# Manager System installation

This section will describe how to download and install a copy of Manager System on a linux machine.

* How to install Manager System
  + Hardware dependencies.
  + Minimum requirements.
  + Decompress and instalation.
  + Tunning sytem to work with Manager System
  + How to download a virtual machine that works with Manager System
* How to delete Manager System from a machine
  + Folder's to remove
  + Remove system modifications after instalation.
* How to contribute to Manager System proyect.
  + How to contribute with new plugins
  + How to contribute with code improvements
  + How to contribute with translations

# Manager System Plugin documentation

This section will describe how to make a plugin for Manager System.

* Manager System architecture.
  + Core
  + Sections
  + Plugins
  + Plugins and Manager System API
* Sections
  + Section skeleton
  + How to make a new section step by step.
* Plugins
  + Plugin skeleton
  + How to make a new plugin step by step.

# Manager System API documentation

Those are the API categories with documentation of all the functions available in the Manager System

* 1. API
     + AJAX
     + Input validation
     + Configuration files
     + Interfaces
       - Network Interfaces manipulation
       - Bluetooth interfaces manipulation
       - GPRS interface manipulation
     + Services
       - DHCP
       - OLSR
       - Fresnel configuration
       - Join networks
     + Tools
       - GPS

## Manager System installation

#### Introduction

The following document describes how to download, install or delete Manager System . Our aim is to provide a portable Open Source Manager System so we hope to complete this guide with information for installing on several architectures and distributions, but right now this document references to architecture and Debian distribution, but as usual on linux distributions this installation can be ported to any linux distribution without much work.

#### How to get Manager System

Manager system can be downloaded as a tar.gz file with all the contents needed, or as a .deb package for debian systems.

#### How to install Manager System

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 2 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>

#### Hardware dependencies.

As told before, Manager System has been developed and tested on architecture. architecture is x86 based. So Manager System should work on any x86 32bit Debian.

Some of the base plugins are developed to work with some of the components and it's particularities. Fortunately, you can modify plugins to work with your hardware and contribute to community with plugins adapted to different hardware. The plugin structure of Manager System make it easy to add, modify or study plugins.

#### Minimum requirements.

The installation of Manager System requires Apache and PHP5 mod installed and running. Probably Manager System should works on other web servers that support PHP5 like Lighttpd.

#### Decompress and instalation.

Once the tar.gz file has been downloaded, those are the steps needed to install Manager System:

* + - Open a terminal and navigate to the path that contains the Manager System tar.gz file.
    - To decompress the file execute:

tar zxvf filename.tar.gz

* + - Move the uncompressed data to a path that can be read by apache daemon. For example to /var/www

sudo mv path\_to\_the\_folder /var/www

* + - Change ownership to www-data

sudo chown -R www-data:www-data /var/www/folder\_name

* + - Open firefox and point web browser to load the folder.

<http://localhost/folder_name>

#### Tunning sytem to work with Manager System

Manager System usually has to change system files. To make it possible you can add www-data to **sudoers** and configure it for no password need. This can be a security hole. So be sure web application on server are secure before doing it.

#### How to download a virtual machine that works with Manager System

A Debian Lenny virtual machine will be released with Manager System installed on it, so any user may try the Manager System and develop over it.

#### How to delete Manager System from a machine

To delete Manager System from the system the folder containing Manager System should be removed and any change made to system files should be reverted.

#### Folder's to remove

Remove folder containing Manager System . If you have changed ownership to **www-data** you may need to remove folder as **root**. So the user that will remove the folder should be on sudoers list.

sudo rm -rf /var/www/folder\_name

#### Remove system modifications after instalation.

Revert changes made to **sudoers** file if needed. (We recommend the use of **visudo** for this)

Remove following line:

%www-data ALL=NOPASSWD: /var/www//bin/

#### How to contribute to Open Source Router Web Manager project.

Manager System has been configured and tested to work on.

#### How to contribute with ports to other architectures

Open Source Router Web Manager has been tested to work with but can be ported to other architectures. Modifications needed to make Open Source Router Web Manager works on other systems can be submitted and will be available for download with Manager System.

#### How to contribute with new plugins

Router services management is usually hardware dependent. That makes that each router and distribution needs it's own plugins, or need to make modifications to plugins to work with different hardware vendors.

I you develop or modify a plugin you can submit it to share it with the community.

Plugins will be available in different stages (stable and testing) and will be sorted by architecture or distribution too.

#### How to contribute with code improvements

Open Source Router Web Manager code is available for download. Probably you will find code improvements or security patches. You can report/submit all the stuff you find to improve Open Source Router Web Manager stability and performance.

#### How to contribute with translations

English is the default language for Open Source Router Web Manager. You can help community with translations to other languages.

# Manager system Plugin documentation

## Manager System architecture

Manager System 1.0 was a monolithic application, adding new functionalities was had and difficult. Our aim when redesigning Manager System was to make an application where adding new functionalities should be as easy as possible.

To achieve that the application architecture defines three main levels. Core, sections and plugins.

### Core

Core is just the functions and control structures needed to navigate thought sections and load plugins. Core differentiate between plugin load and plugin server and loads the necessary plugin files. Core also gives a public API with common functions that can be used by plugins.

### Sections

Sections are just plugin containers. A section will contain related plugins, in that way Core will display plugins within a section to allow an easy navigation.

### Plugins

Plugins are the hearth of Manager System . Plugins are applications that control, modifies or present system parameters. Plugins are grouped in sections.

All the Manager System architecture is designed to make plugin development as easy as possible. So plugins could be developed, studied or modified without need to modify or know Core behaviour.

## Sections

As told before, sections are plugin containers. Now the internal structure of a subsection will be detailed and a tutorial will show how to create a new section step by step.

### Section skeleton

Subsection skeleton is quite simple and easy. A section is composed by a folder and a configuration file. A folder has to be created on plugins folder on Manager System. Inside the folder, a file named configuration.php has to be created with the following content:

<?php

$type="SELECTOR"; // This must contain a SELECTOR VALUE and just is used for integrity check.

$section\_name="NAME OF THE SECTION";

$section\_description="Skeleton plugin for study propourses";

$section\_icon="section.png"; // Icon to use when section not selected

$section\_icon\_selected="section-hv.png"; // Icon to use when section is selected

?>

The content is just a simple piece of PHP code that declares some variables and two default images that can be used to make custom icons.

section.png is the base designed to make the section icon when section is not selected, and section-hv.png is the base designed to make the section icon when section is selected.

