AIR

spaceape

spaceape99@gmail.com

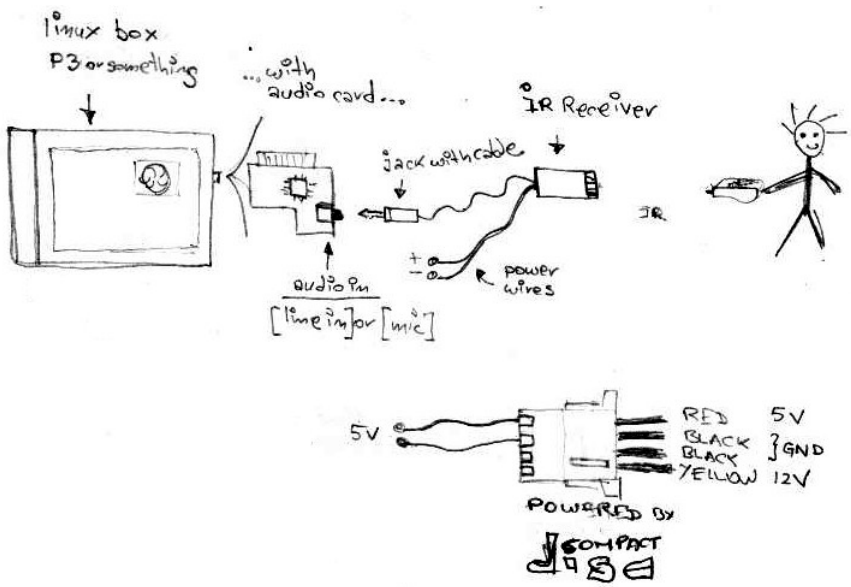
Abstract. AIR stands for „Audio Infrared”. AIR is an effort to enable cheap infrared remote control for computers, using only generic hardware and the PC audio interface as means for signal acquisition.

1. Intro

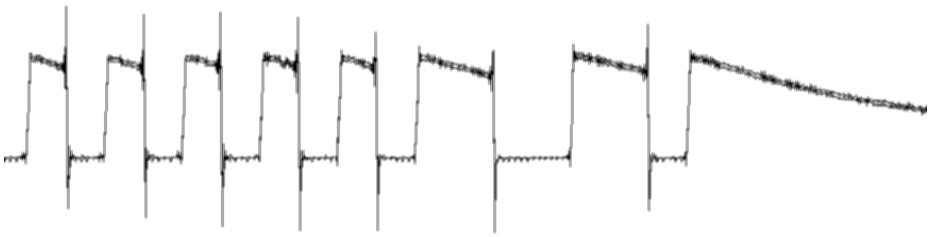
AIR started as a hack for my GNU/Linux desktop computer. The original idea manifested into a rather naïve test program which just decoded the audio and attached functions to the keys based on a plain-text configuration file. Apart from being awfully uninteresting, this “daemon” approach also can't provide any useful realtime feedback (unless one prefers to start it manually at each boot and keep it in a dedicated screen or shell) and for that reason AIR evolved into a KDE Plasma. For testing purposes, the package also builds an executable target – called **airbourne** which incorporates the full functionality of the AIR plasmoid, except for the graphical backend. Throughout this paper, “AIR” actually refers more to the engine rather than to the plasma lib itself.

2. Modus Operandi

The hardware:



Everything up there will work to feed AIR with something it desperately needs: samples of audio.



The IR receiver module is a really simple piece of technology by today's standards. You can salvage an IR receiver from an old TV – you have to be able to recognize **it** and its wiring – or otherwise you can buy or build one yourself without much sweat – an example would be the one here: http://www.lirc.org/ir-audio.html
The most important technical consideration is that (according to AC'97 specs) your module must not output signal of an amplitude higher than $1V_{RMS}$.

