VLC media player API Documentation

Christophe Massiot
Developer (mailto:christophe.massiot@idealx.com)
IDEALX S.A.S. (http://www.idealx.com/)
Industrial Computing

VLC media player API Documentation

by Christophe Massiot

Published $Id: manual.xml 14125 2006-02-01 19:44:56Z courmisch $

Table of Contents

Glossary **5**

1. VLC Overview **1**

LibVLC 1
VLC 1
Modules 2
Threads 2
Code conventions 3

2. VLC interface **5**

A typical VLC run course 5
The message interface 5
Command line options 6
Playlist management 6
Module bank 6
The interface main loop 7
How to write an interface plugin 7

3. The complex multi-layer input **9**

What happens to a file 9
Stream Management 9
Structures exported to the interface 10
Methods used by the interface 12
Buffers management 13
Demultiplexing the stream 13

4. How to write a decoder **15**

What is precisely a decoder in the VLC scheme ? 15
Decoder configuration 15
Packet structures 15
The bit stream (input module) 16
Built-in decoders 17
The MPEG video decoder 18

5. The video output layer **20**

Data structures and main loop 20
Methods used by video decoders 20
How to write a video output plug-in 21
How to write a YUV plug-in 21

6. The audio output layer **22**

Audio output overview 22
Terminology 22
Audio sample formats 22
Typical runcourse 23
Mutual exclusion mechanism 25
Internal structures 25
API for the decoders 27
API for the output module 27
Writing an audio filter 28
Writing an audio mixer 28

A. Ports **30**

Port steps 30
Building 30

B. Advanced debugging **32**
Where does it crash ? 32
Other problems 32
C. Project history **33**
VIA and the Network2000 project 33
Foundation of the VideoLAN project 33
VLC media player design 33
The Opening 33
D. GNU Free Documentation License **35**
0. PREAMBLE 35
1. APPLICABILITY AND DEFINITIONS 35
2. VERBATIM COPYING 36
3. COPYING IN QUANTITY 36
4. MODIFICATIONS 36
5. COMBINING DOCUMENTS 37
6. COLLECTIONS OF DOCUMENTS 38
7. AGGREGATION WITH INDEPENDENT WORKS 38
8. TRANSLATION 38
9. TERMINATION 38
10. FUTURE REVISIONS OF THIS LICENSE 39

Glossary

> **Warning**
>
> Please note that this book is in no way a reference documentation on how DVDs work. Its only purpose is to describe the API available for programmers in VLC media player. It is assumed that you have basic knowledge of what MPEG is. The following paragraph is just here as a reminder :

AC3 : Digital Audio Compression Standard

Specification for coding audio data, used in DVD. The documentation is freely available (http://www.linuxvideo.org/devel/library/ac3-standard_a_52.pdf).

B (bi-directional) picture

Picture decoded from its own data, *and* from the data of the previous and next (that means *in the future*) reference pictures (I or P pictures). It is the most compressed picture format, but it is less fault-tolerant.

DVD : Digital Versatile Disc

Disc hardware format, using the UDF file system, an extension of the ISO 9660 file system format and a video format which is an extension of the MPEG-2 specification. It basically uses MPEG-2 PS files, with subtitles and sound tracks encoded as private data and fed into non-MPEG decoders, along with .ifo files describing the contents of the DVD. DVD specifications are very hard to get, and it takes some time to reverse-engineer it. Sometimes DVD are zoned and scrambled, so we use a brute-force algorithm to find the key.

ES : Elementary Stream

Continuous stream of data fed into a decoder, without any multiplexing layer. ES streams can be MPEG video MPEG audio, AC3 audio, LPCM audio, SPU subpictures...

Field picture

Picture split in two fields, even and odd, like television does. DVDs coming from TV shows typically use field pictures.

Frame picture

Picture without even/odd discontinuities, unlike field pictures. DVDs coming from movies typically use frame pictures.

I (intra) picture

Picture independantly coded. It can be decoded without any other reference frame. It is regularly sent (like twice a second) for resynchronization purposes.

IDCT : Inverse Discrete Cosinus Transform

IDCT is a classical mathematical algorithm to convert from a space domain to a frequency domain. In a nutshell, it codes differences instead of coding all absolute pixels. MPEG uses an 2-D IDCT in the video decoder, and a 1-D IDCT in the audio decoder.

LPCM : Linear Pulse Code Modulation

LPCM is a non-compressed audio encoding, available in DVDs.

MPEG : Motion Picture Expert Group

Specification describing a standard syntax of files and streams for carrying motion pictures and sound. MPEG-1 is ISO/IEC 11172 (three books), MPEG-2 is ISO/IEC 13818. MPEG-4 version 1 is out, but this player doesn't support it. It is relatively easy to get an MPEG specification from ISO or equivalent, drafts are even freely available on the Internet.

P (predictive) picture

Picture decoded from its own data *and* data from a reference picture, which is the last I or P picture received.

PES : Packetized Elementary Stream

A chunk of elementary stream. It often corresponds to a logical boundary of the stream (for instance a picture change), but it is not mandatory. PES carry the synchronization information.

PTS : Presentation Time Stamp

Time at which the content of a PES packet is supposed to be played. It is used for A/V synchronization.

PS : Program Stream

File format obtained by concatenating PES packets and inserting Pack headers and System headers (for timing information). It is the only format described in MPEG-1, and the most used format in MPEG-2.

RGB : Red Green Blue

Picture format where every pixel is calculated in a vector space whose coordinates are red, green, and blue. This is natively used by monitors and TV sets.

SPU : Sub Picture Unit

Picture format allowing to do overlays, such as subtitles or DVD menus.

SCR : System Clock Reference

Time at which the first byte of a particular pack is supposed to be fed to the decoder. VLC uses it to read the stream at the right pace.

SDL : Simple DirectMedia Layer

SDL (http://www.libsdl.org/) is a cross-platform multimedia library designed to provide fast access to the video framebuffer and the audio device. Since version 1.1, it features YUV overlay support, which reduces decoding times by a third.

TS : Transport Stream

Stream format constituted of fixed size packets (188 bytes), defined by ISO/IEC 13818-1. PES packets are split among several TS packets. A TS stream can contain several programs. It is used in streaming applications, in particular for satellite or cable broadcasting.

YUV : Luminance/Chrominance

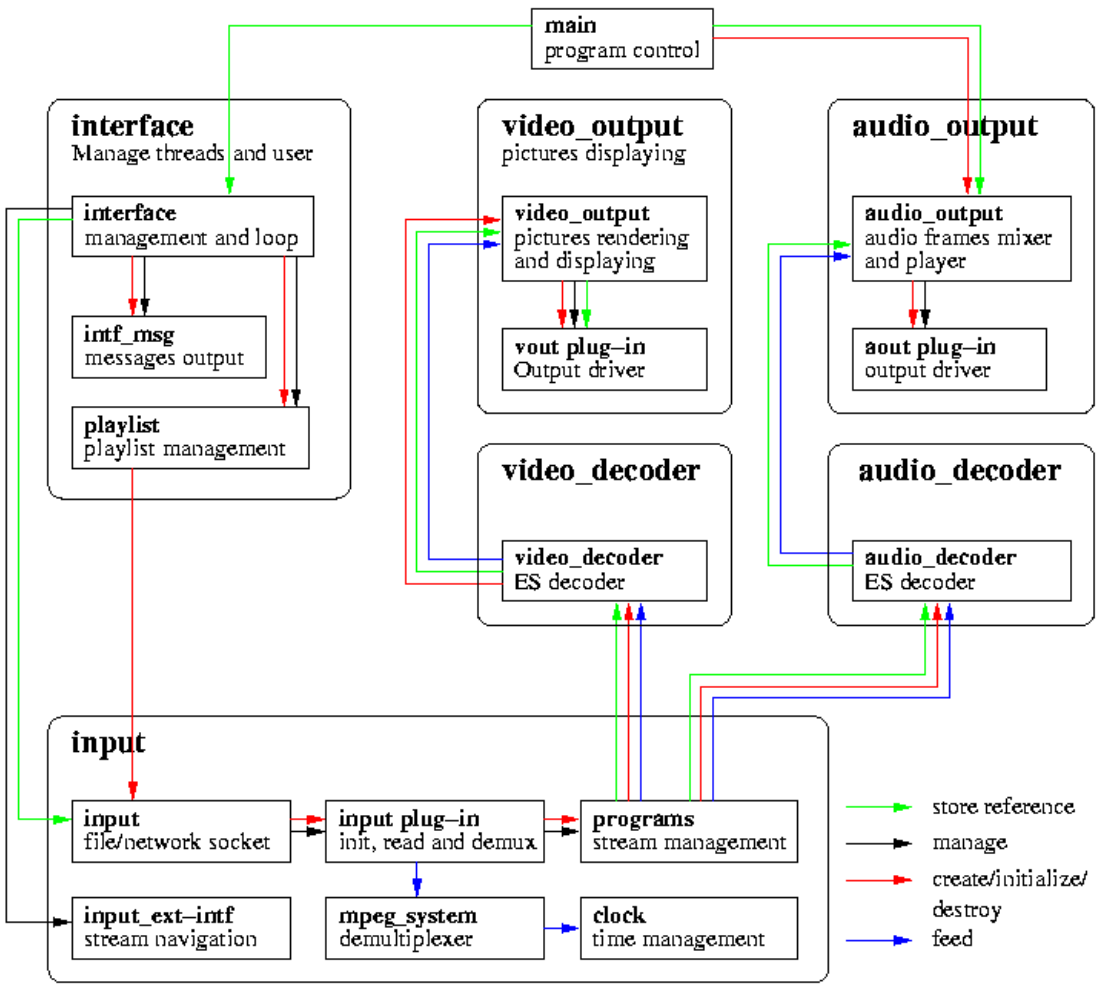
Picture format with 1 coordinate of luminance (black and white) and 2 coordinates of chrominance (red and blue). This is natively used by PAL video system, for backward compatibility with older black and white TV sets. Your eyes distinguish luminance variations much better than chrominance variations, so you can compress them more. It is therefore well suited for image compression, and is used by the MPEG specification. The RGB picture can be obtained from the YUV one via a costly matrix multiply operation, which can be done in hardware by most modern video cards ("YUV acceleration").

Chapter 1. VLC Overview

LibVLC

LibVLC is the core part of VLC. It is a library providing an interface for programs such as VLC to a lot of functionalities such as stream access, audio and video output, plugin handling, a thread system. All the LibVLC source files are located in the `src/` directory and its subdirectories:

- `interface/`: contains code for user interaction such as key presses and device ejection.
- `playlist/`: manages playlist interaction such as stop, play, next, or random playback.
- `input/`: opens an input module, reads packets, parses them and passes reconstituted elementary streams to the decoder(s).
- `video_output/`: initializes the video display, gets all pictures and subpictures (ie. subtitles) from the decoder(s), optionally converts them to another format (such as YUV to RGB), and displays them.
- `audio_output/`: initializes the audio mixer, ie. finds the right playing frequency, and then resamples audio frames received from the decoder(s).
- `stream_output/`: TODO
- `misc/`: miscellaneous utilities used in other parts of libvlc, such as the thread system, the message queue, CPU detection, the object lookup system, or platform-specific code.



VLC

VLC is a simple program written around LibVLC. It is very small, but is a fully featured multimedia player thanks to LibVLC's support for dynamic modules.