The variables that should be defined on configuration.php are:

* $type="SELECTOR"; This is just a control variable. Any section without this variable declared in that way will not be considered a section and will be ignored by the core.
* $section\_name. This variable contains the name of the section and will be displayed by the core. Note that folder name and section name can be different.
* $section\_description. This variable should be filled with a brief description of the content that will be inside this section.
* $section\_icon. This variable should have a relative path to an icon. If this variable points to a valid icon, core will use that icon on navigation menus when the section is not selected.
* $section\_icon\_selected. This variable should have a relative path to an icon. If this variable points to a valid icon, core will use that icon on navigation menus when the section is selected.

### How to make a new section

How can I create a new section? Well that is really easy. In some steps we will create a new section for Manager System.

We want to create a new section called "Devices" that will contain plugins that interacts with system devices.

Those are the steps needed to create the new section.

1. Create a new folder on plugins folder on the Manager System installation path.

mkdir -p /var/www/manager\_system/plugins/devices

1. Copy to the folder that you have just created the icon that will be used to display de section. You can download and use the following icons:



1. With your favourite text editor create a file named configuration.php on the folder created on step 1, and copy the following content:

<?php

$type="SELECTOR"; // This must contain a SELECTOR VALUE and just is used for integrity check.

$section\_name="Section Devices";

$section\_description="System devices management plugin";

$section\_icon="devices.png";

$section\_icon\_selected="devices-hv.png";

?>

With those three simple steps a new section (empty right now) has been added to Manager System.

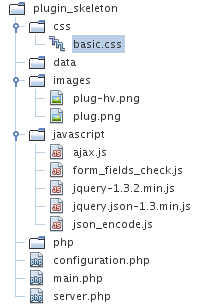
## Plugins

Structure has been designed to make it easy to share, update, modify, delete and study the plugins. To achieve this, plugins should place all the files that they need inside it's own folder structure. This makes really easy move plugins from one section to another, share new plugins with others and modify a plugins with new updates. To study a plugin the fact that all the files that the plugin can use (except the API provided by the core) makes really easy to follow how a plugin works and how to change or improve it's behavior.

### Plugin skeleton

The simplest plugin needs just a folder and two files. The folder will contain all the plugin files and the two fundamental files needed are a configuration file very similar to the configuration file used with sections, and a main PHP file that will be loaded by core. With this minimal configuration, display file will be called by core when the plugin is loaded. But will be also called by core when a request to the plugin is made. To avoid complexity is recommended the use of a display file and a server file, so the minimum number of files for a plugin can be just two but three is recommended.

For more complex plugins, core will search for two main files. One will be used when the plugin is loaded and the other, the server file, will be used when an AJAX request is launched against the plugin. Plugin in fact has two different behaviors, presentation and server. Presentation will show initial display. Server instead will be loaded only when a request to the plugin has been made, and will receive data from request to perform actions. Server can return more data to be displayed too. With Manager System an AJAX framework has been provided to make a easy task the communication between client and server side of the plugin. This is the standard plugin skeleton provided with Manager System :



Now let's detail all the folders and files.

1. Configuration.php
   * This file declares to the core the plugin. Core will check this file and use it's data to assure that the plugin is a valid plugin. Any plugin with bad data on this file or without this file will be ignored by the core.
2. main.php
   * This is the standard display file. It's name can be changed to any other, but should be updated on the configuration file. This file will be loaded by the core when the plugin is loaded. Any plugin without a display file will be ignored by the core.
3. server.php
   * This is the plugin server file. It's name can be changed also but should be declared on configuration file too. This file will be loaded when core detects a request to the plugin.
4. CSS folder.
   * CSS files used by the plugin should be placed here.
   * A basic css file is provided with the plugin skeleton but as ever you can change it's name or have as many css as you want.
5. Data folder
   * This folder can be used by the plugins to store local data or as a temporal data folder. Skeleton contains this folder but the folder is empty by default. You can create as many subfolders as you need and as many files as you need.
6. Images folder
   * This folder should be used to store images, icons and multimedia files that will be used by the plugin. You can create as many subfolders as you need and as many files as you need.
   * The skeleton has two images that can be used as base to make your own icons, plug.png is the image that is designed to be used when the plugin is not selected, and plug\_hv.png is the image that is designed to be used when the plugin is selected.
7. Javascript folder
   * This folder should contain the javascript files used in the plugin
   * Ajax framework files, form validation and jquery are provided with the skeleton.
8. Php folder
   * This folder should contain any other php needed in addition to configuration.php, server.php and main.php.

The use of this folder distribution is recommended but not mandatory. The only mandatory files needed are configuration.php and the files that are declared in configuration.php.

configuration.php structure:

<?php

$type="PLUGIN"; // This is just for integrity checks.

$plugin\_name="NAME OF THE PLUGIN"; // THIS SHOULD BE A LINE.

$plugin\_version="VERSION STRING"; // THIS SUOULD BE A LINE.

$plugin\_author="AUTHOR INFORMATION"; //THIS SHOULD BE A LINE.

$plugin\_description="DESCRIPTION OF THE PLUGIN"; // THIS SHOULD BE A SMALL DESCRIPTION

$plugin\_main\_file="main.php"; // BETER IF USED THE STANDARD main.php

$plugin\_server\_file="server.php"; // BETER IF USED THE STANDARD server.php

$plugin\_icon="images/plug.png"; // BY DEFAULT

$plugin\_icon\_selected="images/plug-hv.png"; // BY DEFAULT

?>

The file format is quite similar to the configuration.php file for a section. Those are the variables that should be included commented.

