

Phorum Developer Reference Manual

Maurice Makaay, Brian Moon, and Thomas Seifert

January 8, 2008

Contents

1	Templates	5
1.1	Introduction	5
1.2	Template structure	5
1.3	How to start your own template	7
1.4	The Phorum template language	8
1.4.1	Introduction	8
1.4.2	General syntax	8
1.4.3	Data types	9
1.4.3.1	Integers	9
1.4.3.2	Strings	10
1.4.3.3	PHP constants	10
1.4.3.4	Template variables	11
1.4.4	Statements	12
1.4.4.1	Display a variable	12
1.4.4.2	In line comments	13
1.4.4.3	DEFINE	13
1.4.4.4	VAR	13
1.4.4.5	IF .. ELSEIF .. ELSE	14
1.4.4.6	LOOP	15
1.4.4.7	INCLUDE	16
1.4.4.8	HOOK	17
1.4.5	Need the power of PHP?	17
2	Modules	19
2.1	Introduction	19
2.2	Terminology	19
2.2.1	Modules	19
2.2.2	Hacks	20
2.2.3	Hooks	20
2.2.4	Hook functions	20
2.3	Writing your own modules	21
2.3.1	Introduction	21
2.3.2	Module information	21
2.3.3	Module file structure	23

2.3.3.1	Introduction	23
2.3.3.2	Single file modules	23
2.3.3.3	Multiple file modules	25
2.3.4	Supporting multiple languages	27
2.3.5	Module data storage	28
2.3.5.1	Introduction	28
2.3.5.2	Storing data for messages	28
2.3.5.2.1	From hooks that get an editable message array as their argument	29
2.3.5.2.2	From other hooks	29
3	Module hooks	31
3.1	Introduction	31
3.2	Templating	31
3.2.1	css_register	31
3.2.2	javascript_register	33
3.3	Control center	34
3.3.1	cc_panel	34
3.4	Message search	34
3.4.1	search_redirect	34
3.4.2	search_output	35
3.5	File storage	35
3.5.1	file_purge_stale	35
3.6	User data handling	36
3.6.1	user_save	36
3.6.2	user_register	36
3.6.3	user_get	36
3.6.4	user_list	37
3.6.5	user_delete	37
3.7	User authentication and session handling	37
3.7.1	user_authenticate	37

List of Tables

2.1	Keys and values in module information	22
-----	---	----

Introduction

This is the Phorum developer reference manual for Phorum version 5.2.x and up. It is not intended for use with older versions of Phorum, although a lot of information will apply.

Please keep in mind that this manual is neither complete, nor final. If you have any remarks about it, please let us know in the development forum on our website. With your contribution, we hope to make this manual a useful tool for Phorum users in understanding and working with our software.

The Phorum development team

Phorum.org

Chapter 1

Templates

1.1 Introduction

Phorum uses a template system for separating application code from presentation code. Application code contains all the logic that is needed for running Phorum. This is PHP code which is maintained by programmers. Presentation code is used to translate the data that is generated by the application code into a HTML page that can be viewed by the end user. This Presentation code can be maintained by HTML designers.

The big advantages of this type of system are that HTML designers will not be bothered with complicated PHP code and that it is easy to create multiple presentation styles for Phorum.

Although there is no application logic in the templates, it is still possible to put presentation logic in there. Presentation logic is only used for things like making decisions on what to show and how to show it and for processing data that has been generated by the application code. For writing presentation logic, a very simple custom programming language is available (more on that will follow when we talk about the [Section 1.4](#)).

1.2 Template structure

A template set is a collection of files that together form a single template. All template sets are stored in their own subdirectory under the directory `{phorumdir}/templates`. If we assume that we have three templates `default`, `template1` and `template2`, then the directory structure for storing these templates would look like this:

```
{phorum dir}
|
+-- templates
|
+-- default
```

```
|
+-- template1
|
+-- template2
```

Inside these template subdirectories, the files for the templates are stored. There, the the following files can be found:

info.php This is a PHP file that is used for describing some properties of the template. This file can define the following variables:

- `$name`
Mandatory variable. This variable hold the name that you want to give to the template. This is the name that will be displayed in template selection boxes. The name of the directory for the template will only be used by Phorum internally.
- `$version`
Mandatory variable. This variable holds the version number for your template. It's used so you can track what version of the template is installed for Phorum. You can use any type of version numbering you like. If you do not know what to use, then simply give your first version of the template version 1, the second number 2, and so on.
- `$template_hide`
Optional variable. If set to a true value, the template will be hidden from user select boxes where the end user can choose the template that he wants to use.

Example 1.2.1 Template information file: \$info.php

```
<?php
// Prevent loading from outside the Phorum application.
if (!defined("PHORUM")) return;

// Template information.
$name = "A brilliant template";
$version = "1.2-beta";
$template_hide = 1;
?>
```

.tpl and .php files These are the files that hold the actual template code. When the Phorum application wants to display a template, it is always referenced by its basename (i.e. without any file extension like `.php` or `.tpl` after it). If the file `<templatebasename>.php` exists in the template directory, then Phorum will use that file as the template input. Else, `<templatebasename>.tpl` will be used.

An example: if Phorum wants to display the "header" template, it will first search for `header.php` in the template directory. If that file does not exist, it will use `header.tpl` instead.

PHP files (`*.php`) contain pure PHP/HTML code. In Phorum template files (`.tpl`) you can additionally make use of the Section 1.4.

Using this system, template authors can completely revert to using pure PHP-code for templates, without using the template language at all. The Phorum development team does not recommend doing this. To keep templates simple, always try to stick to the combination of HTML code and the template language.

Other files and subdirectories In most cases these will be image files which are stored in a subdirectory `images` of the template. But template authors are free to add whatever subdirectories and files they like to the template directory (e.g. Flash based page components, CSS stylesheets, audio files, JavaScript libraries, etc.).