Modules

Modules are located in the `modules/` subdirectory and are loaded at runtime. Every module may offer different features that will best suit a particular file or a particular environment. Besides, most portability works result in the writing of an audio_output/video_output/interface module to support a new platform (eg. BeOS or MacOS X).

Plugin modules are loaded and unloaded dynamically by functions in `src/misc/modules.c` and `include/modules*.h`. The API for writing modules will be discussed in a following chapter.

Modules can also be built directly into the application which uses LibVLC, for instance on an operating system that does not have support for dynamically loadable code. Modules statically built into the application are called builtins.

Threads

Thread management

VLC is heavily multi-threaded. We chose against a single-thread approach because decoder preemptibility and scheduling would be a mastermind (for instance decoders and outputs have to be separated, otherwise it cannot be warrantied that a frame will be played at the exact presentation time), and we currently have no plan to support a single-threaded client. Multi-process decoders usually imply more overhead (problems of shared memory) and communication between processes is harder.

Our threading structure is modeled on pthreads. However, for portability reasons, we don't call `pthread_*` functions directly, but use a similar wrapper, made of `vlc_thread_create`, `vlc_thread_exit`, `vlc_thread_join`, `vlc_mutex_init`, `vlc_mutex_lock`, `vlc_mutex_unlock`, `vlc_mutex_destroy`, `vlc_cond_init`, `vlc_cond_signal`, `vlc_cond_broadcast`, `vlc_cond_wait`, `vlc_cond_destroy`, and structures vlc_thread_t, vlc_mutex_t, and vlc_cond_t.

Synchronization

Another key feature of VLC is that decoding and playing are asynchronous: decoding is done by a decoder thread, playing is done by audio_output or video_output thread. The design goal is to ensure that an audio or video frame is played exactly at the right time, without blocking any of the decoder threads. This leads to a complex communication structure between the interface, the input, the decoders and the outputs.

Having several input and video_output threads reading multiple files at the same time is permitted, despite the fact that the current interface doesn't allow any way to do it [this is subject to change in the near future]. Anyway the client has been written from the ground up with this in mind. This also implies that a non-reentrant library (including in particular liba52) cannot be used without using a global lock.

Presentation Time Stamps located in the system layer of the stream are passed to the decoders, and all resulting samples are dated accordingly. The output layers are supposed to play them at the right time. Dates are converted to microseconds ; an absolute date is the number of microseconds since Epoch (Jan 1st, 1970). The mtime_t type is a signed 64-bit integer.

The current date can be retrieved with `mdate()`. The execution of a thread can be suspended until a certain *`date`* via `mwait` *`( mtime_t date )`*. You can sleep for a fixed number of microseconds with `msleep` *`( mtime_t delay )`*.

> **Warning**
>
> Please remember to wake up slightly *before* the presentation date, if some particular treatment needs to be done (e.g. a chroma transformation). For instance in `modules/codec/mpeg_video/synchro.c`, track of the average decoding times is kept to ensure pictures are not decoded too late.

Code conventions

Function naming

All functions are named accordingly : module name (in lower case) + _ + function name (in mixed case, *without underscores*). For instance : `intf_FooFunction`. Static functions don't need usage of the module name.

Variable naming

Hungarian notations are used, that means we have the following prefixes :

- i_ for integers (sometimes l_ for long integers) ;
- b_ for booleans ;
- d_ for doubles (sometimes f_ for floats) ;
- pf_ for function pointers ;
- psz_ for a Pointer to a String terminated by a Zero (C-string) ;
- More generally, we add a p when the variable is a pointer to a type.

If one variable has no basic type (for instance a complex structure), don't put any prefix (except p_* if it's a pointer). After one prefix, put an *explicit* variable name *in lower case*. If several words are required, join them with an underscore (no mixed case). Examples :

- data_packet_t * `p_buffer;`
- char `psz_msg_date[42];`
- int `pi_es_refcount[MAX_ES];`
- void `(* pf_next_data_packet)( int * );`

A few words about white spaces

First, never use tabs in the source (you're entitled to use them in the Makefile :-). Use **set expandtab** under vim or the equivalent under emacs. Indents are 4 spaces long.

Second, put spaces *before and after* operators, and inside brackets. For instance :

```
for( i = 0; i < 12; i++, j += 42 );
```

Third, leave braces alone on their lines (GNU style). For instance :

```
if( i_es == 42 )
{
    p_buffer[0] = 0x12;
}
```

We write C, so use C-style comments /* ... */.

Chapter 2. VLC interface

A typical VLC run course

This section describes what happens when you launch the vlc program. After the ELF dynamic loader blah blah blah, the main thread becomes the interface thread and starts up in `src/interface/main.c`. It passes through the following steps :

1. CPU detection : which CPU are we running on, what are its capabilities (MMX, MMXEXT, 3DNow, AltiVec...) ?
2. Message interface initialization ;
3. Command line options parsing ;
4. Playlist creation ;
5. Module bank initialization ;
6. Interface opening ;
7. Signal handler installation : SIGHUP, SIGINT and SIGQUIT are caught to manage a clean quit (please note that the SDL library also catches SIGSEGV) ;
8. Audio output thread spawning ;
9. Video output thread spawning ;
10. Main loop : events management ;

Following sections describe each of these steps in particular, and many more.

The message interface

It is a know fact that `printf()` functions are not necessarily thread-safe. As a result, one thread interrupted in a `printf()` call, followed by another calls to it, will leave the program in an undetermined state. So an API must be set up to print messages without crashing.

This API is implemented in two ways. If `INTF_MSG_QUEUE` is defined in `config.h`, every `printf`-like (see below) call will queue the message into a chained list. This list will be printed and flushed by the interface thread once upon an event loop. If `INTF_MSG_QUEUE` is undefined, the calling thread will acquire the print lock (which prevents two print operations to occur at the same time) and print the message directly (default behaviour).

Functions available to print messages are :

- `intf_Msg ( char * psz_format, ... )` : Print a message to `stdout`, plain and stupid (for instance "vlc 0.2.72 (Apr 16 2001)").
- `intf_ErrMsg ( char * psz_format, ... )` : Print an error message to `stderr`.
- `intf_WarnMsg ( int i_level, char * psz_format, ... )` : Print a message to `stderr` if the warning level (determined by -v, -vv and -vvv) is low enough.

> **Note:** Please note that the lower the level, the less important the message is (dayou spik ingliche ?).

- `intf_DbgMsg ( char * psz_format, ... )` : This function is designed for optional checkpoint messages, such as "we are now entering function dvd_foo_thingy". It does nothing in non-trace mode. If the VLC is compiled with --enable-trace, the message is either written to the file `vlc-trace.log` (if TRACE_LOG is defined in config.h), or printed to `stderr` (otherwise).
- `intf_MsgImm, intf_ErrMsgImm, intf_WarnMsgImm, intf_DbgMsgImm` : Same as above, except that the message queue, in case `INTF_MSG_QUEUE` is defined, will be flushed before the function returns.
- `intf_WarnHexDump ( int i_level, void * p_data, int i_size )` : Dumps `i_size` bytes from `p_data` in hexadecimal. `i_level` works like `intf_WarnMsg` . This is useful for debugging purposes.
- `intf_FlushMsg` () : Flush the message queue, if it is in use.

Command line options

VLC uses GNU getopt to parse command line options. getopt structures are defined in `src/interface/main.c` in the "Command line options constants" section. To add a new option This section needs to be changed, along with `GetConfiguration` and `Usage`.

Most configuration directives are exchanged via the environment array, using `main_Put*Variable` and `main_Get*Variable`. As a result, **./vlc --height 240** is strictly equivalent to : **vlc_height=240 ./vlc**. That way configuration variables are available everywhere, including plugins.

> **Warning**
>
> Please note that for thread-safety issues, you should not use `main_Put*Variable` once the second thread has been spawned.

Playlist management

The playlist is created on startup from files given in the command line. An appropriate interface plugin can then add or remove files from it. Functions to be used are described in `src/interface/intf_playlist.c`. `intf_PlaylistAdd` and `intf_PlaylistDelete` are typically the most common used.

The main interface loop `intf_Manage` is then supposed to *start and kill input threads* when necessary.

Module bank

On startup, VLC creates a bank of all available .so files (plugins) in `., ./lib, /usr/local/lib/videolan/vlc (PLUGIN_PATH)`, and built-in plugins. Every plugin is checked with its capabilities, which are :

- MODULE_CAPABILITY_INTF : An interface plugin ;
- MODULE_CAPABILITY_ACCESS : A sam-ism, unused at present ;
- MODULE_CAPABILITY_INPUT : An input plugin, for instance PS or DVD ;
- MODULE_CAPABILITY_DECAPS : A sam-ism, unused at present ;
- MODULE_CAPABILITY_ADEC : An audio decoder ;
- MODULE_CAPABILITY_VDEC : A video decoder ;
- MODULE_CAPABILITY_MOTION : A motion compensation module (for the video decoder) ;

- MODULE_CAPABILITY_IDCT : An IDCT module (for the video decoder) ;
- MODULE_CAPABILITY_AOUT : An audio output module ;
- MODULE_CAPABILITY_VOUT : A video output module ;
- MODULE_CAPABILITY_YUV : A YUV module (for the video output) ;
- MODULE_CAPABILITY_AFX : An audio effects plugin (for the audio output ; unimplemented) ;
- MODULE_CAPABILITY_VFX : A video effects plugin (for the video output ; unimplemented) ;

How to write a plugin is described in the latter sections. Other threads can request a plugin descriptor with `module_Need ( module_bank_t * p_bank, int i_capabilities, void * p_data ). p_data` is an optional parameter (reserved for future use) for the `pf_probe()` function. The returned module_t structure contains pointers to the functions of the plug-in. See `include/modules.h` for more information.

The interface main loop

The interface thread will first look for a suitable interface plugin. Then it enters the main interface loop, with the plugin's `pf_run` function. This function will do what's appropriate, and every 100 ms will call (typically via a GUI timer callback) `intf_Manage`.

`intf_Manage` cleans up the module bank by unloading unnecessary modules, manages the playlist, and flushes waiting messages (if the message queue is in use).

How to write an interface plugin

API for the Module

Have a look the files in directories `modules/misc/control`, `modules/misc/dummy`, `modules/misc/access`, or `modules/gui`. However the GUI interfaces are not very easy to understand, since they are quite big. I suggest to start digging into a non-graphical interface modules first. For example `modules/control/hotkeys.c`.

An interface module is made of 3 entry functions and a module description:

- The module description is made of macros that declares the capabilities of the module (interface, in this case) with their priority, the module description as it will appear in the preferences of GUI modules that implement them, some configuration variables specific to the module, shortcuts, sub-modules, etc.
- `Open ( vlc_object_t* p_object )`: This is called by VLC to initialize the module.
- `Run ( vlc_object_t* p_object )`: really does the job of the interface module (waiting for user input and displaying info). It should check periodically that `p_intf->b_die` is not `VLC_TRUE`.
- `Close ( vcl_object_t * p_object )` function is called by VLC to uninitialize the module (basically, this consists in destroying whatever have been allocated by `Open`)

The above functions take a `vlc_object_t*` as argument, but that may need to be cast into a `intf_thread_t*` depending on your needs. This structure is often needed as a parameter for exported VLC functions, such as `msg_Err()`, `msg_Warn()`, ...

Define `intf_sys_t` to contain any variable you need (don't use static variables, they suck in a multi-threaded application :-).