* $type="PLUGIN"; This is just a control variable. Any plugin without this variable declared in that way will not be considered a plugin and will be ignored by the core.
* $plugin\_name. This variable contains the name of the plugin and can be used by the core to display information about the plugin. Note that folder name and plugin name can be different.
* $plugin\_version. This is information about the version of the plugin. This version number can be managed by plugins to automatically detect new versions and make a plugin for update installed plugins.
* $plugin\_author. Name of the author, or authors of the plugin.
* $plugin\_description. Brief description of the plugin and what is designed to do.
* $plugin\_main\_file. This is the file that will be called by core when the plugin is loaded by client. This must point to a valid file, if don't core will mark plugin as invalid and not display plugin information.
* $plugin\_server\_file. This is the file that will be called by core when an request to the plugin is done by the client. This must point to a valid file, if don't core will mark plugin as invalid and not display plugin information. This file can be the same as $plugin\_main\_file, but is recommended to manage display and server request on different files.
* $plugin\_icon. This icon will be used in navigation menu when the plugin is not selected.
* $plugin\_icon\_selected. This icon will be used in navigation menu when the plugin is selected. main.php structure:

A plugin can interact with core in order to modify core behavior or get information from core variables. This is a list of all those variables that can be used by a plugin and what are those variables designed for:

* Core interaction variables
  + **$\_main\_title**. This variable will load the page title.
  + **$\_plugin\_css**. You can define an array with the css files you want to load. The css must be on the plugin css folder.
  + **$\_plugin\_javascript**. You can define an array with the javascript files you want to load.

Javascript files must be under the plugin javascript folder.

* Predefined variables
  + **$section** contains the section folder name.
  + **$plugin** contains the plugin folder name.
    - $section and $plugin can be used to make a link to this plugin by just reference. For example:

$html="<a href=\"index.php?section=$section&plugin=$plugin\">This plugin</a>";

* + **$base\_plugin** contains the path that must be used as start to includes for plugin includes that need the local path.
  + **$url\_plugin** contains the url base that must be used to include html items such as images.
  + **$API\_core** contains the path to the core API folder.
* Output variable
  + **$html**. Once plugin is finished core will check $html variable and output its content if any is stored. Is better to use $html variable to avoid direct call of the plugin from browsers.

server.php structure:

Server can access to some core variables too. This is a list of all those variables that can be used by a server side of a plugin and what are those variables designed for:

* Predefined variables
  + **$section** contains the section folder name.
  + **$plugin** contains the plugin folder name.
    - **$section** and **$plugin** can be used to make a link to this plugin by just reference. For example:

$html="<a href=\"index.php?section=$section&plugin=$plugin\">This plugin</a>";

* + **$base\_plugin** contains the path that must be used as start to includes for plugin includes that need the local path.
  + **$url\_plugin** contains the url base that must be used to include html items such as images.
  + **$API\_core** contains the path to the core API folder.

#### Output data

* + Server produced data will returned to the ajax call that made the request directly. This is made to give server side a total control over the output headers and data.

### How to make a new plugin

How can I create a new plugin? Well that is really easy as it was to create a new section. In some steps we will create a new plugin for Manager System .

We want to create a new plugin that will be located inside the section called "Devices". The plugin will show mounted partitions, then an AJAX call will be made to request more information about the disk usage of the partition. Those are the steps needed to create the new plugin.

1. Create a new folder for the plugin on section folder.

mkdir -p /var/www/manager\_system/plugins/devices/disk\_usage

1. Copy the plugin skeleton to the new folder. Tutorial will assume that the plugin skeleton can be found on

/home/user/plugin\_skeleton (Note that a plugin skeleton is shipped with the Manager System .

cp -rf /home/user/plugin\_skeleton/\*

/var/www/manager\_system/plugins/devices/disk\_usage/.

1. Now open the configuration.php file and modify values for our new plugin.

<?php

$type="PLUGIN"; // This is just for integrity checks.

$plugin\_name="Disk usage"; // THIS SHOULD BE A LINE.

$plugin\_version="0.1"; // THIS SUOULD BE A LINE.

$plugin\_author="Octavio Benedí"; //THIS SHOULD BE A LINE.

$plugin\_description="Plugin for monitoring disk usage."; // THIS SHOULD BE A SMALL DESCRIPTION

$plugin\_main\_file="main.php"; // BETER IF USED THE STANDARD main.php

$plugin\_server\_file="server.php"; // BETER IF USED THE STANDARD server.php

$plugin\_icon="images/disk-usage.png"; // BY DEFAULT

$plugin\_icon\_selected="images/disk-usage-hv.png"; // BY DEFAULT

?>

1. Next step is to prepare the main.php file, to show data when the plugin is loaded. Just open main.php and prepare the php script.

<?php

// Title of the page:

$\_main\_title="Disk usage management";

// Load our own CSS stylesheet.

$\_plugin\_css=Array("basic.css");

// Load javascript that will be used by the plugin.

$\_plugin\_javascript=Array("jquery-1.3.2.min.js","ajax.js");

// Scan partitions mounted on system: exec('mount -l',$mounted\_partitions);

// Prepare each line to be displayed as html foreach ($mounted\_partitions as $partition)

{

$partition\_list=explode(' ',$partition);

$partitions\_html.='<div><span>'.$partition\_list[0].' mounted on '.$partition\_list[2].'</span></div>';

}

$html='<fieldset><legend>Partitions on system</legend>

<div class="information">'.$partitions\_html.'</div>

<input type="button" onclick="simple\_ajax\_call(\'\',\'output\',\''.$section.'\',\''.$plugin.'\')" value="Display Disk Usage">

</fieldset>

<div id="output"></div>';

?>

1. Our server will return html with the disk usage each time is called.

<?php

include\_once $API\_core.'complex\_ajax\_return\_functions.php';

$html="<fieldset><legend>Disk usage information on the server</legend>";

$html.="<h3>Information received on: ". date(DATE\_RFC822)."</h3>";

$html.='<div class="information>';

exec('df -h',$disk\_usage); foreach ($disk\_usage as $disk)

{

$html.='<div>'.$disk.'</div>';

}

$html.='</div></fieldset>'; echo $html;

?>

1. The CSS file for the plugin is:

/\* CSS for devices plugin. \*/

.information{

margin-bottom:15px; color:#333333;

}

#plugin\_main\_div{ text-align:left;

}

#plugin\_main\_div h2{ color:#E35C50; font-weight:bold;

}

#plugin\_main\_div h3{ color:#444444; font-weight:bold;

text-transform:capitalize;

}

input[type="button"]{ cursor:pointer;

}

Now our new plugin is finished. Is possible to define a new css and declare much more options. But this is just a simple example. And this is the result of the plugin that we have made. Plugin main window is loaded on the right side. When plugin makes an ajax request data on plugin main window is updated.

# AJAX framework

## Introduction to the AJAX framework

The Manager System AJAX framework is designed to provide a easy way to power Manager System plugins with AJAX content. In order to achieve this, some functions and features has been created. The AJAX framework is not a mandatory add-on. It is just an improvement that can be used, replaced or ignored by plugins.

