WebMedia Programming Library

Plugin Programmers Reference

Version 0.1 – 07 May 2001

5284 South Commerce Drive, Suite C134
Salt Lake City, Utah 84107
Phone: +1.801.268.3088
Web: http://embedded.censoft.com

Plugin Overview

The concept of a "plugin" goes back many years. Plugins were found in early versions of Adobe® Photoshop®, and were used to add additional image processing features and capabilities to this popular application. By purchasing additional plugin libraries and copying them into the plugin directory, the program was automatically enhanced the next time it was started. Today, hundreds of applications have adopted the concept of the plugin, and in some cases large, successful companies have been created that only supply plugins for popular applications.

A plugin is really nothing more than an executable library file that exports a known set of functions. Each function is called at a specific time (load, unload, or by the parent application), follows a set of straight-forward rules, and has a predefined set of parameters.

Most operating systems provide library routines to load an executable library (plugin) into memory, locate an exported function by name, recover a pointer to the exported function, and ultimately unload and delete the library. For Linux, these library functions are **dlopen()**, **dlsym()**, and **dlclose()**. It's not possible to completely understand the WebMedia plugin architecture without knowledge of these critical system-level functions.

In operation, plugins are copied into a know directory when the plugin library or application is installed. Later, when the application is launched, this directory is scanned for plugin files, usually based on some kind of naming convention. When a plugin candidate is located, it is loaded into memory and then attempts are made by the application to locate the entire set of predefined exported functions (entry points). As each entry point is located, a pointer to that external function is added to a table which is used to call the plugin function later. If a required entry point is not found, the plugin candidate is removed from memory and the process is repeated on subsequent plugin candidates. This process continues until all possible plugin candidates have been loaded and tested.

Plugins offer tremendous flexibility when designing a system, especially when considering the issues of production testing, subscriber services, pay-per-view, pay-per-use, etc. Some examples are in order:

Manufacturing Test - During mass production, STB systems need to be tested before they are boxed and shipped to make sure they are fully functional. The WebMedia plugin architecture offers an easy way to implement this. For each element that must be tested (video in, video out, tuner, DVD, MPEG decoder, Ethernet, modem, etc.) an individual plugin is created that can prompt the technician, perform tests, and log the results. A "master" plugin is also created that is smart enough to invoke, in sequence, each of the test plugins. All the technician need do to validate a piece of hardware for shipping is to invoke one

WebMedia command; this command will then step though each individual test, informing the technician of what they should see, hear, or confirm. As the tests are completed, a pass/fail log has been generated that can be printed out and shipped. This same methodology would help manufactures support a wide variety of configurations. By combining various WebMedia plugins together, unique hardware configurations can be efficiently tested simply by selecting a group of plugins that match each new hardware model.

Real-time Stock Quotes - For many people, tracking their stocks is a daily activity. Through the use of WebMedia plugins, it would be possible to sell a subscription to real-time stock quotes. The plugin can be delivered electronically, and once it is copied into the system, it can enable the display of quotes, help individuals manage their portfolio, watch for dramatic changes in share price, etc. The plugin could be designed to expire after a day, week, or month, when it simply removes itself from the system. It could also be designed to self-renew, so that as long as the subscription service is paid for, the plugin is resident and operational.

Pay-Per-View DVD – Using plugins, DVD discs could be borrowed, traded, loaned, and distributed freely. When it came time to actually view the DVD, the user would place the order and a DVD playback plugin would be delivered to their system. The plugin could be designed to allow one single play, any number of plays during a time frame (say a weekend), or for any block of time. Once the DVD playback timeframe has expired, the plugin removes itself from the system.

Plugin designers have wide latitude in how they design their plugins and the feature sets they can support. When plugins are designed in concert with back-end database services and network connectivity, extremely flexible architectures can be conceived of and implemented in a short period of time.

WebMedia Plugin Basics

Overview

This section describes some basic information concerning plugins engineered to work under WebMedia. It is important to understand each topic below before developing WebMedia plugins; taking the time to become familiar with these topics will greatly simplify development later, in addition to making the examples at the end of this document more clear.

Plugin Types

There are two general types of plugins: *system* plugins and *control* plugins. The difference is small but important: system plugins are plugins that contain all the functionality necessary to implement proper plugin operation. Control plugins are plugins that contain minimal functionality and are used to control and manage externally forked processes. A few examples are in order:

To implement composite video management, all that is needed is a system plugin. The plugin itself contains all the calls to the device driver or video middleware needed, and the system state, as well as the plugin thread, expire once the command is completed. System plugins are executed as a thread under WebMedia, and after they have performed their required operations, their thread is terminated as the plugin exits. Examples of system plugins would be plugins that set state, change tuner channels, select a video source, or reset hardware.

A control plugin would be needed to control some system component, such as an internal DVD player. Before the DVD plugin is used, a DVD player thread must be forked under the Linux operating system. Communication between the plugin and the DVD player thread is done through shared memory. In this case, commands to the DVD player thread such as stop, play, and seek are simply issued by the plugin to the DVD player thread through the shared memory. Once the DVD play command is delivered to the DVD player thread, it then manages the continuous playback of the DVD program while the WebMedia plugin thread that issued the command terminates. Examples of control plugins would be plugins designed to playback a DVD movie, stream video from a server, or playback local MP3 files.

Before developing a WebMedia plugin, the developer must determine if there needs to be a separate thread to manage the on-going execution of the WebMedia command. If there is none, then a system plugin can be developed. Otherwise, a control plugin would be developed in conjunction with a separate application, and the two would communicate through shared memory.

