HARCToolbox

Table of contents

1 Home

1.1 Welcome to the HARCToolbox project!

Date	Description
2011-01-10	Initial version. Creation of domain www.harctoolbox.org.
2012-05-01	Incorporated IrpMaster and IrMaster as sub projects, just as lirc2xml (moved from my personal home site) and the LIRC CCF patch (published for the first time). General reconstruction of site. Release of version 0.1.2 of IrpMaster, version 0.1.2 of IrMaster, version 0.7.0 of Harctoolbox (unfinished...), as well as the version 0.1.2 of lirc2xml.
2012-06-07	Updated for the release of version 0.2.0 of IrMaster and IrpMaster. The document on transforming XML new.
2012-08-19	Updated for the release of version 0.3.0 of IrMaster and release 0.2.1 of IrpMaster.
2012-11-18	Updated for the release of version 0.3.1 of IrMaster and release 0.2.2 of IrpMaster.
2014-02-02	Reorganized, for version 1.0.0 of IrScrutinizer, etc.
2014-06-12	Updated for the new releases: version 1.1.0 of IrScrutinizer, version 1.0.1 of IrpMaster, IrMaster, and Girr, version 0.3.0 of Jirc, version 0.9.1 of HarcHardware, version 0.2.1 of GuiComponents.
2014-09-27	Updated for the new releases: version 1.1.1 of IrScrutinizer, version 1.0.2 of IrpMaster, IrMaster, and Girr, version 0.9.2 of HarcHardware, version 0.2.2 of GuiComponents.
2015-04-16	Updated for the new releases: version 1.1.2 of IrScrutinizer, version 1.0.3 of IrpMaster, IrMaster, and Girr, version 0.9.3 of HarcHardware.
2015-09-10	Updated for the new releases: version 1.1.3 of IrScrutinizer.
2016-01-08	Arduino hardware new pages. Glossary extended and improved.
2016-04-20	Updated for the new releases: version 1.1.3 of IrScrutinizer. New page IR resources. Glossary extended.

Date	Description
2016-05-04	New page Infrared2Arduino and its API docu.
2016-08-30	New version 1.3 of IrScrutinizer.
2017-03-16	New version 1.4 of IrScrutinizer.
2019-08-12	New entry IrpTransmogrifier. New version 2.0.1 of IrScrutinizer. Deleted Jirc page. (no longer existing as command line program (Lirc2Xml), however still existing as API and component of IrScrutinizer.).

Table 1: Revision history

> **Note:**
> New program IrpTransmogrifier. New release of IrScrutinizer 2.0.1.

1.1.1 History

Since 2006 I have been writing software, designed file formats, and classified remote control signals, revolving around infrared remote control and home automation. The old "main project" (old harctoolbox) is the original project. It deals with descriptions of IR protocols and signals, device descriptions, and descriptions of their interconnection.

In early 2011 I came to the conclusion, that my work on IR protocols was a dead end street. It was better to connect to and use the Internet knowledge, notable the so-called IRP notation, that was excellently formalized by Graham Dixon in early 2010. This started the IrpMaster project, in which the IRP notation was completely implemented (with one exception, which is rather a specification flaw than an omission of practical importance).

For IrpMaster, the goal was correctness and completeness, without any compromises for user friendliness. In particular, it does not contain a graphical user interface. Its command line interface appears as inaccessible to the GUI-centered user. Instead, the project IrMaster was started, to allow a user friendly, interactive access to the functionality of IrpMaster. As a graphical "meta program", it also fulfills some other tasks, see its documentation.

Early 2013 I planned to extend IrMaster with more functionality, in particular the possibility to capture IR signals and to import IR signals from many other sources and file formats. Originally, I planned to extend IrMaster with this new functionality; however, I decided to start on a brand new program, IrScrutinizer, which basically combines almost all that fits in an IR program. One of the reasons was that I was unhappy with the somewhat suboptimal code structure of IrMaster. The first version was released on November 2013. IrMaster was declared as no longer developed, promoted to version 1.0.0, which was released in February 2014.

In the light of these projects, the main project harctoolbox has gone slightly "stale", and it needs to be restructured.

The program *lirc2xml* was posted on the LIRC mailing list in 2009 (without any response :-), and has been residing on my private web page since then. For IrScrutinizer, I wanted this functionality incorporated. A natural way would be to turn lirc2xml to a shared library with JNI interface. However, discouraged by the experiences from DecodeIR, I wanted a pure Java solution, and ported the part of LIRC needed to parse and interpret the configuration files to Java. This is the Jirc project.

LIRC CCF: The idea to modify the LIRC server to allow for directly sent CCF signals was originally sent to me in private mail from Bitmonster (of Eventghost). I made a preliminary version some time later, this was communicated to him and submitted to the then-maintainer Christian Barthelmus, who promptly rejected it (some more details in the documentation). This is its first real "publishing".

There are presently over 80000 lines of source code in Java on this site, code in other languages, or by other authors not counted.

As a working project name, as well as in the Java module names, the name *Harc*, which is simply an acronym for "Home Automation and Remote Control", was used. Unfortunately, that name is far too generic to register as an Internet domain. There is even a Sourceforge project named Harc, (inactive since 2002). For this reason the project was renamed to *HARCToolbox* early 2011, and the domain www.harctoolbox.org created.

1.1.2 Overview

IrpMaster is a very advanced renderer of infrared signals. That is, from an abstract description consisting of a protocol name, and some parameter values, it computes the corresponding infrared signal, as a sequence of gaps and flashes. It is intended to be used both from its API, as well as an interactive (command line) program, both offline (generating files etc) as well as in real-time applications (generating signals to be sent to hardware equipment). The program does not attempt to be "user friendly", and does not, e.g., contain a graphical user interface. *It is now obsoleted by IrpTransmogrifier*.

IrpTransmogrifier is a library and program for IRP notation protocols, with rendering, decoding, analyzing, and code generation applications. This is a new program, written from scratch, that is intended to replace IrpMaster, DecodeIR, and much more. The

project consists of an API library that is also callable from the command line as a command line program.

IrMaster's main purpose is to be a user friendly user interface to IrpMaster. But it does not end there: it also integrates/interfaces with other projects, like the alternative, "classical" IR renderer Makehex, the DecodeIR "inverse renderer", the analyzeIR library, the Harctoolbox main project etc. It can generate export files in different formats, it can address networked hardware directly (presently GlobalCaché, IRTrans LAN module, and a patched LIRC server. (Please note that *IrMaster* and *Ir**p**master* are two different projects. IrMaster depends on IrpMaster, but not vice versa.) *It is now obsoleted by IrScrutinizer*.

IrScrutinizer is a powerful program for capturing, generating, analyzing, importing, and exporting of infrared (IR) signals. For capturing and sending IR signals several different hardware sensors and senders are supported. IR Signals can be imported not only by capturing from one of the supported hardware sensors (among others: IrWidget, Global Caché, and Arduino), but also from a number of different file formats (among others: LIRC, Wave, Pronto Classic and professional, RMDU (partially), and different text based formats; not only from files, but also from the clipboard, from URLs, and from file hierarchies), as well as the Internet IR Databases by Global Caché and by IRDB. Imported signals can be decoded, analyzed, edited, and plotted. A collection of IR signal can thus be assembled and edited, and finally exported in one of the many supported formats. In addition, the program contains the powerful IrpMaster IR-renderer, which means that almost all IR protocols known to the Internet community can be generated.

There are some support projects, independent in the sense of a separate package. First, there is the subproject HarcHardware, containing a number of classes for hardware access. This is believed to be of interest also for other projects. Secondly, the subproject GuiComponents (no documentation page present!) contains a number of, in principle, recyclable components, often in the form of Java Beans. Although it was written and conceived as a support project for IrScrutinizer, it is not unlikely that they can be used directly in other programs.

As described in the introduction, the main project is presently slightly "stale".

In the context of the LIRC project, there is a large body of recoded infrared signals in "LIRC format". For many users of home automation and remote control, it would be desirable to use this knowledge outside of the LIRC framework. Unfortunately, the LIRC format (`lircd.conf`) is is all but well documented. The program lirc2xml therefore uses LIRC itself to decode and export its signals, here to (somewhat arbitrarily) an XML file, that can be further processed.

Finally, the LIRC patch for CCF signals describes an enhancement to the LIRC server.

1.1.3 Content

- Arduino Nano IR hardware, part 1 and part 2.
- IrScrutinizer, very powerful general purpose IR program.

- IrMaster, a predecessor to IrScrutinizer.
- IrpTransmogrifier, the new IR renderer/decoder/analyzer.
- IrpMaster, the old and obsolete IR renderer.
- Glossary and terms.
- HarcHardware, hardware related functionality.
- Girr, a specification of a general exchange format for IR signals
- Generic IR Server, a specification of a general IR server API
- An annotated list of IR resources on the Internet.
- HarcToolbox, the old main project.
- Transforming XML Export from Ir(p)Master, a tutorial article on generating "interesting stuff" (here, C code) from the XML export. Applies to IrpMaster and IrMaster, not IrScrutinizer, thus somewhat obsolete.
- Lirc2xml, a program for extracting IR codes from LIRC files. Predecessor to Jirc, thus obsolete.
- Improved lirc driver for the Raspberry Pi.
- Raspberry Pi daughter board for IR and RF transmission and reception.
- LIRC CCF patch, to make the LIRC server more usable.
- Relationship to other, similar projects.
- Downloads, mostly obsolete.

1.1.4 Other links

- IrScrutinizer tutorial. Aimed at the beginner and quite easy to read.
- SB-Projects' IR Remote Control Theory. A very good place to start leaning about IR protocols. Covers the basics of infrared signals, and its modulation. Not too long, not too "dummy"-oriented.
- JP1 wikiMany articles, of different ages and qualities, are collected here.

1.2 NEWS!!

1.2.1 Latest news!

New versions (1.2) of IrScrutinizer etc. New page: IR resources.

1.2.2 History

2011-01-10
Initial version. Creation of domain www.harctoolbox.org.
2012-05-01
Incorporated IrpMaster and IrMaster as subprojects, just as lirc2xml (moved from my personal home site) and the LIRC CCF patch (published for the first time). General reconstruction of site. Release of version 0.1.2 of IrpMaster, version 0.1.2 of IrMaster, version 0.7.0 of Harctoolbox (unfinished...), as well as the version 0.1.2 of lirc2xml.
2012-06-07

Updated for the release of version 0.2.0 of IrMaster and IrpMaster. The document on transforming XML new.
2012-08-19
Updated for the release of version 0.3.0 of IrMaster and release 0.2.1 of IrpMaster.
2012-11-18
Updated for the release of version 0.3.1 of IrMaster and release 0.2.2 of IrpMaster.
2014-02-02
Whole site reworked. New programs IrScrutinitzer, Jirc. New libraries Girr, HarcHardware. New hardware support Improved Lirc driver for Raspberry Pi. New articles Girs IR server specification, description of a IR/RF daughter board for the Raspberry Pi, and a Glossary. New versions of IrpMaster (1.0.0) and IrMaster (1.0.0).
2014-06-12
New versions of IrScrutinizer, IrpMaster, Girs, Jirc.
2014-09-27
New versions of IrScrutinizer, IrpMaster, Girs, GuiComponents, HarcHardware.
2015-04-16
New versions of IrScrutinizer, IrpMaster, Girs, GuiComponents, HarcHardware.
2016-01-08
New pages: Arduino Nano IR hardware, part 1 and part 2.
2016-04-30
New versions (all called 1.2) of IrScrutinizer, IrpMaster, Girs, HarcHardware. New page: IR resources.

1.3 Interaction with other projects

1.3.1 Revision history

Date	Description
2011-01-09	Initial version, derived from old `harc.xml`.
2014-02-02	Update.

1.3.2 Introduction

HARCToolbox interacts with other projects within the area. It can be pointed out that in the case of Java projects, Harctoolbox uses unmodified jar-files; in the case of shared libraries (`.so` or `.dll`) these are also used in an unmodified state. In no case, Harctoolbox "borrows" code from the projects. Also, in this way additional functionality is implemented, none of which is of essential (like import/export of a certain file format). Differently put: should the need arise to eliminate "derivedness", only minor, nonessential functionality will be sacrificed (or needs to be implemented anew).

Also see the copyright notices in the individual programs/projects, where more third-party software is listed.

1.3.3 LIRC: Linux Infrared Remote Control

LIRC is a well established, mature, and active free software project. It is used in very many free software projects. It contains support for a large number of infrared senders and receivers, some sane hardware designs, other possibly less sane. There are also a large number of user contributed configuration files for different IR remote controls and devices, in general consisting of leaned commands. A network enabled LIRC server consists of the software running on a host, listening on a network socket, containing one or more IR transmitter or transmitter channels. A client sends requests to, e.g., transmit a certain command for a certain device type. Since Harctoolbox can talk to a network LIRC server (see source in the file `lirc.java`), there is a large number of IR senders that Harctoolbox in this way "indirectly" supports. Unfortunately, the configuration files are residing on the LIRC server only; there is no way to request the transmission of a signal the server does not know in its data base. (A patch for this was submitted by myself, but rejected by the maintainer. It is available here.

From its IR data base, Harctoolbox can generate configuration files for LIRC. IrScrutinizer, using Jirc, which is a translation of LIRC to Java, LIRC files can be imported and translated to other formats.

LIRC is licensed under GNU General Public License, version 2 or later. However, Harctoolbox is not a derived work; it contains no LIRC code, and is not linked to any libraries. It optionally "talks" to a LIRC server, but this functionality is entirely optional.

1.3.4 JP1

The JP1 project is a very unique project. It aims at complete control over the remotes made by Universal Electronics (UEIC), which include the brand names "One For All" and "Radio Shack". Through careful study of its hard- and firmware, techniques for custom programming a remote, equipped with a 6-pin connector (on the remote's PCB called "JP1", giving the project its name) was developed. Thus, an off-the-shelf remote can be taken much beyond its original capacities. Most importantly, it can be programmed from a computer to generate "arbitrary" IR-signals.

RemoteMaster is a program for, among other things, creating so-called device updates. These device updates can be produced by Harctoolbox, as rmdu-exports. Thus, for an IR-controlled device in the Harctoolbox database, a suitable JP1-enabled remote control can be made to send the appropriate IR-signals. (There are some details, that will be documented somewhere else.) RemoteMaster is presented as an interactive GUI program, however, it can also (although this is not supported) be called through a Java API. Harctoolbox presently uses version 1.89, which is not the current version. Although it seems to lack all copyright notices, it is referred to as "open source" and GPL.

IrScrutinizer can "almost" import and export RemoteMaster's device upgrade files. Since this is dependent on some rather intricate protocol dependent transformations between the JP1-specific "EFC"-numbers and the function codes, I consider it as a better idea to extend RemoteMaster to import and export the general Girr format of IrScrutinizer.

Another tool from the JP1 project is DecodeIR by John Fine, available under the GNU General Public License, version 2 or later. It consists of a shared library (`DecodeIR.dll` or `DecodeIR.se`), written in C++, together with a Java wrapper (`DecodeIR.jar`). To build that jar file, also this file is needed. The tool attempts to decode an IR-signal in CCF form into a well known protocol with device number, command number, possibly subdevice number and other parameters. See the IR protocols pane in the GUI.

1.3.5 IRScope

Together with appropriate hardware, the Windows program IRScope by Kevin Timmerman is very useful to "record", and optionally analyze unknown IR-signals (again, using the same DecodeIR as above). The log files generated by this program can be directly parsed, see the code in `ict_parse.java` or the IR protocols pane. The program is licensed under GNU General Public License, version 2 or later. Harctoolbox neither uses code or links to it, and is not a derived work.

IrScrutinizer can replace IrScope.

1.3.6 Tonto

Tonto is normally referred to as an alternate configuration ("CCF") editor for the first generation of the Philips Pronto remote controls. It is a free replacement for the original ProntoEdit program, written by Stewart Allen, licensed under the GNU General Public License, version 2. Being written in Java, it runs "everywhere", in contrast to the original Windows-only program. It also contains a Java API. Harctoolbox and IrScrutinizer use the Tonto API (in the form of the file `tonto.jar`) to import CCF files, and to generate CCF files for devices in its data base. (The latter are supposed to be more of a target for aliasing, than a directly usable user interface.) Unfortunately, the project is inactive since 2004.

1.3.7 IRDB

IRDB is a web site that describes itself as "one of the largest crowd-sourced, manufacturer-independent databases of infrared remote control codes on the web, and aspiring to become the most comprehensive and most accurate one". IrScrutinizer can directly import a set of IR signals from this data base. Interestingly, the site uses software from this site extensively.

1.3.8 wakeonlan

Harctoolbox uses wakeonlan (licensed under the GNU Lesser General Public License), a small Java library for implementing WOL-functionality.

1.3.9 Java Readline

The interactive command line uses the Java Readline (licensed under the GNU Lesser General Public License), which of course only makes sense when used in conjunction with the GNU Readline library, which is licensed under GNU General Public License, version 2 or later.

1.3.10 Sunrise/Sunset

Harctoolbox uses the sunrise/sunset Java library (see also Blog) by LuckyCat Labs/Mike Reedell (licensed under the Apache License 2.0). (which is considered as "compatible" with GPL3). This is a Java library for computing sunrise and sunsets. I have implemented some improvements, which I intend to publish shortly.

1.4 Impressum

Responsible for this site is Dr. Bengt Martensson, webmaster@harctoolbox.org. With the exception of my other two sites (www.bengt-martensson.de and www.bengt-martensson-consulting.de), I am not responsible for the content of linked sites, over which I have no control. At the time of the original publishing, no illegal or otherwise questionable content on linked sites was known to me.

1.5 Legal

2 Current

2.1 Current program and articles

Date	Description
2014-02-02	Initial version.
2016-01-08	Updated for Arduino hardware articles.
2016-04-30	Updated with IR resources.
2019-08-12	Moved IrMaster and IrpMaster to old. IrpTransmogrifier new entry. Deleted Jirc.
2022-05-13	Moved Lirc CCF, lirc_rpi and rpi_daughterboard to old. IrpTransmogrifier new entry. Deleted Jirc.

Table 1: Revision history

2.1.1 Content

This page consists of a table of content of currently maintained programs, libraries, and articles.

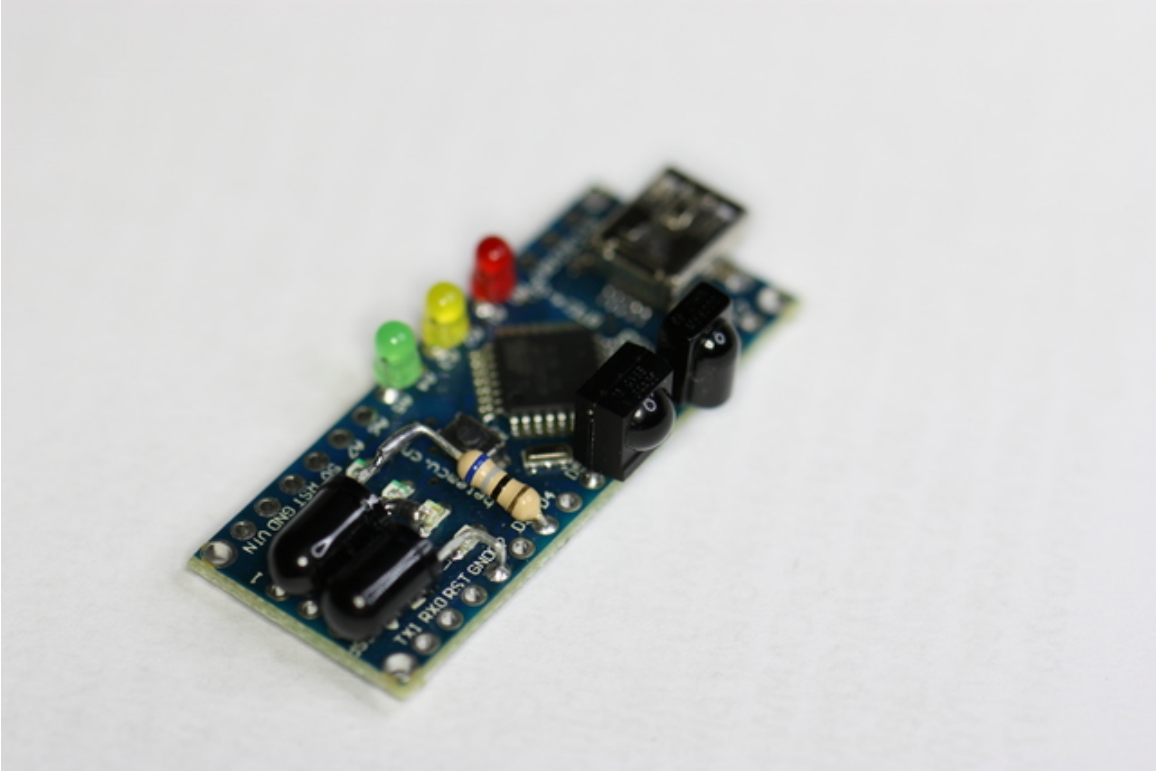
- Arduino Nano as IR sender and receiver.
- Arduino Nano as IR sender and receiver, second part.
- IrScrutinizer, a very advanced IR program.
- IrpTransmogrifier, a parser for IRP notation protocols, with rendering, code generation, recognition applications, as a library and command line program.
- HarcHardware, a number of Java classes for hardware related functionallity.
- Girr (General InfraRed Remote format), a specification for a general IR signal exchange format. Also a library for its implementation.
- Girs (General InfraRed Server), a specification for a class of IR servers.
- Glossary and terms.
- IR signal resources on the Internet. An annotated collection of links.

Downloads.

2.2 Arduino Nano as IR sender and receiver

Date	Description
2016-01-08	Initial version.
2016-02-10	Fixed some broken links. Uploaded some missing pics.
2020-04-21	Minor updates of the things changed during last four years.

Table 1: Revision history



2.2.1 Introduction

In this article, we present a simple, yet quite powerful, microprocessor-based IR sender and receiver as a do-it-yourself project. It is particularly geared towards IrScrutinizer and Lirc, but other usages are also possible.

As presented here, there are components for IR transmission, demodulated- and non-demodulated IR reception, and signaling LEDs. All of these are optional.

For the ease of assembly, components are soldered directly to the holes of the PCB. For pins that should be connected to ground or to +5V, these are sometimes GPIO pins, which in software are assigned a constant LOW or HIGH level.

The current article is aimed at beginners. As a DIY project, it can be considered "simple to medium" on a scale from very simple to very hard. Elementary experience with soldering of electronic components is assumed, as well as installation of system components in Windows or Linux. It is also necessary to be able to program the flash memory of the Arduino with its the Arduino IDE. However, no programming knowledge is required.

The price for the components is around 8 EUR or 9 USD.

There is a second part of this article, covering some details that, for easy accessibility by inexperienced readers, were left out of the current work.

2.2.2 Component selection

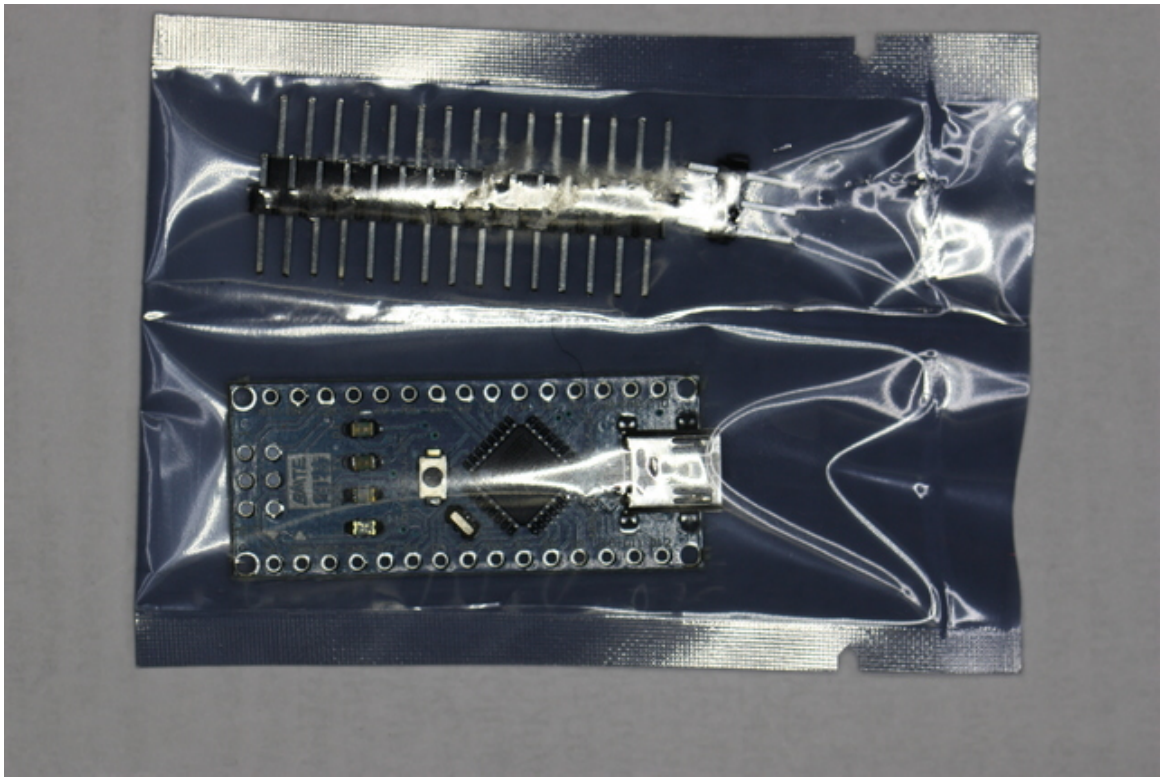
For the ease of reading, this section is written without mentioning alternatives. These are instead discussed in the second part. For the same reason, links to data sheets for the mentioned components is found only in that part.

The appendix contains the complete bill of materials.

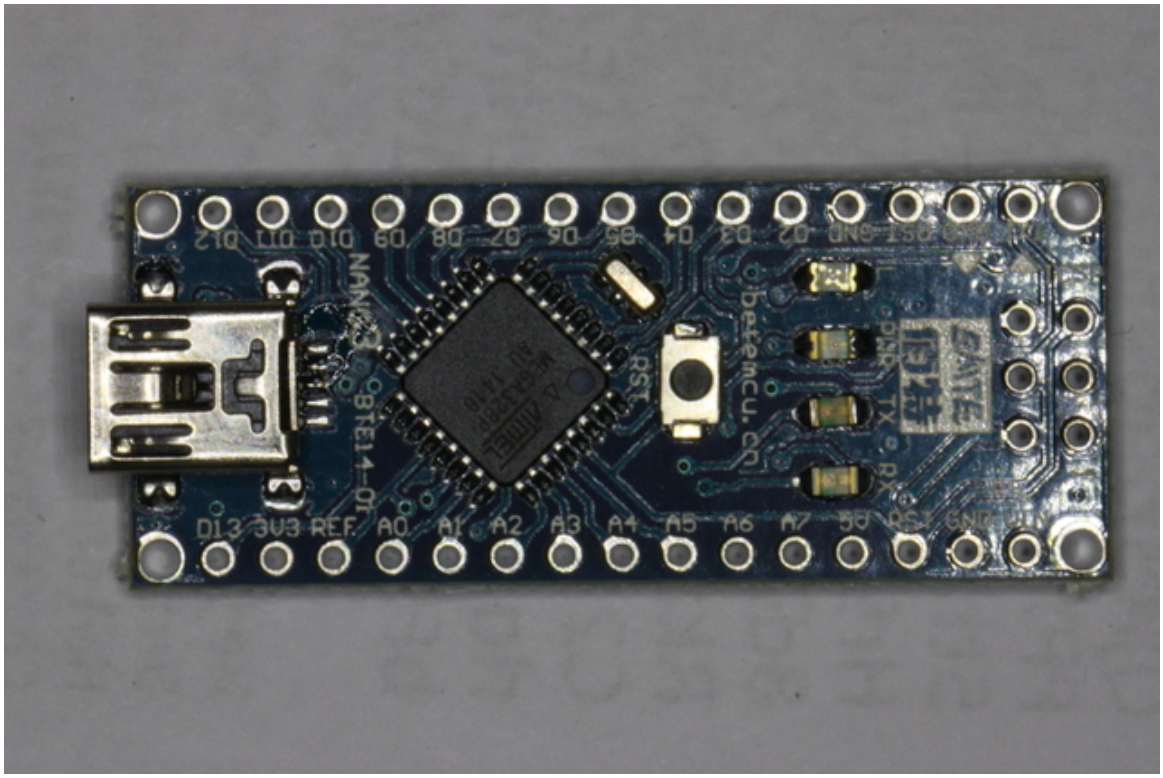
2.2.2.1 Processor board

As processor board, we select an Arduino Nano-clone (Version 3.0 or 3.1). These can be purchased e.g. on Ebay for a very low price (< 3 EUR) from Asian manufacturers. To be useful for us, it should come with a mini-USB socket, and with bare holes (i.e. without pins) at the sides, see the photos. It should also be equipped with an ATmega328 5V, but that appears to hold for everything currently offered.

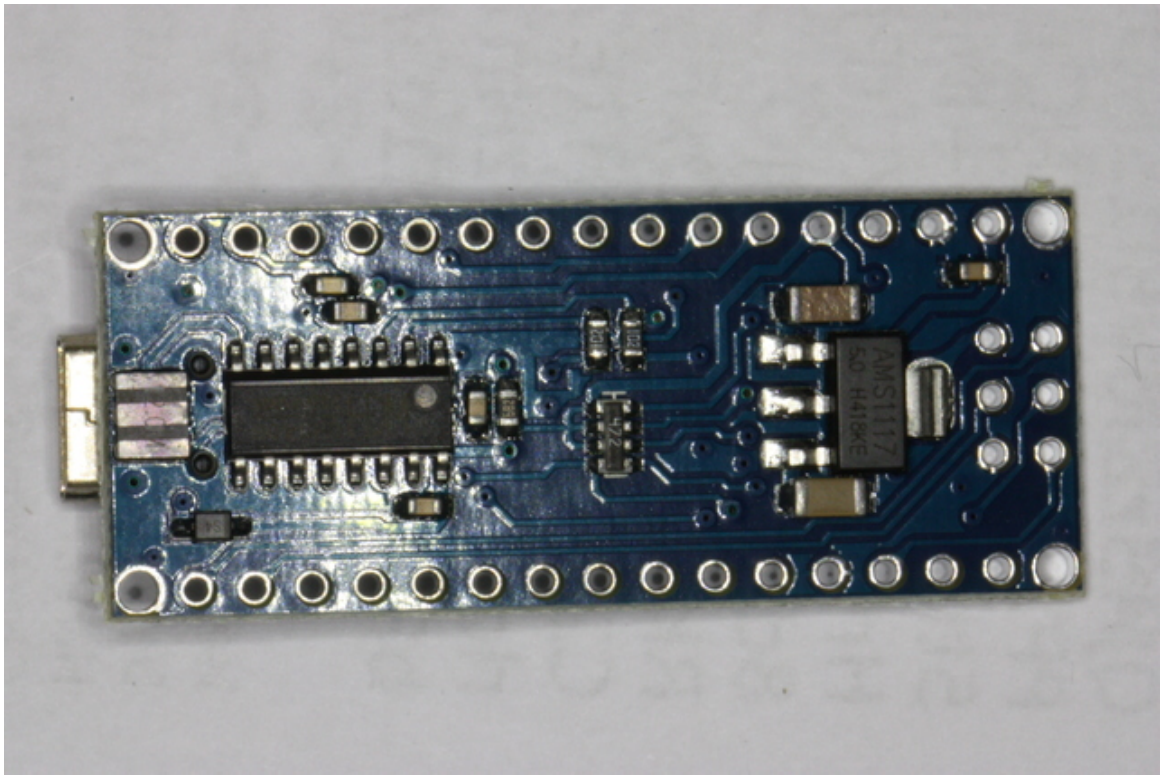
> **Note:**
> The delivery from an Asian seller using the cheapest shipping option may take a month or longer.



The Arduino Nano clone as delivered in an anti-static bag. The solder pins in the upper part of the packing are not needed, and can be disposed.



The board unpacked. Note the mini USB connector to the left.

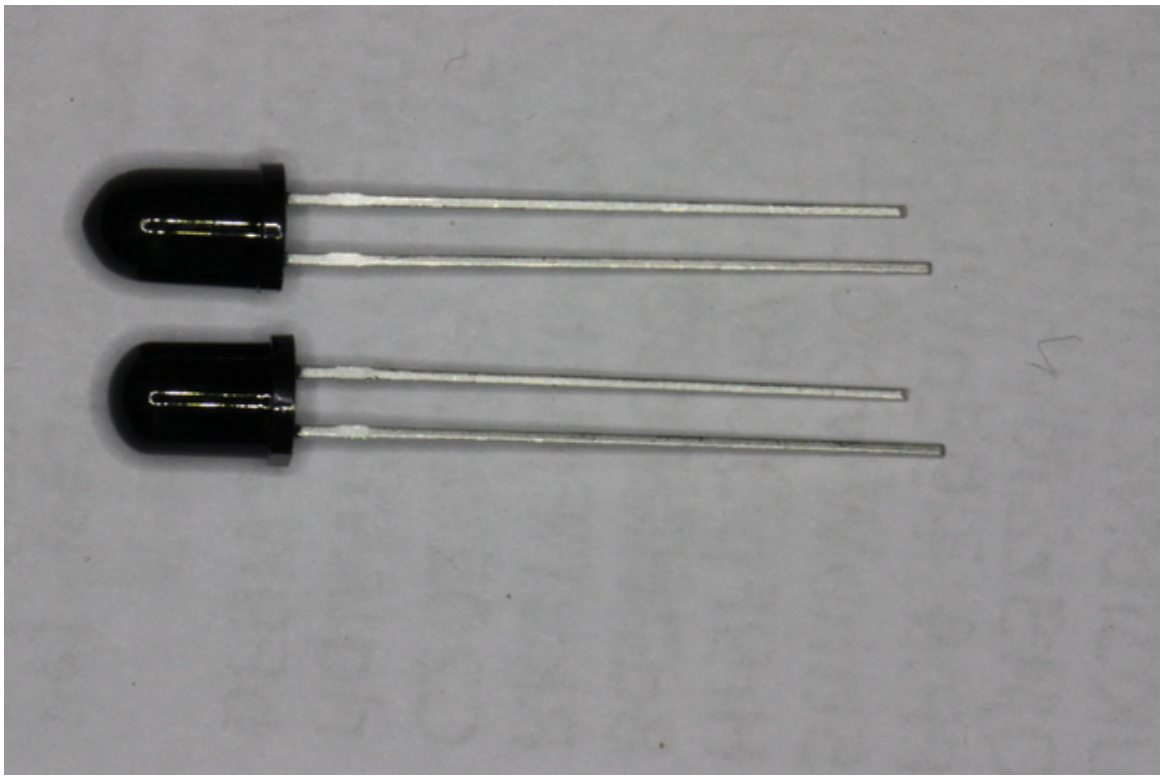


The same board from below.

All of the following components are optional, and can be left out if a certain functionality is not desired. Nevertheless, I recommend incorporating all. The components are really not that expensive.

2.2.2.2 IR sending diodes

For maximal sending performance, we use one IR LED with narrow beam together with one with wider beam, thus combining the advantages. We select the Osram SFH4544 and SFH4546. These are connected in serial, together with a 68 ohm resistor. If sending is not required, these can be left out.



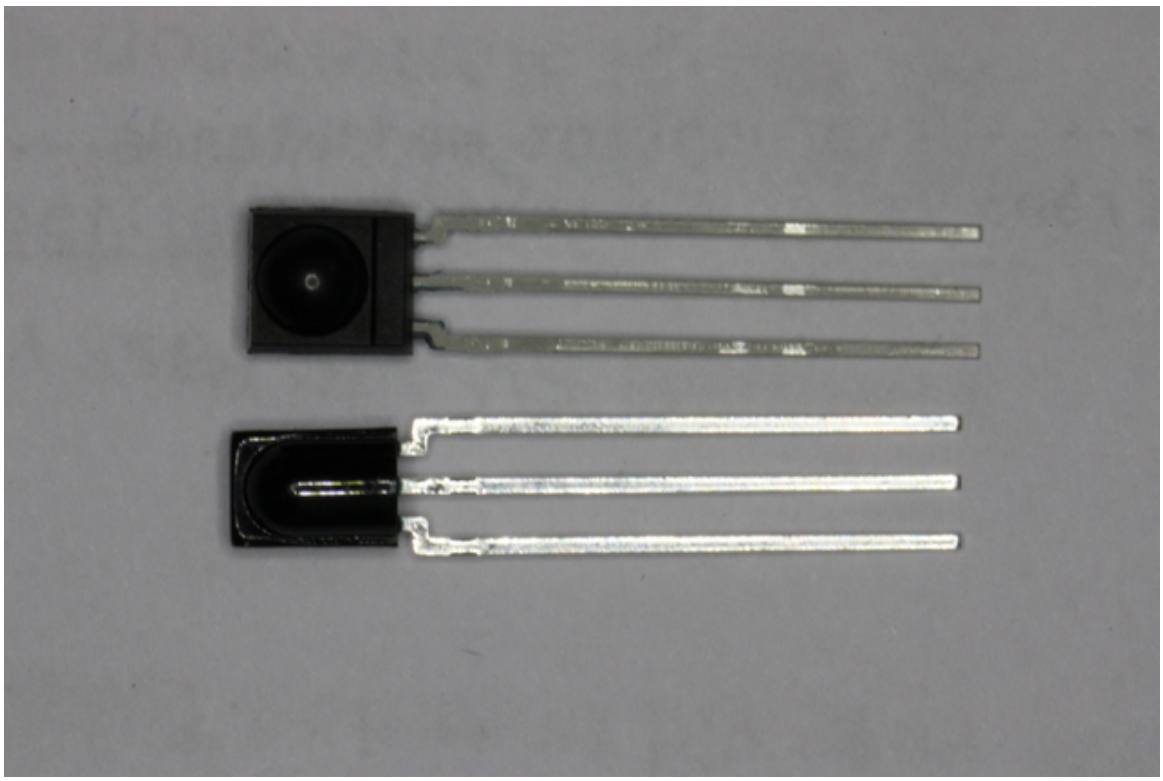
The IR LEDs before assembly. It can be seen that the SFH4544 has a slightly longer housing than the SFH4546. Also note that, as opposed to most other LEDs, the cathode pin is the longer one.

2.2.2.3 Non-demodulating sensor

This allows for very accurate measurements of a possibly completely unknown IR signal, including measurement of the modulation frequency. This use case is called *capturing* or *learning*. It is not well suited for deployment reception of IR signals, for example in the context of Lirc. (Lirc cannot use this sensor.) We select the Vishay TSMP58000.

2.2.2.4 Demodulating Receiver

This is meant for deployment reception of known IR signals, allowing reception over a very long distance. This use case is called *reception*. It is not very well suited for "learning". We select the Vishay TSOP34438.



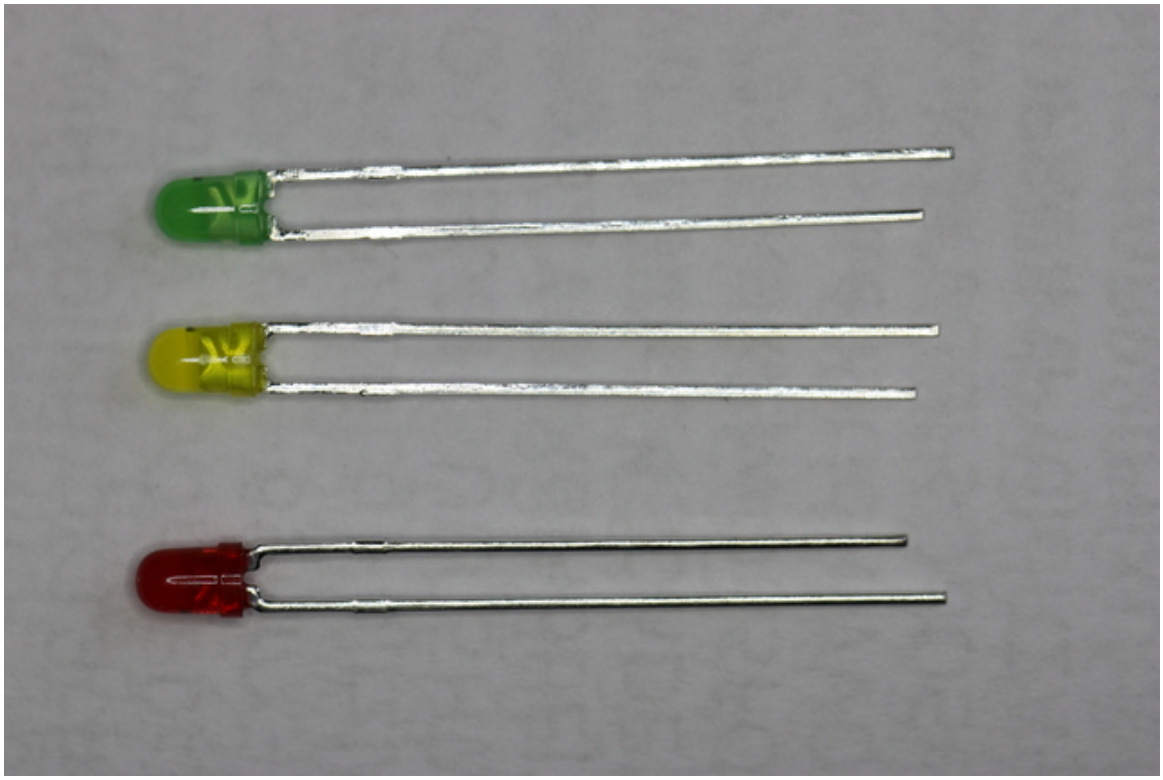
The demodulating sensor TSOP34438 (top) and the non-demodulating sensor TSMP58000 (bottom).

2.2.2.5 LEDs (for visible light)

LEDs must always be connected using series resistors. Since discrete resistors would increase the component count, and thus the assembly difficulty, instead we use LEDs with built-in resistors. The 3mm 5V LEDs from Kingbright fit perfectly into the Arduino holes, and are not very much more expensive than normal LEDs. They are available in red (WP710A10ID5V), yellow (WP710A10YD5V), and green (WP710A10SGD5V); unfortunately not in other colors.

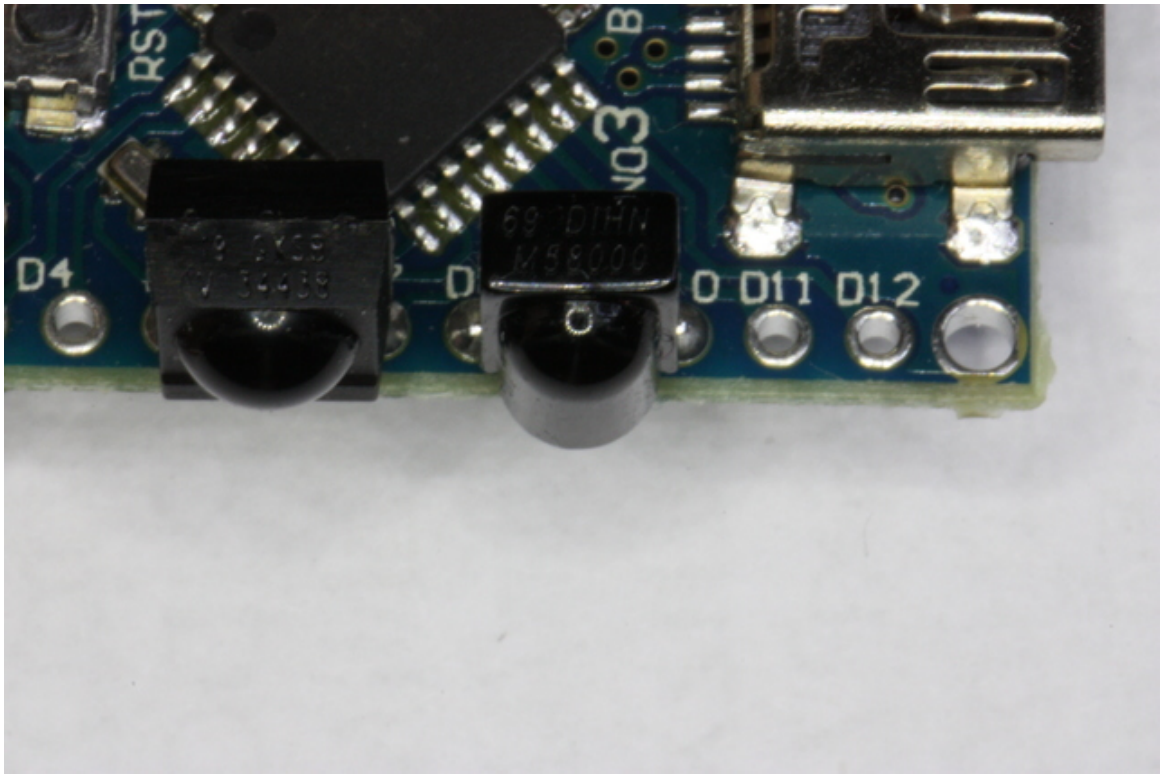
Here, we use a red LED as the first one, a yellow one as second, and a green one as third.

These are purely for signaling and debugging purposes, and can be left out without any functional penalty.

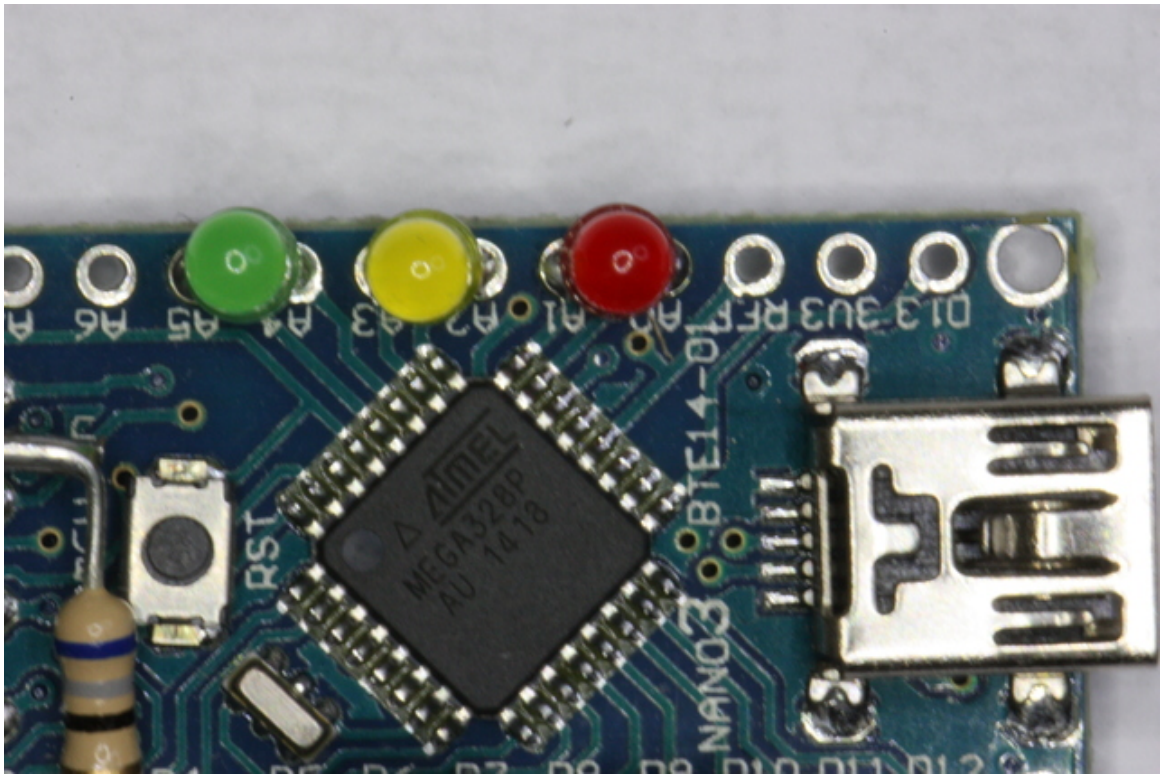


2.2.3 Assembly

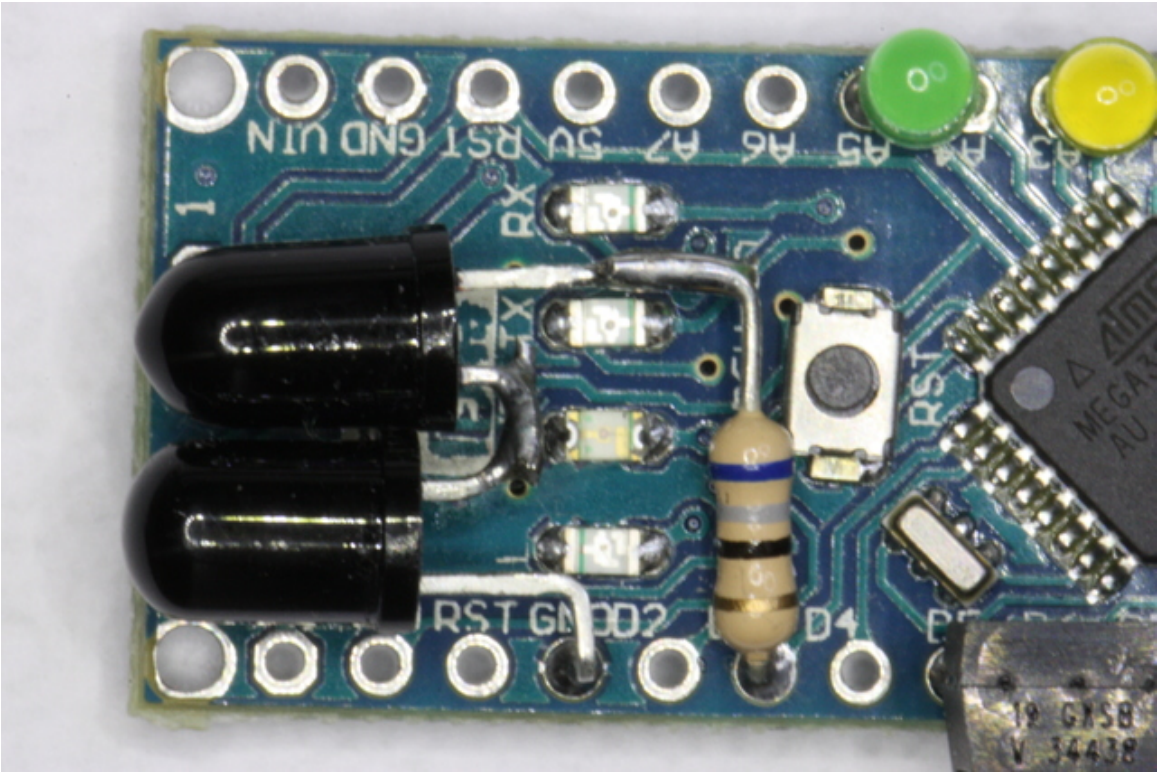
The TSMP58000 goes to the holes D8, D9, and D10; the lense facing outwards (see pictures). Likewise the TSOP34438 goes into pins D5, D6, and D7.



Then, the red LED is soldered to pins A0 and A1. Unfortunately, there is no marking on the housing indicating anode or cathode. Instead, the cathode is the shorter pin, and should go to A0. Likewise, the yellow LED go to pin A2 (shorter pin/cathode) and A3. The green LED go to pin A4 (cathode/short) and A5.



The IR LEDs are slightly harder: The cathode of one (does not matter which) should be connected to the "GND" hole, the forth hole from the right in the upper row (assuming that the mini-USB connector points to the left). Then the anode should be connected to the cathode of the other one. Finally, the anode of the other one should be connected through the resistor to pin D3. On the Osram IR LEDs, the cathode is marked by a flat area on the housing. Here is obviously room for some creativity. Consider my pictures as a suggestion only.



Note:
On the Osram IR LEDS, the cathode pin is the longer one, as opposed to most other LEDs(!!).