To use the AJAX framework, you have to think in server and client sides. Client javascript, has to request actions from the plugin on the server side, and then plugin will serve the result of the actions to the client javascript. This document will explain both client and server AJAX API.

## Client Javascript.

Client side has two files that provides a basic layer to manage AJAX request.

### AJAX calls

To make easiest the AJAX calls, a function named ajax\_call is provided within the plugin skeleton. The function submits form data with an action tag so server side plugin can use data and performs actions. The code of the function is:

function ajax\_call(form\_id,action,section,plugin,output\_id)

{

// This script will serialize the form indicated by an id and submit it

// to the desired page.

// Once the response has arrived it display the response inside the id

// defined in output\_id

var json\_field=json\_encode(form\_id);

submit\_data="section="+section+"&plugin="+plugin+"&action="+action+"&type=complex&"+"form\_ fields="+json\_field;

$.ajax({

type: "POST",

url: "index.php", data: submit\_data,

success: function(datos){

// A JSON array is expected

var ret = eval('(' + datos + ')');

$.each(ret.item, function(i,item){ if (item['type']=="script")

{

eval(item['value']);

}

else if (item['type']=="return")

{

$('#'+output\_id).html(item['value']);

}

else if (item['type']=="html")

{

$('#'+item['id']).html(item['value']);

}

else if (item['type']=="value")

{

$('#'+item['id']).val(item['value']);

}

else if (item['type']=="append")

{

}

});

}

}

});

$('#'+item['id']).append(item['value']);

This code is based on jquery that is default javascript framework for Manager System, but can be modified for being use with prototype, mootools or your own ajax call.

File location: plugin\_base\_folder/javascript/ajax.js

### JSON data management

Ajax call uses JSON to serialize data from a form and send it to the server. This is done to reduce the data send to the server and to make more easy the management of data on the server side.

To make more easy the work with JSON objects a function that serializes data has been provided. The funcion will search for data on a form and serialize it.

The code of the function is:

function json\_encode(form\_id)

{

var fields = new Object();

$("#"+form\_id+" :input").each(function(){ if($(this).attr("type")=="checkbox")

{

if(this.checked)

{

fields[$(this).attr("name")]=$(this).val();

}

}

else if($(this).attr("type")=="radio")

{

if(this.checked)

{

fields[$(this).attr("name")]=$(this).val();

}

}

else

{

if($(this).val()!='')

{

}

}

});

fields[$(this).attr("name")]=$(this).val();

return $.toJSON(fields);

}

File location: plugin\_base\_folder/javascript/json\_encode.js

## Server side AJAX framework.

### AJAX return functions

Server side of a plugin receives data from client and after procesing data and performing actions data has to be sent to the client. To archieve this the Manager System framework provides a response constructor that stores data that has to be sent to the client and finally send the data.

Each data stored, is a tuple that contains type, value and id and will be accessible by the client once submitted.

$response\_array=Array();

$response\_iterator=0;

function response\_additem($type,$value,$id="")

{

global $response\_array; global $response\_iterator;

$response\_array['item'][$response\_iterator]['type']=$type;

$response\_array['item'][$response\_iterator]['value']=$value; if(!empty($id))

{

$response\_array['item'][$response\_iterator]['id']=$id;

}

$response\_iterator++;

}

function response\_return()

{

global $response\_array;

echo json\_encode($response\_array);

}

File location: API/complex\_ajax\_return\_functions.js

### Server JSON manipulation

Server will receive the data in JSON format from the client. Function jsondecode decode the JSON submitted by the client json\_encode function and store it on a PHP array.

function jsondecode ($json) {

$json = substr($json, strpos($json,'{')+1, strlen($json));

$json = substr($json, 0, strrpos($json,'}'));

$json =preg\_replace('/(^|,)([\\s\\t]\*)([^:]\*) (([\\s\\t]\*)):(([\\s\\t]\*))/s','$1"$3"$4:', trim($json));

$json=str\_replace('\\"','"',$json);

$json=str\_replace('\\"','"',$json);

return json\_decode('{'.$json.'}', true);

}

File location: API/json\_api.js

# Input validation

### is\_ip($ip)

This function check if the variable $ip contains a valid ip address.

The function will return true if $ip contains a valid ip address and false otherwise. File location: API/common\_validators.php

Example of use:

if(is\_ip($ip))

{

echo "IP address has been validated.";

}

else

{

echo "IP address is not valid.";

}

### is\_url($url)

This function check if $url contains a valid url address.

The function will return true if $url contains a valid url address and false otherwise. File location: API/common\_validators.php

Example of use:

if(is\_url($url))

{

echo "URL has been validated.";

}

else

{

echo "URL is not valid.";

}

### is\_mac($mac)

This function check if $mac contains a valid mac address.

The function will return true if $mac contains a valid mac address and false otherwise. File location: API/common\_validators.php

Example of use:

if(is\_mac($mac))

{

echo "MAC address has been validated.";

}

else

{

echo "MAC address is not valid.";

}

# Server side form validation

To automatically validate forms, the file API/form\_field\_validators.php provide a set of functions that could be easily extended.

### are\_form\_fields\_valid ($post\_data, $fields\_check\_types,$fileds\_ms\_ctes=Array())

This function will perform check actions provided on $fields\_check\_types over the form data received on

$post\_data.

As an option an array with information about the name of the error divs can be provided on the $fields\_ms\_ctes array.