Naming Conventions

WebMedia plugins are identified by a three-letter extension: **.wmp**. During load, only filenames with this extension are considered plugin candidates. This can be useful in managing a plugin set: if the user desires that one or more plugins should not be loaded (say for debugging purposes), simply changing the extension to something other than **.wmp** will prevent WebMedia from detecting the plugin. There would be no need to delete the file or copy it to another directory.

Plugin Directory

Once a plugin has been developed, it must be copied into a known directory, where all WebMedia plugins reside. This directory location is known to the WebMedia application and is the only directory scanned for plugin candidates. At the present time, the plugin directory location for the Cygnus platform is **/bin**.

Compiling Plugins

Because plugins are simply executable libraries, compiling them under gcc (the compiler/linker supplied with most distributions of Linux) is generally straight-forward. Any number of c++ files may be included in the compilation, and only a few additional programming rules must be observed.

All plugin entry points must be included for the plugin to be considered valid. Although all entry point functions must be included, they may be distributed across separate source files or included in a single file – whatever is easiest for the plugin developer. Each plugin entry-point function must be identified as **extern "C";** this directs the compiler to mark each function as "linkable" and specifies the function's calling, name mangling, and stack requirements to the compiler.

When compiling the plugin, be sure to include the **–shared** command line switch. This produces an object file that can be "linked to" by other programs at runtime, and also enables the use of shared memory when developing control plugins. The following example shows a typical compiler command line:

```
%gcc –shared –o foobar.wmp file1.cpp file2.cpp file3.cpp
```

The example command line above will create a plugin named *foobar.wmp* and is built from three distinct source files.

Recursive Plugins

The WebMedia plugin architecture is completely re-entrant, and plugins can call and invoke other plugins. There is no theoretical limit on the number of recursions available. However, at several hundred levels of recursion, physical memory could become scarce and plugin execution speed could suffer.

To create a command for another plugin, enter the desired command text into a buffer and then call the **dispatcher()** function, passing a pointer to the initialized text buffer. This will send the new plugin command to the appropriate plugin command handler and return with the success/error code. The following code example shows this process:

```
char buf[256];
int result;

sprintf(buf,"stream play %s",streamname);
result = dispatcher(buf);
if(result != 0)
      printf("Error #%d\n",result);
```

In the example above, the “stream play filename” command is written into “buf”, and the name of the stream is recovered from a separate variable. The plugin dispatcher function is called, and once the specified stream begins to play a response is returned, which is checked for errors.

Plugin Keyword

Each plugin must be assigned a unique keyword, which becomes part of the command syntax. This is generally easy to do as each plugin is designed to control a unique piece of the system. The plugin keyword is passed on the command line immediately following invocation of the WebMedia utility, as in the example below:

```
% wmcmd tuner channel up
```

In the example above, the name of the WebMedia utility is “wmcmd”, and the name of the keyword is “tuner”. Following the keyword is one or more parameters which represent the actual command to be invoked (channel, in the case above), and other parameters as required (up, in the case above).

Plugin Help

Each plugin is responsible for supplying it’s own help support. This is done by implementing the **pluginHelp()** entry point function. When this entry point is called, the plugin should output a list of single-line help entries to **stdout**. Each entry is used to describe one supported command. For formatting purposes, each line should start with three space codes.

There are two types of help requests: a complete help listing or help for just one plugin. Here are two examples:

```
% wmcmd help
% wmcmd tuner help
```

In the first example, a complete listing for all plugins is displayed. In the second example, only help for the tuner plugin is displayed. Both help “styles” are implemented outside of the plugins – as long as each plugin supports single-line help entries in response to a **pluginHelp()** function call, everything will work and display properly.

Plugin Entry Points

Overview This section describes in detail the plugin function entry points. This includes what parameters are passed, result codes, and general programming information.

Each plugin must have all entry points specified. If the developer does not need the specified functionality supported by the entry point function, the entry point must still be included and a valid success code returned. As an example, in many cases, the **pluginStartup()** and **pluginShutdown()** entry point functions are not needed. In these cases, be sure to include the function and have it simply return '0' to indicate success.

When developing a plugin, be sure to prefix each of these functions with an **extern "C"** declaration.

startup entry point

```
int pluginStartup(void)
```

This function is used to do any one-time plugin initialization. It is called only once, just after the plugin is loaded and before any other plugin functions are called. This function's primary purpose is to allocate any memory and do any one-time initialization required for proper plugin operation.

No parameters are passed.

This function should return '0' on success. Any other value will be seen by WebMedia as an error.

shutdown entry point

```
int pluginShutdown(void)
```

This function is used to do any cleanup before the plugin is unloaded. It is called only once, just before the plugin itself is unloaded and all memory is returned to the system. This function's primary purpose is to "undo" any allocations or one-time initialization that was done by pluginStartup().

No parameters are passed.

This function should return '0' on success. Any other value will be seen by WebMedia as an error.

name entry point

```
int pluginName(char *result)
```

This function is used to recover the descriptive name for the plugin. This name is used when listing what plugins are available, and can also be used to help expand on what a plugin does. For example, the cryptic name "cvid" may be more convenient to type, but returning something like "composite video source manager" is far more descriptive.

The "result" parameter is a pointer to a buffer used to store the descriptive text.

This function should return ‘0’ on success. Any other value will be seen by WebMedia as an error.

version entry point

```
int pluginVersion(char *result)
```

This function is used to recover the version number for the plugin. Any versioning system can be used as the version result is returned as a string. Just like the pluginName() function, this function is designed to help users determine what version of a specific plugin is being used.

The versioning system used for current WebMedia plugins is x.y.z, where x represents the main program version, y represents minor changes to the main version, and z represents incremental bug fixes and small updates.

The “result” parameter is a pointer to a buffer used to store the version string.