If additional capabilities (such as Open button, playlist, menus, etc.) are needed, consult one of the GUI modules. One of the simpler GUI modules to consult might be `modules/gui/ncurses/ncurses.c`. It is a quite simple complete interface module with playlist interaction, and progress bar, among other things.

Arranging for your Module to get Compiled

If you create a new directory for your module, add a `Modules.am` file in it. In this file, put something like :
`SOURCES_yourmodule = myfile1.c myfile2.c`

Then go to the main `configure.ac` file, and add in the `AC_CONFIG_FILES` section (towards the end of the file) a line similar to the others.

If you don't create a directory for your plugin (but instead just put it in an existing directory), you only have to add the two SOURCES_... lines to the existing `Modules.am` file

This declares your module; it does not arrange for it to be automatically compiled; automatic compilatoin is described further below.

You do not write a `Makefile` for your module. Instead this is done via the bootstrap and configuration process. So now run:

`./bootstrap`

`./configure` *configure-options*

`make`

To build the module manually, go to the directory it resides and type `make libyourmodule_plugin.so` (or .dll, or whatever the file type for a shared library is on your Operating System.)

To *automatically* have your module get built, you also set this in the `configure.ac` file; add your module name to the `default modules` section in one of the `AX_ADD_PLUGINS` directives.

Chapter 3. The complex multi-layer input

The idea behind the input module is to treat packets, without knowing at all what is in it. It only takes a packet, reads its ID, and delivers it to the decoder at the right time indicated in the packet header (SCR and PCR fields in MPEG). All the basic browsing operations are implemented without peeking at the content of the elementary stream.

Thus it remains very generic. This also means you can't do stuff like "play 3 frames now" or "move forward 10 frames" or "play as fast as you can but play all frames". It doesn't even know what a "frame" is. There is no privileged elementary stream, like the video one could be (for the simple reason that, according to MPEG, a stream may contain several video ES).

What happens to a file

An input thread is spawned for every file read. Indeed, input structures and decoders need to be reinitialized because the specificities of the stream may be different. `input_CreateThread` is called by the interface thread (playlist module).

At first, an input plug-in capable of reading the plugin item is looked for [this is inappropriate : we should first open the socket, and then probe the beginning of the stream to see which plug-in can read it]. The socket is opened by either `input_FileOpen`, `input_NetworkOpen`, or `input_DvdOpen`. This function sets two very important parameters : `b_pace_control` and `b_seekable` (see next section).

> **Note:** We could use so-called "access" plugins for this whole mechanism of opening the input socket. This is not the case because we thought only those three methods were to be used at present, and if we need others we can still build them in.

Now we can launch the input plugin's `pf_init` function, and an endless loop doing `pf_read` and `pf_demux`. The plugin is responsible for initializing the stream structures (*`p_input->stream`*), managing packet buffers, reading packets and demultiplex them. But in most tasks it will be assisted by functions from the advanced input API (c). That is what we will study in the coming sections !

Stream Management

The function which has opened the input socket must specify two properties about it :

1. *p_input->stream.b_pace_control* : Whether or not the stream can be read at our own pace (determined by the stream's frequency and the host computer's system clock). For instance a file or a pipe (including TCP/IP connections) can be read at our pace, if we don't read fast enough, the other end of the pipe will just block on a `write()` operation. On the contrary, UDP streaming (such as the one used by VideoLAN Server) is done at the server's pace, and if we don't read fast enough, packets will simply be lost when the kernel's buffer is full. So the drift introduced by the server's clock must be regularly compensated. This property controls the clock management, and whether or not fast forward and slow motion can be done.

   > **Subtilities in the clock management:** With a UDP socket and a distant server, the drift is not negligible because on a whole movie it can account for seconds if one of the clocks is slightly fucked up. That means that presentation dates given by the input thread may be out of sync, to some extent, with the frequencies given in every Elementary Stream. Output threads (and, anecdotically, decoder threads) must deal with it.
   >
   > The same kind of problems may happen when reading from a device (like video4linux's `/dev/video`) connected for instance to a video encoding board. There is no way we could differentiate it from a simple **cat foo.mpg | vlc -** , which doesn't imply any clock problem. So the Right Thing (c) would be to ask the user about the value of `b_pace_control` , but nobody would understand what it means (you are not the dumbest person on Earth, and

obviously you have read this paragraph several times to understand it :-). Anyway, the drift should be negligible since the board would share the same clock as the CPU, so we chose to neglect it.

2. *p_input->stream.b_seekable* : Whether we can do `lseek()` calls on the file descriptor or not. Basically whether we can jump anywhere in the stream (and thus display a scrollbar) or if we can only read one byte after the other. This has less impact on the stream management than the previous item, but it is not redundant, because for instance **cat foo.mpg | vlc -** is b_pace_control = 1 but b_seekable = 0. On the contrary, you cannot have b_pace_control = 0 along with b_seekable = 1. If a stream is seekable, *`p_input->stream.p_selected_area->i_size`* must be set (in an arbitrary unit, for instance bytes, but it must be the same as p_input->i_tell which indicates the byte we are currently reading from the stream).

Offset to time conversions: Functions managing clocks are located in `src/input/input_clock.c`. All we know about a file is its start offset and its end offset (*`p_input->stream.p_selected_area->i_size`*), currently in bytes, but it could be plugin-dependant. So how the hell can we display in the interface a time in seconds ? Well, we cheat. PS streams have a *`mux_rate`* property which indicates how many bytes we should read in a second. This is subject to change at any time, but practically it is a constant for all streams we know. So we use it to determine time offsets.

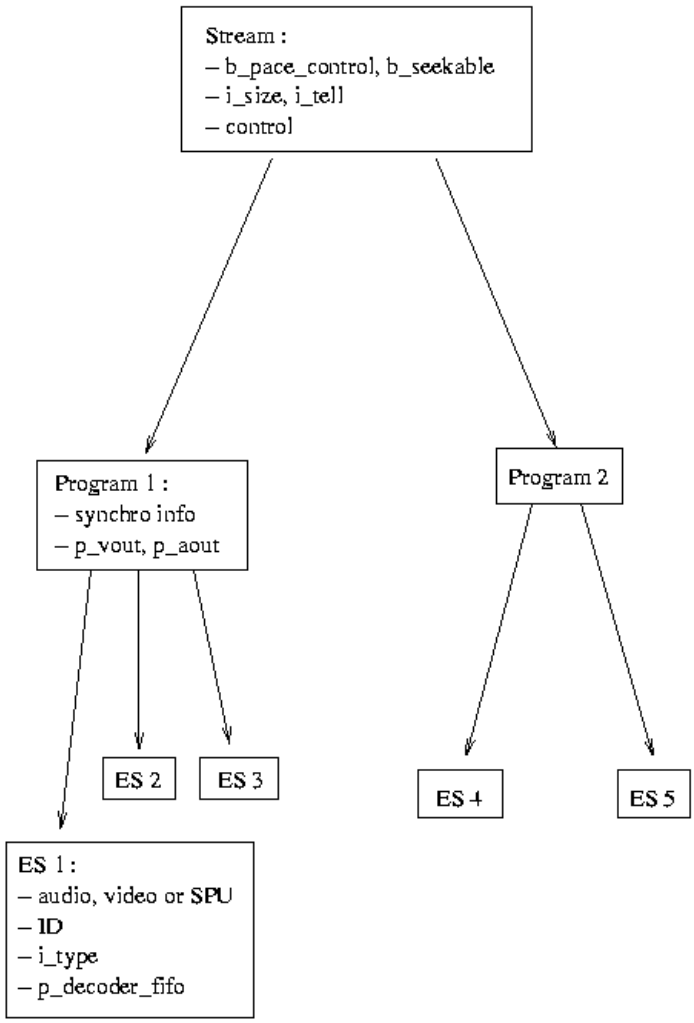
Structures exported to the interface

Let's focus on the communication API between the input module and the interface. The most important file is `include/input_ext-intf.h`, which you should know almost by heart. This file defines the input_thread_t structure, the stream_descriptor_t and all programs and ES descriptors included (you can view it as a tree).

First, note that the input_thread_t structure features two void * pointers, *`p_method_data`* and *`p_plugin_data`*, which you can respectivly use for buffer management data and plugin data.

Second, a stream description is stored in a tree featuring program descriptors, which themselves contain several elementary stream descriptors. For those of you who don't know all MPEG concepts, an elementary stream, aka ES, is a continuous stream of video or (exclusive) audio data, directly readable by a decoder, without decapsulation.

This tree structure is illustrated by the following figure, where one stream holds two programs. In most cases there will only be one program (to my knowledge only TS streams can carry several programs, for instance a movie and a football game at the same time - this is adequate for satellite and cable broadcasting).



p_input->stream : The stream, programs and elementary streams can be viewed as a tree.

> **Warning**
>
> For all modifications and accesses to the `p_input->stream` structure, you *must* hold the p_input->stream.stream_lock.

ES are described by an ID (the ID the appropriate demultiplexer will look for), a `stream_id` (the real MPEG stream ID), a type (defined in ISO/IEC 13818-1 table 2-29) and a litteral description. It also contains context information for the demultiplexer, and decoder information `p_decoder_fifo` we will talk about in the next chapter. If the stream you want to read is not an MPEG system layer (for instance AVI or RTP), a specific demultiplexer will have to be written. In that case, if you need to carry additional information, you can use void * `p_demux_data` at your convenience. It will be automatically freed on shutdown.

Why ID and not use the plain MPEG `stream_id` ?: When a packet (be it a TS packet, PS packet, or whatever) is read, the appropriate demultiplexer will look for an ID in the packet, find the relevant elementary stream, and demultiplex it if the user selected it. In case of TS packets, the only information we have is the ES PID, so the reference ID we keep is the PID. PID don't exist in PS streams, so we have to invent one. It is of course based on the `stream_id` found in all PS packets, but it is not enough, since private streams (ie. AC3, SPU and LPCM) all share the same `stream_id` (`0xBD`). In that case the first byte of the PES payload is a stream private ID, so we combine this with the stream_id to get our ID (if you did not understand everything, it isn't very important - just remember we used our brains before writing the code :-).

The stream, program and ES structures are filled in by the plugin's `pf_init()` using functions in `src/input/input_programs.c`, but are subject to change at any time. The DVD plugin parses .ifo files to know which ES are in the stream; the TS plugin reads the PAT and PMT structures in the stream; the PS plugin can either parse the PSM structure (but it is rarely present), or build the tree "on the fly" by pre-parsing the first megabyte of data.

> **Warning**
>
> In most cases we need to pre-parse (that is, read the first MB of data, and go back to the beginning) a PS stream, because the PSM (Program Stream Map) structure is almost never present. This is not appropriate, though, but we don't have the choice. A few problems will arise. First, non-seekable streams cannot be pre-parsed, so the ES tree will be built on the fly. Second, if a new elementary stream starts after the first MB of data (for instance a subtitle track won't show up during the credits), it won't appear in the menu before we encounter the first packet. We cannot pre-parse the entire stream because it would take hours (even without decoding it).

It is currently the responsibility of the input plugin to spawn the necessary decoder threads. It must call `input_SelectES ( input_thread_t * p_input, es_descriptor_t * p_es )` on the selected ES.