This function will return true if all form fields pass the declared checks, false otherwise. File location:API/form\_fields\_check.php

Example of use:

$fields\_check\_types = Array (

'username' => Array ('ms\_alnum','ms\_mandatory'), 'password' => Array ('ms\_mandatory'), 'cnf\_password' => Array ('ms\_mandatory')

);

foreach (array\_keys($fields\_check\_types) as $id)

{

$fileds\_ms\_ctes[$id] = "error\_field\_$id";

}

return are\_form\_fields\_valid ($post\_data, $fields\_check\_types,$fields\_ms\_ctes);

### is\_filled ($field)

This function check if $field is not empty File location:API/form\_fields\_check.php Example of use:

if(is\_filled($field))

{

echo "Form field not empty.";

}

else

{

echo "Form field empty.";

}

### is\_numerical ($data)

This function check if $data only have numbers. No other character allowed. File location:API/form\_fields\_check.php

Example of use:

if(is\_numerical($data))

{

echo "Form field is composed by numbers.";

}

else

{

echo "Form field has not valid characters.";

}

### is\_text ($data)

This function check if $data only contains letters File location:API/form\_fields\_check.php Example of use:

if(is\_text($data))

{

echo "Form field is composed by letters.";

}

else

{

echo "Form field has not valid characters.";

}

### is\_alnum ($data)

This function check if $data is only contains numbers and letters File location:API/form\_fields\_check.php

Example of use:

if(is\_alnum($data))

{

echo "Form field is alphanumerical.";

}

else

{

echo "Form field is not alphanumerical.";

}

### is\_ip ($ip)

This function check if $ip is a valid ip address File location:API/form\_fields\_check.php Example of use:

if(is\_ip($ip))

{

echo "Form field is an ip address.";

}

else

{

echo "Form field is not an ip address.";

}

### is\_a\_float($data)

This function check if $data is a float in decimal format.

For example, 10.2 or -2.2 will return true, but 10E-2 will return false. File location:API/form\_fields\_check.php

Example of use:

if(is\_a\_float($data))

{

echo "Form field is a float.";

}

else

{

echo "Form field is not a float.";

}

### is\_host ($host)

This function check if $host is a host.

A host can be a server name like [www.libelium.com](http://www.libelium.com/) or an ip address, but not an url like [http://www.libelium.com](http://www.libelium.com/) File location:API/form\_fields\_check.php

Example of use:

if(is\_host($host))

{

echo "Form field is a host.";

}

else

{

echo "Form field is not a host.";

}

### is\_mac ($mac)

This function check if $mac is a MAC address File location:API/form\_fields\_check.php Example of use:

if(is\_mac($mac))

{

echo "Form field is a MAC address.";

}

else

{

echo "Form field is not a MAC address.";

}

### is\_subnet ($address)

This function check if $address is a ip address or a subnet. Subnet should be in the format 192.168.1.0/24.

File location:API/form\_fields\_check.php Example of use:

if(is\_subnet($address))

{

echo "Form field is valid.";

}

else

{

echo "Form field is not valid.";

}

### is\_url ($url)

This function check if $url is an URL address File location:API/form\_fields\_check.php Example of use:

if(is\_url($url))

{

echo "Form field contains an URL.";

}

else

{

echo "Form field do not contain a valid URL.";

}

### is\_email($email)

This function check if $email is an email address File location:API/form\_fields\_check.php Example of use:

if(is\_email($email))

{

echo "Form field is an email address.";

}

else

{

echo "Form field is not an email address.";

}

### is\_hex($data)

This function check if $data is a hexadecimal value File location:API/form\_fields\_check.php

Example of use:

if(is\_hex($data))

{

echo "Form field is hexadecimal.";

}

else

{

echo "Form field is not hexadecimal.";

}

### is\_path($data)

This function check if $data is a path string. Note that only those characters are allowed: Characters from 0 to 9, A to Z and a to z. And the symbols dot, underscore, minus, slash.

File location:API/form\_fields\_check.php Example of use:

if(is\_path($data))

{

echo "Form field is an allowed path.";

}

else

{

echo "Form field is not an allowed path.";

}

# Configuration files framework

## Introduction

The Manager System provides an easy way to write and load configuration files from and to arrays.

### load\_conf\_file ($filepath)

This function will load configuration file provide by $filepath to an array and return it. File location: API/conf\_file.php

Example of use:

$config\_data=load\_config\_file('/etc/myplugin.conf');

### save\_conf\_file ($filepath, $data, $writepath='')

This function saves the array $data to the file provided on $filepath.

This function uses a temporal file, the path for the temporal file can be defined too using $writepath. If no

$writepath is provided the plugin will use $base\_plugin.'data/temp\_config' as default temporal file. File location: API/conf\_file.php

Example of use:

save\_conf\_file('/etc/myplugin.conf',$config\_data);

# System check

### parse\_crypto ()

The function will return an array with the status of the ciphered partitions that can be on . File location: API/parser\_crypto.php

Example of use:

echo print\_r(parse\_crypto(),true);

### parse\_hardware()

The function will return an array with information about hardware modules installed on . This function can not recover information of modules connected over serial interface like zigbee.

File location: API/parser\_hardware.php Example of use:

echo print\_r(parse\_hardware(),true);

### parse\_list\_scan ( $interface )

The function will scan for wireless networks detected by a **WIFI** interface.

The function will return an array with information about networks detected, the array contains the following information for each network:

* id: Cell ID
* mac: MAC of the AP.
* essid: Essid of the network
* mode: Network mode.
* channel: Network channel.
* quality: Quality of the link.
* encr: Encription type of the network. File location: API/parser\_hardware.php Example of use:

$networks=parse\_list\_scan('ath0'); foreach ($networks as $network)

{

echo "Channel ".$network['channel']." for network ".$network['essid'];

}

### proc\_net\_dev()

This function return information about the system interfaces parsed from /proc/net/dev. The function will return an array with information of interfaces.

File location: API/parser\_proc\_net\_dev.php Example of use:

echo print\_r(proc\_net\_dev(),true); sleep(2);

echo print\_r(proc\_net\_dev(),true);

**Interfaces**

**Network interface manipulation**

### is\_active($interface)

This function check if an interface is active. This check is performed in real time when the function is called. The function will return true if the interface is active and false otherwise.

File location: API/is\_active.php Example of use:

if(is\_active('ath0'))

{

echo "Interface ath0 is active";

}

else

{

echo "Interface ath0 is down";

}

### parse\_interfaces ( $filepath )

This function implements a parser for /etc/network/interfaces file. Other systems may have alternates path so the path to interfaces file has to be passed to the function.

The function will return an array with all the interfaces configuration of the $filepath file. File location: API/parser\_interfaces.php

Example of use:

$interfaces=parse\_interfaces('/etc/network/interfaces'); if($interfaces['eth0']['iface']=='dhcp')

{

echo "eth0 interface is using dhcp client to configurate.";

}

else

{

echo "eth0 interface address is ".$interfaces['eth0']['address'];

}

### saveInterfaces ($interface,$post,$read\_path,$write\_path)

This function save a new interface configuration to the interfaces configuration file.