This function should return ‘0’ on success. Any other value will be seen by WebMedia as an error.

keyword entry point

```
int pluginKeyword(char *result)
```

This function is used to recover the main keyword used to identify all commands for this plugin. The keyword is used to identify all commands managed by a specific plugin. As an example, consider the following WebMedia commands:

```
% wmcmd tuner source
% wmcmd tuner channel up
% wmcmd tuner channel down
% wmcmd dvd source
```

In the examples above, the first word, wmcmd, is the actual utility that is being invoked, and all the other words on the command line are passed in as parameters. WebMedia uses the second word (the *first* parameter) to identify which plugin should handle the command. So in the commands above, “tuner” is the keyword for the plugin that handles any tuner interaction, and “dvd” is the keyword for the plugin that manages the DVD player.

Each plugin must have a unique keyword – one that is different from all other keywords in plugins that are loaded. If this is not the case – if two plugins use the same keyword - all commands for a specific keyword will be directed to the first plugin loaded with the keyword in question.

The “result” parameter is a pointer to a buffer used to store the plugin’s keyword.

This function should return ‘0’ on success. Any other value will be seen by WebMedia as an error.

help

```
int pluginHelp(void)
```

entry point This function will display the plugin's help listing – single text lines printed to **stdout** and used to display each command available for this plugin. For proper formatting, three spaces should be included at the beginning of each line

No parameters are passed.

This function should return '0' on success. Any other value will be seen by WebMedia as an error.

command entry point `int pluginCommand(int argc, char *argv[])`

This function is used to issue the actual commands to the plugin, and is the heart of the WebMedia plugin architecture.

WebMedia plugin commands take the following form:

```
% wmcmd tuner channel up
```

The actual utility name is wmcmd, followed by the keyword assigned to each plugin, which in the example above is "tuner". Following the keyword, any number of parameters may be used to specify the actual command.

In operation, WebMedia will parse the command line information passed to it, extracting the keyword and then comparing the keyword against the plugins that have been loaded. If no match is found, an error is returned to the user. If a match is found, the parameters are extracted and the plugin is called through the **pluginCommand()** function. Pointers to the text parameters are passed, along with the number of parameters.

In the example above, the **pluginCommand()** entry point would be called with the argc value set to two. Referencing the pointer at argv[0] would return the text string "channel" and referencing the pointer at argv[1] would return the text string "up". This is the standard argc/argv parameter passing methodology. It is important to note that the wmcmd parameter and the plugin keyword have been stripped out, leaving only the parameters to be parsed.

Because only active parameters are passed through to the plugin, some fairly efficient parsing mechanisms can be developed to quickly determine which plugin command is to be executed. In the examples section you will see a table-driven approach that is efficient and easy to understand.

The "argc" parameter specifies the number of valid parameters passed through the argv array. The "argv" parameter is an array of pointers to ASCIIZ strings that are, in ascending order, the passed in command line parameters.

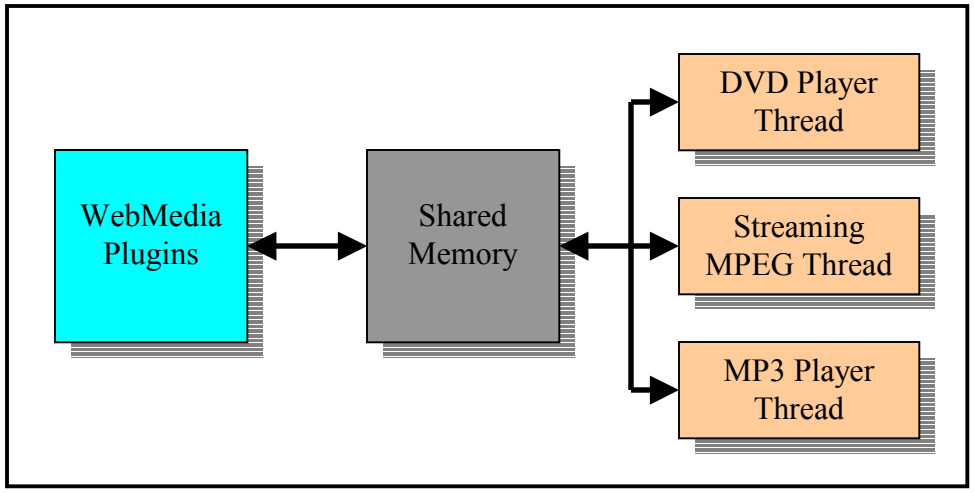
This function should return '0' on success. Any other value will be seen by WebMedia as an error.

Shared Memory

Introduction When a control plugin is needed to communicate with a separately forked process, one of the most effective ways to implement this communication path is through the use of shared memory. By implementing a command issue and acknowledgement sequence in software, any number of forked processes can wait for WebMedia plugins to issue command sequences.

Shared memory is a block of system memory, usually sized in a multiple of 4096 byte blocks, that can be simultaneously accessed by any number of threads or processes. The memory is referenced through a "handle" that is made available to all processes that need to simultaneously access the shared memory. Most operating systems that support preemptive multitasking have libraries that support the allocation, access and de-allocation of shared memory. Under Linux, the functions used by WebMedia are **shmctl(), shmget(), shmat(), and shmdt()**. It's not possible to completely understand the WebMedia plugin architecture without knowledge of these critical system-level functions.

WebMedia has implemented a producer/consumer approach that makes it easy for multiple forked processes (consumers) to wait for commands being issued from WebMedia plugins (producers). Generally, there is one producer of commands (WebMedia) and one-to-many consumers of the commands (each forked process). This is shown in the example below:



Each application that is forked has a companion control plugin implemented under WebMedia. All control commands flow through shared memory. The handshaking implementation is such that when a command is loaded into shared memory, only the proper consumer thread receives the command and handshakes it through. The other consumer threads, as soon as they know the command is not for them, fall back to their original wait/sleep state.

