Ecasound Programmer's Guide

Kai Vehmanen

18112003

Contents

Chapter 1

Preface

This document describes how Ecasound and the related libraries work, how to use them, how to extend and add features and other similar issues. Before reading this document, you should first take a look at other available documentation (especially `Ecasound Users's Guide`).

If not otherwise specified, all documentation refers to the latest Ecasound version.

Chapter 2

Document history

- 18.11.2003 - Typo fixes. Updated documentation to reflect the new naming convention (ecasound refers to the binary, Ecasound refers to the whole package).
- 07.11.2002 - Added documentation for NetECI.
- 25.10.2002 - Added "Checklist for Audio Object Implementations".
- 17.10.2002 - Added a warning against direct use of libecasound and libkvutils. Using ECI is from now on the preferred way of using ecasound as a development platform. Rewrote the "Versioning" section.
- 02.10.2002 - Added the "Protocols and Interfaces" chapter.
- 29.04.2002 - Added chapter about "Unit Tests" [**?**].
- 28.04.2002 - Revised namespace policy (see chapter **??**), replaced references to the obsolete ECA_DEBUG with a description of the new ECA_LOGGER subsystem.
- 27.02.2002 - Rewrote the "Control flows" chapter according to the structural changes made in 2.1dev8. Added a "References" section.
- 31.01.2002 - Reorganization of document structure. New chapter about "Library organization".
- 19.12.2001 - Added chapter **??** about include hierarchies.
- 28.10.2001 - Lots of changes to the "Object maps" chapter.
- 21.10.2001 - Added this history section.

Chapter 3

General programming guidelines

3.1 Design and programming

3.1.1 Open and generic design

Over the years ecasound's core design has been revised many times. After rewriting some code sections hundreds of times, you start to appreciate genericity. :)

3.1.2 Object-orientation

Ecasound is written in C++ (as specified in 1997 ANSI/ISO C++ standard). When designing classes and routines, I often use Eiffel language as a reference. At same time overuse of object-orientation is another thing to avoid. Object-orientation is a very effective design method, but not the only one. Sometimes other approaches work better.

3.1.3 Data hiding

This design principle deserves to be mentioned separately. Whenever possible, the actual data representation and implementation should always be hidden. This allows you to make local implementation changes without affecting other parts of the code base. It cannot be emphasized enough how important this goal is for large software projects like Ecasound.

3.1.4 Design by contract

Design by contract means that when you write a new routine, in addition to the actual code, you also describe routine's behaviour as accurately as possible by defining a set of preconditions and postconditions. These conditions are usually defined using boolean assertations.

Routine must specify all requirements and assumptions. If the caller violates this specification, routine is not responsible for the error. This means that routine mustn't check argument validity. This must be done by the caller.

Routine should also specify, what conditions are true when returning to the caller. By doing this, routine ensures that it works correctly and calling routine knows what has been done.

Ideally, these conditions *prove* that the routine works correctly. Unfortunately writing good conditions is not always easy. Still, the benefits of this approach should be clear. When you call a well-defined routine, a) you know what parameter values it accepts, b) you know what it does and c) if errors occur, it's easier to pinpoint the faulty routine. In practice this is done by using comments and pre/postconditions.

As C++ doesn't directly support pre/postconditions, I've simulated them using the DEFINITION_BY_CONTRACT and DBC tools provided by libkvutils.

3.1.5 Routine side effects

I try to make a clear distinction between routines that have side-effects (=methods, processors, modifiers; routines that change object's state) and const routines (=functions, observers).