$interface is the name of the interface that will be modified. $post should be an array with the configuration of the interface that will be saved. The variable $read\_path should contain the file with the system interfaces configuration, the function will parse $read\_path file and update this configuration with the values of $post for the

$interface. The new configuration file will be saved on $write\_path file. Note that to $write\_path should be writeable by apache

The function will return an array with all the interfaces configuration that has been written to $write\_path. File location: API/save\_interfaces.php

Example of use:

$post\_data=jsondecode($\_POST['form\_fields']);

saveInterfaces($\_POST['interface'],$post\_data,$base\_plugin."data/interfaces",$base\_plugin. "data/interfaces");

### write\_interfaces ( $filepath,$input,$writepath='')

This low level function saves an parsed interfaces array to a configuration file.

$filepath should contain the path to the file where the interfaces configuration file will be stored.

$input should contain the interfaces array that will be saved. (Note that write\_interfaces don't check for interaces defined on the system. write interfaces just look for interfaces on $input and save a configuration file with the interfaces it found.)

$writepath can be provided with an alternate path for temporal file. If $writepath is not defined write\_interfaces will try to write on the folder data inside the plugin path.

Note that to $filepath and $writepath should be writeable by apache.

The function will return an array with all the interfaces configuration of the $filepath file. File location: API/write\_interfaces.php

Example of use:

$interfaces=parse\_interfaces('/etc/network/interfaces'); write\_interfaces('/tmp/interfaces\_copy',$interfaces);

### make\_wpa\_files($hide,$pass,$mode,$ssid,$interface,$channel,$abg)

This low level function can save needed files to stablish a WPA connection. It can save hostapd.conf or wpa\_supplicant.conf files. hostapd.conf is used by hostapd when is used as a AP with WPA, and wpa\_supplicant.conf is needed when connects to an WPA ciphered network.

$hide sets if the network is hidden, $pass has the string with the WPA key, $mode can be setted to master when hostapd.conf should be saved or as managed when a wpa\_supplicant.conf file has to be saved. $ssid has the ssid of the network, $interface the interface name that is going to be used, $channel has to store the network channel and finally $abg sets if wich protocol 11a, 11b or 11g is going to be used, $abg should contain 1 for 11b, 2 for 11g and 3 for 11a.

The function will return true. And files will be stored on /etc/wpa\_supplicant.conf or /etc/hostapd/hostapd.conf in function of $mode value.

File location: API/write\_wpa\_files.php Example of use:

make\_wpa\_files('0','0123456789','master','MY\_OWN','ath0','2','2');

### add\_mac\_filter($mac,$interface,$type)

This function add a mac to the interface whitelist or blacklist. $mac should contain the MAC address, $interface the interface that will add the MAC and $type must contain value 1 for whitelist or value 2 for blacklist. The function will return true.

File location: API/modify\_mac\_filter.php Example of use:

if(is\_mac($\_POST['mac']))

{

add\_mac\_filter($\_POST['mac'],'ath0','2');

}

### del\_mac\_filter($mac,$interface,$type)

This function delete a mac from the interface whitelist or blacklist. $mac should contain the MAC address,

$interface the interface that will add the MAC and $type must contain value 1 for whitelist or value 2 for blacklist. The function will return true.

File location: API/modify\_mac\_filter.php Example of use:

if(is\_mac($\_POST['mac']))

{

del\_mac\_filter($\_POST['mac'],'ath0','2');

}

### change\_list\_type($type,$interface)

This function changes the list type for an interface. $type should contain value 1 for whitelist or value 2 for blacklist. $inteface must have the interface name.

The function will return true.

File location: API/modify\_mac\_filter.php Example of use:

change\_list\_type('2','ath0');

### modify\_mac\_filter($interface,$data,$action,$type,$readpath='',$writepath='')

This function can be called directly but is designed for being called from add\_mac\_filter, del\_mac\_filter or change\_list\_type. $interface should contain interface name, $data the MAC that is going to be modified or added,

$action declares de action to perform, $action values can be 'change' or 'add'. The function will return true.

File location: API/modify\_mac\_filter.php Example of use:

// This will perform the same action as the example for change\_list\_type; modify\_mac\_filter('ath0','','change','2');

## Bluetooth interfaces manipulation

### hciconfig\_parser()

This function parses in real time hciconfig output and generate an array with the information about the Bluetooth interfaces and it's configuration. Data parsed is address, antenna name and antenna state.

The function will return an array with all the Bluetooth interfaces state. File location: API/parser\_hciconfig.php

Example of use:

$hciconfig=hciconfig\_parser();

echo "hci0 address is: ".$hciconfig[0]['address'];

### parse\_hcid($read\_file='')

This function parses a hcid configuration file. A path to the hcid config file can be setted as an argument with

$read\_file variable. If $read\_file variable is not provided or is empty, parse\_hcid will check for data/hcid.conf file on the plugin folder.

The function will return an array with the configuration of the hcid file. File location: API/parser\_hcid.php

Example of use:

$hcid\_conf=parse\_hcid('/etc/bluetooth/hcid.conf'); if($hcid\_conf['iscan']=='enable')

{

echo "Bluetooth iscan is enabled";

}

### save\_hcid($hci\_config,$save\_file='')

This function saves to a file the content of a configuration array passed in the variable $hci\_config. The path to the file to save can be passed with the $save\_file variable. If $save\_file is empty or not setted the file will be saved on data/new\_hcid.conf on the plugin folder.

Apache should be able to write on $save\_file path or in the data folder within the plugin folder. The does not have any return.

File location: API/save\_hcid.php

Example of use:

$hcid\_configuration=jsondecode($\_POST['form\_fields']); save\_hcid($hcid\_configuration,$base\_plugin.'data/hcid.conf');

### hciconfig\_parser()

This function returns an array with information parsed from the output of the hciconfig command. File location: API/parser\_hciconfig.php

Example of use:

$hciconfig=hciconfig\_parser();

## GPRS interface manipulation

### parse\_wvdial ( $filepath )

This function parses wvdial configuration files. The path to the file to parse should be specified with $filepath variable.

The function will return an array with wvdial configuration.

File location: API/parser\_wvdial.php

### list\_operators($readpath)

Libelium has merged information available on internet with the configuration parameters for different mobile companies in the world. list\_operators loads on an array the available information needed to connect. The information about companies has been grouped by country, and contains operator name, APN, username, password and DNS. Not all operators provide or need all the fields so some of them can be empty. The function will return an array with the configuration of the operators file. The array will contain a list of the countries, the count of countries, and for each country a subarray with the count of operators, the name of the operators, and for each operator the APN, username, password and DNS.