2.2.4 Drivers etc

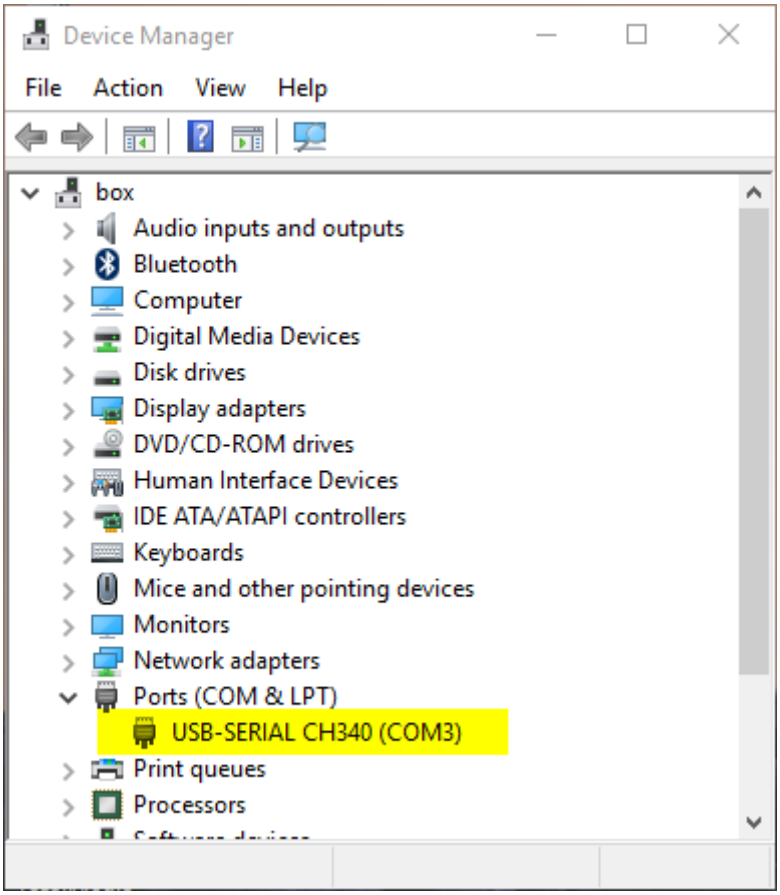
To be used by any program, the device has to be recognized by the operating system by a driver. On Windows, it will be available as a COM device, like `COM12` or such, on Linux and MacOSX a device like `/dev/ttyXXXX`, acting like a serial device.

2.2.4.1 Windows

While the original Nanos use FTDI FT232RL for serial communication, and should be used with FTDI drivers, this is both technically and legally not possible or not feasible for the clones, which instead use a CH340 chip.

My Windows 10 system identifies these boards automagically as `USB-Serial CH340` in the device manager, see the screen-shot below. Using other versions of Windows, it may be necessary to download and install a `CH340` driver manually.

The device shows up in the device manager as in the following screen shot (marked yellow):



Note the device name that the Windows has assigned to it, here `COM3`.

2.2.4.2 MacOS X

This is presently a somewhat tricky issue, and is discussed on a multitude of places in internet. This page contains both a seemingly working driver and some discussion. This will make the board available with a name like `/dev/ttywchusbserialXXX` (for `XXX` a three digit number).

2.2.4.3 Linux

It seems like all Linuxes I have tried automatically detects the board and makes it available as `/dev/ttyUSB0` (or `/dev/ttyUSB1` etc if the first one is already taken). Use the shell command `ls /dev/tty????` to find out exactly which. The device is typically only accessible for root and the members of the group `dialout`, see Appendix.

2.2.5 Firmware

For getting the firmware onto the processor, ether the sources can be compiled, or a binary directly uploaded to the board (often called "flashing"). Being a "dummies" guide, we only cover the second method here. The first method is covered in the second part.

For us, a compiled binary is a text file in a special format (called a hex file). A suitable such is provided here.

In all cases, a program, *avrdude*, is needed to write the date to the flash memory of the board. This must first be installed.

2.2.5.1 Windows

For Windows, the simplest way to install avrdude is probably to install the Arduino IDE. To flash the firmware, download GirsLite-1.0.2-nano-flasher.bat, and make the necessary changes of the pathname of avrdude and the COM-port where the board resides.

2.2.5.2 MacOS X

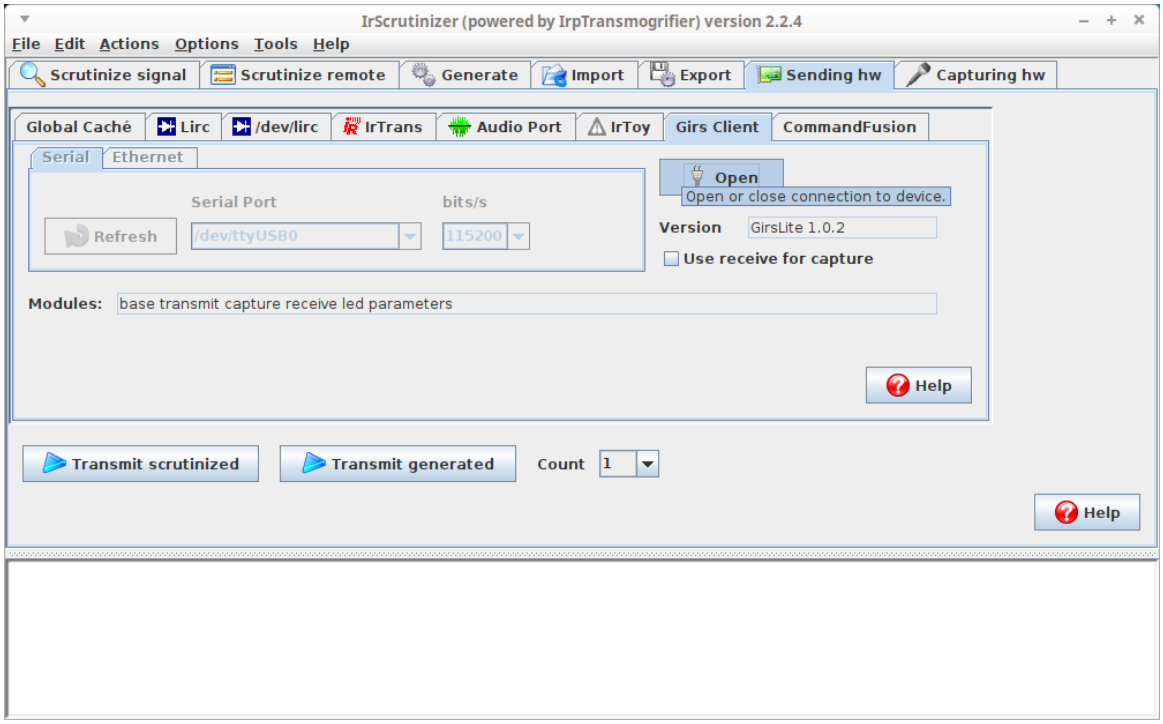
Instructions for a simple upload are not available for MacOS. Please use the standard Arduino IDE as described here.

2.2.5.3 Linux

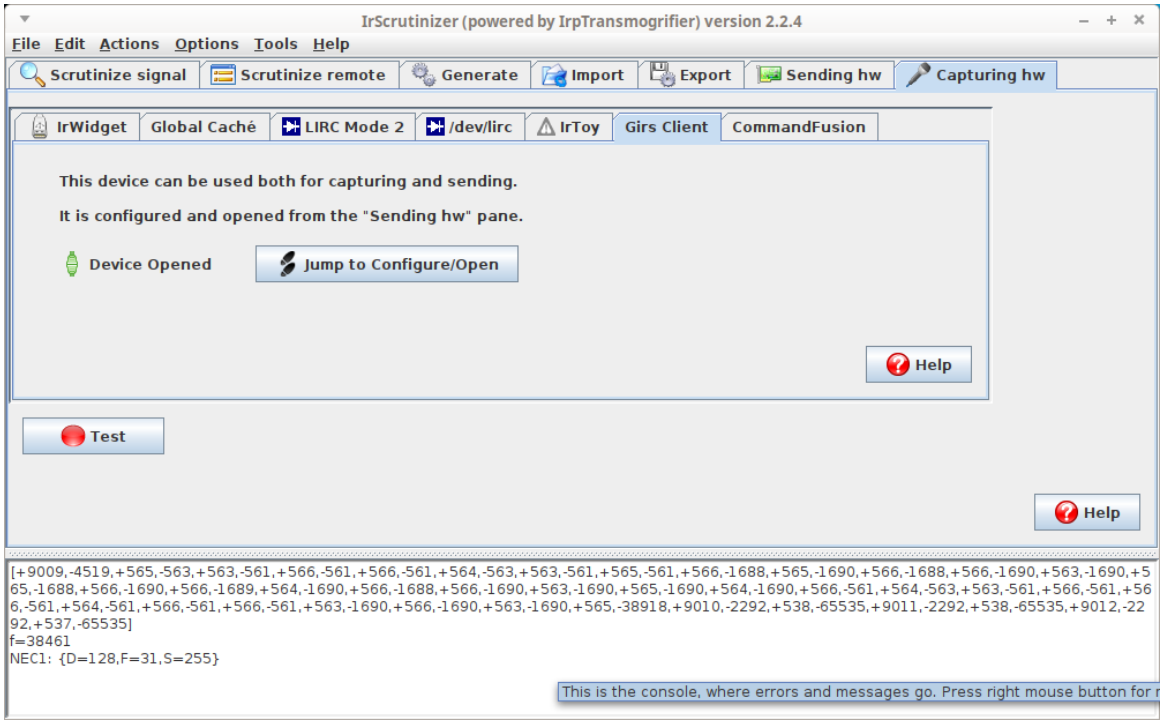
Depending on the Linux distribution used, avrdude can be installed with a command like `sudo dnf install avrdude` (RPM based systems like Redhat including Fedora) or `sudo apt-get install avrdude` (Debian derived systems). Then, connect the board, and make sure that gets the device `/dev/ttyUSB0`. It can now be flashed by the script GirsLite-1.0.2-nano-flasher.sh. This is run without arguments, as it contains the firmware to be flashed. Executing with `sudo` may be necessary, if the system is not configured correctly.

2.2.6 Use in IrScrutinizer

First open the device for sending: Select the `Sending hw -> Girs client` pane. Select "Serial" and then the appropriate serial port. If the board has been connected after the program was started, it may be necessary to press the "Refresh" button to see the port in the pull-down menu. Press "Open". (This probably causes a reset of the board, causing all the LEDs to go on. If this does not happen, it is not a problem either.) The version of the firmware should now be reported in the field to the right of the "Open" button. The red LED should come on to signal that the board is waiting for a command. The checkbox "Use receive for capture" makes IrScrutinizer use the demodulating sensor for captures. This leads normally to inferior results, and should be used only if, e.g., the non-demodulating sensor is missing.



For receiving, select the board by selecting "Capturing hw. -> Girs Client. Then press the "Test" button, (the red LED should now go out and the yellow go on) and, using any IR remote, fire a command at the (non-demodulating) sensor. The measured frequency and the recorded timings should now be printed to the console. Hopefully, there is also a decode.



For more verbose logging of the communication between the program and the Arduino, select `Options -> verbose`. (Please do so before searching help.)

2.2.7 Use in Lirc

Since some time (since release version 0.9.4), the Girs driver is officially included in the Lirc sources. The linked documentation describes in detail how to use the board with Lirc.

2.2.8 ...and now?

The easy part is now finished. There is a second part for nerds...

2.2.9 Appendix A. Bill of material

Quantity	Mfg.#	Manufacturer	Description>
1			Arduino Nano 3.x (clone) with mini USB, bare holes
1	SFH 4546	Osram	Infrared Emitters - High Power Infrared 940nm
1	SFH 4544	Osram	Infrared Emitters - High Power Infrared 940nm

Quantity	Mfg.#	Manufacturer	Description>
1	TSMP58000	Vishay	Infrared Receivers IR Receiver Module
1	TSOP34438	Vishay	Infrared Receivers IR Sensor IC 38kHz
1	WP710A10ID5V	Kingbright	Standard LEDs - Through Hole Red 25mcd 625nm 40 deg 5V resistor
1	WP710A10YD5V	Kingbright	Standard LEDs - Through Hole Yellow 15mcd 588nm 40 deg 5V resistor
1	WP710A10SGD5V	Kingbright	Standard LEDs - Through Hole Green 25mcd 568nm 40 deg 5V resistor
1			Resistor - Through Hole 68ohm, 1/4W

Here is a basket at Mouser, not containing the Arduino Nano board.

2.2.10 Appendix B. Accessing devices in Linux

Linux, being a multi user system by design, protects the devices from access by the "non-authorized". In order to access devices like `/dev/ttyUSB0` without being root, the recommended procedure is to include the current user in the appropriate group, in general `dialout` (or the older name `uucp`) and/or `lock`. How this is done differs between the different Linux distributions, but is typically

```
sudo usermod -aG dialout,lock user
```

(with current user name substituted for *`user`* .

For IrScrutinizer, see also this.

2.3 Arduino Nano as IR sender and receiver, part 2: details

Date	Description
2016-01-08	Initial version.
2016-01-11	Added section for symlink-broken systems.

Table 1: Revision history

2.3.1 Introduction

This document is a supplement to the first part. It contains some additional information, left out of the first part for the ease of accessibility.

A third part is planned, elaborating on further possibilities of AGirs, and showing some hardware solutions, in particular on more capable boards than the ATMega328 of Ardiuino Nano/Uno.

2.3.2 The AGirs Software

Girs is a specification for a modular command language for command language of a server, capable of sending and receiving IR commands, and more. "Modular" means that an implementation can select whether to implement different "subsets" (called modules) or not. AGirs is a project that implements a subset of the Girs specification on the Arduino platform. It is highly configurable using C pre-processor symbols and a configuration file. *GirsLite* is a special configuration of AGirs, aimed at IrScrutinizer and Lirc. Previously, there existed another configuration called *Girs4Lirc*, aimed specifically at Lirc, but it has now been merged into GirsLite.

The AGirs package allows a lot of other nice things, like sending and decoding some IR protocols directly (presently only NEC1 and RC5), support for LCD display and a number of signaling LEDs, named commands (a data base (like Lirc) of commands identified by names like `Play`), support for sending RF commands, Ethernet (TCP and UDP sockets) support (instead of using the serial port). See the documentation in the project.

AGirs is not a finished project. Cooperation is welcome.

2.3.3 Alternative Components

There is nothing magic about the selection I have made in part 1. Many different alternatives exist, which will be discussed next.

The Nano board and the ATMega328 are hardly "hot" any longer. However, its performance and memory configuration is more than enough for simple applications, like in IrScrutinizer and Lirc. If considering "hotter" alternatives, note that ATMega328 boards can supply (up to) 40mA on the GPIO pins, more than enough to shoot an IR signal with one or two high-performance LEDs at quite a distance. The modern 32bit processors typically are specified for 7mA, making a driver transistor, (like 2N7000/BS170) necessary.

The main reason for the selection of the Osram IR LEDs is that the black packaging looks cool :-). Many other alternatives, for example the Vishay TSAL6100 (narrow beam) and the TSAL6200 (wide beam) exist. However, be sure to use IR LEDs with a wavelength of 940-950nm, corresponding to Consumer IR, not a 890nm component, intended for use

with IRDA. An IR LED of the latter type will almost surely "work", but with non-optimal performance.

Of course, using only one LED is also an option. Be sure to check the resistor value, see the Appendix.

(Some) alternatives to the TSMP58000 are TSMP58138, TSMP1138, and TSMP4138 (the latter is found to the right on the IrScrutinizer splash screen). They all have similar properties, very good amplification, but ends at around 60kHz modulation frequency. The smaller sensors, QSE157 (used in the IrWidget, now end-of-life and replaced by OPL551), QSE159 (used in IrToy), Honeywell SDP8610 can also be used, but due to its smaller effective sensor area, makes the sensitive range smaller. Also, the "small" ones in general do not run on 3.3 Volts.

> **Note:**
> Note that the different chips are in general not pin compatible. Bu sure to check the data sheet.

All of the sensors mentioned have inverting output, which is what the firmware expects. There are also sensor with non-inverting output available. These are not directly supported.

The experiments I have made with photo diodes (potentially being able to handle much higher modulation frequencies) show that the range reduces to millimeters.

Alternative IR Receivers: First, note that the last two digits of the number indicates the modulation frequency. For a receiver, this is to be understood as the center frequency of a band-pass filter. An IR receiver marked "38kHz" will work quite well for modulation frequencies between 36 and 40kHz (at least). (See e.g. Figure 5 in the data sheet for TSOP34438). As far as I know, all "universal" IR hardware, IrToy, Iguana, TIRA, USB/UIT, CommandIR,... come with 38kHz receivers. However, if searching the optimal solution for a certain remote or protocol etc, getting the optimal receiver tuned to the actual modulation frequency is definitely not a bad idea.

2.3.4 Testing the firmware

When the sketch is started, as a self test, the (visible light-) LEDs, and the "pin13"-LED, are turned on for two second. This is the first test, requiring no additional hard- or software.

When running the GirsLite sketch, the board is nothing but a server in the sense of the Girs Command Language. It can be accesses by programs like IrScrutinizer and Lirc (using its Girs driver), but also by humans using a terminal program, like the serial monitor of the Arduino IDE. For this, set the baud rate to 115200, and the line ending to carriage return. It is now possible to communicate with the unit using the terminal program. Just type the command to the program, and the unit will respond. One line in, one (possibly long) line out. GirsLite supports the commands `modules`, `version`, `transmit`, `analyze`, `receive`, and `led`. All commands can be abbreviated to

the first letter. For example, to test the `receive` command, just type "r" (without the quotes), followed by return, and fire a suitable IR signal at the receiver. The raw capture will be output to the terminal program. Using the clipboard, it can be pasted to IrScrutinizer, and analyzed. Also the other commands can be tested in this way.

2.3.5 Compiling the sources

First install the latest Arduino IDE (at the time of this writing, this is 1.6.7) from arduino.cc. (Do not use "1.7.x" from `arduino.org`.)

Some proficiency using the Arduino IDE is required. This is covered at the Arduino site.

Download and unpack the AGirs software. Copy, move, or link `src/GirsLib` therein to your Arduino libraries folder. Then download and unpack Infrared4Arduino repository into your Arduino libraries folder.

As is common in the Open Source/Git community, this project contains symbolic links to organize the sources and prevent duplication. This usage is not discouraged in any document I have seen so far, neither in the Git handbook nor in the Github texts. However, on Windows this can cause problems, at least for some Git implementations. Long story short: if `src\GirsLite\GirsLite.cpp` just contains the line `../Girs/Girs.cpp`, then delete the file and replace it by the file `src\Girs\Girs.cpp`.

Now connect the board and start the Arduino IDE on the file `src/GirsLite/GirsLite.ino`. In `Tools -> Boards` select `Arduino Nano`. In `Tools -> Processor` select `ATmega328`. In `Tools -> Port` select the port the board is currently connected to (`COMN` (for some N) on Windows, `/dev/ttyUSB0` or similar with Linux,...). Pressing the "upload" button will now (hopefully!) compile the program and upload ("flash") it onto the board.

2.3.6 Appendix. Computing the current through the IR LEDs

Given n IR LEDs connected in series between ground and a resistor R to the voltage V. Compute the current I through the components! For this, first *guess* the answer I (say 40mA, which is the maximal current from the AtMega328 GPIO pins). Then use the data sheets for the IR LEDs to determine the forward voltage over the i-th IR LEDs, V_i (typically 1.4V). So the voltage over R is thus $V - (V_1 + ... + V_n)$, and, by Ohm's law $I = (V - (V_1 + ... + V_n))/R$. Now go back and check that the original guess was reasonable, and repeat it necessary. For V, this is clearly "somewhat" lower than Vcc, see Figure 29-163 in the ATMega328 datasheet. 4.5V might be a reasonable value.

From a practical point of view, using the standard USB as power, it seems reasonable to simplify the formulas as $I = (4.5 - n\ 1.4)/R$.

2.3.7 Appendix. Configuration of udev for Linux

Linux (the subsystem `udev` really) assigns a connected Arduino a device file name of the form `/dev/ttyACMn`, in some cases `/dev/ttyUSBn`, where `n` is the smallest non-negative integer not yet taken. This can cause unpredictable behavior, not only when using several Arduinos, but also in context of other device using the same names, like IrToys. By using custom rules to udev this difficulty can in general be circumvented.

Since there are so many different manufacturers of "Arduino-compatible" hardware, "all" having different vendor id and product id, there is no single simple answer. Some comes with a unique serial, cheaper clones in general not. As a guide to the reader, not as a definite answer, this is my `/etc/udev/rules.d/10-arduino.rules`.

```
SUBSYSTEMS=="usb", ATTRS{idProduct}=="0043", ATTRS{idVendor}=="2341", SYMLINK+="arduino
 arduino_uno arduino_uno_$attr{serial}"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="7523", ATTRS{idVendor}=="1a86", SYMLINK+="arduino
 arduino_nano_qinheng"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="2303", ATTRS{idVendor}=="067b", SYMLINK+="arduino
 arduino_nano_prolific"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="6001", ATTRS{idVendor}=="0403", SYMLINK+="arduino
 arduino_nano_ftdi arduino_nano_ftdi_$attr{serial}"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="8037", ATTRS{idVendor}=="2341", SYMLINK+="arduino
 arduino_micro"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="8036", ATTRS{idVendor}=="2341", SYMLINK+="arduino
 arduino_leonardo"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="0042", ATTRS{idVendor}=="2341", SYMLINK+="arduino
 arduino_mega arduino_mega_$attr{serial}"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="804f", ATTRS{idVendor}=="2a03", SYMLINK
+="arduino.org arduino_m0_pro"
SUBSYSTEMS=="usb", ATTRS{idProduct}=="2111", ATTRS{idVendor}=="03eb", SYMLINK
+="arduino_m0_pro_prog"
```

Note that these are not "real" names but symbolical links, like extra, alternative names. They are not really usable for the present version of IrScrutinizer. For Lirc, it can be useful however.

2.3.8 References

1. Reference documentation for the Arduino Nano board. For clones (which are in general based on the open-sourced Eagle files), this is also in general quite reliable, the main difference is that the clones do not use a FTDI chip but a CH340 chip.
2. Schematics for the Arduino Nano board. Same remark as above.
3. Data sheet for ATmega328 (and some others).
4. IR LEDs Osram SFH4544, IR LED with narrow beam (half angle ± 10°), data sheet.
5. IR LEDs Osram SFH4546, IR LED with wide beam angle (half angle ± 20°), data sheet.
6. Non-demodulating IR sensor TSMP8000, data sheet.
7. Demodulating IR Receiver TSOP34438, data sheet.

2.4 IrScrutinizer documentation

Note:

This is a reference manual. It is written for completeness and correctness, not for accessibility. For an easier introduction, see this tutorial. Or, just try the program, and come back to the manual if and when you need it.

Warning:

Sending undocumented IR commands to your equipment may damage or even destroy it. By using this program, you agree to take the responsibility for possible damages yourself, and not to hold the author responsible.

Date	Description
2013-11-12	Initial version.
2013-12-01	Next unfinished version.
2014-01-22	Version for release 1.0.0, still leaving much to be desired.
2014-06-07	Version for release 1.1.0, some fixes, much still to be done.
2014-09-21	Version for release 1.1.1, some more fixes and enhancements.
2015-04-07	Version for release 1.1.2, installation issued reworked. Misc. minor improvements.
2015-08-19	Version for release 1.1.3. Misc. minor improvements.
2016-01-13	Added description for user selectable character sets for import and export.
2016-04-29	Version for release 1.2. Misc. minor improvements.
2016-08-30	Version for release 1.3. Misc. minor improvements.
2017-03-12	Version for release 1.4. Misc. minor improvements.
2019-08-09	Version for release 2.0.0. Not complete...
2020-05-03	Updated installation instructions.
2020-05-20	Minor updates for version 2.2.6.
2020-07-30	Updated building instructions.
2022-05-13	Updated for 2.4.0: Considerable update and rewrite, for upcoming release 2.4.0.
2022-12-19	Misc. fixes and updates.

Date	Description
2023-12-17	Accessing HarcHardware command line is now supported in AppImages.
2023-12-26	Removed leftover references to the "minimal json" as well as Fedore dnf tonto.
2024-01-05	Removed GlobalCache IR database.
2024-03-12	Added link to Lirc imports with and without timing info. Updated command line parameters. Misc fixes and updates.

Table 1: Revision history

2.4.1 Introduction

IrScrutinizer is a powerful program for capturing, rendering, analyzing, importing, and exporting of infrared (IR) signals. For capturing and sending IR signals several different hardware sensors and senders are supported. IR Signals can be imported not only by capturing from one of the supported hardware sensors (among others: IrWidget, Global Caché, Command Fusion, and Arduino), but also from a number of different file formats (among others: Lirc, Wave, CML, Pronto Classic and -professional, and different text based formats; not only from files, but also from the clipboard, from URLs, and from file hierarchies), as well as the Internet IR Databases by Global Caché and by IRDB. Imported signals can be decoded, analyzed, edited, and visualized. A collection of IR signal can thus be assembled and edited, and finally exported in one of the many supported formats. In addition, the program contains the powerful IrpTransmogrifier IR-renderer, which means that almost all IR protocols known to the Internet community can be generated.

Written in Java (with the exception of two native libraries), most of the functionality of the program is available on every Java platform. The native library (NRJavaSerial) is presently available for 32- and 64-bit versions of Windows, Linux (x86, amd-64, arm version 7), and MacOsX, and can with moderate effort be compiled for other platforms. (The library DevSlashLirc is available (and meaningful!) for Linux only.)

For someone with knowledge in the problem domain of IR signals and their parametrization, this program is believed to be simple to use. This knowledge is assumed from the reader. Other can acquire that knowledge either from the JP1 Wiki or, e.g., this link.

Note that screen shots are included as illustrations only; they may not depict the current program completely accurately. They come from different versions of the program, using different platforms (Linux and Windows), and using different "look and feels".

Sources are hosted on Github. Bug reports and enhancement requests are welcome (e.g. as discussions, issues), or as contributions (code, testing, documentation, use cases, protocols etc.).

The present document is written more for completeness than for easy accessibility. For an easier introduction, see this tutorial.

Here are the current release notes.

2.4.1.1 Background

In 2011 I wrote an IR signal "engine" called IrpMaster. It can also be invoked as a command line program. Then a program called IrMaster was released, which among other things constitutes a user friendly GUI front end to IrpMaster. The present program, IrScrutinizer, was also based on IrpMaster, and adds functionality from IrMaster, in particular the possibility to collect IR signals, a vastly improved import and export facility, and edit collections of IR commands. IrScrutizer almost completely replaces IrMaster. The final version of the latter was released in February 2014, slightly ironically called version 1.0.0. No further development is planned.

The final version of IrScrutinizer using IrpMaster and the decoding engine DecodeIR is called 1.4.3, and is available here. Since version 2.0.0, the newer IR engine IrpTransmogrifier has replaced IrpMaster, DecodeIR and Analyzer.

2.4.1.2 Copyright and License

The program, as well as this document, is copyright by myself. My copyright does not extend to the embedded "components", like Jirc. IrpTransmogrifier is using ANTLR4 and depends on the run time functions of ANTLR4, which is free software with BSD license.

The "database file" `IrpProtocols.xml` is derived from DecodeIR.html, as well as many, many discussions with, and contributions from, members of the JP1 community, thus I do not claim copyright.

The program uses JCommander by Cédric Beust to parse the command line arguments. It is free software with Apache 2 license.

Icons by Everaldo Coelho from the Crystal project are used; these are released under the LGPL license.

The Windows installer was built with Inno Setup, which is free software by Jordan Russel. To modify the user's path in Windows, the Inno extension `modpath` by Jared Breland, distributed under the GNU Lesser General Public License (LGPL), version 3.

Serial communication is handled by the Neuron Robotics NRSerialPort library licensed under the LGPL v 2.1 license. (This is a fork of the discontinued RXTX library.)

Lirc (Linux Infrared Remote Control) is according to its web site copyright 1999 by Karsten Scheibler and Christoph Bartelmus (with contribution of may others), and is licensed under GPL2. The parts used here have been translated to Java by myself, available with the name Jirc.

For Pronto Classic support, Tonto by Stewart Allen was used. It is licensed under the "Clarified Artistic License".

The program contains icons from Dangerous Prototypes and IrTrans. These are used exclusively in the context of these firms, and only used to illustrate their products.
The icons for JP1 and Lirc are also exclusively used to illustrate the community, their products, and files.

The program and its documentation are licensed under the GNU General Public License version 3, making everyone free to use, study, improve, etc., under certain conditions.

File formats, their description in human- or machine-readable form (DTDs, XML Schemas), are placed in the public domain.

2.4.1.3 Privacy note

Some functions (Help -> Project Home page, Help -> IRP Notation Spec, Help -> Protocol Specs, Tools -> Check for updates) access the Internet using standard http calls. This causes the originating machine's IP-address, time and date, the used browser, and possibly other information to be stored on the called server. If this is a concern for you, please do not use this (non-critical) functionality (or block your computer's Internet access).

2.4.2 Overview

Next a high-level view of the different use cases will be given.

Analyze ("Scrutinize") individual IR Signal/Ir Sequence
An IrSignal or IrSequence can be captured from connected hardware, or imported from files in different formats, the clipboard or from Internet databases. The IrSequence can be broken into a beginning-, repeat- and ending sequence, and decoded, analyzed, and plotted. It can be exported in different formats, or sent to different transmitting hardware.

Analyze/edit/compose/export collections of IR signals (""remotes"**)**
A collection of commands can be assembled either from individual IR signals (as above), captured several at a time, or imported from files in different formats, the clipboard, or from Internet databases. The collection and the individual commands can be edited as in a spreadsheet. It can be exported in a number of different formats.

Generate (render) IR Signals from known protocols
IR Signals can be generated from the Internet's largest protocol data base, containing over 100 protocol. Necessary protocol parameter values are to be entered. Thus generated signals can be analyzed as single signals, incorporated into remotes, or exported to files — also from e.g. intervals of parameters.

2.4.3 Installation

2.4.3.1 Download

The latest official release is always available here.

In addition, there are continuous integration builds, at least most of the time. These are build from a snapshot of the current sources. Because of this, they are more likely to be unstable, contain bugs, or even to be outright useless.

2.4.3.2 General

IrScrutinizer, and all but two of its third-party additions, are written in Java, which means that it should run on almost all current computer: Windows, Linux (including Raspberry Pi), Macintosh, etc. Many of the installation options come with its own contained Java, for the other options, the Java runtime (presently version 8 or later) must be installed. The exception is the native part of `NRJavaSerial`, which are written in C , and invoked as native, shared library (`.dll` in Windows, `.so` in Linux, `.jnilib` in Mac OS X). For Linux, there is also the `DevSlashLirc` library, for accessing Lirc compatible hardware. If the shared libraries are not available on for a particular platform it is not a major problem, as most parts of IrScrutinizer will work fine without it; just the serial hardware access, or the access to certain hardware will be unavailable.

In all versions, IrScrutinizer behave civilized, in that the installation directory is not written to, and thus can be read-only as soon as the installation has taken place.

To use all functionality of the program, it may be necessary to explicitly white-list some ports in a firewall. These are listed here.

There are five different way of installing the program, described next.

2.4.3.3 Windows

Download the Window setup file and double click it. Select any installation directory you like; suggested is `C:\Program Files\IrScrutinizer`. Select creation of the start menu folder and the desktop icon (unless you prefer it otherwise). If administrator rights are present, the recommended installation location is in a subdirectory of `Program Files`. Otherwise the program can be installed in any directory writable by currently available user rights. The setup comes with a Java runtime, but it is also possible to use a compatible, existing installation. IrScrutinizer can now be started from `Start -> IrScrutinizer -> IrScrutinizer`, or from the desktop icon.

Sometimes Windows will refuse to install programs from sources that it does not consider as trustworthy. It may then be necessary on Settings -> Apps & features to select on "Choose where to get apps" to something else than "The Microsoft store only".

If desired, the installer will associate `.girr` files with the program, allowing them to be opened by double clicking them.

The installer will also install the command line program IrpTransmogrifier, which can be called as `irptransmogrifier` from a command line window. This also goes for HarcHardware.

Sometimes, when starting the program for the first time, the Windows Defender Firewall blocks some features in the program, asking for permission to allow the Java virtual machine (typically ending with `javaw.exe`) access. This should be granted.

To uninstall, select the uninstall option from the Start menu, or (on Windows 10, right click `Start -> IrScrutinizer -> IrScrutinizer` and select Uninstall). Or use the operating system's program management for de-installing (Settings -> Apps and Features). Very pedantic people may like to delete the properties file too, see properties.

2.4.3.4 AppImage for 64-bit Linux

An AppImage is a distribution consisting of one large monolithic executable file that is downloaded, made executable, and run, without any installation in the classical sense. At this moment, only 64-bit x86 Linux is supported. It is believed to run on all modern, 64-bit x86 Linux distributions. Just download, make executable, and double click (or start from the command line). It comes with its own Java runtime.

It called using the name `irptransmogrifier`, the IrpTransmogrifier program will be invoked, taking its usual command line arguments. Similarly, if called using the name `harchardware`, the HarcHardware main routine will be invoked.

To uninstall, just delete.

2.4.3.5 MacOS app

Download the disk image file (`*.dmg`). Mount it by double clicking. A mounted disk image `IrScrutinizer` will then appear on the desktop. Opening it will show an app, and a few documentation files. The app can just be dragged to the desktop, to the tray, to `Applications` or any other location the user prefers. IrScrutinizer now behaves like any other app in Mac OS X.

Sometimes the operating system will refuse to install or run programs from sources that it does not consider as trustworthy. It may then be necessary on System Preferences -> Security & privacy -> General to select on "Allow apps downloaded from" to "Anywhere".

To uninstall, just drag the program to the trash can.

The command line programs IrpTransmogrifier and HarcHardware are not supported in this mode. (For this, the generic binary distribution has to be used.)

2.4.3.6 Generic Binary

The generic binary version consists of all the Java classes packed in one executable jar file, together with supporting files, like all the compiled shared libraries for the different operating systems Linux (x64-32, x86-64, armv7), Windows (x86, 32 and 64 bits), and Macintosh. It can be used on all those systems. (In other environments, the shared libraries may be compiled with moderate effort.)

The generic binary distribution also can be used whenever using the setup.exe/app installation is not possible, not up-to-date, or just not desired. It can also be used to update an existing installation on, e.g., Windows.

First make sure that the Java runtime of correct version (currently version 8 or later) is installed.

To install, unpack in an empty directory of your choice; suggested (for Unix/Linux) is `/usr/local/share/irscrutinizer`. On Windows, the program can be started by double clicking the .jar file.

The rest of this section applies to Unix/Linux.

There are a few additional steps, which, on Unix/Linux the script `setup-irscrutinizer.sh` can take care of. If this is not desired, or does not work, these steps will be described next.

Inspect the wrapper `irscrutinizer.sh`, and make changes if necessary.

It is recommended to make links from a directory in the path to the wrapper script, e.g.

```
                    ln -s /usr/local/share/irscrutinizer/irscrutinizer.sh /usr/local/
bin/irscrutinizer
                    ln -s /usr/local/share/irscrutinizer/irscrutinizer.sh /usr/local/
bin/irptransmogrifier
                    ln -s /usr/local/share/irscrutinizer/irscrutinizer.sh /usr/local/
bin/harchardware
```

The desktop file `irscrutinizer.desktop` should, if desired, be installed in `/usr/local/share/applications` alternatively `~/.local/share/applications`.

The program can now be started either as a desktop program, or by typing `irscrutinizer` on the command line. Also the command line programs IrpTransmogrifier and HarcHardware can be started by the command `irptransmogrifier` and `harchardware` respectively. It is also possible to start IrScrutinizer by double clicking on the jar file. In case this brings up the archive manager, the desktop needs to be taught to open executable jar files with the "java" application (i.e. the JVM). For this, select a jar file the file browser, select the properties, and select "java" as the application to "open with". (The details might vary.)

On Linux, it is general necessary to check, and possibly fix, the access to the serial devices. The previously mentioned wrapper does this.

On Gnome-like desktops, by copying the file `girr.xml` to the system's data base of known file types (normally located in `/usr/local/share/applications`), `*.girr` files can be opened in IrScrutinizer by double click.

To uninstall, just delete the files. Some may like to delete the properties file too.

2.4.3.7 Source distribution

Compiling the sources is covered in the Appendix. This generates the generic binary distribution file, which is installed as described above. Alternatively, it can be installed with the command

```
make install
```

possibly with `sudo` or as root.

2.4.3.8 Linux/Unix Serial device access

To access serial devices, including "serial" devices connected over USB, most current Linuxes require the current user to be a member of the group `dialout` (some distributions use the legacy name `uucp`), in some cases (e.g. Fedora) also the group `lock`. To grant user `$USER` this membership, typically a command like `sudo usermod -aG dialout $USER`, and/or `sudo usermod -aG lock $USER` is used.

It has been brought to my attention that Arch Linux (and likely others) still use the older name `uucp` for `dialout`.

Do not run this program as root (except possibly for debugging device access).

2.4.4 Concepts

For anyone familiar with the problem domain, this program is believed to be intuitive and easy to use. Almost all user interface elements have tool-help texts. Different panes have their own pop-up help. In what follows, we will not attempt to explain every detail of the user interface, but rather concentrate on the concepts. Furthermore, it is possible that new elements and functionality has been implemented since the documentation was written.

This program does not disturb the user with a number of annoying, often modal, pop ups, but directs errors, warnings, and status outputs to the *console window*, taking up the lower third of the main window. This window is re-sizeable. There is a context menu for the console, accessible by pressing the right mouse button in it.

In the upper row, there are six pull-down menus, named `File, Edit, Actions, Options, Tools` and `Help`. Their usage is believed to be mainly self explanatory, with some the exceptions.

Options to the program are in general found in the `Options` menu, or its subordinate menus. Some parameters for particular export formats are found in the sub-panes of the `Export` pane. Also the hardware configuring panes contain user parameters.

Options are in general saved persistently between sessions, in the properties file.

The main window is composed of six sub panes denoted by `Scrutinize signal` (for processing a single signal), `Scrutinize remote` (for collecting several signals

to one "remote"), `Render` (generates (renders) an IR signal from protocol name and parameters), `Import`, `Export` and `Hardware`. respectively. These panels will be discussed in Section GUI Elements walk through

2.4.5 GUI Elements walk through

2.4.5.1 The "Scrutinize signal" pane

This panel is devoted to the analysis of a *single IR signal* or IR sequence. A (single) signal is either read from hardware using the "Capture" button (requires that the capturing hardware has been set up, see Section Hardware, imported from a file (using the context menu in the data window, or through `File -> Import -> Import as single sequence`), or pasted from the clipboard. Also, some other panes can transfer data to this pane.

The button `Capt. (cont.)` a program tread is started, that continuously captures signals from the hardware. Only the last signal is kept.

As shown, an IR signal consists of an (sometimes empty) start sequence (red), a (sometimes empty) repeat sequences (blue), and sometimes an ending sequence (green).

For text import, the signal can be in either Pronto Hex format or in raw format (indicated by a leading "+"-sign). The signal is printed in the data window, in the preferred text format, which can be selected from the options menu. The text representation may be edited (assuming sufficient knowledge!), after which the edited signal is analyzed and plotted again by pressing the `Scrutinize` button. The signal may be sent to the sending hardware by pressing the `Transmit` button.

Sometimes the "scrutinization" of a signal changes the numbers in an unexpected way. If desired, the previous content of the data window is recalled by `Actions -> Undo Scrutinizer data`.

The plot can be horizontally zoomed by pressing the left mouse button and dragging. Printing and exported as graph are presently not implemented.

The menu entry `Actions -> Enter test signal` (or its accelerator, the F9 key) enters a test signal.

Using context menus, the result can be sent to the clipboard or saved to a file. The menu entry `clone plot` makes a clone of the plot. This can be used for visually comparing different sequences or signals.

Note that transforming the signal between different formats may introduce rounding errors. In rare cases, this may causing decoding to fail.

A Girr file can be dropped on the text- or plot windows. This will display the concatenation of the signals in the file.

2.4.5.2 The "Scrutinize remote" pane

This panel is devoted to the capturing/import/editing of a collection of IR signals, called *a remote* in the sequel. The panel contains two sub-panels: for parametric signals and for non-parametric, "raw", signals.

A *parametric signal* is determined by its protocol name and the values of the protocol's parameters. A *raw signal* is determined by its timing pattern, and its modulation frequency. It may have one or many decodes, or none. Nevertheless, by definition, a raw signal is determined by its timing, not the decodes.

In both cases, the sub panes consists of tables with a number of columns. Every signal takes up a row in the table. The content of the individual cells (with the exception of its number and date) can be individually edited, like in a spreadsheet program.

By clicking and dragging in the table head, the columns can be rearranged with the mouse.

In both tables, the right mouse button opens a context menu containing a number of ways to manipulate the table, its view, or the data contained therein. By enabling the row selector, the rows can be sorted along any of the present columns.

Clicking a row with the middle mouse button selects that row, and (ir possible) transmits it using the currently selected hardware.

To capture IR signals, first configure the hardware using the hardware pane. Next press the `Capture` button. The program will now run the capturing in a separate thread, so the user just have to press the buttons of the remote. The signals will be received, interpreted, decoded, and entered on subsequent lines in the selected table (raw or parameterized). The capture thread will continue until the captured button is pressed again. (Note that this is completely different from the capture button on the `Scrutinize signal` panel.) The user may mix captures with other activities, like entering information (name, comments,...) in the table.

The export button exports the content of the currently selected table (raw or parameterized) according to the currently selected export format.

A Girr file can be dropped on the parameter as well as the raw table, which will import the contained signals.

The menu entry `Actions -> Enter test signal` (or its accelerator, the F9 key) enters a test signal, either as parametric signal, or as a raw signal.

The parametric table columns

#
A unique number assigned to the signal when creating. It is not editable and not exported.

Timestamp
A timestamp generated when the signal was entered or created. It is not editable and not exported.

Protocol
Name of the protocol. In the table, anything can be entered, however, to transmit or export, it has to match (case insensitively!) a known protocol.
D
The `D` parameter value, as defined in the protocol IRP.
S
The `S` parameter value, as defined in the protocol IRP.
F
The `F` parameter value, as defined in the protocol IRP.
T
The `T` parameter value, as defined in the protocol IRP.
Misc. params
All parameters not called `D`, `S`, `F`, or `T`, using syntax *name* = *value*, for example `X=42 Y=73`. (Note that there are no spaces around the equal ("=") sign, but between the assignments.
Name
Name of the command. This is in principle arbitrary, however, to be able to constitute a meaningful export to a target device, the names have to be unique within the table.
Comment
A textual comment. In most export formats, it is transmitted to the export, and the export format can treat it anyway it desires.

The raw table columns

#
A unique number assigned to the signal when creating. It is not editable and not exported.
Timestamp
A timestamp generated when the signal was entered or created. It is not editable and not exported.
Intro
The intro sequence of the command.
Repeat
The repeat sequence of the command.
Ending
The ending sequence of the command.
Name
Name of the command. This is in principle arbitrary, however, to be able to constitute a meaningful export to a target device, the names have to be unique within the table.
Decode
The result of feeding the signal to the decoder. Not editable, not considered as authoritative information; serves only for information.
Analyze

The result of feeding the signal to the analyzer. Not editable, not considered as authoritative information; serves only for information.

Comment

A textual comment. In most export formats, it is transmitted to the export, and the target can treat it anyway it desires.

Frequency

Modulation frequency in Hz.

2.4.5.3 The "Render" pane

In the upper part of this pane, an IR protocol is selected, identified by name, and the parameters D ("device", in almost all protocols), S ("sub-device", not in all protocols), F ("function", also called command number or OBC, present in almost all protocols), as well as T, "toggle" (in general 0 or 1, only in a few protocols). These number can be entered as decimal numbers, or, by prepending "0x", as hexadecimal numbers. Numbers with a leading "0" are interpreted as octal numbers.

Earlier versions of this program (including documentation) used the word *generate* instead of *render*.

By pressing `Render`, the signal is computed, and the middle window is filled with a textual representation in the form selected by `Options -> Output Text Format`.

For protocols with a toggle, leaving it unassigned (`T` left empty) makes the rendering engine "toggle" in the sense that it changes its value on every invocation, in accordance with the IRP. The rendering engine is invoked not only by `Render`, but also by `Transmit`, `Export`, and the `To ...` buttons.

The Export button initiates an export to a file format currently selected on the Export pane. The three lower buttons transfer the signal(s) to the scrutinize signal panel, the raw remote table, or the parameterized panel.

Accessing a number of different parameter values

For the export and the transfer to the Scrutinize remote tables, not only a single parameter value can be selected, but whole sets:

1. Of course, there is the singleton set, just consisting of one value
2. There is also a possibility to give some arbitrary values, separated by commas. Actually, the commas even separate sets, in the sense of the current paragraph.
3. An interval, optionally with a stride different from 1, can be given, either as `min..max++increment` or `min:max++increment`, or alternatively, simply as *, which will get the min and max values from the parameter's parameter specs.
4. Also, a set can be given as `a:b<<c`, which has the following semantics: starting with `a`, this is shifted to the left by `c` bits, until `b` has been exceeded (reminding of the left-shift operator `<<` found in languages such as C).
5. Finally, `a:b#c` generates c pseudo random numbers between a and b (inclusive). The "pseudo random" numbers are completely deterministically determined from the seed.

The parameters `a` and `b>` are optional. If left out, the values are taken as from the protocol parameters min and max respectively, just as with the `*` form.

2.4.5.4 The Import pane

The import pane allows for selective import of collection of IR commands. Both Internet data bases and file formats are supported. Import can take place from local files or even file hierarchies, from the clipboard or from Internet URLs.

Tree importer

the format of an expandable tree. By placing the mouse cursor over a command, additional information, like decode, is presented. A single command can be selected by mouse click, a sequence of adjacent commands by shift-click, a subset of not necessarily adjacent commands be selected by Ctrl-click, as usual from most GUIs. A single selected command can be transferred to the "Scrutinize signal" pane by pressing `Import signal`. The `Import all` (`Import selection`) button transfers all commands (the selected commands) to the `Scrutinize remote` pane, sub-pane `Parametric remote` (without overwriting already present commands), while the buttons `Import all/raw` and `Import selected/raw` transfer to the sub-pane `Raw remote`.

The key `Transmit selected` transmits the (single) selected signal to the selected sending hardware.

Data bases

Global Caché's Control Tower data base

This is the new data base from Global Caché. Unfortunately, its Terms of Service are fairly restrictive. Also, non-premium users are not allowed to download codes using the API. For this reason, the program really only "browses" the data base.

To use its codes, log in with the browser (the `Web site` button goes to the home page), and have the code set mailed in CSV format (press `Send Code Set`). The mail received can be imported by the import text pane, raw subpane, Name col. = 1, Raw signal col = 2 (or 3), Field separator: comma.

Remotelocator

RemoteLocator is a program to locate and download infrared remotes. The `RemoteLocator` pane constitutes a GUI to RemoteLocator.

Currently, these four freely usable collections are supported:

IRDB
This collection was until recently available at the server `irdb.ik`, which is now defunct. The IRDB format is a very simple CSV format, containing command name, protocol and the device and subdevice parameters, but no meta information. Instead, information on the manufacturer and the device class are gleaned from the file path.

Lirc remotes
Lirc's collection of remotes, which are collected in the Sourceforge project. This format also contains no meta information. The manufacturer information is gleaned from the file path; however there are no information on device class, so all the Lirc remotes have device class "unknown" in the generated list.

GirrLib
Girr is a very versatile format for IR remotes. It is the native format for IrScrutinizer, and also supported by main program RMIR of the JP1 project. Meta information, including manufacturer and device class, is contained in the Girr file. There is a small collection (Girrlib), which is more of a proof-of-concept than a sizeable collection of remotes.

JP1
The JP1 project has a large collection of "device upgrades". These are summarized in an Excel file. This list also contains meta information such as manufacturer and device class. It can be read into the program, and used to browse the therein contained remotes (or rather, "device upgrades"). These can however not be directly loaded, but can with some manual work be translated to Girr files.

To use, first select `Select me to load` to load the list of the manufacturers. Then select, in order, a manufacturer, a device type, and a remote name. The field `kind` now indicates what kind of remote that was found. Pressing `Load` (if enabled) will load the remote into the tree importer, where it can be further manipulated.

Pressing the `Load all` button transfers all present protocol/parameters combinations to the tree.

File importers

There are a number of elements common to most of the sub-panes, so these will be described next.

For file/URL based imports, there is a text field, named File or File/URL. For the latter case, an URL (like `http://lirc.sourceforge.net/remotes/yamaha/RX-V995`) can be entered, for subsequent import without downloading to a local disc. By pressing the `...`-Button, a file selector allows the selection of a local file. For files and URLs, the `Edit/Browse` button allows to examine the selected file/URL with the operating system's standard command.

Several of the formats (the one based on text files, but not on XML (which carries its own character set declaration) allow the user to select the character set to be used for the import to be selected. This option is found as `Options -> Import options -> Character Set...`

Most of the file based importers support dropping a compatible file from the GUI on the import window.

If the option `Open ZIP files` (accessible from `Options -> Import`) is selected, zip files can be selected, opened, and unzipped "on the fly", without the need for the user to unzip to intermediate files.

When pressing one of the `Load`, `Load File/URL`, or `Load from clipboard` button, the selected information is downloaded, and presented in a tree importer.

The file based import formats allow a file to be "dropped" with the mouse on the main window.

Girr (the native format of IrScrutinizer)

The Girr format is the native format of IrScrutinizer. The importer is capable of importing directory hierarchies.

Lirc

The Lirc import feature is based upon Jirc, which is basically a subset of Lirc 0.9.0 translated to Java. The Lirc importer can even import a file system hierarchy by selecting the top directory as File/URL. (Importing the entire lirc.org database with over 100000 commands takes around 1 minute and 1GB of memory.)

Only Lirc remotes with timing information can be imported, not the so-called lircccode remotes. See this article for a more extensive discussion.

CML

The CML format is the format used by the RTI Integration Designer software. Many CML files are available in Internet, in particular on RemoteCentral. Particularly noteworthy is the "megalist" by Glackowitz. IrScrutinizer can import these files to its import tree, making every remote a nodes in the tree. Note that there is no need to unzip such a file; IrScrutinizer will unzip it on the fly.

Command Fusion

The native format that Command Fusion uses, file extension `.cfir`, can be imported here.

Pronto Classic (CCF format)

Many Pronto CCF files are available in Internet, in particular by Remote Central. IrScrutinizer can read in these files to its import tree, even preserving the Pronto "devices" as nodes in the tree.

Pronto Prof. (XCF format)

Pronto Professional XCF files are found for example at Remote Central. IrScrutinizer can read in these files to its import tree, even preserving the Pronto "devices" as nodes in the tree.

This is not extensively tested.

ICT IrScope format

The ICT format, introduced by Kevin Timmerman's IrScope, contains the timing pattern, the modulation frequency, and optionally a name (`note`) of one or many IR signals.

Text format

In the Internet, there are a lot of information in table-like formats, like Excel, describing the IR commands of certain devices. IrScrutinizer has some possibilities of importing these — after exporting them to a text format, like tab separated values (tsv) or comma separated values.

Raw

The sub-pane allows for the parsing of text files separated by a certain characters, like commas, semicolons, or tabs. The separating characters is selected in the `Field separator` combo box. The column to be used as name is entered in the `Name col.` combo box, while the data to be interpreted either as raw data or CCF format, is entered in the `Raw signal col.`. If the `... and subsequent columns` is selected, all subsequent columns are added to the data.

Raw, line-based

This pane tries to interpret a line-based file as a number of named IR commands, using heuristics.

Parameterized

The sub-pane allows for the parsing of text files separated by a certain characters, like commas, semicolons, or tabs. The separating characters is selected in the `Field separator` combo box. The column to be used as name is entered in the `Name col.` combo box, while protocol name and the parameters D, S, and F are entered in their respective combo boxes. They are parsed in the number base selected.

Wave

This pane imports and analyzes wave files, considered to represent IR signals. The outcome of the analysis (sample frequency, sample size, the number of channels, and in the case of two channels, the number of sample times the left and right channels are in phase or anti-phase) is printed to the console.

IrTrans

IrTrans' configuration files (`.rem`) can be imported here.

2.4.5.5 The Export pane

Using the export pane, export files can be generated, allowing other programs to use the computed results. Single signals (from the `Scrutinize signal` pane), collections of signals (from the `Scrutinize remote` pane), or rendered signals can be exported. Exports can be generated in a number of different formats. Some (Girr and text) can contain both the Pronto format and the "raw" format (timings in microseconds, positive for pulses, negative for gaps), as well as other formats. These formats, together with Wave and Pronto Classic, are built-in in the program. However, it is possible to define new export formats by extending a configuration file, see Adding new export formats.

The file names of the exports are either user selected from a file selector, or, if `Automatic file names` has been selected, automatically generated.

The export is performed by pressing the one of the Export buttons. The `...`-marked button allows for manually selecting the export directory. It is recommended to create a new, empty directory for the exports. The just-created export file can be immediately inspected by pressing the `Open last file`-button, which will open it in the "standard way" of the operating system. (Also available on the actions menu.) (Girr exports become a special treatment in order not to invoke another instance of the IrScrutinizer. They are copied to a temporary `.txt` file.) The `Open` button similarly opens the operating systems standard directory browser (Windows Explorer, Konquistador, Nautilus,...) on the export directory.

Some export formats (presently Wave, mode2, and Lintronic) export an IR sequence rather than an IR signal (consisting of an intro sequence, an repetition sequence (to be included 0 or more times), and an (most often empty) ending sequence). When using these formats, the number of repetition sequences to include can be selected.

The character set used for the export can be selected through `Options -> Export options -> Character set...` Note however that this makes sense only for text based formats, including XML.

Some export formats have some more parameters, configured in sub panes. These will be discussed in the context of the containing formats.

(Most of) the formats presently implemented will be described next, in alphabetical order.

AnyMote

Generates a configuration file for the AnyMote IR remote control app for Android and iOS.

Arduino/Raw

Generates a complete C++ sketch for the Arduino. This uses one of the three Arduino Infrared libraries IRremote, IRLib, and Infrared4Arduino. As the name suggests, the raw form of the signals are used.