Attaching to Shared Memory

Each WebMedia plugin and forked application must connect to the same block of shared memory for proper communications. In order to facilitate this, a shared memory library has been created that allows a single function call to create the shared memory connection. The following code example shows how to connect to shared memory:

```
#include "cmdblock.h"
#include "sharedclient.h"

// connect to shared memory
if(clientMemoryAttach() != 0)
  return(-1);     // unable to attach to shared memory

// the cmd_block pointer now references shared memory
// display the current command and status
printf("command: %d\n",cmd_block->command);
printf("status: %d\n",cmd_block->status);

// disconnect from shared memory
clientMemoryDetach();
```

Command Handshaking

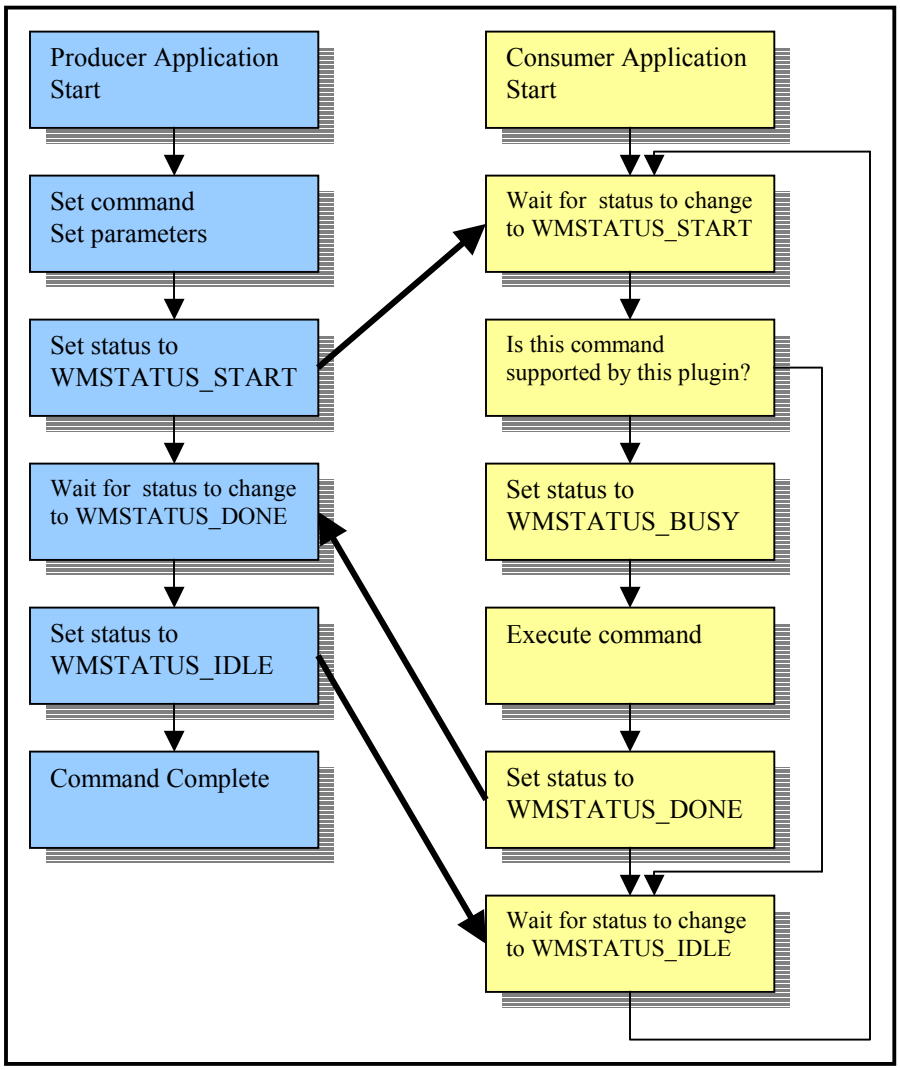
The command producer/consumer state has been implemented in software to provide an efficient and flexible command handshaking operation. In order to do this, a cmd_block structure has been created:

```
typdef struct
{
  int status;
  int command;
  char parm1[256];
  char parm2[256];
  int parm3;
  int parm4;
}WMCmdBlock;
```

Four parameters are currently available: two string buffers and two integer values. The status entry contains the command status, and is selected from the following enumerated values:

```
typedef enum
{
  WMSTATUS_IDLE,
  WMSTATUS_START,
  WMSTATUS_BUSY,
  WMSTATUS_DONE
}WMStatus;
```

To deliver a command to a forked application through shared memory, a minimal amount of handshaking is required. The following flowchart shows the handshaking logic for the producer and consumer threads:



Before a command handshaking sequence can begin, the system must be in an idle state. This is indicated by the status variable containing a WMSTATUS_IDLE value. At this point, the producer thread has not started sending a command, and the consumer threads are in a loop/sleep state, waiting for a change in the status value.

The producer will write in the new command value and any parameters required for the command, and then change the status value to WMSTATUS_START. This is the signal for all consumer threads to examine the command and determine if they should handle the command. If this is not their command, they will enter a sleep/test loop waiting for the status value to change back to WMSTATUS_IDLE, indicating that command delivery is complete.

If this is their command, the consumer thread will set the status value to WMSTATUS_BUSY, indicating the command is being executed or they may also just set the status value directly to WMSTATUS_DONE, indicating the command is completed. At this point, the producer thread is n a sleep/loop state waiting for the consumer thread to indicate it has completed the command by setting WMSTATUS_DONE.

