Design Guide Multi-Language Support

APRIL 2

Lime Technology, Inc
Authored by: Bergware

UNRAID

Introduction

Why multi-language support?

Unraid is growing in popularity around the world. Many users of Unraid are non-native English speakers and the need arises to offer the user interface of Unraid in a language other than English.

Multi-language support within Unraid consists of two aspects:

1. Internationalization (i18n)
   This is the design in the GUI itself, to support easy localization of content and is the main focus of this document. As a developer of plugins, you need to understand the guidelines and design rules to enable support of different languages.
2. Localization (l10n)
   These are the translations of the GUI content into a language different from English. Help of translators is required here, to translate the original English text into a foreign language. These translations are stored in readable text files

"Multi-Language support"
The next level of user experience for Unraid

Prerequisites

Markdown Styling

The Unraid GUI is constructed using so called “page” files, whenever a specific section of the GUI needs to be displayed, the corresponding page files are dynamically loaded and form together the content on screen.

A “page” file consists of two sections which are separated by a triple dash line (---). The first section has directives to name, to define and to place the page within the GUI. An example of such directives is given below.

```
Menu="Main"
Title="Array Devices"
Tag="database"
---
```

The second section, after the triple dash separator, contains the actual content of the page to be displayed. Content can be a mixture of PHP code, HTML code and JavaScript.

Unraid uses the Markdown styling to build content as desired. Any plugins should follow this styling concept too. It will help in easier translation of the page content. An example of a Markdown style page is given below.

```
<form markdown="1" name="confirm_settings" method="POST" action="/update.php" target="progressFrame">
<input type="hidden" name="#file" value="dynamix/dynamix.cfg"/>
<input type="hidden" name="#section" value="confirm"/>

Confirm reboot & powerdown commands:
: <select name="down" size="1">
  <?=mk_option($confirm['down'], "0", "No")?>
  <?=mk_option($confirm['down'], "1", "Yes")?>
  </select>

> Choose if rebooting or powering down the server needs a confirmation checkbox.

Confirm array stop command:
: <select name="stop" size="1">
  <?=mk_option($confirm['stop'], "0", "No")?>
  <?=mk_option($confirm['stop'], "1", "Yes")?>
  </select>
```

```
> Choose if stopping the array needs a confirmation checkbox.

<?if (isset($confirm['sleep'])):?>
Confirm sleep command:
: <select name="sleep" size="1">
  <?=mk_option($confirm['sleep'], "0", "No")?>
  <?=mk_option($confirm['sleep'], "1", "Yes")?>
  </select>
<?endif;?>

&nbsp;
: <input type="submit" name="#apply" value="Apply" disabled><input type="button" value="Done" onclick="done()">
</form>
```

Preparations

Translation Marking

Text which is translated, needs to be marked in the PHP code. This marking can be done using the PHP function _ [underscore] and the text to be translated given as a string parameter.

For example, the following line of PHP code needs to be adjusted as follows to make it translatable.

Original

```
echo “this is text which needs translation”;
```

Adjusted

```
echo _(“this is text which needs translation”);
```

A similar approach is taken for text in HTML sections, i.e. not generated by PHP. Here text needs to be enclosed with _()_ to have it marked as translatable. Note that no quotes are needed here, surrounding it with the markers suffices.

Original

```
<div id=”test”>this is text which needs translation</div>
```

Adjusted

```
<div id=”test”>_(this is text which needs translation)_</div>
```

Special Characters

A number of ‘special’ characters will be filtered out automatically from the text to translate. This is necessary because they have a specific meaning in PHP and would interfere with the translation process itself. The following characters are automatically removed from a string marked for translation:

```
? { } | & ~ ! [ ] ( ) / : * ^ . " '
```

For example, the text

```
echo _(“this is quick & dirty [with more options].”);
```

Is stored in the text file as (special characters and repeated spaces are removed):

```
this is quick dirty with more options=this is quick & dirty [with more options].
```

Note1: whenever possible try to avoid special characters in the translation text. E.g. when a text has a period sign at the end, then enclose the text to translate without including the period sign.