Arduino/Infrared4Arduino

Generates a similar C++ sketch for the Arduino as in the raw case, but tries to use the parametric form whenever possible, i.e., whenever the protocol is one that can be generated by the underlying infrared library. This version supports Infrared4Arduino only.

Arduino/IRremote

Like the preceding, but supports the IRremote library instead. (Currently, needs update to the current IRremote library.)

BracketedRaw

This export format generates an text file of the command(s) in the bracketed raw text format.

C

Generates a C code fragment consisting of declarations of the signals in raw- and CCF format. Intended mostly as an example.

Girr

The program's native format, based on XML. Very flexible and extensible. Can contain information like the raw format, CCF format, UEI learned format, and the Global Caché sendir format. Official documentation site. There are some extra parameters that can be configured in subpanes, described next:

The Girr sub-pane

A style sheet can be selected to be linked in into the exported Girr file. The type of style file (presently xslt and css) can also be selected.

`Fat form raw` can be selected; this means that the raw signals are not given as a text string of alternating positive and negative numbers, but the individual flashes and gaps are enclosed into own XML elements. This can be advantageous if generating XML mainly for the purpose of transforming to other formats.

HTML

The purpose with the HTML export is to generate a browse-able page, not something for other IR-programs to feed upon.

ICT

The ICT format, introduced by Kevin Timmerman's IrScope, contains the timing pattern, the modulation frequency, and optionally a name (`note`) of one or many IR signals. Can be imported by IrScope and RemoteMaster.

irplus

Generates a configuration file for the Android ID remote irplus.

IrToy

A text version of the following

IrToy-bin

Generates a binary file "in IrToy format" that can be send to the IrToy using the *program* `irtoy[.exe]`, see this thread.

IrTrans

This export format generates .rem files for the IrTrans system, using its CCF format.

Lintronic

Simple text protocol for describing a single IrSequence.

Lirc Raw

The Lirc-exports are in `lircd.conf` - raw format. These use the raw Lirc format, except for a few well known protocol (presently NEC1 and RC5). They can be concatenated together and used as the Lirc server data base. Can also be used with WinLirc.

Lirc

Like `Lirc Raw`, but recognizes a large number of the protocols, for which lircd.conf files in "cooked" (opposite of raw) are generated. For other protocols, it falls back to the raw form.

mode2

A primitive debugging "signal format" used by Lirc, consisting on interleaved on- and off durations.

Pronto Classic

This format generates a CCF configuration file to be downloaded in a Pronto, or opened by a ProntoEdit program.

The Pronto Classic sub-pane

A Pronto Classic export consists of a CCF file with the exported signals associated to dummy buttons. The Pronto (Classic) model for which the export is designed is entered in the combo box. Screen size of the Pronto is normally inferred from the model, but can be changed here. The button size of the generated buttons is also entered here.

Pronto Hex Oneshot

This export format takes a single signal, makes a sequence of it consisting of the intro sequence, an arbitrary number of copies of the repeat sequence, and the (in general empty) ending sequence. This is packed into a Pronto Hex format, having the said sequence as its intro, and empty repeat.

Spreadsheet

Simple tab separated value export format for importing in a spreadsheet program. This format is mainly meant to demonstrate a simple format in XSLT, more than a practically useful format.

Text

The text format is essentially the Girr format stripped of the XML markup information.

TV-B-Gone

Variant of the C format, this format generates C code for the TV-B-Gone.

Wave

IR sequences encoded as wave audio files.

The Wave sub-pane

Parameters for the generated Wave export (except for the number of repeats) can be selected here.

2.4.5.6 The "Hardware" pane

The sub-panes of this pane allows for the selection and configuration of the deployed IR sending/capturing hardware.

A opposed to versions earlier than 2.4.0, there is exactly one *selected device*, corresponding to the the selected sub-pane of the `Hardware` pane. This device is, if capable, used for both sending and capturing. The selected device may be opened or not. An opened device may be selected or not. There may be more than opened device. The hardware can also be selected from the tool bar, Options -> Capturing hardware.

Note that by e.g. selecting non-existing hardware or such, there is a possibility encounter long delays, or even to "hang" the program.

After configuring and opening the capturing hardware, the `Test` button can be used for testing the configuration without switching pane.

The currently selected ports etc. are stored in the properties, thereby remembered between sessions. So, for future sessions, only opening the preferred device is necessary.

Capturing parameters

Capturing an IrSequence is governed by the parameters `beginTimeout`, `captureMaxSize`, `endingTimeout`, which can be accessed in the `Options -> Timeouts` submenu. `beginTimeout` and `endingTimeout` bothe use the unit milli seconds. Unfortunately, not all hardware respect these parameters.

beginTimeout
When capturing starts, this determines how long the capturer is waiting for the first flash. Default is 3000ms.

captureMaxSize
This is the maximal duration the capturing hardware will accept. Default is 1000.

endingTimeout
The silence period required at the end of an IrSequence. Default is 300ms.

The "Global Caché" pane.

IrScrutinizer automatically detects alive Global Caché units in the local area network, using the AMX Beacon. However, this may take up to 60 seconds, and is not implemented in very old firmware. Using the `Add` button, the IP address/name of older units can be entered manually.

The user can select one of the thus available Global Caché units, together with IR-module and IR-port (see the Global Caché API specification for the exact meaning of these terms).

For this to work, port 9131/udp must be open in a used firewall

The `Browse` button points the user's web browser to the selected unit.

The reported type and firmware version serves to verify that the communication is working.

`Stop IR`-Button allows the interruption of ongoing transmission, possibly initiated from another source.

/dev/lirc (Linux only)

On Linux, so-called mode-2 capable IR hardware using the `/dev/lirc` device, can be accessed directly, both for sending and receiving. When connected, and in some cases after loading suitable drivers, a device file `/dev/lirc` *n* (for *n* = 0, 1, 2,...) will be created. Of course, this must be read- and/or writeable by the current user; in e.g. Fedora this is the case for member of the group `lirc`.

After opening the device, its properties will be listed. See the man page lirc(4) for an explanation.

Although not a priori impossible, to my knowledge no `/dev/lirc` device implements frequency measurement.

The "Audio Port" Pane

As additional "hardware sending device", IrScrutinizer can generate wave files, that can be used to control IR-LEDs. This technique has been described many times in the Internet the last few years, see for example this page within the Lirc project. The hardware consists of a pair of anti-parallel IR-LEDs, preferably in series with a resistor. Theoretically, this corresponds to a full wave rectification of a sine wave. Taking advantage of the fact that the LEDs are conducting only for a the time when the forward voltage exceeds a certain threshold, it is easy to see that this will generate an on/off signal with the double frequency of the original sine wave. (See the first picture in the Lirc article for a picture.) Thus, a IR carrier of 38kHz (which is fairly typical) can be generated through a 19kHz audio signal, which is (as opposed to 38kHz) within the realm of medium quality sound equipment, for example using mobile devices.

It can only send, not receive/capture.

IrScrutinizer can generate these audio signals as wave files, which can be exported from the export pane, or sent to the local computers sound card. There are some settings available: Sample frequency (44100, 48000, 96000, 192000Hz), sample size (8 or 16 bits) can be selected. Also "stereo" files can be generated by selecting the number of channels to be 2. The use of this feature is somewhat limited: it just generates another channel in opposite phase to the first one, for hooking up the IR LEDs to the difference signal between the left and the right channel. This will buy you double amplitude (6 dB) at the cost of doubling the file sizes. If the possibility exists, it is better to turn up the volume instead.

Most of "our" IR sequences ends with a period of silence almost for the half of the total duration. By selecting the `Omit trailing gap`-option, this trailing gap is left out of the generated data – it is just silence anyhow. This is probably a good choice (almost) always.

Note that when listening to music, higher sample rates, wider sample sizes, and more channels sound better (in general). However, generating "audio" for IR-LEDs is a completely different use case. The recommended settings are: 48000kHz, 8bit, 1 channel, omit trailing gap.

The "Girs Client" (previously "Arduino") Pane

Using this pane, a Girs server (e.g. running on an Arduino equipped with a suitable hardware) can be used to transmit and capture IR signals. The server can be connected to a serial port, or through Ethernet to the local area network, connected by a TCP socket. For the Arduino, the sketch `GirsLite` (or `Girs`) can be used.

The serial port may sometimes be finicky. Sometimes disconnecting and reconnecting the device may help.

To use the device connected to a real or virtual serial port, select the port in the combo box, pressing `Refresh` if necessary. `Open` the port thereafter. The opening of LAN connected device is simliar.

Normally, the Girs server uses the Girs `analyze` command to capture IR signals, which is normally deploying a non-demodulating sensor, providing a frequency measurement. If this is not desired (for example when the non-demodulating sensor is missing), selecting `Use receive for capture`, instead the `receive` command will be used, which normally uses a demodulating sensor. In this case, no frequency measurement will be produced, and instead the fallback frequency (selectable/changeable as `Options -> Fallback frequency`) will be used.

The "CommandFusion" Pane

Using this pane, the CommandFusion learner can be used to transmit and capture IR signals.

After connecting the Command Fusion learner to a USB port, select the the actual (virtual) serial port in the combo box, pressing `Refresh` if necessary. `Open` the port thereafter.

The "IrWidget" pane

Using this pane, the IrWidget can be used to capture IR signals.

After connecting the IrWidget to a USB port, select the the actual (virtual) serial port in the combo box, pressing `Refresh` if necessary. `Open` the port thereafter. If using the original Kevin Timmerman widget, for example built and sold by Tommy Tylor, the option `lower DTR abd RTS on opening` must be selected.

2.4.5.7 Key bindings and accelerators

The GUI contains a number of key bindings and accelerators. These can be found in menus and in the text of GUI elements. A more substantial description is planned, but not yet written.

2.4.6 Command line arguments

Normal usage is just to double click on the jar-file, or possibly on some wrapper invoking that jar file. However, there are some command line arguments that can be useful either if invoking from the command line, or in writing wrappers, or when configuring custom commands in Windows.

Girr files given as command line argument will be imported to the `parametric remote` table. Accordingly, if the system is configured to associate `*.girr` with IrScrutinizer, these files can be opened by double clicking them.

The options `--version` and `--help` work as they are expected to work in the GNU coding standards for command line interfaces. Use the `--help`-command to see the

complete list of command line parameters. The `-v/--verbose` option set the verbose flag (which can also be accessed under the GUI as `Options -> verbose`, causing commands like sending to IR hardware printing some messages in the console.

Using the option `--properties` (or `-p`), an alternative property file can be used. If installing more than one version of the program, this is recommended practice.

The option `--nuke-properties` makes the program delete the property file, and exit immediately.

The option `--scaling` (or `-s`, `--scale`) set the scaling of the GUI. Accepted values and their semantics depend on the JVM used.

IrpTransmogrifier as well as HarcHardware as command line program can be invoked with the AppImage installation by calling it using the name `irptransmogrifier` or `harchardware` (through copying or linking).

The MacOS App does not support calling IrpTransmogrifier or HarcHardware; install the generic binary version if needed.

The program delivers well defined and sensible exit codes, listed in the code.

For automating tasks, or for integrating in build processes or Makefiles or the like, it is probably a better idea to use IrpTransmogrifier instead, which has a reasonably complete command line interface.

2.4.7 Adding new export formats

Only a few of the many export formats are defined in the main Java code (`Girr`, `Text`, `Pronto Classic` and `Wave`), the rest are defined in files in the directory `exportformats.d`, located in the root of the install directory. By adding a file here, the user can simply add his/her own export formats according to own needs. An export format consists of a number of properties, together with a small "program" written in the transformation language XSLT, for transforming a Girr-XML-tree to the desired text format.

The rest of this section documents the format of these files, and is supposed to be read only when needed. Fundamental knowledge of XML and XSLT transformations are assumed.

2.4.7.1 Format of the files in exportformats.d

The file is an XML file supposed to be a valid instance of the XML Schema `exportformats.xsd`, although it is read without validation. The semantics is believed to be essentially self explaining, or clear from the examples already in there. An export format is packed in an element of type `exportformats:exportformat`. It contains the following attributes:

`name`
Text string used for identifying the format.

`extension`

Text string denoting preferred file extension (not including period) of generated files.

`multiSignal`

Boolean (value: `true` and `false`). Denotes if several signals can be represented in one export, or only one.

`simpleSequence`

Boolean, (values `true` of `false`). If true, the export really describes an sequence rather than am signal (with intro-, repeat- , and ending-sequences), therefore the user must explicitly state a particular number of repetitions for an export.

The element `exportformats:exportformat`contains as a child element the XSLT transformation, which is an element of type `xsl:stylesheet`, with attributes as in the examples.

The task of the transformaton is to transform a Girr XML DOM tree in the "fat" format, with `remotes` as root element, into the desired text format. It may be advisable to use the already present formats as guide.

For testing and developing new export formats in this for, the `main` routine of the class `org.harctoolbox.irscrutinizer.exporter.DynamicRemoteSetExportFormatMai` may be used. It can be used to transform a Girr file from the command line.

After editing or adding export format files, they can be reloaded either by re-starting the program, or by selecting `Options` -> `Export formats database` -> `Reload`.

2.4.8 Properties

Under Windows, the persistent properties are stored in `%LOCALAPPDATA%\IrScrutinizer\IrScrutinizer.properties.xml` (typically `%HOME%\AppData\Local\IrScrutinizer\IrScrutinizer.properties.xml`). Using other operating systems, it is stored according to the FreeDesktop specification. Per default, this is `$HOME/.config/IrScrutinizer/properties.xml`. It is not deleted by un-install or by an update of the program. If weird problems appear, for example after an update, try deleting this file, from the GUI `File -> Set properties to default`. There is also a command line option `--nuke-properties` that can be used to conveniently delete the present property file, without remembering its exact path.

2.4.9 Questions and Answers

2.4.9.1 How do I verify the integrity of my downloads?

On the official download site, the checksums (using the MD5, SHA-1, and SHA-512 algorithms) are available in the files `checksums.md5`, `checksums.sha1`, and `checksums.sha512`. With appropriate software, these can be used for validating the downloaded files.

To prevent from tampering of *both* the program files *and* the checksum files, the checksum files are also PGP signed with the cryptograph key for

`Bengt Martensson <barf@bengt-martensson.de>`, finger print `5F492CB76BEB2E6B1CAA63A3EE1F17D4ECF8AF9C`. The full key can be downloaded here, or found in the program as `Tools -> Author's Public PGP key`. (For practical reasons, the shapshot releases are, in general, not signed.)

How to check these checksums and signatures differ between different programs and operating systems, and is therefore not described here. However, searching on the Internet will yield many descriptions.

2.4.9.2 My virus program says that your program contains virus or malware.

My programs do not come with virus. Downloading from un-official sites are discouraged, in particular if they do not keep the signed checksum files. (Most likely, they do not.)

First check the integrity of the downloaded files as described in the previous question. Assuming that the problem remains, this is a problem with your virus program, and not with the present program, and it should be handled as such. If it is a release version, please inform the author of the virus program, and/or create a white list. What you do not need to do, is to tell me, or "the world". Remember, it is a problem with the virus program, not with my software.

2.4.9.3 Why have RemoteMaster Import and Export been removed?

The RemoteMaster export and import found in earlier versions (up to 2.3.0) of this program have been removed. The reason is that the current version of that program contains elaborate facilities for the import and export of Girr files — the native format for IrScrutinizer. As opposed to the previous import and export of the current program, they contain elaborate rules for transforming between protocols and the JP1 executors.

2.4.9.4 Can I run more than one instance simultaneously?

Yes, you can fire up several instances, for example, one for sending and another for receiving the sent signals. The only possible problem is that the properties are read from/written to the same file, so the instance that ends last will overwrite the properties of the first one. This does not have to be a problem. However, if this is an issue, you can start the program with the `--properties` *filename*, pointing to an alternative properties file. With Windows, you can duplicate the icon, and edit the command line on one of those.

2.4.9.5 I miss DecodeIR!

Support for DecodeIR was *removed*, not just hidden and deprecated, because keeping it would increase maintenance and installation effort substantially. For accessing DecodeIR, it is recommended to install version 1.4.3 (which is the final release with DecodeIR) in parallel to the current one, see the previous answer for details.

2.4.9.6 Firewall issues?

Except for the usual HTTP(S) ports, the program uses some TCP and UDP ports for communicating with some of the supported hardware. These are:

- For GlobalCache this is TCP port 4998. Its discovery beacon uses UDP port 9131.
- Lirc uses a TCP port, per default 8765.
- AGirs on an ethernet connected board uses a TCP port, per default 33333.

2.4.9.7 Does IrScrutinizer completely replaces IrMaster?

Almost. Using MakeHex as renderer (or more correctly, its Java version) is not implemented. (The practical usage of this feature is probably *very* limited, and IrMaster is still available, should it ever be needed.) The "war dialer" is also not implemented, but see next question. For the wave export, some rarely used options (the possibility to select big-endian format (for 16-bit samples), the possibility *not* to half the carrier frequency, and the possibility to select sine (instead of square) for modulation) have been removed. Finally, there is some stuff that simply works differently, like the export function.

2.4.9.8 How do I emulate the war dialer of IrMaster?

Use `Scrutinize remote -> Parametric Remote`. Fill in the table with signals to be tested, either using the pop-up button (right mouse in the table) Advanced -> Add missing F's, or from the Render pane, using suitable parameter intervals (see this), and transfer them using the `To parametric remote` button. Then test the candidate signals one at a time by transmit-ting them, using suitable sending hardware. The comment field can be used for note taking.

A "war dialer" like in IrMaster may be implemented in a later version.

2.4.9.9 Can I use this program for conveniently controlling my favorite IR controlled device from the sofa?

No, the program is not meant for that. While you definitely can assemble a "remote" on the `scrutinize remote` panel, and transmit the different commands with mouse commands (appropriate hardware assumed), the program is intended for developing codes for other deployment solutions.

Check out JGirs, which is an IR server implementing the Girs specification. It can be considered as "IrScrutinizer as a server".

2.4.9.10 The pane interface sucks.

Yes. There are several use cases when the user would like to see several "panes" simultaneously. Also, it should be possible to have several windows of the same sort (like the `scrutinize signal`) simultaneously. Replacing the top level panes with something "Eclipse-like" (sub-windows that can be rearranged, resized, moved, iconized) is on my wish list.

2.4.9.11 What about the "fishy" icon?

It is a Babel fish, as found in The Hitchhiker's Guide to the Galaxy, having the property, that "... if you stick one in your ear, you can instantly understand anything said to you in any form of language". This symbolizes the program's ability to "understand" (read and write) a large number of different IR formats.

2.4.9.12 I did something funny, and now the program does not startup, with no visible error messages.

Try deleting the properties file. (Note the command line option `--nuke-properties` which will do exactly that, without having to manually find the file or its name.) If that does not help, try starting the program from the command line, which may leave hopefully understandable error message on the console.

2.4.9.13 (Windows) When I double click on the IrScrutinizer symbol, instead of the program starting, WinRar (or some other program) comes up.

The program that comes up has "stolen" the file association of files with the extension `.jar`. Restore it. (Allegedly, WinRar can gracefully "unsteal" file associations.)

2.4.9.14 (Windows) Why is the program so slow at start-up?

Normal start up time is a few seconds, about the same time as it takes for (e.g.) a web browser to start. If it takes considerably longer than that, the problem is almost surely hardware related, i.e., the program looks for possibly non-existent hardware. To fix this, make sure that the selected capture- and sending devices (these are save between sessions!) do not correspond to something non-existent. The most "innocent" settings are: capture: Lirc mode 2, and sending: Audio port. These are also the defaults when the program starts up for the first time. Also, consider deleting possible "junk devices" in the Windows device manager under `COM & LPT` (unused Bluetooth-to-serial ports etc.).

2.4.9.15 (Linux) I get error messages that lock files cannot be created, and then the serial hardware do not work.

This answer needs updating.

When starting IrScrutinizer, or by pressing the `Refresh` button, error messages occur like

```
check_group_uucp(): error testing lock file creation Error
details:Permission deniedcheck_lock_status: No permission to
create lock file.
```

(and a number of them...) The problem is that the library `rxtx` (like some other program accessing a serial interface) wants to create a lock file in a particular directory. This is/was traditionally `/var/lock`, which is often a symbolic link to `/var/run/lock`. This directory is normally writable by members of the group `lock`. So your user-account should probably be a member of that group. (How to perform this is different in different

Linux distributions.) The rxtx library delivered with IrScrutinizer expects the lock directory to be `/var/lock`. However, recently some Linux-distributions (e.g. Fedora 20), instead are using `/var/lock/lockdev` as its lock directory (while `/var/lock` still is a link to `/var/run/lock`. To solve this, I recommend, if possible, installing rxtx provided by the Linux distribution used, i.e. not using the one with IrScrutinizer. For example, on Fedora, the command is

```
sudo dnf install rxtx
```

which installs the library in `/usr/lib/rxtx` or `/usr/lib64/rxtx`, depending on the operating system. (Other distributions uses other commands, for example `apt-get` on Debian-like systems.) Finally, the correct installation directory of the library (`librxtxSerial-2.2pre1.so`) has to be given to the JVM running IrScrutinizer. For this, see the wrapper `irscrutinizer.sh` and the comments therein, and make the necessary adaptations.

2.4.9.16 What is on the splash screen?

From left to right, a Global Caché iTach Flex, an IrToy, and a low-cost clone of an Arduino Nano, the latter equipped with a non-demodulating IR detector (TSMP4138) for capturing and an IR diode (SFH415) for sending. These are all hardware which work well with IrScrutinizer, both for sending and capturing.

2.4.9.17 Why do I get multiple "Really exit?" confirmation questions when exiting?

If the "Scrutinize Remote" parametric table is non-empty and not saved, the user is asked to acknowledge the exit, since he/she may otherwise loose carefully entered data. Exactly the same thing goes for the raw table. So, it is not "multiple", it may be at most two; they are not identical (but very similar). There is also a "Do not ask this again" checkbox.

2.4.10 Appendix. Building from sources

"IrScrutinizer" is one subproject within harctoolbox.org. It depends on several other subprojects within harctoolbox. The repository IrScrutinizer consists of this subproject.

The released versions are found on the download page. The development sources are maintained on my GitHub repository. Forking and pull requests are welcome!

I go to great lengths ensuring that the program runs equally well on all supported platforms. I do not care too much that all aspects of the build runs equally well on all platforms. I build with Linux (Fedora), the continuous integration build runs on GitHub Actions (deploying Ubuntu). Other platforms are treated step-motherly.

2.4.10.1 Dependencies

As any program of substantial size, IrScrutinizer uses a number of third-party components. All of these are also free software, carrying compatible licenses. The dependent packages need to be installed also in the host-local Maven repository in order for the build to work. In some cases (basically the ones with a version not ending with `-SNAPSHOT`), the packages are uploaded to the Maven central repository, and will be automatically downloaded to the local host by a Maven invocation.

There are some scripts to aid downloading and building, described next. It is assumed that git and Maven are installed and available as commands `git` and `mvn` respectively.

Of course, it is also possible to manually download or clone these packages from my Github repositories, then build and install them locally by `mvn install`.

IrpTransmogrifier, Girr, HarcHardware, Jirc

These are all Java packages that are required to build IrScrutinizer. HarcHardware requires nrjavaserial. They can be downloaded and built by the script `common/scripts/build-harctoolbox-project.sh`, using an argument of `IrpTransmogrifier`, `Girr`, `HarcHardwar`, or `Jirc`. See the file `.github/workflows/maven.yml` for the complete commands.

DevSlashLirc

This library is used to access `/dev/lirc`-hardware. It is used by the Linux version only. It is a Java JNI library, written in Java and C++. It is written by myself, and available here.

The subdirectories `native/Linux-amd64`, `native/Linux-i386`, and `native/Linux-arm` contain compiled versions for the x86_64, x86_32, and ARM processors respectively.

The package can be downloaded, and the Java part built, by the script `common/scripts/build-harctoolbox-project.sh` using the argument `DevSlashLirc` (see `.travis.yml` for an example).

nrjavaserial

This is a fork of the legacy RXTX library for serial communication with Java. Version 5.2.1 is currently used.

JCommander

Normally, this component is downloaded and installed automatically by Maven.

Tonto

Tonto can be downloaded and installed by the script `common/scripts/build-tonto.sh`. It requires the Apache ant build system to be installed as the command `ant`.

Note that the shared library `libjnijcomm`, which is required by the Tonto program for communicating with a Pronto remote through a serial interface, is not required for use with IrScrutinizer, and can therefore be left out. Using the option `-n` to the script (see `.travis.yml` for an example), the script will not try to build and install the shared library.

2.4.10.2 Building

The Maven "software project management and comprehension tool" is used as building system. Modern IDEs like Netbeans and Eclips integrate Maven, so build etc can be initiated from the IDE. Of course, the shell command `mvn install` can also be used. It creates some artifacts which can be used to run IrScrutinizer in the `IrScrutinizer/target` directory.

Two additional shell tools are needed. These are:

- The `unix2dos` and `dos2unix` utilities, typically in the `dos2unix` package.
- The `icotool` utility, typically in the `icoutils` package.

2.4.10.3 Windows setup.exe creation

For building the Windows setup.exe, the Inno Installer version 6 is needed. To build the Windows `setup.exe` file, preferably the work area should be mounted on a Windows computer. Then, on the Windows computer, open the generated file `IrScrutinizer/target/IrScrutinizer_inno.iss` with the Inno installer, and start the compile. This will generate the desired file `IrScutinizer-` *version* `.exe`.

Alternatively, the "compatibility layer capable of running Windows applications" software application Wine (included in most Linux distributions) can run the ISCC compiler of Inno. The Maven file `IrScrutinizer/pom.xml` contains an invocation along these lines, conditional upon the existence of the file `../Inno Setup 6/ISCC.exe`.

2.4.10.4 Mac OS X app creation

The Maven build creates a file `IrScrutinizer-` *version* `-macOS.dmg`. This file can be directly distributed to the users, to be installed according to these instructions.

The icon file `IrScrutinizer.icns` was produced from the Crystal-Clear icon `babelfish.png` in 128x128 resolution, using the procedure described here.

2.4.10.5 AppImage creation

To build the x86_64 AppImage, define `bundledjdk_url_sans_file` and `bundledjdk` in `pom.xml` to point to a suitable JDK distrubution file. If a file with name given by `bundledjdk` is not present in the top level directory, it can be downloaded by the script `tools/get-jdk-tar.sh`. Then the maven goal `make-`

`appimage` (which will be invoked during a normal build) will build an appimage for x86_64.

2.4.10.6 Test invocation

For testing purposes, the programs can be invoked from their different target directories. IrScrutinizer can be invoked as

```
$ java -jar target/IrScrutinizer-version-jar-with-dependencies.jar
```

or, if the shared libraries are required, with *path-to-shared-libraries* denoting the path to a directory containing the shared libraries.

```
$ java -Djava.library.path=path-to-shared-libraries -jar target/
IrScrutinizer-version-jar-with-dependencies.jar
```

IrScrutinizer can also be started by double clicking the mentioned jar file, provided that the desktop has been configured to start executable jar with a Java virtual machine.

2.4.10.7 Installation

Maven does not support something like `make install` for deployment installing a recently build program on the local host. Instead, the supplied `tools/Makefile` can install the program to normal Linux locations (in the Makefile `INSTALLDIR`),

```
sudo make -f tools/Makefile install
```

Equivalently, the just created generic-binary package `IrScrutinizer/target/IrScrutinizer-*-bin.zip`) can be installed using these instructions.

2.5 IrpTransmogrifier: Parser for IRP notation protocols, with rendering, code generation, and recognition applications.

> **Note:**
> Possibly you should not read this document! If your are looking for a user friendly GUI program for generating, decoding, and analyzing IR signals etc, please try the GUI program IrScrutinizer, and get back here if (and only if) you want to know the details on IR signal generation and analysis.

Date	Description
2019-08-06	Initial version, for IrpTransmogrifier Version 1.0.0.
2019-08-11	Minor tweaks for version 1.0.1.

Date	Description
2019-12-27	Update for version 1.2.4.
2020-05-19	Update for version 1.2.6. More of an improvement and clarification than a description of new features.
2020-06-27	Update for version 1.2.7: IRP database maintenance.
2021-06-03	Update for version 1.2.10: Minor improvements.
2022-05-07	Update for version 1.2.11: Spelling fixes etc.
2024-01-08	Minor clarifications: Described predicates in prefer-overs and non-emptyness requirement for repeats.

Table 1: Revision history

2.5.1 Release notes

Release notes for the current version

2.5.2 Introduction

The *IRP notation* is a domain specific language for describing IR protocols, i.e. ways of mapping a number of parameters to infrared signals. It is a very powerful, slightly cryptic, way of describing IR protocols, that was first developed by John Fine in 2003. In early 2010, Graham Dixon (mathdon in the JP1-Forum) wrote a specification.

This project contains a parser, a number of basic classes, and utilities for IRP protocols. It does not contain a graphical user interface (GUI). See Main principles for a background. Instead, the accompanying program IrScrutinizer provides GUI access to most of the functionality described here.

This is a new program, written from scratch, that is intended to replace IrpMaster, DecodeIR, and ExchangeIR, and much more, like potentially replacing hand written decoders/renders. The project consists of an API library that is also callable from the command line as a command line program.

The name indicates that the program transforms ("transmogrifies") IRP form protocols to and from all sort of different things.

For understanding this document and program, a basic understanding of IR protocol is assumed. However, the program can be successfully used just by understanding that an "IRP protocol" is a "program" in a particular domain specific language for turning a number of parameters into an IR signal, and the present program is a compiler/interpreter/decompiler of that language. Some parts of this document requires more IRP knowledge, however.

2.5.2.1 Background

This program can be considered as a successor of IrpMaster. The IRP parser therein is based upon a ANTLR, using the now obsolete version 3. The "new version" ANTLR4 is really not a new version, but a completely different tool, fixing most of the quirks that were irritating in IrpMaster (like the "ugliness" of embedding actions within the grammar and no left recursion). Unfortunately, this means that using version 4 instead of version 3 is not like updating a compiler or such, but necessitates a complete rewrite of the grammar and the actions.

It turned out that IrScrutinizer version 1 (i.e. the version based upon IrpMaster for generating and DecodeIR for decoding) had a fundamental problem: Although DecodeIR and IrpMaster (the latter through the data base `IrpProtocols.ini`) agree on most (but not all) protocols, they relied upon two different, dis-coupled sources of protocol information. If the data base for the sending protocols were changed or extended, this affected sending only, while decoding was not changed, and potentially conflicted with the sending protocols. Also, both DecodeIR and the IR Analyzer, used in IrScrutinizer versions before 1.3, have shown to be essentially impossible to maintain and extend.

2.5.2.2 Dependencies

The program is written entirely in Java, using no native libraries. Therefore, it runs on anything for which Java (presently at least version 8) is available. Except for building (using Maven and a number of its plugins), testing of the program (using TestNG), and of course the Java framework, it depends on ANTLR4, StringTemplate, and the command line decoder JCommander.

In order to avoid circular dependencies, it is prohibited to use any other Harctoolbox software.

2.5.2.3 Documentation principles

The role of program documentation has changed considerably the last few decades. Ideally, a program does not need any documentation at all, if its operation is entirely obvious. Unfortunately, as non-trivial problems almost never have trivial solutions, so some sort of documentation is almost always necessary for programs solving complex programs. The next best thing is that the program contains its own documentation, for GUI programs through tool-tips, info-popups etc, for command line programs through help texts. This can contain precise information over the syntax and semantic of, for example, commands. Since they are integrated in the program and maintained together with the program source, it is less likely that it lacks behind the actual program behavior, than in the traditional program manual like this. They may also change faster than reference manual.

Documentation for API, and structures like XML Schemas should preferably be integrated into the source itself, using technologies like Javadoc, Doxygen, and, for XML

documents, including Schemas etc, be documented using the available documentation elements.

Separate program documentation, on paper or electronically (like the present document), should then play another role than in the last century. It should be centered around explaining the concepts of the program, the "why", and not the "how". Some details that cannot in a natural way be explained within the program with a few sentences can also be covered.

The present document is written in that spirit. We do not explain all commands and parameters, at least not the ones that are (more-or-less) obvious. Also, since the program changes more often than the manual, it is written to be a "nice" document, not to exactly document a particular version of the program.

Since a text like this is unlikely to be read sequentially from start to finish, redundancy and repetitions are not considered evil and a sign of stylistic inaptitude, like in other type of documents.

All sort of suggestions for improving the documentation are always welcome. Also, reports of useless or misleading error messages, as well as suggestion on improving them, are solicited.

One or more tutorial Youtube videos are planned, explaining the program and its principles more informally.

2.5.2.4 Acknowledgement

I would like to acknowledge the influence of the JP1 forum, both the programs (in particular of course DecodeIR, and the discussions (in particular with Dave Reed ("3FG") and Graham Dixon ("mathdon")). This work surely would not exist without the JP1 forum.

2.5.2.5 Copyright and License

2.5.3 Main principles

2.5.3.1 Design principles

It is easier and more logical to put a GUI on top of a sane API, then to try to extract API functionality from a program that was never designed for this but built around and into the GUI. The goal of *this* project is an API and a command line program.

Performance considerations

Performance consideration, both time and space, were given secondary priorities. Also, the decoding mechanism is intrinsically much slower that DecodeIR. A single decode can take several hundred milli-seconds. However, in normal interactive use from the command line or through IrScrutinizer, this does not introduced any noticeable delays.

IR signal formats

The current program reads raw format, and Pronto Hex format IR signals. These formats can also be output. Although many more formats exist, it is not planned to extend this. The "conversion expert" is IrScrutinizer (which is even symbolized by its icon, the Babel fish).

Data types

Everything that is a physical quantity (durations and frequency), are real numbers (in the Java code `double`),

Durations are always given in micro seconds. Unless in a context where the IRP says otherwise, all others quantities are given in (pure) SI units without prefix. So are duty cycle and relative tolerance both a real number between 0 and 1, not a number of percents. Modulation frequency is given in Hz, not it kHz (with the exception of the `GeneralSpec`, since this is defined in the specification).

Integer literals can be given in base 16 (using prefix "0x"), base 8 (using prefix "0"), base 2 (using prefix "0b"), as well as in base 10 (no prefix, omitting leading zeros).

Integer quantities are in principle arbitrarily large, but in most cases limited to 63 bits, since the implementation uses Java's `long`, 64 bits long, amounting to 63 bits plus sign. (Work is ongoing to remove this restriction, by, alternatively using Java's `BigInteger`.)

Internationalization

Being a command line program and API library, this project is not a a candidate for internationalization.

2.5.4 Theory and general concepts

2.5.4.1 Main concepts

IrSequence

Sequence of time durations, in general expressed in microseconds, together with a modulation frequency. The even numbered entries normally denote times when the IR light is on (disregarding modulation), called "flashes" or sometimes "marks", the other denote off-periods, "gaps" or "spaces". They always start with a flash, and end with a gap.

> **Note:**
> In some communities (Lirc and Linux, IRremote), the ending gap is not considered to be a part of the IrSequence/IrSignal. This must be taken into consideration if importing signals from these communities.

IrSignal

Consists of three IR sequences, called

1. *start sequence* (or "intro", or "beginning sequence"), sent exactly once at the beginning of the transmission of the IR signal,
2. *repeat sequence*, sent "while the button is held down", i.e. zero or more times during the transmission of the IR signal (although some protocols may require at least one copy to be transmitted),
3. *ending sequence*, sent exactly once at the end of the transmission of the IR signal, "when the button has been released". Only present in a few protocols.

Any of these can be empty, but not both the intro and the repeat. A non-empty ending sequence is only meaningful with a non-empty repeat.

By "sending an IrSignal *n* times" we shall mean sending an IrSequence consisting of one copy of the intro sequence, *n* - 1 copies of the repeat sequence, and one copy of the ending sequence, unless the intro sequence is empty, in which case the IrSequence has *n* repeats, followed by the ending sequence.

Numerical equality between durations

Two durations *d1* and *d2* are considered numerically equal, if *either* the difference is absolute less or equal than `absolute-tolerance` *or* the difference divided by the largest is less than or equal than `relative-tolerance`. (see `IrCoreUtils.approximatelyEquals`).

Parsing of IR signals as text

We need a new simple text format for raw IR signals:

Bracketed raw format

Simple text format for formatting of a raw IR signal, introduced with the current program. Intro-, repeat, and (optionally) ending sequences are enclosed by brackets ("[]", sometimes called "square brackets" or "square parenthesis"). Signs ("+" for flashes, "-" for gaps), as well as interleaved commands or semicolons, are accepted, but ignored (even if they are not interleaved). Optionally, the beginning sequence can be preceeded by `Freq=`*frequency_in_Hz*`Hz` giving the modulation frequency. If not explicitly given, a default value (dependent on the reading program, but in general 38000Hz) is assumed.

Example (an RC5-signal):

```
Freq=36000Hz[][+889,-889,+1778,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,
+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-90886][]
```

Although we only consider the Pronto Hex and the raw form, parsing of text representation is a non-trivial task. IrpTransmogrifier presently contains three different parsers, tried in order until one succeeds. These are classes implementing the interface `IrSignalParser`, namely:

ProntoParser
Tries to interpret its argument as Pronto Hex, possibly after concatenating all lines of its argument.

BracketedIrSignalParser
Tries to parse an IrSignal in the bracketed text format, as defined above.

MultilineIrSignalParser
If the number of lines is 1, 2, or 3, these are considered intro-, repeat-, and ending sequence respectively. If there are more lines, these are concatenated, and considered an intro-sequence, without repeat-, and ending sequence.

Programs can add other formats and their corresponding parsers using the API. For an example, in IrScrutinizer, a parser for the Global Caché sendir format is added, see `org.harctoolbox.irscrutinizer.Interpretstring.interpretString()`.

Cleaner

Physically measured IR signals/sequences are in general "dirty", consisting of a number of close, but not quite equal, measured durations. To be useful for further processing, the signal/sequence needs to be cleaned. The naive way to implementing such a cleaning algorithm is to round off all durations to a multiple of an (more ore less arbitrarily selected) time unit. The cleaner in IrpTransmogrifier takes a more sophisticatedly approach: The collected durations found in the sequence(s) are bundled into "bins" (disjoint intervals), according to this closeness concept. Every duration belonging to a bin is "close" (determined by those parameters) to the bin middle. All the durations within the bin are then replaced by the average of its members. It is thus not guaranteed that the distance between a duration and its replacement will be consistent with `absolute-tolerance` and `relative-tolerance`.

If invoked on a number of IR signals/sequences, the histogram is formed over all sequences, and the corresponding "cleaning" then applied to the individual signals/sequences.

The cleaner is not a separate function in the command line program, but can be invoked together with the `decode` and `analyze` commands.

Repeat finder

A repeat finder is a program that, when fed with an IrSequence, tries to identify a repeating subsequence in it, returning an IrSignal containing intro-, repeat-, and ending sequence, compatible with the given input data. IrpTransmogrifier can optionally run its data for the decode and analyze command though its built-in repeat finder. The repeat finder is configurable using the following parameters:

`absolute.tolerance`
See Numerical equality between durations. Current default: 100.0.

`relative-tolerance`
See Numerical equality between durations. Current default: 0.3.

`minrepeatgap`
To be recognized as a repeat sequence, the final gap must be at least this long. Default: 5000.0.

> **Note:**
> In some cases, the repeat finder is doing a "too good" job and may squash a signal so that the decoder does not produce the expected result. For example, a `JVC` signal is reduced beyond the `JVC` IRP. (To handle this case, the relaxed protocol `JVC_squashed` was entered into the IRP data base.)

The Configuration file/IRP protocol database

The configuration file `IrpProtocols.ini` of IrpMaster has been replaced by an XML file, called `IrpProtocols.xml`. The XML format is defined by a W3C schema, named irp-protocols, having the XML name space `http://www.harctoolbox.org/irp-protocols`. This format has many advantages over the previous, simpler, format, as it gives access to different XML technologies, for example for formatting and transforming. It can contain embedded (X)HTLM fragments, useful for for writing documentation fragments. The file is to be thought of as a data base of protocols and their properties and parameters, not as a configuration file for the present program. It can contain different parameters that can be used by different programs, for example, tolerance parameters for decoding. For this, arbitrary string-valued parameters are permitted. It is up to an interpreting program to determine the semantic. Within IrpTransmogrifier, this is used for providing protocol-specific values to the parameters.

If setting the attribute `type="xml"`, the element's content is considered an XML document fragment, i.e., arbitrary (well-formed) xml content can be present.

There is also an XSLT stylesheet `IrpProtocols2html.xsl`, which translates this file to readable HTML, for reading in a HTML browser. This makes it possible to browse the said file in an XSLT-enabled browser just by opening it. Unfortunately, this very practical mechanism is since recently considered a security problem. Firefox (and possibly other browsers) therefore unfortunately disables it for local files (URIs using `file:` scheme).)

More than one protocol data base file ("`IrpProtocols.xml`") can be deployed. This gives a user or a program the possibility to strictly divide his/her/its own entries from the official ones. A "small" file modifying and/or adding a few protocols is often called a "patch file", although there is actually no difference between the first and subsequent files, and the semantics is identical. All files are required to be valid XML, and should be valid with respect to the given XML schema.

From the command line, the option `--configfile` can be given several times, or several files can be given as argument, separated by commas (","). From the API, see the function `IrpDatabase.patch(File)` (and others).

The content of the patch file is basically merged into the data base, amending the information already there, with the exception that an empty entry deletes the original one. The exact rules are as follows:

- If a patch file contains an empty protocol element, the protocol with the same name will be (if present) removed from the data base. Otherwise, its content is amended into the present one.
- For the protocol properties (both the XML properties and the normal ones) similar rules apply: An empty property element removes that property from the protocol. A non-empty property in a protocol in the patch file is added to the end of the list that is the value of that property in the protocol (unless it is already present).
- However, the "properties" irp and documentation are different, since there can only be one of those in a protocol. For these, an entry in the patch file overwrites the original entry.

Syntax and semantics of the file is believed to be essentially self explaing. The exact syntax is given by the schema, and is therefore not repeated here.

Note that the program contains some functions for the maintenance of the data base.

Protocol parameters

Most protocol parameters are given as the `parameter` element. However, the protocol name, its IRP, and its documentation are handled differently:

`name`
The name of the protocol. It is specified in the mandatory attribute `name` in the `protocol` element. The name is folded to lowercase for searches and comparisons, which are therefore done case insensitively.

`irp`
The IRP form of the protocol, as text.

`documentation`
Also a separate, but optional, element. Any textual information can be put here. Arbitrary (valid) (X)HTML code can be included here, for example formatting instructions, tables, or links. A processing program may select to render the HTML content, pass it through, or just to ignore it. (For example the current version of the command line program will, using the command option `list --documentation` produce an "dumb" version, while `list --html` will give the original text, enclosed in a HTML `div` element.

The following parameters may be given for any protocol in the data base. They override the global values for the current protocol.

`absolute-tolerance`
See Numerical equality between durations. Current default: 100.0.
`relative-tolerance`
See Numerical equality between durations. Current default: 0.3.
`frequency-tolerance`
Tolerance in Hz for frequency comparisons. Set to -1 to disable frequency check. Current default: 2000.0.
`minimum-leadout`
Minimal duration in micro seconds that is accepted as final duration. Current default: 20000.0.
`prefer-over`
If a signal has multiple decodes, the present protocol is preferred over the one mentioned as `prefer-over`. May be given several times. Alternatively, the content of the element may have two parts: a predicate and a protocol name, separated by a semi colon ("`;`"). The predicate is evaluated for the current, to be preferred, protocol and its parameters. Only if the predicate (which is an `expression` in the sense of IRP) evaluates to non-zero, the preferance takes effect. (See the protocol `Pioneer` for an example.) (Normally, a special protol should be preferred over a more general one.)
`alt_name`
Alternative name, ("alias", "synonym") for the present protocol.
`reject-repeatless`
When decoding, normally the repeat sequence may match 0 times, i.e. not at all. If this parameter is `true` at least one repeat must be present for a match to be recognized. (Strictly speaking, this would have been possible to achieve by using "+" instead of "*" for the repeat indicator, but this would have other disadvantages.)
`decodable`
Setting this to `false` prohibits the program from trying to use this protocol for decoding. Normally, this is only used for portocols that are so involved that protocol decoding is impossible or not feasible.
`decode-only`

Setting this to `true` makes the program refuse to render the protocol. Should be used only for "protocols" that denote incomplete or otherwise flawed decodes ("relaxed" protocols), for example with missing parts or non-matching checksums.

Other parameters are allowed, but ignored by IrpTransmogrifier. They may be used by another program. Also, new parameters may be introduced in the future.

Some more commands and hints are given as comments in the file itself.

2.5.5 Use cases

In this section, we will discuss the different use cases from a high-level, theoretical standpoint, without delving into the usage of the program. The subsequent section Subcommands, which to a certain degree mirrors the present, will cover the usage of the program.

2.5.5.1 Rendering

Given a protocol name, present in the protocol data base (alternatively, an IRP protocol given explicitly with the `--irp` option), and a set of valid parameter values, an IrSignal is computed. This use case corresponds to IrpMaster (or MakeHex).

In earlier versions of IrScrutinizer, the word "generate" was used instead of "render". These words can be considered as synonyms.

2.5.5.2 Decoding

General

This use case corresponds to DecodeIR: given a numerical IR signal/sequence, find one (or more) parameter/protocol combination(s) that could have generated the given signal. This is implemented by trying to parse the given signal with respect to a number of candidate protocols, per default all. It is thus very robust, systematic, and customizable, but comparatively slow.

While every conforming IRP form with only one repeat also can be usable for rendering, the same is unfortunately not true for decoding. A few protocols cannot be used for decoding. Non-recognizable protocols are marked by setting the `decodable` parameter to `false`. To be useful for decoding, the IRP protocol should adhere to some additional rules:

- Non-deterministic grammars (like, for example, "A? A*") must be avoided.
- The "+" form of repetitions is discouraged in favor of the "*" form.
- The width and shift of a Bitfield must be constant (i.e. not dependent on protocol parameter).
- The decoder is capable of *simple* equation solving (e.g. `Arctech`), but not of complicated equation solving (e.g. `Fujitsu_Aircon_old`).

Presently all but the protocols `zenith`, `nec1-shirrif`, (non-constant bitfield width); `RTI_Relay_alt`, `fujitsu_aircon_old` (would require non-trivial equation solving)) are decodable. (Note how the non-decodable protocols `RTI_Relay_alt` and `fujitsu_aircond_old` was made decodable (`RTI_Relay` and `fujitsu_aircon`) by a changed parameterization.)

Multiple decodes

In many cases, a signal produces more than one decode. This is not necessarily an error nor a deficiency of the decoder. However, many protocols are effectively a special case of another protocol, for example, an signal that decodes to the Apple protocol by mathematical necessity is also a valid NEC1-f16 signal. It thus makes sense for the decoder to have an Apple decode to "override" a NEC1-f16 decode. This is specified by the `prefer-over` parameter. With this mechanism, the preferences are under control of the configuration file, not hard soldered into the program code as in DecodeIR.

Loose matches, "Guessing"

Many captured signals do not quite match the protocol they are supposed to match. However, in order to give the user of a device maximal comfort, the firmware in a receiving device is in general quite "forgiving", and accepts slightly flawed signals. The degree of "forgiveness" should be balanced against the possibility of "false positives": that the device erroneously considers "noise" of some form (often commands for a different device) as one of its commands. It is thus desirable for a program of this type to find a near match, "guess", when an proper match fails.

Generally speaking, it is my principle to "make everything as simple as possible, *but not simpler*". Sometimes a signal has several decodes, or none at all. Sometimes only a part of the data is used for the match. In these cases, the program always prefer to tell the truth (and the full truth) to the user, instead of, in the name of user friendlyness, to perform questionable simplifications. Unfortunately, a badly informed user tend to prefer a program delivering exactly one answer to every question over a program presenting the full truth.

There are a few different issues:

Matching of durations, non-leadouts

Non-ending durations are matched according to the numeric equality criterion, using the parameters `absolutetolerance` and `relativetolerance`.

Matching of ending durations

The ending duration has a slightly different role. When capturing, the ending duration is the time you have verified that nothing more is coming, not a real measurement. (Some "communities", like Lirc and IrRemote, do not consider an ending duration at all.) For this reason, it does not make sense to check the ending duration the same way as the

other, non-ending durations. There is instead a parameter `minimum-leadout` (possibly protocol dependent); the ending duration is considered as passed if it is longer or equal to `minimum-leadout`.

Matching of modulation frequency

There is a parameter `frequencytolerance`, which defaults to 2000. The frequency test is considered passed if the absolute difference between measured and expected frequency is less or equal to `frequencytolerance`. Setting `frequencytolerance` negative disables the frequency test, i.e., all values pass the test.

If an asymmetric interval is needed, instead the parameters `frequency-lower` and `frequency-upper` can instead be used, specifying the lowest and highest frequency that is to be accepted.

All of these parameters are to be given in Hz, not kHz.

It turns out that decoding of IrSignals and IrSequences are two fairly different use cases:

Matching of IR Signals (Intro-, Repeat and Ending sequence)

The task is to match an IR signal (with intro-, repeat- and ending sequence) to a protocol, turn out, "practically", to be more involved than just matching the different sequence to each other. The to-be-matched signal is often empirically measured, and its decomposition into sequences the result of entering a measured signal into a repeat finder. For a short measured sequence, the repeat part may not be identified as such. For a decoding program to be considered practically usable, this problem, and related problems, must be addressed and handled correctly.

For "strict" matching of a given IR signal, the intro-, repeat-, and ending sequences are required to match their theoretical counterparts (within numerical tolerances). If this fails, it may be sensible to convert the signal to an IrSequence (normally by concatenating the intro, repeat and ending sequences), and try to decode as IrSequence, as described in the next section.

A strict decode result of an IrSignal is a number (ideally exactly one) of decodes, each one containing a protocol and a set of corresponding parameters. (Technically, the function `Decoder.decodeIrSignal(IrSignal)` returns a `Decoder.SimpleDecodeSet`, being an `Iterable<Decoder.Decode>`).

Matching of IR Sequences

A completely faithful decoding an IR Sequence is theoretically a more complicated undertaking. Starting at position 0, in general several decodes can start there — although of course "normally" only one. Such a decode matches an intro sequence, zero or more repeat sequences, and the ending sequence. It thus matches a certain number of durations, less than or equal to the duration of the input IrSequence. In the case that the decode is shorter than the input signal, the

process repeats with the IrSequence that is remaining, leading to an exponential growth of decodes — at least in the general case. (Technically, the function `Decoder.decode(ModulatedIrSequence)` returns a `Decoder.DecodeTree`, which is an `Iterable<Decoder.TrunkDecodeTree>`, where each `Decoder.TrunkDecodeTree` consists of one decode (with a variable number of repeat-matches) ("trunk") followed by another `Decoder.DecodeTree`.)

2.5.5.3 Code generation for rendering and/or decoding

The task is: For a particular protocol and a particular target (C, C++, Java, Python,...), generate target code that can render or decode signals with the selected protocol. As opposed to the previous use cases, efficency (memory, execution time) (for the generated code) is potentially an issue.

Two mechanisms are available, XML and StringTemplate, described in the following two sections.

Code generation with XML

The program generates basically an XML version of `IrpProtocols.xml` (with the IRP protocol replaced by a much more "parsed" XML version). It is the task of the user to supply am XML transformation, for example using XSLT, that transfers the XML file to the expected target format. The program does not come with an XSLT engine, so this has to be invoked independently on the XML export. It is recommended to use XSLT version 2 for writing the transformation.

This is presently used for generating the Lirc export format (which is basically another XSLT transformation) of IrScrutinizer.

Code generation using StringTemplate

For code generation, the template engine StringTemplate can also be used. As opposed to the XML case, the program contains the transformation engine.

Target dependent code is not considered a part of this project, but is found in a separate project.

2.5.5.4 General code analysis

This use case is not really connected to parsing IRP, but fits in the general framework. This has been inspired by to the Analyzer and the RepeatFinder in Graham Dixon's ExchangeIR.

Theoretically speaking, this is an Inverse problem. Given one or many IR signals/sequences, the problem is to find out what could have generated it, in the form of an IRP protocol. This problem in general does not have a unique solution; instead the "simplest" one is selected out of the possible solutions.

In a way, this is a more abstracted version of the decoding problem. There are around 10 different "template IRP" (for example, different forms of bi-phase and PWM modulation) that are tried. For every of those templates, a quantity, "weight" is computed, quantifying (in a somewhat arbitrary manner) how "complicated" the answer is. The template that produces the "simplest" answer, i.e. the one with the least weight, is selected.

The form of the final answer can be influenced by a number of different parameters, use the command `irptransmogrifier analyze --help` to list them.

2.5.5.5 IRP database maintenance

The command line program also contains some functions for maintenance of the IRP database, described next.

Validation

An IRP database is required to be valid with respect to the a W3C schema. For this, the common option `--validate` makes the program's XML parser read all files in validating mode, and stops if the input is not conforming.

Output of parsed database

The parsed database (possibly after merging of several files) can be output using the `list --dump` command. This option generate an XML file on the output. All other output is suppressed. This output can be, possibly after minimal hand editing (update version?) used as new `IrpProtocols.xml`.

