plug

COLLABORATORS			
	TITLE : plug		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 29, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

plug

1.1 S0

```
LightWave Plug-in Architecture -- Stuart Ferguson 5/3/95

1  Plug-In Interface
    1.1  Server Identification
    1.2  Server Activation Function
    1.3  The Global Function
    1.4  Plug-in and Built-in Servers

2  Server Interface
    2.1  Plug-in Initialization and Cleanup
       (1)  Startup usage
       (2)  Shutdown usage
    2.2  Activation Function
       (3)  Activation function args
       (4)  ActivateFunc type
    2.3  Global Function
       (5)  GlobalFunc types
       (6)  GlobalFunc types
    2.4  The Global Server Class
       (7)  Global activation data
    2.5  External Function Entry Points
       (8)  XCALL Definitions
    2.6  Single-Service Plug-ins
       (9)  Activate usage
    2.7  Multiple-Service Plug-ins
       (10)  ServerRecord type

3  Common Globals
    3.1  Host Display Info
       (11)  Host Display Info declaration
    3.2  Monitor Objects
       (12)  Monitor types
       (13)  Monitor declarations

4  Example Plug-in Service
    4.1  String Transform Class
       (14)  Test types
       (15)  Test types
```

1.2 s1

There are two parts to the system-generic plug-in interface: the host side and the server side. The host is the application program which wants to load external code modules to perform some generic type of operation. Servers are imported routines (which can be loaded plug-ins or internal built-ins) which implement a specific instance of a generic type of service.

The host interface provides facilities to create server classes, register plug-in modules, and perform lazy loading and activation of registered servers. There is a fairly elaborate name and type mapping scheme which allows a great deal of flexibility in how modules are used, but which still provides a fairly simple interface for those who do not need the full facility.

The server interface provides an easy method to write programs that will operate as plug-ins. Different classes of plug-in services will require different host interfaces, but the loading and initialization part of the server interface is standard and works with the host portion of the system.

1.3 s1.1

The plug-in interface is designed to allow the host to have any number of servers loaded to perform as many different functions as the host wants to define. The servers in the system are referenced by a combination of class and name.

A Server Class is a string which determines the type of service which the server can perform. This might be strings like "TEXTURE" or "FileRequester". Many servers can have the same class, and all servers of the same class have the same host interface.

A Server Name is a string which refers to a specific server within a given class. This might be something like "FractalNoise3D" or "Default". The name must be unique among the servers of the same class.

The names for class and server identification should be byte strings containing characters only the the ASCII range 33-127. By convention these string contain no spaces and no characters outside 7-bit ASCII. Case is significant in distinguishing different classes and servers within classes.

1.4 s1.2

Every server has a single 'activation' function. This is the function which the host calls to access the service provided by the server. For some servers this one function will perform the whole action and for others this will only be a prelude to a sequence of actions. Servers which must remain loaded after they return from their activation function must be locked by the host while there are actions pending or they may be unloaded.

1.5 s1.3

The activation function for every server is called with a 'global' function pointer which provides access to the internal global state of the host system. The server calls the function with a string identifing the global data requested and a flag for how it will be used. The host can service this request, or the request can be passed on to global plug-in servers.

1.6 s1.4

Servers can be either plug-in or built-in. A plug-in server is implemented as a file containing code that can be loaded and unloaded as needed. A built-in server is implemented as a callback within the host itself. Having both allows the host to provide a standard set of servers which it handles the same way it handles plug-in servers, without having to unbundle their functionality in a way that can be replaced or used by other programs.

1.7 s2

A plug-in server is written like any ordinary C program, but instead of a single "main()" entry point, a server has a different primary entry point and several possible additional entry points. The server is linked with initializaion code (different from the normal shell or Workbench init) which places these interfaces where the host can access them.

There are two main types of plug-in modules: those providing a single server and those providing multiple servers. It is simple to have one server per module, but it can be more efficient and useful to define many servers with a single code file.

All servers require an activation function, and all plug-ins have the option of providing initialization and cleanup functions. The header for server types is ‘splug.h’.

1.8 s2.1

In both the single and multiple versions of the plug-in module, there are optional entry points which allow the module to initialize itself when it is first loaded and to clean itself up before being unloaded. If the plug-in code does not contain functions with these names, no attempt will be made to call them.