Combining all this, the full tree for a typical template would look like this:

```
{phorum dir}
|
+-- templates
    |
    +-- templatename
        |
        +-- info.php
        |
        +-- *.tpl
        |
        +-- images
            |
            +-- *.gif, *.jpg, *.png
```

1.3 How to start your own template

Although you can start writing a new template totally from scratch, it is of course much easier to take an existing template and modify that one for your needs. Here are the steps that you have to take for accomplishing this:

- **Copy the default template**

Take the default template directory from `{phorumdir}/templates/default` and copy it over to another directory, for example `{phorumdir}/templates/mytpl`.

- **Edit `info.php` for your template**

Edit `{phorumdir}/templates/mytpl/info.php`. In this file you have

to edit at least the `$name` variable, e.g. to `$name = "My very own template";`

You can hide the template from the user template selection boxes by setting `$template_hide = 1`. If you do this, you can only select this template through the admin interface.

- **Configure Phorum to use your template**

Open Phorum's admin page `{phorumurl}/admin.php` and go to "General Settings". There you will find the option "Default Template". Set that option to your own template. You also have to configure the template in the settings of each single forum where you want the template to appear.

That is it! You are now using your own template. From here on, you can start tweaking the template files in your `{phorumdir}/templates/mytpl` directory.

Phorum uses its own template language to allow for dynamic templates without using PHP. More information on this can be found in the section about the [Section 1.4](#).

1.4 The Phorum template language

1.4.1 Introduction

The largest part of the code that can be found in Phorum template files (`*.tpl`) is plain HTML. To be able to use and display the dynamic data that has been generated by Phorum (like message information, lists of private messages and search results), Phorum uses a custom template language which can be used to mix the HTML code with dynamic data. The template language is a very simple programming language with only a few statements to use. This section will describe the template language in detail.

1.4.2 General syntax

Templates are built using HTML code. Embedded in this HTML code, there can be template language statements. All template statements in the templates are surrounded by "{" and "}" characters. Here's a simple example of what a template could look like:

Example 1.4.1 Template example

```
<html>
  <head>
<title>{HTML_TITLE}</title>
  </head>
  <body>
Your username is: {USER->username}

  {IF USER->username "george"}
    <b>Hello, George!</b>
  {/IF}
  </body>
</html>
```

1.4.3 Data types

The template language supports four data types to use in statements:

- Section [1.4.3.1](#)
- Section [1.4.3.2](#)
- Section [1.4.3.3](#)
- Section [1.4.3.4](#)

1.4.3.1 Integers

Integers are formatted as a sequence of numbers.

Example 1.4.2 Integer values

```
403
90
4231
```

Here is an example of template code in which integers are used:

Example 1.4.3 Code using integer values

```
{VAR INTEGERVAR 1000}
The variable INTEGERVAR is {INTEGERVAR}.

{IF INTEGERVAR 333}
  The INTEGERVAR has the value 333.
{/IF}
```

1.4.3.2 Strings

Strings are sequences of characters within quotes (both double and single quotes can be used).

Example 1.4.4 String values

```
"this is a string value"  
"My 1st string!"  
'Single quoted string is possible too'
```

Now if you need the quote which you used to surround the string with inside the string itself, you must escape it using `\` or `\'`. This is consistent with the way that PHP strings are escaped.

Example 1.4.5 Escaped quotes in string values

```
"this is a \"string\" value"  
'Single quoted \'string\' value'  
"You can use both \" and \' for strings!"
```

Here are some examples of template code in which strings are used:

Example 1.4.6 Code using string values

```
{VAR QUESTION "Do you know what \"fubar\" means?"}  
{VAR CORRECT "That was the right answer!"}  
{VAR INCORRECT "No.. you were wrong!"}  
  
{IF ANSWER 'Fucked Up Beyond All Recognition'}  
  {CORRECT}  
{ELSE}  
  {INCORRECT}  
{/IF}
```

1.4.3.3 PHP constants

It is possible to define constants within PHP. This is done using the `define()` PHP statement. Here's an example:

```
<?php define("MY_CONSTANT", "The constant value") ?>
```

You can reference PHP constants from the template language by using its name, without any quotes. So the constant that was defined in the code above, can be used like this in a template:

Example 1.4.7 Code using a PHP constant definition

```
The value of my PHP constant is {MY_CONSTANT}
```

Apart from defining your own PHP constants, you can also use constants that are already defined by PHP. Two useful constants to use are `true` (value = 1) and `false` (value = 0). Using these, you can write template code like this:

Example 1.4.8 Code using built-in PHP constants

```
{VAR SOME_OPTION true}

{IF SOME_OPTION true}
    The option SOME_OPTION is true.
{/IF}
```

1.4.3.4 Template variables

About the most important data type for the template language is the template variable. Template variables are used by Phorum to store dynamic data, which can be used by your templates. You can also use the variables for storing dynamic data of your own from the templates. Template variables can contain both simple values and complex arrays of data.

You can reference a template variable by using the variable's name, without any quotes. This is the same type of notation as the one that is used for referencing PHP constants (see Section 1.4.3.3). If there are both a constant and a variable with the same name, the value of the constant will take precedence over the template variable.

Example 1.4.9 Template variables

```
NAME
HTML_TITLE
MESSAGES
```

In case the variable represents an array, you can reference the array elements by using the following pointer notation:

Example 1.4.10 Referencing elements in a template variable array

```
ARRAYVARIABLE->SIMPLE_ELEMENT
ARRAYVARIABLE->ARRAY_ELEMENT->SIMPLE_ELEMENT
```

Within a template, variables are used like this:

Example 1.4.11 Code using template variables

```
{VAR MY_VAR "Assign a value to a variable from the template"}

You username is: {USER->username}<br/>
The current forum's name is: {NAME}<br/>

{LOOP MESSAGES}
  Subject: {MESSAGES->subject}<br/>
{/LOOP}
```

What variables are available for what template pages is fully determined by Forum.