The option `--xml` does the same as `--dump`, except that the XML comments in the original files are suppressed.

Other options

With the command `list --prefer-overs`, the protocol's `prefer-over`s are printed, transitively. The option `--check-sorted` checks the correct (alphabetic) sorting of the commands (with respect to their names). There are also a number of other options listing properties of the individual protocols, for example `--classify, --display, --warning, --weight`.

2.5.6 Extensions to, and deviation from, IRP semantic and syntax

This implementation of the IRP has a number of extensions, and a few deviations to the current specification version 2. These will be described in detail next.

For the complete syntax, see the ANTLR grammar.

2.5.6.1 Data types

While the original specification uses exclusively unsigned integers, here numbers that are intrinsically "physical" (modulation frequency, durations, duty cycle) are floating numbers, in the code `double`.

Integer numbers are in general implemented with Java's `long`, effectively limiting the number of bits to 63. In the future, it is planned to remove this restriction, using Java's BigInteger.

2.5.6.2 Repetitions

Possibly the major difficulty in turning the IRP Specification into programming code was how to make sense of its repetition concept. Most formalisms on IR signals (for example the Pronto format) considers an IR signal as an introduction sequence (corresponding to pressing a button on a remote control once), followed by a repeating signal, corresponding to holding down a repeating button. Any, but not both of these may be empty. In a few relatively rare cases, there is also an ending sequence, send after a repeating button has been released. Probably 99% of all IR signals fit into the intro/repetition scheme, allowing ending sequence in addition should leave very few practically used IR signals left. In "abstract" IRP notation, these are of the form A,(B)*,C with A, B, and C being (possibly empty) `bare irstream`s. Not all three may be empty; if B is empty, then C must also be empty.

In contrast, the IRP notation reminds of they syntax and semantics of regular expressions: There may be any numbers of (infinite) repeats, and they can even be hierarchical (repetitions within repetitions). There does not appear to be a consensus on how this extremely general notation should be practically thought of as a generator of IR signals.

The predecessor program IrpMaster tried to be very smart here, by trying to implement all aspects, with the exception of hierarchical repetitions (repetitions within repetitions). This never turned out to be useful. The present program takes a simpler approach, by simply prohibiting multiple (infinite) repetitions.

2.5.6.3 Parameter Specifications

In the first, now obsolete, version 1 of the IRP notation the parameters of a protocol had to be declared with the allowed max- and min-value. This is not present in the current specification version. I have re-introduced this, using the name `parameter_spec`. For example, the well known NEC1 protocol, the Parameter Spec reads: `[D:0..255,S:0..255=255-D,F:0..255]`. (D, S, and F have the semantics of device, sub-device, and function or command number.) This defines the three variables D, S, and F, having the allowed domain the integers between 0 and 255 inclusive. D and F must be given, however, S has a default value that is used if the user does not supply a value. The software enforces that all values without default values are supplied, and within the stated limits. If, and only if, the parameter specs is incomplete, there may occur run-time errors concerning not assigned values. It is the duty of the IRP

author to ensure that all variables that are referenced within the main body of the IRP are defined either within the parameter specs, defined with "definitions" (Chapter 10 of the specification), or assigned in assignments before usage, otherwise a run-time error will occur (in the code, a `NameUnassignedException` will be thrown).

The preferred ordering of the parameters is: D, S (if present), F, T (if present), then the rest in alphabetical order,

The formal syntax is as follows, where the semantic of the '@' will be explained in a following section:

```
parameter_specs:
    '[' parameter_spec (',' parameter_spec )* ']'
    | '['  ']'

parameter_spec:
      name     ':' number '..' number ('=' expression)?
    | name '@' ':' number '..' number  '=' expression
```

2.5.6.4 GeneralSpec

For the implementation, I allow the four parts (three in the original specification) to be given in any order, if at all, but I do not disallow multiple occurrences — it is quite hard to implement cleanly and simply not worth it. (For example, ANTLR does not implement exclusions. The only language/grammar I know with that property is SGML, which is probably one of the reasons why it was considered so difficult (in comparison to XML) to write a complete parser.)

The default frequency is 38kHz, not 0kHz as in the specification. For the cases of modulation frequency not "really" known, but "sufficiently close" to 38kHz, it is recommended to rely on the default, not to state "38k" explicitly,

GeneralSpecs, duty cycle

Without any very good use case, I allow a duty cycle in percent to be given within the `GeneralSpec`, for example as `{37k,123,msb,33%}`. It is currently not used for anything, but preserved through the processing and can be retrieved using API-functions. If some, possibly future, hardware needs it, it is there.

2.5.6.5 Persistency of variables

In the specification and in forum contributions, all variables in a IRP description appear to be consider as intrinsically persistent: They do not need explicit initialization, if they are not, they are initialized to an undefined, random value. This may be a reasonable model for a particular physical remote control, however, from a scientific standpoint it is less attractive. In the current work, there is a way of denoting a variable, typically a toggle of some sort, as persistent by appending an "@" to its name in the parameter specs. An initial value (with syntax as default value in the parameter spec) is here mandatory.

For example, a toggle is typically declared as `[T@:0..1=0]` in the parameter specs. It is set to its initial value the first time the protocol is used. Rendering such a protocol typically updates the value, as given in an assignment, a 0-1 toggle goes like T=1-T). As opposed to variables that has not been declared as persistent, it retains its value between the invocations.

An instance of the `Protocol` class keeps the corresponding protocols's persistence variables values between invocations, unless explicitly changed by parameter assignments. In the command line program, this makes no sense, however.

2.5.6.6 Comments

Comments in the C syntax (starting with `/*` and ended by `*/`) are allowed, and ignored. Also, C++-style comments ("//", extending to (and including) the end of line) are accepted. (In the specifications, embedded comments in the IRP are not present.)

The function `list --irp` lists the IRP as given in the data base (preserving comments and whitespace), while `list --parsedirp` list the parsed version. The latter has comments and whitespace removed, and observes the `--radix` argument.

2.5.6.7 Whitespace

All white space, including line breaks, are ignored when parsing. (In the original spec, the IRP form had to be on one line.)

The function `list --irp` lists the IRP as given in the data base (preserving comments and whitespace), while `list --parsedirp` list the parsed version (with comments and whitespace removed).

2.5.6.8 Extents

The specification writes ``*An extent has a scope which consists of a consecutive range of items that immediately precede the extent in the order of transmission in the signal. ... The precise scope of an extent has to be defined in the context in which it is used.*", and, to my best knowledge, nothing more. I consider it as specification hole. I have effectively implemented this interpretation: “An extent has a scope which consists of a consecutive range of all non-extent items that immediately precede the extent in the order of transmission in the signal, starting with the first element after the last preceding extent, or from the start if there is no preceding extent.” Differently put: Every extent encountered resets the duration count.

2.5.6.9 Multiple definitions allowed

It turned out that the preprocessing/inheritance concept necessitated allowing several definition objects. These are simply evaluated in the order they are encountered, possibly overwriting previous content.

2.5.6.10 Names

The IRP documentation defines a name as starting with an uppercase letter, followed by an arbitrary number of uppercase letters and digits. I have, admittedly somewhat arbitrarily, extended it to the C-name syntax: Letters (both upper and lower cases) and digits allowed, starting with letter. Underscore "_" counts as letter. Case is significant.

2.5.6.11 Name spaces

There is a difference in between the IRP documentation and the implementation of the Makehex program, in that the former has one name space for both *assignments* and *definitions*, while the latter has two different name spaces. IrpTransmogrifier (just as the precessor IrpMaster) has one name space, as in the documentation. (This is implemented with the `NameEngine` class.)

2.5.6.12 Expressions

Several extensions to the expressions have been made. Note that, informally speaking, an expression is an integer, that, in different contexts is differently interpreted: as integer value, (potentially infinite) bit pattern (using 2-complement representation), and logical (0 is `false`, everything else is `true`).

For a summary of the complete syntax of `expression`, see the grammar.

Terminology

In this project, we use the terms *para_expression* (denoting an *expression* enclosed within parentheses) and *expression* instead of the specification's *expression* and *bare_expression*, since it was felt that the latter was wieldy and incompatible with normal-day usage.

Literals

Numerical literals can be given not only on base 10 (as in the specification), but also in bases 2 (with prefix `0b`), base 8 (prefix `0`), as well as base 16 (prefix `0x`).

A few pre-defined literals are introduced for convenience and readability. These are:

- `UINT8_MAX` = 2^8 - 1 = 0xFF = 255,
- `UINT16_MAX` = 2^16 - 1 = 0xFFFF = 65535,
- `UINT24_MAX` = 2^24 - 1 = 0xFFFFFF = 16777215,
- `UINT32_MAX` = 2^32 - 1 = 0xFFFFFFFF = 4294967295, and
- `UINT64_MAX` = 2^64 - 1 = 0xFFFFFFFFFFFFFFFF = 18446744073709551615.

Unary operators

In addition to the specification's unary minus ("-"), some additional unary operators have been implemented, described next.

Logical NOT, "!"

The exclamation point, logical not, acts like in C: it turns everything that evaluates to 0 (zero, `false`) into 1 (`true`), everything else to 0 (`false`).

Bit inversion, "~"

This operator turns all 0 to 1 and all 1 to 0 in the binary representation.

BitCount Function "#"

IrpMaster introduced the BitCount function as a unitary operator, denoted by "#". This is useful in many situations, for example, odd parity of `F` can be expressed as `#F%2`, even parity as `1-#F%2`. It is implemented through the Java Long.bitCount-function.

Binary operators

There are also some added binary operators, which will be described next.

The specification contains the *bitwise* operators "&", "|", and "^", having the syntax and semantics of C. In addition, the current implementation adds the following two *logical* operators.

Logical AND, "&&"

The logical operator && (with short-circuiting as in C and Perl) works as follows: To evaluate the expression `A && B`, first `A` is checked for being 0 or not. If 0, then A (which happens to be 0) (`false`) is returned, *without evaluating* `B`. If however, `A` is nonzero, `B` is evaluated, and the resulting value is returned.

Logical OR, "||"

The logical operator || (with short-circuiting as in C and Perl) works as follows: To evaluate expression `A || B`, first `A` is checked for being 0 or not. If non-zero, then A is returned, *without evaluating* `B`. If however, `A` is 0, `B` is evaluated, and the resulting value is returned.

Left shift "<<"

The left shift operators "<<" is implemented, with syntax and semantics as in the C and Java programming languages. (See this discussion.)

Right shift ">>"

The (arithmetic) right shift operator ">>" with syntax and semantics as ">>" (*not* ">>>", denoting logical shift) in the Java programming language. (Differently put, it preserves the leading bit, not shifting in 0.)

Numerical comparison operators, "<", "<=", ">", ">=", "==", "!="

The comparison operators have been added. They have the same syntax and semantics as in C, taking two numerical operators to 0 (false) or 1 (true).

Ternary operator

Conditional operator ?:

Similarly, the ternary operator `A ? B : C`, returning B if A is true (non-zero), otherwise C, has been implemented. As opposed to other operators (with the exception of exponentiation "**"), it is right associative.

2.5.6.13 Preprocessing and inheritance

Reading through the protocols, the reader is struck by the observation that there are a few general abstract "families", and many concrete protocol are "special cases". For example all the variants of the NEC* protocols, the Kaseikyo-protocols, or the rc6-families. Would it not be elegant, theoretically as well as practically, to be able to express this, for example as a kind of inheritance, or sub-classing?

For a problem like this, it is easily suggested to invoke a general purpose macro preprocessor, like the C preprocessor or m4. I have successfully resisted that temptation, and am instead offering the following solution: If the IRP notation does not start with "{" (as they all have to do to confirm with the specification), the string up until the first "{" is taken as an "ancestor protocol", that has hopefully been defined at some other place in the configuration file. Its name is replaced by its IRP string, with a possible parameter spec removed — parameter specs are not sensible to inherit. The process is then repeated up until, currently, 5 times.

The preprocessing takes place in the class `IrpDatabase`, in its role as data base manager for IRP protocols.

Example

This shows excepts from an example configuration file. Let us define the "abstract" protocol `metanec` by

```
                        [protocol]
                        name=metanec
                        irp={38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)
[A:0..UINT32_MAX]
```

having an unspecified 32 bit payload, to be subdivided by its "inherited protocols". Now we can define, for example, the NEC1 protocol as

```
                        [protocol]
                        name=NEC1
```

```
                        irp=metanec{A = D | 2**8*S | 2**16*F | 2**24*(~F:8)}
[D:0..255,S:0..255=255-D,F:0..255]
```

As can be seen, this definition does nothing else than to stuff the unstructured payload with D, S, and F, and to supply a corresponding parameter spec. The IrpMaster class replaces "`metanec`" by `{38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)`" (note that the parameter spec was stripped), resulting in an IRP string corresponding to the familiar NEC1 protocol. Also, the "Apple protocol" can now be formulated as

```
                        [protocol]
                        name=Apple
                        irp=metanec{A=D | 2**8*S | 2**16*C:1 | 2**17*F | 2**24*PairID}
 \
                        {C=1-(#F+#PairID)%2,S=135} \
                        [D:0..255=238,F:0..127,PairID:0..255]
```

The design is not cast in iron, and I am open to suggestions for improvements. For example, it seems reasonable that protocols that only differ in carrier frequency should be possible to express in a concise manner.

2.5.7 Installation

2.5.7.1 Installation of binaries

The most convenient way to install the program is to install IrScrutinizer, version 2.0.0 or later. For the Windows and the generic binary distribution, this will install a wrapper for IrpTransmogrifier too. The AppImage will start IrpTransmogrifier instead of IrScrutinizer, if called with the last component of the name equals to `irptransmogrifier` (for example, through a link ending with that name). (However, the MacOS installation presently does not support command line IrpTransmogrifier.)

Also RemoteMaster comes with IrpTransmogrifier and a wrapper (`irptransmogrifier.sh` or `irptransmogrifier.bat`) to start it as command line program.

IrpTransmogrifier can of course be installed separately. The latest released version can be found here. The program is basically just an executable jar-file. There is no "installer". Instead, unpack the binary distribution in a preferably empty directory. Start the program by invoking the wrapper (`irptransmobrifier.bat` on Windows, `irptransmogrifier.sh` on Unix-like systems like Linux and MacOS.) from the command line. Modify and/or relocate the wrapper(s) if desired or necessary. Do not double click the wrappers, since this program runs only from the command line. (Do not use the wrapper `irptransmogrifier` in the top directory of the source tree: it is intended only for development in the source tree; and is not intended for deployment.)

The program runs can be installed in a read-only location.

The Macintosh app for IrScrutinizer presently does not come with support for running IrpTransmogrifier as command line program.

The program presently requires Java 8 JRE or later to run. Some distributions of IrScrutinizer come with their own Java installations, that can run IrpTransmogrifier.

2.5.7.2 Building from sources

On Github, the latest official source- and binary distribution is found. Also, the built development version can be found.

The project uses Maven as its build system. Any modern IDE should be able to open/import and build it as Maven project. Of course, Maven can also be run from the command line, like

```
mvn install
```

Third-party Java dependencies (jars)

The program depends on ANTLR4, Stringtemplate, as well as the command line decoder JCommander. When using Maven for building, these are automatically downloaded and installed to a local repository.

2.5.8 Usage of the program from the command line

Next it will be described how to invoke the program from the command line. The reader is assumed to possess an elementary command of the command line usage.

The usage message from `irptransmogrifier help --short` gives an extremely brief summary:

```
Usage: IrpTransmogrifier [options] <command> [command_options]
Commands:
   analyze         Analyze signal: tries to find an IRP form with parameters
   bitfield        Evaluate bitfield given as argument.
   code            Generate code for the given target(s)
   decode          Decode IR signal given as argument
   expression      Evaluate expression given as argument.
   help            Describe the syntax of program and commands.
   lirc            Convert Lirc configuration files to IRP form.
   list            List protocols and their properites
   render          Render signal from parameters
   version         Report version

Use
    "IrpTransmogrifier help" for the full syntax,
    "IrpTransmogrifier help <command>" for a particular command.
    "IrpTransmogrifier <command> --describe" for a description,
    "IrpTransmogrifier help --common" for the common options.
    "IrpTransmogrifier help --logging" for the logging related options.
```

Using from the command line, this is a program with sub commands. Before the sub command, common options can be given. After the command, command-specific options can be specified. Commands and option names can be abbreviated, as long as the abbreviation is unique. They are matched case sensitively, and can be abbreviated as long as the abbreviation is unambiguous.

All options have a long form, starting with two dashes, like `--sort`. (In rare cases, there might be more than one long name.) They can be abbreviated as long as the abbreviation remains unique. Most options also have a short, one letter form, starting with a single dash (like `-s`). For brevity, this document will not mention the short form. This information, if required, can instead be easily found using the `help` command.

Note that `help` and `version` are commands, not options, as in most other command line programs. (For compatibility reasons, also the options form works.)

The commands are briefly described next. Since the program contains its own documentation facility, the description is not aimed at being complete, but more to to comment upon the general idea behind.

2.5.8.1 analyze

The `analyze` command takes as input one or several sequences or signals, and computes an IRP form that corresponds to the given input (within the specified tolerances). The input can be given either as Pronto Hex or in raw form, optionally with signs (ignored). Several raw format input sequences can be given by enclosing the individual sequences in brackets ("[]"). However, if using the `--intro-repeat-ending` option, the sequences are instead interpreted as intro-, repeat-, and (optionally) ending sequences of an IR signal.

For raw sequences, an explicit modulation frequency can be given with the `--frequency` option. Otherwise the default frequency, 38000Hz, will be assumed.

Using the option `--input`, instead the content of a file can be taken as input, containing sequences to be analyzed, one per line, blank lines ignored. Using the option `--namedinput`, the sequences may have names, immediately preceeding the signal.

Input sequences can be pre-processed using the options `--chop`, `--clean`, and `--repeatfinder`.

The input sequence(s) are matched using different "decoders". Normally the "best" decoder match is output. With the `--all` option, all decoder matches are output. Using the `--decode` option, the used decoders can be further limited.
The presently available decoders are: `TrivialDecoder, Pwm2Decoder, Pwm4Decoder, Pwm4AltDecoder, XmpDecoder, BiphaseDecoder, BiphaseInvertDecoder, BiphaseWithStartbitDecoder, BiphaseWithStartbitInvertDecoder, BiphaseWithDoubleToggleDecoder, SerialDecoder`.

The options `--statistics` and `--dump-repeatfinder` (the latter forces the repeatfinder to be invoked) can be used to print extra information. The common options `--absolutetolerance, --relativetolerance, --minrepeatgap` determine how the repeat finder breaks the input data. The options `--extent, --invert, --lsb, --maxmicroseconds, --maxparameterwidth, --maxroundingerror, --maxunits, --parameterwidths, --radix`, and `--timebase` determine how the computed IRP is displayed.

Using the `--girr` option, a Girr file can be produced. This embeds the generated IRP protocol in the file.

2.5.8.2 bitfield

The `bitfield` command computes the value and the binary form corresponding to the bitfield given as input. Using the `--nameengine` argument, the bitfield can also refer to names.

As an alternatively, the `expression` command sometimes may be used. However, a bitfield has a length, which an expression, evaluating to an integer value, does not.

2.5.8.3 code

Used for generating code for different targets.

2.5.8.4 decode

The `decode` command takes as input one or several sequences or signals, and output one or many protocol/parameter combinations that corresponds to the given input (within the specified tolerances). The input can be given either as Pronto Hex or in raw form, optionally with signs (ignored). Several raw format input sequences can be given by enclosing the individual sequences in brackets ("[]").

For raw sequences, an explicit modulation frequency can be given with the `--frequency` option. Otherwise the default frequency, 38000Hz, will be assumed.

Using the option `--input`, instead the content of a file can be taken as input, containing sequences to be analyzed, one per line, blank lines ignored. Using the option `--namedinput`, the sequences may have names, immediately preceeding the signal.

In the Harctoolbox world, IR sequences start with a flash (mark) and ends with a non-zero gap (space). In some other "worlds", the last gap is omitted. These signal are in general rejected. The option `--trailinggap duration` adds a dummy duration to the end of each IR sequence lacking a final gap.

Input sequences can be pre-processed using the options `--clean`, and `--repeatfinder`.

The common options `--absolutetolerance`, `--relativetolerance`, `--minrepeatgap` determine how the repeat finder breaks the input data.

Debugging

To debug why a certain signal/sequence does not decode the way expected, the logging facility and the `--protocol` argument (to reduce the logging output) can be useful to pinpoint the decoding process.

2.5.8.5 demodulate

This command demodulates its argument IrSequence, emulating the use of a demodulating IR receiver. This means that all gaps less than or equal to the threshold are squeezed into the preceeding flash. Typically the threshold is taken around the period of the expected modulation frequency.

2.5.8.6 help

This command list the syntax for the command(s) given as argument, default all. Also see the option `--describe` of the individual commands.

2.5.8.7 lirc

This command reads a Lirc configuration, from a file, directory, or an URL, and computes a corresponding IRP form. No attempt is made to clean up, for example by rounding times or finding a largest common divider.

2.5.8.8 list

This command list miscellaneous properties of the protocol(s) given as arguments. There are a large number of options for enabling or suppressing certain kind of output; use the command `irptransmogrifier list --help` for a list.

2.5.8.9 render

This command is used to compute an IR signal from one or more protocols, "render" it. The protocol can be given either by name(s) (or regular expression if using the `--regexp` option), or, using the `--irp` options, given explicitly as an IRP form. The parameters can be either given directly with the `--nameengine` option, or the `--random` option can be used to generate random, but valid parameters. (This is essentially a developer's and tester's option.) With the `--count` or `--number-repeats` option, instead an IR sequence is computed, containing the desired number of repeats.

The syntax of the name engine is as in the IRP specification, for example: `--nameengine {D=12,F=34}`. For convenience, the braces may be left out. Spaces around the equal sign "=" and around the comma "," are allowed, as long as the name engine is still only one argument in the sense of the shell — it may need to be enclosed within single or double quotes.

2.5.8.10 version

Reports version number and license.

2.5.9 Debugging/logging possibilities

The project contains quite powerful logging facilities, based on Java's `java.util.logging` framework. "Logging" is somewhat of a mis-normer for a program that runs only seconds, instead it is a form of debugging. Logging takes place according to the different levels: from highest to lowest `OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL`. Default level is `WARNING`.

The reader is assumed to know the basic principles for logging. Ideally, setting a lower and lower level should reveal more and more details on the inner workings of the program. Unfortunately, this is (at least presently) not always the case. Still, it may be useful for finding out why a particular signal did not decode as expected.

There are command line options, not only for setting the general log level, but also for changing the logging format, the logging file, for generating the log in XML format, and for setting the log level individually for different classes. See `help --logging` for the full list of these options.

2.5.10 The API

A Java programmer can access the functionality through a number of API functions.

The API is documented in standard Javadoc style, which can be installed from the source package, just like any other Java package. For the convenience of the user, the Javadoc API documentation is also available here current, released version only).

The released versions of project is available in the Maven central repository, and can easily be integrated into other Maven projects. For this, include the lines

```
        <dependency>
            <groupId>org.harctoolbox</groupId>
            <artifactId>IrpTransmogrifier</artifactId>
            <version>1.2.10</version>  <!-- or another supported version -->
        </dependency>
```

in the `pom.xml` of the importing project.

2.5.11 Appendix: ANTLR4 Grammar

This appendix shows the grammar file for IRP. It is used to generate the Java code for the IRP parser. It is also (in contrast to the ANTLR3 grammar used in IrpMaster) quite readable for humans.

```
/*
Copyright (C) 2017 Bengt Martensson.
```

```
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program. If not, see http://www.gnu.org/licenses/.
*/

grammar Irp;

// 1.7
// class Protocol
// Extension: * instead of ?, parameter_specs
protocol:
    generalspec bitspec_irstream definitions* parameter_specs? EOF
;

// 2.2, simplified
// Difference: This a simplified version; implementing exclusions is not really
// mainstream... Some silly input is not rejected.
// My semantics: read left-to-right, later entries overwrite.
// class GeneralSpec
generalspec:
    '{' generalspec_list '}'
;

generalspec_list:
    /* Empty */
    | generalspec_item (',' generalspec_item )*
;

// extension: dutycycle_item
generalspec_item:
    frequency_item
    | unit_item
    | order_item
    | dutycycle_item
;

frequency_item:
    number_with_decimals 'k'
;

dutycycle_item:
    number_with_decimals '%'
;

unit_item:
    number_with_decimals ('u' | 'p')?
;

// enum BitDirection
order_item:
    'lsb' | 'msb'
;

// 3.2
// abstract class Duration
// Note: spec did not consider extent as a duration
```

```
duration:
    flash
    | gap
    | extent
;

// class Flash extends Duration
// called flash_duration in spec
flash:
    name_or_number ('m' | 'u' | 'p')?
;

// class Gap extends Duration
// called gap_duration in spec
gap:
    '-' name_or_number ('m' | 'u' | 'p')?
;

// class NameOrNumber
// Extension: Spec allowed number (integers) only
name_or_number:
    name
    | number_with_decimals
;

// 4.2
// class extent (extends Duration)
// Semantics: An extent is a gap, with all preceding durations in the
// containing bare_irstream subtracted. More than one extent in one
// bare_irstream are thus allowed. The "counting" starts anew after each extent.
extent:
    '^' name_or_number ('m' | 'u' | 'p')?
;

//  5.2
// abstact class BitField extends IrpObject
// class FiniteBitField extends BitField
// class InfiniteBitField extends BitField
bitfield:
    '~'? primary_item ':' '-'? primary_item (':' primary_item)? # finite_bitfield
    | '~'? primary_item '::' primary_item                    # infinite_bitfield
;

// abstract class PrimaryItem
primary_item:
    name
    | number
    | para_expression
;

// 6.2
// class IrStream
irstream:
    '(' bare_irstream ')' repeat_marker?
;

// class BareIrStream
bare_irstream:
    /* Empty */
    | irstream_item (','  irstream_item)*
;

// interface IrStreamItem
// Note: extent is implicit within duration
irstream_item:
```

```
    variation
    | bitfield  // must come before duration!
    | assignment
    | duration
    | irstream
    | bitspec_irstream
;

// 7.4
// class BitSpec
bitspec:
    '<'  bare_irstream ('|' bare_irstream)* '>'
;

// 8.2
// class RepeatMarker
// NOTE: Semantically, at most one infinite repeat in a protocol makes sense.
repeat_marker:
    '*'
    | '+'
    | number '+'?
;

// class BitspecIrstream
bitspec_irstream:
    bitspec irstream
;

// 9.2
// class Expression
// called expression in spec
para_expression:
    '(' expression ')'
;

// called bare_expression in spec
expression:
                    primary_item
    |               bitfield
    |               '~'             expression
    |               '!'             expression
    |               '-'             expression
    |               '#'             expression
    | <assoc=right> expression '**'                expression
    |               expression ('*'  | '/' | '%')   expression
    |               expression ('+'  | '-')         expression
    |               expression ('<<' | '>>')        expression
    |               expression ('<=' | '>=' | '>' | '<') expression
    |               expression ('==' | '!=')        expression
    |               expression '&'                  expression
    |               expression '^'                  expression
    |               expression '|'                  expression
    |               expression '&&'                 expression
    |               expression '||'                 expression
    | <assoc=right> expression '?'                  expression ':' expression
;

expressionEOF:
    expression EOF
;

// 10.2
// (class NameEngine)
definitions:
    '{' definitions_list '}'
```

```
;

definitions_list:
    /* Empty */
    | definition (',' definition)*
;

definition:
    name '=' expression
;

// 11.2
assignment:
    name '=' expression
;

// 12.2
// Variations are only allowed within infinite repeats.
variation:
    alternative alternative alternative?
;

alternative:
    '[' bare_irstream ']'
;

// 13.2
// class Number
number:
      INT
    | HEXINT
    | BININT
    | 'UINT8_MAX'
    | 'UINT16_MAX'
    | 'UINT24_MAX'
    | 'UINT32_MAX'
    | 'UINT64_MAX'
;

// class numberWithDecimals extends Floatable
number_with_decimals:
    number
    | float_number
;

// Due to the lexer, have to take special precautions to allow name-s
// to be called k, u, m, p, lsb, or msb. See Parr p.209-211.
// class Name
name:
    ID
    | 'k'
    | 'u'
    | 'p'
    | 'm'
    | 'lsb'
    | 'msb'
;

// class ParameterSpecs
parameter_specs:
    '[' parameter_spec (',' parameter_spec )* ']'
    | '['  ']'
;

// class ParameterSpec
```

```
parameter_spec:
      name     ':' number '..' number ('=' expression)?
    | name '@' ':' number '..' number  '=' expression
;

// class FloatNumber
float_number:
    '.' INT
    | INT '.' INT
;

// Extension: Here allow C syntax identifiers;
// Graham allowed only one letter capitals.
ID:
    ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
;

INT:
    ( '0' .. '9')+
;

HEXINT:
    '0x' ( '0' .. '9' | 'a' .. 'f' | 'A' .. 'F' )+
;

BININT:
    '0b' ( '0' | '1' )+
;

// Extension: Not present by Graham.
COMMENT: // non-greedy
    '/*' .*? '*/'                       -> skip
;

LINECOMMENT:
    '//' ~('\n'|'\r')* '\r'? '\n'       -> skip
;

WS:
    [ \t\r\n\u000C]+                    -> skip
;
```

2.6 The Girr format for universal IR Commands and remotes.

Date	Description
2014-01-28	Initial version.
2016-04-28	Updated to current version.
2019-07-21	Updated to current version; IrpMaster replaced by IrpTransmogrifier. Say "Pronto Hex" instead of "CCF".
2020-05-29	Some minor updates. Added Maven integration (which previously was found in README.md).

Date	Description
2021-06-03	Several minor fixes and clarifications, in particular embedded protocols.
2022-04-23	Some minor improvements.

Table 1: Revision history

2.6.1 Background and Introduction

There are several Internet sites in whole or in part dedicated to infrared control of customer electronics. Very soon the question on exchange of IR signals, individual or as a set of commands from one remote or one device, comes up. For individual IR signals, the Pronto Hex, sometimes called CCF, is the one most used. This describes *one* signal, without a name or any other attributes. To use, the user will most likely have to copy-paste the information from a downloaded file, or a forum contribution, into his/her application program. For several signals, this unsystematic procedure is both tedious and error prone.

Some manufacturers publish the IR commands for their products, often as tables as Excel list or as PDF documents. There are also some quite impressive user contributed collections around in Excel format, e.g. for Sony and Yamaha equipment. Often, these lists contain not only the Pronto Hex form, but even a protocol/parameter form. These lists definitely mark a step in the right direction. With sufficient skills with the involved tools it is often possible to transfer a whole set of commands, possibly even preserving names, with a few clicks. However, this is still a manual process, that is not suited for automation.

On the other hand, there are a few file formats around, describing a complete setup of a programmable remote control, like the Philips Pronto CCF file format or the XCF format of the Pronto Professional. These describe a complete setup, including layout of buttons and pages, font selection and other items not of interest for the exchange of IR signals. The "device updates" (rmdu-files) of RemoteMaster also falls into this category: They do contain the IR Signals, either as a raw representation or in a protocol/parameter format, but also a number of key bindings, more-or-less specific to a particular JP1 remote.

The Lirc project however has a data base file format, containing named commands, grouped into named "remotes". However, the Lirc format was never intended as an exchange format, and, as a general rule, only Lirc program can read Lirc files. Also, Lirc has not a viable concept of intro- and repeat sequences. Lirc also does not handle meta data (manufacturer, device type, model, etc.

This leads to our mission:

2.6.1.1 Mission

To define a universal format for the exchange of IR signals, encompassing both for protocol/parameter form, and different textual formats, like Pronto Hex. The format should describe the IR signals with their names, (not their semantics). The commands should be bundled within "remotes". It should be readable and writable by both humans and programs. The format should be free/open for everyone to implement, in open or proprietary contexts. It should use open technology, so that tools can be implemented using currently spread free software.

Everyone is invited to implement this format in other programs, or tools for the format.

2.6.2 Program support

IrScrutinizer uses Girr as its preferred format for import and export of IR signals. It can interactively import and export from many different file formats and data bases.

RMIR (sometimes called RemoteMaster) is a powerful program for programming so-called JP1-remotes. Since version 2.11, it can export and import Girr files directly.

Jirc can generate Girr files from Lirc configuration files. (It is also included in IrScrutinizer.)

IrpTransmogrifier can generate the output from the `decode` and the `analyze` commands in Girr format. (In order to avoid circular dependencies, it does not use the support library described here.)

GirrLib is a small collection of Girr files.

2.6.3 Copyright

The rights to the described format, as well as the describing file are in the public domain. That also goes for the present document. Note that this is in contrast to other documents on www.harctoolbox.org for which no copying or re-distribution rights are granted, or the therein contained software, which is licensed under the Gnu General Public License, version 3.

2.6.4 The name of the game

Pronounce "Girr" like "girl", as one word (not G.I.R.R.). It should be used as a proper noun, capitalized (not uppercase). Preferred file extension is `girr`, but this is not necessary. Also, e.g. `xml` is possible.

2.6.5 Requirements on a universal IR command/remote format

It should be an XML file determined by an XML Schema. It should, however, be usable without validating parsers etc.

The formal rules (enforced by Schema) should be as non-intrusive as possible, possibly prohibiting "silliness", but otherwise requiring at most a minimum of formal syntactic sugar.

A remote is in principle nothing else than a number of commands. In particular, it should not determine the semantics of the commands, nor does it describe how to control a device that can be commanded by the said remote. Names for commands can be "arbitrary", in any language or character set, using any printable characters including white space. However, it is recommended to us "simple" names in English, without non-ASCII characters or embedded whitespace. (A `displayName` can be used for localized names, with arbitrary characters etc.) A well defined semantic of command names is not guaranteed. However, in some cases uniqueness in the purely syntactical sense is required, for example ensuring that all commands within a particular `commandSet` have unique names.

It can be assumed that all signals consists of an intro-, an repeat-sequence (any of which, but not both, may be empty), and an optional ending sequence.

It should be possible to describe signals either as parametrized protocols, in raw form, or in Pronto Hex form. If several forms are present, it should be clear which one is the primitive form, from which the others are derived.

It should be suitable both for human authoring (with a minimum of redundancy), as well as machine generation (where simple structure may be more important than minimum redundancy).

It should be a container format, namely extensible with respect to textual representation of IR Signals and -sequences.

2.6.6 Demarcation

- The present work aims at a description for remotes, not devices (e.g. in the sense of this). Thus, command names are free form strings, with no semantics inferred.
- Only unidirectional "commands" are considered, not data communication.
- It is only attempted to define IR signals and "sufficiently similar" signals. One such signal/sequence consists of a sequence of durations, namely alternating on- and off-times. Except for the "normal" IR signals, this includes RF signals of frequencies 433, 318, 868 MHz etc. used e.g. for controlling power switches.

2.6.7 Informal overview of the Girr format

There are four different possible root element types in the format: `remotes`, `remote`, `commandSets`, and `commands`. All can be the root element of a conforming Girr document, although some software may not handle all of them. (The previous versions of our supporting library only supported `remotes` as root element, but this restriction has been lifted.) Basically, the element `remotes` contains one or more `remotes`, each containing one or more `commandSets`, each containing a number of `commands`.

2.6.7.1 command

This element models a command, consisting essentially of a name and an IR signal, in one or several different representations. Names can consist of any printable characters including white space, and carries a priori no semantics.

Consider the following example:

```
<command name="play" displayName="Play |&gt;" comment="" master="parameters">
    <parameters protocol="nec1">
        <parameter name="D" value="0"/>
        <parameter name="F" value="0"/>
    </parameters>
    <raw frequency="38400" dutyCycle="0.50">
        <intro>+9024 -4512 +564 -564 +564 -564 +564 -564 +564 -564 +564
               -564 +564 -564 +564 -564 +564 -564 +564 -1692 +564 -1692 +564 -1692 +564
 -1692
               +564 -1692 +564 -1692 +564 -1692 +564 -1692 +564 -564 +564 -564 +564
 -564 +564
               -564 +564 -564 +564 -564 +564 -564 +564 -564 +564 -1692 +564 -1692 +564
 -1692
               +564 -1692 +564 -1692 +564 -1692 +564 -1692 +564 -1692 +564 -39756
         </intro>
         <repeat>+9024 -2256 +564 -96156</repeat>
    </raw>
    <ccf>0000 006C 0022 0002 015B 00AD 0016 0016 0016 0016 0016
          0016 0016 0016 0016 0016 0016 0016 0016 0016 0016 0016
          0016 0041 0016 0041 0016 0041 0016 0041 0016 0041 0016
          0041 0016 0041 0016 0041 0016 0016 0016 0016 0016 0016
          0016 0016 0016 0016 0016 0016 0016 0016 0016 0016 0016
          0041 0016 0041 0016 0041 0016 0041 0016 0041 0016 0041
          0016 0041 0016 0041 0016 05F7 015B 0057 0016 0E6C
    </ccf>
    <format name="uei-learned">00 00 2F 00 D0 06 11 A0 08 D0 01 1A
           01 1A 01 1A 03 4E 01 1A 4D A6 11 A0 04 68 01 1A BB CE 22
           01 11 11 11 12 22 22 22 21 11 11 11 12 22 22 22 23 82 45
    </format>
</command>
```

(Details on syntax and semantics are given in the next section.)

In the `parameters` element, parameters and protocol can be given. They can be completely given, or they may be inherited from parent element of type `commandSet`.

The raw and the Pronto Hex form may be given next, as above. Finally, one or may auxiliary formats of the signal can be given.

Fat Format

For the ease of further processing of the result, the sequences within the `<raw>` element can alternatively be given in the "fat" format, where each flash (on-period) and each gap (off-period) are enclosed in their own element, like in the following example:

```
<command name="play" displayName="Play |&gt;" comment="" master="parameters">
    <parameters protocol="nec1">
        <parameter name="D" value="0"/>
        <parameter name="F" value="0"/>
    </parameters>
    <raw frequency="38400" dutyCycle="0.50">
```

```
        <intro>
           <flash>9024</flash>
           <gap>4512<gap>
           <flash>564</flash>
           <gap>564</gap>
           <flash>564</flash>
           <gap>564</gap>
           ...
```

2.6.7.2 commandSet

`commandSet`s bundles "related" commands together. They may contain `parameters` elements, in which case the protocol name and the parameeters therein are inherited to the contained `command`s.

The use of `commandSet`s is somewhat arbitrary. They can be used e.g. to structure a remote containing a few different protocols, or one protocol and a few different device numbers nicely, in particular if hand writing the Girr file. However, protocol and their parameters can also be given as `parameters` within the `command` element.

Often, a device can be controlled in one of several "id-s", corresponding to different IR signals. See the Oppo BDP-83 as example. The different id-s can be conveniently modeled as different `commandSet`s. These are (almost) identical, typically differing only in the device/subdevice parameter in the protocol. In other cases, devices implement a "new" and an "old" set of commands; see for example a Philips TV (RC5 and RC6 protocols) or a Denon AVR receiver (old "Denon" protocol and new "Denon-K" protocol).

2.6.7.3 remote

A `remote` is an abstract "clicker", containing a number of `command`s. The name of the contained commands must be unique within a `commandSet`, but the same name may be present in more than one `commandSet`.

2.6.7.4 remotes

`remotes`, as the name suggests, is a collection of `remote`s, identified by a unique name.

2.6.7.5 Embedded protocols

The parameter form references to a protocol using its name. Normally, this is assumed to be known to a processing program. However, it is also possible to embed additional protocols in a Girr file, and to refer to it by its defined name in the declaration.

2.6.8 Detailed description of syntax and semantics of the Girr format

2.6.8.1 Version

This article describes the Girr format version 1.2, identified by the attribute `girrVersion`, expected in the root element of an instance. (Not to be confused with the version of the support library.)

2.6.8.2 Namespace

The Girr namespace is `http://www.harctoolbox.org/Girr`.

It is recommended to parse instances with a namespace- and XInclude-aware parser.

2.6.8.3 Imported namespaces

Except for the `namespace` namespace (`http://www.w3.org/XML/1998/namespace`), the namespaces `XInclude` (`http://www.w3.org/2001/XInclude`) and `html` (`http://www.w3.org/1999/xhtml`) are imported. For embedding of additional protocols, the `irp` namespace (`http://www.harctoolbox.org/irp-protocols`) is used; imported only on demand. XInclude- and html elements can be used at appropriate places, see the schema.

2.6.8.4 Schema

The grammar of Girr is formally described as an XML schema residing in the file girr_ns.xsd. It contains internal documentation of the semantics of the different elements. The official schema location is http://www.harctoolbox.org/schemas/girr_ns.xsd.

Here is generated schema documentation (thanks to Gerald Manger).

2.6.9 Stylesheets

A Girr file can be viewed in the browser, provided that it is associated with a style sheet. This is either a cascading style sheet (css), which essentially tells the browser how different elements are to be rendered, or an XSLT style sheet, which internally translates the XML document to a HTML document, normally with embedded style information. A description of these techniques is outside of the scope of the current document (see this document as an introduction); an example is given as simplehtml.xsl.

To use, add a line like

```
<?xml-stylesheet type="text/xsl" href="simplehtml.xsl"?>
```

to the Girr file. (Some programs, like IrScrutinizer, can do this automatically.) Note that some browsers, like Firefox, for security reasons limits the usage of style sheets.

XSLT style-sheets can however be used for other purposes than the name suggests. The export mechanism of IrScrutinizer consists essentially of the application of XSLT stylesheets on the Girr fat format.

2.6.10 Supporting Java library

For importing and exporting Girr files to Java programs, a Java library is provided. It is documented by its Javadoc documentation. As opposed to the specification as such, it is licensed under the Gnu General Public License, version 3.

At the time of writing, the library carries the version number 2.2.10.

Previous versions only supported import and export of documents having `remotes` as root element.

The library requires the IrpTransmogrifier classes.

2.6.11 GirrLib

I maintain a small library, GirrLib, available at GitHub. It consists of "my" collection of Girr files. Although not actively maintained, contributions are welcome. The files therein are in the public domain.

2.6.12 Integration in Maven projects

This project can be integrated into other projects using Maven. For this, include the lines

```
<dependency>
    <groupId>org.harctoolbox</groupId>
    <artifactId>Girr</artifactId>
    <version>1.2.3</version>  <!-- or another supported version -->
</dependency>
```

in the `pom.xml` of the importing project. This will also include the [IrpTransmogrifier](http://harctoolbox.org/IrpTransmogrifier.html) jar.

2.6.13 Sources

The sources, both the Java library, the schema, and the current document, are maintained at this Github repository. API documenation (current development version) is available here.

2.6.14 Appendix. Parametrized IrSignals

The purpose of this section is to make the article more self-contained. Information herein are described in greater detail elsewhere.

The Internet community has classified a large number of IR Protocols, see e.g. this listing. These protocols consist of a name of the protocol, a number of parameters and their allowed domains, and a recipe on how to turn the parameters into one, two, or three IR sequences, making up an IR signal. This recipe is often expressed in the IRP Notation, which is a compact formal representation of the computations involved. For particular values of the parameters, a rendering engine computes the resulting IR signal, often in Pronto Hex format, or in raw format.

2.7 General InfraRed Server Command Language

Date	Description
2014-01-06	Initial version.
2014-09-16	Added the communication section.

Table 1: Revision history

2.7.1 Requirements

- Demarcation: This deals only with sending, receiving (including decoding), storing etc of IR signals. Not: serial and other text base communication, nor the acting on received signals.
- However, RF signals for remote control are included, since they only differs from IR signals by using another carrier signal.
- Modularized, named modules containing commands (like Java interfaces). Written in capitalized CamelCase.
- Inheritance within modules, multiple inheritance
- Extensible: Developers can define new modules
- Only the Basic module mandatory, containing the commands `version` and `modules`.
- Very low-weight, should be implementable on e.g. Arduino.
- As "dumb" as possible.
- Basic version: text socket/serial interface. Versions using json, xml/soap, http/rest possible.
- Authentication as optional module, several submodules for different sort of authentication.
- Command structure: `command [subcommand] [options] [arguments]`
- Response structure: TBD.
- Names for IR commands, hardware: arbitrary strings using the English language, case sensitive matched using charsets.
- Our command names: C-syntax; lowercase only, underscore discouraged. “get” and “set” left out unless necessary for uniqueness or understandability (like `getcommand`).

2.7.2 Specification

Typography: module names are in **bold**, command names `monopitch`.

2.7.2.1 Introduction

This list is not an unrealistic Christmas wish list, but a list of modules, only the first one mandatory. Through the module concept, a conforming GIRS server can be anything from an Arduino with just an IR sender LED and a sketch a few pages long, and a

fat server with several input- and output-devices, (each) having several transmitters, combined with a full blown data base, with user administration and authentication.

Note that there is a number of properties for e.g. LIRC that has been rejected here, in particular the ability to execute commands. (These should be handled by another program.)

A capable server should probably also implement some sort of discovery beacon, for example AMX style.

2.7.3 Modules

2.7.3.1 Base

This is the only mandatory module.

`version`

- returns manufacturer, manufacturer's version number, or another useful version string.

`modules`

- returns list of implemented modules, separated by whitespace.

2.7.3.2 NamedRemotes

Support of remotes identified by name, like LIRC.

`remotes`

- argument named/uei: What type of remotes to report.
- returns: list of remotes, either names or manufacturer/device-type/setupid

`commands`

- argument: remote in a supported format (mandatory)
- returns: tab(?) separated list of command names, in currently selected char set.

`database` (module database)

argument: data base name. Required.

2.7.3.3 UeiRemotes

Support of remotes identified by manufacturer, device type (both arbitrary strings), and a setup number (most commercial data bases)

`manufacturers`

- returns: tab separated list of manufacturers.

`devicetypes`

- argument: manufacturer
- returns: tab separated list of device types

`setupids`

- arguments: manufacturer, device
- return list of setup ids.

`database (module database)`
- argument: data base name. Required.

`database-password` (module database)
- argument:password.

2.7.3.4 OutputDevices

Allows for accessing several devices; several instances of the same type: Names like “Greg's GlobalCaché”. (Configuration of these over this API is not intended.) Each has their own set of transmitters.

`outputdevices`
- returns:list of known devices

`outputdevice` Set default output-device
- argument: device name

`outputdevicecapacities`
- argument device name (optional, defaulted)
- result: list of capacites. Possible values (extensible): fmax, fmin, zero_frequency_tolerant. Inherit to transmitters.

2.7.3.5 InputDevices

Allows for accessing several devices; several instances of the same type: Names like “Greg's GlobalCaché”. Configuration of these over this API is not planned. An input device does not possess transmitters.

`inputdevices`
- returns: list of known devices

`inputdevice` Set default input device
- argument: device name

2.7.3.6 Transmitters

Same commands as OutputDevices. (????)

`transmitters (module transmitters)`
- argument: output-device

`output-device`
- returns: list of transmitters, max-number-transmitters-enabled

`settransmitters` Selects default transmitter for the output device selected. ***(TBD: Alternative: ditch the default transmitter and this command, thus transmitter argument mandatory.)***

`transmittercapacities`
- arguments:
  - output-device (optional, use default if not given)
  - `transmitter` transmitter (only one!)

- result: list of capacites.Possible values (extensible): ir (connected to IR LED). Rf (connected to RF modulator) hard-carrier=frequency (in particular for RF, 433M, 868M (Hz or suffix M,k)). Inherits from outputdevicecapacities.

2.7.3.7 Transmit

Access may be restricted through user rights. There is always a default output device; if the **OutputDevices** module is implemented, there may be more.

`transmit` (semantic for repeats may be implementation dependent)

- subcommands (at least module (??) has to be implemented):
  - `ccf` (module ccf). Parameter: CCF string
  - `raw` (module raw). Parameter: frequency, duty cycle, intro, reps, ending.
  - `Irp` (module irp). Parameters: protocol name OR irp-protocol, parameters.
  - `Name` (module named-command). Parameters: remote (one of the suppoted formats), command name
- options:
  - transmitters (module transmitter) (optional (or not?))
  - output-device (optional, otherwise use default)
  - transmit-id (module transmit-id) (optional)
  - # sends (default 1)
  - wait (wait for command completion)
- returns: (after completion) confirmation command, with transmitter and transmit-id

`stop` (module `Stop`)

- Argument: output-device, transmitter, transmit-id (optional)

2.7.3.8 Capture

for capturing (“learning”) of new remotes. Dumb command, intelligence should sit in the calling program.

`analyze`

- Arguments: (all having sensible defaults.)
  - input-device
  - start-timeout
  - capture-timeout
  - ending-timeout
- Returns: frequency, raw ir-sequence, optionally duty cycle.

2.7.3.9 Receive

for receiving commands, possibly for deployment solutions. Dumb command, intelligence should sit in the caller. Identifying start separately (like for volume control) not supported.

`receive`

- Arguments:
  - return format(TBD)
  - input-device
  - timeout
  - filter, syntax, (syntax, semantics TBD)
- subcommand named (module named-command)
  - Return value: received command name (+ remote)
- subcommand decode (module decoder)
  - Return value: protocol name, parameters

`relay` (module relay), to send events to other servers

- Arguments:
  - return format (TBD)
  - `protocol` (http/tcp/udp/shell?)
  - portnumber
  - ipaddress
  - filter (TBD)

2.7.3.10 Store

allows for uploading new commands to the server. May be restricted through authentication and user rights.

`store`

- arguments: data base (optional), name, remote (in a supported format), signal in a form dependent on the subcommands.
- subcommands
  - ccf (module ccf)
  - raw (module raw)
  - irp (module irp(?))

`commit` Stores the recently downloaded commands persistently.

- Argument: data base name (optional)

2.7.3.11 Command

allows for downloading commands from the server. Inverse of store. May be restricted through authentication and user rights.

`getCommand`

- Argument:
  - data base (optional)
  - output format: ccf, raw, irp,... (also other can be supported)
- Return: command in desired format.

2.7.3.12 Authentication

Several different models for access control are possible, and can be implemented through different modules. The first just requires a password to all the services. The second allows user based restrictions: Some commands/subcommands/arguments might be restricted to some users. Of course, sending passwords unencrypted over the net is not to be considered very secure, so preferably ssh or similar, or a challenge-response system should be used.

`login (module password), for password protected services`

- argument: password

`login` (module UserPassword), for user/password protected services, possibly with different rights for different users.

- argument: user, password

`sshlogin` TBD (module ssh)

`logout`

2.7.3.13 Charset

Determines charset used for input and output.

`charset`

- argument: charset name.

2.7.4 Communication

Communication is typically taking place over a bidirectional ASCII stream, like serial, "terminal", connection or a through a TCP socket. The commands sent to the GIRS server should be of the form `command [subcommand] [options] [arguments]`, where `command` can be abreviated as much as unabiguity allows (typically to the initial character). The form of the responses should be a "natural" ASCII response in the form of one line (typically); the tokens separated by whitespace.

Interacting with a GIRS server through static or dynamic linking can also be possible, either by decoding a command line, or with a number of API functions.

2.8 Architecture Concept

Date	Description
2014-09-20	Initial version.

Table 1: Revision history

2.8.1 Preliminaries

2.8.1.1 Sending/Receiving

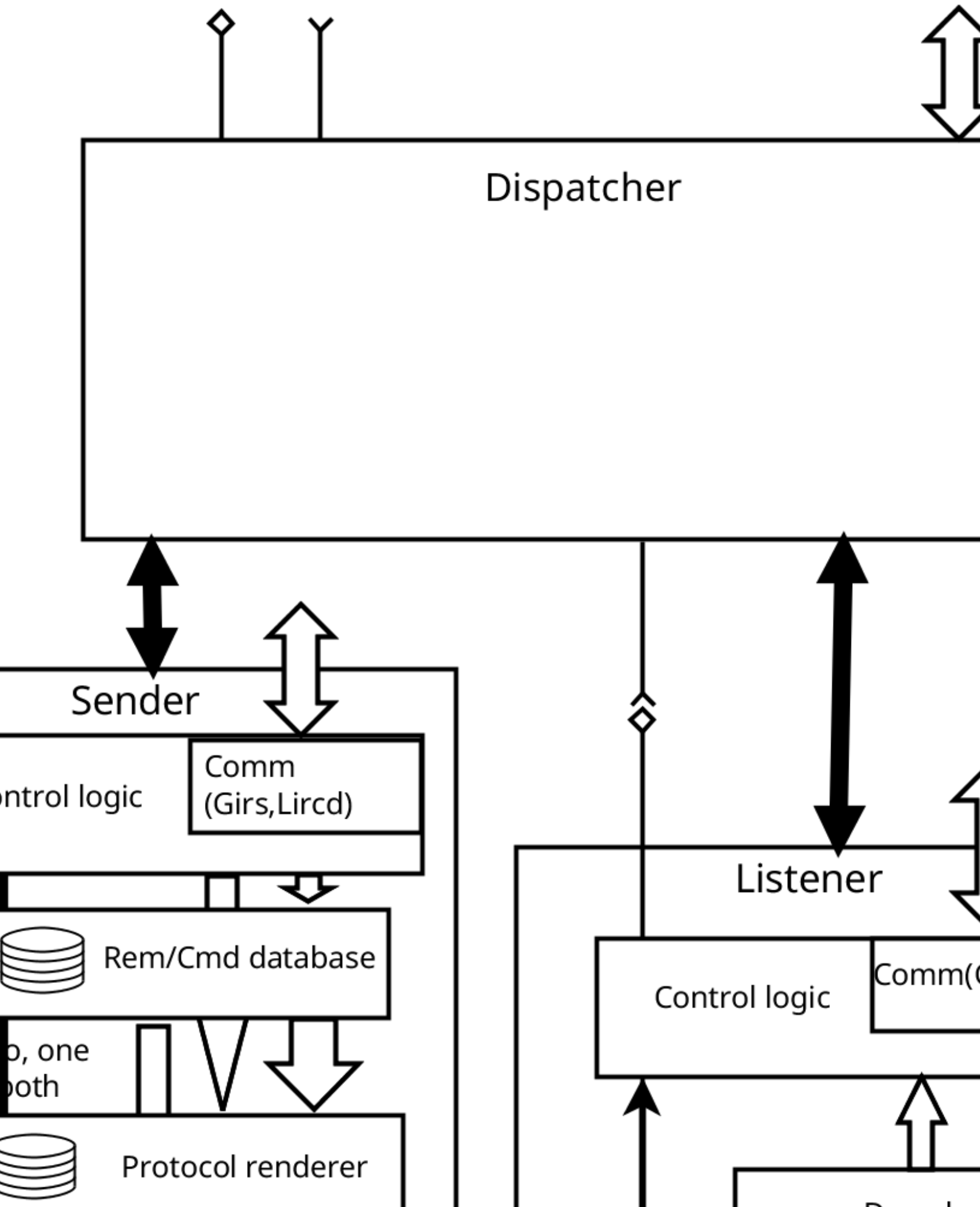
Traditionally, IR hardware and software combines sending and receiving capacities in one unit — unless of course only one of these functinallities are supported. Instead, we argue that sending and receiving of IR signals are fundamentally different activities which preferrably are threaded separately. 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.