Note2: special characters present in the translation text file *before* the equal (=) sign, will cause the translation file fail to load and has a misalignment of the GUI as result.

Help Text Marking

In Markdown style, help text are lines starting with a greater-than (>) sign and help text may consist of one or more lines depending on the amount of help explanation which is given.

To translate Help text, it is required to enclose the text section with a unique opening tag and a corresponding closing tag.

Original

```
> this is help text line 1
> this is help text line 2
```

Adjusted

```
:help1001
> this is help text line 1
> this is help text line 2
:end
```

The opening tag “:help” must be on its own line without any preceding characters. It is followed by a unique number, for plugins, it is recommended to start the numbering of help sections from “1001” upwards. This is to avoid numbering conflicts with the help of the GUI itself. Each section, which exists in the source, must be uniquely numbered.

The same unique tag name “:helpX” will be used in the translation files to make a relation possible between the original help text and the translated help text.

The closing tag “:end” again must be on its own line without any preceding characters and signals the end of the help text. Make sure there is always a matching opening and closing tag surrounding the help text.

Text Section Marking

Similar to Help Text marking, it is possible to mark larger sections of text by enclosing them with an opening and closing tag. This way of marking is beneficial to translate larger text sections at once instead of multiple smaller pieces (one-liners).

Original

```
this is text which needs translation, line 1
this is text which needs translation, line 2
```

Adjusted

```
:plug1001
this is text which needs translation, line 1
this is text which needs translation, line 2
:end
```

The opening tag “:plug” must be on its own line without any preceding characters. It is followed by a unique number, for plugins it is recommended to start the numbering of ‘plug’ sections from “1001” upwards. This is to avoid numbering conflicts with the GUI itself. Each ‘plug’ section, which exists in the source, must be uniquely numbered

The same unique tag name “:plugX” will be used in the translation files to make a relation possible between the original text section and the translated text section.

The closing tag “:end” again must be on its own line without any preceding characters and signals the end of the text section. Make sure there is always a matching opening and closing tag surrounding the text section.

Working with Variables

Sometimes it is necessary to translate text which include variables, or in other words; working with dynamically generated text.

Such text can be translated by using the **sprintf()** function of PHP. It might be necessary to rewrite your code to include the sprintf() function, as illustrated by the example below.

Original

```
echo “there are $SURPRISES surprises in the box”;
```

Adjusted

```
echo sprintf(_(“there are %s surprises in the box”), $SUPRISES);
```

The translation text file may look as follows

```
there are %s surprises in the box=er zijn %s verrassingen in de doos
```

Single & Plural Words

The implementation of single and plural words is very basic. Both occurrences need to be added to the translation. Some languages have different plural forms depending on the number of items, but this is not supported here. Choose the most likely form or use parenthesis () to make it any number, e.g. “I have X book(s)”, though this is less pretty.

Original

```
echo sprintf(_("I have %s book".($NUMBER==1 ? '' : 's')), $NUMBER);
```

Translation strings

```
I have %s book=Ik heb %s boek

I have %s books=Ik heb %s boeken
```

Text Arrays

There are three text arrays defined with the names "Months_array", "Days_array" and "Numbers_array". These text arrays have the format:

```
text_array=element1:translation1 element2:translation2 element3:translation3 … etc
```

Only the translation part of the elements needs to be adjusted, this is the part after the colon. Do not modify the structure otherwise it can't be read properly.

Scripts called by *Post* or *Get* actions

When a PHP script is called by a JavaScript post or get operation, it will *not have* any translations information, and hence it needs to include the translations module in order to be able to make translations.