The stream descriptor also contains a list of areas. Areas are logical discontinuities in the stream, for instance chapters and titles in a DVD. There is only one area in TS and PS streams, though we could use them when the PSM (or PAT/PMT) version changes. The goal is that when you seek to another area, the input plugin loads the new stream descriptor tree (otherwise the selected ID may be wrong).

Methods used by the interface

Besides, `input_ext-intf.c` provides a few functions to control the reading of the stream :

- `input_SetStatus ( input_thread_t * p_input, int i_mode )` : Changes the pace of reading. `i_mode` can be one of `INPUT_STATUS_END, INPUT_STATUS_PLAY, INPUT_STATUS_PAUSE, INPUT_STATUS_FASTER, INPUT_STATUS_SLOWER.`

  Note: Internally, the pace of reading is determined by the variable `p_input->stream.control.i_rate`. The default value is `DEFAULT_RATE`. The lower the value, the faster the pace is. Rate changes are taken into account in `input_ClockManageRef`. Pause is accomplished by simply stopping the input thread (it is then awaken by a pthread signal). In that case, decoders will be stopped too. Please remember this if you do statistics on decoding times (like `src/video_parser/vpar_synchro.c` does). Don't call this function if `p_input->b_pace_control` == 0.

- `input_Seek ( input_thread_t * p_input, off_t i_position )` : Changes the offset of reading. Used to jump to another place in a file. You *mustn't* call this function if `p_input->stream.b_seekable` == 0. The position is a number (usually long long, depends on your libc) between `p_input->p_selected_area->i_start` and `p_input->p_selected_area->i_size` (current value is in `p_input->p_selected_area->i_tell`).

  Note: Multimedia files can be very large, especially when we read a device like `/dev/dvd`, so offsets must be 64 bits large. Under a lot of systems, like FreeBSD, off_t are 64 bits by default, but it is not the case under GNU libc 2.x. That is why we need to compile VLC with -D_FILE_OFFSET_BITS=64 -D__USE_UNIX98.

  Escaping stream discontinuities: Changing the reading position at random can result in a messed up stream, and the decoder which reads it may segfault. To avoid this, we send several NULL packets (ie. packets containing

nothing but zeros) before changing the reading position. Indeed, under most video and audio formats, a long enough stream of zeros is an escape sequence and the decoder can exit cleanly.

- `input_OffsetToTime ( input_thread_t * p_input, char * psz_buffer, off_t i_offset )` : Converts an offset value to a time coordinate (used for interface display). [currently it is broken with MPEG-2 files]
- `input_ChangeES ( input_thread_t * p_input, es_descriptor_t * p_es, u8 i_cat )` : Unselects all elementary streams of type `i_cat` and selects `p_es`. Used for instance to change language or subtitle track.
- `input_ToggleES ( input_thread_t * p_input, es_descriptor_t * p_es, boolean_t b_select )` : This is the clean way to select or unselect a particular elementary stream from the interface.

Buffers management

Input plugins must implement a way to allocate and deallocate packets (whose structures will be described in the next chapter). We basically need four functions :

- `pf_new_packet ( void * p_private_data, size_t i_buffer_size )` : Allocates a new data_packet_t and an associated buffer of i_buffer_size bytes.
- `pf_new_pes ( void * p_private_data )` : Allocates a new pes_packet_t.
- `pf_delete_packet ( void * p_private_data, data_packet_t * p_data )` : Deallocates `p_data`.
- `pf_delete_pes ( void * p_private_data, pes_packet_t * p_pes )` : Deallocates `p_pes`.

All functions are given `p_input->p_method_data` as first parameter, so that you can keep records of allocated and freed packets.

Buffers management strategies: Buffers management can be done in three ways :

1. *Traditional libc allocation* : For a long time we have used in the PS plugin `malloc()` and `free()` every time we needed to allocate or deallocate a packet. Contrary to a popular belief, it is not *that* slow.
2. *Netlist* : In this method we allocate a very big buffer at the beginning of the problem, and then manage a list of pointers to free packets (the "netlist"). This only works well if all packets have the same size. It is used for long for the TS input. The DVD plugin also uses it, but adds a *refcount* flag because buffers (2048 bytes) can be shared among several packets. It is now deprecated and won't be documented.
3. *Buffer cache* : We are currently developing a new method. It is already in use in the PS plugin. The idea is to call `malloc()` and `free()` to absorb stream irregularities, but re-use all allocated buffers via a cache system. We are extending it so that it can be used in any plugin without performance hit, but it is currently left undocumented.

Demultiplexing the stream

After being read by `pf_read` , your plugin must give a function pointer to the demultiplexer function. The demultiplexer is responsible for parsing the packet, gathering PES, and feeding decoders.

Demultiplexers for standard MPEG structures (PS and TS) have already been written. You just need to indicate `input_DemuxPS` and `input_DemuxTS` for `pf_demux`. You can also write your own demultiplexer.

It is not the purpose of this document to describe the different levels of encapsulation in an MPEG stream. Please refer to your MPEG specification for that.

Chapter 4. How to write a decoder

What is precisely a decoder in the VLC scheme ?

The decoder does the mathematical part of the process of playing a stream. It is separated from the demultiplexers (in the input module), which manage packets to rebuild a continuous elementary stream, and from the output thread, which takes samples reconstituted by the decoder and plays them. Basically, a decoder has no interaction with devices, it is purely algorithmic.

In the next section we will describe how the decoder retrieves the stream from the input. The output API (how to say "this sample is decoded and can be played at xx") will be talked about in the next chapters.

Decoder configuration

The input thread spawns the appropriate decoder modules from `src/input/input_dec.c`. The `Dec_CreateThread` function selects the more accurate decoder module. Each decoder module looks at decoder_config.i_type and returns a score [see the modules section]. It then launches `module.pf_run()`, with a decoder_config_t, described in `include/input_ext-dec.h`.

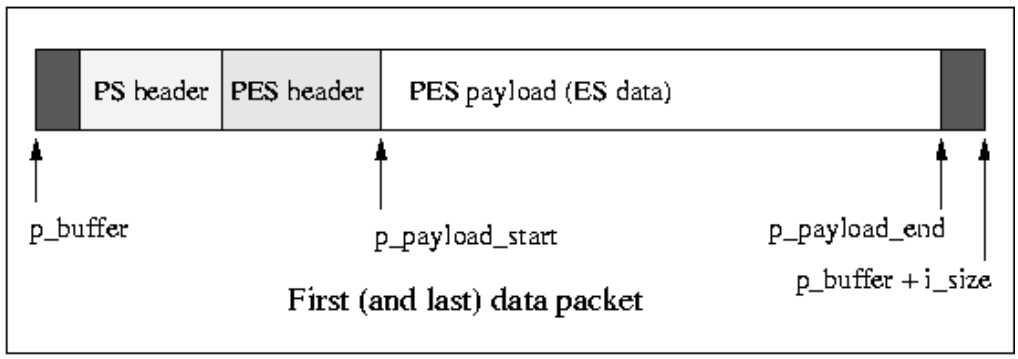
The generic decoder_config_t structure, gives the decoder the ES ID and type, and pointers to a stream_control_t structure (gives information on the play status), a decoder_fifo_t and *`pf_init_bit_stream`*, which will be described in the next two sections.

Packet structures

The input module provides an advanced API for delivering stream data to the decoders. First let's have a look at the packet structures. They are defined in `include/input_ext-dec.h`.

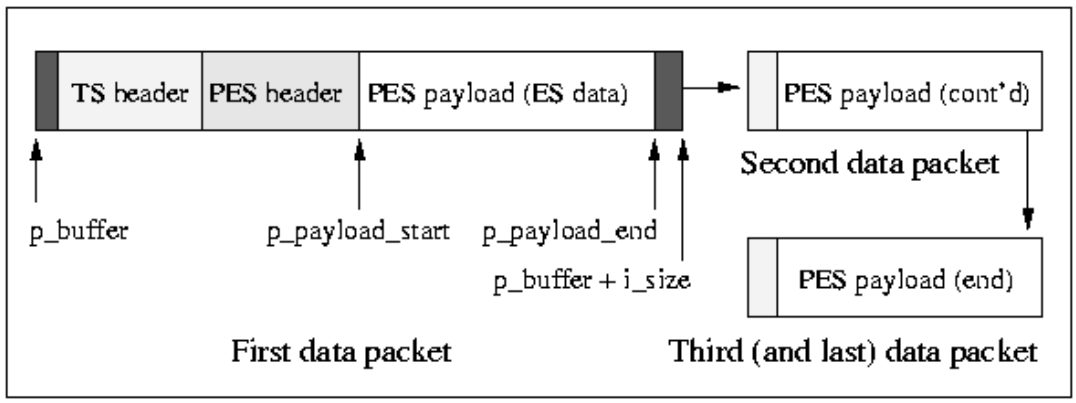
data_packet_t contains a pointer to the physical location of data. Decoders should only start to read them at *`p_payload_start`* until *`p_payload_end`*. Thereafter, it will switch to the next packet, *`p_next`* if it is not `NULL`. If the *`b_discard_payload`* flag is up, the content of the packet is messed up and it should be discarded.

data_packet_t are contained into pes_packet_t. pes_packet_t features a chained list (*`p_first`*) of data_packet_t representing (in the MPEG paradigm) a complete PES packet. For PS streams, a pes_packet_t usually only contains one data_packet_t. In TS streams though, one PES can be split among dozens of TS packets. A PES packet has PTS dates (see your MPEG specification for more information) and the current pace of reading that should be applied for interpolating dates (*`i_rate`*). *`b_data_alignment`* (if available in the system layer) indicates if the packet is a random access point, and *`b_discontinuity`* tells whether previous packets have been dropped.



PES packet

In a Program Stream, a PES packet features only one data packet, whose buffer contains the PS header, the PES header, and the data payload.



PES packet

In a Transport Stream, a PES packet can feature an unlimited number of data packets (three on the figure) whose buffers contains the PS header, the PES header, and the data payload.

The structure shared by both the input and the decoder is decoder_fifo_t. It features a rotative FIFO of PES packets to be decoded. The input provides macros to manipulate it : `DECODER_FIFO_ISEMPTY, DECODER_FIFO_ISFULL, DECODER_FIFO_START, DECODER_FIFO_INCSTART, DECODER_FIFO_END, DECODER_FIFO_INCEND`. Please remember to take *`p_decoder_fifo->data_lock`* before any operation on the FIFO.

The next packet to be decoded is DECODER_FIFO_START(*p_decoder_fifo). When it is finished, you need to call `p_decoder_fifo->pf_delete_pes( p_decoder_fifo->p_packets_mgt, DECODER_FIFO_START( *p_decoder_fifo ) )` and then `DECODER_FIFO_INCSTART( *p_decoder_fifo )` to return the PES to the buffer manager.

If the FIFO is empty (`DECODER_FIFO_ISEMPTY`), you can block until a new packet is received with a cond signal : `vlc_cond_wait( &p_fifo->data_wait, &p_fifo->data_lock )`. You have to hold the lock before entering this function. If the file is over or the user quits, *`p_fifo->b_die`* will be set to 1. It indicates that you must free all your data structures and call `vlc_thread_exit()` as soon as possible.

The bit stream (input module)

This classical way of reading packets is not convenient, though, since the elementary stream can be split up arbitrarily. The input module provides primitives which make reading a bit stream much easier. Whether you use it or not is at your option, though if you use it you shouldn't access the packet buffer any longer.