2.8.1.2 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.).

2.8.2 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
ntrol logic
Comm
(Girs,Lircd)
Rem/Cmd database
o, one
both
Protocol renderer
Listener
Control logic
Comm(

2.8.2.1 The Sender

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.

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.

2.8.2.2 The Listener

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.

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.

2.8.2.3 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.

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.

2.8.3 Implementation notes

2.8.3.1 Sender

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 several Java drivers for IR hardware, that can be used more or less directly.

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.

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.

2.8.3.2 Listener

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.

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.

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.

2.8.3.3 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?)

2.8.4 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 `org.harctoolbox.harchardware.ir.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.

2.9 Glossary and terms

Date	Description
2013-12-01	Initial version.
2013-12-20	More stuff added.
2014-02-02	Even more stuff added.
2014-09-18	Yet more stuff added.
2016-01-08	Substantial improvements and extensions.
2016-04-29	Some more entries.
2017-03-19	Some more entries.
2019-08-03	Minor updates, triggered by IrpTransmogrifier.
2019-12-25	Yet another batch of minor updates.
2022-05-13	Some updates in the context of IrScrutinizer 2.4.0.

Table 1: Revision history

2.9.1 Glossary

Here we explain and define some of the used terms. In most cases (but not all!), this correspond to established usage in the Internet, e.g. in the JP1 forum. It should also be pointed out that in some cases, in particular when comparing programs by others to my own, the assessment should be considered as subjective.

Substantial program/program packages are written capitalized, (not uppercase), in some case in CamelCase, and in Roman typeface, like a proper noun. "Small" programs are written as `code`. For example, we write "Lirc", not "LIRC", or `lirc`; and `irsend`.

AMX Beacon
A daemon program (or protocol) implemented in some networked components. It is used for periodically announce their existence, and some of their properties.

AnalysIR
Commercial infrared analyzer and decoder program for Windows. By its makers characterized as "... the leading tool available for analysing, decoding and reverse engineering infrared remote control protocols". Last release appears to be from July 2016. Web site.

Analyzer

We use the word "analyzer" for a program that takes one or more IR signals/ sequences as input, and from them computes one (or several alternative) IRP-Forms (with parameter values) that is compatible with the data, i.e. that could have generated the given data. IrpTransmogrifier contains an implementation, used in the current version of IrScrutinizer.

ANTLR

"A powerful parser generator for reading, processing, executing, or translating structured text or binary files. ... From a grammar, ANTLR generates a parser that can build and walk parse trees." Website, see also documentation site.

Arduino

A single-board micro-controller, based upon a 8-bit Atmel AVR micro-controller, or a 32-bit Atmel ARM. Official site. Can be used for implementing sending and/or capturing of infrared signals with additional hardware attached to its IO-pins, see for example this project. The main components of the Arduino eco systems are:

- The Arduino IDE (Integrated Development Environment) (which is an open source Java IDE program containing an editor and a code preprocessor. It invokes the tool chain),
- A set of includes and libraries for the processor, mainly GNU software,
- A tool chain (compiler, linker, "binutils") for the processor, mainly GNU software (installed with the IDE),
- A bootloader,
- A number of hardware boards based on "open hardware".

Arduino library manager

One part of the Arduino ecosystem. Consists of a number of user-contributed, not officially supported libraries for Arduino. From the Arduino IDE, selecting "Tools" -> "Manage libraries" a GUI frontend to these libraries is opened, allowing to install and update any of these libraries.

Bracketed raw format

Simple text format for formatting of a raw IR signal, introduced by IrpTransmogrifier. Intro-, repeat, and (optionally) ending sequences are enclosed by brackets ("[]", sometimes called "square brackets" or "square parenthesis"). Signs ("+" for flashes, "-" for gaps), as well as interleaved commands or semicolons, are accepted, and ignored (even if they are not in correct order). Optionally, the beginning sequence can be preceeded by `Freq=`*frequency_in_Hz*`Hz` giving the modulation frequency. If not explicitly given, a default value (dependent on the reading program) is assumed. Example (an RC5-signal): `Freq=36000Hz[][+889,-889,+1778,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-90886][]`.

Capturing (sometimes "learning")

The process of recording an unknown physical IR sequence as a sequence of on- and off-durations, preferably together with a measure of its modulation frequency. See (Non-demodulating) IR receiver. Cf. receiving IR signals.

CCF (text signal format)
Sometimes used as a synonym for the Pronto Hex format. Not to be confused with the ccf file format of the Pronto Classic remotes!
ccf file format
Not to be confused with the CCF text signal format! Configuration file format for the Pronto Classic. File extension is `ccf`. Can be edited by the ProntoEdit program. Has been completely reverse engineered, and the open-source program Tonto is able both to interactively edit them, as well as non-interactively through an API. IrScrutinizer can import and export ccf files, using the said API.
Cleansed signal
Given a captured dirty signal, numerically "close" duration values are lumped into one single value. Often combined with a repeat finder.
CML
Proprietary binary format by RTI for their IR database files. Has been reverse engineered; IrScrutinizer can import it. One well known CML file is the Mega List Database, which is a huge database of IR codes maintaned by Glackowitz from the Remote Central Forums.
Command
Here, an IR signal with a name, like "Play".
Consumer IR (CIR)
Consumer IR deals with IR control of various devices, often audio or video. Wikipedia article. Not to be confused with IRDA. Typically uses wave lengths of 940–950nm.
CVS (comma separated values)
Primitive data base format, one record consisting of one line, the entries separated from one another by a comma (,) (or sometimes another character). One possible file extension `csv`. Can be read directly by spreadsheet programs.
Decode (noun)
Given an IrSequence or an IrSignal, a *decode* is a protocol, together with parameter values, that are compatible with the given IrSequence/IrSignal, i.e. could have generated the original signal. Note that the determination is governed by numerical uncertainties, so that small deviations from the perfect signal are accepted. Furthermore, one signal/sequence may have none, one, or more valid decodes.
DecodeIR
Library for the decoding of IrSequences. Originally written by John Fine, extended by others; used by many widely spread programs as a shared library, often using JNI. The current version is 2.45. License: public domain.Binaries for Windows, Linux, and Mac, source code at SourceForge. Arduino version (also available in the Arduino library manager). Now superseded by IrpTransmogrifier.
Demodulating IR Receiver
An integrated circuit that receives a modulated IR signal and recovers the original signal with the modulation removed. The modulation frequency of the signal must "match" the frequency of the demodulator. Receiver chips are typically marked

TSMP*XXYY*, where *XX* (two or three digits) denotes a vendor specific type, and *YY* the modulation frequency in kHz. (Data sheet for a typical product.) Not suited for capturing of unknown signals, since it removes the modulation frequency without identifying it.

Device Number

See protocol parameters. Denoted by D in IRP protocols.

`/dev/lirc`

Using Linux, a device node for a connected, supported IR device. Despite the name, it is a part of the Linux kernel, not Lirc, thus available also if Lirc is not installed. Supported by IrScrutinizer, using the DevSlashLirc library.

Device Type

Class of components, like TV, VCR, Satellite receiver, etc.

DevSlashLirc

Library for object oriented access to `/dev/lirc` -hardware using Java or C++ (currently). Source repository.

Dirty Signal/Sequence

A physically measured signal or sequence containing random measurement errors, "dirt". Such a signal/sequence contains several numbers that are close, but not equal.

Duration

A duration is either a gap or a flash.

Duty Cycle

The percentage of the time the the modulation pulse is on. Typically 50% or slightly less.

Ending sequence

See IrSignal.

Eventghost

"EventGhost is an advanced, easy to use and extensible automation tool for MS Windows. It can use different input devices like infrared or wireless remote controls to trigger macros, that on their part control a computer and its attached hardware." Licensed under GPL2. Home page.

ExchangeIR

Library for IR signal analysis and exchange by Graham Dixon. Licensed under the GPL3 license. Some interesting parts are an Analyzer, a repeat finder, and functions for the UEI learned format. These parts has been translated to Java by myself: source.

Executor

An embedded program fragment for the rendering and transmission of one or several protocols. One executor can manage several protocols; also, for one protocol there may be several alternative executors. An executer has its own parametrization, more-or-less similar to the parametrization of the protocol. Used in JP1 Remotes and RemoteMaster.

Flash (or "mark", "on-period")

Period of time when the IR light is "on", or flashed with the selected modulation frequency. See IrSequence.

Function Number
See protocol parameters. In IRP protocols, denoted by F. In the JP1 community, often the synonym OBC is used.
Gap (or "space", "off-period", "Pause")
Period of time when the IR light is off. See IrSequence.
Generating, sometimes called rendering
The process of evaluating an IrProtocol for a particular parameter value, rendering an IrSignal. Commonly used rendering programs/engines are the older MakeHex, IrpMaster (included in versions 1 of IrScrutinizer, and the more modern and capable IrpTransmogrifier (included in the current versions of IrScrutinizer).
Girr (Generic IR Remote)
A general XML-based exchange formats for IR Signals. Really a container format that can contain any of the Pronto Hex, raw format, protocol/parameter format, as well as other text formats like Global Caché sendir. For a full description, see the full documentation.
Global Caché
Manufacturer of IR sending hardware, in some cases with learning and/or receiving possibilities. Web site. Supported in IrScrutinizer.
GPL3 license
The current version of the GNU General Public License. Used by my software projects (with some exceptions), and many so-called open-source software projects. The basic idea is the licensee is allowed to use, enhance etc. the software (also in a commerical product and context), but is not allowed to turn it, or a derived product, into non-free software.
Iguana USB IR transceiver
A family of USB transmitter/receivers for IR signals. Supported by IrScrutinizer on Lirc (only) by using the `/dev/lirc` interface.
IR (Infrared light)
According to Wikipedia, infrared light are light (electromagnetic radiation) of wavelength between 700nm and 1mm. For control of consumer electronics (CIR), according to Wikipedia, wavelengths around 870 nm and 930-950 nm (latter preferred), in comparison to IrDA (850-900nm) are used. Almost always generated by an IR LED.
IrDA
IrDA is a method for data exchange between PCs and portable devices. It is no longer to be considered as state-or-the-art, and has been almost completely replaced by Bluetooth and WiFi. Many devices with IrDA hardware are still around, but they are, possibly with a few exceptions, unsuitable for consumer IR.
IR LED (light emitting diode)
Semiconductor component capable of sending light with the desired IR wavelength. A typical representative is the Osram SFH 4546.
(IR) Protocol

An algorithm for turning a number of parameter values into an IR signal. It defines the necessary parameters and their allowed values. By convention, the most frequently changing parameter is called "F" (function number). Almost all protocols have a "device number" called "D". Many protocols have a "sub-device" number, called "S". A few protocols have a toggle parameter, in general called "T", and being persistent. A protocol may also have other parameters, with "arbitrary" names.

IrMaster

A program for generating, analyzing, importing, and exporting of infrared signals. Now discontinued, replaced by IrScrutinizer.

IrpMaster

A program and API library for rendering IRP protocols version 2. See its documentation. Comes with a powerful (but slightly hard to use) command line interface. For GUI usage, see IrMaster and IrScrutinizer. Note that the word "IrpMaster" sometimes refers to the command line program, sometimes to the rendering engine contained in IrMaster and IrScrutinizer version 1. Now discontinued and obsolete; replaced by IrpTransmogrifier.

IRP Notation

Compact, slightly cryptical, notation for defining an IrProtocol. Specification.

IrpTransmogrifier

Library and command line program for the IRP notation protocols, with rendering, code generation, and recognition applications. Reference manual. Effectively replaces IrpMaster, DecodeIR", and ExchangeIR.

IrScope

Program that accompanies the IrWidget, also by Kevin Timmerman. Originally a support to the IrWidget, was further developed (in particular through Graham Dixon) to a fairly general and capable IR analyzing program, supporting also DecodeIR and ExchangeIR. The program was a major inspiration source for IrScrutinizer.

IrScrutinizer

IrScrutinizer is a powerful program for capturing, generating, analyzing, importing, and exporting of infrared signals. Reference manual.

irsend

Program that implements a simple client for the Lircd server daemon, for sending commands to a lircd server. Contained in the Lirc package. For a complete list or commands implemented, see its man page. IrScrutinizer contains a GUI version with (approximately) the same functionality, under `Tools -> Named Command Sender`. For a Python alternative, see Lirconian. For a Java alternative, see JavaLircClient.

IR Sequence

Sequence of time durations, in general in expressed microseconds, together with a modulation frequency. The even numbered entries normally denote times when the IR light is on (modulated), called "flashes" or "marks", the other denote off-periods "gaps" or "spaces". They always start with a flash, and end with a gap. Sometimes the

flashes are written with a leading "+"-, and the gaps with a leading "-"-sign. This has only a decorative purpose; all involved numbers are still positive.

IR Signal

Consists of three IR sequences, called

1. *start sequence* (or "intro", or "beginning sequence"), sent exactly once at the beginning of the transmission of the IR signal,
2. *repeat sequence*, sent "while the button is held down", i.e. zero or more times during the transmission of the IR signal (although some protocols may require at least one copy to be transmitted),
3. *ending sequence*, sent exactly once at the end of the transmission of the IR signal, "when the button has been released". Only present in a few protocols.

Any of these can be empty, but not both the intro and the repeat. A non-empty ending sequence is only meaningful with a non-empty repeat.

IrToy

An "open hardware" project by Dangerous Prototypes, see the product page. Consists of a microprocessor PIC18F2550, a demodulating IR-receiver, a non-demodulating IR-receiver, an IR-LED, and a USB-connector. Thus, is usable both for learning, including frequency measurements, receiving demodulated R-signals, and sending IR signals. Supported by IrScrutinizer up to and including 2.3.1, but not 2.4.0.

IrTrans

A series of IR products from the firm with the same name. IrScrutinizer and IrMaster support the Ethernet models (preferably with the "IR data base"), for sending only. Not supported in the current version 2.4.0.

IrWidget

An "open hardeware" project by Kevin Timmerman. Project page. That page presents many different versions, but the most spread version (commercially available by Tommy Tyler; unknown if still active) consists of a micro processor PIC12F629, a non-demodulating sensor (QSE15x), and a USB serial FTDI interface. Supported by Kevin's IrScope, as well as IrScrutinizer.

Java Native Interface (JNI)

A technique for having a Java program calling a native shared library (DLL in Windows, "Shared object" (`.so`) in other operating systems). See the Wikipedia article.

JP1

Community for customizing JP1 Remotes. Forum.

JP1 Remote

Customer remotes manufactured by Universal Electronics Inc., manufactured by many different names, like One for all etc. Has been reverse engineered by the JP1 community. Can be programmed through a connector, called (after the PCB label) JP1. The main tool for this is RemoteMaster.

JSON

A standard for using human readable text to transfer structured data, as an alternative to XML. See the Wikipedia article.

Lirc
An open source project for sending and receiving IR signals from Linux. Official web site. First release in May 1996, current version is 0.10.1, released in September 2017. Also packaged in all major Linux distributions. IrScrutinizer supports sending named commands to a Lirc server, and can also import and export files in `lircd.conf` format.

lircd
The main daemon of the Lirc project. Accepts commands on a Unix domain socket, alternatively on a TCP socket, by default 8765. Accepts commands to send IR signals in Remote/Command format only. To send commands to a running lircd, often a program called irsend is used.

lircd.conf
The main Lirc configuration file; a data base of remotes and its contained commands. Typically residing in `/etc/lirc/lircd.conf`. Also used to denote the file format. The files in the Lirc data base are in this format. Although its syntax and semantics is documentated, should not be considered a viable exchange format.
Can be imported by IrScrutinizer — since it contains a substantial amount of Lircd, translated to Java.

MakeHex
A predecessor to IrpMaster. Adheres to an earlier version ("Version 1") of the IRP Notation. For the original C++ program by John Fine, neither a GUI nor a command line interface are present; the parameters are given to the program by editing the data base files. A Java translation (by myself) exists, which has a command line interface, available here.

Mode2
1. A simple test format consisting of interleaved "on- (`pulse`) and off-durations (`space`).
2. A simple program contained in Lirc, printing the mode2 text format of received IR sequences to standard output.
3. In Lirc, notation for drivers/plugins capable of generating/evaluating data containing timing information.

Modulation frequency
During the "on" periods, the IR light is not just constantly on, but "flashed" on and off at a frequency called the modulation frequency, typically between 36kHz to 40kHz, in some cases higher (up to 56kHz), or much higher (455kHz, Bang & Olufsen equipment). This reduces noise sensitivity and power consumption, and also allows higher currents through the IR LED (that thus does not have to be able to survive the high current continuously). Also see Duty cycle.

(Non-demodulating) IR receiver
IR receiver that outputs the received IR signal essentially as received, i.e. without removing a modulation. Preferred sensor component is a chip with pre-amplifier like TSMP58000, at least for "moderate" modulation frequencies (< 60kHz).

Original Button Code

Acronym for "Original Button Code", a synonym for function number used in the JP1 community.

Parametric IR signal

An Ir Signal given as a protocol and an assignment to its parameters. The opposite is a raw signal. Of course, a renderer may compute the raw, numerical IR Sequences, but these are *considered* secondary, it is *defined* by its protocol and parameters values.

PCF

IR signal format, not to be confused with the pcf file format of the Pronto NG remotes! This is a proprietary and encrypted form of IR signals. As far as I am aware, it is presently not known how to decode this representation.

pcf file format

Like the xcf format, this is a ZIP file containing an XML file with the real payload, and a number of icon files. Unfortunately, the enclosed IR signals are in the PCF format, thus possible to decrypt only by the ProntoEditNG program.

Persistent variable

A *persistent variable* in an IR protocol may, but need not, be given a value before generating. If not, it retains its value from previous invocation, or, for the first invocation, has a default value.

Pronto Hex

IR signal format. Often called "CCF", "hex", or "Pronto". Consist of a sequence of four-digit hexadecimal numbers. For the interpretation, see the Appendix. It is a very popular format, e.g. for textual the exchange in Internet forums.

Pronto Classic

Legendary Family of advanced touch-screen remote controls. Manufactured by Philips between 1998 and 2004. Consists of the models TS1000, TSU2000, TSU6000, RC5000, RC5000i, RC5200, RC9200, RU890, RU940, RU970, USR5, RAV2K, RAV2KZ1. Configurable/programmable by a GUI program "ProntoEdit", as well as the open-source program Tonto.

ProntoEdit

Windows program for programming the Pronto remotes. Exists in different versions for different Pronto series. From its owner Philips now discontinued, but available for download at RemoteCentral.

Pronto frequency code

The second number in the Pronto hex representation. For *f* in Hertz, this is the four-digit hexadecimal number given as 1000000/(*f**0.241246). It can be conveniently computed by the Time/Frequency Calculator in IrScrutinizer, available under its Tools menu.

Pronto NG (New Generaton)

Later generation of Pronto touch screen programmable remotes. Uses the pcf format as their configurations. Can be read by ProntoEditNG.

Pronto Professional (`*.xcf` configuration files)

Later generation of Pronto touch screen programmable remotes. Uses the xcf format as their configurations. Consists of the models TSU9800, TSU9600, TSW9500,

TSU9400, TSU9300, TSU9200, TSU9500 (Philips) and RC9001 (Marantz). Discontinued in 2010.

Protocol Parameters

See IR Protocol.

protocols.ini

Data base file for RemoteMaster. Despite the name, it does not descibe protocols in our sense, but rather executors, their properties and parameterization.

properties (of an interactive program)

The part of the program's state saved between sessions for each user; saved to disk, or, sometimes with Windows, in the Windows registry.

Raw IR sequence/signal

A raw Ir Sequence is a sequence of (in general) measured on-off durations. It may or may not have one or many decodes, but these are *considered* to be secondary; its is *defined* by its numeric time durations. Often written with signs: a "+" indicates a flash, a "-" indicates a gap. A text format expressing the durations in microseconds is called "raw format", even if the signal as such is parametric.

Receiving IR signals (deployment)

The use case of receiving an a priori partially known (typically through its protocol, in particular, the modulation frequency) signal, identifying it completely (typically its parameters protocol parameters), and possibly initiating an action. Cf. the other use case capturing.

Repeat finder

A repeat finder is a program that, when fed with an IrSequence, tries to identify a repeating subsequence in it, and returns an IrSignal containing intro-, repeat-, and ending sequence, compatible with the given input data. The library ExchangeIr contains a repeat finder, which was used in IrScrutinizer versions up to and including version 1.2. IrpTransmogrifier also contains a repeat finder, which is used in the current version of IrScrutinizer.

Repeat sequence

See IrSignal.

Remote

1. A collection of commands with unique names.
2. A piece of hardware with buttons etc.

Remote/Command format

Given appropriate data base entries, the name of a remote (or device) together with the name of a command, can identify a command uniquely.

RemoteMaster

Powerful open-source tool for the programming of JP1 remotes. see Manual. Download current released version. Sources. Recent version can import and export Girr files.

RMDU file

Configuration file for "device update" for RemoteMaster, describing the configuration of a (universal) remote for one particular device. Contains parametrized commands,

unfortunately not in the protocol/parameter form, but parametrized by an executor and *its* parameters.

RMIR file

Configuration file for a complete remote configuration for RemoteMaster, describing the configuration of a (universal) remote, in general containing one or several device updates. and *its* parameters.

scrutinize (verb)

"To examine in detail with careful or critical attention."

sendir (Global Caché) format

Text format used by Global Caché devices for expressing an IR signal, together with some additional information (number of sends, transmitter). IrScrutinizer can translate to and from this format.

Start sequence

See IrSignal.

StringTemplate

"A Java template engine ... for generating source code, web pages, emails, or any other formatted text output. StringTemplate is particularly good at code generators, multiple site skins, and internationalization / localization." Website, see also Github documentation.

Sub device Number

See protocol parameters. In IRP protocols, denoted by S.

TVS (tab separated values)

Like CSV but using a tab character (ASCII character 9). File extension `.tsv`, or other.

Toggle

Persistent variable in an IrProtocol, in general alternating between 0 and 1, between different invocations. I.e., if the first invocation has the toggle value 0, all even invocations will have the value 1 of the toggle, all even the value 0, independent of the number of repeat sequences. Also see protocol parameters.

Tonto

An open source re-implementation of ProntoEdit for the Pronto Classic, as well as an Java API library for reading and manipulating CCF files. Now discontinued. Author is Stuart Allen. Sources at Github. The API library is used in IrScrutinizer to read from, and export to, CCF files.

Transmitter

Some IR senders have more than one sending channel, called transmitter, allowing for example to control different equipment independently, even if they are using the same commands. These are called transmitters. Note that by definition, every IR sender has at least one transmitter, but only in the case of multiple transmitters, a selection is meaningful.

Transmogrify (verb)

"To change or alter greatly and often with grotesque or humorous effect"

UEI learned format

Proprietary internal format for a single IR signal from Universal Electronics Inc. By convention formatted as a sequence of two-digit hexadecimal numbers. Has been reverse engineered in ExchangeIR. Supported in previous versions of IrScrutinizer, but not in the current. (Rationale.)

wave file format

An IR sequence rendered with halfed modulation frequency, as a sequences of equdistant samples (in general with sample frequency 44.1kHz or 48kHz) considered as an audio signal. It is supposed to be "playbacked" through an audio system connected to a pair of IR LEDs connected in anti-parallel, which will again double the carrier frequency. A common mis-conception is that a stereo signal is used for this. IrScrutinizer supports both the generation of wave files, as well as its import and analysis.

WinLirc

According to its its web site, it "...is the Windows equivalent of Lirc". It is not a port of Lirc, nor does it share any code with Lirc. Appears currently not to be maintained; last release was 0.9.0i released in May 2014. Statements on Lirc in these pages are not necessarily true for WinLirc.

xcf configuration file

Configuration file format for the Pronto Professional line of remotes. Consists of a ZIP file containing one configuration file in XML-format, as well as a number of supplementary icon images. The XML file is very easy to understand (for programmers!), and can contain IR signals in different formats, like CCF format (usable!) and PCF format (encrypted, thus not usable).

XML

Here, a data base file in certain, marked up, text file format.

XML Schema

XML Schema (also called XSD, for "Xml Schema Language") is an XML language for describing the syntax of XML documents. See the Wikipedia article.

XSLT (Extensible Stylesheet Language Transformations)

XSLT is an XML language for transforming XML documents into other XML documents, HTML-pages, or plain text. See the Wikipedia article. The programs here use only XSLT version 1.0.

2.9.2 Appendix. Semantics of the Pronto HEX (CCF) format.

> **Note:**
> There are a few very old guides to the format circulating on the internet. These were written in the previous century, as the subject was not very well understood. Although likely very valuable at the time they were written, however, now they are basicall completely unsuitable. Please do not read, and in particular, do not recommend to others. This appendix contains all needed to know — at least in 2016.

An IR signal in Pronto CCF form consists of a number of 4-digit hexadecimal numbers. For example:

```
0000 006C 0022 0002 015B 00AD 0016 0041 0016 0016 0016 0016
0016 0016 0016 0016 0016 0016 0016 0016 0016 0016 0016 0016
0016 0041 0016 0016 0016 0016 0016 0016 0016 0016 0016 0016
0016 0016 0016 0041 0016 0041 0016 0016 0016 0016 0016 0016
0016 0016 0016 0016 0016 0016 0016 0016 0016 0016 0016 0041
0016 0041 0016 0041 0016 0041 0016 0041 0016 0041 0016 06FB
015B 0057 0016 0E6C
```

The first number, here 0000, denotes the type of the signal. 0000 denotes a raw IR signal with modulation, while 0100 denotes a non-modulated raw IR signal. There are also a small number of other allowed values, denoting signals in protocol/parameter form, notably 5000 for RC5-protocols, 6000 for RC6-protocols, and 900A for NEC1-protocols.

The second number, here 006C, denotes a frequency code. For the frequency *f* in Hertz, this is the number 1000000/(*f**0.241246) expressed as a four-digit hexadecimal number. In the example, 006C corresponds to 1000000/(0x006c * 0.241246) = 38381 Hertz. (It can be conveniently computed by the Time/Frequency Calculator in IrScrutinizer, available under the Tools menu.)

The third and the forth number denote the number of *pairs* (= half the number of durations) in the start- and the repeat sequence respectively. In the example, there are 0x0022 = 34 starting pairs, and 2 repeat pairs.

Next the start- and the repeat-sequences follow; their length being given by the third and the forth number, as per above. The numbers therein are all time durations, the ones with odd numbers on-periods, the other ones off-periods. These are all expressed as multiples of the period time; the inverse value of the frequency given as the second number. For this reason, "frequency" must be a non-zero number also for the non-modulated case, denoted by the first number being 0100. In the example, the fifth number 0x015B denotes an on-period of 0x015B**periodtime* = 347/*f* = 347/38381 = 0.009041 seconds = 9.041 microseconds.

In particular, all sequences start with an on-period and end with an off-period.

In the Pronto representation, there is no way to express an ending sequence.

2.10 HarcHardware

Warning:

Sending undocumented IR commands to your equipment may damage or even destroy it. By using this program, you agree to take the responsibility for possible damages yourself, and not to hold the author responsible.

Date	Description
2014-02-01	Initial version, for version 0.9.0.

Table 1: Revision history

2.10.1 Introduction

This is a collection of classes for accessing hardware for home automation and remote, in particular infrared, control.

It is intended as a support library for application programs, not as a user friendly program aimed at end users. It is currently used by programs like IrScrutinizer and IrMaster.

There are classes for infrared sending and receiving by communicating with IR sending/receiving hardware, for serial communication as well as ethernet communication with sockets or HTTP, classes for implementing LAN beacons and beacon listeners, among others.

It is largely designed upon interfaces, so that application programs can access different hardware, e.g. for IR sending, in a uniform way. See the class `Main` for an example.

2.10.2 Copyright and License

The program, as well as this document, is copyright by myself. My copyright does not extend to the embedded "components" ExchangeIR, DecodeIR. ExchangeIR was written by Graham Dixon and published under GPL3 license. Its Analyze-function has been translated to Java by Bengt Martensson. DecodeIR was originally written by John S. Fine, with later contributions from others. It is free software with undetermined license. IrpMaster is using ANTLR3.4 and depends on the run time functions of ANTLR3, which is free software with BSD license.

The program uses JCommander by Cédric Beust to parse the command line arguments. It is free software with Apache 2 license.

Serial communication is handled by the RXTX library, licensed under the LGPL v 2.1 license.

JSON handling is implemented using the "fast and minimal JSON parser for Java" by Ralf Sernberg, licensed under the Eclipse Eclipse Public License Version 1.0.

The program and its documentation are licensed under the GNU General Public License version 3, making everyone free to use, study, improve, etc., under certain conditions.

2.10.3 API documentation

API documentation, generated by javadoc.

2.11 Infrared4Arduino — Yet another infrared library for the Arduino

Date	Description
2016-05-04	Initial version.

Table 1: Revision history

2.11.1 Introduction

This is yet another infrared library for the Arduino. (Although its name does not start with YA...) It is a major rewrite of Chris Young's IRLib, (GitHub repo), which itself is a major rewrite of a library called *IRremote*, published by Ken Shirriff in his blog, now maintained on GitHub. It uses Michael Dreher's IrWidget (article in German), see also this forum contribution.

The classes `IrWidget` and `IrWidgetAggregating` are based on Michael's code. The classes `IrReceiverSampler` and `IrSenderPwm`, and in particular the file `IRremoteInt.h`, are adapted from Kevin's and Chris' work. The remaining files are almost completely written from scratch, although the influence of Kevin and Chris is gratefully acknowledged.

This work is a low-level library (like IRremote and IRLib) that can be used in other projects, like AGirs, which is an high-level program taking commands interactively from a user or a program through a bi-directional `Stream`. The goal has been to provide a sound, object oriented basis for the fundamental basis, not to provide maximal functionality, the maximal number of protocols supported, or the most complete support of different hardware. A clean design and high readability, without being "too" inefficient, has been the highest priority. Dynamic memory allocation with `new` and `delete` is used extensively. The user who is afraid of this can create his required objects at the start of the run, and keep them. Most classes are immutable. The classes are `const`-correct.

2.11.2 API

2.11.2.1 Types

There are some project specific data typedefs in `InfraredTypes.h`. For durations in microseconds, the data type `microseconds_t` is to be used. If desired/necessary, this can be either `uint16_t` or `uint32_t`. For durations in milliseconds, use the type `millisecons_t`. Likewise, use `frequency_t` for modulation frequency in Hz (*not* kHz as in the IRremote/IRLib).

For "sizes", `size_t`, the standard C type, is used.

Implementation dependent types like `int` are used if and *only if* it is OK for the compiler to select any implementation allowed by the C++ language.

`unsigned int` is used for quantities that can "impossibly" be larger than 65535.

2.11.2.2 IrSequences and IrSignals

An `IrSequence` is a vector of durations, i.e. sequence of interleaving gaps and spaces. It does not contain the modulation frequence. As opposed to IRremote and IRLib,

our sequences always start with a space and end with a gap. It is claimed to be a more relevant representation than the one of IRremote and IRLib.

An `IrSignal` consists of a modulation frequency and three `IrSequence`s: intro-, repeat-, and ending sequence. All of these, but not all, can be empty. If repeat is empty, intro has to be non-empty and ending empty. The intro sequence is always sent first, then comes a zero or more repeat sequences, and finally the ending sequence. To send a signal $n > 0$ times shall mean the following: If the intro is non-empty, send intro, n - 1 repeats, and then the ending. If the intro is empty, send n repeats, and then then ending.

2.11.2.3 Class construction

For some receiving and transmitting classes, multiple instantiations are not sensible, for other it may be. In this library, the classes that should only be instantiated once are implemented as singleton classes, i.e. with no public constructor, but instead a static "factory" method (`newThing()`) that delivers a pointer to a newly constructed instance of `Thing`, provided that `Thing` has not been instantiated before. The classes, where multiple instances is sensible, come with public constructors. (However, the user still has to take responsibility for avoiding pin- and timer-conflicts.)

2.11.2.4 Hardware configuration

For hardware support, the file `IRremoteInt.h` from the IRremote project is used. This means that all hardware that project supports is also supported here (for `IrReceiverSampler` and `IrSenderPwm`). (Actually, a small fix, borrowed from IRLib, was used to support Arduinos with ATMega32U4 (Leonardo, Micro).) However, IrWidgetAggregating is currently supported on the boards Uno/Nano (ATmega328P), Leonardo/Micro (ATmega32U4), and Mega2560 (ATmega2560).

Several of the sending and receiving classes take a GPIO pin as argument to the constructor. However, the sending pin of `IrSenderPwm` and the capture pin of `IrWidgetAggregating` are not configurable, but (due to hardware limitations) have to be taken from the following table:

	Sender Pin	Capture Pin
Uno/Nano (ATmega328P)	3	8
Leonardo/Micro (ATmega32U4)	9	4
Mega2560 (ATmega2560)	9	49

2.11.3 Timeouts

All the receiving classes adhere to the following conventions: When initialized, it waits up to the time `beginningTimeout` for the first on-period. If not received within that period, it returns with a timeout. Otherwise, is starts collecting data. It will collect data until one of the following occurs:

- A silence of length `endingTimeout` has been detected. This is the normal ending. The detected last gap is returned with the data.
- The buffer gets full. Reception stops.

2.11.4 User parameters

As opposed to other infrared libraries, there are no user changeable parameters as CPP symbols. However, the timer configuration is compiled in, depending on the CPP processors given to the compiler, see the file `IRremoteInt.h`.

2.11.5 Files

As opposed to the predecessor projects, this project has a header (`*.h`) file and an implementation file (`*.cpp`, sometimes missing) for each public class.

2.11.6 Error handling

Simple answer: there is none. If a function is sent erroneous data, it just silently ignores the request, or does something else instead. This (unfortunately) seems to be the standard procedure in Arduino programming.

I am used to exception based error handling, for some reason this is not used by the Arduino community.

Constructive suggestions are welcome.

2.11.7 Protocols

Comparing with the predecessor works, this project may look meager, currently supporting only two protocols (NEC1 and RC5). It is planned to generate the corresponding C++ code automatically from the IRP notation. (For this reason, contributed implementations of more protocols are not solicited.)

2.11.8 Sending non-modulated signals.

RF signals (433 MHz and other carrier frequencies) do not use the IR typical modulation. Also there are a few IR protocols (like Revox, Barco, Archer) that do not using modulation. These signals can be sent by the class `IrSenderNonMod`, after connecting suitable hardware capable of sending non-modulated (IR- *or* RF-) signals to the GPIO pin given as argument to the constructor.

2.11.9 Dependencies

This library does not depend on any other libraries; only the standard Arduino environment.

2.11.10 Questions and answers

2.11.10.1 What is the difference between the IrReceiver* and the IrWidget* classes?

They are intended for two different use cases, receiving and capturing. Differently put, "receive" uses a demodulating receiver (TSOPxxx, etc.), "capture" a non-demodulating decoder (TSMPxxx, OPLxxx QSExxx, etc.). Note that this terminology is not universally accepted (yet!).

2.11.11 Coding style

My goal is to write excellent code, even though I do not always succeed :-). "Everything as simple as possible, but not simpler." Cleanliness, logical structure, readability and maintainability are the most important requirements. Efficiency (runtime and/or space) is also important, although it normally comes on second place. [The Arduino Style Guide](https://www.arduino.cc/en/Reference/APIStyleGuide) has different goals (essentially optimizing for novice programmers, "Some of these run counter to professional programming practice"). It is therefore not given priority in this project.

2.11.12 Documentation

The main documentation for the classes is found in the source files themselves. It can be extracted to a browse-able documentation using the program Doxygen. After installing the program, fire up the program in the source directory. It will generate documentation in a subdirectory `apidoc`. To browse, open the just created `apidoc/index.html` in a browser. For convenience, API documentation of the current official version is also found here.

The documentation is written for the *user* of the library, not the developer. For this reason, the file `Arduino.h` has been deliberately excluded from the documentation, to keep it centered on the main issues for the programming on the target system.

2.11.13 Multi platform coding

For someone used to, e.g., Netbeans or Eclipse, the Arduino IDE feels "somewhat" primitive and limited. In particular, it does not support debugging. Mainly for this reason, the code in the present library is designed to compile, and at least to some extent, run in a normal C++ environment on the host compiler. For this, some code modifications, in particular, a customized `Arduino.h` was needed. If the preprocessor symbol `ARDUINO` is defined, just includes the standard Arduino `Arduino.h` is included, otherwise (i.e. for compiling for the host), some more-or-less dummy stuff are defined, allowing compiling for, and execution/debugging on the host.

This way, certain types of problems can be solved much faster. The drawback is that the code is "polluted" with ugly `#ifdef ARDUINO` statements, which decreases readability and makes maintenance harder.

The subdirectory `tests` contains test(s) that run on the host. The supplied `Makefile` is intended for compiling for the host as target. It creates a library in the standard sense (`*.a`), and can be used to build and run tests in subdirectory `tests`.

With the provided `Doxyfile`, Doxygen will document only the (strict) Arduino parts, not the "portable C++".

2.11.14 License

The entire work is licensed under the GPL3 license. Chris' as well as Ken's code is licensed under the LGPL 2.1-license. Michael's code carries the GPL2-license, although he is willing to agree to "or later versions".

2.11.15 Links

The GitHub repository.

API documentation.

2.12 IR signal resources on the Internet — an annotated collection

Date	Description
2016-04-29	Initial version.
2017-03-12	Removed AWE Europe; it is not open to the public any more.

Table 1: Revision history

2.12.1 Introduction

This page collects a number or Internet resources, that I have found usable for myself. There is an emphasis on free and open source resources. It is not by any means attempts to be an exhaustive list. Nevertheless, suggestions for additional entries are welcome.

> **Warning:**
> The different services below may impose their own rules and conditions for their usage. It is the responsibility of the reader to observe these.

> **Warning:**
> The correctness of the data cannot be guaranteed. Sometimes the data is incomplete, unparseable, or outright wrong. Sending undocumented IR signals to your equipment may put it in an unwanted state, or even damage it, possibly even irrevocably. By using this information, you agree to take the responsibility for possible damages yourself, and not to hold the author responsible.

With the exception of GirrLib, I am not connected with any of the sources below. However, I sometimes participate in the JP1 and RemoteCentral forums, and on the Lirc mailing list.

2.12.2 Downloadable data bases

Collections that can be (legally) downloaded in its entirety are listed here.

Name	Format	Home page	Browse	Download links	Direct download	Git Repository clone address	Comment
IRDB	Protocol/ parameters as csv text files.	Github project	Manufacture			https:// github.com/ probonopd/ irdb	Data content for irdb as per below.
Mega List	CML			RemoteCent Beta			Can be imported in IrScrutinizer.
Lirc-remotes	Lircd.conf	Home page	List Source Forge			git:// git.code.sf.n p/lirc-remotes/ code	The entire directory tree be imported in IrScrutinizer.
JP1 master list	Excel text	Forum		Download	Direct DL		Mostly textual information, protocol/ parameters, and index (links) of downloadable files on the JP1 forum. Download only after registration and login at the JP1 forum.
GirrLib	protocol/ parameters, Hex, and/ or Raw,	Github project	Manufacture			https:// github.com/ bengtmarten GirrLib.git	Not intended as a deployment IR data

Name	Format	Home page	Browse	Download links	Direct download	Git Repository clone address	Comment
	packed as Girr.						base, but rather as a collection of Girr examples.

2.12.3 General Services and file collections

Name	Format(s)	Home page	Reg. reqd.	Comment
IRDB	Protocol/ parameters, Hex, UEI, Raw	irdb.tk	No	Also offers rendering and decoding online (powered by the software from this site). Organized after manufacturers/ device-type/ protocol, not as devices. Does not "really" support protocols with other parameters than D (device), S (subdevice), and F (function). Directly supported by IrScrutinizer.
ControlTower	Hex, sendir	Home page	Yes	Non-premium users are allowed to download 5 code sets per day, per mail-only. Not directly supported by IrScrutinizer; use text/raw import of the mailed lists, Parameters name col. = 1, Raw signal col = 2 (or 3), field separator: , (comma), do not enable "Multi col.

Name	Format(s)	Home page	Reg. reqd.	Comment
				name" or "and subseq. columns".
Global Caché Infrared Database (old)	Hex, sendir	Home page	Yes	No longer actively maintained. The successor is ControlTower (previous row). Directly supported in IrScrutinizer.
JP1 lookup	-	Home page	No	Mostly textual information and links to downloadable files in the JP1 repository.
JP1 repository	RMDU and others	Dev. types	Yes	JP1 device upgrades are of limited use outside of the JP1 project, since they refer to executors instead of protocols.
Remotecentral	Mostly proprietary file formats, sometimes Hex for individual commands.	Index	No	Files organized after the remote's hardware. Pronto Classic ccf and Pronto professional xcf (most) can be imported in IrScrutinizer.
Remotecodelist		Index	No	Collection of miscellaneous links and document, mostly manuals for remotes.

2.12.4 Specialized services/data bases

Description	format	Home page	Reg. reqd.	Comment
Rob Humphries Sony Remote Control Codes	textual (protocol/ parameters)	Home page	No	Codes for Sony equipment, from

Description	format	Home page	Reg. reqd.	Comment
				2007. See also this Excel list.
Yamaha master list	protocol/ parameters as PDF	Download page	No	"...master collection of Yamaha IR Hex code files for all types of Yamaha's ...", from 2008.
Pioneer	different	Download server	No	Manufacturer supplied information, mostly not machine parseable.
Oppo	Hex and Protocol/ parameters as Excel files	Oppo homepage, select Customer Service, product, etc.	No	Manufacturer supplied information.
Logitech	Logitech proprietary	Search page	Yes	Only usable with Logitech harmony remotes.

3 Old programs and docs

3.1 Discontinued projects and old articles

Date	Description
2014-02-02	Initial version.
2018-11-13	Moved IrpMaster and IrScrutinizer from current to here.
2022-05-13	Moved Lirc CCF, lirc_rpi and rpi_daughterboard from current to here.

Table 1: Revision history

3.1.1 Content

- IrMaster, a general purpose IR program, now discontinued and superseded by IrScrutinizer.
- IrpMaster, the IR renderer, as a library and command line program. Now discontinued, replaced by IrpTransmogrifier

- HarcToolbox, the old main project, no longer developed. Note that the name "HarcToolbox" denotes both this project, and the web site. Sorry for the confusion.
- Transforming XML Export from Ir(p)Master, a tutorial article on generating "interesting stuff" (here, C code) from the XML export of IrpMaster and IrMaster. It is obsolete, since IrScrutinizer uses a different format (Girr) that is not quite compatible with the very simplistic XML format of IrMaster and IrpMaster. Actually, the XML export of these programs became obsolete before I had the time to document them...
- Lirc2xml, a program for extracting IR codes from LIRC files. This is a patch to Lirc 0.9.0, and produces a command line C program. It is superseded by Jirc, written entirely in pure Java, thus attractive to integrate into other programs, like IrScrutinizer, offering a nice GUI.
- LIRC CCF patch, to make the LIRC server able to send signals not residing on the server.
- Raspberry Pi Improved Lirc driver.
- Raspberry Pi daughterboard of IR and RF sending and receiving.

Downloads.

3.2 IrpMaster: a program and API for the generation of IR signals from IRP notation

> **Note:**
> It may not be necessary to read this document. If your are looking for a user friendly GUI program for generating IR signals etc, please try the program IrScrutinizer (or its predecessor IrMaster), and get back here if (and only if) you want to know the detail on IR signal generation.

3.2.1 Revision history

Date	Description
2011-08-15	Initial version.
2012-04-24	Converted to the document format of Apache Forrest. The program documentation is now generated from that file. Many minor fixes and updates.
2012-06-03	Minor updates for upcoming release 0.2.0.
2012-08-19	Minor updates for upcoming release 0.2.1.
2012-11-18	Minor updates for upcoming release 0.2.2.
2014-01-27	Minor updates for upcoming release 1.0.0.
2014-05-30	Minor updates for upcoming release 1.0.1.
2016-04-29	Minor updates for release 1.2.

3.2.2 Revision notes

Release notes for the current version

3.2.3 Introduction

The "IRP notation" is a domain specific language for describing IR protocols, i.e. ways of mapping a number of parameters to infrared signals. It is a very powerful, slightly cryptic, way of describing IR protocols. In early 2010, Graham Dixon (mathdon in the JP1-Forum) wrote a specification. Up until this program was released, there has not been a usable implementation of the IRP-notation in the sense of a program that takes an IRP Protocol together with parameter values, and produces an IR signal. (The MakeHex program operates on a previous, more restricted version of the IRP notation. The MakeLearned program has severe restrictions, most importantly, its sources are not available.) The present work is a Java program/library that is hoped to fill that gap. It is written in Java 1.6, may or may not run with Java 1.5, but definitely not with earlier Java versions. It, optionally, calls the shared library DecodeIR on Windows, Linux, or Macintosh, but has no other "impurities" in the sense of Java. It can be used as a command line program, or it can be used through its API. For parsing the IRP-Notation, the tool ANTLR is used, generating the parser automatically from the grammar.

This project does not contain a graphical user interface (GUI). See Main principles for a background. Instead, the accompanying program IrScrutinizer (and its predecessor IrMaster) provides a GUI for the present program, among many other things.

For understanding this document, and the program, a basic understanding of IR protocol is assumed. However, the program can be successfully used just by understanding that an "IRP protocol" is a "program" in a particular "domain specific language" for turning a number of parameters into an IR signal, and the present program is a compiler/interpreter of that language. Some parts of this document requires more IRP knowledge, however.

3.2.3.1 Spelling, pronunciation

The mandatory section... :-; Preferred spelling is "IrpMaster", with "I" and "M" capitalized (just as the Java class). Pronounce it any way you like.

3.2.3.2 Synergies within other projects

I hope that this program/library should be useful to other projects involved in IR signals. It makes the vast knowledge of the JP1 project available to other programs. It can be used off-line, to manually or automatically produce e.g. configuration files containing rendered IR signal in some popular format, like the Pronto format. More exciting is to implement a real time "IR engine", that can generate and transmit IR signals in any of the known formats.

3.2.3.3 Copyright and License

The program, as well as this document, is copyright by myself. Of course, it is based upon the IRP documentation, but is to be considered original work. The "database file" IrpProtocols.ini is derived from DecodeIR.html, thus I do not claim copyright.

The program uses, or interfaces with (the different is slightly blurred), other projects. ExchangeIR was written by Graham Dixon and published under GPL3 license. Its Analyze-function has been translated to Java by myself, and is used in by the present program. DecodeIR was originally written by John S. Fine, with later contributions from others. It is free software with undetermined license. IrpMaster depends on the runtime functions of ANTLR3, which is free software with BSD type license.

The program and its documentation are licensed under the GNU General Public License version 3, making everyone free to use, study, improve, etc., under certain conditions.

3.2.4 Main principles

3.2.4.1 Design principles

It is my opinion that it is better to get the functionality and the API right, before you do a graphical user interface (GUI). It is much easier and logical to put a GUI on top of a sane API, then to try to extract API functionality from a program that was never designed sanely but built around the GUI. (Look at WinZip for a good example of the latter. Or almost any Windows program, commercial or freeware...)

I have tried to follow the IRP document as closely as possible, in particular with respect to the grammar and the syntax. However, the Execution model of Chapter 14, turned out not to be usable.

Performance consideration were given minimal priorities. As it stands, rendering a single IR signal typically takes less than 1 ms, so this seems justified. Some debugging statements are covered by functionally superfluous if-statements in order not to have to evaluate arguments (`to String()` etc) not needed anyhow.

Everything that is a "time" is internally represented as a double precision number. Most output formats however, are in some integer format. I here use the principle of sound numerics, do all computations with "high precision" (i.e. double precision) as long as ever possible, then transform to the lower precision form (i.e. integer formats) only in the final step.

All "integer quantities" like expressions, are done in Java long format, 64 bits long, amounting to 63 bits plus sign. Already the `metanec`-example would not work with Java int's. The performance penalty over using `int` (32 bits) is believed to be neglectable.

Differently put, all parameters are limited to Java's long, and can thus be no larger than 2^63-1 = 9223372036854775807. A real-life protocol where this limit is exceeded is not known to me.

Versions prior to 0.2.1 also limited the length of bitfields, also the concatenation of bitfields, to 64. In version 0.2.1 this restriction has been removed, and arbitrary length (concatenation of) bitfields are allowed, as long as the parameters are less than 2^63-1. (Example: The concatenation of bitfields `A:50,B:50` produces a concatenated bitfield of length 100, which is accepted by IrpMaster 0.2.1, but rejected by prior versions.) (Thanx to 3FG for pointing this out to me.)

I do not have the slightest interest in internationalization of the project in its present form — it does not contain a user friendly interface anyhow.

3.2.4.2 Repetitions

Possibly the major difficulty in turning the IRP Specification into programming code was how to make sense of the repetition concept. Most treatises on IR signals (for example the Pronto format) considers an IR signal as an introduction sequence (corresponding to pressing a button on a remote control once), followed by a repeating signal, corresponding to holding down a repeating button. Any, but not both of these may be empty. In a few relatively rare cases, there is also an ending sequence, send after a repeating button has been released. Probably 99% of all IR signals fit into the intro/repetition scheme, allowing ending sequence in addition should leave very few practically used IR signals left. In "abstract" IRP notation, these are of the form A,(B)+,C with A, B, and C being "bare irstreams".

In contrast, the IRP notation in this concept reminds they syntax and semantics of regular expressions: There may be any numbers, and they can even be hierarchical. There certainly does not appear to be a consensus on how this very,... general ... notation should be practically thought of as a generator of IR signals. The following, "finite-automaton interpretation" may make sense: An IRP with several repetitions, say, A(B)+C(D)+E, can be thought of as a remote control reacting on single and double presses. Pressing the key and holding it down produces first A, then B's as long as the button is pressed. An action such as shortly releasing the key and immediately pressing it again then sends one C, and keeps sending D's as long as button is kept pressed. When released, E is sent. Similarly, hierarchical repetitions (repetitions containing other repetitions) may be interpreted with some secondary "key" being pressed and/or released while a "primary button" is being held down — possibly like a shift/meta/control modifier key on a keyboard or a sustain/wah-wah-pedal on a musical instrument?

The present program does not implement hierarchical repetitions. However, an unlimited number of non-hierarchical repetitions are allowed, although not in the form if the class IrSignal — it is restricted to having three parts (intro, repeat, ending). Also, the repetition pattern (...)* is rejected, because it does not make sense as an IR signal.

The command line interpreter contains an "interactive mode", entered by the argument `--interactive`. This way the intrinsic finite state machine (see above) inherent in an IRP with repetitions can be interactively traversed, probably in the context of debugging.

3.2.5 Command line usage

3.2.5.1 Installing binaries

There is no separate binary distributions of IrpMaster. The user who do not want to compile the sources should therefore install the binary distribution of IrScrutinizer, which contains everything needed to run IrpMaster from the command line. Installing that package, either the Windows installer or the ZIP file, will install a wrapper, which is the preferred way to invoke IrpMaster.

3.2.5.2 Usage of the program from the command line

I will next describe how to invoke the program from the command line. Elementary knowledge of command line usage is assumed.

