error code from `done`.

If a saver encounters an error while writing a file, it should return `IPSTAT_FAILED` from its `sendLine` and `done` callbacks. Note that the first `sendLine` call is a saver's first opportunity to signal an error to LightWave, so its callbacks will continue to be called after the error is detected and until `sendLine` is called. It's a good idea for savers to include an error field in their priv_data so that their callbacks can respond appropriately until LightWave can be told that something's gone wrong.

Example

See the ancounter animation loader and the iff image loader and saver samples.

Introduction

This is the documentation for the LightWave 3D Server Development Kit (SDK) for versions of LightWave beginning with 6.0. Although it will refer specifically to LightWave, other NewTek products may also support LightWave plug-ins. Documentation for versions prior to 6.0 and strategies for supporting multiple versions and products are discussed on the compatibility page.

I'll refer to myself in the first person in this introduction, just so you know that the documentation was in fact written by a human being, but to give you fair warning, this is a technical reference with more than a few dry spots. It makes significant demands on the reader, and depending on your degree of familiarity with LightWave, the C language, and 3D graphics programming, many parts might be completely opaque the first time through.

That's normal. Don't be discouraged by it. The information covered in this documentation is inherently difficult. It's not just you.

Because of this, I've tried very hard to be clear, concise and accurate. I've built on the work of others whose knowledge of this material comes directly from writing the host side of the plug-in API (application programming interface), and some of those same people have made an effort to explain the difficult parts to me. I've corrected a number of mistakes pointed out by readers of earlier drafts, but inevitably, errors and omissions remain, and chances are you'll find at least one.

In the following sections, I recommend books that provide the background you'll need, review a particular programming concept that may be unfamiliar to some, and give a quick tour of the plug-in system with links to important parts of the documentation. For developers with pre-6.0 plug-in experience, I'll also highlight some of the major changes that first appeared in LightWave 6.0.

Ernie Wright
December 2001

Programming Prerequisites

These pages document the C language interface to LightWave. They assume that you're comfortable with writing C code, so they won't teach you C. Specifically, they won't discuss abstract dynamic library concepts or the writing of re-entrant, thread-safe code. They also won't teach you 3D graphics programming or, for lack of a better term, the LightWave user paradigm. All of this information is available from other, better sources.

My favorite book for learning C programming is

- Al Kelley and Ira Pohl, ***A Book on C***, 4th ed., Addison-Wesley, ISBN 0201183994

You might also want a good algorithms book. I have

- Robert Sedgewick, ***Algorithms in C***, Addison-Wesley, ISBN 0201514257

3D graphics relies heavily on trigonometry and linear algebra, but in most cases you don't need an advanced knowledge of those subjects. You do need to know what sine, cosine and tangent are, and you need to know how to do vector and matrix arithmetic. Many of the books below include an appendix that reviews the basics. For greater depth, visit the nearest university bookstore and pick up whatever textbook they're using for the introductory courses.

Any good introductory textbook on graphics programming should provide an adequate foundation for understanding the fundamental concepts of computer graphics. I like

- F.S. Hill, ***Computer Graphics***, Macmillan, ISBN 0023548606
- Alan Watt, ***Fundamentals of Three-Dimensional Computer Graphics***, Addison-Wesley, ISBN 0201154420

but there are others. Note that both of these are more than 10 years old. That's okay, though. The fundamentals really haven't changed much in that amount of time. For more advanced texts, the canon would include

- Andrew Glassner (series ed.), ***Graphics Gems***, vol. I - V, Academic

Press, ISBNs 0122861663, 0120644819, 0124096735, 0123361559, 0125434553
- James Foley et al., ***Computer Graphics: Principles and Practice***, 2nd ed. in C, Addison-Wesley, ISBN 0201848406
- Alan Watt and Mark Watt, ***Advanced Animation and Rendering Techniques: Theory and Practice***, Addison-Wesley, ISBN 0201544121
- David Ebert et al., ***Texturing and Modeling***, 2nd ed., Academic Press, ISBN 0122287304
- James Murray and William vanRyper, ***Encyclopedia of Graphics File Formats***, 2nd ed., O'Reilly & Associates, ISBN 1565921615
- William Press et al., ***Numerical Recipes in C***, 2nd ed., Cambridge University Press, ISBN 0521431085
- Jackie Neider et al., ***OpenGL Programming Guide*** (the red book), Addison-Wesley, ISBN 0201632748

Also consider the two compilation volumes of Jim Blinn's articles in *IEEE Computer Graphics and Applications* and the often ground-breaking papers in the annual ACM SIGGRAPH *Proceedings*. In addition to these, you'll also occasionally find chapters and articles specific to writing LightWave plug-ins in trade books and magazines, some of which are written by members of the LightWave programming team. Don't forget the LightWave user manual, the best source of information about how the program works. And, of course, you'll find supplementary material at every level of complexity on the Internet.