The bit stream allows you to just call `GetBits()`, and this functions will transparently read the packet buffers, change data packets and pes packets when necessary, without any intervention from you. So it is much more convenient for you to read a continuous Elementary Stream, you don't have to deal with packet boundaries and the FIFO, the bit stream will do it for you.

The central idea is to introduce a buffer of 32 bits [normally WORD_TYPE, but 64-bit version doesn't work yet], bit_fifo_t. It contains the word buffer and the number of significant bits (higher part). The input module provides five inline functions to manage it :

- u32 `GetBits` *`( bit_stream_t * p_bit_stream, unsigned int i_bits )`* : Returns the next *`i_bits`* bits from the bit buffer. If there are not enough bits, it fetches the following word from the decoder_fifo_t. This function is only guaranteed to work with up to 24 bits. For the moment it works until 31 bits, but it is a side effect. We were obliged to write a different function, `GetBits32`, for 32-bit reading, because of the << operator.
- `RemoveBits` *`( bit_stream_t * p_bit_stream, unsigned int i_bits )`* : The same as `GetBits()`, except that the bits aren't returned (we spare a few CPU cycles). It has the same limitations, and we also wrote `RemoveBits32`.
- u32 `ShowBits` *`( bit_stream_t * p_bit_stream, unsigned int i_bits )`* : The same as `GetBits()`, except that the bits don't get flushed after reading, so that you need to call `RemoveBits()` by hand afterwards. Beware, this function won't work above 24 bits, except if you're aligned on a byte boundary (see next function).
- `RealignBits` *`( bit_stream_t * p_bit_stream )`* : Drops the n higher bits (n < 8), so that the first bit of the buffer be aligned an a byte boundary. It is useful when looking for an aligned startcode (MPEG for instance).
- `GetChunk` *`( bit_stream_t * p_bit_stream, byte_t * p_buffer, size_t i_buf_len )`* : It is an analog of `memcpy()`, but taking a bit stream as first argument. *`p_buffer`* must be allocated and at least *`i_buf_len`* long. It is useful to copy data you want to keep track of.

All these functions recreate a continuous elementary stream paradigm. When the bit buffer is empty, they take the following word in the current packet. When the packet is empty, it switches to the next data_packet_t, or if unapplicable to the next pes_packet_t (see `p_bit_stream->pf_next_data_packet`). All this is completely transparent.

> **Packet changes and alignment issues:** We have to study the conjunction of two problems. First, a data_packet_t can have an even number of bytes, for instance 177, so the last word will be truncated. Second, many CPU (sparc, alpha...) can only read words aligned on a word boundary (that is, 32 bits for a 32-bit word). So packet changes are a lot more complicated than you can imagine, because we have to read truncated words and get aligned.
>
> For instance `GetBits()` will call `UnalignedGetBits()` from `src/input/input_ext-dec.c`. Basically it will read byte after byte until the stream gets realigned. `UnalignedShowBits()` is a bit more complicated and may require a temporary packet (*`p_bit_stream->showbits_data`*).

To use the bit stream, you have to call *`p_decoder_config->pf_init_bit_stream( bit_stream_t * p_bit_stream, decoder_fifo_t * p_fifo )`* to set up all variables. You will probably need to regularly fetch specific information from the packet, for instance the PTS. If *`p_bit_stream->pf_bit_stream_callback`* is not `NULL`, it will be called on a packet change. See `src/video_parser/video_parser.c` for an example. The second argument indicates whether it is just a new data_packet_t or also a new pes_packet_t. You can store your own structure in *`p_bit_stream->p_callback_arg`*.

Warning

When you call `pf_init_bit_stream`, the `pf_bitstream_callback` is not defined yet, but it jumps to the first packet, though. You will probably want to call your bitstream callback by hand just after `pf_init_bit_stream`.

Built-in decoders

VLC already features an MPEG layer 1 and 2 audio decoder, an MPEG MP@ML video decoder, an AC3 decoder (borrowed from LiViD), a DVD SPU decoder, and an LPCM decoder. You can write your own decoder, just mimic the video parser.

> **Limitations in the current design:** To add a new decoder, you'll still have to add the stream type as there's still a a hard-wired piece of code in `src/input/input_programs.c` .

The MPEG audio decoder is native, but doesn't support layer 3 decoding [too much trouble], the AC3 decoder is a port from Aaron Holtzman's libac3 (the original libac3 isn't reentrant), and the SPU decoder is native. You may want to have a look at `BitstreamCallback` in the AC3 decoder. In that case we have to jump the first 3 bytes of a PES packet, which are not part of the elementary stream. The video decoder is a bit special and will be described in the following section.

The MPEG video decoder

VLC media player provides an MPEG-1, and an MPEG-2 Main Profile @ Main Level decoder. It has been natively written for VLC, and is quite mature. Its status is a bit special, since it is splitted between two logicial entities : video parser and video decoder. The initial goal is to separate bit stream parsing functions from highly parallelizable mathematical algorithms. In theory, there can be one video parser thread (and only one, otherwise we would have race conditions reading the bit stream), along with a pool of video decoder threads, which do IDCT and motion compensation on several blocks at once.

It doesn't (and won't) support MPEG-4 or DivX decoding. It is not an encoder. It should support the whole MPEG-2 MP@ML specification, though some features are still left untested, like Differential Motion Vectors. Please bear in mind before complaining that the input elementary stream must be valid (for instance this is not the case when you directly read a DVD multi-angle .vob file).

The most interesting file is `vpar_synchro.c`, it is really worth the shot. It explains the whole frame dropping algorithm. In a nutshell, if the machine is powerful enough, we decoder all IPBs, otherwise we decode all IPs and Bs if we have enough time (this is based on on-the-fly decoding time statistics). Another interesting file is `vpar_blocks.c`, which describes all block (including coefficients and motion vectors) parsing algorithms. Look at the bottom of the file, we indeed generate one optimized function for every common picture type, and one slow generic function. There are also several levels of optimization (which makes compilation slower but certain types of files faster decoded) called `VPAR_OPTIM_LEVEL`, level 0 means no optimization, level 1 means optimizations for MPEG-1 and MPEG-2 frame pictures, level 2 means optimizations for MPEG-1 and MPEG-2 field and frame pictures.

Motion compensation plug-ins

Motion compensation (i.e. copy of regions from a reference picture) is very platform-dependant (for instance with MMX or AltiVec versions), so we moved it to the `plugins/motion` directory. It is more convenient for the video decoder, and resulting plug-ins may be used by other video decoders (MPEG-4 ?). A motion plugin must define 6 functions, coming straight from the specification : `vdec_MotionFieldField420, vdec_MotionField16x8420, vdec_MotionFieldDMV420, vdec_MotionFrameFrame420, vdec_MotionFrameField420, vdec_MotionFrameDMV420`. The equivalent 4:2:2 and 4:4:4 functions are unused, since these formats are forbidden in MP@ML (it would only take longer compilation time).

Look at the C version of the algorithms if you want more information. Note also that the DMV algorithm is untested and is probably buggy.

IDCT plug-ins

Just like motion compensation, IDCT is platform-specific. So we moved it to `plugins/idct`. This module does the IDCT calculation, and copies the data to the final picture. You need to define seven methods :

- `vdec_IDCT` *`( decoder_config_t * p_config, dctelem_t * p_block, int )`* : Does the complete 2-D IDCT. 64 coefficients are in *`p_block`*.
- `vdec_SparseIDCT` *`( vdec_thread_t * p_vdec, dctelem_t * p_block, int i_sparse_pos )`* : Does an IDCT on a block with only one non-NULL coefficient (designated by *`i_sparse_pos`*). You can use the function defined in `plugins/idct/idct_common.c` which precalculates these 64 matrices at initialization time.
- `vdec_InitIDCT` *`( vdec_thread_t * p_vdec )`* : Does the initialization stuff needed by `vdec_SparseIDCT`.
- `vdec_NormScan` *`( u8 ppi_scan[2][64] )`* : Normally, this function does nothing. For minor optimizations, some IDCT (MMX) need to invert certain coefficients in the MPEG scan matrices (see ISO/IEC 13818-2).
- `vdec_InitDecode` *`( struct vdec_thread_s * p_vdec )`* : Initializes the IDCT and optional crop tables.
- `vdec_DecodeMacroblockC` *`( struct vdec_thread_s *p_vdec, struct macroblock_s * p_mb );`* : Decodes an entire macroblock and copies its data to the final picture, including chromatic information.
- `vdec_DecodeMacroblockBW` *`( struct vdec_thread_s *p_vdec, struct macroblock_s * p_mb );`* : Decodes an entire macroblock and copies its data to the final picture, except chromatic information (used in grayscale mode).

Currently we have implemented optimized versions for : MMX, MMXEXT, and AltiVec [doesn’t work]. We have two plain C versions, the normal (supposedly optimized) Berkeley version (`idct.c`), and the simple 1-D separation IDCT from the ISO reference decoder (`idctclassic.c`).

Symmetrical Multiprocessing

The MPEG video decoder of VLC can take advantage of several processors if necessary. The idea is to launch a pool of decoders, which will do IDCT/motion compensation on several macroblocks at once.

The functions managing the pool are in `src/video_decoder/vpar_pool.c`. Its use on non-SMP machines is not recommanded, since it is actually slower than the monothread version. Even on SMP machines sometimes...

Chapter 5. The video output layer

Data structures and main loop

Important data structures are defined in `include/video.h` and `include/video_output.h`. The main data structure is picture_t, which describes everything a video decoder thread needs. Please refer to this file for more information. Typically, *`p_data`* will be a pointer to YUV planar picture.

Note also the subpicture_t structure. In fact the VLC SPU decoder only parses the SPU header, and converts the SPU graphical data to an internal format which can be rendered much faster. So a part of the "real" SPU decoder lies in `src/video_output/video_spu.c`.

The vout_thread_t structure is much more complex, but you needn't understand everything. Basically the video output thread manages a heap of pictures and subpictures (5 by default). Every picture has a status (displayed, destroyed, empty...) and eventually a presentation time. The main job of the video output is an infinite loop to : [this is subject to change in the near future]

1. Find the next picture to display in the heap.
2. Find the current subpicture to display.
3. Render the picture (if the video output plug-in doesn't support YUV overlay). Rendering will call an optimized YUV plug-in, which will also do the scaling, add subtitles and an optional picture information field.
4. Sleep until the specified date.
5. Display the picture (plug-in function). For outputs which display RGB data, it is often accomplished with a buffer switching. *`p_vout->p_buffer`* is an array of two buffers where the YUV transform takes place, and p_vout->i_buffer_index indicates the currently displayed buffer.
6. Manage events.

Methods used by video decoders

The video output exports a bunch of functions so that decoders can send their decoded data. The most important function is `vout_CreatePicture` which allocates the picture buffer to the size indicated by the video decoder. It then just needs to feed (void *) *`p_picture->p_data`* with the decoded data, and call `vout_DisplayPicture` and `vout_DatePicture` upon necessary.

- picture_t * `vout_CreatePicture` *`( vout_thread_t *p_vout, int i_type, int i_width, int i_height )`* : Returns an allocated picture buffer. *`i_type`* will be for instance `YUV_420_PICTURE`, and i_width and i_height are in pixels.

> **Warning**
>
> If no picture is available in the heap, `vout_CreatePicture` will return NULL.