There is a lot of functionality crammed in the command line interface. The usage message of the program gives an extremely brief summary:

```
Usage: one of
        IrpMaster --help
        IrpMaster [--decodeir] [--analyze] [-c|--config <configfilename>] --version
        IrpMaster [OPTIONS] -n|--name <protocolname> [?]
        IrpMaster [OPTIONS] --dump <dumpfilename> [-n|--name <protocolname>]
        IrpMaster [OPTIONS] [--ccf] <CCF-SIGNAL>|<RAW-SEQUENCE>
        IrpMaster [OPTIONS] [--ccf] "<INTRO-SEQUENCE>" ["<REPEAT-SEQUENCE>" ["<ENDING-
SQUENCE>"]]
        IrpMaster [OPTIONS] [-n|--name] <protocolname> [PARAMETERASSIGNMENT]
        IrpMaster [OPTIONS] [-i|--irp] <IRP-Protocol> [PARAMETERASSIGNMENT]

where OPTIONS=--stringtree <filename>,--dot <dotfilename>,--xmlprotocol
 <xmlprotocolfilename>,
-c|--config <configfile>,-d|--debug <debugcode>|?,-s|--seed <seed>,-q|--quiet,
-P|--pass <intro|repeat|ending|all>,--interactive,--decodeir,--analyze,--lirc
 <lircfilename>,
-o|--outfile <outputfilename>, -x|--xml, -I|--ict, -r|--raw, -p|--pronto, -u|--uei,
--disregard-repeat-mins, -#|--repetitions <number_repetitions>.

Any filename can be given as `-', meaning stdin or stdout.
PARAMETERASSIGNMENT is one or more expressions like `name=value' (without spaces!).
One value without name defaults to `F`, two values defaults to `D` and `F`,
three values defaults to `D`, `S`, and `F`, four values to `D`, `S`, `F', and `T`, in
 the order given.

All integer values are nonnegative and can be given in base 10, 16 (prefix `0x'),
8 (leading 0), or 2 (prefix `0b' or `%'). They must be less or equal to 2^63-1 =
 9223372036854775807.

All parameter assignment, both with explicit name and without, can be given as
 intervals,
like `0..255' or '0:255', causing the program to generate all signals within the
 interval.
Also * can be used for parameter intervals, in which case min and max are taken from
the parameterspecs in the (extended) IRP notation.
```

Note that if using the wrapper previously described, it has already added the option `--config` *`standard_confile`* to the command line.

We will next explain this very brief description somewhat more verbosely:

- The first version simply produces the help message, as per above.
- The second version will will print the versions of the program, and optionally, the version of the configuration file and the DecodeIR dynamic library. The `--version` argument should normally be given last, since it is executed immediately when the command line is parsed.
- The third version prints the IRP string of the protocol with the given name to the terminal.
- In the forth version, a CCF string (or, alternatively, a string in raw format (with leading "+"), or in UEI learned format) is read in, and, depending on the to the other options invoked, translated to another format, or sent to DecodeIR and/or AnalyzeIR.
- The fifth version differs form the forth version in that an intro-, and optionally a repeat-, and an ending sequence are explicitly given in raw format, each as a separate argument. In most shells, this means that they have to be enclose within quotes.
- The sixth version dumps either the whole IRP data base, or just the protocol given as argument, to the file name used as the argument to the `--dump` option (use `-` for standard output).
- The sixth version uses the name of an IRP protocol (using the `-n` or `--name` option), to be found in the data base/configuration file specified by the `-c` or `--config` option, that protocol is used to render an IR signal or sequence using the supplied parameters (more on that later).
- Finally, the last version allows the user to enter an explicit IRP string using the `-i` or `--irp`-option, to be used to render the signal according to the parameters given.

In the simplest and most general form, parameter assignments are made on the command line in one argument of the type *`name=value`*. On both sides of the "="-signs, there should not be any spaces. (More precisely, it is required that all assignments are made within a single "argument" to the program, which is determined by the command line interpreter. Thus writing the arguments within single or double quotes, extra spaces can be parsed.) After named parameters are given (possibly none), up to four "standard" parameters can be given. These are, in order D, S, F, and T (which per convention in the JP1 community stands for "Device", "Subdevice", "Function" (also called OBC or command number), and "Toggle"). If using -1 as the value, that parameter is considered as not being assigned. One value without name defaults to `F', two values defaults to `D' and `F', three values defaults to `D', `S', and `F', and four to `D', `S', `F', and `T', in the order given. For example,

```
E=12 34 -1 56 1
```

assigns the value 12 to E, the value 34 to D, the value of 56 to F, and 1 to T, while S is not assigned anything at all. Parameters can be given not only in decimal notation, but

also as hexadecimal (using prefix 0x) binary (using prefix 0b or %), or octal (using prefix 0).

If the command line cannot be parsed the usage message will be printed. If you are unsure of exactly what is wrong, consider issuing "`-d 1`" (the debug option with argument 1) as the first argument on the command line, which may produce more verbose error messages.

Using the `-r` or `--raw` option, the output is given in "raw form" (in JP1-Forum jargon, this is a sequence of positive numbers (indicating "flashes", or on-times in micro seconds) and negative numbers (indicating "gaps" or off-times, where the absolute value indicates the duration in micro seconds. Carrier frequency is specified separately). Alternatively, or additionally, using the `-p` or `--pronto` option, output is produced in the so-called Pronto format, see e.g. this document. This format is popular in several IR using Internet communities, like Promixis (known for their (commercial) products Girder and NetRemote), as well as EventGhost. Optionally, these can be wrapped into an XML skeleton, offering an ideal platform for translating to every other IR format this planet has encountered. If desired, the output of the program is directed to a particular named file using the `-o` *`filename`* or `--output` *`filename`* option. (There is also a possibility (using the `--ict` or `-I` option) to generate output files in IRScope's ict-format, but I am not sure this was as wise design decision: it may be a better idea to generate additional formats by post-processing the XML file.)

Preventing intro sequence in repeat sequence

Motivated by this thread in the JP1 forum, I have been thinking over the "correct" way to render signals of this type `... (...)+`. This is a real issue, to determine the correct behavior when e.g. a program is sent the instruction "send the signal one time", and not an academic question like "keypress shorter than 6ms" or de-bouncing circuitry.

The Pronto notation is normally described as "intro part exactly once, repetition part if and as long as the button is held down". I.e., zero or more times. Therfore, IMHO, the IRP `I (R)+` should properly be rendered as having intro sequence `I R`, which is what IrpMaster normally does. However, in a sense, this can be considered as ugly, awkward, and redundant. If I recall properly, there is a flag in the LIRC configuration called something like "send_repeat_least_once", which should be exactly what we need.

The option called `--disregard-repeat-mins` will make IrpMaster render the intro sequence without repetition part, also in the `... (...)+` case.

3.2.5.3 Iterating over input parameter ranges

Either for generating configuration files for other programs, or for testing, there is a very advanced feature for looping over input parameter sets. For all of the parameters to a protocol, instead of a single value, a set can be given. The program then computes all IR signals/sequences belonging to the Cartesian product of the input parameter sets. There are five types of parameter sets:

1. Of course, there is the singleton set, just consisting of one value
2. There is also a possibility to give some arbitrary values, separated by commas. Actually, the commas even separate sets, in the sense of the current paragraph.
3. An interval, optionally with a stride different from 1, can be given, either as `min..max++increment` or `min:max++increment`, or alternatively, simply as *, which will get the min and max values from the parameter's parameter specs.
4. Also, a set can be given as `a:b<<c`, which has the following semantics: starting with `a`, this is shifted to the left by `c` bits, until `b` has been exceeded (reminding of the left-shift operator `<<` found in languages such as C).
5. Finally, `a:b#c` generates c pseudo random numbers between a and b (inclusive). The "pseudo random" numbers are completely deterministically determined from the seed, optionally given with the `--seed` option. As of version 0.2.2 `a` and `b>` are optional. If left out, the values are taken as from the protocol parameters min and max respectively, just as with the `*` form.

See the test file `test.sh` (include in the distributions) for some examples. Of course, using the command line, some of the involved characters, most notably the *, has a meaning to the command line interpreter and may need "escaping" by a backslash character, or double or single quotes.

There is also an option, denoted -# or `--repetitions` taking an integer argument, that will compute that many "copies" of the IR signal or sequence. This may be of interest for signals that are non-constant (toggles being the simplest example) or for profiling the program.

3.2.5.4 Debugging possibilities

There are a number of different debug parameters available. Use `-d` or `--debug` with "?" as argument for a listing:

```
$ java -jar IrpMaster.jar --debug ?
Debug options: Main=1, Configfile=2, IrpParser=4, ASTParser=8, NameEngine=16,
 BitFields=32, Parameters=64, Expressions=128,
IrSignals=256, IrStreamItems=512, BitSpec=1024, DecodeIR=2048, IrStreams=4096,
BitStream=8192, Evaluate=16384
```

For every debug option, there is an integer of the form 2^n associated with it. Just add the desired numbers together and use as argument for the `-d` or `--debug` command. There are also commands for debugging the parsed version of the IRP: Notably the `--stringtree` *`filename`* option (produces a LISP-like parsed representation of the so-called AST (abstract syntax tree)). `--dot`*`filename`* produces a dot-file, that can be translated by the open-source program `dot` contained in the Graphviz project, producing a nice picture (e.g. in any common bitmap format) of the current IRP protocol, and `--xmlprotocol` *`filename`* producing an XML representation. It may be possible in the future to use any of these representations to e.g., write a C code generator for a particular protocol.

Some of the classes contain their own main methods (for those not familiar with the Java jargon: these can be called as programs on their own) allowing for both debugging and pedagogical exploration, together possibly with other possibilities. In particular, this goes for the Expression class, One day I am going to document this...

```
java -classpath IrpMaster.jar org.harctoolbox.IrpMaster.Expression -d  'a + b
 *c**#d' {a=12,b=34,c=56,d=4}
(+ a (* b (** c (BITCOUNT d))))
1916
```

3.2.5.5 Third-party Java archives (jars)

For the DecodeIR-integration, IrpMaster requires a small support package, DecodeIR.jar, which is distributed together with IrpMaster. It consists of the compiled `DecodeIRCaller.java` from DecodeIR (full name `com.hifiremote.decodeir.DecodeIRCaller.class`), and `com.hifiremote.LibraryLoader.class` from RemoteMaster, which is also free software. To get rid of some (in this context) annoying messages, it was necessary to create a (very) lightly modified version, which can be found on the download page. IrpMaster also requires the runtime libraries of the parser generator ANTLR, which is also free software but licensed under a BSD-License. I distribute the whole (binary) package `antlr-3.4-complete.jar`. (No usable "runtime version" is known to me.)

3.2.6 Extensions to, and deviation from, IRP semantic and syntax

3.2.6.1 Parameter Specifications

In the first, now obsolete, version of the IRP notation the parameters of a protocol had to be declared with the allowed max- and min-value. This is not present in the current specification. I have reinvented this, using the name parameter_spec. For example, the well known NEC1 protocol, the Parameter Spec reads: `[D:0..255,S:0..255=255-D,F:0..255]`. (D, S, and F have the semantics of device, sub-device, and function or command number.) This defines the three variables D, S, and F, having the allowed domain the integers between 0 and 255. D and F must be given, however, S has a default value that is used if the user does not supply a value. The software requires that the values without default values are actually given, and within the stated limits. If, and only if, the parameter specs is incomplete, there may occur run-time errors concerning not assigned values. It is the duty of the IRP author to ensure that all variables that are referenced within the main body of the IRP are defined either within the parameter specs, defined with "definitions" (Chapter 10 of the specification), or assigned in assignments before usage, otherwise a run-time error will occur (technically an `UnassignedException` will be thrown).

The preferred ordering of the parameters is: D, S (if present), F, T (if present), then the rest in alphabetical order,

The formal syntax is as follows, where the meaning of the '@' will be explained in the following section:

```
parameter_specs:
 '[' parameter_spec (',' parameter_spec )* ']' | '['  ']'

parameter_spec:
 name ':' number '.' '.' h=number ('=' i=bare_expression)?
 | name '@' ':' number '.' '.' number '=' bare_expression
```

3.2.6.2 The GeneralSpec

For the implementation, I allow the four parts (three in the original specification) to be given in any order, if at all, but I do not disallow multiple occurrences — it is quite hard to implement cleanly and simply not worth it. (For example, ANTLR does not implement exclusions. The only language/grammar I know with that property is SGML, which is probably one of the reasons why it was considered so difficult (in comparison to XML) to write a complete parser.)

Persistency of variables

Graham, in the specification and in following forum contributions, appears to consider all variables in a IRP description as intrinsically persistent: They do not need explicit initialization, if they are not, they are initialized to an undefined, random value. This may be a reasonable model for a particular physical remote control, however, from a scientific standpoint it is less attractive. I have a way of denoting a variable, typically a toggle of some sort, as persistent by appending an "@" to its name in the parameter specs. An initial value (with syntax as default value) is here mandatory. It is set to its initial value by the constructor of the Protocol class. Calling the renderIrSignal(...) function or such of the Protocol instance typically updates the value (as given in an assignment, a 0-1 toggle goes like T=1-T). As opposed to variables that has not been declared as persistent, it (normally) retains its value between the invocations of renderIrSignal(...). A toggle is typically declared as `[T@:0..1=0]` in the parameter specs.

Comments and line breaks

Comments in the C syntax (starting with `/*` and ended by `*/`) are allowed and ignored. Line breaks can be embedded within an IRP string by "escaping" the line break by a backslash

Data types

The IRP documentation clearly states that the carrier frequency is a real number, while everything else is integers. Unfortunately, users of the IRP notation, for example in the DecodeIR.html document, has freely used decimal, non-integer numbers. I have implemented the following convention: Everything that has a unit (second or Hz), durations and frequency, are real numbers (in the code double precision numbers).

Extents

The specification writes ``*An extent has a scope which consists of a consecutive range of items that immediately precede the extent in the order of transmission in the signal. ... The precise scope of an extent has to be defined in the context in which it is used.*", and, to my best knowledge, nothing more. I consider it as specification hole. I have, starting with IrpMaster 0.2.2, implemented the following: Every extend encountered resets the duration count.

Multiple definitions allowed

It turned out that the preprocessing/inheritance concept necessitated allowing several definition objects. These are simply evaluated in the order they are encountered, possibly overwriting previous content.

Names

Previous programs (makehex, makelearned) have only allowed one-letter names. However, in DecodeIR.html there are some multi-letter names. The IRP documentation allows multi-letter names, using only capital letters. I have, admittedly somewhat arbitrarily, extended it to the C-name syntax: Letters (both upper and lower cases) and digits allowed, starting with letter. Underscore "_" counts as letter. Case is significant.

Also there are a few predefined, read-only variables, mainly for debugging, although a practical use is not excluded. To distinguish from the normal, and not to cause name collision, they start by a dollar sign. Presently, these are: `$count` (numbers the call to a `render*-()`-function, after the constructor has been called), `$pass`(Requested pass in a `--pass`-argument, (or from API call), not to be confused with the following), `$state` (current state (intro=0, repeat=1, ending=2,...) of parsing of an IRP), `$final_state` (undefined until the final state has been reached, then the number of the final state). For example, the OrtekMCE example `{...}<...>([P=0][P=1][P=2],4,-1,D:5,P:2,F:6,C:4,-48m)+[...]` could be written with `$state` as `(4,-1,D:5,$state:2,F:6,C:4,-48m)+` (disregarding last frame).

GeneralSpecs, duty cycle

Without any very good reason, I allow a duty cycle in percent to be given within the GeneralSpec, for example as `{37k,123,msb,33%}`. It is currently not used for anything, but preserved through the processing and can be retrieved using API-functions. If some, possibly future, hardware needs it, it is there.

Namespaces

There is a difference in between the IRP documentation and the implementation of the Makehex program, in that the former has one name space for both *assignments* and *definitions*, while the latter has two different name spaces. IrpMaster has one name space, as in the documentation. (This is implemented with the `NameEngine` class.)

Shift operators (not currently implemented)

It has sometimes been suggested (see this thread) to introduce the shift operators "<<" and ">>" with syntax and semantics as in C. This far, I have not done so, but I estimate that it would be a very simple addition. (The reader might like to have a look at my example, which possibly would have been more naturally expressed with left shifts than with multiplication with powers of two.)

Logical operators (also not implemented)

In particular in the light of current discussion on the F12 protocol, in my opinion more useful would be the logical operators `&&`, `||`, and `?:`, having their short circuiting semantics, like in languages such as C, Perl,..., but unless, e.g. Pascal. Recall, the expression `A && B` is evaluated as follows: First `A` is checked for being 0 or not. If 0, then 0 is returned, without even evaluating `B`. If however, `A` is nonzero, `B` is evaluated, possibly to a "funny" type and is returned. The F12 protocol (cf. the latest version 2.43 of DecodeIR.html) could then probably be written like `<...>(introsequence, (H && repetitionsequence*))` or `<...>(H ? longsequence+ : shortsequence)`.

BitCount Function

Generally, I think you should be very reluctant to add "nice features" to something like IRP. However, in the applications in DecodeIR.html, the phrase "number of ones", often modulo 2 ("parity"), occurs frequently in the more complicated protocols. This is awkward and error prone to implement using expressions, for example: `F:1 + F:1:1 + F:1:2 + F:1:3 + F:1:4 + F:1:5 + F:1:6 + F:1:7`. Instead, I have introduced the BitCount function, denoted by "#". Thus, odd parity of `F` will be `#F%1`, even parity `1-#F%2`. It is implemented by translating to the Java Long.bitCount-function.

3.2.6.3 Preprocessing and inheritance

Reading through the protocols in DecodeIR.html, the reader is struck by the observation that there are a few general abstract "families", and many concrete protocol are "special cases". For example all the variants of the NEC* protocols, the Kaseikyo-protocols, or the rc6-families. Would it not be elegant, theoretically as well as practically, to be able to express this, for example as a kind of inheritance, or sub-classing?

For a problem like this, it is easily suggested to invoke a general purpose macro preprocessor, like the C preprocessor or m4. I have successfully resisted that temptation, and am instead offering the following solution: If the IRP notation does not start with "{" (as they all have to do to confirm with the specification), the string up until the first "{" is taken as an "ancestor protocol", that has hopefully been defined at some other place in the configuration file. Its name is replaced by its IRP string, with a possible parameter spec removed — parameter specs are not sensible to inherit. The process is then repeated up until, currently, 5 times.

The preprocessing takes place in the class IrpMaster, in its role as data base manager for IRP protocols.

Example

This shows excepts from a virtual configuration file. Let us define the "abstract" protocol `metanec` by

```
[protocol]
name=metanec
irp={38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)[A:0..4294967295]
```

having an unspecified 32 bit payload, to be subdivided by its "inherited protocols". Now we can define, for example, the NEC1 protocol as

```
[protocol]
name=NEC1
irp=metanec{A = D | 2**8*S | 2**16*F | 2**24*(~F:8)}[D:0..255,S:0..255=255-D,F:0..255]
```

As can be seen, this definition does nothing else than to stuff the unstructured payload with D, S, and F, and to supply a corresponding parameter spec. The IrpMaster class replaces "`metanec`" by `{38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)`" (note that the parameter spec was stripped), resulting in an IRP string corresponding to the familiar NEC1 protocol. Also, the "Apple protocol" can now be formulated as

```
[protocol]
name=Apple
irp=metanec{A=D | 2**8*S | 2**16*C:1 | 2**17*F | 2**24*PairID} \
{C=1-(#F+#PairID)%2,S=135} \
[D:0..255=238,F:0..127,PairID:0..255]
```

The design is not cast in iron, and I am open to suggestions for improvements. For example, it seems reasonable that protocols that only differ in carrier frequency should be possible to express in a concise manner.

3.2.6.4 The Configuration file/IRP protocol database

There is presently not a "official" IRP database. MakeHex comes with a number of protocol files with the `.irp`-extension, but that is another, obsolete and much less powerful format. MakeLeaned also comes with a number of "irp-files", in the new format, but incomplete. The DecodeIR.html-file presently comes closest: it has a number (upper two-digit) of IRPs, however, often not even syntactically confirming to the specification, and often with the description of the protocol at least partially in prose ("C is the number of ..."), parseable only by humans, not by programs.

Possibly as an intermediate solution, I invented the `IrpProtocols.ini` file. This file has a format similar to ini-files under Windows. For every protocol, it contains name and

an IRP-string, possibly also a documentation string. The latter can, in principle, contain HTML elements, i.e. it can be an HTML fragment.

3.2.6.5 Syntax and semantics of the IrpProtocols.ini file

Every protocol is described in a section starting with the key `[protocol]`. Then there are a few keywords describing different properties:

- `name` The name of the protocol. This is folded to lowercase for searches and comparisons.
- `irp` The IRP string representation. This may continue over several lines if the line feeds are escaped by a backslash ("\"), i.e. having the backspace as last character on the line.

Other keywords are allowed, but ignored. Then, optionally, there may be a section `[documentation]`, that, in principle, could contain e.g. an HTML-fragment. The documentation section continues until the next `[protocol]` is encountered.

3.2.6.6 Requirements for an IRP data base

I have created the present `IrpProtocols.ini` by hand editing the DecodeIR.html-file. I would welcome if the community can settle for one endorsed format for such a data base. It can be one file, or many files: One file per protocol is easier for the developer, in particular if several developers are using a version management system (with or without file locking), but less convenient for the user.

It would be highly desirable in the future to be able just to maintain one file (or set of files). Some possibilities for this are:

1. Have one master file, for example in XML format, that after preprocessing *generates* both DecodeIR.html, and a protocol description file. There is also the possibility of having a program like IrpMaster parsing the master file directly.
2. Extend `protocol.ini`("belonging to RemoteMaster") with the IRP information. Leaves the problem of duplicated "documentation" between DecodeIR.html and protocols.ini.
3. Formalizing the IRP-Strings within DecodeIR.html, e.g. by using div or span elements with class-attributes, (and formatting with, for example, better CSS style sheets) so that the IRP information can be unambiguously read out.

3.2.6.7 Integration with DecodeIR

Optionally (when installed and selected with the `--decodeir` option) the computed IR signal is sent to DecodeIR, to check DecodeIR's opinion on the nature of the signal. This gives a magnificent possibility for automated tests, not only of the present program, but also of DecodeIR. Note in particular that there are very advance possibilities for testing not only a single signal, but for testing whole ranges of signals, a list of signals, "random"

inputs, equidistant inputs, or inputs achieved by shifting, see the section on parameter iterating.

The shared library is sought first in architecture dependent sub-directories, like in RemoteMaster, `.\windows` on Windows, `./Linux-amd64` and `./Linux-i386` on 64- and 32-bit Linux respectively, etc, then in system libraries, for example given on the command line to the Java VM, using the `-Djava.library.path=` option.

There is some fairly hairy programming in `DecodeIR.java` for identifying some different cases.

The enclosed script `test.sh` runs under a Unix/Linux shell such as bash or sh. It should also run within Cygwin on Windows. It does not run with the standard Windows command line interpreter. Note that it might need some adjustment of file paths etc.

Possibly because I did not find any more logical way to dispose it, the current distribution contains a class (with main()-method) named EvaluateLog that can be used to evaluate the output of the above script. Use like

```
java -classpath IrpMaster.jar IrpMaster/EvaluateLog protocols.log
```

3.2.7 The API

The Java programmer can access the functionality through a number of API functions.

The class `IrpMaster` is the data base manager. The class is immutable, constructed from a file name (or an InputStream), and can deliver assorted pieces of information from the data base. Most interesting is the `newProtocol()`-function that generates a Protocol-object from parsing the IRP-string associated with the requested protocol name. It contains a very elaborate `main()`-function for command line use -- strictly speaking this is "the program" that is described herein. Actually, that `main()`-function does not necessarily belong to the `IrpMaster` class, but could be located somewhere else.

Instances of the Protocol class are constructed (essentially) from a String, containing the IRP representation to be parsed. Once constructed (and IRP-String parsed), the Protocol instances can render IrSignals and IrSequences for many different parameter values. This is done with the `render(...)` and `renderIrSignal(...)` functions, producing `IrSequences` and `IrSignals` respectively:

An `IrSequence` is a sequence of pulse pairs. It does not know weather it is supposed to repeat or not. In contrast, an `IrSignal` has one introductory `IrSequence`, one repetition `IrSequence` (either, but not both, of these can be empty), and an (in most cases empty) ending `IrSequence`.

The API is documented in standard Javadoc style, which can be installed from the source package, just like any other Java package. For the convenience of the reader, the Javadoc API documentation is also available here.

3.2.7.1 Example of API usage

The task is to write a command line program, taking, in order, the configuration file name, a protocol name, a device number and a function/command/obc number, and send the corresponding IR signal to a GlobalCaché GC-100-06 networked IR-transmitter, having IP address 192.168.1.70, using its IR Port 1. For this, we use the GlobalCaché functionality of the HarcHardware, which is also GPL-software written by myself. This task is solved with essentially just a few lines of code.

3.2.8 References

1. IrScrutinizer. A program, also by myself, than, among other things, provides a user friendly GUI for IrpMaster.
2. IrMaster. A program, also by myself, than, among other things, provides a user friendly GUI for IrpMaster.
3. Specification of IRP Notation, Graham Dixon. Also in PDF version for download. A very thorough specification.
4. Discussion thread on the IRP documentation
5. DecodeIR.html. (The link points to a slightly nicer formatted wiki page, though). Contained within the current distribution of DecodeIR. Subversion repository.
6. Makehex. Source, binary. A functional predecessor of the present program. Operates on a predecessor of the current version of the IRP. Written in C++, also available as DLL (within the first link). Java translation by myself.
7. MakeLeaned. Windows, binary only, source unavailable. GUI only, no API. Not maintained since 2005. Almost certainly incomplete with respect to current IRP specification. Discussion thread in JP1-Forum.

3.3 IrMaster documentation

Note:

Development of this program has been discontinued. It has been superseeded by IrScrutinizer, offering much more functionality.

Warning:

Sending undocumented IR commands to your equipment may damage or even destroy it. By using this program, you agree to take the responsibility for possible damages yourself, and not to hold the author responsible.

3.3.1 Revision history

Date	Description
2011-10-23	Initial version.
2012-04-14	Many minor fixes and updates for the upcoming 0.1.2. (version not published)

Date	Description
2012-04-24	Converted to the document format of Apache Forrest. The program documentation is now generated from that file.
2012-06-06	Updated for upcoming version 0.2.0. Many minor impovements. Plotting and Audio new.
2012-08-19	(Not) updated for upcoming version 0.3.0.
2012-11-18	Updated for upcoming version 0.3.1.
2014-01-27	Updated for upcoming version 1.0.0.

3.3.2 Introduction

This is what the program can do: From a data base of known IR signals "recepies" (known as the IRP-notation, essentially corresponding to the collected know-how of the community in machine readable form), IR signals corresponding to certain parameter values can be computed. Export files in different formats can be generated, for usage of other programs. For this, two alternative renders (my own IrpMaster as well as the older Makehex) are available. By using the clipboard, IR signals in Pronto format (for example from Internet articles) can be directly sent to the analyzers AnalyzeIR and DecodeIR. An entered or computed signal can be sent proper hardware, or plotted. For investigating possible non-documented IR signals of owned equipment, a "war dialer" can send whole parameter regions of IR signals. For the latter possibilities, hardware support in the form of GlobalCaché, IRTrans, a LIRC server, or an IR-audio setup, is required. A simple calculator intended for simple computations on IR signal timings is provided.

This program is not a "program with a GUI", nor is it a GUI for a particular program. Rather, it is a "Meta program", offering a GUI for a number of IR related programs, presently IrpMaster (advanced IR rendering program by myself), Makehex (older IR rendering program), DecodeIR (tries to identify an IR signal), and AnalyzeIR (which is my name of the Analyze-Function of the ExchangeIR-Library), and the PtPlot library used for plotting of IR signals. Future extensions to other, possibly not yet written, programs are possible.

Note that I have written two different programs with quite similar names: IrMaster, the present one, a GUI, and Ir**p**Master, a library and a command line program, an IR signal render, but without a GUI. Please do not confuse.

In the sequel, the word "the program" will denote either the "shell" IrMaster, or the GUI together with its "clients" IrpMaster, Makehex, AnalyzeIR, and DecodeIR, as is hopefully clear from the context.

For someone with knowledge in the problem domain of IR signals and their parameterization, this program is believed to be very simple to use. This knowledge is

assumed from the reader. Other can acquire that knowledge either from the JP1 Wiki or, e.g., this link.

Note that the screen shots are included as illustrations only; they may not depict the current program completely accurately. They come from different versions of the program, using different platforms (Linux and Windows), and using different "look and feels".

The present document is written more for completeness than for easy accessability. Possibly, in the future, there will be a user's manual as well as a reference manual.

Release notes for the current version.

3.3.2.1 Copyright and License

The program, as well as this document, is copyright by myself. My copyright does not extend to the embedded "components" Analyze, Makehex, DecodeIR, and PtPlot. Makehex was written by John S. Fine (see LICENSE_makehex.txt), and has been translated to Java by Bengt Martensson. ExchangeIR was written by Graham Dixon and published under GPL3 license. Its Analyze-function has been translated to Java by Bengt Martensson. DecodeIR was originally written by John S. Fine, with later contributions from others. It is free software with undetermined license. PtPlot is a part of the Ptolemy Project at the EECS department at UC Berkeley, licensed under the UC Berkeley copyright. IrpMaster is using ANTLR3.4 and depends on the run time functions of ANTLR3, which is free software with BSD license.

The "database file" IrpProtocols.ini is derived from DecodeIR.html, thus I do not claim copyright.

The `main()` method of the Wave class uses JCommander by Cédric Beust to parse the command line argument. (Likely, I will use it much more in the future.) It is free software with Apache 2 license.

Icons by Everaldo Coelho from the Crystal project are used; these are released under the LGPL license.

The Windows installer was built with Inno Setup, which is free software by Jordan Russel. To modify the user's path in Windows, the Inno extension modpath by Jared Breland, distributed under the GNU Lesser General Public License (LGPL), version 3.

The program and its documentation are licensed under the GNU General Public License version 3, making everyone free to use, study, improve, etc., under certain conditions.

3.3.2.2 Privacy note

Some functions (Help -> Project Homepage, Help -> IRP Notation Spec, Help -> Protocol Specs, Tools -> Check for updates) access the Internet using standard http calls. This causes the orginating machine's IP-address, time and date, the used browser, and possibly other information to be stored on the called server. If this is a concern for you,

please do not use this (non-critical) functionallity (or block your computer's internet access).

3.3.3 Installation

3.3.3.1 General

IrMaster, and all but one of its third-party additions, are written in Java, which means that it should run on every computer with a modern Java installed; Windows, Linux, Macintosh, etc. Java 1.6 or later is required. The one exception is DecodeIR, which is written in C++, and invoked as a shared library (`.dll` in Windows, `.so` in Linux, etc). If DecodeIR is not available on your platform it is not a major problem, as IrMaster will work fine without it; just the DecodeIR-related functions will be unavailable.

There is unfortunately no good `make install` or such in the source distribution, so also source code distribution users are recommended to install the binary distribution. Also, all necessary third-party components are included in the binary distribution.

Both under Windows as well as under other operating systems, IrMaster (and IrpMaster) behave civilized, in that they do not write in the installation directory after the initial installation. In both cases (in contrast to the source distribution), the distribution contains everything needed including third party libraries like DecodeIR, AnalyzeIR, MakeHex (Java version) and its irp-files.

Under Windows, the properties are stored in `%LOCALAPPDATA%\IrMaster\IrMaster.properties.xml` using Windows Vista and later (on my Windows 7 system, this is `%HOME%\AppData\Local\IrMaster\IrMaster.properties.xml`), otherwise in `%APPDATA%\IrMaster\IrMaster.properties.xml`. Using other operating systems, it is stored under `$HOME/.irmaster.properties.xml`. It is not deleted by uninstall. (If weird problems appear when updating, try deleting this file.)

3.3.3.2 Windows

Download the Window setup file, save, and double click. Accept the license. Select any installation directory you like; suggested is `C:\Program Files\IrMaster`. Unless reason to do so, create the start menu folder, the desktop icon, and allow the program to add the application directory to your path (for IrpMaster as command line program). Administrator rights are probably needed, at least if you are installing in a directory like `Program Files`. (The *should* not be needed otherwise, but Windows Vista and later are always good for a surprise...) IrMaster can now be started from `Start -> IrMaster -> IrMaster`, or from the desktop icon.

To uninstall, select the uninstall option from the Start menu. Very pedantic people may like to delete the properties file too, see above.

3.3.3.3 MacOSX

Download and double click the binary distribution. Unpack it to a directory of your choice, e.g. on the desktop. Just double clicking the file IrMaster.jar should now start the program. Otherwise, try the "Other systems" instructions and adapt the wrapper irmaster.sh. This also includes invoking as command line program "IrpMaster".

3.3.3.4 Other systems (Linux etc)

For some reason, double clicking an executable jar file in my Gnome installation does not start the program, but starts a browser for the jar file (which is really a form of Zip-Archieve). Instead:

Create an installation directory (suggestion; `/usr/local/irmaster`), and unpack the current binary distribution therein. Examine the wrapper `irmaster.sh`, and, if desired, make desired changes to it with your favorite text editor. Then make two symbolic links from a directory in the path (suggestion; `/usr/local/bin` to the newly installed `irmaster.sh`, using the names `irmaster` and `irpmaster`. Example (using the suggested directories)

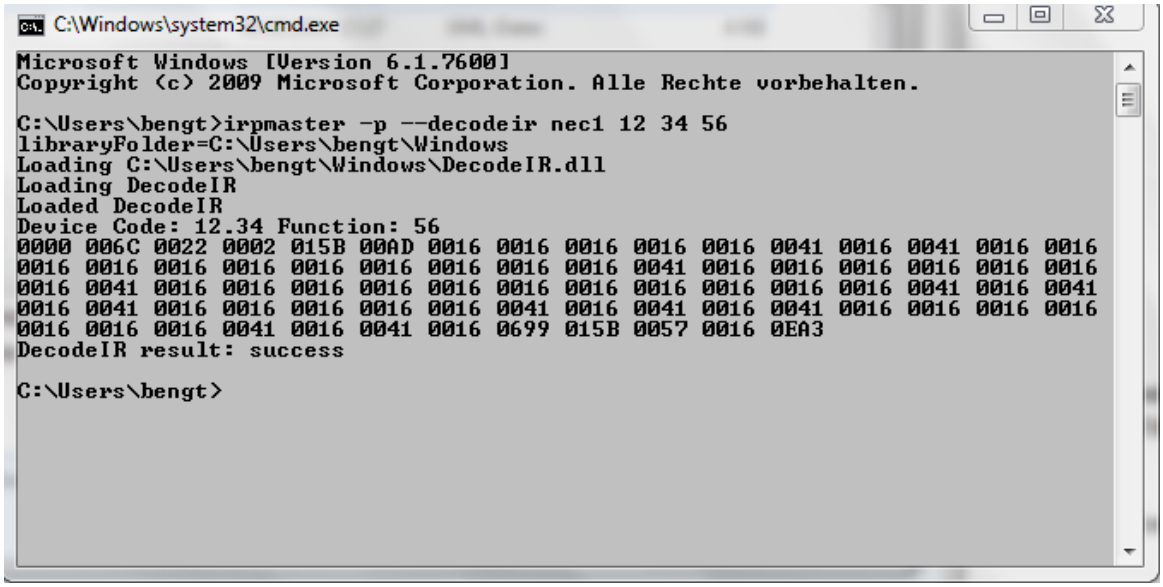
```
cd /usr/local/bin
ln -s ../irmaster/irmaster.sh irmaster
ln -s ../irmaster/irmaster.sh irpmaster
```

(`su` (or `sudo`) may be necessary to install in the desired locations.)

To uninstall, just delete the files. Very pedantic people may like to delete the properties file too, see above.

3.3.3.5 Wrapper for IRPMaster

Both under Windows and Unix-like systems, a wrapper for the command line program IrpMaster is installed. The user can simply open a command window (called anything like "xterm", "Terminal", "Shell", "DOS-Box", "cmd",...) and in that command window can call the program IrpMaster by simply typing `irpmaster`, followed by its arguments. See the screen shot below, that shows the generation of the Pronto form of a NEC1 signal for device 12, subdevice 34, command number 56, together with subsequent invocation of DecodeIR on the computed result. (The output on non-windows system is entirely similar.)



3.3.4 Usage

As stated previously, for anyone familiar with the problem domain, this program is believed to be easy to use. Almost all user interface elements have toolhelp texts. In what follows, we will not attempt to explain every detail of the user interface, but rather concentrate on the concepts. Furthermore, it is possible that new elements and functionality has been implemented since the documentation was written.

This program does not disturb the user with a number of annoying, often modal, pop ups, but directs errors, warnings, and status outputs to the *console window*, taking up the lower third of the main window. Starting with version 0.2.0, this window is resizeable. There is a context menu for the console, accessible by pressing the right mouse button in it.

In the upper row, there are four pull-down menus, named File, Edit, Options, and Help. Their usage is believed to be self explanatory, with the exception of the entries in the Options menu. The latter mimics the Options subpane, and are explained later.

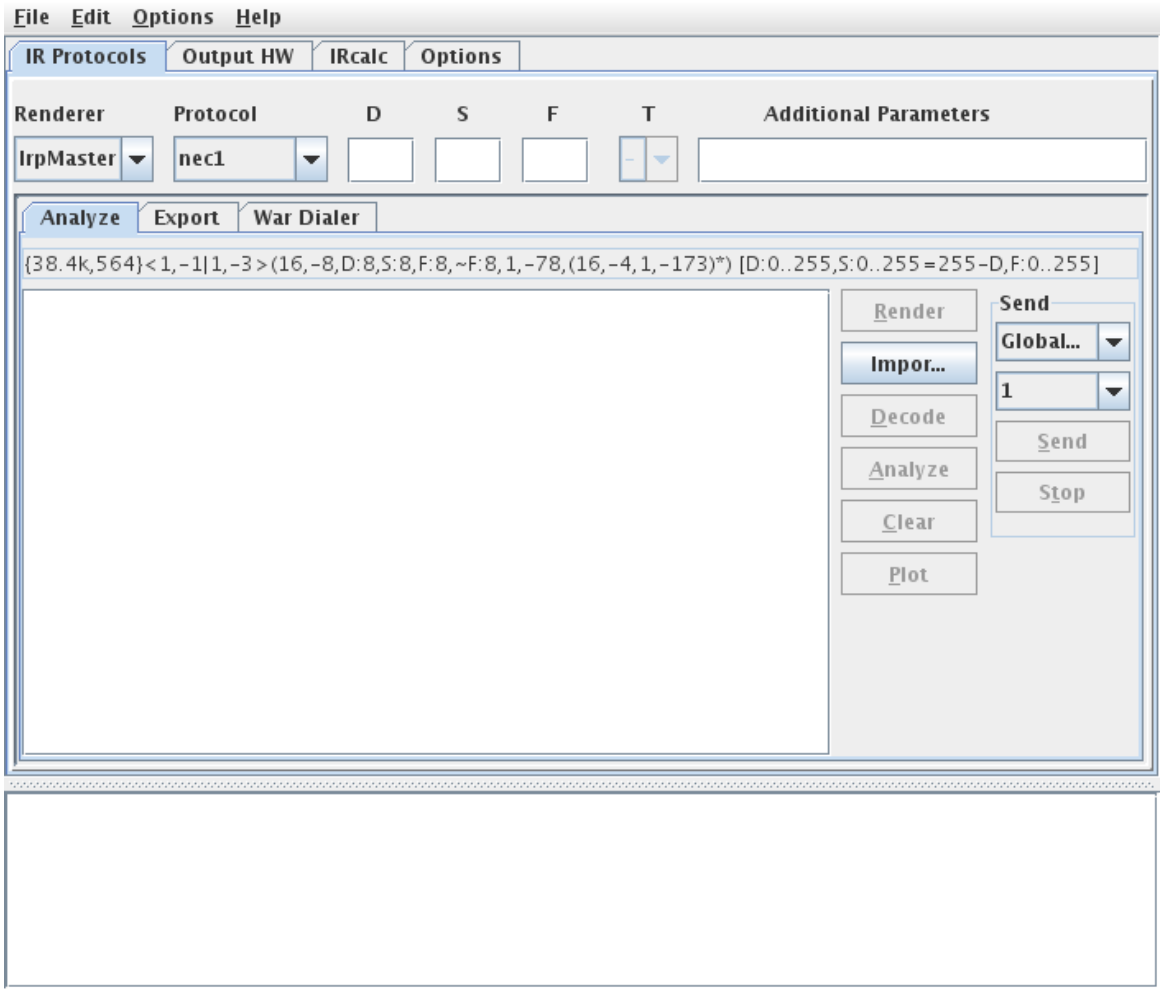
The main window is composed of presently two sub panes denoted by "IR Protocols" and "Hardware" respectively. These will be discussed now.

3.3.4.1 The "IR Protocols" pane

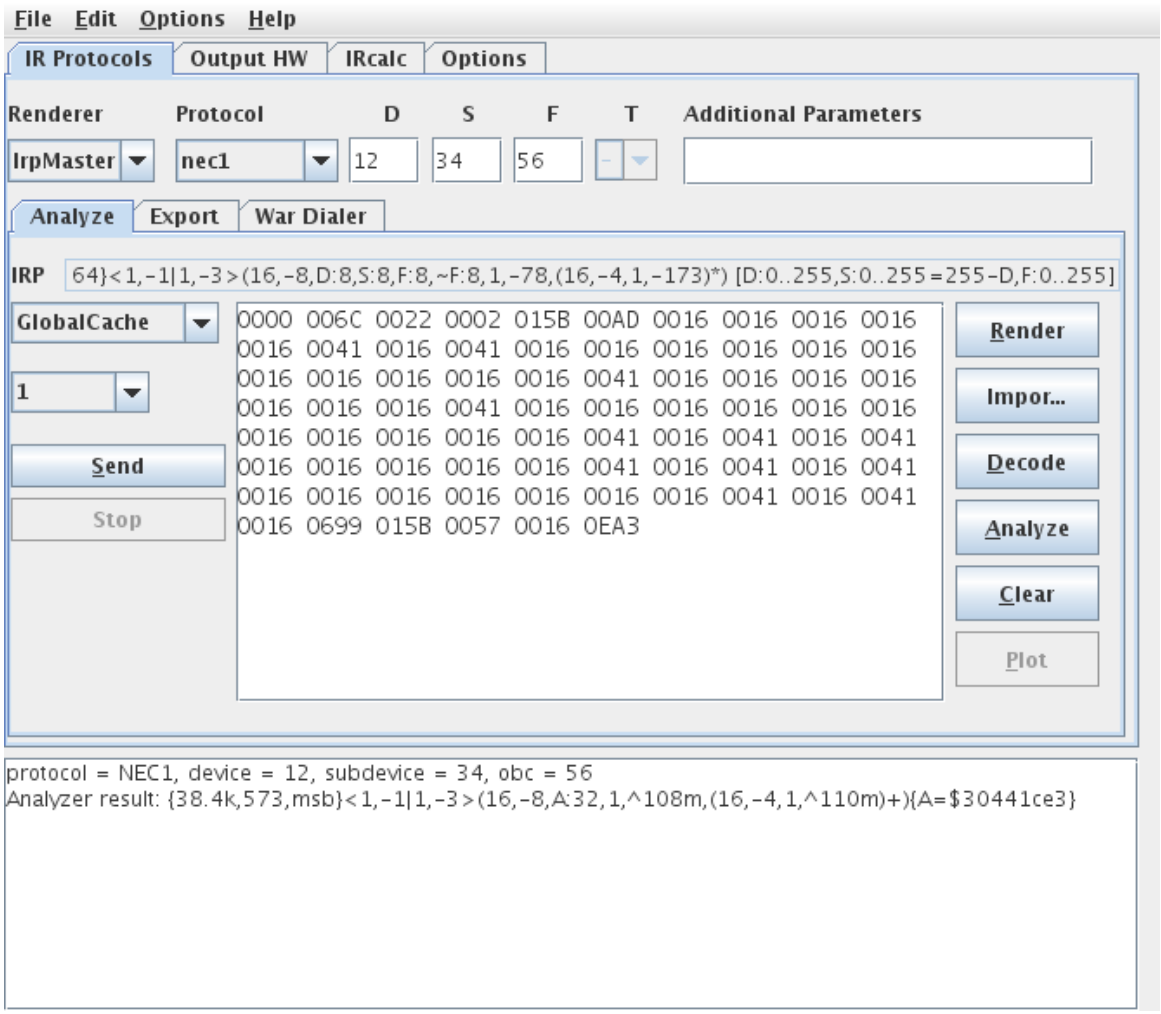
In the upper third of this pane, a render program (IrpMaster or Makehex) can be selected, together with the IR protocol identified by name, and the parameters D ("device", in almost all protocols), S ("sub-device", not in all protocols), F ("function", also called command number or obc, present in almost all protocols), as well as "T", toggle (in general 0 or 1, only in a few protocols). These number can be entered as decimal numbers, or, by prepending "0x", as hexadecimal numbers. Note that the supported protocols are different between the different rendering engines. Not all possibilities of IrMaster are available when using the simpler render Makehex.

The lower two thirds of the window is occupied by another pane setup, consisting of the sub-panes "Analyze", "Export", and "War dialer".

The "Analyze" pane

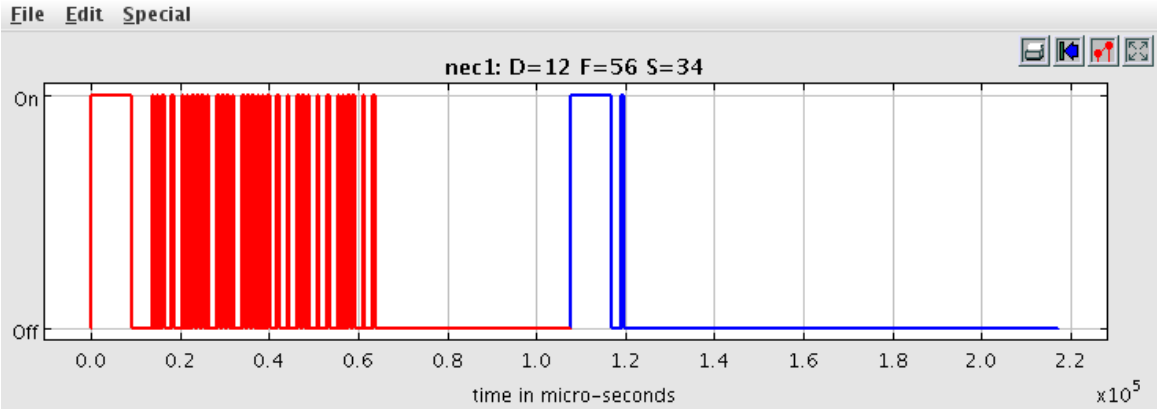


By pressing "Render", the signal is computed, and the middle window is filled with the Pronto representation of it. Pressing "Decode" sends the Pronto representation to DecodeIR. Pressing "Analyze" sends it to AnalyzeIR. In both cases, the programs will send their output to the console window, as can be seen below.



Using context menus, the result can be sent to the clipboard or saved to a file, after invoking a file selector. It is also possible to fill the code window by pasting from the clipboard. Pressing the "Import" button allows to import to import a wave file (see this for a background) or a file in ict-format, for example from the IRScope-Program. The imported IR sequence can be subsequently Decode-d, Plot-ted, and Analyze-d, etc. Note that an ICT file produce by IRScope may contain several IR signals, which cannot be easily separated. They will be imported as one giant signals, with all the gaps and spaces concatenated together. This is a flaw in the IRScope program.

By pressing the "Plot" button, either an IR signal in the CCF window, or a newly rendered signal corresponding to the selected parameters, is shown in a popup plot window, using the PtPlot library.



Using its own controls, the plot can be zoomed (press left button and drag), printed, or exported to encapsulated PostScript. Once created, it cannot be changed (other than zooming, resizing etc), just closed. However, an "unlimited" number of such popup windows are possible.

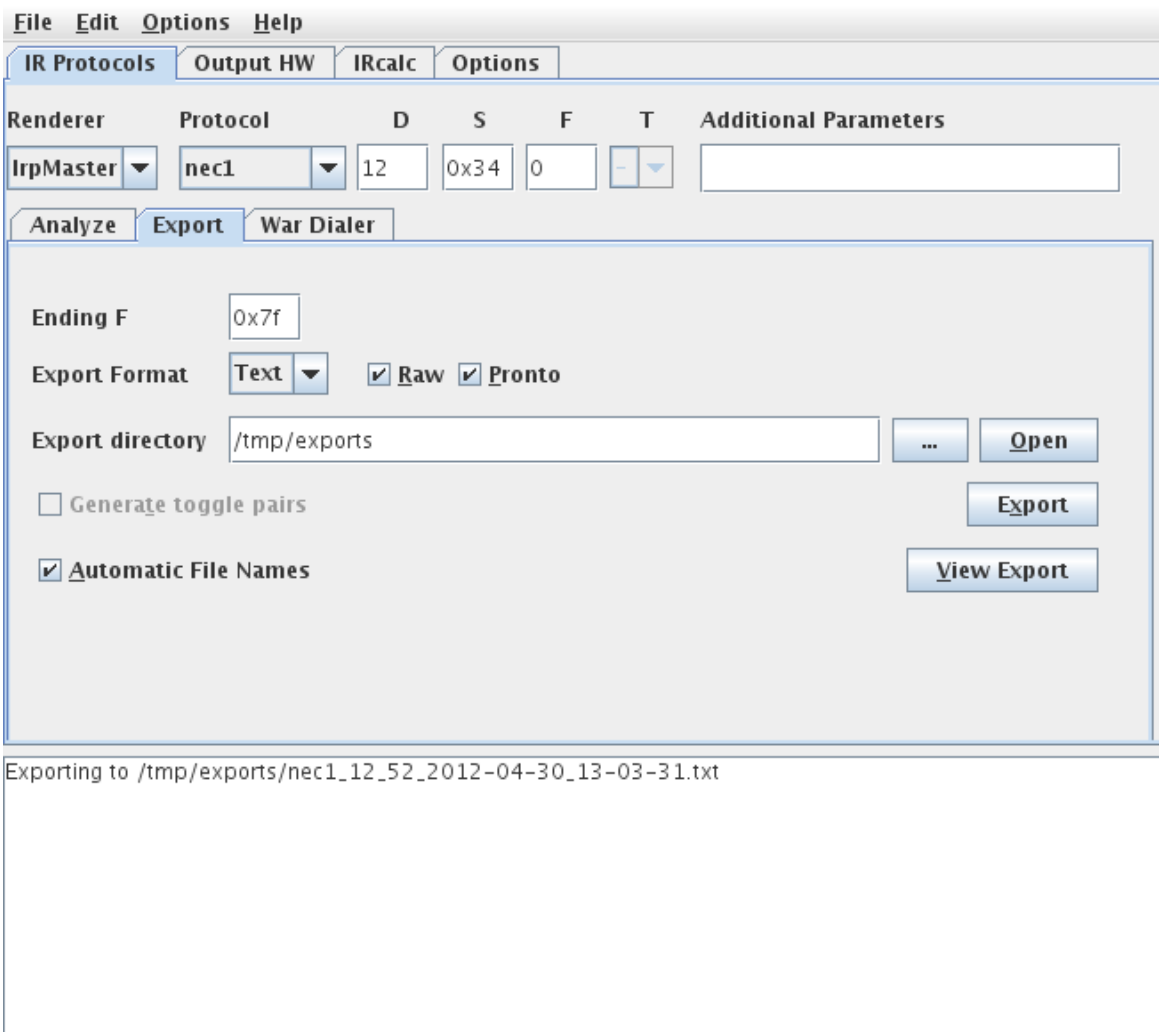
The right part of the middle window allows for sending the code in the code window to hardware, presently GlobalCaché, IRTrans, or LIRC (requires a patch), or an audio-IR-setup, any number of times the user desires. These run in their own threads, and can be stopped anytime.

In all cases, if the CCF window is non-empty and starts with "0000", DecodeIR/Analyze operates on the actual Pronto code, which may even be manually manipulated by the user. If it start with a "+"-character, it is attempted to interpret it as a "raw" signal, consisting of a number of gaps in pulses, given in microseconds. If it consists of a number of hexadecimal two-digit numbers, it is attempted to interpret the signal as a UEI learned signal.

If the window is empty, new signal is rendered according to the parameters, and subsequently used for sending or plotting. There is thus no need to "render" before plotting or sending.

In rare cases, transforming the signal to the Pronto format may introduce some rounding errors causing DecodeIR to fail to indicate some IR signals it would otherwise identify.