To make monitoring side effects easier, all Ecasound classes should be const-correct. A object is const-correct if a function taking only a single argument that is a const reference to that object is not able, without explicit casting, to obtain a non-const reference to that same object (or a portion thereof) from within the function body. [**?**]

3.1.6 Sanity checks

Sanity checks are done only to prevent crashes. All effects and operators happily accept "insane" parameters. For instance you can give -100.0% to the amplifier effect. This of course results in inverted sample data. I think this a reasonable approach. After all, Ecasound is supposed to be a tool for creative work and experimenting. It's not meant for e-commerce. ;)

3.1.7 Error handling

Two specific things worth mentioning: First, the standard UNIX-style error handling, where functions performing actions return an integer value, is *not* used in Ecasound. As described in the above section *Routine side effects*, all routines are either modifiers or observers, not both. So when using Ecasound APIs, you first perform an action (modifying function), and then afterwards check what happened (using an observer function).

3.1.8 Exceptions

C++ exceptions are used in Ecasound. Exception based error handling has its problems, but in some cases it is clearly the best option. Exceptions are most useful in situations where controlled error recovery is very difficult, and in situations where errors occurs only very rarely. This allows callers to avoid constantly checking returns values for functions that in normal use never fail. Another special case is handling critical errors that occur in class contructors.

Using exceptions for anything other than pure error handling is to be avoided at all cost. And when exceptions are used, their use must be specified in function prototypes. This is important, as clients need to know what exceptions can be thrown. C++ unfortunately doesn't require strict exception prototypes, so this issue requires extra care.

A list of specific cases where exceptions are used follows:

AUDIO_IO - open() This method is used for initializing external connections (opening files or devices, loading shared libraries, opening IPC connections). It's impossible to know in advance what might happen. In many cases it is also useful to get more verbose information about the problem that caused open() to fail. Throwing an exception is an excellent way to achieve this.

ECA_CHAINSETUP - enable()

ECA_CHAINSETUP - load_from_, save() and save_to_

ECA_SESSION - constructor

3.2 Coding style

3.2.1 Variable and type naming

Variable names are all lower case and words are separated with underscores (int very_long_variable_name_with_underscores). Class data members are marked with *_rep* postfix. Data members which are pointers are marked with *_repp*. Index-style short variable names (*n*, *m*, etc.) are only used in local scopes. Enum types have capitalized names (*Some_enum*).

3.2.2 Package specific

libecasound, ecasound, ecatools, libkvutils Class names are all in upper case and words separated with underscores (class ECA_CONTROL_BASE). This a standard style in Eiffel programming.

libqtecasound, qtecasound, ecawave Qt-style is used when naming classes (class QELevelMeter), otherwise same as above.

3.2.3 Private classes

Some classes are divided into public and private parts. This is done to make it easier to maintain binary-level compatibility between library versions, and to get rid of header file dependecies.

Private classes have a *_impl* postfix in their name. They are usually stored into separate files which also use the *_impl* notation.

For instance the ECA_ENGINE class (eca-engine.h) has a private class ECA_ENGINE_impl (eca-engine_impl.h). Access to ECA_ENGINE_impl is only allowed to ECA_ENGINE member functions. In addition, the private header file (eca-engine_impl.h) is only included from the ECA_ENGINE implementation file (eca-engine.cpp). This allows us to add new data members to ECA_ENGINE_impl without breaking the binary interface.

3.2.4 Unit tests

Unit tests are used for verifying that modules work as intended. A test for component, with a public interface defined in "prefix-component.h", should located in "prefix-component_test.h". The test itself should implement the ECA_TEST_CASE interface. In addition, generic test cases should be added to ECA_TEST_REPOSITORY - see "libecasound/eca-test-repository.cpp".

3.3 Physical level organization

Ecasound libraries and applications are divided into *distribution packages*, *directories* and *file groups*.

3.3.1 Include hierarchies

Include statements that are not stricly necessary should be dropped! Not only do they cause unwanted dependencies, they also create more work for the compiler. Ecasound already takes painfully long to compile, so there's no room for sloppy include work. Few distinct rules follow:

- In header files, no extra header files should be defined. For instance in many cases it's enough to state that object SOME_TYPE is a class without need for the full implementation; so instead of "#include "sometype.h", use "class SOME_TYPE;".
- For modules with separate implementation and header files, dependencies to other modules don't need to be stated in both.
- Direct dependencies to outside modules must always be mentioned directly. It's easy to unknowingly include a required header file via some other header file. This should be avoided as it hides real dependencies.
- When including headers for more special services, it's good to add a comment why this header file is needed.