But in the event none of this has failed to dissuade you from learning to program by writing LightWave plug-ins, I will review one programming concept that's fundamental to the way plug-ins work and which may not be easily understood solely by osmosis. If you already know what a callback is, feel free to skip ahead.

Function pointers

LightWave plug-ins make extensive use of function pointers. For people of my programming generation who grew up on BASIC, FORTRAN and Pascal, function pointers seem a bit exotic at first glance. In a linear, self-contained program, there are relatively few reasons to use them. But function pointers are just another kind of variable, and they become quite useful when two separate modules need to execute each other's code.

The type definition for a particular function pointer might look like this:

```
typedef int ( *FooFunc )( int, double );
```

This says that FooFunc is a function that returns an int (note: not an int *) and takes an int and a double as arguments. Given this definition, you can now declare variables of type FooFunc *,

```
FooFunc *foo;
```

You can write a FooFunc function,

```
int myfoo( int count, double size );
{
   return ( int )( size * count );
}
```

and assign this to your FooFunc variable,

```
foo = myfoo;
```

You can also pass FooFuncs as arguments to other functions.

```
int bar( FooFunc *foo );
```

Equivalently, you can explicitly prototype the `foo` function in `bar`'s function header.

```
int bar( int ( *foo )( int, double ));
```

The standard C runtime library contains at least two functions that take function pointers as arguments, `bsearch` and `qsort`, both usually prototyped in `stdlib.h`. The prototypes look something like this:

```
void *bsearch( const void *key, const void *a, size_t n, size_t size,
   int ( *compar )( const void *, const void * ));
void qsort( void *a, size_t n, size_t size,
   int ( *compar )( const void *, const void * ));
```

For both of these, you write the comparison function that ranks two elements from the array you're sorting or searching, and you pass this function as an argument to `bsearch` or `qsort`. You can sort or search almost anything using these functions, as long as you can write an appropriate comparison function.

Callbacks

The comparison function for `bsearch` and `qsort` is an example of a *callback*, a function you write for other modules to call. The C runtime calls your comparison function whenever it needs to rank two elements from your array.

Callbacks are common in user interface code for modern windowed environments, where they're used to handle "events" triggered by user actions. LightWave's built-in user interface facility uses callbacks in exactly this way, but callbacks are also used elsewhere in the plug-in API. Layout handler class plug-ins contain callbacks that are called at certain points during rendering, and Modeler plug-ins use callbacks to enumerate the points and polygons of an object.

You refer to callbacks, of course, by using function pointers.

A Quick Tour

This section is a brief, informal overview of the plug-in system and the way plug-ins work. It points to other areas of the documentation so that you know where the details are explained. You might also want to read through Part 1 of the Box tutorial in the Articles section. It covers much of the same ground by a different route, taking you step by step through the creation of a simple plug-in.

If you're a plug-in oldtimer from the days before LightWave 6.0 and you just want to get caught up, feel free to skip ahead.

Plug-ins are dynamically linked libraries of code that extend LightWave's capabilities. LightWave ships with dozens of plug-ins, and the source code for many of these is included in the plug-in SDK.

Plug-ins are divided into different types, called classes. These aren't actual C++ classes, although the idea is pretty much the same. The different classes plug into LightWave at different points and do different things. There are classes for

- loading and saving images, movies, objects and scenes
- moving items and modifying parameter channels

- rendering surfaces, textures, volumes and environments
- image processing
- creating and manipulating geometry
- custom color picker and file dialogs
- displaying rendered output
- command-based scene alterations
- controlling other plug-ins, and
- providing services accessible to other plug-ins.