1.4.4 Statements

The template language has a number of statements that can be used for executing templating actions and decisions.

- Section [1.4.4.1](#)
- Section [1.4.4.2](#)
- Section [1.4.4.3](#)
- Section [1.4.4.4](#)
- Section [1.4.4.5](#)
- Section [1.4.4.6](#)
- Section [1.4.4.7](#)
- Section [1.4.4.8](#)

1.4.4.1 Display a variable

Function This is both the most simple and the most important template statement there is. Using this statement, you can display the contents of a value.

Syntax {<VALUE>}

Example 1.4.12 Display a variable

```
The name of the current forum is: {NAME}
```

Example code

1.4.4.2 In line comments

Function Sometimes, it's useful to explain what you are doing when writing complicated templating code. In that case you can use comments to document what you are doing. You can also use comments to add general info to the template (like in the example below).

Syntax `{! <COMMENT TEXT>}`

The `<COMMENT TEXT>` can contain any characters you like, except for `"}`.

Example 1.4.13 Add in line comments

```
{! This template was created by John Doe and his lovely wife ↵  
   Jane }
```

Example code

1.4.4.3 DEFINE

Function Using this statement, you can set definitions that can be used by the Phorum software. These are mainly used for doing settings from the template file "settings.tpl" to tweak Phorum's internal behaviour.

Definitions that have been set using this statement are not available from other template statements.

Syntax `{DEFINE <PHORUM DEFINITION> <VALUE>}`

What you can use for `<PHORUM DEFINITION>` is fully determined by the Phorum software (and possibly modules). The `<VALUE>` can be any of the data types that are supported by the template language (see Section 1.4.3).

Example 1.4.14 DEFINE statement usage

```
{DEFINE list_pages_shown 5}
```

Example code

1.4.4.4 VAR

Function Using this statement, you can set variable definitions that can be used by the Phorum template language.

Syntax {VAR <TEMPLATE VARIABLE> <VALUE>}

<TEMPLATE VARIABLE> can be an existing or a new variable name (see Section 1.4.3.4). The <VALUE> can be any of the data types that are supported by the template language (see Section 1.4.3).

Example 1.4.15 VAR statement usage

```
{VAR MY_VAR "This is my first variable!"}
{VAR MY_VAR OTHER_VAR}
{VAR MY_VAR 1234}

{VAR IS_COOL true}
{IF IS_COOL}
    Yes, this is cool
{/IF}
```

Example code

1.4.4.5 IF .. ELSEIF .. ELSE ..

Function Using these statements, you can control if certain blocks of code in your template are processed or not, based on a given <CONDITION>. This can for example be useful if you want certain parts of the page to be only visible for registered users.

Syntax {IF <CONDITION>}
 .. conditional code ..
[{ELSEIF <CONDITION>}
 .. conditional code ..]
[{ELSE}
 .. conditional code ..]
{/IF}

<CONDITION> Syntax: [NOT] <TEMPLATE VARIABLE> [<VALUE>]

The <TEMPLATE VARIABLE> in a <CONDITION> has to be an existing variable name. The <VALUE> can be any of the data types that are supported by the template language (see Section 1.4.3).

If a <VALUE> is used, the <TEMPLATE VARIABLE> will be compared to the <VALUE>. If the <VALUE> is omitted, then the condition will check whether the <TEMPLATE VARIABLE> is set and not empty.

A condition can be negated by prepending the keyword NOT to it.

Multiple conditions can be chained using the keywords AND or OR.

Example 1.4.16 IF .. ELSEIF .. ELSE .. statement usage

```
{IF NOT LOGGEDIN}
    You are currently not logged in.
{ELSEIF USER->username "John"}
    Hey, it's good to see you again, mr. John!
{ELSE}
    Welcome, {USER->username}!
{/IF}

{IF ADMINISTRATOR true OR USER->username "John"}
    You are either an administrator or John.
{/IF}

{IF VARIABLE1 VARIABLE2}
    Variable 1 and 2 have the same value.
{/IF}
```

Example code**1.4.4.6 LOOP**

Function The LOOP statement is used for looping through the elements of array based template variables (for example arrays of forums, messages and users).

Syntax {LOOP <ARRAY VARIABLE>}
 {<ARRAY VARIABLE>}
 {/LOOP <ARRAY VARIABLE>}

The <ARRAY VARIABLE> has to be the name of an existing template variable containing an array.

Within the LOOP, the active array element is assigned to a variable that has the same name as the <ARRAY VARIABLE> that you are looping over. In our example below, we are looping over USERS, which is an array of user data records. Within the loop, USERS is no longer the array of users itself, but the user data record for a single user instead.

Example 1.4.17 LOOP statement usage

```
<ul>
{LOOP USERS}
    <li>{USERS->username}</li>
{/LOOP}
</ul>
```


Example code

1.4.4.7 INCLUDE

Function Include another template in the template.

Syntax {INCLUDE [ONCE] <INCLUDE PAGE>}

The <INCLUDE PAGE> can be any of the data types that are supported by the template language (see Section 1.4.3).

By specifying the keyword ONCE before the name of template to include, you can make sure that that template is only included once per page.

Example 1.4.18 INCLUDE statement usage

```
{INCLUDE "paging"}

{VAR include_page "cool_include_page"}
{INCLUDE include_page}

{INCLUDE ONCE "css"}
```

Example code

Limitation It is not possible to use a dynamic INCLUDE statement (one where the <INCLUDE PAGE> is set through a template variable) within a LOOP statement, in case the included template needs to have access to the active LOOP element. There is no problem if you use a static INCLUDE statement (one where the <INCLUDE PAGE> is set through a string value).