3.3.2 Distribution packages

As an example, *ecasound* and *qtecasound* are distributed as separate packages. This decision has been made because a) they are clearly independent, b) they have different external dependencies, and c) they address different target uses.

3.3.3 Directories

It's convenient to organize larger sets of source code into separate directories. For instance in Ecasound, *libecasound* and *ecatools* are in two separate directories.

3.3.4 File groups

Although files are divided in directories and subdirectories, there's still a need to logically group a set of source files based on their use and role in the overall design. As the use of C++ namespaces is very limited in Ecasound (to avoid portability problems), filename prefixes are used for grouping files. Here's a short list of commonly used prefixes.

audioio*.{cpp,h} Audio device and file input/output.

audiofx*.{cpp,h} Audio effects and other DSP-related code.

audiogate*.{cpp,h} Gate operators.

eca-*.{cpp,h} Core functionality.

midi-*.{cpp,h} MIDI input/output devices, handlers and controller code.

osc-*.{cpp,h} Oscillator and other controller sources.

qe*.{cpp,h} Generic prefix for files utilizing both Qt and Ecasound libraries.

samplebuffer-*.{cpp,h} Routines and helper functions for processing audio data buffers.

You should note that these are just recommendations - there are no strict rules on how files should be named.

3.3.5 C++ std namespace

The preferred way to access C++ standard library functions is to use explicit namespace selectors ("std::string") in public headers files, and *using* declarations in the implementation parts ("using std::string"). It's also possible to import the whole std namespace ("using namespace std;") in the beginning of an implementation file.

3.4 Documentation style

Javadoc-style class documentation is the preferred style. Class members can be documented either when they are declared (header files), or when they are defined. Especially when specifying complicated interfaces, it's better to put documentation in the definition files. This way the header files remain compact and serve better as a reference.

Here's a few general documentation guide lines:

Use of 3rd person "Writes samples to memory." instead of "Write samples to memory."

Sentences start with a verb "Writes samples to memory." instead of "Samples are written to memory."

This instead of the "Get controllers connected to this effect." instead of "Get controllers connected to the effect.

3.5 Versioning

All Ecasound releases have a distinct version number. The version number syntax is *x.y[.z][-extraT*, where x and y are the major and minor numbers, and z is an optional revision number. To test major changes, separate *-preX* or *-rcX* versions can be distributed before the actual new release.

In addition, all Ecasound libraries have a separate interface version. The libtool-style *version:revision:age* versioning is used. See the libtool documentation for details.

One important thing to note is that the library interface version numbers are tied to the source-code level interfaces, not the binary interfaces. Because binary interfaces are not explicitly versioned, applications should always statically link against the Ecasound libraries. Also, any private source-code interfaces, ie. header files with a *_impl.h* postfix, are not part of the versioned public interface. Applications should not rely on these interfaces!

All changes in the public interfaces are documented in library specific *ChangeLog* files. These files are usually located in the top-level source directory of the versioned library.

One thing to note is that Ecasound's versioning practises have changed quite a few times during the project's history. The rules described above only apply to Ecasound 2.2.0 and newer releases.

Chapter 4

How Ecasound works?

4.1 Example use cases

Here's a few common use cases how Ecasound can be used.

4.1.1 Simple non-interactive processing

One input is processed and then written to one output. This includes effect processing, normal sample playback, format conversions, etc.

4.1.2 Multitrack mixing

Multiple inputs are mixed into one output.

4.1.3 Realtime effect processing

There's at least one realtime input and one realtime output. Signal is sampled from the realtime input, processed and written to the realtime output.