All plug-ins have access to functions that provide information or services. These are called globals, and they can be used to get item positions, object geometry, surface settings, camera parameters, system and locale information, and a lot of other data. Globals are also used to build platform-independent user interfaces and to display common interface elements like file dialogs, color pickers, messages, and envelope and texture editors. You can even write your own globals.

A few plug-in classes can also issue commands. Most commands parallel actions the user can take through the LightWave interface. While globals are used primarily to read parameters, commands are used to set them.

More than half of the plug-in classes are handler classes. Unlike plug-ins that run when they're invoked and then exit, handlers have a persistent lifetime. They supply callbacks that LightWave can call at the appropriate time to perform their tasks. Most handlers are involved in rendering and are called at each frame to, for example, move objects, paint surfaces, or append the frame to a movie file. A few handlers respond to user interface events and manage interface objects.

Handlers can be applied, or invoked, multiple times. An item motion handler, for example, can control the motion of several different items in a scene. Each invocation of a handler is called an *instance*, and for each instance, a handler will create some data that it uses to keep track of that instance. The instance data is normally where handlers hold user settings and precalculated parameters, but it can be anything useful to the handler.

Handlers provide callbacks for loading and saving their instance data in scene and object files. The actual reading and writing of data in these files is accomplished through file I/O functions provided by LightWave. A global allows plug-ins of any class to use these same functions with other files, which can be useful for creating and reading platform-independent configuration files for your plug-in, for example.

The file I/O functions are one of several mechanisms shared by multiple classes with similar needs. Two more are the image I/O system used by image and animation loaders and image savers, and the raytracing functions used by shaders, volume renderers and filters.

Every plug-in has an activation function. This is the entry point for the plug-in, the function LightWave calls to begin the interaction between the program and your plug-in. For non-handlers, this is where all of the work of the plug-in is done, but for handlers, this is only where the plug-in tells LightWave how to find the plug-in's callbacks. The activation function has the same form for all plug-in classes, with a single argument that differs for each class.

A plug-in file can contain more than one plug-in. Each file contains an array of server records, one for each plug-in in the file. The server record for a plug-in lists the name and the class of the plug-in and the address of the plug-in's activation function. The server record array is an external data structure, a data block in the file that the operating system can locate by name. When LightWave first loads a plug-in, it asks the operating system to return the address of the server record array, and then it finds in that array the addresses of the activation functions the file contains. It can later call the activation function and, for handlers, obtain the addresses of other functions in the file.

Most plug-ins provide a user interface, and they normally display it as part of their activation function processing. Handler classes have associated interface classes whose activation functions are dedicated to this purpose. You can build your interfaces using platform-specific elements, but the SDK provides a complete, platform-independent system for building interfaces with elements that have LightWave's look and feel. This system is described on the Panels and XPanels pages.

So that you don't have to understand all of this all at once, start by trying to compile the example source that's included with the SDK. Once you're compiling successfully, you can experiment by altering the examples before moving on to creating your own plug-ins.

What's New

If you've written plug-ins for versions of LightWave prior to 6.0, much of the current API will seem familiar, but a lot has changed.

Classes - There are new classes for anim loading, channels, custom nulls in Layout, environment (backdrop) rendering, multiple plug-in mastering in Layout, custom Modeler tools, procedural textures, and volumetrics. Many of the other classes have been significantly enhanced, and nearly all of them have changed at least slightly to reflect the new architecture.

Globals - The number of globals has nearly doubled, and many of the familiar ones have grown substantially. Among the new globals are a number of user interface components, including the XPanels alternative to classic panels, standard access to the current color picker, and off-screen bitmaps that can be blitted onto panels. A revamped envelopes global is joined by related channels and variant parameter globals. Layout can tell you about the state of its interface, and modified handler instances can update Layout.

Plug-ins can save images through any installed image saver and use file I/O functions for creating and reading block-structured files. They can incorporate any of the installed procedural textures using globals for evaluating them and for displaying a standard texture editor to the user. Globals are available to help with managing presets and displaying previews. Layout now exposes detailed geometry data for every object in the scene and allows plug-ins to read and modify surface settings and manage particle system data.

