# Architecture Concept

## Table of contents

| Date | Description |
| --- | --- |
| 2014-09-20 | Initial version. |

Table 1: Revision history

# 1 Preliminaries

## 1.1 Sending/Receiving

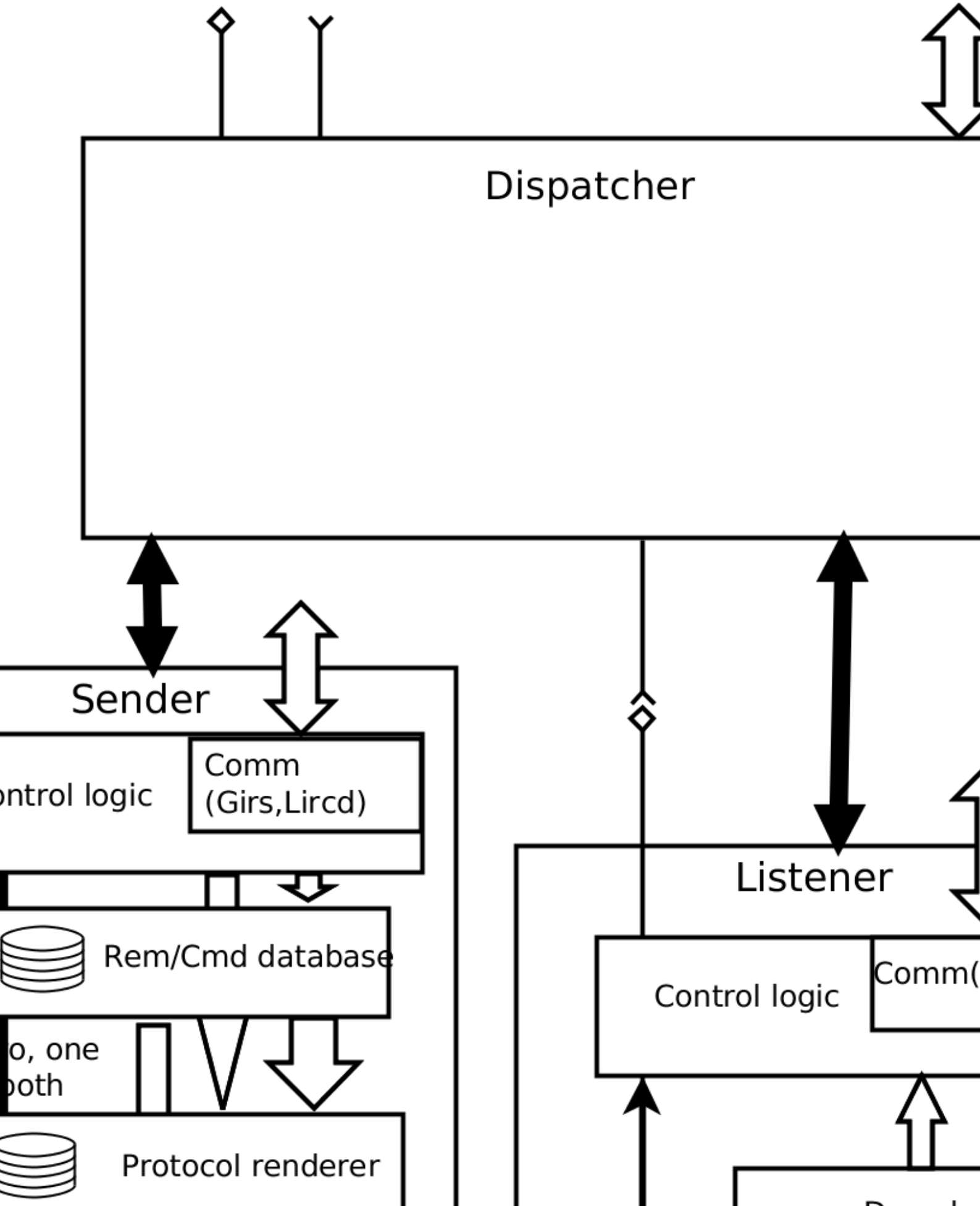
Traditionally, IR hardware and software combines sending and receiving capacities in one unit — unless of course only one of these functinallities are supported. Instead, we argue that sending and receiving of IR signals are fundamentally different activities which preferrably are threaded separately. Of course, this does not prohibit a particular implementation to implement both of these aspects, just like a program may contain both a wordprocessor and a music player.