4.1.4 One-track recording

One input is processed and written to one or more outputs.

4.1.5 Multitrack recording

The most common situation is that there are two separate chains. First one consists of realtime input routed to a non-realtime output. This is the recording chain. The other one is the monitor chain and it consists of one or more non-realtime inputs routed to a realtime output. You could also route your realtime input to the monitoring chain, but this is not recommended because of severe timing problems. To synchronize these two separate chains, Ecasound uses a special multitrack mode (which should be enabled automatically).

4.1.6 Recycling a signal through external devices

Just like multirack recording. The only difference is that realtime input and output are externally connected.

4.2 Audio signal routing

Basic audio flow inside an Ecasound chainsetup is as follows: Audio data is routed from input audio objects to a group of chains. In the chains audio data is processed using chain operators. After processing data is routed to output objects.

Using internal loop devices, it's also possible to route signals from one chain to another. Looping causes extra latency of one engine cycle.

Routing of signals is based on the ability to assign inputs and outputs to multiple chains. Assigning an input object to multiple chains divides the audio signal generating multiple copies of the original input data. Similarly with an output object, data from multiple chains is mixed together to one output object.

4.3 Control flow

4.3.1 Batch operation

When Ecasound is run in batch mode, the program flow is simple. To store the session data, a ECA_SESSION object is first created. The created object is then passed as an argument for ECA_CONTROL class constructor.

All required configuration of inputs, outputs and chain operators is done using the services provided by ECA_CONTROL. Once a valid chainsetup is ready for processing, batch operation is initiated by issuing ECA_CONTROL::run(). This function will block until processing is finished.

4.3.2 Interactive operation

Interactive operation is similar to batch operation. The important difference is that processing is started with ECA_CONTROL::start(). Unlike run(), start() does not block the calling thread. This makes it possible to continue using the ECA_CONTROL interface while engine is running in the background.

Two important concepts to understand when working with ECA_CONTROL are the *selected* and *connected* chainsetups. ECA_CONTROL allows working with multiple chainsetups, but only one of them can be edited at a time, and similarly only one at a time can be connected to the processing engine. For instance if you add a new input object with add_audio_input(), it is added to the selected chainsetup. Similarly when you issue start(), the connected chainsetup is started.

Chapter 5

Library organization

The primary source for class documentation is header files. A browsable version of header documentation is at `www.wakkanet.fi/~kaiv/ecasound/Documentation/doxygen_pages.html` Anyway, let's look at the some central classes.

5.1 Interfaces for external use

The following classes of libecasound are designed as primary interfaces for external use. The approach is based on the Facade (GoF185) design pattern. The primary goals are concentrating functionality, and maintaining a higher level of interface stability.

5.1.1 ECA_CONTROL - eca-control.h

ECA_CONTROL represents the whole public interface offered by libecasound. The primary purpose of ECA_CONTROL is to offer a consistent, straightforward interface for controlling Ecasound. The interface is also designed to be more stable than other parts of the library.

On important part of ECA_CONTROL is the functionality for interpreting EOS (Ecasound Option Syntax) and EAIM (Ecasound Interactive Mode) commands.

5.1.2 ECA_CONTROL_INTERFACE - eca-control-interface.h

C++ implementation of the Ecasound Control Interface (ECI) API. See section "Ecasound Control Interface" for more information.

5.2 Core classes

This section introduces the core classes, which define the central data types and are responsible for the main program logic.

5.2.1 AUDIO_IO_PROXY_SERVER - audioio-proxy-server.h

Implements a audio input/output subsystem that adds a second layer of buffering between the main processing engine and non-realtime audio input and output objects.

Double buffering is needed to guarantee a realtime constrained data stream even when dealing with non-realtime objects like disk files.

5.2.2 CHAIN - eca-chain.h

Class representing one abstract audio signal chain. CHAIN objects consist of chain operators, controllers and their state information.