Handlers - The local arguments to handler activation functions have changed. `create`, `destroy` and so on are still in there, but they've been reorganized and standardized. The interface activation functions associated with handlers now receive a structure rather than just their instance data, and many classes will be able to use this structure to draw their interface controls onto LightWave's own panels. Some handlers can now be run in Modeler to provide previews, and since the world looks different in Modeler, this case needs to be treated carefully.

Other Changes - The ServerRecord now includes an optional array of tag strings for each plug-in. Among other things, these tags allow you to list language-specific names for your plug-ins. If your plug-in supplies a list of names, the plug-in name LightWave displays to the user will depend on the locale of the user's system.

The `XCALL_INIT` macro has been deprecated, meaning that it's no longer required. You can still use it, but it doesn't do anything on any currently supported platform and isn't likely to return in the future.

All class name and global identifier strings have been assigned preprocessor symbols. For future compatibility, you should use these symbols, rather than the string literals, in ServerRecord references and calls to the `global` function.

Further Information

For updates, additional example source code, contact information, and information about the LightWave plug-in developers' Internet mailing list, visit NewTek's websites,

http://www.newtek.com
http://www.lightwave-outpost.com

LightWave Plug-in Server Development Kit

If you're thinking about writing plug-ins for LightWave, you've come to the right place!

For timely information about updates and general discussion of LightWave plug-in programming, join the plug-in mailing list.

See the change history to find out how releases and patches have affected the SDK.

In addition to the documentation, the SDK comprises header files, source files (for the small amount of server code linked to every plug-in), and sample plug-in source code. You can peruse these here or download them in a Zip file.

- `include` directory
- `sample` directory
- `source` directory
- `lwsdk7.zip`, which contains all three directories

Mesh Info

The LWMeshInfo structure describes the geometry of an object. You can get one of these from the Scene Objects and Object Info globals and from the access structure passed to the displacement handler `evaluate` function. What it contains can vary depending on how and when you obtain it. This structure is defined in the lwmeshes.h header file.

```
   typedef int LWPntScanFunc (void *, LWPntID);
   typedef int LWPolScanFunc (void *, LWPolID);

   typedef struct st_LWMeshInfo {
      void         *priv;
      void         (*destroy)     (LWMeshInfoID);
      int          (*numPoints)   (LWMeshInfoID);
      int          (*numPolygons) (LWMeshInfoID);
      int          (*scanPoints)  (LWMeshInfoID, LWPntScanFunc *, void *);
      int          (*scanPolys)   (LWMeshInfoID, LWPolScanFunc *, void *);
      void         (*pntBasePos)  (LWMeshInfoID, LWPntID, LWFVector pos);
      void         (*pntOtherPos) (LWMeshInfoID, LWPntID, LWFVector pos);
      void *       (*pntVLookup)  (LWMeshInfoID, LWID, const char *);
      int          (*pntVSelect)  (LWMeshInfoID, void *);
      int          (*pntVGet)     (LWMeshInfoID, LWPntID, float *vector);
      LWID         (*polType)     (LWMeshInfoID, LWPolID);
      int          (*polSize)     (LWMeshInfoID, LWPolID);
      LWPntID      (*polVertex)   (LWMeshInfoID, LWPolID, int);
      const char * (*polTag)      (LWMeshInfoID, LWPolID, LWID);
      int          (*pntVPGet)    (LWMeshInfoID, LWPntID, LWPolID,
                                     float *vector);
      unsigned int (*polFlags)    (LWMeshInfoID, LWPolID);
      int          (*pntVIDGet)   (LWMeshInfoID, LWPntID, float *vector,
                                     void *);
      int          (*pntVPIDGet)  (LWMeshInfoID, LWPntID, LWPolID,
                                     float *vector, void *);
   } LWMeshInfo;
```

`priv`

An opaque pointer to private data used internally by the mesh info functions.

`destroy``( meshinfo )`

Frees resources allocated by the process that created this LWMeshInfo. Call this when you're finished with the mesh info. *Note that this field may be NULL, indicating that you shouldn't attempt to free the mesh info.* Test the value of this field before trying to use it.

`npts =` **`numPoints`**`( meshinfo )`

Returns the number of points in the object. If the object contains dynamically created geometry, e.g. subdivision patches or metaballs, this number may include both the control points and the points created by subdividing.

`npols = **numPolygons**( meshinfo )`

Returns the number of polygons in the object, which may include polygons created by subdividing.