When the consumer thread has completed the command, it indicates such by setting the status value to WMSTATUS_DONE, as previously indicated. This is the signal to the producer thread to recover any return values (usually deposited into the parm3 integer variable) and then indicate command completion by setting the status value to WMSTATUS_IDLE. When the consumer thread sees the status revert to WMSTATUS_IDLE, this is it's signal to return back to it's beginning state and wait for another command.

The following code sample demonstrates how to issue a command from the producer (plugin) side. This is just one example; there are numerous ways to implement the logic described above:

```
#define TIMEOUT 5
int tick;

// make sure cmd_block is idle
if(cmd_block->status != WMSTATUS_IDLE)
  return(-1);

// issue the command
cmd_block->command = WMCMD_SOME_COMMAND;
strcpy(cmd_block->parm1,"parm1");
cmd_block->parm3,12345
cmd_block->status = WMSTATUS_START;

// wait for the command to finish
tick = time(NULL);
while(1)
{
  if(cmd_block->status == WMSTATUS_DONE)
    break;
  usleep(1000);
  if((tick + TIMEOUT) <= time(NULL))
  {
    cmd_block->command = WMCMD_NONE;
    cmd_block->status = WMSTATUS_IDLE;
    return(-2);
  }
}
cmd_block->command = WMCMD_NONE;
cmd_block->status = WMSTATUS_IDLE;
```

The following code sample demonstrates how to respond to a command from the consumer (forked application) side. This is just one example; there are numerous ways to implement the logic described above:

```
int cmd;

// loop forever, waiting for command sequences
while(1)
{
  // sleep for .1 second, then check for a command
  usleep(100000);
  if(cmd_block->status != WMSTATUS_START)
```

```
    continue;

  // command found, is it one of mine?
  if((cmd_block->command == WMCMD_MYCMD_STOP) ||
     (cmd_block->command == WMCMD_MYCMD_PLAY) ||
     (cmd_block->command == WMCMD_MYCMD_PAUSE))
  break;
}

// command found, and it's one I should handle
cmd = cmd_block->command;

// set status to done, wait for the producer to handshake
cmd_block->status = WMSTATUS_DONE;
while(cmd_block->status != WMSTATUS_IDLE)
  ;

// return the recovered command
return(cmd);
```

The examples at the end of this document show how the command handshaking logic has been implemented.

System Plugin Example

Introduction The following plugin example is a fully functional implementation of .wav file playback under WebMedia. Since this is a system plugin, once audio playback begins, the plugin will not return successfully until the entire audio file has been played. This plugin was originally developed to add sound feedback during menu navigation.

This is a good example of a plugin that could be either a system plugin or a control plugin. If this plugin is used to payback a 1-second audio cue, then everything is fine. If this plugin is used to play a full song, the plugin will not return until after the song playback is completed – which could be several minutes. In the latter case, it make more sense to write an application that would be responsible for audio playback, and write the plugin as a control plugin.

```
/*************************************************************************
*                                                                        *
* COPYRIGHT (c) 2001   Century Software   All Rights Reserved            *
*                                                                        *
* This software is the property of Century Software and shall not be     *
* reproduced or copied in whole or  used in whole or in  part as the     *
* basis for the manufacture or sale of items, nor shall such copy be     *
* sold or constitute part of a sale without written permission.          *
*                                                                        *
* RESTRICTED RIGHTS LEGEND                                               *
*                                                                        *
* Use, duplication,  or disclosure by the government is subject to       *
* restriction as set forth in paragraph (b)(3)(b) of the Rights in       *
* Technical Data and Computer Software clause in DAR 7-104.9(a).         *
*                                                                        *
* CENTURY SOFTWARE                                                       *
* 5284 SOUTH COMMERCE DRIVE - SUITE C-134                                *
* SALT LAKE CITY, UTAH   84107                                           *
*                                                                        *
*************************************************************************
*                                                                        *
* FILENAME: wavplugin.cpp                                                *
*                                                                        *
*************************************************************************/



/*
**
** Imported "Include" Files
**
*/
#include <stdio.h>
#include <stdlib.h>



/*
```

```
**
** Local Structure Definitions
**
*/
typedef struct
{
  char *cmd;
  int (*func)(int argc, char *argv[]);
}CmdEntry;



/*
**
** Local Variable Declarations
**
*/
static char *plugin_name = "Audio WAV player";
static char *plugin_version = "1.0.0";
static char *plugin_keyword = "wav";



/*
**
** Command: play
**
*/
static int playFunc(int argc, char *argv[])
{
  char buf[256];

  sprintf(buf,"/usr/bin/sox %s -t ossdsp /dev/dsp",argv[0]);
  system(buf);
  return(0);
}



/*
**
** Command List
**
*/
static CmdEntry cmdlist[] =
{
  "play",playFunc,
  NULL,NULL
};



/*
**
** This function will initialize the wavaudio module and prepare it
** for subsequent operations.
**
```

```
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginStartup(void)
{
  return(0);
}



/*
**
** This function will closeout the wav audio module and shut it down.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginShutdown(void)
{
  return(0);
}



/*
**
**
**
*/
extern "C"
int pluginName(char *result)
{
  strcpy(result,plugin_name);
  return(0);
}



/*
**
**
**
*/
extern "C"
int pluginVersion(char *result)
{
  strcpy(result,plugin_version);
  return(0);
}
```

```
/*
**
** This function will return the subsystem identification string
** for the wav audio module ("wav").
**
** The "result" parameter is a pointer to the buffer where the
** subsystem identification string will be stored.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginKeyword(char *result)
{
  strcpy(result,plugin_keyword);
  return(0);
}



/*
**
** This function will display the commands available for the
** wav audio module.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginHelp(void)
{
  printf("   %s play filename\n",plugin_keyword);
  return(0);
}



/*
**
** This function will execute the specified wav audio module command.
**
** The "argc" parameter specifies the number of string parameters
** found in the argv array. The "argv" parameter is an array of
** pointers to the command and it's parameters.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginCommand(int argc, char *argv[])
{
```

```
  int count,result;

  // scan the list of commands for a match
  count = 0;
  while(1)
  {
    // check for end of list (indicates unknown command)
    if(cmdlist[count].cmd == NULL)
    {
      result = -1;
      break;
    }

    // check for command match
    if(strcmp(cmdlist[count].cmd,argv[0]) == 0)
    {
      result = (cmdlist[count].func)(argc - 1,&argv[1]);
      break;
    }

    // bump to next command
    ++count;
  }

  // return result and exit
  return(result);
}
```