5.2.3 ECA_CHAINSETUP - eca-chainsetup.h

ECA_CHAINSETUP is the central class for storing user-visible objects. All inputs, output, chain operator and controller objects belonging to one logical setup are attributes of on ECA_CHAINSETUP object.

5.2.4 ECA_ENGINE - eca-engine.h

ECA_ENGINE is the actual processing engine. It is initialized with a pointer to a ECA_CHAINSETUP object, which has all information needed at runtime. In other words ECA_ENGINE is used to execute the chainsetup. You could say ECA_ENGINE renders the final product according to instruction given in ECA_CHAINSETUP.

Processing is started with the exec() member function and after that, ECA_ENGINE runs on its own. If 'batch_mode' is selected (parameter to exec()), one started ECA_ENGINE will run until a 'finished' condition is met and then exit automatically. Finished means that we have read all available data from input sources. Of course if some input has infinite length (soundcards for example), processing will never finish. To get around this limitation, it's possible to set the processing length (see ECA_CONTROL_OBJECTS::set_chainsetup_processing_length_in_seconds()).

If batch mode is not enabled, engine will just perform the init phase and starts waiting for further instructions. These instructions can be send to the engine using the ECA_ENGINE::command() member function.

ECA_ENGINE has the following states:

not_ready - ECA_SESSION object is not ready for processing or ECA_ENGINE hasn't been created

running - processing

stopped - processing hasn't been started or it has been stopped before completion

finished - processing has been completed

error - an error has occured during prosessing

5.2.5 ECA_SESSION - eca-session.h

ECA_SESSION is an abtraction used to represents a group of chainsetups. At any time, only one chainsetup object at a time can be active (connected). For modification, one chainsetup can be set as 'selected'. This means that all configuration operations are targeted to the selected chainsetup.

The only public access to ECA_SESSION objects is through ECA_CONTROL objects.

5.2.6 MIDI_SERVER - midi-server.h

Engine that handles all MIDI input and output.

5.2.7 SAMPLEBUFFER - samplebuffer.h

Basic unit for representing blocks of sample data. The data type used to represent single samples, valid value ranges, channel count and system endianess are all specified in "samplebuffer.h" and "sample_specs.h".

5.3 Feature and capability interface classes

Many libecasound classes have similar attribute sets and capabilities. To make use of these shared features, most common features have their own virtual base classes. All objects that have a particular feature, inherit the same virtual base class. This makes object grouping and management easier and less error prone.

5.3.1 DYNAMIC_PARAMETERS<T> - dynamic-parameters.h

Implemented by all classes that provide a set of generic parameters of type *T*. Parameter can be observed and modified, and they usually are identiefied by a unique name and a more verbose description. Number of parameters can vary dynamically. Other objects can access these parameters without detailed knowledge of the object itself.

5.3.2 ECA_AUDIO_FORMAT - eca-audio-format.h

Implemented by all classes that have a current audio format that can be observed and modified.

5.3.3 ECA_AUDIO_POSITION - eca-audio-position.h

Implemented by all classes that need to maintain current audio position and length.

5.3.4 ECA_SAMPLERATE_AWARE - eca-samplerate-aware.h

Implemented by all classes that need knowledge of current sampling rate.

5.3.5 MIDI_CLIENT - midi-client.h

Implemented b all classes that require a connection to an instance of MIDI_SERVER.

5.4 Object interfaces

Object interfaces define the behaviour for common objects used by libecasound. The core classes rarely operate on specific object types, but instead use object interfaces (abstract interfaces). Object interfaces are usually abstract C++ classes (instances of these classes cannot be created as some of functions don't yet have a concreate implementation, ie. they pure virtual functions).

5.4.1 AUDIO_IO - audioio.h

Virtual base class for all audio I/O objects. Different types of audio objects include files, audio devices, sound producing program modules, audio server clients, and so on.