`result = **scanPoints**( meshinfo, pointscan_func, mydata )`

Enumerate the points in the object. The callback you supply is called for each point in the object. The `mydata` argument is passed to the callback and can be anything it might require. Enumeration stops if your callback returns a non-zero value, and this value is then returned by `scanPoints`. Otherwise it returns 0.

`result = **scanPolys**( meshinfo, polyscan_func, mydata )`

Enumerate the polygons in the object.

`**pntBasePos**( meshinfo, point, pos )`

Get the base, or initial, position of a point.

`**pntOtherPos**( meshinfo, point, pos )`

Get an alternate position for the point. This may be the same as the base position or it may be the position of the point after some transformation. The nature of the alternate position depends on how the mesh info was created.

`vmap = **pntVLookup**( meshinfo, vmap_type, vmap_name )`

Select a vertex map for reading by `pntVGet`. The vmap is given by its four-character identifier and its name string. The function returns a pointer that can be used later in `pntVSelect` to quickly select this vmap again. The pointer is NULL if no vmap was found with the given ID and name. The Scene Objects global allows you to examine the vmap database and retrieve the names of existing vmaps of a given type.

`dim = **pntVSelect**( meshinfo, vmap )`

Select a vmap for reading vectors. The vmap is identified by a pointer returned by `pntVLookup`. The function returns the vmap's dimension (the number of values per point).

`ismapped = **pntVGet**( meshinfo, point, val )`

Read the vmap vector for a point. The vector is read from the vmap selected by a previous call to `pntVSelect`. If the point is mapped (has a vmap value in the selected vmap), the `val` array is filled with the vmap vector for the point, and `pntVGet` returns true. The `val` array must have at least as many elements as the number returned by `pntVSelect`.

See also `pntVIDGet`.

`type = `**`polType`**`( meshinfo, polygon )`

Returns the type of a polygon. "Polygon" here refers to a number of different kinds of geometric atoms, including things like curves and bones. The polygon type codes are an extensible set of four-character identifiers. The header file `lwmeshes.h` defines the most common ones.

`LWPOLTYPE_FACE` - face
`LWPOLTYPE_CURV` - higher order curve
`LWPOLTYPE_PTCH` - subdivision control cage polygon
`LWPOLTYPE_MBAL` - metaball
`LWPOLTYPE_BONE` - bone

`nvert = `**`polSize`**`( meshinfo, polygon )`

Returns the number of vertices belonging to the polygon.

`point = `**`polVertex`**`( meshinfo, polygon, vert_index )`

Returns the point ID for a polygon vertex. Vertex indexes range from 0 to `nvert` - 1.

`tagname = `**`polTag`**`( meshinfo, polygon, tagID )`

Returns the tag string of the given type associated with the polygon. A null string pointer means that the polygon does not have a tag of that type. `lwmeshes.h` defines the most common polygon tags.

`LWPTAG_SURF`

The name of the surface applied to the polygon.

`LWPTAG_PART`

The name of the polygon group the polygon belongs to.

`ismapped = `**`pntVPGet`**`( meshinfo, point, polygon, val )`

Like `pntVGet`, but reads the per-polygon, or discontinuous, vmap vector for a polygon vertex. See also `pntVPIDGet`.

`flags = `**`polFlags`**`( meshinfo, polygon )`

Returns the flags associated with the polygon. the `EDPF_CCSTART` and `EDDF_CCEND` bits determine whether the first and last points in `LWPOLTYPE_CURV` polygons are control points rather than actual vertices. (The constants for these flags are defined in lwmeshedt.h.)

`ismapped = `**`pntVIDGet`**`( meshinfo, point, val, vmap )`
`ismapped = `**`pntVPIDGet`**`( meshinfo, point, polygon, val, vmap )`

Like `pntVGet` and `pntVPGet`, but these take the vertex map ID as an additional argument, so that it isn't necessary to first call `pntVSelect` to select the vertex map. This is important when your plug-in might be running in multiple threads, since the thread may change between the `pntVSelect` call and the `pntVGet` or `pntVPGet` calls.

Example

The SceneScan sample uses an LWMeshInfo obtained from the Object Info global to build arrays of points and polygons for an object, including vmap and surface data. See the `getObjectDB` function in `objectdb.c`.