If you really need this kind of functionality though, you can work around this limitation by assigning the active LOOP element to a new template variable, prior to including the dynamic <INCLUDE PAGE>. Example:

```
{! include_page holds the dynamic page to include }
{VAR include_page "some_page"}

{LOOP loop_variable}
  {! Makes loop_variable available as temp_variable in ↵
   the include }
  {VAR temp_variable loop_variable}
  {INCLUDE include_page}
{/LOOP}
```

This way you can access the active LOOP element from the included template through temp_variable. If you would access loop_variable from there, you'd see that it does not contain the active LOOP element, but the full array that you are looping over instead.

1.4.4.8 HOOK

Function The HOOK statement can be used to run a module hook from a template. By using hooks in the templates, you have an easy way for modules to add data to a page, without having to change the templates too much. Because these hooks need an activated module that acts upon them, creating HOOK statements is certainly for advanced users only.

Syntax {HOOK <HOOK NAME> [<ARG1> <ARG2> .. <ARGn>]}

Both the <HOOK NAME> and the arguments that are used in the HOOK statement can be any of the data types that are supported by the template language (see Section 1.4.3).

How hook functions are called Depending on the number of arguments that are used in the HOOK statement, different type of calls are made to the hook function for the given <HOOK NAME>.

- *No arguments:*
the hook function is called without any arguments at all:
`hook_function()`
- *One argument:*
The single argument is used directly for calling the hook function:
`hook_function($ARG1)`
- *Multiple arguments:*
The arguments are wrapped in an array, which is then used for calling the hook function:
`hook_function(array($ARG1, $ARG2, .. $ARGn))`

Example 1.4.19 HOOK statement usage

```
{HOOK "template_hook"}

{LOOP MESSAGES}
  {HOOK "show_message" MESSAGES}
{/LOOP}

{VAR HOOKNAME "my_magic_hook"}
{HOOK HOOKNAME "my argument"}
```

Example code

1.4.5 Need the power of PHP?

Template writers for whom the template language is too limited can break into PHP at any point in the templates, using the regular `<?php ... ?>` syntax. It is not mandatory at all to use the Phorum template language for your templates.

The biggest drawback here, is that knowledge of the Phorum internals is required if you want to work with the data that has been generated by Phorum.

Most template writers will normally only be using HTML and the Phorum template language.

To prevent confusion between PHP code blocks and template statements (which are both surrounded by "{" and "}" characters), always use a whitespace after an opening "{" character in your PHP code. So instead of writing:

```
<?php if ($this = true) {print "It's true";} ?>
```

you now have to write:

```
<?php if ($this = true) { print "It's true"; } ?>
```


This way you can mix PHP code with template code without running into problems.

Chapter 2

Modules

2.1 Introduction

This section describes Phorum's module system. It is targeted at developers who want to do customization and extend the functionality of Phorum. Modules are the preferred way to achieve this.

For much of this document, we will be talking about an example module "foo". Of course you will not name your module "foo", but something much more appropriate. If e not familiar with the terms "foo" and "bar", you can visit [Wikipedia](#) to see why we chose them.

Be sure to read at least the CAUTIONS AND SECURITY ISSUES section, before making your own modules.

2.2 Terminology

2.2.1 Modules

Modules are self contained pieces of software, that can be added to Phorum to change or extend its functionality. Modules can do this without having to change anything in the standard Phorum distribution files or database structure.

The big advantage of modules this is that upgrading the Phorum code is easy (no file changes to redo after upgrading) and that modules can be easily uninstalled when needed.

Installing a module means: drop the code in the Phorum mods directory, go to the admin "Modules" page, enable the module and enjoy! One additional thing that might be needed, is editing one or more template files to display data that is generated by the module.

2.2.2 Hacks

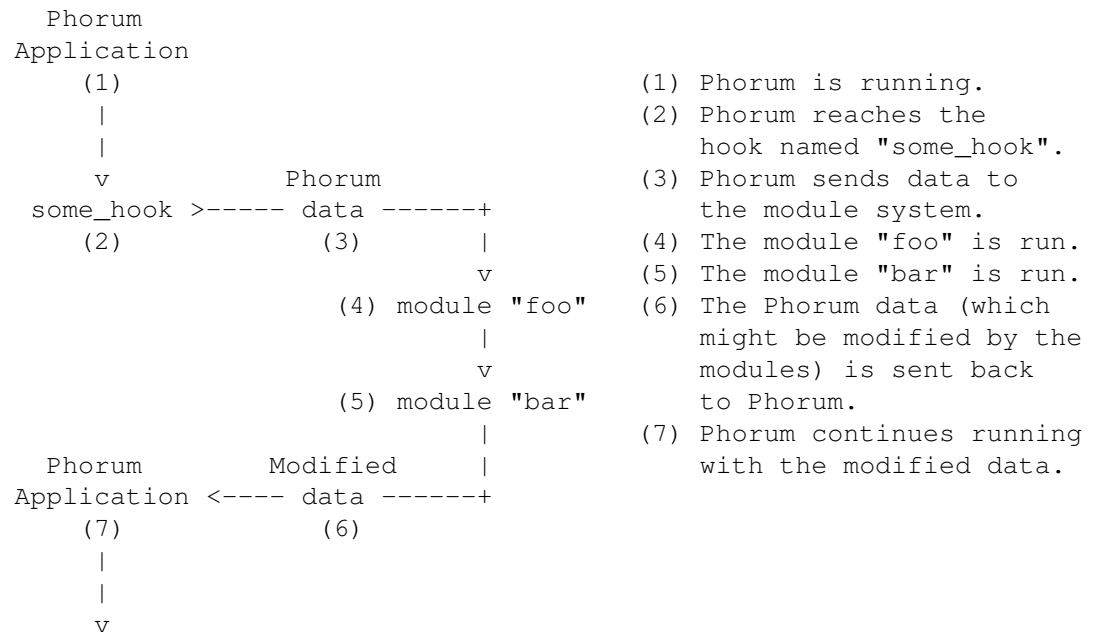
The moment it is necessary to make changes to the standard Phorum distribution files or database structure to implement some kind of functionality, we are talking about a hack (even if the changes that have to be made are accompanied by a drop in module).