The Export pane



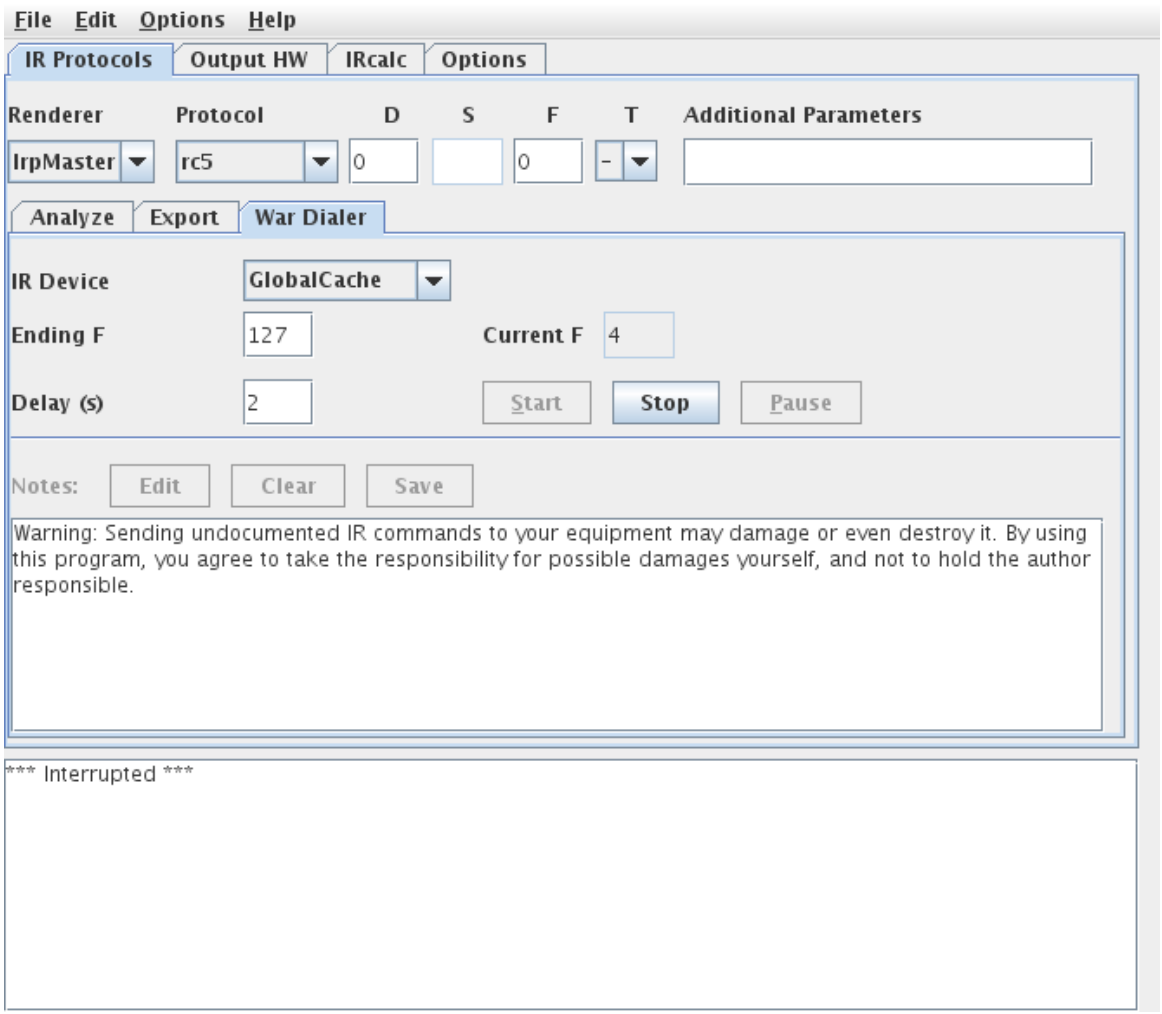
Using the export pane, export files can be generated. These allow e.g. other programs to use the computed results. Using IrpMaster as rendering engine, exports can be generated in text files, XML files, LIRC-format, Lintronic-format, or as Wave-files, the first two both in Pronto format and in "raw" format (timings in microseconds, positive for pulses, negative for gaps). The XML export is intended as a starting point for generate yet other formats, by invoking easily written transformations on it. In this article I demonstrate how to generate C code from it using XSLT transformations. The LIRC-exports are in lirc.conf-format using the raw format. They can be concatenated together and used as the LIRC server data base, typically `/etc/lirc/lircd.conf`. Of courses, they can also be used with WinLirc. For the wave-export, parameters are "inherited" from the Output HW/Audio pane. Options for the wave-export, as well as some of its usage, is explained there. Optionally, for protocols with toggles, both toggle pairs may optionally be included in the export file by selecting the "Generate toggle pairs"-option. Export file names are

either user selected from a file selector, or, if "Automatic file names" has been selected, automatically generated.

The export is performed by pressing the "Export" button. The "..."-marked button allows for manually selecting the export directory. It is recommended to create a new, empty directory for the exports. The just-created export file can be immediately inspected by pressing the "View Export"-button, which will open it in the "standard way" of the used operating system. The "Open" button similarly opens the operating systems standard directory browser (Windows Explorer, Konquistador, Nautilus,...) on the export directory.

The export formats Wave and Lintronic export an IR sequence rather than an IR signal (consisting of an intro sequence, an repetition sequence (to be included 0 or more times), and an (most often empty) ending sequence). Therefore, using the Wave and Lintronic formats, the number of repetition sequences to include can be selected.

The "War Dialer" pane



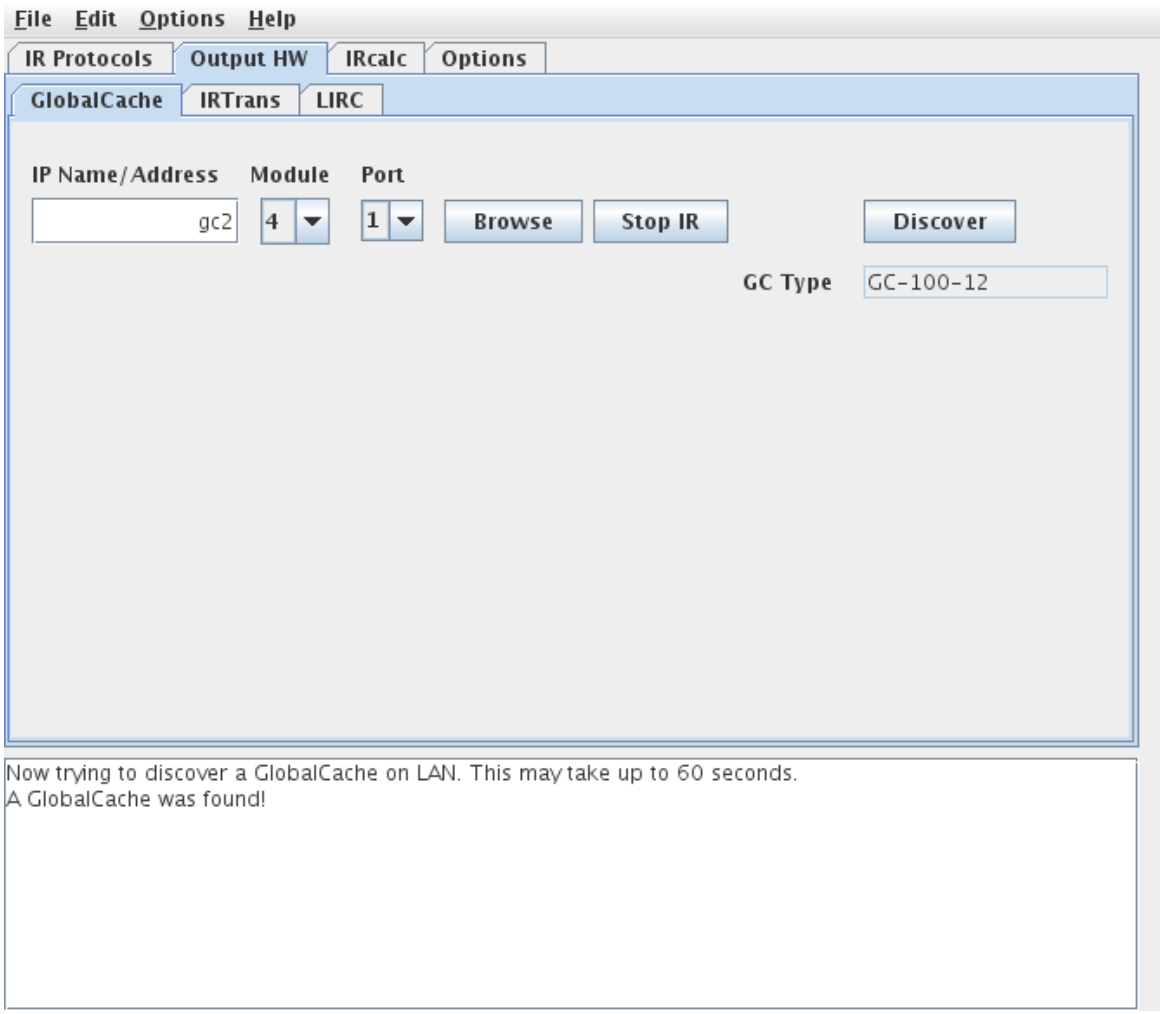
This functionality is intended for the search for undocumented IR commands for customer equipment. It allows for sending a whole interval of IR signals to the equipment, and taking notes when something reacts on the sent signal. The "Start" and "Stop" functions are probably self explaining; the "Pause" button allows for interrupting and later resuming, but is presently not implemented. A note-taking function is planned but presently not implemented: when "Edit" is pressed, a "Document" pops up with current IR signal and time, allowing the user to write a note on that signal, which can later be saved by "Save".

3.3.4.2 The "Hardware" pane

The sub-panes of this pane allows for the selection and configuration of the employed IR hardware.

The "GlobalCache" pane.

This allows for configuring GlobalCaché support. The Discover-button attempts to find the identifying beacon of GlobalCaché modules (only present on recent firmware). If successful, will fill in the IP-Box, its model, the default IR-module and IR-port (see the GlobalCaché API specification for the exact meaning of these terms). In any case, the IP Name/address window, the module and port selection can be changed manually. The Browse-button directs the selected browser to the built-in WWW-server of the module, while the "Stop IR"-Button allows the interruption of ongoing transmission, possibly initiated from another source.



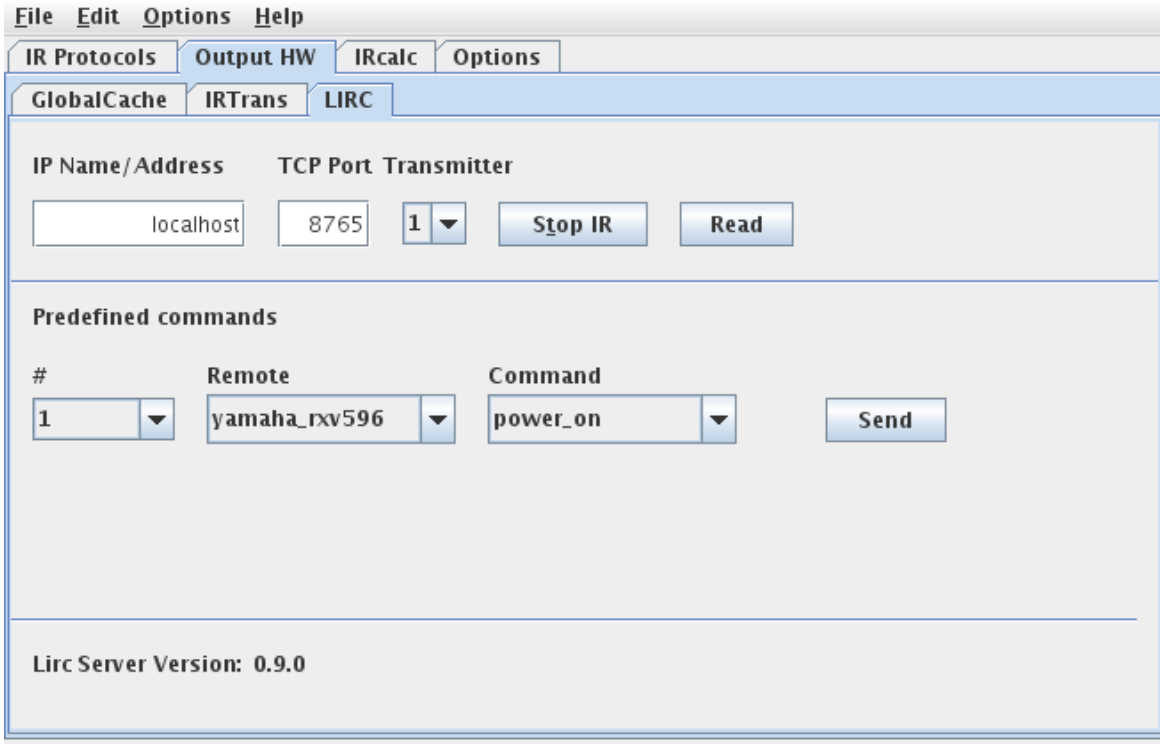
The "LIRC" pane

To be fully usable for IrMaster, the LIRC server has to be extended to be able to cope with CCF signal not residing in the local data base, but sent from a client like IrMaster, thus mimicing the function of e.g. a GlobalCaché. The needed modification ("patch") is in detail described here. However, even without this patch, the configuration page can be used to send the predefined commands (i.e. residing it its data base `lirc.conf`). It can be considered as a GUI version of the `irsend` command.

The LIRC server needs to be started in network listening mode with the `-l` or `--listen` option. Default TCP port is 8765.

After entering IP-Address or name, and port (stay with 8765 unless a reason to do otherwise), press the "Read" button. This will query the LIRC server for its version (to replace the grayed out "<unknown>" of the virgin IrMaster), and its known remotes and their commands. Thus, the "Remote" and "Command" combo boxes should now be selectable. After selecting a remote and one of its command, it can be sent to the LIRC

server by pressing the "Send" button. If (and only if) the LIRC server has the above described patch applied, selecting "LIRC" on the "Analyze" and "War Dialer" panes now works.



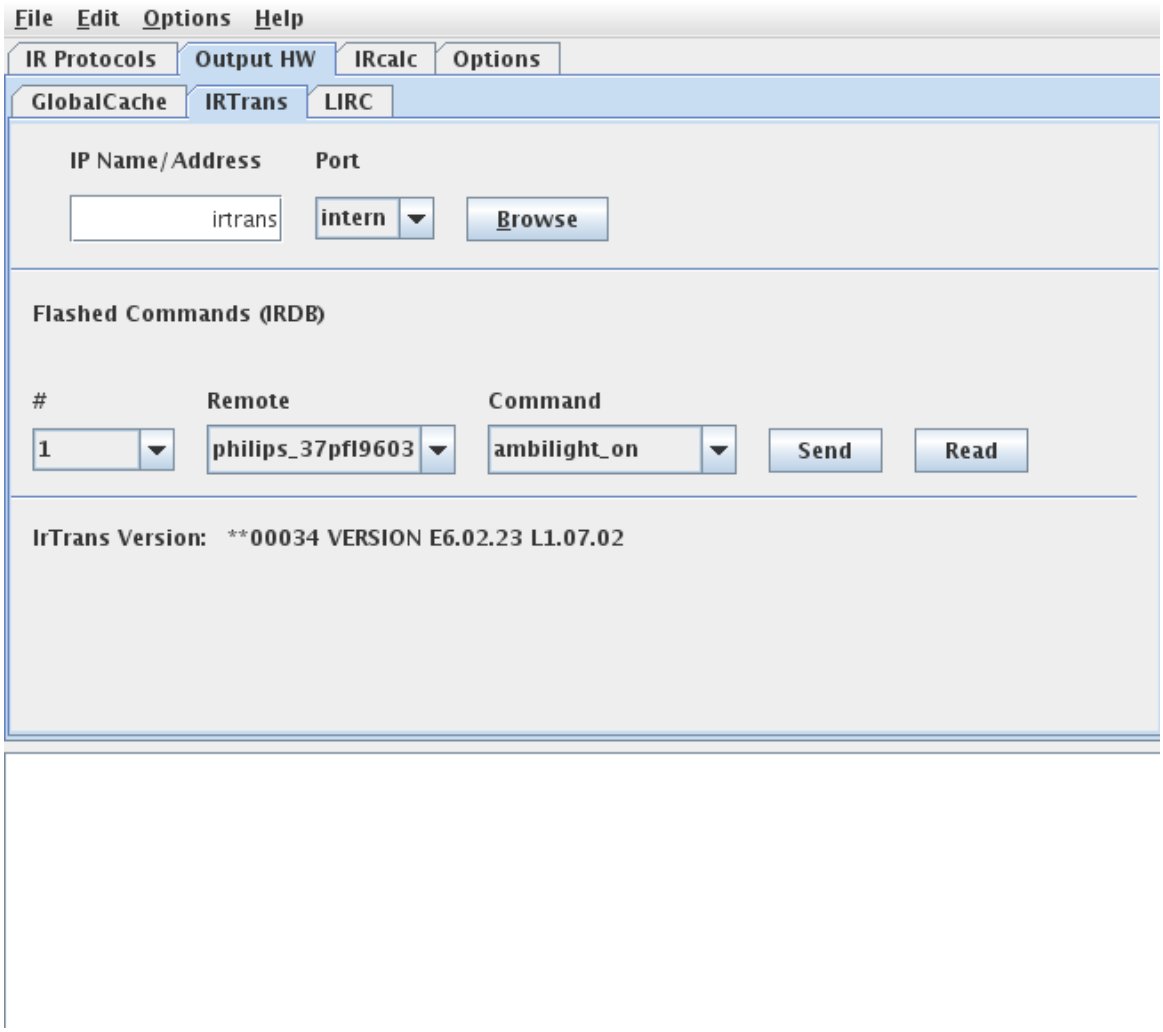
Due to LIRC's pecular form of API stop command, the "Stop IR" command presently does not work. See this thread in the LIRC mailing list for a background.

The "IRTrans" pane

The configuration of IRTrans is similar to LIRC, so it will be described more briefly.

Enter IP name or -address and select an IR port (default "intern"). If the Ethernet IRTrans contains an "IR Database" (which is a slightly misleading term for an internal flash memory, that can be filled by the user), its commands can be sent from this pane. By pressing the "Read" button, the known remotes and commands are loaded, and the version of the IRTrans displayed. The selected command can now be sent by the "Send" button. (However, this functionality is otherwise not used by IrMaster.) Selecting

"IRTrans" on the "Analyze" and "War dialer" pane should now work. The IRTrans module is then accessed using the UDP text mode.



The "Audio" Pane

As additional hardware device, IrMaster can generate wave files, that can be used to control IR-LEDs. This technique has been described many times in the internet the last few years, see for example this page within the LIRC project. The hardware consists of a pair of anti-paralell IR-LEDs, preferably in series with a resistor. Theoretically, this corresponds to a full wave rectification of a sine wave. Taking advantage of the fact that the LEDs are conducting only for a the time when the forward voltage exceeds a certain threshold, it is easy to see that this will generate an on/off signal with the double frequency of the original sine wave. (See the first picture in the LIRC article for a picture.) Thus, a IR carrier of 38kHz (which is fairly typical) can be generated through a 19kHz audio signal, which is (as opposed to 38kHz) within the realm of medium quality sound equipment, for example using mobile devices.

IrMaster can generate these audio signals as wave files, which can be exported from the export pane, or sent to the local computers sound card. There are some settings available: Sample frequency (42100, 48000, 96000, 192000Hz), sample size (8 or 16 bits) can be selected. Also "stereo" files can be generated by selecting the number of channels to be 2. The use of this feature is somewhat limited: it just generates another channel in opposite phase to the first one, for hooking up the IR LEDs to the difference signal between the left and the right channel. This will buy you double amplitude (6 dB) at the cost of doubling the file sizes. If the possibility exists, it is better to turn up the volume instead.

Data can be generated in little-endian (default) or big-endian format. This applies only to 16-bit sample sizes.

As an experimental option, the carrier frequency division as described above can be turned off (the "Divide carrier" checkbox). This is only meaningful for sample frequencies of 96kHz and higher, and for "audio equipment" able to reproduce frequencies like 36kHz and above.

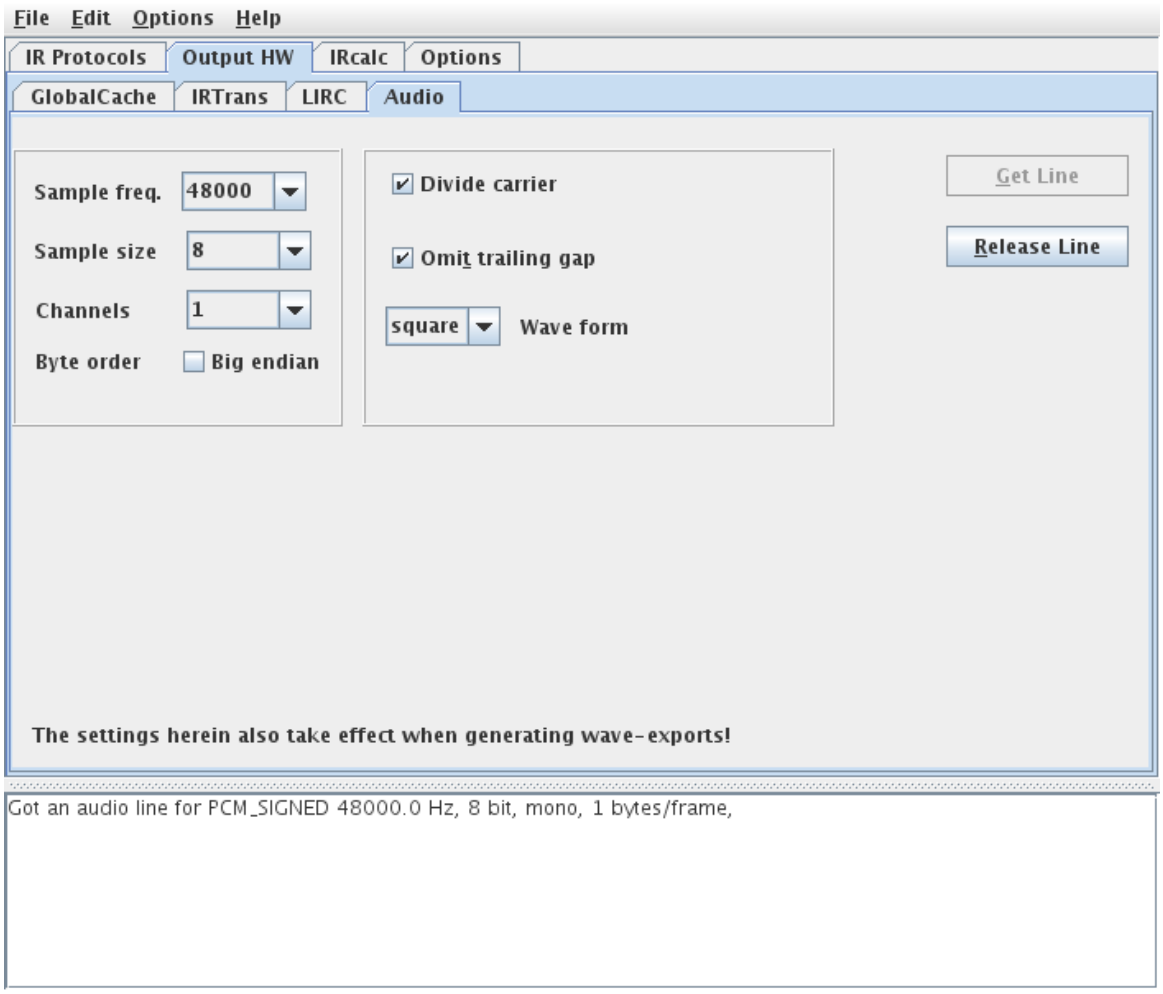
Most of "our" IR sequences ends with a period of silence almost for the half of the total duration. By selecting the "Omit trailing gap"-option, this trailing gap is left out of the generated data -- it is just silence anyhow. This is probably a good choice (almost) always.

Finally, the wave form on the modulation signal can be selected to either sine or square wave. For practical usage, my experiments shown no real performance difference.

Note that when listening to music, higher sample rates, wider sample sizes, and more channels sound better (in general). However, generating "audio" for IR-LEDs is a completely different use case. The recommended settings are: 48000kHz, 8bit, 1 channel, divide carrier, omit trailing gap, and square wave form.

Note that the settings on this pane also take effect when exporting wave files from the export pane.

By pressing "Get Line" a "line" to the audio system on the local computer is allocated. This is actually superflous, since the send-functions make this automatically anyhow. It will possibly be removed in future versions.



3.3.4.3 Command line arguments

Normal usage is just to double click on the jar-file, or possibly on some wrapper invoking that jar file. However, there are some command line arguments that can be useful either if invoking from the command line, or in writing wrappers, or when configuring custom commands in Windows.

```
Usage:
        irmaster [-v|--verbose] [-d|--debug debugcode] [-p|--properties propertyfile]
 [--version|--help]
or
        irmaster IrpMaster <IrpMaster-options-and-arguments>
```

The options `--version` and `--help` work as they are expected to work in the GNU coding standards for command line interfaces: The `-v/--verbose` option set the verbose flag, causing commands like sending to IR hardware printing some messages in the console. The debug option `-d/--debug` takes an argument, and the result is stuffed into the debug parameter in the program. This is passed to invoked programs that are free

to interpret it any way it likes. For example, here is how IrpMaster interprets the debug variable. This option is likely of interest only to developers, who have to look in the code to see what is sensible.

The second form invokes IrpMaster as a the command line program on the rest of the arguments. It is a convenience feature for just having one user entry point in the distributed jar file.

For automating tasks, or for integrating in build processes or Makefiles or the like, it is probably a better idea to use IrpMaster instead, which has a reasonably complete command line interface.

The program delivers well defined and sensible exit codes.

3.3.5 References

1. IrpMaster. Also a GPL3-project by myself. Much harder to read than the present document :-). See also this discussion thread in the JP1 forum.
2. The Harctoolbox project, also a GPL3-project by myself. It is used for the interface to GlobalCaché and IrTrans, as well as some minor support routines, that I did not feel for duplicating.
3. DecodeIR. This shared library tries to identify protocol name and parameters of an IR signal in raw form. Thus, it is in a sense, it implements the "inverse mapping" of IrpMaster.
4. GlobalCaché, a manufacturer of Ethernet connected IR hardware. Note that I have only tried with the GC-100 series, but the IR sending models of theiTach family are believed to work too. (Feel free to send me one :-).)
5. IRTrans, another manufacturer of Ethernet connected IR-hardware. The "IRTrans Ethernet" module, preferably with "IRDB Option" (internal flash memory), is directly supported by the current software.
6. LIRC, Linux InfraRed Control This project contain drivers for almost everything IR-related. The present project is able to use a modified LIRC-server for transmitting IR signals.

3.4 Using and Transforming XML export

Date	Description
2012-06-06	Initial version.
2013-12-20	Added note.

Table 1: Revision history

> **Note:**
> This document is semi-obsolete! It describes transformations on the XML export of IrpMaster and IrMaster. The XML exports of these programs has been superseeded by the newer

IrScrutinizer and the much more developed Girr format, making a Girr-version of the current document desirable.

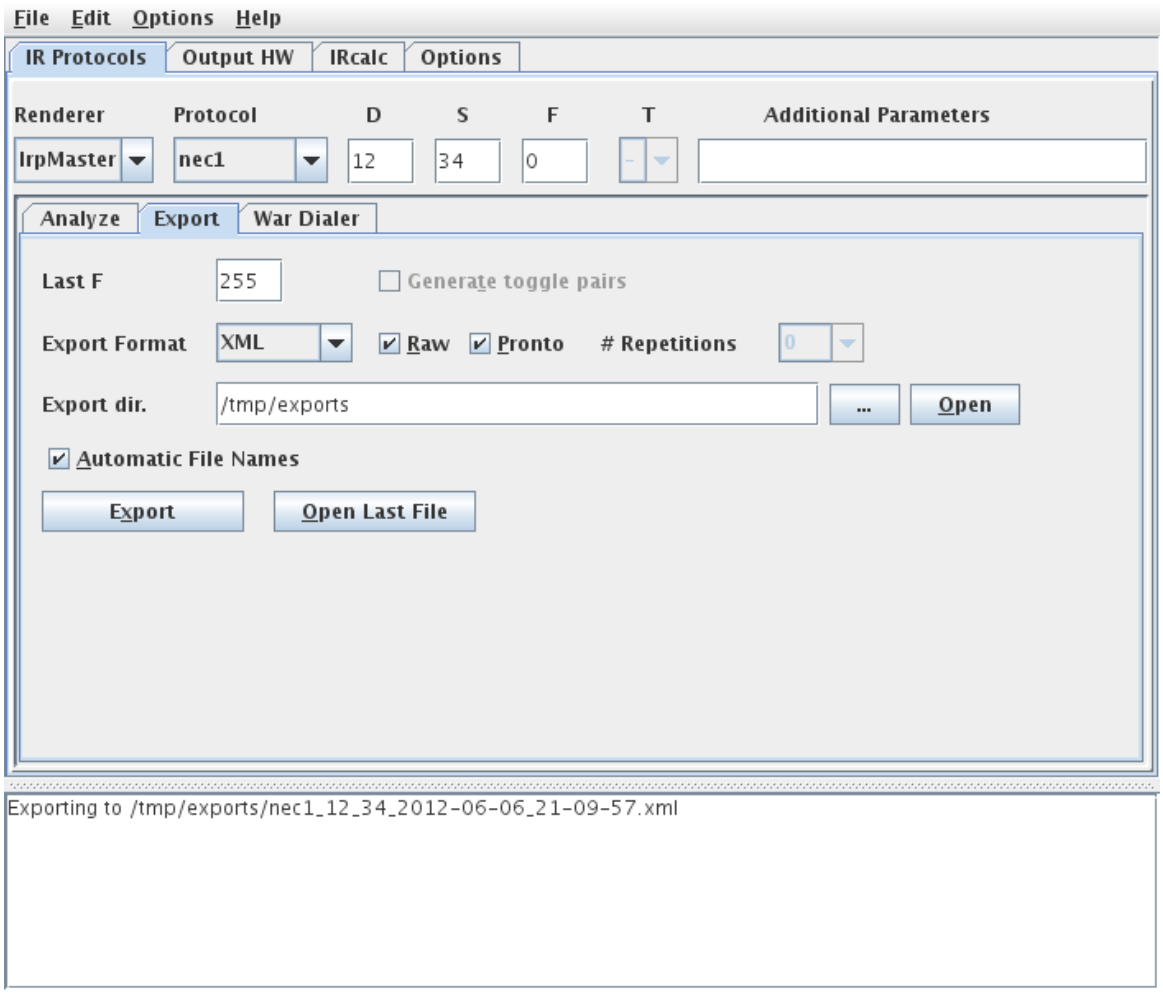
3.4.1 Description

IrpMaster and IrMaster can generate several different export formats. However, the wish may arise to either support yet another format, or to do something special with the generated IR signals. For this purpose, Ir(p)master can generate export in the form of an XML file. XML is an text based format that is readable both by humans and machines. Most importantly, there is a large number of parsers and other tools available, also as free software. As one of many alternatives, we will here look at the rather well known and reasonably easy to learn is XSLT (Extensible Stylesheet Language Transformations). This is a language, itself in XML, for transforming an XML document to either another XML document, to an HTML document, or to a text file.

In this tutorial article, I will demonstrate how to use XSLT to generate something reasonably nontrivial, namely C code, from the XML export. The C file is required to contain both the Pronto signal as C string, as well as integer arrays or durations in micro seconds. The length of the intro sequence, repeat sequence, and, if present, the ending sequence should be contained as well. Finally, we require it to be "neatly" formatted and indented.

The emphasis in this article is to show how this can be easily achieved, without going into details, either on XML, XSLT, C, or anything else. The reader interesting in a similar task can likely just adopt the XSLT file given here. For this, the XSLT-file is put in the public domain.

We want to export all NEC1 codes with device number (D) 12, sub device number (S) 34, and all the possible function (F) numbers ranging from 0 to 255. The XML export is created either with IrMaster, like this



or by using IrpMaster:

```
irpmaster  --xml --raw --outfile ircode.xml nec1 12 34 \*
```

Note that we will need both the raw form (durations in microseconds), as well as the Pronto CCF form, therefore leaving both the "Raw" and the "Pronto" selection checked, or using both `--raw` and `--pronto` options to IrpMaster. This results in the following XML export. On this XML file, the XSLT program (often called *stylesheet*) is invoked, resulting in this generated C code.

A similar, fairly straight-forward, project would be to generate `.rem` configuration files using the CCF format for the IrTrans devices.

3.4.2 irpxml2c.xsl

In this section, the code of the stylesheet irpxml2c is presented. There are a number of tutorials on XSLT available, so the code will not be explained in detail.

The code contains embedded text strings, where whitespace are preserved. We require that the generated C file to looks "neat", i.e., properly indented. Therefore, the indentation of the XSLT-file by necessity looks "ugly", and not neatly indented.

There are a number of different XSLT processors available, most of them free software. How to invoke the XSLT processesor is different from different processors, so it will not be discussed here.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copying and distribution of this file, with or without modification,
     are permitted in any medium without royalty provided the copyright
     notice and this notice are preserved.  This file is offered as-is,
     without any warranty.
-->

<!--

This file serves as an example on how to use the XML export from Ir(p)Master
to relatively easily generate new output formats. Here we, mostly as an example,
generates C code from the XML file. Although possibly useful as-is, it is intended
as an example, not as a productive solution.

Author: Bengt Martensson

-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="text" />

    <xsl:template match="/protocol">
        <xsl:text>/* This file was automatically generated by irpxml2c.xsl */

const int frequency = </xsl:text>
<xsl:value-of select="@frequency"/><xsl:text>;
</xsl:text>
<xsl:text>const char *protocol = "</xsl:text><xsl:value-of select="@name"/><xsl:text>";
</xsl:text>
<xsl:text>const int intro_burst_length = </xsl:text><xsl:value-of
 select="count(signal[position()=1]/raw/intro/flash)"/>
<xsl:text>;
const int repeat_burst_length = </xsl:text><xsl:value-of
 select="count(signal[position()=1]/raw/repeat/flash)"/>
<xsl:text>;
const int ending_burst_length = </xsl:text><xsl:value-of
 select="count(signal[position()=1]/raw/ending/flash)"/>
<xsl:text>;

</xsl:text>
        <xsl:apply-templates select="signal" mode="pronto"/>

        <xsl:apply-templates select="signal" mode="raw"/>
    </xsl:template>

    <xsl:template match="signal" mode="pronto">
<xsl:text>/* Signal D = </xsl:text>
        <xsl:value-of select="@D"/>
        <xsl:text>; S = </xsl:text>
        <xsl:value-of select="@S"/>
        <xsl:text>; F = </xsl:text>
        <xsl:value-of select="@F"/>
```

```
        <xsl:text> */
const char *pronto_</xsl:text>
        <xsl:value-of select="/protocol/@name"/>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@D"/>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@S"/>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@F"/>
        <xsl:text> = "</xsl:text>
        <xsl:value-of select="pronto"/>
        <xsl:text>";
</xsl:text>
    </xsl:template>

    <xsl:template match="signal" mode="raw">
       <xsl:text>const int raw_</xsl:text>
        <xsl:value-of select="/protocol/@name"/>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@D"/>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@S"/>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@F"/>
        <xsl:text>[] = { </xsl:text>
        <xsl:apply-templates select="raw"/>
        <xsl:text> };
</xsl:text>
    </xsl:template>

    <xsl:template match="raw">
        <xsl:apply-templates select="*/*"/>
    </xsl:template>

    <xsl:template match="flash"><xsl:value-of select="."/><xsl:text>,</xsl:text></
xsl:template>

    <xsl:template match="raw/*[position()=last()]/gap[position()=last()]"><xsl:text>-</
xsl:text><xsl:value-of select="."/></xsl:template>
    <xsl:template match="gap"><xsl:text>-</xsl:text><xsl:value-of select="."/
><xsl:text>,</xsl:text></xsl:template>

</xsl:stylesheet>
```

3.4.3 Automatic generation using make

It may be desirable to use some sort of automated procedure to generate the C code, for example as part of a project build. The (still) most common build system is Make, so we will show an example utilizing a Makefile. This will, provided that the pathnames are correct, invoke IrpMaster to generate an XML file, and subsequently Apache Xalan as an XSLT-processor to transform it into C code.

```
JAVA=/opt/jdk1.6.0_30/bin/java
XALAN=$(JAVA) -jar /usr/local/apache-forrest-0.9/lib/endorsed/xalan-2.7.1.jar
IRPMASTER=irpmaster

ircode.c: ircode.xml irpxml2c.xsl
 $(XALAN) -IN $< -XSL irpxml2c.xsl -OUT $@
```

```
ircode.xml:
 $(IRPMASTER) --pronto --raw --xml --outfile $@ nec1 12 34 \*
```

3.5 Harctoolbox: Home Automation and Remote Control

FIXME (BM):

In the background of the two programs IrpMaster and IrMaster, the work reported herein is to some extent obsolete, in particular the IR protocol descriptions. In the near future, I intend to restructure this project slightly, possibly splitting it in several subprojects. The current "release", called 0.7.0, is really nothing but a snapshot to support IrMaster. It differs from 0.6.0 in particular since the peculiar IR engine has been replaced by IrpMaster, but also through a number of small fixes.

3.5.1 Revision history

Date	Description
2009-07-18	Initial version, as `harc.html`.
2011-01-10	Restructured. Main part now called `harctoolbox_doc.html`.
2011-04-29	Somewhat updated. The document is presently not quite up-to-date, see the fixme section.

3.5.2 Introduction

The present document is aimed more at a high-level description of the system, rather than being a user's manual. It describes most aspects of Harctoolbox at most very roughly. Much is not described at all. Some information in this file may even refer to the previous version, and no more be accurate.

Since I make no assumptions on reaching a large number of users with this release, I think that my creativity is better spent on something else than writing detail documentation for something only experts will use; and these experts probably do not need detailed documentation anyhow...

3.5.3 Overview of the system

The "system" consists of a number of data formats, and Java classes operating on those formats. It has been the goal to formulate file formats allowing for a very clean description of infrared protocols, the commands of devices to be controlled (in particular, but not exclusively audio- and video equipment), networking components, topology of an installation, as well as macros. From these very general universal formats configuration files in other formats, used by other systems, can automatically be generated.

There is also a quite universal GUI that gives access to most of the functionality within the Java classes. This has been written to demonstrate the possibilities for the expert user. Convenience and accessibility for the novice user has not been a priority. (Such user interfaces are conceivable, however.)

Directly supported communication hardware and software are GlobalCache GC-100, IRTrans LAN (only the LAN version is supported directly), RS232 serial communication through the GlobalCaché or ser2net, TCP sockets, HTTP, RF 433 and 868 MHz through EZcontrol T10 as well as through an IR->RF translator like Conrad Eggs or Marmitek pyramids. Indirectly, through LIRC, a vast number of IR senders are supported.

3.5.4 Data model

Next a somewhat technical description of the file formats will be given. These are all defined in the form of XML files with a rather strict DTD. The discussion to follow will focus on the concepts, not on the details. Of course, the semantics of the files are defined by the DTD file.

Necessary theoretical background on IR signals can be found for example in this article.

3.5.4.1 The command names

Harctoolbox has higher requirements on the data quality of its input files than other related projects. For example, in LIRC, a configuration file consists of a device name, and a number of commands with associated IR signals. The names of the commands there are in principle completely arbitrary, and it appears to be common to try to follow the vendor's naming. In Harctoolbox, there is instead a fixed (but of course from the developer extensible) list of command names, intended to uniquely denote a command for a device. A command name in principle is a verb, not a noun, and should describe the action as appropriate as possible. There is, for example, no command `power`, instead there are three commands: `power_on`, `power_off`, and `power_toggle` (having the obvious semantics). Also, a command which toggles between play and pause status may not be called `play`, but should be called `play_pause`.

The names are defined in the XML file `commandnames.xml`, from which a Java enum `command_t` is created through an XSLT style-sheet `mk_command_t.xsl`. Further rules for command names are found as comments in that file.

3.5.4.2 The protocol files

> FIXME (BM):
>
> This section is obsolete. The IR engine has been replaced by IrpMaster.

By "(infrared) protocol" we mean a mapping taking a few parameters (one of those a "command number", one a "device number", sometimes others) to an infrared signal. A protocol file is an XML file describing exactly how the IR signal is made up from the parameters. The format is quite close to an XML version of the "IRP notation" of

the JP1-Project. It is a machine readable description on how to map the parameters into a modulated infrared signal, consistent with a technical description. The protocol is identified by its name. A protocol takes a certain number of parameters, although sometimes one is defaulted.

At his point, the reader may like to compare this prose description of the protocol we (and the JP1 project) call `nec1` with the XML code in `nec1.xml`. Note that our description, using an arbitrary subdevice number, corresponds to the author's "Extended Nec protocol".

The naming of the different protocols is of course somewhat arbitrary. In general, I have tried to be compatible with the JP1-Project.

It can be noted that the supported radio frequency protocols are nothing but IR-signals with the carrier consisting of infrared 950nm light substituted by suitable radio carrier, typically of 433 MHz.

Ideally, most users should not have to worry with the protocol files. This is only necessary when introducing a device with a IR-protocol that has not yet been implemented. At the time of this writing the 17 protocols have been implemented, this covers the most important ones, but not all known to e.g. the JP1 project.

3.5.4.3 Device files

A device file is an XML file describing the commands for controlling a device. In Harctoolbox a device file truly describes the device and its commands, stripped from all information not pertaining to the very device, like key binding on a remote, button layout, display name, the IR blaster it is connected to, location, IP-address, MAC-address, etc. (This is in contrast to many other systems, like Pronto CCF-files or JP1 device updates).

There may be many different types of commands for the device, like IR, RF signals (this is, at least for the few cases presently supported, nothing else but IR signals with the infrared light as carrier replaced by an radio signal, for Europe of 433 or 868 MHz frequency), commands over serial RS232 interfaces or TCP sockets, or utilizing a WEB API. Also "browsing" the device (pointing a Web browser to its www server), pinging and sending WOL-packages are considered commands, as well as suppling power, sometimes "in reverse direction" (like a motorized blind going up or down). Possibly the same command can be issued over different media. Some commands may take arguments or deliver output. For this (and other) reasons, care should be taken to use the "correct" command names, not just a phrase the manufacturer found cool. Commands are grouped in *commandsets*, consisting of commands having its *type* (ir, serial, tcp,...), device number etc in common.

IR signals within a device file may contain codes in Pronto CCF format in addition (or instead) if the structured information (protocol, device number, command number etc). Actually, "exporting in XML format" means generating an XML file augmented with the raw CCF codes. In may cases, also so-called cooked Pronto codes (Background, written by remotecentral) are included, as well as JP1 protocol information.

The device configuration file is processed by an xinclude-aware parser, allowing a certain include-file structure, that can be used to structure the data.

Example

As an example, consider the Oppo DV-983HDVD player with serial support. This is supported by Harctoolbox with the file `oppo_dv983.xml`. Its commands can be downloaded directly from the manufacturer (hats off!), both the infrared and the serial commands. As can be found in the spreadsheet on the IR code, the device uses the previously mentioned `nec1` protocol, with device number equal to 73. This corresponds to the first command set in the mentioned device file. The serial commands form another commandset, subdivided into *commandgroups*, depending on whether they take an argument and/or deliver output. Note that some commands (for example `play`) are available both as IR and as serial commands.

Other interesting examples are the `*_dbox2.xml` files (referring to the German dbox with the open source tuxbox software) , each containing two (`sagem_dbox2.xml`, `philips_dbox2.xml`), or three (`nokia_dbox2.xml`) different infrared command sets as well as an elaborate web-api command set. Another very interesting example is the Denon A/V-Receiver `denon_avr3808.xml` having several infrared command sets using the `denon` protocol (which, ironically, is called the "Sharp protocol" by the firm Denon), as well as several command sets using the `denon_k` (Denon-Kaseikyo protocol). Then there is a large number of "serial" commands, available through both the serial box as well as through the (telnet) tcp port 23.

Importers

Since Harctoolbox is so picky with command names and their semantics, the value of an import facility is limited — necessary information is simply not there (or is wrong). There exists a large number of IR signal data in the Internet (for example from LIRC configuration files, JP1 device updates, or the large collection (mainly CCF) of files on Remotecentral. Presently, Harctoolbox has "importers" for Pronto/CCF and JP1's device upgrades in RemoteMaster format. I "sort-of" wrote a LIRC-to-CCF translator a few years ago, possibly I will finish it someday. However, the importers have as their goal to create a first iteration of a device file (not even guaranteed to be valid XML!) to be tweaked manually.

Exporters

Writing an exporter is in principle easier. Harctoolbox presently can export the IR signals of a device in CCF format, LIRC-format (either a particular device, or all devices connected to a particular LIRC server defined in the home file), JP1's device upgrades in RemoteManager format, as well as the rem-files used by IRTrans. Individual IR-signals can be exported in wav-format for usage with an audio output driving an IR LED after full wave rectification, see for example this article This feature is presently not available through the GUI.

Many other things are possible. I have had some success creating a program that, given an XML configuration file, creates a full JP1-type image that can be flashed on a URC-7781 (that is, not just one or a few device updates).

3.5.4.4 The "home file"

The protocol and device files described up until now are a sort of universal data base — common and invariant to every user, at least in principle. In contrast, the "home file" (possibly the name is not very well chosen) describes the individual setup ("home"). It is a good idea to think of the device files as class definitions, classes which can be instantiated one or more times, in that one or more devices of the same class are present in the home configuration, each having its individual (instance-)name.

It is instructive to have a look at the supplied file `home.xml` at this point. In the home file the different devices are defined as class instances. They can be given alternate names (aliases) and groups can — for different purposes — be defined. For example, this can be useful for generating GUIs taking only a certain group of devices into account. Gateways are defined: a gateway is some "gadget" connecting some other media together, for example, the GlobalCache (among other things) connects the "tcp connector" on the local area network (`lan`) to its output connectors, which may be e.g. an infrared blaster or stick-in LED controlling an IR-device. Devices that can be controlled declare the said device/connector combination as a "from-gateway", or indirectly as a from-gateway-ref (using the XML idref facility). (Yes, there are a lot of details here which ideally sometime should be described in detail.) Thus, a routing is actually defined: how to send commands to the device in question. Note that there may be many, redundant, paths. The actual software is actually using this redundancy, thus implementing a certain failure-safeness. The actual from-gateways, and their associated paths, are tried in order until someone succeeds in sending the command. (Of course, only to the extent that transmission failures can be detected: non-reachable Ethernet gateways are detected, humans blocking the way between an IR-blaster and its target are not...).

Also the interconnection between AV devices can be described here, see the example. Thus, it is possible to send high-level input selecting commands like "turn the amplifier `my_amplifier` to the DVD player `my_dvdplayer`", and the software will determine what command (IR or other) to send to what device. (This is later called "select mode".)

There is a great potential in this concept: Consider for example a "Conrad Egg transmitter", which for our purposes is nothing but IR->RF gateway. Assume that a IR stick-on emitter is glued to the egg, and connected to a Ethernet -> IR gateway. If there is, say a RF controlled Intertechno switch, interfacing with an electric consumer, it is possible to just issue the command for turning the electric consumer on or off, and the software will find out that it has to send the appropriate IR signal to the IR gateway.

However, writing the configuration file is a job for the expert...

3.5.5 Scripting and macros

In the previous version, a simple XML macro facility was implemented. In this version, it has been replaced by the incorporating the scripting engined Jython. This is simply a Java implementation of the very popular and wide spread scripting language Python. This approach has some definite advantages to the previous "solution": It is possible for the user to "script" the Harctoolbox engine, "macros", using a well established and documented syntax and semantics. It also gives access to internal Java objects as Python objects (which of course may introduce some problems too...).

3.5.6 Basic Java classes

There is a large number of Java classes operating on the data objects. Some classes operates on protocols, some on device classes (through device files), some on device instances in the sense of the home file. In most cases when it is sensible to call use the class individually, it contains a `main`-method, i.e. can be called from the command line. In general, there are a number of arguments. A usage message can be generated in the usual GNU way, using `--help` as argument.

3.5.7 Program usage

The main entry point in the main `jar`-file is called `Main`. Its usage message reads:

```
Usage: one of
        harctoolbox --version|--help
        harctoolbox [OPTIONS] [-P] [-g|-r|-l [<portnumber>]]
        harctoolbox [OPTIONS] -P <pythoncommand>
        harctoolbox [OPTIONS] <device_instance> [<command> [<argument(s)>]]
        harctoolbox [OPTIONS] -s <device_instance> <src_device_instance>
where OPTIONS=-A,-V,-M,-C <charset>,-h <filename>,-t ir|rf|www|web_api|tcp|udp|serial|
bluetooth|on_off|ip|special|telnet|sensor,-T 0|1,-# <count>,-v,-d <debugcode>,-
a <aliasfile>, -b <browserpath>, -p <propsfile>, -w <tasksfile>, -z <zone>,-c
 <connectiontype>.
```

Using the `-g` (as well as no argument at all, to allowing for double clicking the `jar`-file) starts Harctoolbox in GUI mode, described in the next section. Invoking Harctoolbox with the `-r`, `-l` *portnumber* starts the readline and port listening mode respectively. Otherwise Harctoolbox will run in non-interactive mode, executing one command or macro, and then exit. If the `-P`-argument is given, commands are interpreted by the Python interpreter, otherwise they are interpreted as *device command [argument(s)...]*.

3.5.7.1 Non-interactive mode

The `-s` option enables the select mode, described previously. Otherwise, the arguments are considered as a device instance name, followed by a command name, and optionally by arguments for the command. If the command name is missing or "?", the possible command names for the device will be listed.

The remaining options are as follows:

-A
switch only audio on target device (if possible)
-V
switch only video on target device (if possible)
-M
use so-called smart-memory on some devices
-h *home-filename*
use *home-filename* as home file instead of the default one
-m *macro-filename*
use *macro-filename* as home file instead of the default one
-t *type*
prefer command of type *type* regardless of ordering in home file (if possible)
-T *zero_or_one*
for codes with toggles (like RC5), set the toggle value to the argument.
-v
verbose execution.
-d *debug code*
set debug code. See `debugargs.java` for its precise meaning. Use -1 to turn on all possible debugging.
-a *aliasfile*
Normally, aliases (allowing the software accept e.g. "enter" and "select" as synonyms for "ok") are taken from the official `command.xml`. This option allows the usage of another alias file.
-b *browserpath*
Allows using an alternative path to the browser used to invoke `browse`-commands, instead of the default one.
-p *propsfile*
Allows using an alternative properties file, instead of the default one.

The following options apply only to the select mode

-z *zone*
Select for zone *zone* (if possible)
-c *connection*
Prefer connection type *connection* for the selection (if possible)

3.5.7.2 Readline mode

The "Readline mode" is an interactive command line mode, where the user types the commands one at a time. If GNU readline is available, the extraordinary facilities of GNU readline allows not only to edit present command and to recall previous commands, but also for an intelligent completion of relevant names for macros, devices, and commands. If GNU readline is not available, Harctoolbox's "readline mode" will still work, only these "comfort features" are missing. The semantics of the typed command are like the

non-interactive arguments. There are also some extra commands, introduced by "--"; these in general correspond to the command line options described above. The normal command line options are ignored.

3.5.7.3 Port listen mode

Starting Harctoolbox in port listening mode starts a multithreaded server, listening for commands on the TCP port given as argument, default 9999. The present version also listens to UDP connections using the same port number. The server responds to a connection on that port and spawns off a new thread for each connection. It listens to commands on that port, sending output back. The semantics of the command line sent to the server is the same as for the non-interactive invocations, with the addition of the commands `--quit`, which makes the session/thread close, and `--die`, which in addition instructs the server not to spawn any more threads, but to die when the last session has ended.

If the `-P`-argument is given, the issued commands are interpreted though the Python interpreter. In this case, the default portnumber is 9998.

3.5.7.4 The GUI

The present GUI was not designed for deployment. It does not offer a user friendly way for allowing a nontechnical user to control his home or home theater. Rather, the goal was a research-type GUI, to allow the expert user to access to most of the functionality of the Java classes, without having to look in the Javadoc class documentation.

Hopefully, in the near future, there will be one or more "cool" GUIs for the system. This need not be additions to the present system, but rather integrations with other technologies and projects, like Openremote.

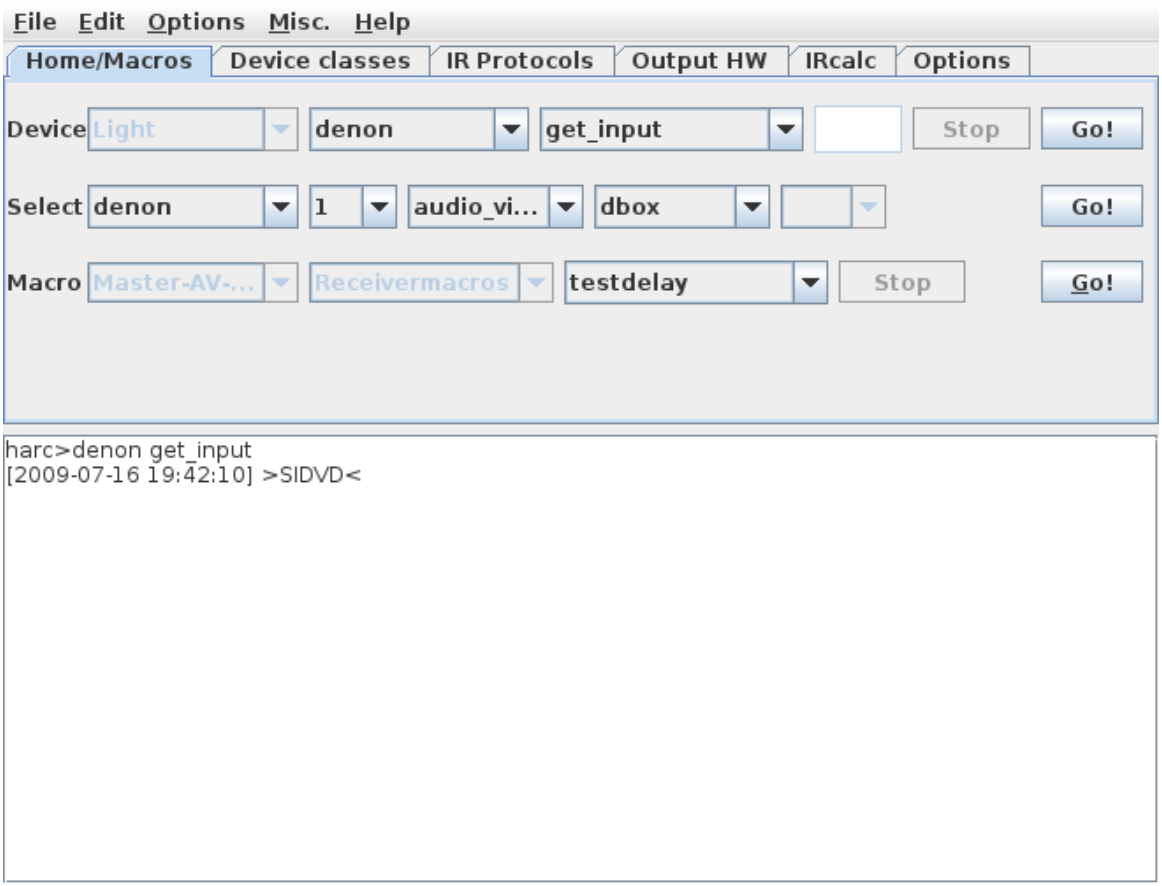
The main properties of the present GUI will be described next.

The GUI consists of a title bar with pull-down menus for File, Edit, Options, Misc., and Help. These are believed to be more-or less self explanatory. There are six panes, that will be described in order. Many interface elements have a short tool-text help messages, which are displayed when the cursor is hoovering above the element. The lower part of the main window is occupied by "the console". The latter is a read-only "pseudo paper roll console", listing commands, tracing- and debugging information as directed by the user's selections, as well as command output and error messages.

Except for the mandatory about-popup (which is of course non-modal!), popups are not used.

The GUI resides almost completely within the file `gui_main.java`. It was designed using the Netbeans IDE version 6.5.

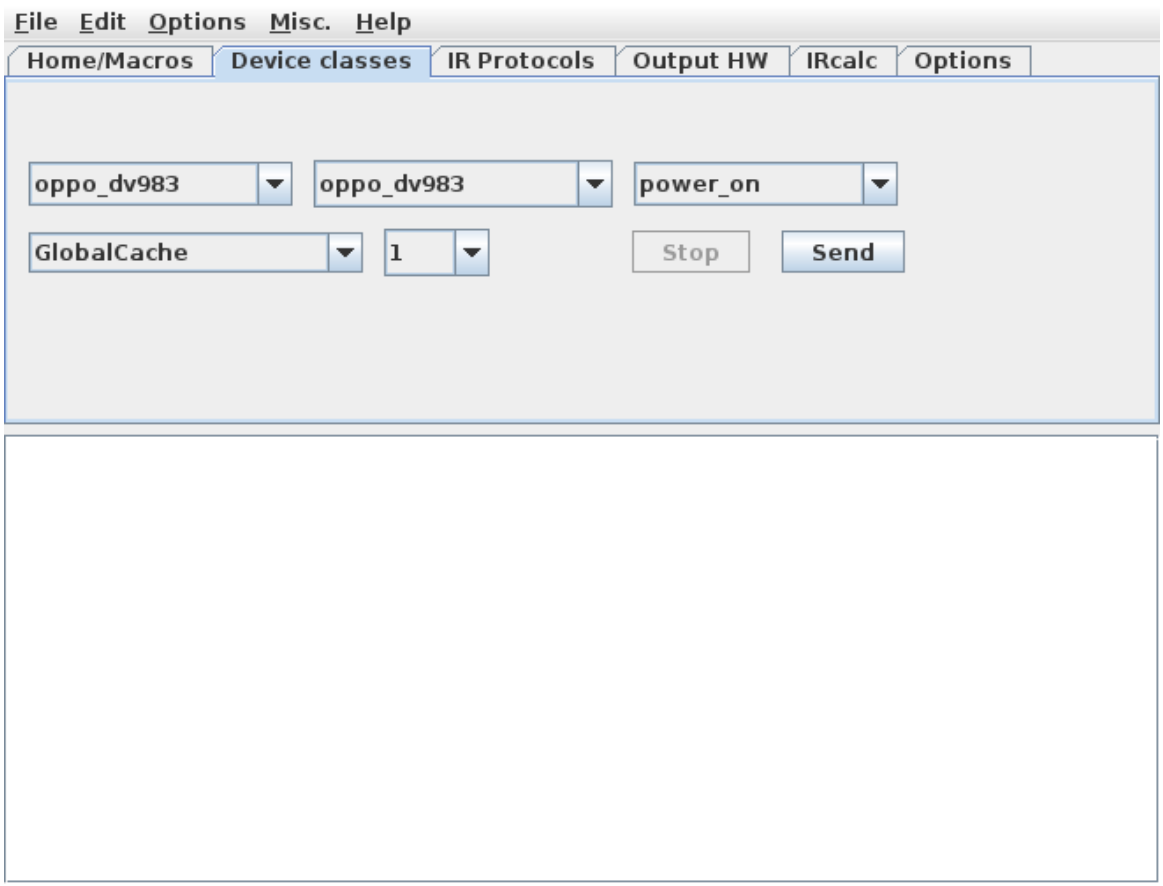
The Home/Macros pane



This pane corresponds to using Harctoolbox through the Home configuration file. Devices, using their instance names as defined in the home configuration file are sent commands, possibly with *one* argument, possibly returning output in the console. (Commands taking two or more arguments cannot be sent through the GUI.) The first row is for sending commands to devices, the second for the select mode, while the third one can execute macros. Note that both the execution of macros and of commands are executed in separate threads.

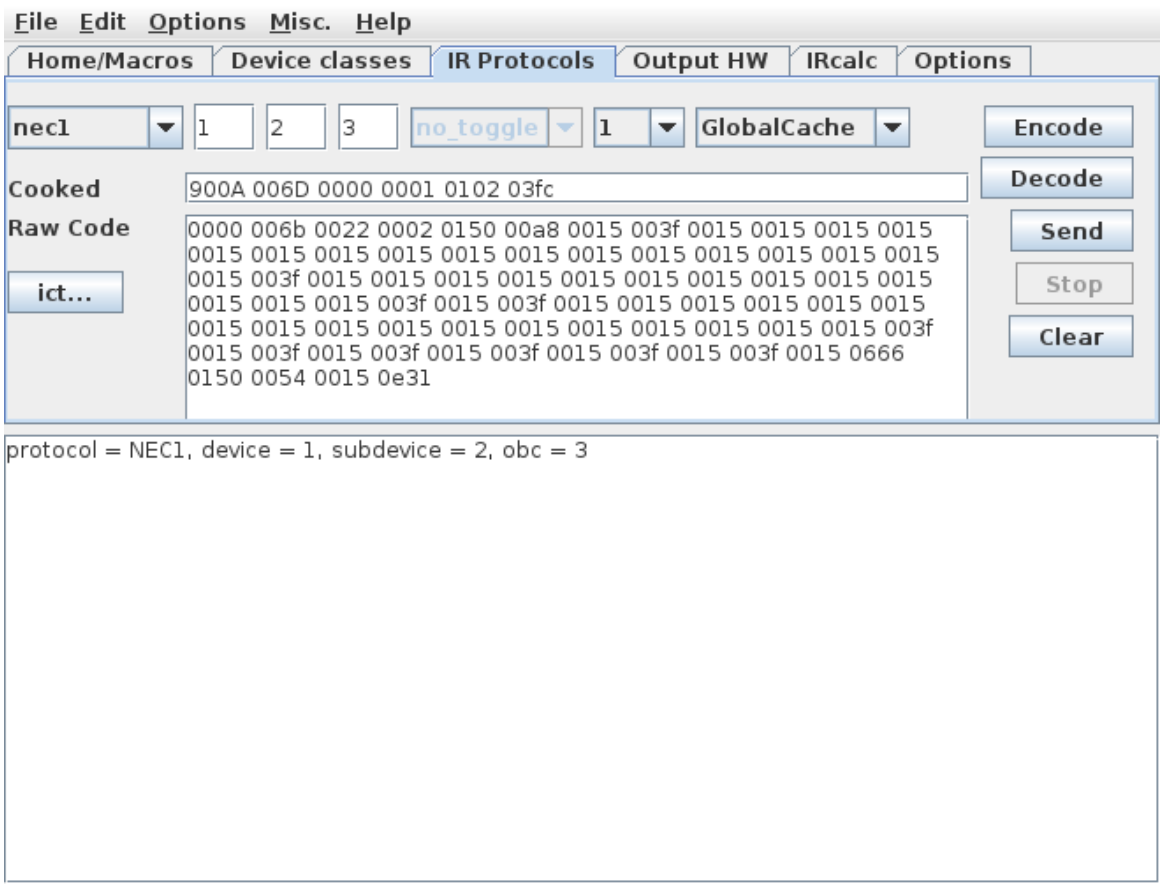
This pane is the only one coming close to "deployment usage". The other panes can be useful for setting up a system, defining and testing new devices or protocols, or for "research usage".

The Device classes pane



This pane allows for sending *infrared* signals (no other command type!) to the using either a GlobalCache or an IRTrans, that has been selected using the "Output HW" pane, including output connector to use. The home configuration file is not used. The devices are called by their class names.

The IR Protocols pane

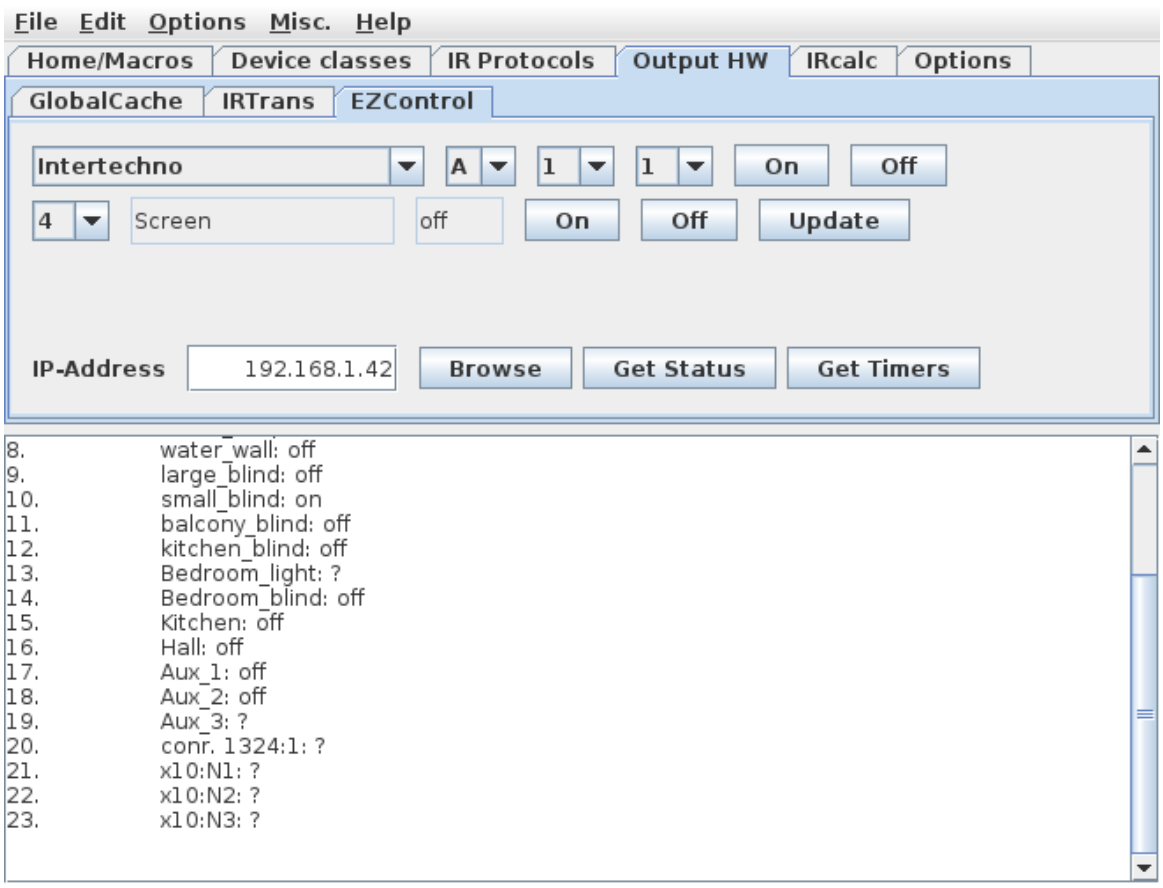


This pane has more of research character. For a protocol in the protocol data base, a device number, possibly a subdevice number, and a command number is entered, pressing the "Encode" button causes the corresponding IR code in Pronto CCF format to be computed and displayed. Pressing the send button causes the code to be sent to a GlobalCache or IRTrans that was selected in the "Output HW" pane. Note that it is possible to hand edit (including pasting from the clipboard) the content of the raw code before sending. Whenever there is content in the raw code text area, the decode button can be invoked, sending the content to the DecodeIR library, thus trying to identify an unknown IR signal (if possible).

Log files from the Irscope program (using `.icf` as their file extension) can be imported using the `icf` button.

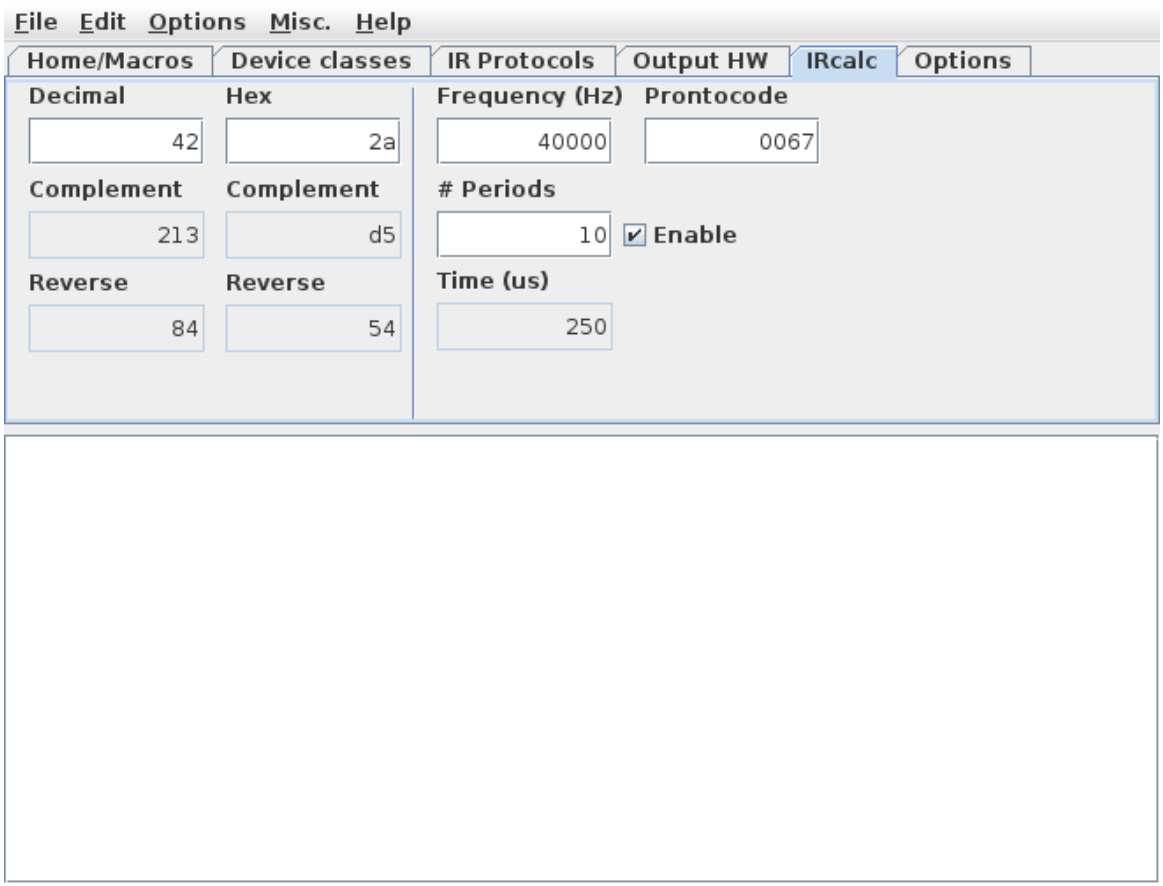
There presently appears to be some "glitches" in the button enabling code; click e.g. in the "raw code" text area to enable buttons that are incorrectly disabled.

The Output HW pane



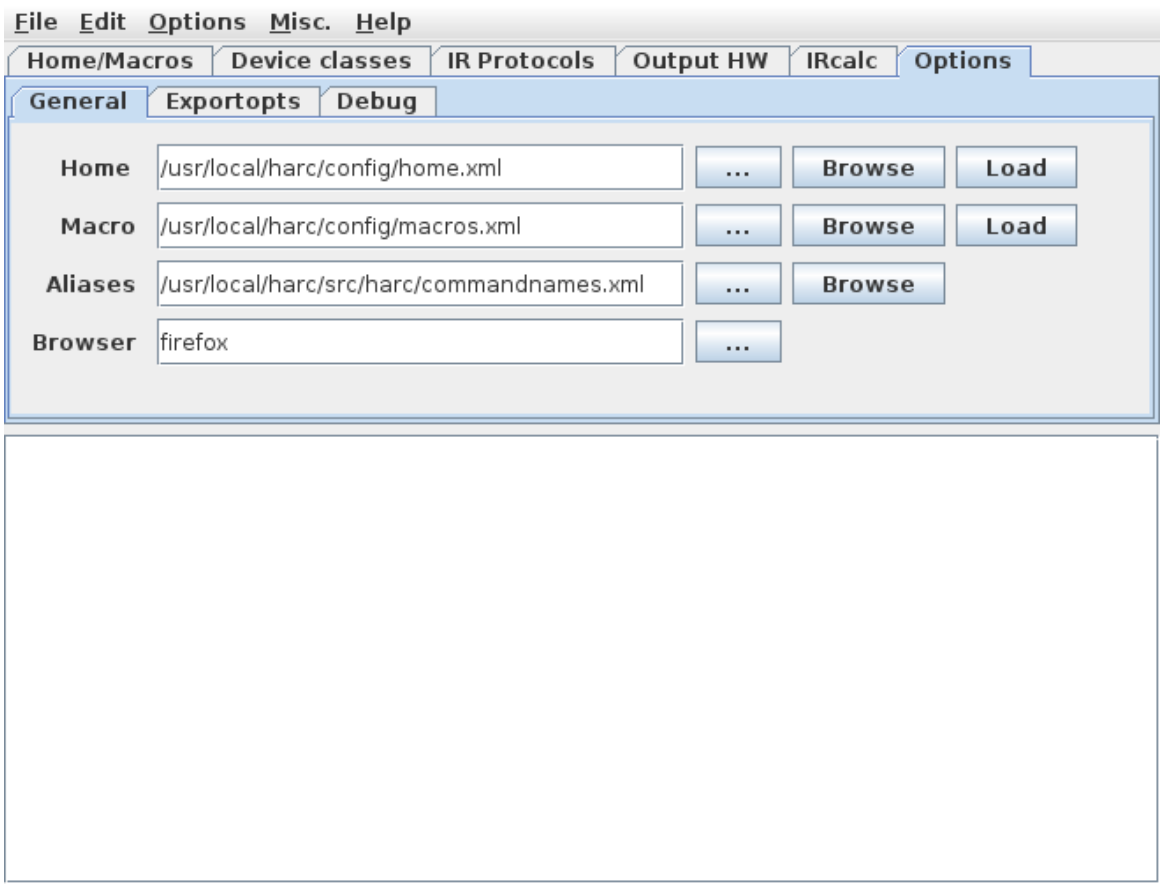
This pane has three subpanes: GlobalCache (for selecting the GlobalCache, and its output connector, used on the Device classes and on the IR Protocols pane), IRTrans (dito), and EZControl. The latter is sort of an interactive toolbox for the EZcontrol T10, allowing to send different commands, inquiry the status of one or all of the preselected switches, as well as getting a list of its programmed timers.

The IRcalc panel



This pane is a sort-of spreadsheet for computing IR signals in the Pronto or JP1 context. The exact way it works is left as an exercise for the reader...

The Options panel



This pane allows the user to set a few more options. On-off options are sometimes instead available through the Options pull-down menu.

3.5.7.5 Properties

Harctoolbox uses a properties file in XML format. For some of the properties there is no sensible access in the GUI. For this reason, it may therefore sometimes be necessary to manually edit this file with a text editor (or XML editor).

3.6 lirc2xml: Extracting information from LIRC configuration files

Date	Description
2010-03-20	Initial version.
2012-04-21	Moved from www.bengt-martensson.de/harc/lirc2xml.html to www.harctoolbox.org. Compatibility with current version or Lirc, 0.9.0, documented. Minor errors in document fixed.

Date	Description
2012-05-01	Created version 0.1.2, only small changes. Created binaries. Minor improvements.
2013-02-17	Created version 0.1.3, containing an important bugfix. Updated patch and Linux 64-bit executable (not the other binaries).

Table 1: Revision history

3.6.1 Introduction

Converting LIRC configuration files to a raw representation is no easy task. The meaning of the different parameters are to some extent documented through comments in the code, but are not at all straightforward. (Interestingly, the LIRC documentation is referring to WinLIRC for documentation...) I have therefore written `lirc2xml` as an extension to LIRC itself. It uses internal LIRC functions from `transmit.c` to render the signals. For this reason, it is available as a patch to the LIRC sources.

The program generates the CCF (often called "Pronto format") representation of the raw signal, embedded in an XML file. Optionally, the DecodeIR library is invoked to attempt to decode the signals.

The program was announced on the Lirc mailing list on 2009-08-22. It did not receive any attention. Ideally, I would like to see the program in the future LIRC distribution, however I have a strong suspicion that LIRC-ers would rather see *foobar*`2lirc` than `lirc2`*foobar*...

The program runs from the command line in a Linux environment. Current version is called 0.1.3. It was developed under Lirc 0.8.6, but works without changes with the current version, 0.9.0.

3.6.2 Usage

The usage is given by:

```
Usage: lirc2xml [options] [config-file]
         -h --help                      display this message
         -v --version                   display version
         -o --output=filename           XML output filename
         -r --remote=remotename         Include only this remote in the export
         -d[debug_level] --debug[=debug_level]
```

The Lirc configuration file, defaulting to `lirc.conf` in the current directory, is read and translating into an output file, defaulting to `./`*basename-of-configfile*`.xml`. Per default, all remotes in the configuration file are translated to the output file, unless, using the `-r`/`--remote`-option is given, in which case only the selected remote is processed.

If configured, John Fine's library DecodeIR is invoked for each signal. It will try to identify the signal as one of a number of known IR signal protocol.

3.6.2.1 Limitation

The present version does not render toggle signals (more precisely: it renders them only for one value of the toggle). Also, the treatment of repeats may be incorrect. A bug related to repeats showed up in NEC1-like signal, and was fixed in version 0.1.3.

3.6.3 Installation

The following instruction is intended for users with some experience with installation programs under Linux.

1. (Optional, but highly recommended.) Install `libDecodeIR.so` to a directory like `/usr/local/lib`, where the linker will find it.
2. Download and unpack lirc-0.9.0. Probably other, similar, versions will also work.
3. Change to the top of the recently unpacked tree and apply the patch by `patch -p1 < path-to-lirc2xml.patch`.
4. Regenerate some autoconfig files by `autoreconf -f -i -s`.
5. Configure by, e.g., `./configure --with-driver=none --enable-decodeir --enable-debug`. Error messages from configure concerning `lirc_tell_me_what_version_is` can be ignored.
6. The command `make` will now, in addition to all the usual Lirc programs, create the `lirc2xml` program, located in the `tools` subdirectory.
7. `make install` will install it together with the rest of the programs in the Lirc package, per default in `/usr/local/bin`. Just copying `tools/lirc2xml` should (in this case) suffice (although in general not recommended practice).

3.6.4 Known bug

Command names in Lirc files containing less-than characters (<) and ampersands (&) will generate nonvalid XML. I have no intention to fix this in the short time span. It is hard to write waterproof code for that without using xml libraries; that kind of characters in command names is silly anyhow, and broken XML it can also easily be fixed with a text editor.

3.6.5 lirc2rmdu

Mainly as a proof of concept, I wrote a simple back-end to the lirc2xml program: *lirc2rmdu*, written in Python. (Possibly some would argue, that it should be called xml2rmdu, however I felt that would be more misleading.) It transforms the xml file from lirc2xml to a rmdu-file to be used by RemoteMaster program to design a "device upgrade" in the JP1 context. Note that, for several different reasons, the output of the program is not meant to be a perfect rmdu file, but rather a first starting point, however saving a lot of work transforming what can be automatically transformed. In particular,

the device number and -parameters need some fixing. possibly with a text editor outside of RemoteMaster -- actually I manage to "hang" the program on a few occations. Fixing this may be possibly by parsing RemoteMaster's `protocols.ini` file, but I am not sure that is a wise way of spending my time...

```
Usage:
        lirc2rmdu.py [-r remote] <xmlfile>
```

The program will extract the remote given by the -r argument (or the first one if not given), and write it to a file in the current directory, using a named taken from the remote's name.

3.6.6 Downloads

3.6.6.1 Sources etc

- lirc2xml.patch current version 0.1.3.
- lirc2rmdu

3.6.6.2 Binaries

- Binary for Linux 64bin. Just dropping in (e.g.) `/usr/local/bin`, while dropping `libDecodeIR.so` (64 bit version) (see below) in `/usr/local/lib` should be enough.
- Binary for Linux 32bin. (Unfortunately, the old version 0.1.2) Just dropping in (e.g.) `/usr/local/bin`, while dropping `libDecodeIR.so` (32 bit version) (see below) in `/usr/local/lib` should be enough.
- Cygwin binaries for Windows. (Unfortunately, the old version 0.1.2) This executable was created using a lot of dirty tricks... It was compiled under Cygwin, but contains the necessary Cygwin libraries, so it should be usable without Cygwin. Just unpack in an empty directory, start a DOS-box, and cd to said directory. Cygwin users should probably delete the Cygwin dlls, since they should already reside in their system, and may otherwise cause a conflict. It contains DecodeIR, but hard linked in, not as a dll.

3.6.6.3 Third party software

- lirc in current version, 0.9.0. Also "sufficiently similar" versions will do.
- DecodeIR; either the precompiled `libDecodeIR.so` (`DecodeIR.dll`) (version 2.43) for Windows, Linux (32 and 64 bit), and Mac OS X can be downloaded from the JP1-forum. The source code is also found in the JP1 forum. (Update: Current version is 2.44; the links go to 2.43 (which however is still usable) and needs to be updated.)

3.7 Improved LIRC driver for the Raspberry Pi

Date	Description
2014-01-12	Initial version.

Table 1: Revision history

3.7.1 Introduction

Using the Raspberry Pi, with its flexible GPIO pins, for LIRC IR control and reception is a natural wish. Sending IR diodes and IR receivers can easily be attached to the GPIO pins. Aron Szabo wrote a LIRC driver for the Raspberry, as a development of the LIRC serial driver. This article describes a further development of Aron's driver.

The present article assumes fundamental knowledge about LIRC and the Raspberry Pi. It is not a tutorial.

3.7.1.1 Copyright

3.7.2 Features

Next the features introduced will be described.

3.7.2.1 Multiple transmitter support

The RPi has at least 17 usable GPIO pins, so why should a LIRC driver only support using one? Furthermore, the LIRC driver API has `LIRC_SET_TRANSMITTER_MASK` ioctl. For the user, the `irsend` command the directive `SET-TRANSMITTERS` (taking as argument a list of numbers), so also LIRC can handle several transmitters (although very few drivers have implemented it). I, somewhat randomly, selected to support up to 8 transmitters. Should more be needed, only the compilation constant `LIRC_RPI_MAX_TRANSMITTER`. As a consequence, the module parameter `tx_mask`, which states the initial state of the transmitters selected, has been introduced.

I could not find any recommendation on the numbering of the transmitters in the LIRC documentation. Since the only driver implementing multiple transmitters (CommandIR) numbered the transmitters starting from 1, I have decided to do likewise.

3.7.2.2 Support for "modulation frequency" 0

Almost all modern IR protocols use a modulation frequency, however, some do not (Revox, Barco, Archer). Possibly more interesting, sending RF devices such as the RX433 are not designed to be fed with modulated signals. Many LIRC drivers, like Aron's, have a boolean parameter `softcarrier`, that govern the generation of the

modulation. If turned off, no modulation will be generated. However, being a module parameter, it cannot be changed in a running module, and it can also not have a different value for different transmitters.

The modification presented here simply allows the frequency parameter in the ioctr call LIRC_SET_SEND_CARRIER to have the value 0, in which case no carrier will be generated.

To my knowledge, other LIRC drivers no not support non-modulated IR signals in the sense of frequency = 0.

3.7.2.3 Inversion of outputs

The module parameter `invert`, which inverts the on/off-status of the outputs (for all transmitters) is new.

3.7.3 Installation

This module is no different from other LIRC modules when it comes to compiling and installation. There are a number of LIRC pages on the Internet.

Just copying the compiled binary module `lirc_rpi.ko` to the target system may work.

3.7.4 Module arguments

The module arguments are described in this section. These are parameters that are passed to the module on startup.

`gpio_out_pins` (array of integers)
GPIO output/transmitter pins of the BCM processor as array. The first is called transmitter #1 (not 0). Valid pin numbers are: 0, 1, 4, 8, 7, 9, 10, 11, 14, 15, 17, 18, 21, 22, 23, 24, 25. Default is none.

`gpio_in_pin` (integer)
GPIO input pin number of the BCM processor. Valid pin numbers are: 0, 1, 4, 8, 7, 9, 10, 11, 14, 15, 17, 18, 21, 22, 23, 24, 25. Default is none.

`sense` (bool)
Override auto detection of IR receiver circuit (0 = active high, 1 = active low).

`softcarrier` (bool)
Software carrier (0 = off, 1 = on, default on)

`invert` (bool)
Invert output (0 = off, 1 = on, default off)

`tx_mask` (integer)
Transmitter mask (default: 0x01)

`debug` (bool)
Enable debugging messages.

The name `gpio_out_pins` is incompatible with Aron's version, using the name `gpio_out_pin`. This also goes for the default values: Aron uses 18 and 17 for

`gpio_in_pin` and `gpio_out_pin`, my version has both undefined as default. To minimize confusion, I have decided just to publish one version of the driver.

3.7.5 Further work

I would be very interested in getting the driver to work with a non-demodulation sensor, like the QSE159 (see this article), to arrive at an estimate of the modulation frequency.

Except for this, I do not plan to continue the development, or to "maintain" the driver.

3.7.6 Downloads

- lirc_rpi.c. The source.
- lirc_rpi.ko. Compiled kernel module. Sometimes it suffices just to copy this to the RPi.

3.7.7 Resources

- Aron's site.
- Sort-of announcement on LIRC mailing list.
- Diskussion thread on the driver. This driver is discussed on page 6.
- Another guide on LIRC on the RPi.

3.8 Raspberry Pi daughter board for IR and RF transmission and reception

Date	Description
2014-01-11	Initial version.

Table 1: Revision history

3.8.1 Description

This article shows a daughter board for fitting to the GPIO Pins of a Raspberry Pi. It is mainly intended for use with LIRC. It necessitated the development of an an improved LIRC driver.

It is built upon a "TriPad" board. It contains of a 2 x 13 board connector and a number of standard components. There is an IR transmitting LED (SFH415) with an NPN driver transistor (BC547) driver transistor. It is designed to be able to withstand 100mA of *continuous* current through the IR LED. There is also an IR-demodulator TSOP4838. For the LIRC world uncommon is the non-demodulating IR sensor QSE159, which makes it possible to determine the modulation frequency of a captured signal. This works with 5V supply voltage, that could cause a problem for the 3.3V operating GPIO pins. Selecting an open collector model like QSE159 solves this problem however. Unfortunately, at the time of this writing, supporting drivers have not been written yet.

Also contained are an RF 433MHz sender (TX433) and Receiver (RX433). The wire coils are antennas for these. Since the photos were made, an 100nF capacitator has been added between power supply and ground of the RF receiver, to reduce disturbances. From the LIRC drivers, the receiver is addressed exactly like a demodulating IR-receiver, while the transmitter should not be feed with a modulated signal, thus posing another software problem for LIRC. This will be discussed further here.

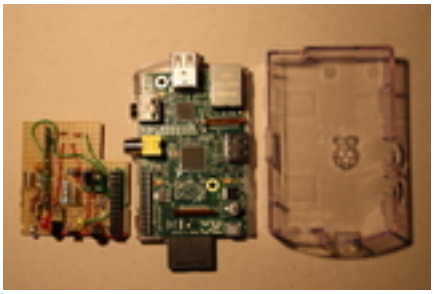
There are also a number of "Debug-LEDs", one of them being a "dummy-IR", to demonstrate the multi-transmitter facility of the improved LIRC driver.

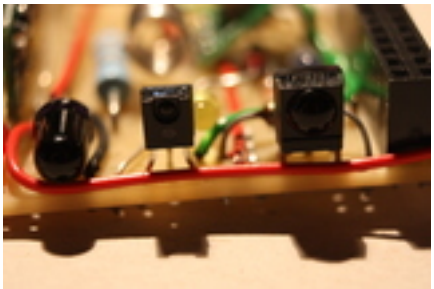
Hooking up these components to the GPIO pins of the Raspberry has been described in many places on the Internet. To be mentioned is in particular Aron Szabo's page. Therefore, we do not describe the circuit in detail.

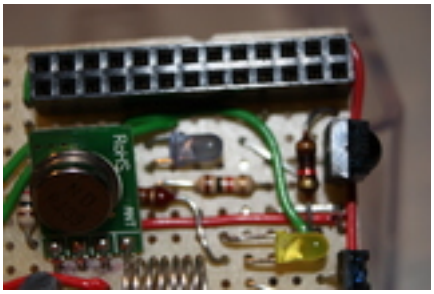
As can be seen on the pictures, I use a case from www.oneninedesign.co.uk. The daughter board fits quite nicely therein. Since it is transparent, it does not interfere with the IR functionality. At least I think/hope that it is transparent also for infrared wavelengths...

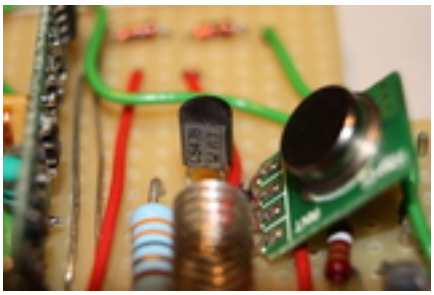
Using the hardware herein to send RF commands for controlling RF switches will be discussed in a future article.

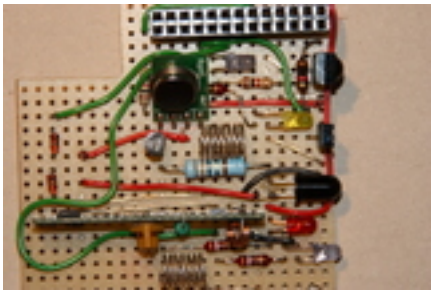
3.8.2 Photos

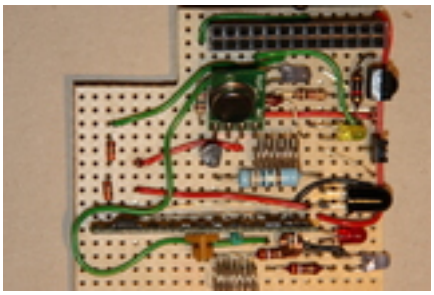


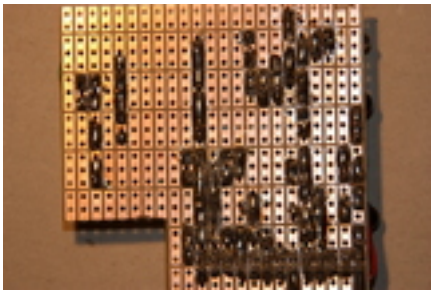


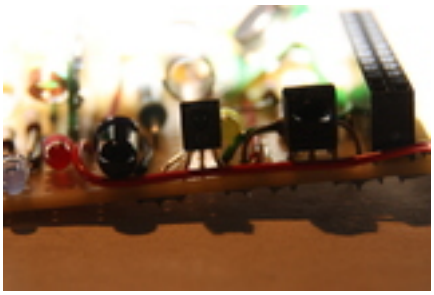


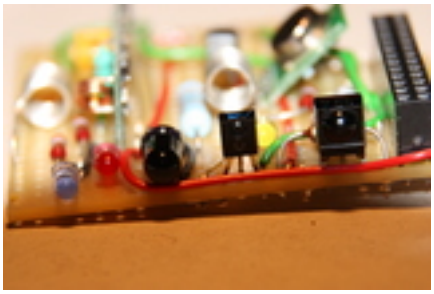


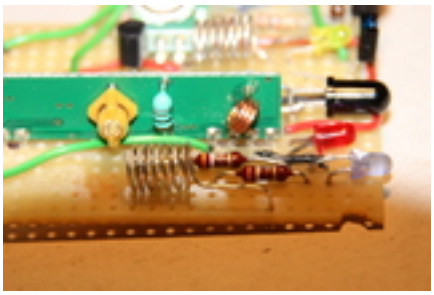


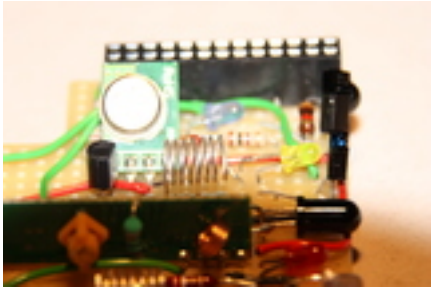


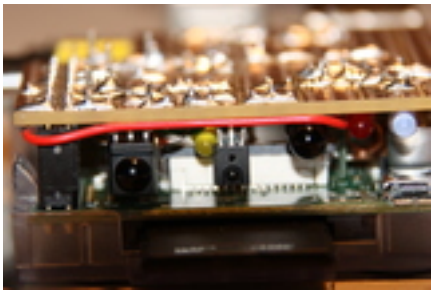


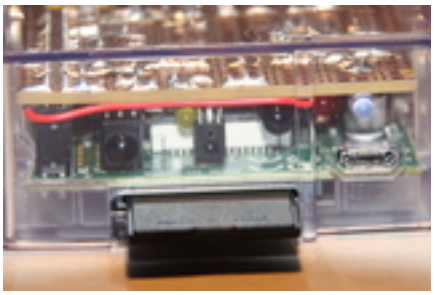


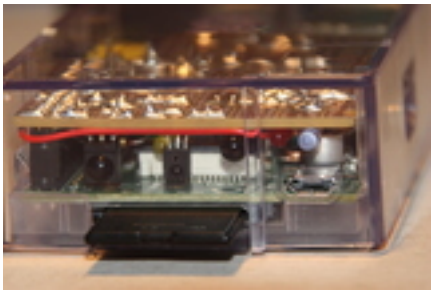












3.9 Enabling LIRC to send CCF signals not residing on the server

Date	Description
2012-04-30	Initial version.
2015-09-10	Updated for the current version.

Table 1: Revision history

3.9.1 Introduction

Some networked devices used for sending IR codes do not store the IR signals locally, but have them sent on every invocation. This goes for e.g. the GlobalCaché IR senders. Other devices, like the LIRC server (`lircd`) keep its own data base of known signal. It can (if started with the --listen option) respond to commands of the type "send *command* from the remote *remote number* of times. It can only send those commands in its data base, typically in the file `/etc/lirc/lircd.conf` (previously `/etc/lircd.conf`). Some devices, like the IRTrans Ethernet with IR Database can both accept commands of the type remote/command and IR signals in raw form.

It is hard to say that one method is universally better than the other. "Keeping the data base on the server, not on the clients" certainly sounds like a good idea in general, however, "the server" is these days not necessarily a big, "fat" machine, and the clients are not necessarily small and dumb. A *server* is by definition something that offers services to clients, but may be small and weak in terms of resources. Furthermore, there may be several such, possibly running on comparatively weak and primitive embedded devices, with limited memory, allowing only a "small" number of flash memory updates, etc. When updating, the database needs to be distributed, possibly a protocol like ssh needs to be implemented for this, etc. Furthermore, as should be clear from my program IrScrutinizer, sometimes the use case is to experimentally try out unknown IR signals. To generate a database for this test, download it to the "server" just in order to perform the tests does not seems like a good idea.

I have submitted a previous version of the patch to the then-maintainer Christoph Bartelmus, who was vehemently against it, meaning that it is cleaner if data bases reside on the server. The issue has been discussed on the LIRC mailing list also when another person submitted a similar patch.

The here presented patch enhances the LIRC server `lircd` to be able to accept commands in the listening socket to send an arbitrary IR signal in Pronto CCF format. Mostly for illustration, the fix also patches the simple client program `irsend` to be able to send these signals.

Implementing the same functionality on WinLirc is non-trivial, since it has completely different code.

The idea of patching LIRC to allow this functionality was originally suggested to me in 2006 by Bitmonster, the creator of the original version of Eventghost.

3.9.2 Installation

The following instruction is intended for users with some experience with installation programs under Linux.

The current version is found as a fork in the Lirc sources on Sourceforge.
Clone that fork using a command like `git clone ssh://bengtmartensson@git.code.sf.net/u/bengtmartensson/lirc u-bengtmartensson-lirc`. Then build using the normal instructions. (Note that these have changed somewhat the last few years.)

3.9.3 Extensions to the irsend command

The `irsend` command is contained in the LIRC distribution and (in its unpatched form) is documented here. This patch enhances it with the two *DIRECTIVES* `SEND_CCF_ONCE` and `SEND_CCF_START`. Example:

```
irsend SEND_CCF_ONCE 0000 006C 0022 0002 015B 00AD 0016 0016 0016 0041 0016 0016 0016
 0041 0016 0041 0016 0041 0016 0041 0016 0016 0016 0041 0016 0016 0016 0041 0016 0016
```

```
 0016 0016 0016 0016 0016 0016 0016 0041 0016 0016 0016 0041 0016 0016 0016 0041 0016
 0041 0016 0016 0016 0016 0016 0016 0016 0041 0016 0016 0016 0041 0016 0016 0016 0016
 0016 0041 0016 0041 0016 0041 0016 0699 015B 0057 0016 0EA3
```

sends a pre-rendered NEC1, Device=122, Command=29-command to the LIRC connected device. This turns out to be the power-on command for Yamaha receivers like the RX-V 1400. Another example, assuming that a suitable Yamaha receiver is connected, the command

```
irsend SEND_CCF_START 0000 006C 0022 0002 015B 00AD 0016 0016 0016 0041 0016 0016 0016
 0041 0016 0041 0016 0041 0016 0041 0016 0016 0016 0041 0016 0016 0016 0041 0016 0016
 0016 0016 0016 0016 0016 0016 0016 0041 0016 0016 0016 0041 0016 0016 0016 0041 0016
 0041 0016 0016 0016 0016 0016 0016 0016 0041 0016 0016 0016 0041 0016 0016 0016 0016
 0016 0041 0016 0041 0016 0041 0016 0699 015B 0057 0016 0EA3
```

sends the Volume+-command (NEC1 Device=122, Command=26) "forever" (until it hits `--repeat-max`). Warning: this may have "interesting" side effects on your loudspeakers, your amplifier, your ears, or your neighbors... :-). Do not try this at home...

3.9.4 Known bugs/limitations

The handling of repetitions is not correct. This is due to a limitation in LIRC, that cannot represent a complex intro signal *and* a complex repeat signal.

It would have been a good idea if the patch also modified the LIRC version string, e.g. by appending "`_ccfpatch`".

3.9.5 Downloads

- The old version of the patch for Lirc 0.9.0. *Of historical interest only. Do not use with the current version.*

4 API Documentation

4.1 Index of API documentation

Date	Description
2019-08-12	Initial version.
2020-01-14	Updated with new locations.

Table 1: Revision history

4.1.1 Javadoc index

- DevSlashLirc Java API documentation
- Girr API documentation
- HarcHardware API documentation
- IrpMaster API documentation

- IrScrutinizer API documentation
- IrpTransmogrifier API documentation
- Jirc API documentation

4.1.2 Doxygen index

- DevSlashLirc C++ API documentation
- Infrared4Arduino.
- AGirs.

4.2 IrpMaster 1.4.2 API

5 Downloads

5.1 HARCToolbox downloads

Date	Description
2012-05-01	Initial version.
2012-06-06	Updated for the releases 0.2.0 of IrMaster and IrpMaster.
2012-08-19	Updated for the release 0.3.0 of IrMaster and 0.2.1 of IrpMaster.
2012-11-18	Updated for the release 0.3.1 of IrMaster and 0.2.2 of IrpMaster.
2013-02-17	Updated for the release 0.1.3 of lirc2xml.
2014-02-02	Updated for the current setup.
2014-06-12	Updated for the IrScrutinizer 1.1.0 etc.
2014-09-27	Updated for the IrScrutinizer 1.1.1 etc.
2015-04-16	Updated for the IrScrutinizer 1.1.2 etc.
2015-09-10	Updated for the IrScrutinizer 1.1.3 etc.
2016-04-30	Updated for the IrScrutinizer 1.2 etc.
2016-08-30	Updated for the IrScrutinizer > 1.2 etc.
2017-07-09	Somewhat reorganized.

Table 1: Revision history

5.1.1 Introduction

The current software is distributed through the Github repositories. This page servers two purposes:

- It provides some convenience links to the Github repositories.
- It provides links to some older versions, and some other files, not found on Github.

5.1.2 Current software

- Latest released version of IrScrutinizer.
- Snapshot of the most current, non-released, IrScrutinizer.

Other software is available in the Github repositories.

5.1.3 Old versions

Older versions of my software is made available here.

- IrMaster
  - IrMaster 1.0.1 setup.exe for Windows.
  - IrMaster 1.0.0 sources
  - IrMaster 1.0.0 binaries, including third-party libraries
  - IrMaster version 0.1.2 sources.
  - IrMaster version 0.1.2 binaries, including IrpMaster binaries and third party libraries.
  - setup.exe for Windows of IrMaster version 0.1.2, including IrpMaster binaries and third party libraries.
  - IrMaster version 0.2.0 sources.
  - IrMaster version 0.2.0 binaries, including IrpMaster binaries and third party libraries.
  - setup.exe for Windows of IrMaster version 0.2.0, including IrpMaster binaries and third party libraries.
  - IrMaster setup.exe for Windows of IrMaster version 0.3.0, including IrpMaster binaries and third party libraries.
  - IrMaster version 0.3.0 sources.
  - IrMaster version 0.3.0 binaries, including IrpMaster binaries and third party libraries.
  - IrMaster version 0.3.1 sources.
  - IrMaster version 0.3.1 binaries, including IrpMaster binaries and third party libraries.
  - IrMaster setup.exe for Windows of IrMaster version 0.3.1, including IrpMaster binaries and third party libraries.
- IrScrutinizer (*current version found here.*)

  - HarctoolboxBundle-1.2-sources, the combined sources for IrScrutinizer 1.2, IrpMaster, Girr, HarcHardware.
  - IrScrutinizer 1.1.3 binaries, including third-party libraries,
  - IrScrutinizer 1.1.3 setup.exe for Windows,
  - IrScrutinizer 1.1.3 as Mac OS X App in compressed format,
  - HarctoolboxBundle-1.1.3-sources, the combined sources for IrScrutinizer 1.1.3, IrpMaster, Girr, HarcHardware.
  - IrScrutinizer 1.1.2 binaries, including third-party libraries
  - IrScrutinizer 1.1.2 setup.exe for Windows
  - IrScrutinizer 1.1.2 as App in compressed disk image for Mac OS X
  - IrScrutinizer 1.1.1 sources
  - IrScrutinizer 1.1.1 binaries, including third-party libraries
  - IrScrutinizer 1.1.1 setup.exe for Windows
  - IrScrutinizer 1.1.0 setup.exe for Windows
  - IrScrutinizer 1.1.0 sources
  - IrScrutinizer 1.1.0 binaries, including third-party libraries
  - IrScrutinizer 1.0.0 sources
  - IrScrutinizer 1.0.0 binaries, including third-party libraries
  - IrScrutinizer 1.0.0 and IrMaster 1.0.0 setup.exe for Windows, in one package!
  - IrScrutinizer version 0.1.1 sources.
  - IrScrutinizer version 0.1.1 binaries, including IrpMaster binaries and third party libraries.
  - IrScrutinizer setup.exe for Windows of IrMaster version 0.1.1, including IrpMaster binaries and third party libraries.
- IrpMaster
  - IrpMaster 1.0.2 sources
  - IrpMaster 1.0.1 sources
  - IrpMaster 1.0.0 sources
  - IrpMaster version 0.1.2 sources.
  - IrpMaster version 0.2.0 sources.
  - IrpMaster version 0.2.1 sources.
  - IrpMaster version 0.2.2 sources.
- Misc
  - HarctoolboxBundle-1.0.0-sources, the combined sources for IrScrutinizer 1.1.2, IrpMaster, Girr, HarcHardware. (GuiComponents has been integerated into IrScrutinizer.)
  - Girr 1.0.2 sources
  - Girr 1.0.1 sources
  - Girr 1.0.0 sources
  - Jirc 0.3.0 sources
  - Jirc 0.2.0 sources
  - GuiComponents 0.2.2 sources
  - GuiComponents 0.2.1 sources

    - GuiComponents 0.2.0 sources
    - HarcHardware 0.9.2 sources
    - HarcHardware 0.9.1 sources
    - HarcHardware 0.9.0 sources
    - HarcHardware 0.8.0 sources
    - IrCalc 0.2.0 sources from my Subversion repository.
    - Lirc 0.9.0 patch for CCF processing.
    - lirc2xml.patch version 0.1.3.
    - lirc2rmdu.py. A post-processor for lirc2xml.
- Old HarcToolbox
    - Harctoolbox 0.5.0 (then called "Harc"). Source and binaries included.
    - Harctoolbox 0.6.0 sources.
    - Harctoolbox 0.6.0 binaries.
    - Harctoolbox 0.7.0 sources from my Subversion repository.
    - Harctoolbox 0.7.0 binaries, including third part jar-archives.
- lirc2xml
    - lirc2xml-0.1.2-windows.zip
    - lirc2xml-0.1.2.patch
    - lirc2xml-0.1.3.patch
    - lirc2xml.patch
    - lirc2xml_amd64
    - lirc2xml_amd64_0.1.2
    - lirc2xml_i386

5.1.4 Old IR protocols

> **Note:**
> These files are now considered obsolete, since IrScrutinizer, IrMaster and IrpMaster can generate these protocols, and export them in a number of different formats.

- Intertechno.xml Pronto codes for the Intertechno (and similar) RF switches ("with code wheel").
- conrad.txt Pronto codes for the RF switches once manufactured by Conrad electronics, sometimes known as RS-200.

6 All

6.1 Site Linkmap Table of Contents

This is a map of the complete site and its structure.

- MyProj _____ *site*
    - Home _____ *home*
        - Welcome _____ *index* : Homepage

    - Improved lirc_rpi driver _____ *lirc_rpi* : Improved LIRC driver for the Raspberry Pi
    - Raspberry Pi IR & RF hardware _____ *rpi_daughterboard* : Raspberry Pi IR and RF hardware daughter board
    - Lirc CCF patch _____ *lirc_ccf* : Making Lirc server handle CCF signals
- API Documentation _____ *api_documentation*
    - API doc. index _____ *index* : API documentation (javadoc)
    - DevSlashLirc Java API doc _____ *devslashlirc_doc_java* : DevSlashLirc Java API documentation
    - DevSlashLirc C++ API doc _____ *devslashlirc_doc_cpp* : DevSlashLirc C++ API documentation
    - Girr API doc _____ *girr_doc* : Girr API documentation
    - HarcHardware API doc _____ *harchardware_doc* : HarcHardware API documentation
    - IrpMaster API doc _____ *irpmaster_doc* : IrpMaster API documentation
    - IrScrutinizer API doc _____ *irscrutinizer_doc* : IrScrutinizer API documentation
    - IrpTransmogrifier API doc _____ *irptransmogrifier_doc* : IrpTransmogrifier API documentation
    - Jirc API doc _____ *jirc_doc* : Jirc API documentation
    - Infrared4Arduino API doc _____ *infrared-apidocs* : Infrared4Arduino API documentation
    - AGirs API doc _____ *agirs-apidocs* : AGirs API documentation
- Downloads _____ *downloads*
    - Download index _____ *index* : Download cool software!
- All _____ *all*
    - Sitemap _____ *linkmap* : Site Linkmap
    - Whole Site HTML _____ *whole_site_html*
    - Whole Site PDF _____ *whole_site_pdf*