Raytracing Functions

Several plug-in classes receive pointers to raytracing functions that allow them to probe the scene from any point of view.

These functions aren't valid in all contexts, since they depend on having information about the scene that may not always exist. When the Surface Editor renders its preview thumbnail, for example, it evaluates the active shaders, but in this previewing context, the `rayCast` and `rayShade` fields of the LWShaderAccess will be NULL. Always ensure that raytracing function pointers are valid before using them.

You may also need to safeguard against infinite recursion. A ray fired in the evaluation callback of a shader or (particularly) a volumetric may cause that callback to be re-entered. Shaders can use the `bounce` member of the LWShaderAccess to monitor the recursion level.

```
   typedef double LWRayTraceFunc (const LWDVector position,
                     const LWDVector direction, LWDVector color);

   typedef int    LWIlluminateFunc (LWItemID light,
                     const LWDVector position, LWDVector direction,
                     LWDVector color);

   typedef double LWRayCastFunc (const LWDVector position,
                     const LWDVector direction);

   typedef double LWRayShadeFunc (const LWDVector position,
                     const LWDVector direction,
                     struct st_LWShaderAccess *);
```

`len = `**`rayTrace`**`( position, direction, color )`

Trace a ray from the given location in the given direction in world coordinates. The return value is the length of the ray (or -1.0 if infinite) and the color coming from that direction. The direction argument is the outgoing direction and must be normalized (a unit vector).

`position`
: The world coordinates of the source of the ray.

`direction`
: A unit-length vector, the outgoing direction of the ray in world coordinates.

`color`

Storage for the color of the spot hit by the ray.

`lit = `**`illuminate`**`( lightID, position, direction, color )`

This function obtains the light ray (color and direction) hitting the given position from the given light at the current time step. The return value is zero if the light does not illuminate the given world coordinate position at all. The color includes effects from shadows (if any), falloff, spotlight cones and transparent objects between the light and the point.

`lightID`

The light, given by its LWItemID.

`position`

The world coordinates of the spot at which the illumination will be tested.

`direction`

Storage for the direction of the light ray computed by the function.

`color`

Storage for the color of the light ray.

Two special light IDs, `LWITEM_RADIOSITY` and `LWITEM_CAUSTICS`, allow shaders and pixel filters to account for global illumination. When using these IDs, the `direction` argument becomes an input rather than an output, specifying the desired sampling direction.

`len = `**`rayCast`**`( position, direction )`

This is a quicker version of the rayTrace function which only returns the distance to the nearest surface (or -1.0). It performs neither shading nor recursive raytracing.

`position`

The world coordinates of the source of the ray.

`direction`

A unit-length vector, the outgoing direction of the ray in world coordinates.

`len = `**`rayShade`**`( position, direction, shaderAccess )`

Trace a ray to the nearest surface and evaluate the basic surface parameters and any shaders on that surface. The LWShaderAccess structure passed (and owned) by the caller is filled in with the result and no more processing is done.

`position`
: The source of the ray in world coordinates.

`direction`
: A unit-length vector, the outgoing direction of the ray in world coordinates.

`shaderAccess`
: A pointer to an empty ShaderAccess structure that will be filled in by the function.

Tables

These pages list by category a total of 2743 symbols from the SDK headers. The symbols are cross-referenced to the header in which they're defined, and for the classes and globals, to the document page on which they're described. To make the pages a little faster, the header is only listed if it differs from that of the previous symbol in the list. If a header isn't shown on the same line as a particular symbol, look for it on a line above it.

File I/O

Availability LightWave 6.0
Component Layout, Modeler
Header lwio.h

This global provides functions for reading and writing data in files. The state structures returned by the `open` functions are the same as those passed to the handler `save` and `load` callbacks, making it possible for handlers to create and read custom preset files with the same code they use to write and read their settings in scene and object files. This is also an easy way for any kind of Layout plug-in to create block-structured, platform-independent configuration and data files.