Control Plugin Example

Introduction The following plugin sample shows the producer side of a control plugin designed to issue control commands to a forked process that decodes and displays a streaming MPEG2 data stream.

```
/*************************************************************************
*                                                                        *
* COPYRIGHT (c) 2001   Century Software   All Rights Reserved            *
*                                                                        *
* This software is the property of Century Software and shall not be     *
* reproduced or copied in whole or  used in whole or in  part as the     *
* basis for the manufacture or sale of items, nor shall such copy be     *
* sold or constitute part of a sale without written permission.          *
*                                                                        *
* RESTRICTED RIGHTS LEGEND                                               *
*                                                                        *
* Use, duplication,  or disclosure by the government is subject to       *
* restriction as set forth in paragraph (b)(3)(b) of the Rights in       *
* Technical Data and Computer Software clause in DAR 7-104.9(a).         *
*                                                                        *
* CENTURY SOFTWARE                                                       *
* 5284 SOUTH COMMERCE DRIVE - SUITE C-134                                *
* SALT LAKE CITY, UTAH   84107                                           *
*                                                                        *
*************************************************************************
*                                                                        *
* FILENAME: mpegplugin.cpp                                               *
*                                                                        *
*************************************************************************/



/*
**
** Imported "Include" Files
**
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include "audio.h"
#include "cmdblock.h"
#include "dispatcher.h"
#include "errorbase.h"
#include "sharedserver.h"
#include "srcmpeg.h"
#include "state.h"



/*
**
** Local Constant Definitions
```

```
**
*/
#define TIMEOUT 2



/*
**
** Local Enumeration Definitions
**
*/



/*
**
** Local Structure Definitions
**
*/
typedef struct
{
  char *cmd;
  int (*func)(int argc, char *argv[]);
}CmdEntry;



/*
**
** Local Variable Declarations
**
*/
static char *plugin_name = "Hardware MPEG Support";
static char *plugin_version = "1.0.0";
static char *plugin_keyword = "mpeg";



/*
**
**
**
*/
static int loadFrame(void)
{
  int tick;

  // make sure cmd_block is idle
  if(cmd_block->status != WMSTATUS_IDLE)
    return(ERRORBASE_MPEG + 10);

  // issue the stop command
  cmd_block->command = WMCMD_VIEWER_FRAME_LOAD;
  strcpy(cmd_block->parm1,"Dialog");
  strcpy(cmd_block->parm2,"dialogstream.html");
  cmd_block->status = WMSTATUS_START;
```

```
  // wait for the command to finish
  tick = time(NULL);
  while(1)
  {
    if(cmd_block->status == WMSTATUS_DONE)
      break;
    usleep(1000);
    if((tick + TIMEOUT) <= time(NULL))
    {
      cmd_block->command = WMCMD_NONE;
      cmd_block->status = WMSTATUS_IDLE;
      //      return(ERRORBASE_MPEG + 11);
      return(0);
    }
  }
  cmd_block->command = WMCMD_NONE;
  cmd_block->status = WMSTATUS_IDLE;

  return(0);
}



/*
**
** Command: source
**
*/
static int sourceFunc(int argc, char *argv[])
{
  if(VAPISrcMPEGSelect() != 0)
    return(ERRORBASE_MPEG + 0);
  strcpy(state_source,plugin_keyword);
  loadFrame();
  return(0);
}



/*
**
** Command: stop
**
*/
static int stopFunc(int argc, char *argv[])
{
  int tick;

  // make sure cmd_block is idle
  if(cmd_block->status != WMSTATUS_IDLE)
    return(ERRORBASE_MPEG + 1);

  // issue the stop command
  cmd_block->command = WMCMD_MPEG_STOP;
  cmd_block->status = WMSTATUS_START;

  // wait for the command to finish
```

```
  tick = time(NULL);
  while(1)
  {
    if(cmd_block->status == WMSTATUS_DONE)
      break;
    usleep(1000);
    if((tick + TIMEOUT) <= time(NULL))
    {
      cmd_block->command = WMCMD_NONE;
      cmd_block->status = WMSTATUS_IDLE;
      return(ERRORBASE_MPEG + 2);
    }
  }
  cmd_block->command = WMCMD_NONE;
  cmd_block->status = WMSTATUS_IDLE;

  return(0);
}



/*
**
** Command: play
**
*/
static int playFunc(int argc, char *argv[])
{
  int tick;

  // make sure cmd_block is idle
  if(cmd_block->status != WMSTATUS_IDLE)
    return(ERRORBASE_MPEG + 3);

  // issue the stop command
  strcpy(cmd_block->parm1,argv[0]);

  // start the mpegplayer process
  if ( fork() == 0 ) /* child */
    {
        execl( "/bin/mpegplayer", "mpegplayer", "", NULL);
    }

  cmd_block->command = WMCMD_MPEG_PLAY;
  cmd_block->status = WMSTATUS_START;

  // wait for the command to finish
  tick = time(NULL);
  while(1)
  {
    if(cmd_block->status == WMSTATUS_DONE)
      break;
    usleep(1000);
    if((tick + TIMEOUT) <= time(NULL))
    {
      cmd_block->command = WMCMD_NONE;
      cmd_block->status = WMSTATUS_IDLE;
```

```
      return(ERRORBASE_MPEG + 4);
    }
  }
  cmd_block->command = WMCMD_NONE;
  cmd_block->status = WMSTATUS_IDLE;

  return(0);
}



/*
**
** Command: pause
**
*/
static int pauseFunc(int argc, char *argv[])
{
  int tick;

  // make sure cmd_block is idle
  if(cmd_block->status != WMSTATUS_IDLE)
    return(ERRORBASE_MPEG + 5);

  // issue the stop command
  cmd_block->command = WMCMD_MPEG_PAUSE;
  cmd_block->status = WMSTATUS_START;

  // wait for the command to finish
  tick = time(NULL);
  while(1)
  {
    if(cmd_block->status == WMSTATUS_DONE)
      break;
    usleep(1000);
    if((tick + TIMEOUT) <= time(NULL))
    {
      cmd_block->command = WMCMD_NONE;
      cmd_block->status = WMSTATUS_IDLE;
      return(ERRORBASE_MPEG + 6);