Although there is nothing wrong with writing hacks, the Phorum team wants to urge you to try if you can write a module before resorting to a hack. Especially if you are going to publish your changes to the public. Modules are the preferred way of modifying Phorum functionality, because that will make both upgrading your distribution and having your modification adopted by others easier.

2.2.3 Hooks

The Phorum core and Phorum modules are interconnected through hooks. Hooks are points in the application where Phorum stops and runs its data through the modules that are configured to handle the hook. The modules can act upon and change this data.

The following image visualizes what happens when Phorum reaches a hook point in the application, for which two modules ("foo" and "bar") have been configured.



2.2.4 Hook functions

A module contains PHP functions that act as hook functions. Hook functions will receive some data from Phorum through their arguments and have to return the (possibly modified) data, which will then go either back to Phorum or to the input of another module which also handles the same hook (see Section 2.2.3). Based on this, the most

basic (and useless) hook function you could write would look somewhat like this (see XXX for an explanation of the naming scheme that was used for the function):

```
function phorum_mod_foo_some_hook ($data) {  
    return $data;  
}
```

The exact nature of the data that is sent to the hook functions depends solely on the hook that is run. See Chapter 3 for a description of all supported hooks, including a specification of the type of data that is sent.

2.3 Writing your own modules

2.3.1 Introduction

This section will explain to you how to roll your own Phorum modules. We will start out by explaining some of the Section 2.2 that relates to modules. After that, we will explain a very important part modules: the Section 2.3.2. This contains information for both Phorum (what hooks to run in what order, version dependancies) and module users (title, description and other interesting facts). From there on we will walk you through all the possibilities that modules have.

2.3.2 Module information

Module information is the glue between your module and Phorum. It provides information to Phorum about your module. Before we explain how to add this module information to your module, we will first explain what data can be put in there and how that data is formatted.

Module information is formatted using lines of plain text. Each line contains a piece of information about the module. The general format for each of the lines in the module information is:

```
<key>: <value>
```

Empty lines are allowed between these key/value pairs. Below, you can find a list of the keys and values that can be used in the module information.

It is allowed to use multiple hook lines in your module information, so your module can act upon multiple hooks. When doing this, it is also allowed to use the same hook function for handling different hooks in your module (assuming the hooks are compatible).

Here is an example of what the module information for our example module "foo" might look like:

<key>	<value>
title	<p>This is the title for the module that is displayed in the "Modules" page of the admin interface.</p> <p>Example:</p> <pre>title: Foo</pre>
desc	<p>This is the description that is displayed along with the title in the admin interface, to give a little more information about the module. Using HTML in the <value> part is allowed.</p> <p>Example:</p> <pre>desc: This is a very cool module to do stuff.</pre>
hook	<p>This describes which Section 2.2.4 are called for which Phorum hooks. The value consists of two fields, separated by a pipe " " symbol. The first field contains the name of the hook that this module is hooking into. The second field contains the name of the hook function that will be called for the hook.</p> <p>Example:</p> <pre>hook: some_hook phorum_mod_foo_some_hook</pre>
priority	<p>This can be used for changing priorities and dependancies for modules and hooks. Possible values are (in order in which they are processed):</p> <ul style="list-style-type: none"> • run module before after * • run module before after <other module name> • run hook <hook name> before after * • run hook <hook name> before after <other module name> <p>Examples:</p> <p>Run this module before all other modules:</p> <pre>priority: run module before *</pre> <p>Run this module before the bbcode module.</p> <pre>priority: run module before bbcode</pre> <p>Run the "format" hook for this module before the "format" hook of the</p>

Example 2.3.1 Module information

```
title: Foo
desc: This is the Foo module for Phorum. Nothing exciting...
version: 1.0.2
release_date: Jan 1st, 2008
url: http://www.phorum.org
author: John Doe <johndoe@example.com>
require_version: 5.2.2
category: user_features

hook: some_hook|phorum_mod_foo_some_hook
hook: some_other_hook|phorum_mod_foo_some_other_hook
hook: yet_another_hook|phorum_mod_foo_some_other_hook

priority: run some_hook before some_other_module
```

What this module info does, is telling Phorum that when it gets to "some_other_hook", it will have to call the function `phorum_mod_foo_some_other_hook()` in your module. It also tells that for "yet_another_hook" the same function has to be called. It will also take care that the hook "some_hook" is run before the same hook in the module "some_other_module".

2.3.3 Module file structure

2.3.3.1 Introduction

This section describes the file structure of Phorum modules. This structure contains things like the Section 2.3.2, Section 2.2.4 and possibly additional stuff like templates, translations, modules settings, images, scripts, classes, etc.

If your module only needs module information and hook functions to function, it is possible to use the Section 2.3.3.2. If you need more than that, then use the Section 2.3.3.3.

2.3.3.2 Single file modules

Single file modules are useful in case no additional files have to be distributed with your module. Because the module consists of only one single file, it is very easy to distribute. Beware though that the moment that you want to support for example a settings screen, multiple languages or custom images, you will have to switch to the multiple file module structure. Switching does mean some extra work for your users. So only use this format for modules for which you are sure that you do not need additional files in the future.