Global Call

```
   LWFileIOFuncs *fiof;
   fiof = global( LWFILEIOFUNCS_GLOBAL, GFUSE_TRANSIENT );
```

The global function returns a pointer to an LWFileIOFuncs. The structure returned by the open functions is described on the file I/O page.

```
   typedef struct st_LWFileIOFuncs {
      LWSaveState * (*openSave) (const char *name, int ioMode);
      void          (*closeSave)(LWSaveState *save);
      LWLoadState * (*openLoad) (const char *name, int ioMode);
      void          (*closeLoad)(LWLoadState *load);
   } LWFileIOFuncs;
```

`sstate = openSave( name, iomode )`

Open a file for writing. The mode can be one of the following.

LWIO_ASCII
: Create a text file. Write operations will be line-buffered.

LWIO_BINARY
: Create a binary file. Block writes will always use 2-byte integers for block sizes.

LWIO_BINARY_IFF
: Create a binary file. Block sizes will be 4-byte integers for the first two nesting levels and 2-byte integers at deeper levels, corresponding to the chunk and subchunk scheme used in

LightWave object files.

closeSave(sstate)
: Close a file opened by `openSave`.

lstate = **openLoad**(name, iomode)
: Open a file for reading. The `iomodes` are the same as those for `openSave`.

closeLoad(lstate)
: Close a file opened by `openLoad`.

Example

This code fragment creates a text file and writes the contents of a structure. It uses the Observer structure and the `write_obs` function defined in the Example section of the file I/O page.

```
#include <lwserver.h>
#include <lwio.h>

LWFileIOFuncs *fiof;
LWSaveState *save;
Observer obs = {
   4.0f, "EDT", 2000, 4, 24, 2, 5, 30,
   37.75f, -122.55f,
   1, 40.0f, 30.0f, 100.0f,
   2000.0f
};

fiof = global( LWFILEIOFUNCS_GLOBAL, GFUSE_TRANSIENT );
if ( !fiof ) return AFUNC_BADGLOBAL;

if ( save = fiof->openSave( "testio.txt", LWIO_ASCII )) {
   write_obs( save, &obs );
   fiof->closeSave( save );
}

if ( save = fiof->openSave( "testio.bin", LWIO_BINARY )) {
   write_obs( save, &obs );
   fiof->closeSave( save );
}
```

Globals

Header	Symbolic Name	Document
lwdisplay.h	LWHOSTDISPLAYINFO_GLOBAL	display.html
lwdyna.h	LWDYNACONVERTFUNC_GLOBAL	dynaconv.html
	LWDYNAMONITORFUNCS_GLOBAL	modmon.html
	LWDYNAREQFUNCS_GLOBAL	modreq.html
lwenvel.h	LWCHANNELINFO_GLOBAL	chaninfo.html
	LWENVELOPEFUNCS_GLOBAL	anenvel.html
lwhandler.h	LWINSTUPDATE_GLOBAL	instupdt.html
lwhost.h	LWCOLORACTIVATEFUNC_GLOBAL	colorpik.html
	LWDIRINFOFUNC_GLOBAL	dirinfo.html
	LWFILEACTIVATEFUNC_GLOBAL	filereq2.html
	LWFILEREQFUNC_GLOBAL	filereq.html
	LWFILETYPEFUNC_GLOBAL	filetype.html
	LWLOCALEINFO_GLOBAL	locale.html
	LWMESSAGEFUNCS_GLOBAL	message.html
	LWPRODUCTINFO_GLOBAL	prodinfo.html
	LWSYSTEMID_GLOBAL	sysid.html
lwimage.h	LWIMAGELIST_GLOBAL	imglist.html
	LWIMAGEUTIL_GLOBAL	imgutil.html
lwio.h	LWFILEIOFUNCS_GLOBAL	fileio.html
lwmeshes.h	LWOBJECTFUNCS_GLOBAL	sceneobj.html
lwmodeler.h	LWFONTLISTFUNCS_GLOBAL	fontlist.html
	LWSTATEQUERYFUNCS_GLOBAL	modstate.html
lwmonitor.h	LWLMONFUNCS_GLOBAL	laymon.html
lwmtutil.h	LWMTUTILFUNCS_GLOBAL	mtutil.html
lwpanel.h	LWCONTEXTMENU_GLOBAL	conmenu.html
	LWPANELFUNCS_GLOBAL	panel.html
	LWRASTERFUNCS_GLOBAL	raster.html
lwpreview.h	LWPREVIEWFUNCS_GLOBAL	preview.html
lwprtcl.h	LWPSYSFUNCS_GLOBAL	particle.html
lwrender.h	LWBACKDROPINFO_GLOBAL	bkdpinfo.html
	LWBONEINFO_GLOBAL	boneinfo.html
	LWCAMERAINFO_GLOBAL	caminfo.html
	LWCOMPINFO_GLOBAL	compinfo.html
	LWFOGINFO_GLOBAL	foginfo.html
	LWGLOBALPOOL_GLOBAL	gmempool.html
	LWGLOBALPOOL_RENDER_GLOBAL	gmempool.html