More specialized interface classesa are AUDIO_IO_DEVICE (for realtime audio objects) and AUDIO_IO_BUFFERED (for POSIX-style buffered i/o). There's also a special AUDIO_IO_MANAGER interface for managing multiple audio objects of same type inside one chainsetup.

5.4.2 CHAIN_OPERATOR - eca-chainop.h

Virtual base class for chain operators.

5.4.3 CONTROLLER_SOURCE - ctrl-source.h

Virtual Base class for all controller sources.

5.4.4 MIDI_IO - midiio.h

Virtual base for objects capable of reading and writing raw MIDI data.

5.4.5 MIDI_HANDLER - midi-server.h

Virtual base class for objects capable of receiving and processing MIDI data.

5.5 Object interface implementations - plugins

Majority of the classes in libecasound fall to this category. They implement the behaviour of some object interface type. As other parts of the library only use the object interfaces, these implementation classes are fairly isolated. Changes made inside object implementation have no effect to other parts of the library. Similarly new object interface implementations can be added without modifying the core classes.

5.6 Utility classes

5.6.1 eca-logger.h - ECA_LOGGER

Singleton class that provides an interface to Ecasound's logging subsystem. Libecasound sends all log messages to this interface. The actual logger implementation can be done in many ways. For example in the console mode user-interface of Ecasound, TEXTDEBUG class implements the ECA_LOGGER_INTERFACE class interface. It sends all messages that have a suitable debug level to the console's standard output. On the other hand, in qtecasound, ECA_LOGGER_INTERFACE is implemented using a Qt widget.

New ECA_LOGGER_INTERFACE implementations can be registered at runtime with the *ECA_LOGGER::attach_logger()* member function (declared in eca-logger.h).

5.7 Object maps

Object maps are central repositories for commonly used objects. Their main purpose is to add flexibility to handling different object types - especially to handling dynamic addition and removal of whole object types. They provide the following services:

- listing all object types in any of the available categories (for instance, list all effect types)
- creating new object instances based on keyword strings (for instance, returns an mp3 object if "foo.mp3" is given as keyword)
- adding new object types (object map item is identified by tuple of "keyword, regex expression, object type")
- removing object types
- reverse mapping objects back to keyword strings

In Ecasound, all access to object maps goes throuh the library-wide ECA_OBJECT_FACTORY class, which provides a set of static functions to access the object maps.

This system may sound a bit complex, but in practise it is quite simple and makes a lot of things more easier. For instance, when adding new object types to the library, you only have to add one function call which registers the new object; no need to modify any other part of the library. It also makes it possible to add new types at runtime, including dynamically loaded plugins.

One special use-case is where an application linked against libecasound adds its own custom object types on startup. All parts of libecasound can use the custom objects, although they are not part of library itself.

All objects defined in libecasound are registered in the file *eca-static-object-maps.cpp*.

5.7.1 eca-object.h - ECA_OBJECT

A virtual base class that represents an Ecasound object. All objects handled by the object factory must inherit this class.

5.7.2 eca-object-factory.h - ECA_OBJECT_FACTORY

The public interface to Ecasound's object maps. All its functions are static.

Chapter 6

Adding new features and components to Ecasound?

6.1 Things to remember when writing new C++ classes

6.1.1 Copy constructor and assignment operator

Always take a moment to check your copy constructor and the assign operation (=operation()). Basicly you have three alternatives:

- Trust the automatically created default definitons. If you don't have any pointers as data members, this isn't necessarily a bad choice at all. At least the compiler remembers to copy all members!
- If you have pointers to objects as class data members, you should write definitions for both the copy-constructor and the assign operation.
- If you are lazy, just declare the two functions as null functions, and put them in _private_ access scope. At least this way nobody will use the functions by accident!

6.2 Audio objects

To implement a new audio object type, you must first select which top-level class to derive from. Usually this is either AUDIO_IO (the top-level class), AUDIO_IO_BUFFERED (a more low level interface) or AUDIO_IO_DEVICE (realtime devices).