Single file modules consist of one single PHP file. The name of this file is not restricted. We advice you to use `mod_<modulename>.php` though for clarity and consistency with other module (e.g. `mod_foo.php`). This file contains both the mod-

ule information and the hook function definitions. For storing the module informaton, a special PHP comment is used. This comment must look like the following:

```
/* phorum module info
<module information lines go here>
*/
```

Using the example module info from Example 2.3.1, the complete single file module would look like this (see XXX why we use the check on PHORUM at the start of this file):

Example 2.3.2 Single file module

```
{phorum_dir}/mods/mod_foo.php
<?php

if(!defined("PHORUM")) return;

/* phorum module info
title: Foo
desc: This is the Foo module for Phorum. Nothing exciting...
version: 1.0.2
release_date: Jan 1st, 2008
url: http://www.phorum.org
author: John Doe <johndoe@example.com>
require_version: 5.2.2
category: user_features

hook: some_hook|phorum_mod_foo_some_hook
hook: some_other_hook|phorum_mod_foo_some_other_hook
hook: yet_another_hook|phorum_mod_foo_some_other_hook

priority: run some_hook before some_other_module
*/

function phorum_mod_foo_some_hook ($data) {
    // Do stuff for "some_hook".
    return $data;
}

function phorum_mod_foo_some_other_hook ($data) {
    // Do stuff for "some_other_hook" and "yet_another_hook".
    return $data;
}

?>
```

Installation of a single file module is done by putting the PHP file (e.g. mod_foo.php) directly in the directory {phorumdir}/mods/ and activating the module from

the "Modules" screen in your admin interface.

2.3.3.3 Multiple file modules

These modules are useful in case you need additional files to be stored with your module, for example a settings screen, language files or custom images.

They are stored in their own subdirectory below the directory `{phorumdir}/mods/`. If you have a module named "foo", you will have to create a directory `{phorumdir}/mods/foo/` for storing all module files.

Inside this subdirectory, you will have to create a least two files:

- A file called `info.txt`. This file contains the module information for your module (see Section 2.3.2).
- The PHP file which contains the hook function definitions for your module. The basename of this file should be the same as the name of the module subdirectory. So for our example module "foo", you will have to create a file named `foo.php`.

Using the example module info from Example 2.3.1, the complete multiple file module would look like this (see XXX why we use the check on PHORUM at the start of the PHP file):

Example 2.3.3 Multi file module

```
{phorum dir}/mods/foo/info.txt
title: Foo
desc: This is the Foo module for Phorum. Nothing exciting...
version: 1.0.2
release_date: Jan 1st, 2008
url: http://www.phorum.org
author: John Doe <johndoe@example.com>
require_version: 5.2.2
category: user_features

hook: some_hook|phorum_mod_foo_some_hook
hook: some_other_hook|phorum_mod_foo_some_other_hook
hook: yet_another_hook|phorum_mod_foo_some_other_hook

priority: run some_hook before some_other_module
```

```
{phorum dir}/mods/foo/foo.php
<?php

if(!defined("PHORUM")) return;

function phorum_mod_foo_some_hook ($data) {
    // Do stuff for "some_hook".
    return $data;
}

function phorum_mod_foo_some_other_hook ($data) {
    // Do stuff for "some_other_hook" and "yet_another_hook".
    return $data;
}

?>
```

So far, the module has exactly same functionality as the single file module from Section 2.3.3.2. From here on, the functionality can be extended. Some of the possibilities are:

- Using custom files for your module (images, classes, libs, etc.);
- Letting your module support multiple languages. (See XXX about creation of language files)
- Creating a settings screen for your module; (See XXX about creation of settings screens)
- Adding template files for your module; (See XXX about module template files)

2.3.4 Supporting multiple languages

This feature is supported by the Section [2.3.3.3](#).

If your module includes text that will be displayed to end users, you should strongly consider making it support multiple languages. This will allow Phorum installations that use a different language(s) to display output of your module in the same language(s), instead of the language you have written the module in.

For supporting multiple languages, the first thing to do is add the following line to your module information file `info.txt`:

```
hook: lang|
```

There is no actual hook function configured here, because the "lang" hook is only used as a marker for Phorum. It tells Phorum that your module supports multiple languages.

Next, you must provide at least one language file with your module. Language files are stored in a subdirectory name "lang" inside your module directory. In our sample module, the full directory would be `{phorumdir}/mods/foo/lang/`. The language files must be named identical to the main language files that Phorum uses. To include both English and French, your module would require the following file structure:

```
{phorum dir}/
|
+-- mods/
|
+-- foo/
|
+-- info.txt
|
+-- foo.php
|
+-- lang/
|
+-- english.php
|
+-- french.php
```

The structure of your language files will be almost identical to that of the main Phorum language files. However, for your own language files it is advisable to add an extra level in the language variables, to avoid conflicts with language string from other modules or Phorum itself. Here is an example of how you could do that:

```
<?php
$PHORUM["DATA"]["LANG"]["mod_foo"] = array(
    "Hello"    => "Hello!",
    "Bye"      => "Good bye!"
);
?>
```

Here, the extra inserted level is `["mod_foo"]`. To access the "Hello" string from your module code you would use the PHP variable:

```
$PHORUM["DATA"] ["LANG"] ["mod_foo"] ["Hello"]
```

When you want to use the language string from a template file, the you would use the following Section 1.4.3.4:

```
{LANG->mod_foo->Hello}
```

In case a Phorum installation is using a language that your module does not support, Phorum will automatically attempt to fallback to English. So it is highly recommend that you include an `english.php` language file in all your modules. If both the current language and English are not found, Phorum will be unable to load a language for your module and will display empty space instead of your language strings.

