

# General InfraRed Server Command Language

## Table of contents

1 Requirements.....	2
2 Specification.....	2
2.1 Introduction.....	2
3 Modules.....	3
3.1 Base.....	3
3.2 NamedRemotes.....	3
3.3 UeiRemotes.....	3
3.4 OutputDevices.....	4
3.5 InputDevices.....	4
3.6 Transmitters.....	4
3.7 Transmit.....	5
3.8 Capture.....	5
3.9 Receive.....	5
3.10 Store.....	6
3.11 Command.....	6
3.12 Authentication.....	7
3.13 Charset.....	7
4 Communication.....	7

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

Table 1: Revision history

## 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 Specification

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

### 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.

## 3 Modules

### 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.

### 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.

### 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.

**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 capacities. Possible values (extensible): fmax, fmin, zero\_frequency\_tolerant. Inherit to transmitters.

**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

**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 capacities. 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.

### 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 supported 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)

### 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.

### 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)

**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)

**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.

### 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**

### 3.13 Charset

Determines charset used for input and output.

**charset**

- argument: charset name.

## 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 abbreviated as much as unambiguity 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.