The second step is to implement the various virtual functions declared in the parent classes. These functions can be divided into four categories: 1) attributes (describes the object and its capabilities), 2) configuration (routines

used for setting up the object), 3) main functionality (open, close, input, output, etc) and 4) runtime information (status info).

Adding the new object to Ecasound is much like adding a new effect (see the next section). Basicly you just add it to the makefiles and then register it to the appropriate object map (see below).

6.2.1 Checklist for Audio Object Implementations

1. Check the read_buffer() and write_buffer() change the internal position with either set_position_in_samples() or change_position_in_samples() functions of ECA_AUDIO_POSITION. Also, when writing a new file, extend_position() should also be called. All this is done automatically if using read_samples() and write_samples() from AUDIO_IO_BUFFERED.

2. If implementing a proxy object, separately consider all public functions of audioio-proxy.h (whether to reimplement or use as they are).

3. Check that open() and close() call AUDIO_IO::open() and AUDIO_IO::close(), and in the right order.

4. If the object supports seeking, seek_position() must be implemented.

6.3 Effects and other chain operators

Write a new class that inherits from CHAIN_OPERATOR or any of its successors. Implement the necessary routines (init, set/get_parameter, process and a default constructor) and add your source files to libecasound's makefiles. Then all that's left to do is to add your effect to *libecasound/eca-static-object-maps.cpp*, *register_default_objects()*. Now the new effect can be used just like any other Ecasound effect (parameters control, effect presets, etc).

Another way to add effects to Ecasound is to write them as LADSPA plugins. The API is well documented and there's plenty of example code available. See `www.ladspa.org` for more information.

6.4 Differences between audio objects and chain operators

Design-wise, audio objects and effects (chain operators) aren't that far away from each other. Many audio apps don't separate these concepts at all (for instance most UG based synthesizers). In Ecasound though, there are some differences:

Input/output:

- audio objects can be opened for reading writing or read&write
- effects are modeless

- audio objects read from, or write to a buffer
- effects get a buffer which they operate (in-place processing)

Audio format:

- audio objects have a distinct audio format (sample rate, bits, channels)
- effects should be capable of accepting audio data in any format (this is usually easy as Ecasound converts all input data to its internal format)

Control:

- audio objects can be opened, closed, prepared, started and stopped
- effects don't have a running state

Position:

- audio objects have length and position attributes
- effects just process buffers and don't know about their position

A good example of the similarity between the two types are LADSPA oscillator plugins. Although they are effects, you can easily use them as audio inputs by specifying:

```
"ecasound -i null -o /dev/dsp -el:sine_fcac,440,1"
```

6.5 LADSPA plugins

Ecasound supports LADSPA-effect plugins (Linux Audio Developer's Simple Plugin API). See `LAD mailing list web site` for more info about LADSPA. Other useful sites are `LADSPA home page` and `LADSPA documentation`.

Chapter 7

Application development using the Ecasound framework

7.1 Console mode ecasound - [all languages]

This is the easiest way to take advantage of Ecasound features in your own programs. You can fork ecasound, pipe commands to ecasound's interactive mode or you can create chainsetup (.ecs) files and load them to ecasound. You'll be able to do practically anything. The only real problem is getting information from ecasound. You'll have to parse ecasound's ascii output if you want to do this. To make this a bit easier, ecasound offers the wellformed output mode **??** and dump-* commands. These can be used to easily parse configuration and status information.

7.2 Ecasound Control Interface - [C++, C, Python]

Idea behind Ecasound Control Interface (ECI) is to take a subset of functionality provided by libecasound, write a simple API for it, and port it to various languages. At the moment, at least C++, C and Python implementations of the ECI API are available and part of the main Ecasound distribution. ECI is heavily based on the Ecasound Interactive Mode (EIAM), and the services it provides.