- `vout_LinkPicture` *`( vout_thread_t *p_vout, picture_t *p_pic )`* : Increases the refcount of the picture, so that it doesn't get accidently freed while the decoder still needs it. For instance, an I or P picture can still be needed after displaying to decode interleaved B pictures.
- `vout_UnlinkPicture` *`( vout_thread_t *p_vout, picture_t *p_pic )`* : Decreases the refcount of the picture. An unlink must be done for every link previously made.

- vout_DatePicture (vout_thread_t *p_vout, picture_t *p_pic) : Gives the picture a presentation date. You can start working on a picture before knowing precisely at what time it will be displayed. For instance to date an I or P picture, you must wait until you have decoded all previous B pictures (which are indeed placed after - decoding order != presentation order).
- vout_DisplayPicture (vout_thread_t *p_vout, picture_t *p_pic) : Tells the video output that a picture has been completely decoded and is ready to be rendered. It can be called before or after vout_DatePicture.
- vout_DestroyPicture (vout_thread_t *p_vout, picture_t *p_pic) : Marks the picture as empty (useful in case of a stream parsing error).
- subpicture_t * vout_CreateSubPicture (vout_thread_t *p_vout, int i_channel, int i_type) : Returns an allocated subpicture buffer. i_channel is the ID of the subpicture channel, i_type is DVD_SUBPICTURE or TEXT_SUBPICTURE, i_size is the length in bytes of the packet.
- vout_DisplaySubPicture (vout_thread_t *p_vout, subpicture_t *p_subpic) : Tells the video output that a subpicture has been completely decoded. It obsoletes the previous subpicture.
- vout_DestroySubPicture (vout_thread_t *p_vout, subpicture_t *p_subpic) : Marks the subpicture as empty.

How to write a video output plug-in

A video output takes care of the system calls to display the pictures and manage the output window. Have a look at plugins/x11/vout_x11.c. You must write the following functions :

1. int vout_Probe (probedata_t *p_data) : Returns a score between 0 and 999 to indicate whether it can run on the architecture. 999 is the best. p_data is currently unused.
2. int vout_Create (vout_thread_t *p_vout) : Basically, initializes and opens a new window. Returns TRUE if it failed.
3. int vout_Init (vout_thread_t *p_vout) : Creates optional picture buffers (for instance ximages or xvimages). Returns TRUE if it failed.
4. vout_End (vout_thread_t *p_vout) : Frees optional picture buffers.
5. vout_Destroy (vout_thread_t *p_vout) : Unmaps the window and frees all allocated resources.
6. int vout_Manage (vout_thread_t *p_vout) : Manages events (including for instance resize events).
7. vout_Display (vout_thread_t *p_vout) : Displays a previously rendered buffer.
8. vout_SetPalette (vout_thread_t *p_vout, u16 *red, u16 *green, u16 *blue, u16 *transp) : Sets the 8 bpp palette. red, green and blue are arrays of 256 unsigned shorts.

How to write a YUV plug-in

Look at the C source plugins/yuv/transforms_yuv.c. You need to redefine just the same transformations. Basically, it is a matrix multiply operation. Good luck.

Chapter 6. The audio output layer

Audio output overview

This chapter documents the audio output layer known under the "audio output 3" codename. It has first been released with VLC version 0.5.0. Previous versions use an antic API, which is no longer documented nor supported. You definitely should write new code only for aout3 and later.

The audio output's main purpose is to take sound samples from one or several decoders (called "input streams" in this chapter), to mix them and write them to an output device (called "output stream"). During this process, transformations may be needed or asked by the user, and they will be performed by audio filters.

(insert here a schematic of the data flow in aout3)

Terminology

- *Sample* : A sample is an elementary piece of audio information, containing the value for all channels. For instance, a stream at 44100 Hz features 44100 samples per second, no matter how many channels are coded, nor the coding type of the coefficients.
- *Frame* : A set of samples of arbitrary size. Codecs usually have a fixed frame size (for instance an A/52 frame contains 1536 samples). Frames do not have much importance in the audio output, since it can manage buffers of arbitrary sizes. However, for undecoded formats, the developer must indicate the number of bytes required to carry a frame of n samples, since it depends on the compression ratio of the stream.
- *Coefficient* : A sample contains one coefficient per channel. For instance a stereo stream features 2 coefficients per sample. Many audio items (such as the float32 audio mixer) deal directly with the coefficients. Of course, an undecoded sample format doesn't have the notion of "coefficient", since a sample cannot be materialized independantly in the stream.
- *Resampling* : Changing the number of samples per second of an audio stream.
- *Downmixing/upmixing* : Changing the configuration of the channels (see below).

Audio sample formats

The whole audio output can viewed as a pipeline transforming one audio format to another in successive steps. Consequently, it is essential to understand what an audio sample format is.

The audio_sample_format_t structure is defined in include/audio_output.h. It contains the following members :

- *i_format* : Define the format of the coefficients. This is a FOURCC field. For instance 'fl32' (float32), 'fi32' (fixed32), 's16b' (signed 16-bit big endian), 's16l' (signed 16-bit little endian), AOUT_FMT_S16_NE (shortcut to either 's16b' or 's16l'), 'u16b', 'u16l','s8 ', 'u8 ', 'ac3 ', 'spdi' (S/PDIF). Undecoded sample formats include 'a52 ', 'dts ', 'spdi', 'mpga' (MPEG audio layer I and II), 'mpg3' (MPEG audio layer III). An audio filter allowing to go from one format to another is called, by definition, a "converter". Some converters play the role of a decoder (for instance a52tofloat32.c), but are in fact "audio filters".
- *i_rate* : Define the number of samples per second the audio output will have to deal with. Common values are 22050, 24000, 44100, 48000. i_rate is in Hz.

- *i_physical_channels* : Define the channels which are physically encoded in the buffer. This field is a bitmask of values defined in audio_output.h, for instance AOUT_CHAN_CENTER, AOUT_CHAN_LEFT, etc. Beware : the numeric value doesn't represent the number of coefficients per sample, see aout_FormatNbChannels() for that. The coefficients for each channel are always stored interleaved, because it is much easier for the mixer to deal with interleaved coefficients. Consequently, decoders which output planar data must implement an interleaving function. Coefficients must be output in the following order (WG-4 specification) : left, right, left surround, right surround, center, LFE.
- *i_original_channels* : Define the channels from the original stream which have been used to constitute a buffer. For instance, imagine your output plug-ins only has mono output (AOUT_CHAN_CENTER), and your stream is stereo. You can either use both channels of the stream (i_original_channels == AOUT_CHAN_LEFT | AOUT_CHAN_RIGHT), or select one of them. i_original_channels uses the same bitmask as i_physical_channels, and also features special bits AOUT_CHAN_DOLBYSTEREO, which indicates whether the input stream is downmixed to Dolby surround sound, and AOUT_CHAN_DUALMONO, which indicates that the stereo stream is actually constituted of two mono streams, and only one of them should be selected (for instance, two languages on one VCD).

> **Note:** For 16-bit integer format types, we make a distinction between big-endian and little-endian storage types. However, floats are also stored in either big endian or little endian formats, and we didn't make a difference. The reason is, samples are hardly stored in float32 format in a file, and transferred from one machine to another ; so we assume float32 always use the native endianness.
>
> Yet, samples are quite often stored as big-endian signed 16-bit integers, such as in DVD's LPCM format. So the LPCM decoder allocates an 's16b' input stream, and on little-endian machines, an 's16b'->'s16l' converter is automatically invoked by the input pipeline.
>
> In most cases though, AOUT_FMT_S16_NE and AOUT_FMT_U16_NE should be used.

The aout core provides macros to compare two audio sample formats. AOUT_FMTS_IDENTICAL() tests if i_format, i_rate, i_physical_channels and i_original_channels are identical. AOUT_FMTS_SIMILAR tests if i_rate and i_channels are identical (useful to write a pure converter filter).

The audio_sample_format_t structure then contains two additional parameters, which you are not supposed to write directly, except if you're dealing with undecoded formats. For PCM formats they are automatically filled in by aout_FormatPrepare(), which is called by the core functions when necessary.

- *i_frame_length* : Define the number of samples of the "natural" frame. For instance for A/52 it is 1536, since 1536 samples are compressed in an undecoded buffer. For PCM formats, the frame size is 1, because every sample in the buffer can be independantly accessed.
- *i_bytes_per_frame* : Define the size (in bytes) of a frame. For A/52 it depends on the bitrate of the input stream (read in the sync info). For instance for stereo float32 samples, i_bytes_per_frame == 8 (i_frame_length == 1).

These last two fields (which are *always* meaningful as soon as aout_FormatPrepare() has been called) make it easy to calculate the size of an audio buffer : i_nb_samples * i_bytes_per_frame / i_frame_length.

Typical runcourse

The input spawns a new audio decoder, say for instance an A/52 decoder. The A/52 decoder parses the sync info for format information (eg. it finds 48 kHz, 5.1, 196 kbi/s), and creates a new aout "input stream" with aout_InputNew(). The sample format is :

- i_format = 'a52 '
- i_rate = 48000

- i_physical_channels = i_original_channels = AOUT_CHAN_LEFT | AOUT_CHAN_RIGHT | AOUT_CHAN_CENTER | AOUT_CHAN_REARLEFT | AOUT_CHAN_REARRIGHT | AOUT_CHAN_LFE
- i_frame_length = 1536
- i_bytes_per_frame = 24000

This input format won't be modified, and will be stored in the aout_input_t structure corresponding to this input stream : p_aout->pp_inputs[0]->input. Since it is our first input stream, the aout core will try to configure the output device with this audio sample format (p_aout->output.output), to avoid unnecessary transformations.

The core will probe for an output module in the usual fashion, and its behavior will depend. Either the output device has the S/PDIF capability, and then it will set p_aout->output.output.i_format to 'spdi', or it's a PCM-only device. It will thus ask for the native sample format, such as 'fl32' (for Darwin CoreAudio) or AOUT_FMT_S16_NE (for OSS). The output device may also have constraints on the number of channels or the rate. For instance, the p_aout->output.output structure may look like :

- i_format = AOUT_FMT_S16_NE
- i_rate = 44100
- i_channels = AOUT_CHAN_LEFT | AOUT_CHAN_RIGHT
- i_frame_length = 1
- i_bytes_per_frame = 4

Once we have an output format, we deduce the mixer format. It is strictly forbidden to change the audio sample format between the mixer and the output (because all transformations happen in the input pipeline), except for i_format. The reason is that we have only developed three mixers (float32 and S/PDIF, plus fixed32 for embedded devices which do not feature an FPU), so all other types must be cast into one of those. Still with our example, the p_aout->mixer.mixer structure looks like :

- i_format = 'fl32'
- i_rate = 44100
- i_channels = AOUT_CHAN_LEFT | AOUT_CHAN_RIGHT
- i_frame_length = 1
- i_bytes_per_frame = 8

The aout core will thus allocate an audio filter to convert 'fl32' to AOUT_FMT_S16_NE. This is the only audio filter in the output pipeline. It will also allocate a float32 mixer. Since only one input stream is present, the trivial mixer will be used (only copies samples from the first input stream). Otherwise it would have used a more precise float32 mixer.

The last step of the initialization is to build an input pipeline. When several properties have to be changed, the aout core searches first for an audio filter capable of changing :

1. All parameters ;
2. i_format and i_physical_channels/i_original_channels ;
3. i_format ;

If the whole transformation cannot be done by only one audio filter, it will allocate a second and maybe a third filter to deal with the rest. To follow up on our example, we will allocate two filters : a52tofloat32 (which will deal with the conversion and the downmixing), and a resampler. Quite often, for undecoded formats, the converter will also deal with the downmixing, for efficiency reasons.