Including and using the translations module should be done at the beginning of the script and involves the following lines of code

```
$docroot = $docroot ?? $_SERVER['DOCUMENT_ROOT'] ?: '/usr/local/emhttp';

$_SERVER['REQUEST_URI'] = '';

require_once "$docroot/webGui/include/Translations.php";
```

By default, the translations module uses the available general translations, which are stored in the "translations.txt" file (see section 'translation text files').

The variable $_SERVER['REQUEST_URI'] is used to include additional translations and specifies the text translation file name without extension, e.g. “main” or “dashboard” or the name of your plugin’s translations, e.g. “myplugin”.
If used, only a single reference name may be given in this variable, otherwise set it to an empty string.

Translation Function “my_lang()”

The translation function “my_lang()” is a purpose build function to support translation of keywords, such as dates, numbers, time and devices in a given text string. There are four different actions available, specified by the second parameter.

Date translation

This is the default action when no second parameter is given. It performs translation of names of months, names of days, and time references such as ‘today’, ‘yesterday’ and ‘x day(s)/week(s)’ ago. For example

```
my_lang(“Tuesday, 31 March 2020, 13:44 (two days ago)”);
```

Number translation

This action translates numbers written as text, e.g. *one*, *two*, *ten*. The numbers 0 to 30 are defined. For example

```
my_lang(“this array has twelve devices”,1);
```

Time translation

This action translates time related words hour(s), minute(s) and second(s). For example

```
my_lang(“duration: 10 hours, 5 minutes, 1 second”,2);
```

Device translation

This action translates internal device names such as parity, disk1, disk10 and capitalize them with a space between name and number. For example

```
my_lang("disk5",3);
```

JavaScript Source

Translations in JavaScript scripts, which are loaded using the "<script>" tag, can be done by enclosing a text string with _(), similar to what is done in PHP source code.

```
swal({title:_("this is text to translate"), text:_("more text"), type:'warning'});
```

A separate translation text file is used to do the translations in JavaScript source.

Translations

Translation Text Files

Once all the text in the source is properly marked for translation, it becomes necessary to create a translation text file which holds the translated text.

Each language needs its own translation text file, it is not possible to save different language translations in the same file.

Translation text files are files stored in UTF-8 format with linux (LF) line endings.
To modify translation text files, a text editor with proper support is required. E.g. notepad++.

The root directory of language support is

```
/usr/local/emhttp/languages
```

Under this root directory are one or more sub-directories created, with each sub-directory holding the translation text files for a particular language.

The sub-directories are named according to the ISO 639-1 language codes in lowercase appended with the ISO 3166-1 country codes in uppercase. For example, Dutch language is stored in the sub-directory: **nl_NL**.

The stock GUI uses eleven translation text files, which you can find in each sub-directory.

```
 - translations.txt -- these are general translations and loaded each time
 - dashboard.txt    -- these are translations for the dashboard section
 - main.txt         -- these are translations for the main section
 - shares.txt       -- these are translations for the shares section
 - users.txt        -- these are translations for the users section
 - settings.txt     -- these are translations for the settings section
 - plugins.txt      -- these are translations for the plugins section
 - docker.txt       -- these are translations for the docker section
 - vms.txt          -- these are translations for the vms section
 - tools.txt        -- these are translations for the tools section
 - javascript.txt   -- these are translations for javascript scripts
```

The content of each translation text file is separated into two parts

Part 1

These are single line entries which are in the format:

```
original english text=translated foreign text
```

Only text after the equal sign (=) needs to be modified (translated). The original English text at the left stays untouched.

Removing a line or omitting a translation after the equal sign, results in the GUI displaying this line with the original English text as given in the source code.

The translated text may have 'special characters', such as slashes, parenthesis or brackets which are not included in the key text, but which are used to display text accordingly. E.g.

```
Bar color - Text=Bar (color) - Text
```

The characters '*' and '**' are used to display text in italics and bold respectively. E.g.

```
Array must be Stopped to change=Array must be **Stopped** to change
```

Translators should try to make their translations with similar length as the original text and avoid space issues in the GUI.