## 1.2 Learning

Capturing (often called "learning") should be taken out of the requirements. Capturing and analyzing unknown signals is a completely different use case from (deployment) sending and receiving of IT signals. There are other software, like the IrScrutinizer that are optimized for this use case. Capturing of the signals of a present remote is done during installation time, the result saved to a data base used by the sender and/or receiver. (Alternatively, in very many cases, the IR signals for a particular device/remote can be found in public or propritary data bases found on the Internet.).

# 2 Main Concept

A three-component model consisting of a *sender*, a *receiver*, and a *dispatcher*, as shown in the figure below, is proposed. These should be designed as exchangable components.


Dispatcher
Sender
Comm
(Girs,Lircd)
ntrol logic
Rem/Cmd database
o, one
oth
Protocol renderer
Listener
Control logic
Comm(

## 2.1 The Sender

### 2.1.1 Top level Interface

The sender encapsulates sending of IR signals. It should be considered as a server, communicating with its clients using the Girs command language. Basically, it is sent commands to send particular IR signals through the IR hardware it commands. This is specified in the Transmit module of Girs. With moderate effort, it would is also be possible to implement the somewhat more limited protocol of the Lirc daemon Lircd.

The specification of the to-be sent IR signals can take place in three different ways:

- Raw IR sequence/signal; basically equivalent is the CCF, also called Pronto Hex, format. This is simply a list of durations in microseconds of interleaved pulse and gap durations.
- Protocol/Parameter format.
- Remote/Command format, this is simply the name of a remote or device, together with the name of its commands.

It is possible that a particular implementation is not implementing all three forms, but just one or two. However, leaving out the raw format is probably not productive.

### 2.1.2 Internal behavior

It is assumed that suitable hardware is connected to the computer running the sender. Note that it is possible and sensible that a driver implemented on separate hardware (e.g. an Arduino) also communicates using Girs.

A "low level send driver" accepts commands of sending a certain, raw, IR Sequences, using a particular transmitter. A "high level driver" adds the capability to send an IR signal consisting of an intro sequence, a repeat sequence, and (in some cases) an ending sequence a certain number of times, as well as the possibility to interrup an ongoing transmission of a repeating signal.

Timing issues are the sole responsibility of the driver layer. I.e., the upper layers deliver sending requests to the driver, and the driver executes the requests. After a request has been made, a stop request may however be issued (if the software implements it).

In the case of a raw transmission requested, the control logic directs the IR sequence or the IR signal to the send driver.

If protocol/parameter form is implemented, the sender contains a protocol renderer and possibly a data base of protocols, computing the raw signal corresponding to the requested protocol/parameter combination.

If the remote/command format is implemented, the sender has to contain a data base, like the LIRC configuration file(s). If the data base contains the already rendered signals, these can be send directly to the driver. If a protocol renderer is implemented, it is also

possible for the remote/command data base to contain protocol/parmeter data, which are sent to the protocol renderer, which in its turn forwards it to the driver.

## 2.2 The Listener

### 2.2.1 Top level interface

The listener encapsulates the receiving and decoding of IR signals, and its translation into an event or command response.

Often, a received and correctly identified IR signals is expected to generate an event of some kind, for example a Linux input event. Alternatively/additionally, the listener can implement a Girs server, and respond to Receive commands.

### 2.2.2 Internal Behavior

It is assumed that suitable hardware is connected to the computer running the listener. Note that it is possible and sensible that a driver implemented on separate hardware (e.g. an Arduino) also communicates using Girs.

The driver delivers its received signals as sequencies of pulses and gaps. This data is delivered to a *decoder*, which decodes the sequences either as a protocol/parameters set, or a remote/command set, analogusly with the sender, which was described above.

## 2.3 The Dispatcher

An IR system like Lirc not only sends IR signals and reacts on received IR signals, it also invokes other actions, like starting programs etc., in the response to received IR signals. Lirc even can act on other input events, as if they where received IR signals (see the section below. Instead, we here advocate the separation of generating/receiving IR signals and acting on them. The component for handling the events we will call the *dispatcher*. Since the demarcation to general home/computer automation and remote control is unclear, we will not go into any details.

### 2.3.1 Top level interface

The dispatcher can receive events and messages, not only from the IR sender and listener here, but in the general case, also from other sources, like other sensors or input events. From this input, it can generate other events, invoke other programs, send messages over the network, etc.

# 3 Implementation notes

## 3.1 Sender

### 3.1.1 Drivers

There exists a number of "drivers" (or "plugins") for IR hardware for different programs. Unfortunately, these are not always possible to use in another context. For example, the Lirc dynamically loaded drivers are not meant to be used outside of Lirc, for example, they do not implement a simple send of an IR sequence (instead there needs to be a `remote`, and a `command`). To use these drivers (for the reason given, "plugin" is really the better term), it will be necessary to create a "mini-Lirc" to support them. These are by definition for Linux, or at least a Unix-like operating system. Also WinLirc and Eventghost should be examined for its possiblity to "donate drivers".