When this initialization is over, the "decoder" plug-in can run its main loop. Typically the decoder requests a buffer of length i_nb_samples, and copies the undecoded samples there (using GetChunk()). The buffer then goes along the input pipeline, which will do the decoding (to 'fl32'), and downmixing and resampling. Additional resampling will occur if

complex latency issues in the output layer impose us to go temporarily faster or slower to achieve perfect lipsync (this is decided on a per-buffer basis). At the end of the input pipeline, the buffer is placed in a FIFO, and the decoder thread runs the audio mixer.

The audio mixer then calculates whether it has enough samples to build a new output buffer. If it does, it mixes the input streams, and passes the buffer to the output layer. The buffer goes along the output pipeline (which in our case only contains a converter filter), and then it is put in the output FIFO for the device.

Regularly, the output device will fetch the next buffer from the output FIFO, either through a callback of the audio subsystem (Mac OS X' CoreAudio, SDL), or thanks to a dedicated audio output thread (OSS, ALSA...). This mechanism uses aout_OutputNextBuffer(), and gives the estimated playing date of the buffer. If the computed playing date isn't equal to the estimated playing date (with a small tolerance), the output layer changes the date of all buffers in the audio output module, triggering some resampling at the beginning of the input pipeline when the next buffer will come from the decoder. That way, we shall resynchronize audio and video streams. When the buffer is played, it is finally released.

Mutual exclusion mechanism

The access to the internal structures must be carefully protected, because contrary to other objects in the VLC framework (input, video output, decoders...), the audio output doesn't have an associated thread. It means that parts of the audio output run in different threads (decoders, audio output IO thread, interface), and we do not control when the functions are called. Thus, much care must be taken to avoid concurrent access on the same part of the audio output, without creating a bottleneck which would cause latency problems at the output layer.

Consequently, we have set up a locking mechanism in five parts :

1. *p_aout->mixer_lock* : This lock is taken when the audio mixer is entered. The decoder thread in which the mixer runs must hold the mutex during the mixing, until the buffer comes out of the output pipeline. Without holding this mutex, the interface thread cannot change the output pipeline, and a decoder cannot add a new input stream.
2. *p_input->lock* : This lock is taken when a decoder calls aout_BufferPlay(), as long as the buffer is in the input pipeline. The interface thread cannot change the input pipeline without holding this lock.
3. *p_aout->output_fifo_lock* : This lock must be taken to add or remove a packet from the output FIFO, or change its dates.
4. *p_aout->input_fifos_lock* : This lock must be taken to add or remove a packet from one of the input FIFOs, or change its dates.

Having so many mutexes makes it easy to fall into deadlocks (ie. when a thread has the mixer lock and wants the input fifos lock, and the other has the input fifos lock and wants the mixer lock). We could have worked with fewer locks (and even one global_lock), but for instance when the mixer is running, we do not want to block the audio output IO thread from picking up the next buffer. So for efficiency reasons we want to keep that many locks.

So we have set up a strong discipline in taking the locks. If you need several of the locks, you *must* take them in the order indicated above. For instance if you already the hold input fifos lock, it is *strictly forbidden* to try and take the mixer lock. You must first release the input fifos lock, then take the mixer lock, and finally take again the input fifos lock.

It might seem a big constraint, but the order has been chosen so that in most cases, it is the most natural order to take the locks.

Internal structures

Buffers

The aout_buffer_t structure is only allocated by the aout core functions, and goes from the decoder to the output device. A new aout buffer is allocated in these circumstances :

- Whenever the decoder calls aout_BufferNew().
- In the input and output pipeline, when an audio filter requests a new output buffer (ie. when b_in_place == 0, see below).
- In the audio mixer, when a new output buffer is being prepared.

> **Note:** Most audio filters are able to place the output result in the same buffer as the input data, so most buffers can be reused that way, and we avoid massive allocations. However, some filters require the allocation of an output buffer.
>
> The core functions are smart enough to determine if the buffer is ephemer (for instance if it will only be used between two audio filters, and disposed of immediately therafter), or if it will need to be shared among several threads (as soon as it will need to stay in an input or output FIFO).
>
> In the first case, the aout_buffer_t structure and its associated buffer will be allocated in the thread's stack (via the alloca() system call), whereas in the latter in the process's heap (via malloc()). You, codec or filter developer, don't have to deal with the allocation or deallocation of the buffers.

The fields you'll probably need to use are : p_buffer (pointer to the raw data), i_nb_bytes (size of the significative portion of the data), i_nb_samples, start_date and end_date.

Date management

On the first impression, you might be tempted to think that to calculate the starting date of a buffer, it might be enough to regularly fetch the PTS i_pts from the input, and then : i_pts += i_nb_past_samples * 1000000 / i_rate. Well, I'm sorry to deceive you, but you'll end up with rounding problems, resulting in a crack every few seconds.

Indeed, if you have 1536 samples per buffer (as is often the case for A/52) at 44.1 kHz, it gives : 1536 * 1000000 / 44100 = 34829.9319727891. The decimal part of this figure will drive you mad (note that with 48 kHz samples it is an integral digit, so it will work well in many cases).

One solution could have been to work in nanoseconds instead of milliseconds, but you'd only be making the problem 1000 times less frequent. The only exact solution is to add 34829 for every buffer, and keep the remainder of the division somewhere. For every buffer you add the remainders, and when it's greater than 44100, you add 34830 instead of 34829. That way you don't have the rounding error which would occur in the long run (this is called the Bresenham algorithm).

The good news is, the audio output core provides a structure (audio_date_t) and functions to deal with it :

- *aout_DateInit(audio_date_t * p_date, u32 i_divider)* : Initialize the Bresenham algorithm with the divider i_divider. Usually, i_divider will be the rate of the stream.
- *aout_DateSet(audio_date_t * p_date, mtime_t new_date)* : Initialize the date, and set the remainder to 0. You will usually need this whenever you get a new PTS from the input.
- *aout_DateMove(audio_date_t * p_date, mtime_t difference)* : Add or subtract microseconds from the stored date (used by the aout core when the output layer reports a lipsync problem).
- *aout_DateGet(audio_date_t * p_date)* : Return the current stored date.
- *aout_DateIncrement(audio_date_t * p_date, u32 i_nb_samples)* : Add i_nb_samples * 1000000 to the stored date, taking into account rounding errors, and return the result.

FIFOs

FIFOs are used at two places in the audio output : at the end of the input pipeline, before entering the audio mixer, to store the buffers which haven't been mixed yet ; and at the end of the output pipeline, to queue the buffers for the output device.

FIFOs store a chained list of buffers. They also keep the ending date of the last buffer, and whenever you pass a new buffer, they will enforce the time continuity of the stream by changing its start_date and end_date to match the FIFO's end_date (in case of stream discontinuity, the aout core will have to reset the date). The aout core provides functions to access the FIFO. Please understand than none of these functions use mutexes to protect exclusive access, so you must deal with race conditions yourself if you want to use them directly !

- *aout_FifoInit(aout_instance_t * p_aout, aout_fifo_t * p_fifo, u32 i_rate)* : Initialize the FIFO pointers, and the aout_date_t with the appropriate rate of the stream (see above for an explanation of aout dates).
- *aout_FifoPush(aout_instance_t * p_aout, aout_fifo_t * p_fifo, aout_buffer_t * p_buffer)* : Add p_buffer at the end of the chained list, update its start_date and end_date according to the FIFO's end_date, and update the internal end_date.
- *aout_FifoSet(aout_instance_t * p_aout, aout_fifo_t * p_fifo, mtime_t date)* : Trash all buffers, and set a new end_date. Used when a stream discontinuity has been detected.
- *aout_FifoMoveDates(aout_instance_t * p_aout, aout_fifo_t * p_fifo, mtime_t difference)* : Add or subtract microseconds from end_date and from start_date and end_date of all buffers in the FIFO. The aout core will use this function to force resampling, after lipsync issues.
- *aout_FifoNextStart(aout_instance_t * p_aout, aout_fifo_t * p_fifo)* : Return the start_date which will be given to the next buffer passed to aout_FifoPush().
- *aout_FifoPop(aout_instance_t * p_aout, aout_fifo_t * p_fifo)* : Return the first buffer of the FIFO, and remove it from the chained list.
- *aout_FifoDestroy(aout_instance_t * p_aout, aout_fifo_t * p_fifo)* : Free all buffers in the FIFO.

API for the decoders

The API between the audio output and the decoders is quite simple. As soon as the decoder has the required information to fill in an audio_sample_format_t, it can call : p_dec->p_aout_input = aout_InputNew(p_dec->p_fifo, &p_dec->p_aout, &p_dec->output_format).

In the next operations, the decoder will need both p_aout and p_aout_input. To retrieve a buffer, it calls : p_buffer = aout_BufferNew(p_dec->p_aout, p_dec->p_aout_input, i_nb_frames).

The decoder must at least fill in start_date (using an audio_date_t is recommended), and then it can play the buffer : aout_BufferPlay(p_dec->p_aout, p_dec->p_aout_input, p_buffer). In case of error, the buffer can be deleted (without being played) with aout_BufferDelete(p_dec->p_aout, p_dec->p_aout_input, p_buffer).

When the decoder dies, or the sample format changes, the input stream must be destroyed with : aout_InputDelete(p_dec->p_aout, p_dec->p_aout_input).

API for the output module

An output module must implement a constructor, an optional destructor, and a p_aout->output.pf_play function. The constructor is the function which will be called when the module is loaded, and returns 0 if, and only if the output device could be open. The function may perform specific allocation in p_aout->output.p_sys, provided the structure is deallocated in the destructor.

In most cases, the p_aout->output.pf_play function does nothing (the only exception is when the samples can be processed immediately, without caring about dates, as in the file output). The job is then done by the IO callback which you are supposed to provide.

On modern sound architectures (such as Mac OS X CoreAudio or SDL), when the audio buffer starves, the operating system automatically calls a function from your application. On outdated sound architectures (such as OSS), you have to emulate this behavior. Then your constructor must spawn a new audio IO thread, which periodically calls the IO callback to transfer the data.

When it is called, the first job of the IO callback will be determine the date at which the next samples will be played. Again, on modern platforms this information is given by the operating system, whereas on others you have to deduce it from the state of the internal buffer. Then you call aout_OutputNextBuffer(p_aout, next_date, b_can_sleek), which will return a pointer to the next buffer to write, or NULL if none was available. In the latter case, it is advised to write zeros to the DSP.

The value of the last parameter (b_can_sleek) changes the behavior of the function. When it is set to 0, aout_OutputNextBuffer() will run an internal machinery to compensate for possible drift. For instance if the PTS of the next buffer is 40 ms earlier than the date you ask, it means we are very late. So it will ask the input stage to downsample the incoming buffers, so that we can come back in sync. No specific behavior is thus expected from your module.

On the contrary, when b_can_sleek is set to 1, you tell the output layer not to take any actions to compensate a drift. You will typically use this when you've just played silence, and you can deal with buffers which are too early by inserting zeros (zeros in this case will not break the audio continuity, since you were playing nothing before). Another case of use is with S/PDIF output. S/PDIF packets cannot be resampled for obvious reasons, so you *must* use b_can_sleek = 1.