The Startup function, if present, will be called when the plug-in is first loaded into the host system. The return value is global data for the server which is passed to the Activate and Shutdown entry points as ‘ serverData’. A zero return value (null pointer) indicates failure, so even a plug-in with no data should return something.

```
    void *
  Startup (void)
```

If provided, the server's Shutdown function is called just before the server module is unloaded from the host. Any allocated server data should be freed at this point. Note that even though it is an error, this function may be called even when the server is locked, so correct cleanup should be done in this case as well.

(2) Shutdown usage

```
    void
  Shutdown (
    void                    *serverData)
```

1.9 s2.2

All servers have a single activation function which is the entry point for the host to get access to the service provided by the server. The activation function gets passed the version number for the service implementation, the 'global' pointer to access global host data, class-specific 'local' data, and private data maintained by the plug-in. The version number is application-defined, but typically it represents the revision of the interface that the host expects the server to use. Typically a server will not attempt to operate if the version number is greater than it expects. The ' serverData' is returned by the Startup entry point in a plug-in. The global function can be called to get global data from the host enviroment, and the contents of the 'local' pointer are defined by the type of service.

(3) Activation function args

```
  long             version,
  GlobalFunc      *global,
  void            *local,
  void            *serverData
```

The activation function returns an error code if the attempt to call failed because of some clash between the server and the host environment. If the server was able to process the request, even it failed to complete it, it should return AFUNC_OK. If the version number is not a value which the server can explicitly handle it should return AFUNC_BADVERSION. If there is some global data the server cannot get which it requires it should return AFUNC_BADGLOBAL. Severe problems with the contents of the local data, such as some necessary pointer in the local data being null, may be reported by returning AFUNC_BADLOCAL. Any other errors from the server (running out of memory, bad filenames, user aborts, etc.) must be provided for by the specific plug-in protocol.

(4) ActivateFunc type

```
  typedef int     ActivateFunc (<Activation function args>);

  #define AFUNC_OK                0
```

```
#define AFUNC_BADVERSION        1
#define AFUNC_BADGLOBAL         2
#define AFUNC_BADLOCAL          3
```

1.10 s2.3

The global function passed by the host to the server is a special function which returns the pointer to some global data given by a string ID. These data blocks will often contain function pointers, but can be anything.

(5) GlobalFunc types

```
typedef void *          GlobalFunc (const char *, int);
. . .
```

When a server calls the global function, it passes a string which identifies the global data required and a code for the way the data will be used. If the data pointer is not available, null is returned. The ID's that will be recognized depends on the host, on the available global plug-ins and perhaps on the server class.

The use code depends on how the result of the call will be used. If the returned pointer will only be used for the course of the activation function itself, the TRANSIENT code should be used. If the data will be used after the activation function returns, such as in a server that requires locking, the ACQUIRE code should be used. In this case there must be a matching RELEASE call made when the data pointer is no longer required. RELEASE calls need only be made for ACQUIRE calls which returned a non-null pointer. The return value from a release mode global data call is undefined.

(6) GlobalFunc types

```
. . .
#define GFUSE_TRANSIENT         0
#define GFUSE_ACQUIRE           1
#define GFUSE_RELEASE           2
```

1.11 s2.4

The server class given by the name "Global" is special in that it allows multiple plug-in servers to share common data or routines. In fact, the members of the Gobal class are extensions to the set of ID strings that can be passed to the "global" function.

When a server calls the global function with an ID string, the host can service the request itself or has the option of pass unrecognized ID's to Global class servers of the same name. For example, if the ID is "Mambo Functions," the host may recognize this itself and return a pointer value. If it does not recognize it, it may attempt to activate a server of class "Global" with name "Mambo Functions." If such a server exists, it may be locked or unlocked, depending on the

use type of the global request, and it will be called to get the value of the global pointer for the orignal requester.

The activation function of a Global server is called with a GlobalService structure which will be initialized with the ID string for the request. The server must fill in the data pointer with a value which will be returned to the client, which may be null if the server wishes to deny the request. The string is passed as data so that the same activation function may be used for multiple servers.

(7) Global activation data

```
typedef struct st_GlobalService {
  const char      *id;
  void            *data;
} GlobalService;
```

1.12 s2.5

Functions in the plug-in get called directly by the host, and this is a funky thing in some systems since they are different environments. The XCALL_ and XCALL_INIT macros take care of everything for all different systems and compilers, so these can be used to make multi-platform servers from a single source code.

XCALL_ is used on the return type, e.g. XCALL_(int) for an external entry point returning an int. XCALL_INIT is used as the first statement of the function. Both must be used for full compatibility, but XCALL_INIT is only non-null for Manx small-code modules.