File location: API/list\_operators.php Example of use:

Just a call to the functions with the name of the file is needed.

$data=list\_operators($base\_plugin.'data/operators.txt'); echo '<pre>'.print\_r($data,true).'</pre>';

This code will produce the following output (Note that .. means that some data has been cut for display performance):

Array (

[list] => Australia//Austria//Belgium//Canada//..//Taiwan//Thailand//Turkey//UK//USA [count] => 36

[Australia] => Array (

[count] => 4

[list] => Optus//Telstra//Three//Vodafone [Optus] => Array

(

[apn] => internet [username] => [password] =>

[dns] => 202.139.83.3, 192.65.91.129

)

[Telstra] => Array (

[apn] => telstra.internet [username] =>

[password] =>

[dns] => 139.130.4.4, 203.50.170.2

)

..

)

[Austria] => Array (

[count] => 6

[list] => Connect Austria//Max Online//Max Online Business//Max Online Metro//Mobilkom A1//Tele.Ring

[Connect Austria] => Array (

[apn] => web.one.at [username] => user specific [password] => user specific

[dns] => 194.24.128.100, 194.24.128.102

)

[Max Online] => Array (

[apn] => gprsinternet [username] => GPRS [password] =>

[dns] => 213.162.64.1, 213.162.64.2

)

..

### function save\_gprs($post,$write\_path='')

This function saves a wvdial configuration file with the content of an array. The array fields can be Apache should be able to write on $save\_file path or in the data folder within the plugin folder.

The does not have any return.

File location: API/save\_gprs.php Example of use:

$data=jsondecode($\_POST['form\_fields']); save\_gprs($data);

## Services

**DHCP server manipulation**

### parse\_dhcp\_server()

This function parses dnsmasq configuration file and stores the configuration on an array. The function will return an array with the dnsmasq configuration.

File location: API/parser\_dhcp\_server\_new.php Example of use:

$entries=parse\_dhcp\_server('ath0');

if(isset($entries['ath0'])

{

echo "dhcp over ath0 is setted.";

}

### update\_dhcp($dhcp\_configuration)

This function checks $dhcp\_configuration array (usually provided by a html form) and stores data on an array that can be used with save\_dhcp\_server function.

The function will return an array with the dnsmasq configuration that can be understood by save\_dhcp\_server(). This is the fields that can be defined on $dhcp\_configuration:

* dhcp\_server\_XXXX: Usually used interfaces are eth0, ath0 or ath1 but future supported interfaces can be added for DHCP server. If this field contains string 'on', the function will process the interface DHCP configuration fields.
* dhcp\_start\_XXXX: This field must contain a valid IP Address. This address will be the first value used by dhcp server for new dhcp request.
* dhcp\_end\_XXXX: This field must contain a valid IP Address. This will be the last available address that dhcp server can assign to dhcp requests.
* dhcp\_expire\_XXXX: This field must be a number. This is the number of hours that the dhcp lease will stand for.

File location: API/save\_dhcp\_server\_new.php Example of use:

$dhcp\_configuration=update\_dhcp($form\_fields); save\_dhcp\_server($dhcp\_configuration);

### save\_dhcp\_server($dhcp\_configuration,$save\_file='')

This function saves a dnsmasq configuration file with the values stored on $dhcp\_configuration. A path for the file to write can be passed within $save\_file variable. If no $save\_file variable is passed or is empty

$base\_plugin.'data/dnsmasq.more.conf' will be used as default. The function will return an array with the dnsmasq configuration.

File location: API/save\_dhcp\_server\_new.php Example of use:

$dhcp\_configuration=jsondecode($\_POST['form\_fields']); save\_dhcp\_server($dhcp\_configuration);

## OLSRD manipulation

### parse\_olsrd ( $filepath )

This function parses olsrd configuration file. A path to the configuration file should be passed in the $filepath variable.

The function return an array containing all the configuration parameters parsed. File location: API/parser\_olsrd.php

Example of use:

parse\_olsrd('/etc/olsrd/olsrd.conf');

### parse\_olsrd\_txt\_info ($path)

This function parses olsrd text info provided by the plugin olsrd\_txtinfo.so.0.1 A path to the file generated by the olsrd daemon should be passed in the $filepath variable.

The function return an array containing all the brothers ip parsed, the number of brothers, the topology and the number of the number of topology subarrays.

File location: API/parser\_olsrd\_txt\_info.php Example of use:

<?php

echo print\_r(parse\_olsrd\_txt\_info('olsrd.txt'),true);

?>

This is an example of the output

Array (

[brothers] => Array (

|  |  |  |
| --- | --- | --- |
| [0] | => | 192.168.1.250 |
| [1] | => | 192.168.1.14 |
| [2] | => | 192.168.1.110 |
| [3] | => | 192.168.1.251 |

)

[num\_brothers] => 4

[num\_top] => 2 [top] => Array

(

[0] => Array

(

[1] => 192.168.1.250

[2] => 192.168.1.14

)

[1] => Array

(

[1] => 192.168.1.14

[2] => 192.168.1.250

)

)

)

### save\_olsrd($post,$write\_path='',$read\_path='')

This function will save data stored in $post (usually a html form) on the file declared on $write\_path, aditionally a path to a olsd file can be provided with $read\_path. If $write\_path is empty or not declared

$base\_plugin.'data/new\_olsd.conf' will be used as defalult value, if $read\_path is empty or not declared

/etc/olsrd/olsrd.conf will be used.

The function will call parse\_olsr($read\_path) and data provided by parse\_olsr will be updated with data in $post and saved to $write\_path.

This allows to have a subset of configure options available for users on an html form and a default skeleton. Skeleton will be completed with form values and write a new olrsd configuration file declared on $write\_path. The function has no return value.

File location: API/save\_olsrd.php Example of use:

$post\_data=jsondecode($\_POST['form\_fields']); save\_olsrd($post\_data,$base\_plugin.'data/saved\_data',$base\_plugin.'data/read\_data');

## Fresnel configuration

### set\_fresnel($interface,$distance)

The objetive of fresnel configuration is to set better configuration parameters for a given distance for wifi interfaces.