Once you have a buffer, you just have to transfer it to the DSP, for instance : memcpy(dsp_buffer, p_buffer->p_buffer, p_buffer->i_nb_bytes).

Writing an audio filter

An audio filter module is constituted of a constructor, a destructor, and a p_filter->pf_do_work function. The constructor is passed a p_filter structure, and it returns 0 if it is able to do the *whole* transformation between p_filter->input and p_filter->output. If you can do only part of the transformation, say you can't do it (if the aout core doesn't find a fitting filter, it will split the transformation and ask you again).

Note: Audio filters can be of three types :

- Converters : change i_format (for instance from float32 to s16).
- Resamplers : change i_rate (for instance from 48 kHz to 44.1 kHz).
- Channel mixers : change i_physical_channels/i_original_channels (for instance from 5.1 to stereo).

Audio filters can also combine any of these types. For instance you can have an audio filter which transforms A/52 5.1 to float32 stereo.

The constructor must also set p_filter->b_in_place. If it's 0, the aout core will allocate a new buffer for the output. If it's 1, when you write a byte in the output buffer, it destroys the same byte in the input buffer (they share the same memory area). Some filters can work in place because they just do a linear transformation (like float32->s16), but most filters will want b_in_place = 0. The filter can allocate private data in p_filter->p_sys. Do not forget to deallocate it in the destructor.

The p_filter->pf_do_work gets an input and an output buffer as arguments, and process them. At the end of the processing, do not forget to set p_out_buf->i_nb_samples and p_out_buf->i_nb_bytes, since they aren't inited by the aout core (their values can change between input and output and it's not quite predictible).

Writing an audio mixer

Writing an audio mixer is very similar to writing an audio filter. The only difference is that you have to deal with the input buffers yourself, and request for new buffers when you need to. Between two calls to pf_do_work, the position in the buffer is remembered in p_input->p_first_byte_to_mix (it isn't always the start of the buffer, since input and output buffers can be of different length). It is your job to set this pointer at the end of pf_do_work.

For more details, please have a look at the float32 mixer. It's much more understandable than lines of documentation.

Appendix A. Ports

Port steps

Basically, porting to a new architecture boils down to follow the following steps :

1. *Building the VLC :* That may be the most difficult part, depending on how POSIX the architecture is. You have to produce valid C.
2. *Having video :* If your architecture features an X server, it should be straightforward, though you might have problems with xvideo or xshm. Otherwise you can try to use SDL if it is supported, or end up writing your own video output plugin.
3. *Having audio :* If your architecture features an OSS compatible DSP or ALSA, you can reuse an existing plugin. Otherwise you will have to write your own audio output plugin.
4. *Accessing DVDs :* You are going to need a write access to the DVD device. Every system has specific ioctl() for key negociation with the DVD drive, so we have set up an abstration layer in `plugins/dvd/dvd_ioctl.c`. You might need to add stuff here. Some operating systems won't give you access to the key negociation (MacOS X), so you will have to write a kernel extension or you will only be able to read unencrypted DVDs. Other operating systems might only give you read access to the DVD device if you are root. Your mileage may vary.
5. *Writing a native interface :* If your system doesn't support GTK or Qt, you will have to write a native interface plugin (for instance Aqua or Win32). You may also need to rewrite the video output plugin if you're currently using a slow compatibility layer.
6. *Optimizing :* If your architecture features a special set of multimedia instructions (such as MMX) that is not supported by VLC, you may want to write specific optimizations. Heavy calculation parts are : IDCT (see idct plugin), motion compensation (see motion plugin), and YUV (see video output) if you don't use the YUV overlay support of your video board (SDL or XVideo extension).

Building

This is probably the most complicated part. If your platform is fully POSIX-compliant (such as GNU/Linux), it should be quick, otherwise expect troubles. Known issues are :

- Finding a compiler : We use `gcc` on all platforms, and `mingw32` to cross-compile the win32 port. If you don't you're probably in *very big* trouble. Good luck.
- Finding GNU make : Our `Makefile` is heavily GNU make specific, so I suggest you install it.
- Running the `configure` script : This is basically a shell script, so if you have a UNIX shell on your platform it shouldn't be a problem. It will probe your system for headers and libraries needed. It needs adequate `config.sub` and `config.guess`, so if your platform is young your provider may have supplied customized versions. Check with it.
- Compiling the VLC binary : This is the most difficult. Type **make** or **gmake** and watch the results. It will probably break soon on a parse error. Add the headers missing, fix mistakes. If you cannot make it to also compiles on other platforms, use #ifdef directives. Add tests for functions or libraries in `configure.in` and run **autoheader** and **autoconf**. Always prefer tests on #ifdef HAVE_MY_HEADER_T, instead of #ifdef SYS_MYOPERATINGSYSTEM. You may especially experience problems with the network code in `src/input/input.c`.
- Threads : If your system has an exotic thread implementation, you will probably need to fill the wrappers in `include/threads.h` for your system. Currently supported implementations include the POSIX pthreads, the BeOS threads, and the Mach cthreads.

- Linking : You will need special flags to the compiler, to allow symbol exports (otherwise plug-ins won't work). For instance under GNU/Linux you need `-rdynamic`.
- Compiling plug-ins : You do not need external plug-ins at first, you can build all you need in (see `Makefile.opts`). In the long run though, it is a good idea to change `PCFLAGS` and `PLCFLAGS` to allow run-time loading of libraries. You are going to need libdl, or a similar dynamic loader. To add support for an exotic dynamic loader, have a look at `include/modules_core.h` . Currently supported implementations include the UNIX dynamic loader and the BeOS image loader.
- Assembling : If you use specific optimizations (such as MMX), you may have problem assembling files, because the assembler syntax may be different on your platform. Try without it at first. Pay attention to the optimization flags too, you may see a *huge* difference.

VLC should work both on little endian and big endian systems. All load operations should be aligned on the native size of the type, so that it works on exotic processors like Sparc or Alpha. It should work on 64-bit platforms, though it has not been optimized for it. A big boost for them would be to have a WORD_TYPE = u64 in `include/input_ext-dec.h`, but it is currently broken for unknown reasons.

If you experience run-time problems, see the following appendix and pray for you to have **gdb**...

Appendix B. Advanced debugging

We never debug our code, because we don’t put bugs in. Okay, you want some real stuff. Sam still uses `printf()` to find out where it crashes. For real programmers, here is a summary of what you can do if you have problems.

Where does it crash ?

The best way to know that is to use gdb. You can start using it with good chances by configuring with `--enable-debug`. It will add `-g` to the compiler `CFLAGS`, and activate some additional safety checks. Just run **gdb vlc**, type **run myfile.vob**, and wait until it crashes. You can view where it stopped with **bt**, and print variables with **print <C-style>**.

If you run into troubles, you may want to turn the optimizations off. Optimizations (especially inline functions) may confuse the debugger. Use `--disable-optimizations` in that case.

Other problems

It may be more complicated than that, for instance unpredictable behaviour, random bug or performance issue. You have several options to deal with this. If you experience unpredictable behaviour, I hope you don’t have a heap or stack corruption (eg. writing in an unallocated space), because they are hard to find. If you are really desperate, have a look at something like ElectricFence or dmalloc. Under GNU/Linux, an easy check is to type **export MALLOC_CHECK_=2** before launching vlc (see **malloc(3)** for more information).

VLC offers a "trace-mode". It can create a log file with very accurate dates and messages of what it does, so it is useful to detect performance issues or lock-ups. Compile with `--enable-trace` and tune the `TRACE_*` flags in `include/config.h` to enable certain types of messages (log file writing can take up a lot of time, and will have side effects).

Appendix C. Project history

VIA and the Network2000 project

The whole project started back in 1995. At that time, students of the École Centrale de Paris (http://www.ecp.fr/) enjoyed a TokenRing network, managed by the VIA Centrale Réseaux association (http://www.via.ecp.fr/), and were looking for a solution to upgrade to a modern network. So the idea behind Network2000 was to find a project students would realize that would be interesting, would require a high-quality network, and could provide enough fame so that sponsors would be interested.

Someone came up with the idea of doing television broadcast on the network, so that students could watch TV in their room. This was interesting, mixed a lot of cool technologies, and provided fame because no one had written a free MPEG-2 decoder so far.

Foundation of the VideoLAN project

3Com (http://www.3com.com/), Bouygues (http://www.bouygues.com/) and la Société des Amis were interested and financed the project, which was then known after the name of VideoLAN.

The VideoLAN team, in particular Michel Lespinasse (mailto:walken@via.ecp.fr) (current maintainer of LiViD (http://www.linuxvideo.org/)'s mpeg2dec) and Régis Duchesne (mailto:hpreg@via.ecp.fr), started writing code in 1996. By the end of 1997 they had a working client-server solution, but it would crash a lot and was hard to extend.

At that time it was still closed-source and only-for-demo code.

VLC media player design

In 1998, Vincent Seguin (mailto:seguin@via.ecp.fr) (structure, interface and video output), Christophe Massiot (mailto:massiot@via.ecp.fr) (input and video decoder), Michel Kaempf (mailto:maxx@via.ecp.fr) (audio decoder and audio output) and Jean-Marc Dressler (mailto:polux@via.ecp.fr) (synchronization) decided to write a brand new player from scratch, called VideoLAN Client (VLC), so that it could be easily open sourced. Of course we based it on code written by our predecessors, but in an advanced structure, described in the first chapter (it hasn't been necessary to change it a lot).

At the same time, Benoît Steiner (mailto:benny@via.ecp.fr) started the writing of an advanced stream server, called VideoLAN Server (VLS).

Functional test seeds have been released internally in June 1999 (vlc-DR1) and November 1999 (vlc-DR2), and we started large scale tests and presentations. The French audience discovered us at Linux Expo in June 1999, presenting our 20 minutes of Golden Eye (which is now a legend among developers :-). At that time only a network input was possible, file input was added later, but it remained kludgy for a while.

In early 2000, we (especially Samuel Hocevar (mailto:sam@via.ecp.fr), who is still a major contributor) started working on DVDs (PS files, AC3, SPU). In the summer 2000, pre-release builds have been seeded (0.2.0 versions), but they still lacked essential features.

In late 2000, Christophe Massiot (mailto:massiot@via.ecp.fr) with the support of his company, IDEALX (http://www.idealx.com/), rewrote major parts of the input to allow modularization and advanced navigation, and StÃ©phane Borel (mailto:stef@via.ecp.fr) worked on a fully-featured DVD plug-in for VLC.

The Opening

For Linux Expo in February 2001, the Free Software Foundation (http://www.gnu.org/) and IDEALX (http://www.idealx.com/) wanted to make live streaming of the 2001 FSF awards from Paris to New York. VideoLAN was the chosen solution. Finally it couldn't be done live because of bandwidth considerations, but a chain of fully open-source solutions (http://www.via.ecp.fr/~massiot/encoding.html) made it possible to record it.

At the same time, the president of the École Centrale Paris (http://www.ecp.fr/) officially decided to place the software under GNU General Public Licence, thanks to Henri Fallon (mailto:henri@via.ecp.fr), Jean-Philippe Rey (mailto:jprey@ads.ecp.fr), and the IDEALX team.

VideoLAN software is now one of the most popular open source DVD players available, and has contributors all around the world. The last chapter of this appendix is not written yet :-).

Appendix D. GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary

word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.