(8) XCALL Definitions

```
<XCALL_ and XCALL_INIT system-specific definition>
```

The activation function as well as any function pointers returned from the activation function need the XCALL treatment. Startup and Shutdown do not.

1.13 s2.6

A single-service plug-in is a C program with an entry point for the activation callback and global symbols for the class and name of the server. There are also optional entry points for initialization and cleanup.

This plug-in must contain a global character string with the name 'ServerClass'. This string defines the class of this server and the server will not be loaded if this string does not match the service type string requested by the host.

It must also contain a global character string called 'ServerName' which holds the name for this specific server.

The activation function must be called Activate, which takes the arguments as described above.

(9) Activate usage

```
  XCALL_(int)
Activate (<Activation function args>)
```

1.14 s2.7

A multiple-service plug-in is a C program which defines multiple servers through a standard set of global symbols. In particular, a multiple server module must contain a global array with the name 'ServerDesc' composed of elements of the ServerRecord type. The last record in the array must have a null class name pointer.

(10) ServerRecord type

```
typedef struct st_ServerRecord {
  const char      *class;
  const char      *name;
  ActivateFunc    *activate;
} ServerRecord;
```

The plug-in module may also have Startup and Shutdown entry points, and all the activate functions in the plug-in will get the same serverData as returned from the Startup function. The assumption is that the servers all share a module for some logical reason, so the sharing of global data is not unreasonable.

1.15 s3

There are a few global data types which are so basic that they are the same across plug-in hosts or are used in a wide range of plug-in interfaces.

1.16 s3.1

A plug-in may need to open windows to get user input, and since they run in the host's context, they will need to do this using the host's display information. This info, which can be normally accessed with the "Host Display Info" ID string, contains information about the windows and display context used by the host. If this ID yeilds a null pointer, the server is probably running in a batch mode and has no display context.

The fields of the HostDisplayInfo structure vary from system to system. On the Amiga, the screen pointer is provided for custom screens and is null for Workbench applications. The window pointer is the main application window or null if there is none. On X systems, the window session handle is passed, as well as the ID of the main application window, if any. On Win32 systems, the application instance and main window are provided.

(11) Host Display Info declaration

```
typedef struct st_HostDisplayInfo {
    #ifdef _AMIGA
  struct Screen   *screen;
  struct Window   *window;
    #endif

    #ifdef _XGL
  Display         *xsys;
  Window           window;
    #endif

    #ifdef _WIN32
  HANDLE           instance;
  HWND             window;
    #endif
} HostDisplayInfo;
```

This structure is defined in the 'hdisp.h' header file.

1.17 s3.2

Monitors are simple data structures defining an interface which the server can use to give feedback to the host on its progress in performing some task. They are sometimes passed servers to give feedback on the progress of the particular operation, and can sometimes be accessed from within a server that wants to show its progress on a slow operation using the host's normal feedback display.

A Monitor consists of some generic data and three functions: init, step and done. The 'init' function is called first with the number of steps in the process to be monitored, which is computed by the server. As the task is processed, the 'step' function is called with the number of steps just completed (often one). These step increments should eventually add up to the total number and then the 'done' function is called, but 'done' may be called early if there was a problem or the process was aborted. The 'step' function will return one if the user requested an abort and zero otherwise.

(12) Monitor types

```
typedef struct st_Monitor {
  void             *data;
  void            (*init) (void *, unsigned int);
  int             (*step) (void *, unsigned int);
  void            (*done) (void *);
```

```
  } Monitor;
```

The server is masked from any errors in the monitor that may occur on the host side of the interface. If there is a problem with putting up a monitor, the functions will still return normally, since the monitor is for user feedback and is not that critical.

There are some macros provided to call a monitor which will do nothing if the monitor pointer is null. MON_INCR is used for step sizes greater than one and MON_STEP is used for step sizes exactly one.

(13) Monitor declarations

```
#define MON_INIT(mon,count)     if (mon) (*mon->init) (mon->data,
   count)
#define MON_INCR(mon,d)         (mon ? (*mon->step) (mon->data, d) :
   0)
#define MON_STEP(mon)           MON_INCR (mon, 1)
#define MON_DONE(mon)           if (mon) (*mon->done) (mon->data)
```

These structures and macros are described in the 'moni.h' header file.

1.18 s4

This describes a hypothetical server class and creates some samples of plug-in modules using it. This serves as a testbed for third parties to create test plug-ins, so it should have some general capability.

1.19 s4.1

This server class will perform manipulations on character strings, like reverse them, capitalize them, etc. This class will be "StringXfrm".