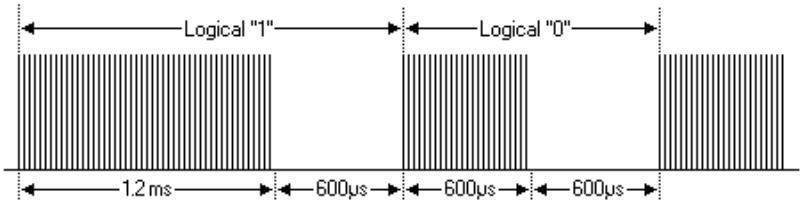
3. Status

Air in currently in pre-release stages – and it will likely stay that way for a while, until it is tested with a resonable number of remotes (and hence – protocols) [[for a list of protocols or a crash course in how IR remotes work, you can check out http://sbprojects.com/knowledge/ir/ir.htm]]. Although it may be relatively simple to write software for a particular remote, the main problem in trying to build an universal receiver like this one is that you have to (at least try to) account for the many different standards that apply to different manufacturers, technologies, etc. For the writing and testing, I (rather randomly) picked a remote from the small mountain that I - and I'm quite sure everyone else - keeps around (it has a large number of keys and the protocol proved to be quite easy to "hack" and nice).

What AIR can do for now is:

a) Manage various signal modulation methods and encoding schemes

There are way too many RC makes/models so it's pretty much impossible to account for all, that's why AIR has a frontend for human-editable config files (.rcd files) that help it make sense out of the received signals. Each individual command coming from a remote can be broken down into more basic pieces that are called "symbols". They can take many forms, depending on which modulation method the manufacturer chose:

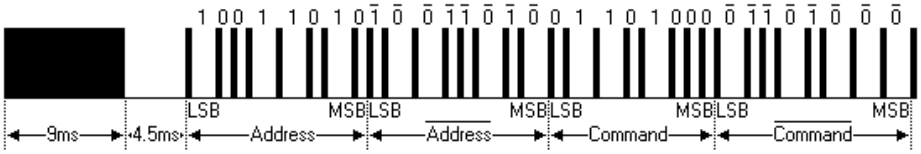


For instance, a pulse-width modulated signal like the one above can be described like this within a .rcd file:

```
#symbol for logical "1"
sym 1 [+1.2 -0.6];
#symbol for logical "0"
sym 0 [+0.6 -0.6];
```

Note: A symbol name is a single alphanumeric char; numeric ones are special, in the sense that they are later used to make up an unique numeric code for each command. The magnitude of the fp numbers within the square brackets tell how long the fronts are (in miliseconds) while the sign tells whether the measured front is on a rising or falling edge – with the mention that the signs don't actually have to alternate, and (although uncommon) the symbols are not limited to just two fronts.
Timings must match with an accuracy of 20% of its length or **else**...

As you might have already gathered, a command is generally made up of strings of such symbols, some of which have a numeric equivalent.



The protocol in the picture sends 32 bits per command (16 for the actual command and 16 check bits) plus a START symbol. according to sbprojects.com it was developed by NEC and it bears striking resemblance to the one my remote uses, which is a Hyundai.

AIR calls a construct like this a "tag" and allows one to define as many of them as necessary (for the average remote one would probably need one or two tags to cover the whole protocol). There is hardly a convention that governs IR protocols, so unfortunately, most of the people are going to have to analyze the signal, "crack" it and write the **tag**s accordingly themselves.
Tags have predefined length and they handle numeric data via the use of **registers**. In our example, there are 4 groups of 8 bits each which correspond to 4 different registers – and they have to be predeclared, say:

```
#register for ADDRESS
reg a = 0;
#register for ADDRESS
reg b;
#register for COMMAND
reg c;
#register for COMMAND
reg d;
```

Register names are made up of a single alphabet character (preferably lowercase) and can optionally specify a numeric (decimal) value to be initialized with.
Tags are hence used to specify a format to be matched by the computer when a command is issued by the remote, as well as to map the numeric symbols to different registers.

```
#here we have a tag that begins with a S (start) symbol followed by 4
groups of 8 #special numeric symbols each (8bits), that are mapped to
registers a, b, c, d


tag Saaaaaaaabbbbbbbbccccccccddddddddd {
…
}
```