Part 2

These are multi line entries used to translate multiple lines at once.
Multi line translations have a unique opening and closing tag:

:helpXX - unique opening tag used for the built-in help text
:plugXX - unique opening tag used for any multi line text section
:end   - closing tag

It is not allowed to remove or alter these tags, and only do translation of the text between the opening and closing tags!

Help text is written in Markdown style, translations must follow the same style, e.g. lines starting with a '>' must be obeyed.

The built-in help text of Unraid is very extensive, a translator may opt to leave this as original English, i.e. leave the text untouched or alternatively remove it.

Loading of Translation Text Files

When translation text files are loaded, the following files and sequence is used.

1. translations.txt – this text file is always loaded and is the first file
2. “section.txt” – this text file is the corresponding section, e.g. main.txt
3. “plugin.txt” – this text file (if existing) holds additional plugin translations
4. “javascript*.txt” – this is one or more text files with an extension

Because “section.txt” is loaded after translations.txt, it may overwrite earlier defined translations. This approach is used to allow different translations of the same phrase in different places in the GUI.

You can create your own language text file which should be named after the page name used in your plugin. E.g. if your plugin loads a page called “MyPlugin.page”, the corresponding translation text file should be named: “myplugin.txt” (all lowercase).

If there are translations needed for JavaScript source files, then they have to be stored in the text file javascript*.txt, where ‘*’ is a unique name, e.g. your plugin name.

The translation text file(s) which comes with a plugin needs to be manually stored in the corresponding language sub-directory. At the moment of this writing no tooling is available to do this automated.

Because “plugin.txt” is loaded as the very last file, it can both add new translations and overwrite existing translations. Keep in mind that translations which are overwritten are only displayed with the new translation on your plugin page, anywhere else the ‘standard’ translation will be used.

More information about the translation text files is given in the READ-ME-FIRST file, which is included in the zip archive with the English translations.

UNRAID LEGACY SUPPORT

Why Legacy Support

Translations are introduced in Unraid version 6.9. Any plugins also supporting translations will need to run on Unraid 6.9 or higher.

It might be desirable to keep using the updated plugin with translation support on older versions of Unraid, and to not force the user to upgrade to Unraid 6.9, even though this is the recommendation in general.

A number of precautions and adaptations are required for the plugin to support both translations and legacy Unraid versions. These are explained in more detail here after.

Legacy.php script

To make life easier for plugin authors who wish to add legacy support, a specific PHP script is made available, which needs to be included with each plugin that has intended legacy support. The preferred location of this PHP script is the "include" folder, but may be different as long as the reference to the PHP script is correct.

The PHP script checks whether the new translation functions exist, and if not present will create alternative functions to make backward compatibility possible.

Content of the PHP script

```
<?PHP
// Compatibility functions to support Unraid legacy versions without multi-language

if (!function_exists('_')) {
   function _($text) {return $text;}
}
if (!function_exists('parse_lang_file')) {
   function parse_lang_file($file) {return;}
}
if (!function_exists('parse_text')) {
 function parse_text($text) {return
   preg_replace_callback('/_\((.+?)\)_/m',function($m){return
   $m[1];},preg_replace(["/^:((help|plug)\d*)$/m","/^:end$/m","/^\\$(translations =
   ).+;/m"],['','','\\$$1true;'],strpos($text,"---\n")===false?$text:explode("---
   \n",$text,2)[1]));}
}
```

```
if (!function_exists('parse_file')) {
 function parse_file($file,$markdown=true) {return $markdown ?
   Markdown(parse_text(file_get_contents($file))) :
   parse_text(file_get_contents($file));}
}
if (!function_exists('my_lang')) {
   function my_lang($text) {return $text;}
}
if (!$noscript) echo "<script>if (typeof _ != 'function') function _(t) {return
t;}</script>";
?>
```