	LWINTERFACEINFO_GLOBAL	intinfo.html
	LWITEMINFO_GLOBAL	iteminfo.html
	LWLIGHTINFO_GLOBAL	lightinf.html
	LWOBJECTINFO_GLOBAL	objinfo.html
	LWSCENEINFO_GLOBAL	sceneinf.html
	LWTIMEINFO_GLOBAL	timeinfo.html
	LWVIEWPORTINFO_GLOBAL	viewinfo.html
lwshelf.h	LWSHELFFUNCS_GLOBAL	shelf.html
lwsurf.h	LWSURFACEFUNCS_GLOBAL	surface.html
lwsurfed.h	LWSURFEDFUNCS_GLOBAL	surfed.html
lwtxtr.h	LWTEXTUREFUNCS_GLOBAL	txtrfunc.html
lwtxtred.h	LWTXTREDFUNCS_GLOBAL	txtred.html
lwvparm.h	LWVPARMFUNCS_GLOBAL	vparam.html
lwxpanel.h	LWXPANELFUNCS_GLOBAL	xpanel.html

Classes

Header	Symbolic Name	Document
lwanimlod.h	LWANIMLOADER_HCLASS LWANIMLOADER_ICLASS	animload.html
lwanimsav.h	LWANIMSAVER_HCLASS LWANIMSAVER_ICLASS	animsave.html
lwchannel.h	LWCHANNEL_HCLASS LWCHANNEL_ICLASS	channel.html
lwcmdseq.h	LWMODCOMMAND_CLASS	cs.html
lwcustobj.h	LWCUSTOMOBJ_HCLASS LWCUSTOMOBJ_ICLASS	custobj.html
lwdialog.h	LWFILEREQ_CLASS LWCOLORPICK_CLASS	freq.html colorpik.html
lwdisplce.h	LWDISPLACEMENT_HCLASS LWDISPLACEMENT_ICLASS	displace.html
lwenviron.h	LWENVIRONMENT_HCLASS LWENVIRONMENT_ICLASS	environ.html
lwfilter.h	LWIMAGEFILTER_HCLASS LWIMAGEFILTER_ICLASS LWPIXELFILTER_HCLASS LWPIXELFILTER_ICLASS	imgfilt.html pxlfilt.html
lwframbuf.h	LWFRAMEBUFFER_HCLASS LWFRAMEBUFFER_ICLASS	framebuf.html
lwgeneric.h	LWLAYOUTGENERIC_CLASS	generic.html
lwglobsrv.h	LWGLOBALSERVICE_CLASS	globserv.html
lwimageio.h	LWIMAGELOADER_CLASS LWIMAGESAVER_CLASS	imgload.html imgsave.html
lwlaytool.h	LWLAYOUTTOOL_CLASS	laytool.html
lwmaster.h	LWMASTER_HCLASS LWMASTER_ICLASS	master.html
lwmeshedt.h	LWMESHEDIT_CLASS	me.html
lwmodtool.h	LWMESHEDITTOOL_CLASS	metool.html
lwmotion.h	LWITEMMOTION_HCLASS LWITEMMOTION_ICLASS	itemmot.html
lwobjimp.h	LWOBJECTIMPORT_CLASS	objload.html

lwobjrep.h	LWOBJREPLACEMENT_HCLASS LWOBJREPLACEMENT_ICLASS	objrep.html
lwscenecv.h	LWSCENECONVERTER_CLASS	scenecvt.html
lwshader.h	LWSHADER_HCLASS LWSHADER_ICLASS	shader.html
lwtexture.h	LWTEXTURE_HCLASS LWTEXTURE_ICLASS	texture.html
lwvolume.h	LWVOLUMETRIC_HCLASS LWVOLUMETRIC_ICLASS	volume.html