Our package HarcHardware contains some Java drivers for some IR hardware, that can be used more or less directly.

### 3.1.2 Protocol Renderer

It is fairly straight-forward to write a simple renderer for a particular protocol, like NEC1 or RC5. A very advanced general and extensible renderer is IrpMaster, which is a GPL3 licensed Java program.

### 3.1.3 Remote/Command data base

A format is needed for importing (and possibly exporting) of IR signals. Another format is needed for internally persistently storing the signals, for example to a disk file. These formats may or may not coincide. As external import format, the Girr format is suggested, or possibly a restriction thereof — for example a implementation without a protocol renderer should require that all signals are present in either raw or Pronto Hex (CCF) format.

For migration of Lirc configuration files, IrScrutinizer can be used.

## 3.2 Listener

### 3.2.1 Communication logic

It would be possible to implement a Lirc compatibility mode by writing on the Lirc socket, typically `/var/run/lirc/lircd`. That way, Lirc "client programs" like `irexec`, can be (re-)used.

### 3.2.2 Drivers

Many (most?) drivers for IR receiving hardware are not usable to receive general IR signals (not even with "normal" and known modulating frequency). Instead, they try to

decode the signal itself, the react only on their "own" protocol, and in the case of a match (and only then!), they deliver a decode, typically as an integer. In Lirc, these drivers are called LIRCCODE drivers.

Typically, the hardware is not intended to be a "generic" component, but may be e.g. a TV card with an IR receiver, just intended to react to a bundled hardware remote.

This type of driver does not fit into the model here. Instead, it may be possible to turn such a driver into a "listener" in its own right, sending events ("received command 42 from the TV card") to a dispatcher.

Otherwise, the comments in the sender sections apply here too.

### 3.2.3 Decoding

Decoding can take place either protocol-oriented or command-oriented (trying to determine which one of the know commands that fits). It is believed that the first one is the more systematic, and normally the better approach, so we will only consider it here.

It is fairly straight-forward to write a decoder for a particular standard protocol, like NEC1 and RC5. A very versatile decoder is DecodeIr, knowing over 100 different protocols. It is widely used and tested. Unfortunately, partially due to its chaotic code base, it is effectively not maintainable nor extendable, and its API also has some problems.

## 3.3 Dispatcher

There are a number of possiblities to implement a dispatcher. The Lirc program `irexec` is a simple such. I have also written a (presently not published) simple dispatcher in Java, presently reacting on IR signals received from an Arduino, generating net events etc. as configured from an XML file. The OpenRemote project contains a rule engine based on Drools giving very interesting possibilities for elaborate "dispatching". For Windows users, invoking Eventghost as dispatcher is also an interesting option. This program allows the programming of e.g. if-then-else rules with simple graphic programming. (When will this clever — Python! — program be ported to non-Windows?)

# 4 Comparision with Lirc (Lircd)

The daemon Lircd takes the role of all of the components sender, listener, and (to some extent) dispatcher.

Lirc listens for sending requests either on a Unix domain socket (typically `/var/run/lirc/lircd`), or a TCP socket (default 8765). Sending request can be generated by a Lirc client like the command line program `irsend`. (Another Lirc client, implemented in Java, is found in our HarcHardware package, in the class LircClient. This is integerated in IrScrutinizer.) A sending request contains a remote/command combination, together with a number of repetitions. It will use the data base ("configuration file") to render the IR signal.

When a (reading) client opens the Lirc socket (the Unix domain socket or the TCP socket), Lircd starts listening to IR signals. If a signal arrives, it is tried to decode it to any of the known commands in its data base ("configuration file"). If decoding is successful, the name of the identfied remote/command is written to the communication socket. Lircd may also send events to another daemon (`irexec`) that can invoke other actions, like starting certain programs or invoking other events, like X Window system events. It is even possible to have Lircd injecting events into the Linux input layer.

Using the `devinput` driver and a `/dev/input/event`*`N`* input device, any Linux input device can be cloaked as an IR receiver. This may be an IR receiver with kernel (-module) support (like the IguanaIR or a MCE receiver), but may also be a completely different kind of animal. In this use case, The Linux kernel takes the role of our listener, in some cases even explicitly decoding protocols such as NEC1 and RC5, while the Lircd daemon is nothing else than a dispatcher.