Always try to reuse strings that are already in the main Phorum language files itself. Only create custom strings when there is no alternative available. Having more text to translate is more work for translators and using core language strings helps in keeping the used terminology consistent.

2.3.5 Module data storage

2.3.5.1 Introduction

Sometimes, modules will have to store some data in the Phorum system. For example an avatar module would have to store what avatar a user want to show and a poll module would have to add the question, answers and voting results for a poll to messages in which a poll is added.

This section description what standard methods are available for letting modules store their data in the Phorum system. Of course, as a module writer, you can divert from this and use any kind of storage that you like. You are in no way limited to only use Phorum specific methods here.

2.3.5.2 Storing data for messages

If your module needs to store data along with a Phorum message, the easiest way is to make use of the meta information array that is attached to each message array (`$message["meta"]`). This array is a regular PHP array, which is stored in the database as serialized data (see [PHP's serialize manual](#)). Because Phorum and other modules make use of this meta data as well, you should never squash it, neither access the meta data in the database directly. Instead use the methods described in this section.

To prevent name space collisions with other modules or Phorum, it is good practice to create only one key in the meta data array named `mod_<yourmodule>` (in our example: `mod_foo`). If your module needs to store only one single value, then put it directly under this key:

```
$message["meta"]["mod_foo"] = "the single value";
```

If multiple values need to be stored, then put an array under the key. This array can be as complicated as you like:

```

$message["meta"]["mod_foo"] = array(
    "key1"    => "value1",
    "key2"    => "value2",
    "complex" => array(
        0 => "what",
        1 => "a",
        2 => "cool",
        3 => "module"
    )
);

```

because the meta data is stored as serialized data in the database, it is not possible to handle the data you store in there through SQL queries.

When storing information in the meta data from a hook function, you can encounter two different situations, which both need a different way of handling: hooks that get an editable message array as their argument and hooks that don't.



2.3.5.2.1 From hooks that get an editable message array as their argument

There are some hooks that send a full message structure to the hook functions. These can change the message structure before returning it to Phorum. Examples are the hooks "[?]" and "[?]". For these kind of hooks, you can update the meta information in the message structure and be done with it. Here's an example of what this could look like in your hook function:

```

function phorum_mod_foo_before_post ($message)
{
    // Make sure that we have an array for mod_foo in the meta data.
    if (!isset($message["meta"]["mod_foo"]) ||
        !is_array($message["meta"]["mod_foo"])) {
        $message["meta"]["mod_foo"]["foodata"] = array();
    }

    // Add some fields to the mod_foo data.
    $message["meta"]["mod_foo"]["foodata"] = "Some data";
    $message["meta"]["mod_foo"]["bardata"] = "Some more data";

    // Return the updated message. Phorum will take care of
    // storing the "mod_foo" array in the database.
    return $message;
}

```

2.3.5.2.2 From other hooks For other hooks, the proper way to store information in the meta data is to first retrieve the current message data (including the current meta data) using the `phorum_db_get_message()` function. After this, merge the information for your module with the existing meta data and store the updated data in the database

using the `phorum_db_update_message()` function. Here is an example of what this could look like in your hook function:

```
function phorum_mod_foo_some_hook ($data)
{
    // Somehow you get the id for the message. Here we assume
    // that it is stored in the $data hook parameter.
    $message_id = $data["message_id"];

    // Retrieve the message from the database.
    $message = phorum_db_get_message ($message_id);

    // Extract the current meta data.
    $meta = $message['meta'];

    // Make sure that we have an array for mod_foo in the
    // meta data.
    if (!isset($meta["mod_foo"]) || !is_array($meta["mod_foo"])) {
        $meta["mod_foo"] = array();
    }

    // Add some fields to the mod_foo data.
    $meta["mod_foo"]["foodata"] = "Some data";
    $meta["mod_foo"]["bardata"] = "Some more data";

    // Store the updated meta data in the database.
    phorum_db_update_message($message_id, array("meta" => $meta));

    // Return the data that we got as input for this hook
    // function.
    return $data;
}
```

Changing meta data for a message this way will ensure that the existing meta data is kept intact.

Chapter 3

Module hooks

3.1 Introduction

To satisfy the webmaster that needs every bell and whistle, or those that want to make their web site unique, the Phorum team created a very flexible hook & module system. The hooks allow a webmaster to create modules for doing things like using external authentication, altering message data before it is stored, adding custom information about users or messages, ec. Almost anything you can think of can be implemented through the hook & module system.

This chapter describes all the hooks that are available within the Phorum code. It is mainly targeted at developers that want to write modules.

3.2 Templating

3.2.1 `css_register`

Modules can provide extra CSS data for CSS code that is retrieved through the `css.php` script. Extra CSS definitions can be added to the start and to the end of the base CSS code. Modules that make use of this facility should register the additional CSS code using this hook.

Call time:

At the start of the `css.php` script.

Hook input:

An array, containing the following fields:

- **`css`**
The name of the css file that was requested for the `css.php` script. Phorum requests either "css" or "css_print". The module can use this parameter to decide whether CSS code has to be registered or not.
- **`register`**
An array of registrations, filled by the modules. Modules can register their CSS

code for inclusion in the base CSS file by adding a registration to this array. A registration is an array, containing the following fields:

- **module**
The name of the module that adds the registration.
- **where**
This field determines whether the CSS data is added before or after the base CSS code. The value for this field is either "before" or "after".
- **source**
Specifies the source of the CSS data. This can be one of:
 - * **file(<path to filename>)**
For including a static CSS file. The path should be absolute or relative to the Phorum install directory, e.g. "file(mods/foobar/baz.css)". Because this file is loaded using a PHP include() call, it is possible to include PHP code in this file. Mind that this code is stored interpreted in the cache.
 - * **template(<template name>)**
For including a Phorum template, e.g. "template(foobar::baz)"
 - * **function(<function name>)**
For calling a function to retrieve the CSS code, e.g. "function(mod_foobar_get_css)"
- **cache_key**
To make caching of the generated CSS data possible, the module should provide the css.php script a cache key using this field. This cache key needs to change if the module will provide different CSS data.