A new server class is completely defined by the semantics of the activation function for the class. The activation function takes a pointer argument from the host, 'local' which is a reference to data for the particular service the host needs performed. It also gets a 'global' function pointer which will return global data as needed by the server.

The 'local' pointer will point to a StringLocal structure which holds the data for the current operation. This is a null-terminated string and the length of the string buffer, plus a temporary scratch buffer and its length. The server will overwrite 'buf' with the result, and will set the 'overflow' status flag if the buffers were too short.

(14) Test types

```
  typedef struct st_StringLocal {
    char            *buf;
    char            *tmpBuf;
    int              len, tmpLen;
    int              overflow;
  } StringLocal;
  . . .
```

For the string transform class of server, the global function can return a ‘progress’ function which can be called by the server to give the user feedback about its progress. This is returned using a string ID of "Progress Function."

(15) Test types

```
  . . .
  typedef void            StringProgress (void);
```

We’ll stick these definitions into the ‘t_plug.h’ header file for test modules to use.

1.20 s4.2

The functions to do string transformations are all the same. They all get the same arguments as defined by the format of the activation function. The local pointer is specific to the string transform class. There is no ‘ serverData’ for any of the transforms since there is no Startup function.

(16) String transform arguments

```
  long                     version,
  GlobalFunc              *global,
  StringLocal             *local,
  void                    *serverData
```

String activation functions may return with an error code if the version number is wrong or if the global progress function is not available.

1.21 s4.2.1

The length operation gets the length of the string and prints that as a number into the string buffer.

(17) Length function body

```
{
  if (version != 1)
    return AFUNC_BADVERSION;

  if (local->len < 10)
    local->overflow = 1;
  else
    sprintf (local->buf, "%ld", strlen (local->buf));

  return AFUNC_OK;
}
```

1.22 s4.2.2

The reverse operation copies the characters from the main buffer to the temp buffer in reverse order and then copies them back. This could use a swap operation to reverse them in place, but this method demonstrates using the temp buffer and returning an overflow if the temp buffer is too small. This also calls the progress function as it swaps.

(18) Reverse function body

```
{
  StringProgress          *progress;
  int                      len, i;

  if (version != 1)
    return AFUNC_BADVERSION;

  progress = (*global) ("Progress Function", GFUSE_TRANSIENT);
  if (!progress)
    return AFUNC_BADGLOBAL;

  len = strlen (local->buf);
  if (local->tmpLen <= len) {
    local->overflow = 1;
    return AFUNC_OK;
  }

  for (i = 0; i < len; i++) {
    local->tmpBuf[i] = local->buf[len - i - 1];
    (*progress) ();
  }
  local->tmpBuf[len] = 0;

  strcpy (local->buf, local->tmpBuf);
  return AFUNC_OK;
}
```

1.23 s4.2.3

This just passes through the string and converts each letter to
uppercase, calling the progress function as it goes. This will also
use the global empty string if there are no characters passed.

    (19) Capitalize function body

```
  {
    StringProgress          *progress;
    const char              *empty;
    char                    *c;

    if (version != 1)
      return AFUNC_BADVERSION;

    progress = (*global) ("Progress Function", GFUSE_TRANSIENT);
    empty = (*global) ("EmptyStringText", GFUSE_TRANSIENT);
    if (!progress || !empty)
      return AFUNC_BADGLOBAL;

    if (local->buf[0]) {
      for (c = local->buf; *c; c++) {
        (*progress) ();
        if (*c >= 'a' && *c <= 'z')
          *c = *c - 'a' + 'A';
      }
    } else
      strncpy (local->buf, empty, local->len - 1);

    return AFUNC_OK;
  }
```

1.24 s4.2.4

This doubles each character in the string by copying the buffer to the
temp buffer and moving twice as many characters back into the buffer
from there. This will fail if the buffers are not big enough.

    (20) Double function body

```
  {
    StringProgress          *progress;
    int                      len, i;

    if (version != 1)
      return AFUNC_BADVERSION;

    progress = (*global) ("Progress Function", GFUSE_TRANSIENT);
    if (!progress)
      return AFUNC_BADGLOBAL;

    len = strlen (local->buf);
    if (local->tmpLen < len || local->len - 1 < len * 2) {
      local->overflow = 1;
      return AFUNC_OK;
```

```
    }

    strcpy (local->tmpBuf, local->buf);
    for (i = 0; i < len; i++) {
      local->buf[i * 2]     = local->tmpBuf[i];
      local->buf[i * 2 + 1] = local->tmpBuf[i];
      (*progress) ();
    }
    local->buf[len * 2] = 0;

    return AFUNC_OK;
  }
```

1.25 s4.3

A plug-in module is really a wrapper around the activation function, and can be implemented several ways. They can be single-service plug-ins, multiple-service plug-ins, or built-in. This test includes one of each.