There are six function definitions

_	This is the actual translation function; the parameter is the text to translate. For legacy systems, it will return the unaltered source text
parse_lang_file	This is the file parsing function; it reads the source file and does the necessary parsing to construct a translation array. For legacy systems it has no specific purpose and just returns without action
parse_text	This is the text parsing file function; it parses text to make it suitable for translation. For legacy systems, it parses the text and indicates that translations are true, mimicking the behavior introduced in Unraid 6.9 and is used to ensure backward compatibility
my_lang	This is a purpose build function; it is used for translating specific context, like dates, numbers, timestamps and device names. For legacy systems, it will return the unaltered source text
parse_file	This is the file parsing function; it is the replacement of the PHP 'require_only' function to do translations on PHP script includes. For legacy systems it parses the original text with or without Markdown
$noscript	This is a variable name; it is *false* by default and indicates whether the JavaScript translation function '_' must be created or not. For legacy systems, the function returns the unaltered source text

Plugin Content Preparations

With the legacy PHP script included, the next step is to prepare the content of the plugin for legacy support. We start first with preparations of *.page files.

Here we assume that all preparations to do translations are already done, according to the guidelines explained earlier in this document.

The original content of the plugin should be embedded by the following code

```
$plugin = name_of_the_plugin;
$translations = file_exists("$docroot/webGui/include/Translations.php");
require_once "$docroot/plugins/$plugin/include/Legacy.php";
?>
<?if (!$translations):?>
<?eval('?>'.parse_file("$docroot/plugins/$plugin/name_of_the_file.page"))?>
<?else:?>
---------
 original content of the plugin page file, including translation tagging
---------
<?endif;?>
```

The code above will check if translations are supported by the system. If so, it will proceed by loading the original content of the plugin page file and the included translation tagging to do the actual translations.

If this is a legacy system without translation support, it will parse the original plugin page file content to make it backwards compatible. This is achieved by reloading the source page file and do parsing on itself before presenting the content to the system.

Assign the name of the plugin to this variable, e.g. "myplugin"

`$plugin = name_of_the_plugin;`

This is the reference to the Legacy.php script, make sure it exists in this location

`$docroot/plugins/$plugin/include/Legacy.php`

This is the reference to the source file, in other words the file which holds the plugin content we want to process (itself).

`$docroot/plugins/$plugin/name_of_the_file.page`

Plugin Scripts called by *Post* or *Get* actions

A different approach for legacy support is required for plugin scripts, which are called by JavaScript post or get actions. The original content of these scripts should be proceeded by the following code

```
$plugin = name_of_the_plugin;
$docroot = $docroot ?: $_SERVER['DOCUMENT_ROOT'] ?: '/usr/local/emhttp';
$translations = file_exists("$docroot/webGui/include/Translations.php");

if ($translations) {
  // add translations
  $_SERVER['REQUEST_URI'] = 'myplugin';
  require_once "$docroot/webGui/include/Translations.php";
} else {
  // legacy support (without javascript)
  $noscript = true;
  require_once "$docroot/plugins/$plugin/include/Legacy.php";
}
---------
 original content of the plugin script, including translation tagging in PHP only
---------
```

The code above will check if translations are supported by the system. If so, it will proceed by loading the stock translation functions of the GUI.

If this is a legacy system without translation support, it will load the legacy support functions instead.

It is important to know that no text parsing is performed here, and consequently translation tagging can only be done using the PHP _ [underscore] function, e.g.

```
_("text to translate");
```

Assign the name of the plugin to this variable, e.g. "myplugin"

```
$plugin = name_of_the_plugin;
```

This is the reference to the Legacy.php script, make sure it exists in this location

```
$docroot/plugins/$plugin/include/Legacy.php
```

Make a reference to the translation text file. This is the name in lowercase without file extension. E.g. "myplugin"

```
$_SERVER['REQUEST_URI'] = 'myplugin';
```