    }
  }
  cmd_block->command = WMCMD_NONE;
  cmd_block->status = WMSTATUS_IDLE;

  return(0);
}

/*****************************************************************************
           ***\
**
**          Function:   int volumeupFunc()
**          Desc:       Increments the volume level by 1 unit
**          Acecpts:    int argc = number of args
**                            char *argv[] = Arguments
**          Returns:    int; 0 on success, or an ERRORBASE error code
**
```

```
\*************************************************************************
            ***/
static int
volumeupFunc(int argc, char *argv[])
{
            SetAudio(AV_CHAN_MPEG, 0, 1);
            return (0);
} /* end of volumeupFunc() */

/*************************************************************************
            ***\
**
**          Function:   int volumedownFunc()
**          Desc:       Decrements the volume level by 1 unit
**          Accepts:    int argc = number of args
**                            char *argv[] = Arguments
**          Returns:    int; 0 on success, or an ERRORBASE eror code
**
\*************************************************************************
            ***/
static int
volumedownFunc(int argc, char *argv[])
{
            SetAudio(AV_CHAN_MPEG, 0, -1);
            return (0);
} /* end if volumedownFunc() */

/*************************************************************************
            ***\
**
**          Function:   int volumemuteFunc()
**          Desc:       Mutes the volume
**          Accepts:    int argc = Number of args
**                            char *argv[] = Arguments
**          Returns:    int; 0 on success, ERRORBASE value on error
**
\*************************************************************************
            ***/
static int
volumemuteFunc(int argc, char *argv[])
{
            fprintf(stderr, "Muting mpeg\n");
            SetAudio(AV_CHAN_MPEG, 1, 0);
            return (0);
} /* end of volumemuteFunc() */

/*************************************************************************
            ***\
**
**          Funciton:   int volumeunmuteFunc()
**          Desc:       Unmutes the volume
**          Accepts:    int argc = Number of args
**                            char *argv[] = Arguments
**          Returns:    int; 0 on success, ERRORBASE value on error
**
\*************************************************************************
            ***/
```

```
static int
volumeunmuteFunc(int argc, char *argv[])
{
            fprintf(stderr, "Unmuting mpeg\n");
            SetAudio(AV_CHAN_MPEG, 0, 0);
            return (0);
} /* end of volumeunmuteFunc() */

/*
**
** Command List
**
*/
static CmdEntry cmdlist[] =
{
  "source",sourceFunc,
  "stop",stopFunc,
  "play",playFunc,
  "pause",pauseFunc,
  "volup", volumeupFunc,
  "voldn", volumedownFunc,
  "mute", volumemuteFunc,
  "unmute", volumeunmuteFunc,
  NULL,NULL
};



/*
**
** This function is called once, just after the plugin has been loaded
** and all plugin entrypoints have been registered. This function will
** be called before any other plugin entrypoint functions are called.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
*/
extern "C"
int pluginStartup(void)
{
  return(0);
}



/*
**
** This function is called once, just before the plugin is unloaded.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
```

```
*/
extern "C"
int pluginShutdown(void)
{
  return(0);
}



/*
**
** This function is called to recover the plugin's name/description.
**
** The "result" parameter is a pointer to the buffer where the plugin's
** name/description will be stored.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
*/
extern "C"
int pluginName(char *result)
{
  strcpy(result,plugin_name);
  return(0);
}



/*
**
** This function is called to recover the plugin's internal version number.
**
** The "result" parameter is a pointer to the buffer where the plugin's
** internal version number will be stored.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
*/
extern "C"
int pluginVersion(char *result)
{
  strcpy(result,plugin_version);
  return(0);
}



/*
**
** This function is called to recover the plugin's command keyword.
**
```

```
** The "result" parameter is a pointer to the buffer where the plugin's
** command keyword will be stored.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
*/
extern "C"
int pluginKeyword(char *result)
{
  strcpy(result,plugin_keyword);
  return(0);
}



/*
**
** This function is called to display this plugin's command set.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
*/
extern "C"
int pluginHelp(void)
{
  printf("   %s source\n",plugin_keyword);
  printf("   %s stop\n",plugin_keyword);
  printf("   %s play file\n",plugin_keyword);
  printf("   %s pause\n",plugin_keyword);
  printf("   %s mute\n",plugin_keyword);
  printf("   %s unmute\n",plugin_keyword);
  printf("   %s volup\n",plugin_keyword);
  printf("   %s voldn\n",plugin_keyword);
  return(0);
}



/*
**
** This function is called to execute one of the plugin's commands.
**
** The "argc" parameter specifies the number of string parameters
** found in the argv array. The "argv" parameter is an array of
** pointers to the command and it's parameters.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a plugin entrypoint.
**
```

```
*/
extern "C"
int pluginCommand(int argc, char *argv[])
{
  int count,result;

  // scan the list of commands for a match
  count = 0;
  while(1)
  {
    // check for end of list (indicates unknown command)
    if(cmdlist[count].cmd == NULL)
    {
      result = ERRORBASE_MPEG + 7;
      break;
    }

    // check for command match
    if(strcmp(cmdlist[count].cmd,argv[0]) == 0)
    {
      result = (cmdlist[count].func)(argc - 1,&argv[1]);
      break;
    }

    // bump to next command
    ++count;
  }

  // return result and exit
  return(result);
}
```