1.26 s4.3.1

The reverse operation is implemented as a single-service plug-in, so there is one global class and server name. The activation function is called ‘Activate’ (which it must be).

The C program module itself includes the headers for the test system and server-side plug-ins. The source file is ‘tp_rev.c’.

(21) Test Reverse plug-in program

```
  #include <splug.h>
  #include "t_plug.h"
  #include <string.h>

  char            ServerClass[] = "StringXfrm";
  char            ServerName[]  = "REVERSE";

    XCALL_(int)
  Activate (<String transform arguments>)
  {
    XCALL_INIT;
    <Reverse function body>
  }
```

1.27 s4.3.2

The capitalize and double operations are implemented as one multiple-service plug-in with two servers. The activation function entry points can have any name and are not exported symbols. They are associated with their server name in the exported array of ServerRecords which has the required name 'ServerDesc'. The source file for this is 'tp_cpdb.c'.

(22) Test Caps and Double plug-in program

```
#include <splug.h>
#include "t_plug.h"
#include <string.h>

  static XCALL_(int)
Capitalize (<String transform arguments>)
{
  XCALL_INIT;
  <Capitalize function body>
}

  static XCALL_(int)
Double (<String transform arguments>)
{
  XCALL_INIT;
  <Double function body>
}

const char              class[] = "StringXfrm";
ServerRecord            ServerDesc[] = {
  { class, "CAPS",        Capitalize },
  { class, "DOUBLE",      Double },
  { NULL }
};
```

1.28 s4.3.3

The length operation will be implemented as a built-in. As result, all we need is a local function entry point of any name in the host program.

(23) Test host utilities

```
  static int
ActLength (<String transform arguments>)
{
  <Length function body>
}
. . .
```

1.29 s4.3.4

This test program includes a global server which just returns a string for clients to use when passed an empty string. This global's activation function gets a GlobalService request block with the 'id' set to the requested string. In this case, this MUST be the same as the server name. The function returns a global pointer in the 'data' field.

(24) Test Global Server plug-in

```
#include <splug.h>
#include <string.h>

char            ServerClass[] = "Global";
char            ServerName[]  = "EmptyStringText";
char            result[] = "** Empty String **";

  XCALL_(int)
Activate (
  long             version,
  GlobalFunc      *global,
  GlobalService   *local,
  void            *data)
{
  XCALL_INIT;
  if (strcmp (local->id, ServerName) != 0)
    return AFUNC_BADLOCAL;

  local->data = result;
  return AFUNC_OK;
}
```

1.30 s5

Methods for creating plug-ins have been developed for each of the target platforms. Versions of a plug-in can be created for the different systems from a single source code with different linking instructions. Each case that follows includes an implicit makefile rule to create a ".p" plug-in module from an object file. The macro SLIB stands for the directory where the startup code and server libraries are located. SINC is the include directory and OTHER_LIBS would be any other libraries need by the module.

The final section shows how to add your plug-in to the LightWave host.

1.31 s5.1

Linking under SAS/C on the Amiga requires replacing the normal startup code with plug-in startup code, 'serv_s.o'. This can be done by using the "startup" option when using "sc link" or by placing 'serv_s.o' first in the "FROM" list when using slink. Modules must also be linked with the server library. Object modules should be built without stack checking.

(25) Makefile examples

```
.o.p:
  sc link $(CFLAGS) startup=$(SLIB)serv_s.o $*.o\
   $(SLIB)server.lib $(OTHER_LIBS) pname=$@
. . .
```

The math options to the compiler must be chosen so that doubles are 64-bit IEEE values. There is currently some difficulty using the "math=68881" option, however, having to do with the startup code in serv_s.a.

1.32 s5.2

Linking under the Manx compiler on the Amiga requires using the plug-in startup code 'serv_m.o', which must be placed first in the list of objects passed to ln. The linker will warn about ".begin" and "_geta4" overriding library symbols, which is correct behavior in this case. They should also be linked with the "server_m" library to get the Manx server library.

(26) Makefile examples

```
. . .
.o.p:
  ln -o $@ $(SLIB)serv_m.o $*.o -lserver_m $(OTHER_LIBS)
. . .
```

Manx modules must use 32-bit ints and IEEE format floating point values.