AIR uses shift-left and bitwise OR to account for each new bit(s) that are to be mapped to a register. In our example, the remote sends the least significant bit first – hence all the registers will load the bits in a reversed order. Also, note that for this protocol the relevant information (ADDRESS and COMMAND codes) comes in pair

with a verification code, which is in fact the logical complement of the first one. If they fail to match, the command shall be discarded. Again, there is no standard here, and different remote would most likely use a different scheme, with different number of bits and different type of verification if at all. To account for all of these possible variations, for each tag AIR asks you to explicitly tell it how verification (if any) is done and how to pass the information from the registers to the command interpreter, via the use of an assembler-like dialect:

```
tag Saaaaaaaabbbbbbbbccccccccddddddddd {
    swp a;          ← "swap" the bits for registers a, b, c, d
    swp b;
    swp c;
    swp d;
    cpl b;          ← complement b
    equ a b;        ← test if a equals b
    halt z;         ← drop the command if not
    cpl d;          ← complement d
    equ c d;        ← test if c equals d
    halt z;         ← drop the command if not
    send a c;       ← send the command to the interpreter.
};
```

b) load user keymaps, which assign system() commands to various keys on the RC;

Each decoded address/command pair can have a corresponding <node> within an user-defined .xml file. Commands (<c> nodes) are grouped within address (<a> nodes)

```
<root>

    <a id="0"> <!-- group commands with address number == 0 →

        <c id=COMMAND_ID
           topic=A_NAME_FOR_THIS_COMMAND
           set=SYSTEM_COMMAND_TO_LAUNCH_ON_HOST_COMPUTER
           get=SYSTEM_COMMAND_TO_RETURN_A_MESSAGE_FOR_AIR/>

    </a>

</root>
```

Furthermore, AIR allows one to launch different "profiles" (I called them **Apps**) that load different keymaps on top of an already-running profile, allowing some keys to be overridden. For instance, the "Desktop" profile will enable one to launch two other profiles, say "Music" and "Video". Within the "Music" profile, the volume and player keys can be assigned to a music player, while in "Video" they will correspond to a different application. While within one of the latter profiles, user can still access those keys from the "Desktop" profile which hadn't been reassigned.

A piece of a real world keymap it is shown here:

```
<root>
  <a id="0">
    <!-- Player functions -->
    <c id="75"
       topic="STOP"
       set ="dcop amarok player stop"
       get =""/>

    <c id="76"
       topic="PLAY"
       set ="dcop amarok player play"
       get =""/>

    <c id="77"
       topic="PAUSE"
       set ="dcop amarok player pause"
       get =""/>

    <!-- Player volume -->
    <c id="2"
       topic="PVOL+"
       set ="dcop amarok player setVolumeRelative +5"
       get ="dcop amarok player getVolume"/>

    <c id="3"
       topic="PVOL-"
       set ="dcop amarok player setVolumeRelative -5"
       get ="dcop amarok player getVolume"/>
….
```

c) blend in with the other plasmoids on a KDE desktop, look awesome and do what is supposed to:

This is a screenshot of my KDE panel, where AIR is sitting quite comfortably. Text is actually animated, in an old-schoolish ticker-like manner.

4. Objectives

- proper documentation;
- to test a decent number of remote-controls and include corresponding (.rcd) definition files for reference;
- to create some sort of AI which will attempt “guessing” the protocol, encoding scheme, etc from bare audio signal with minimal user intervention
- to provide some sort of graphical interface for configuration
- although i'm not really a fan of these, some sort of OSD popup window could prove useful
- to transform those cumbersome multimedia keyboards into trash

5. Summary

Cons	Pros
* a performant ADC like the capture interface of a soundcard is a slight overkill for the purpose of acquiring IR signals	Too many to count
* as you can see, AIR keeps a list of system commands in plain text format – it might just not be a good ideea to run **Air** as root	
* can occasionally give your skype or ym friends a different kind of buzz	