Specific tasks ECI is aimed at:

- 1. automating (scripting in its traditional sense)
- 2. frontends (generic / specialized)
- 3. sound services to other apps

7.3 NetECI - [various]

NetECI is a network version of the ECI API. When Ecasound is started in daemon mode (the –daemon option), it creates a server for incoming TCP connections. Client applications can connect to this socket and use the connection to control and observe the active session. Multiple clients can connect to the same session.

The protocol is identical to one used in ECI. Clients write EIAM commands to the socket, followed by a CRLF pair. The server will reply using the well-formed output mode syntax (see **??**).

See implementation of *ecamonitor* (part of ecatools), for a working example.

7.4 Libecasound's ECA_CONTROL class - [C++]

Note! Direct use of libecasound and libkvutils is not recommended anymore! Please use the Ecasound Control Interface (ECI) instead.

By linking your program to libecasound, you can use the ECA_CONTROL class for controlling Ecasound. This is a large interface class that offers routines for controlling all Ecasound features. It's easy to use while still powerful. Best examples are the utils in ecatools directory (most of them are just a couple screenfuls of code). Also, qtecasound and ecawave heavily use ECA_CONTROL. Here's a few lines of example code:

```
--cut--
ECA_SESSION esession;
ECA_CONTROL ctrl (&esession);
ctrl.new_chainsetup("default");
[... other setup routines ]
ctrl.start(); // starts processing in another thread (doesn't block)
--cut--
```

If you don't want to use threads, you can run the setup in batch mode:

```
--cut--
ECA_SESSION esession;
ECA_CONTROL ctrl (&esession);
ctrl.add_chainsetup("default");
[... other setup routines ]
ctrl.run(); // blocks until processing is finished
--cut--
```

7.5 Ecasound classes as building blocks - [C++]

Note! Direct use of libecasound and libkvutils is not recommended anymore! Please use the Ecasound Control Interface (ECI) instead.

You can also use individual Ecasound classes directly. This means more control, but it also means more work. Here's another short sample:

```
--cut--
- create a SAMPLE_BUFFER object for storing the samples
- read samples with an audio I/O object - for example WAVEFILE
- process sample data with some effect class - for example EFFECT_LOWPASS
- maybe change the filter frequency with EFFECT_LOWPASS::set_parameter(1, new_value)
- write samples with an audio I/O object - OSSDEVICE, WAVEFILE, etc.
--cut--
```

Chapter 8

Protocols and Interfaces

8.1 Ecasound Interactive Mode - Well-Formed Output Mode

By issuing the EIAM command "int-output-mode-wellformed", Ecasound will start printing all messages using the following format:

```
<message> = <loglevel><sp><msgsize>(<genmsg> | <returnmsg>)

<loglevel> = <integer>      ; loglevel number
<msgsize = <integer>        ; size of content in octets
<genmsg> = <contentblock>   ; generic log message
<returnmsg> = <sp><returntype><contentblock>
                            ; EIAM return value message
<contentblock> = <crlf><content><crlf><crlf>
                            ; actual content of the message
<returntype> = ``i'' | ``li'' | ``f'' | ``s'' | ``S'' | ``e''
                            ; type of the return value (see ECI/EIAM docs)
<content> = *<octet>        ; zero or more octets of message content

<sp> = 0x20                 ; space
<octet> = 0x00-0xff         ; 8bits of data
<crlf> = <cr><lf>           ; new line
<cr> = 0x0d                 ; carriage return
<lf> = 0x0a                 ; line feed
<integer> = +<digit>        ; one or more digits
<digit> = 0x30-0x39         ; digits 0-9
```

Chapter 9

References

Bibliography

[ISO14822] ISO/IEC 14882 - Programming language C++

[Lakos96] Lakos, John: Large-Scale C++ Software Design, Addison Wesley, 1996

[Larman01] Larman, Craig: Applying UML and Patterns, 2nd ed., Prentice Hall, 2001

[Stroustrup97] Stroustrup, Bjarne: C++ Programming Language, 3rd edition, Addison Wesley, 1997