1.33 s5.3

Plug-in modules under Windows are just DLLs created by linking with 'serv_w.obj' and 'server.lib'. There is no need to create a ".lib" or ".exp" file for the DLL, but the ".def" file should contain an export statement for the global address '_mod_descrip'. A usable default def file is provided as "serv.def" in the main include directory. There is no DLL entry point function.

(27) Makefile examples

```
. . .
.obj.p:
  link32 -dll -out:$@ -def:$(SINC)serv.def $*.obj\
```

```
     $(SLIB)serv_w.obj server.lib $(OTHER_LIBS)
  . . .
```

Structure alignment may vary among different compilers and can cause problems when trying to communicate between the host and a plug-in DLL. The LightWave host is compiled using Microsoft alignment rules. Here's an excerpt on structure alignment from the MS Visual C documentation:

"Applications should generally align structure members at addresses that are 'natural' for the data type and the processor involved. For example, a 4-byte data member should have an address that is a multiple of four.

"This principle is especially important when you write code for porting to multiple processors. A misaligned 4-byte data member, which is on an address that is not a multiple of four, causes a performance penalty with an 80386 processor and a hardware exception with a MIPS® RISC processor. In the latter case, although the system handles the exception, the performance penalty is significantly greater. The following guidelines ensure proper alignment for processors targeted by Win32:

```
    (28)  Alignment Table

  "Type                   Alignment
   ----                   ---------
   char                   Align on byte boundaries
   short (16-bit)         Align on even byte boundaries
   int and long (32-bit)  Align on 32-bit boundaries
   float                  Align on 32-bit boundaries
   double                 Align on 64-bit boundaries
   structures             Largest alignment requirement of any member
   unions                 Alignment requirement of the first member
```

"The compiler automatically aligns data in accordance with these requirements, inserting padding in structures up to the limit (default pack size) specified by the /Zp option or #pragma pack. For example, /Zp2 permits up to 1 byte of padding, /Zp4 permits up to 3 bytes of padding, and so on. The default pack size for Windows 3.x is 2, whereas the default for Win32 is 8."

1.34 s5.4

Plug-in modules under IRIX are shared object modules linked with 'serv_u.o' and 'libserver.lib'. The DSO should export the "_mod_descrip" symbol and use "serv_u" as startup code.

The link line should include any other libraries that the plug-in would need as a stand-alone program. Since most libs on the SGI are themselves DSOs, this adds very little to the size of the plug-in and adds no extra runtime overhead.

```
    (29)  Makefile examples
  . . .
```

```
  .o.p:
    ld -shared -exported_symbol _mod_descrip -L$(SLIB)\
     $(SLIB)serv_u.o $*.o -o $@ -lserver $(OTHER_LIBS)
```

Normally the plug-in DSO's are loaded in a way that forces resolution of all symbols. This allows the host program to report undefined symbols to the plug-in developer. If the name of the module includes the substring "__lazy" (lowercase), however, then the lazy evaluation mode is used, allowing the module to contain undefined symbols as long as they are not referenced by any executed code. This is normally not needed unless you are using code from an external vendor which includes undefined but unused symbols, like the HIIP library from Elastic Reality.

1.35 s5.5

LightWave and Modeler read the names of servers from their startup configuration files. This method is much faster than scanning a directory path and allows for some user customization of plug-in names (such as national localization). It does require that the config file be accurate, since the host will blindly attempt to use servers that may not exist. This is non-fatal but may be disconcerting to the user.

For Modeler, for example, the config file on the Amiga is "MOD-config", on the SGI is ".lwmrc" and on Windows is "LWM.CFG". This file can contain any number of lines of the following form:

```
    (30)  Config file examples

  Plugin <class> <name> <module> <user name>
  . . .
```

Each line describes a single server given by Class and Name. The module is the plug-in file containing the server and the user name is the string to display on the interface for describing the server's function. Class, name and module are delimited by spaces, and the user name is the rest of the line. Here are some examples (lines wrap for readability -- each statement has to be a single line).

```
    (31)  Config file examples
  . . .
  Plugin CommandSequence Demo_AllBGLayers layerset.p Include Background
  Plugin CommandSequence Demo_NextEmptyLayer layerset.p Next Empty
  Plugin MeshDataEdit Demo_MakeSpikey z:lw/plugin/spikey.p Spikey
     Subdivide
  Plugin ImageLoader PDQ_Targa pdq/targa.lwp Truevision Targa Image
```