This function sets interface configuration to fit a given distance. Once setted the values will be saved on

$base\_plugin.'data/fresnel\_'.$interface.'.conf' so configuration can be loaded each time the system restarts.

$interface specifies the interface that will be setted and $distance is the distance in kilometers that will be used for configuration parameters.

The function has no return.

File location: API/set\_fresnel.php Example of use:

set\_fresnel('ath0','1.4');

## Join networks configuration

### save\_join($join,$conf\_file='',$rules\_file='')

provides several interfaces that can be joined to allow that networks on interface A can reach networks on interface B. Save\_join function is used to write a file that will configure to allow those interface joins, and calls write\_rule to save the new rule for manager system usage.

$join should be a vector where it's components are numered from 0 to 2, rules are defined on positions [0][1][2]. The rules should contain on first and last component the interfaces name, and in the second component the rule that can have values:

Right: first interface can reach second interface only. Bidirectional: Both interfaces can reach eachother.

Left: second interface can reach first interface only. The function has no return.

File location: API/save\_join\_new.php Example of use:

$rule=array('eth0','Bidirectional','ath0'); save\_join($rule);

### write\_rule($join,$base,$fp)

Write\_join function is used to write a new rule to a configuration file that will be used by to allow those interface joins.

**$join** should be a vector where it's components are numbered from 0 to 2, rules are defined on positions [0][1][2]. The rules should contain on first and last component the interfaces name, and in the second component the rule that can have values:

* **Right**: first interface can reach second interface only.
* **Bidirectional**: Both interfaces can reach eachother.
* **Left**: second interface can reach first interface only.

**$base** can have a pointer to the first element on the $join vector. This is a feature that can be used by advanced developers to minify the memory comsumption when they need to iterate on several rules on one vector, by default should contain value 0. **$fp** should be the file descriptor of the file that will contain the rules.

The function has no return.

File location: API/save\_join\_new.php Example of use:

$rule=array('eth0','Bidirectional','ath0');

$fp=fopen($base\_plugin.'data/join','w'); write\_rule($join,0,$fp);

### delete\_join\_rule($rule\_number,$conf\_file='',$rules\_file='')

delete\_join\_rule function will delete a rule from the configuration files on .

Function should be called with the possition of the rule to delete. A custom configuration file and custom rules file can be passed to function as optional arguments.

The function has no return.

File location: API/save\_join\_new.php Example of use:

$rule=array('eth0','Bidirectional','ath0'); save\_rule($join);

delete\_join\_rule(0);

## Tools

**GPS manipulation**

### parse\_NMEA($input)

This function parses some of the NMEA strings and stores it's data on a more usable array. The NMEA strings that can be parsed and it's array components that will be output are:

* $GPGGA
  + type: Message ID, GGA protocol header
  + utc: UTC time (hhmmss.sss) o lat: Latitude **(**ddmm.mmmm) o ns: North/South Indicator
  + lat-google: Latitude converted to a value that can be understanded by google maps (positive for north negative for south)
  + long: Longitude (dddmm.mmmm)
  + ew: East/West Indicator
  + long-google: Longitude converted to values that can be understanded by google maps (positive for east, negative for west)
  + gpsqual: Position Fix Indicator
  + numsat: Satellites used. A number between 0 and 12 represents the number of valid satellites in sight.
  + hdp: Horizontal dilution of precision.
  + alt: altitude.
  + un\_alt: units of the altitude value.
  + geoidal: Geoid separation.
  + un\_geoidal: Geoid value units.
  + dpgs: GPGS value.
* $GPVTG
  + type: Message ID, VTG protocol header
  + trkdeg1:Measured heading
  + t: True
  + trkdeg2:Measured heading
  + m: Magnetic1
  + spdknots: Measured horizontal speed
  + knots: Knots
  + spdkmph: Measured horizontal speed
  + kph: Kilometers per hour
* $GPRMC
  + type: Message ID, RMC protocol header
  + utc: UTC time (hhmmss.sss)
  + statusrmc: A=data valid or V=data not valid
  + lat: Latitude **(**ddmm.mmmm)
  + ns: North/South Indicator
  + long: Longitude (dddmm.mmmm)
  + ew: East/West Indicator
  + speed: Speed over ground o track: Course over ground o date: Date (ddmmyy)
  + magvar: Magnetic variation
  + mag\_ew: E=east or W=west
* $GPGSA
  + type: Message ID, GSA protocol header
  + selectmode: Manual or automatic. o mode: Fix not available, 2D or 3D o sat1: Sv on channel 1
  + sat2: Sv on channel 2
  + sat3: Sv on channel 3 o sat4: Sv on channel 4 o sat5: Sv on channel 5 o sat6: Sv on channel 6 o sat7: Sv on channel 7 o sat8: Sv on channel 8 o sat9: Sv on channel 9
  + sat10: Sv on channel 10 o sat11: Sv on channel 11 o sat12: Sv on channel 12
  + pdop: Position dilution of precision.
  + hdop: Horizontal dilution of precision.
  + vdop: Vertical dilution of precision.
* $GPGSV
  + type: Message ID, GSV protocol header
  + satmessages: 1 to 3.
  + messnum: message number. o satview: Satellites in view. o satnum\_1: satellite ID.
  + elevdeg\_1: Elevation.
  + azimuthdeg\_1: Azimuth
  + SNR\_1: Range 0 to 99, or null when not tracking.
  + satnum\_2: satellite ID.
  + elevdeg\_2: Elevation.
  + azimuthdeg\_2: Azimuth
  + SNR\_2: Range 0 to 99, or null when not tracking.
  + satnum\_3: satellite ID.
  + elevdeg\_3:Elevation.
  + azimuthdeg\_3: Azimuth
  + SNR\_3: Range 0 to 99, or null when not tracking.
  + satnum\_4: satellite ID.
  + elevdeg\_4: Elevation.
  + azimuthdeg\_4: Azimuth
  + SNR\_4: Range 0 to 99, or null when not tracking.

File location: API/parser\_NMEA.php Example of use:

$response=parse\_NMEA($line);

if (!empty($response['type'])) // ONLY SHOW RECOGNIZED CHAINS

{

switch ($response['type'])

{

case 'GPGGA':

if ($response['numsat']>=0)

{

echo "google's latitude ".$response['lat-google']; echo "google's longitude ".$response['long-google']; break 2;

}

}

}