Plugin Template

Introduction This template can be used as a basis for new WebMedia plugin development. Copy the source out of this document and save into a template file.

```
/*************************************************************************
*                                                                        *
* COPYRIGHT (c) 2001   Century Software   All Rights Reserved            *
*                                                                        *
* This software is the property of Century Software and shall not be     *
* reproduced or copied in whole or  used in whole or in  part as the     *
* basis for the manufacture or sale of items, nor shall such copy be     *
* sold or constitute part of a sale without written permission.          *
*                                                                        *
* RESTRICTED RIGHTS LEGEND                                               *
*                                                                        *
* Use, duplication,  or disclosure by the government is subject to       *
* restriction as set forth in paragraph (b)(3)(b) of the Rights in       *
* Technical Data and Computer Software clause in DAR 7-104.9(a).         *
*                                                                        *
* CENTURY SOFTWARE                                                       *
* 5284 SOUTH COMMERCE DRIVE - SUITE C-134                                *
* SALT LAKE CITY, UTAH   84107                                           *
*                                                                        *
*************************************************************************
*                                                                        *
* FILENAME: webmedia_plugin_template.cpp                                 *
*                                                                        *
*************************************************************************/



/*
**
** Imported "Include" Files
**
*/
#include <stdio.h>
#include <stdlib.h>



// the following structure is used to create the command list. It contains
// a pointer to the command text and a pointer to the command function.
// this can be left as-is.

/*
**
** Local Structure Definitions
**
*/
typedef struct
{
  char *cmd;
  int (*func)(int argc, char *argv[]);
```

```
}CmdEntry;


// in the static variables below, enter a descriptive plugin name,
// version number (or use the one provided), and a unique plugin
// keyword. These variables are used throughout the plugin.

/*
**
** Local Variable Declarations
**
*/
static char *plugin_name = "";
static char *plugin_version = "1.0.0";
static char *plugin_keyword = "";



// The function below is a template function that can be used to
// implement commands. The argc/argv parameters being passed are
// the in the standard form; however only the parameters themselves
// are passed. If a command does not require parameters, the argc
// value will be zero and no pointers will be passed in the argv array.
// also note that the command 'zzz' will be found in the command list
// and associated with 'zzzFunc'.

/*
**
** Command: zzz
**
*/
static int zzzFunc(int argc, char *argv[])
{
  return(0);
}



// the list below contains the command text and a reference to the
// function used to execute the command. Replace zzz and zzzFunc
// with real commands and function names. Duplicate the line below
// for each command to be supported.

/*
**
** Command List
**
*/
static CmdEntry cmdlist[] =
{
  "zzz",zzzFunc,
  // add additional commands here. Use the line above as an example.
  NULL,NULL
};
```

```
/*
**
** This function will initialize the plugin module and prepare it
** for subsequent operations.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginStartup(void)
{
  return(0);
}



/*
**
** This function will closeout the plugin module and shut it down.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginShutdown(void)
{
  return(0);
}



/*
**
** This function returns the descriptive plugin name.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginName(char *result)
{
  strcpy(result,plugin_name);
  return(0);
}



/*
**
** This function returns the plugin version.
```

```
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginVersion(char *result)
{
  strcpy(result,plugin_version);
  return(0);
}



/*
**
** This function will return the subsystem identification string
** for the plugin module.
**
** The "result" parameter is a pointer to the buffer where the
** subsystem identification string will be stored.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginKeyword(char *result)
{
  strcpy(result,plugin_keyword);
  return(0);
}



// in the function below, change the printf statement to reflect
// a plugin command. add additional lines, one for each plugin
// command. For clear formatting, notice that there are three
// space codes preceeding the help line.

/*
**
** This function will display the commands available for the
** plugin module.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginHelp(void)
{
  printf("   %s zzz parm1 parm2 ... parmN\n",plugin_keyword);
  return(0);
}



/*
**
```

```
** This function will execute the specified plugin module command.
**
** The "argc" parameter specifies the number of string parameters
** found in the argv array. The "argv" parameter is an array of
** pointers to the command and it's parameters.
**
** If successful, '0' is returned. If an error occurs during function
** execution, a non-zero value is returned that describes the error.
**
** This function is a module entrypoint.
**
*/
extern "C"
int pluginCommand(int argc, char *argv[])
{
  int count,result;

  // scan the list of commands for a match
  count = 0;
  while(1)
  {
    // check for end of list (indicates unknown command)
    if(cmdlist[count].cmd == NULL)
    {
      result = -1;
      break;
    }

    // check for command match
    if(strcmp(cmdlist[count].cmd,argv[0]) == 0)
    {
      result = (cmdlist[count].func)(argc - 1,&argv[1]);
      break;
    }

    // bump to next command
    ++count;
  }

  // return result and exit
  return(result);
}
```