Note: in case "file" or "template" is used as the source, you are allowed to omit the cache_key. In that case, the modification time of the involved file(s) will be used as the cache key.

It is okay for the module to provide multiple cache keys for different situations (e.g. if the CSS code depends on a group or so). Keep in mind though that for each different cache key, a separate cache file is generated. If you are generating different CSS code per user or so, then it might be better to add the CSS code differently (e.g. through a custom CSS generating script or by adding the CSS code to the \$PHORUM['DATA']['HEAD_DATA'] variable. Also, do not use this to only add CSS code to certain phorum pages. Since the resulting CSS data is cached, it is no problem if you add the CSS data for your module to the CSS code for every page.

Hook output:

The same array as the one that was used for the hook call arguments, possibly with the "register" field updated. A module can add multiple registrations to the register array.

3.2.2 javascript_register

Modules can provide JavaScript code that has to be added to the Phorum pages. Modules that make use of this facility should register the JavaScript code using this hook.

Call time:

At the start of the javascript.php script.

Hook input:

An array of registrations. Modules can register their JavaScript code for inclusion by adding a registration to this array. A registration is an array, containing the following fields:

- **module**
The name of the module that adds the registration.
- **source**
Specifies the source of the JavaScript data. This can be one of:
 - **file(<path to filename>)**
For including a static JavaScript file. The path should be absolute or relative to the Phorum install directory, e.g. "file(mods/foobar/baz.js)". Because this file is loaded using a PHP include() call, it is possible to include PHP code in this file. Mind that this code is stored interpreted in the cache.
 - **template(<template name>)**
For including a Phorum template, e.g. "template(foobar::baz)"
 - **function(<function name>)**
For calling a function to retrieve the JavaScript code, e.g. "function(mod_foobar_get_js)"
- **cache_key**
To make caching of the generated JavaScript code possible, the module should provide a cache key using this field. This cache key needs to change if the module will provide different JavaScript code.

Note: in case "file" or "template" is used as the source, you are allowed to omit the cache_key. In that case, the modification time of the involved file(s) will be used as the cache key.

It is okay for the module to provide multiple cache keys for different situations (e.g. if the JavaScript code depends on a group). Keep in mind though that for each different cache key, a separate cache file is generated. If you are generating different JavaScript code per user or so, then it might be better to add the JavaScript code differently (e.g. through a custom JavaScript generating script or by adding the code to the \$PHORUM['DATA']['HEAD_DATA'] variable). Also, do not use this to only add JavaScript code to certain phorum pages. Since the resulting JavaScript data is cached, it is no problem if you add the JavaScript code for your module to the code for every page.

Hook output:

The same array as the one that was used as the hook call argument, possibly extended with one or more registrations.

3.3 Control center

3.3.1 cc_panel

This hook can be used to implement an extra control center panel or to override an existing panel if you like.

Call time:

Right before loading a standard panel's include file.

Hook input:

An array containing the following fields:

- **panel:** the name of the panel that has to be loaded. The module will have to check this field to see if it should handle the panel or not.
- **template:** the name of the template that has to be loaded. This field should be filled by the module if it wants to load a specific template.
- **handled:** if a module does handle the panel, then it can set this field to a true value, to prevent Phorum from running the standard panel code.
- **error:** modules can fill this field with an error message to show.
- **okmsg:** modules can fill this field with an ok message to show.

Hook output:

The same array as the one that was used for the hook call argument, possibly with the "template", "handled", "error" and "okmsg" fields updated in it.

3.4 Message search

3.4.1 search_redirect

Phorum does not jump to the search results page directly after posting the search form. Instead, it will first do a redirect to a secondary URL. This system is used, so Phorum can show an intermediate "Please wait while searching" page before doing the redirect. This is useful in case searching is taking a while, in which case users might otherwise repeatedly start hitting the search button when results don't show up immediately.

This hook can be used to modify the parameters that are used for building the redirect URL. This can be useful in case a search page is implemented that uses more fields than the standard search page.

Call time:

Right before the primary search redirect (for showing the "Please wait while searching" intermediate page) is done.

Hook input:

An array of `phorum_get_url()` parameters that will be used for building the redirect URL.

Hook output:

The possibly updated array of parameters.

3.4.2 search_output

This hook can be used to override the standard output for the search page. This can be useful for search modules that implement a different search backend which does not support the same options as Phorum's standard search backend.

Call time:

At the end of the search script, just before it loads the output template.

Hook input:

The name of the template to use for displaying the search page, which is "search" by default.

Hook output:

The possibly updated template name to load or NULL if the module handled the output on its own already.

3.5 File storage

3.5.1 file_purge_stale

This hook can be used to feed the file storage API function `phorum_api_file_purge_stale()` extra stale files. This can be useful for modules that handle their own files, using a custom link type.

Call time:

Right after Phorum created its own list of stale files.

Hook input:

An array containing stale files, indexed by `file_id`. Each item in this array is an array on its own, containing the following fields:

- `file_id`: the file id of the stale file
- `filename`: the name of the stale file
- `filesize`: the size of the file in bytes
- `add_datetime`: the time (epoch) at which the file was added
- `reason`: the reason why it's a stale file

Hook output:

The same array as the one that was used for the hook call argument, possibly extended with extra files that are considered to be stale.

3.6 User data handling

3.6.1 user_save

This hook can be used to handle the data that is going to be stored in the database for a user. Modules can do some last minute change on the data or keep some external system in sync with the Phorum user data. In combination with the [user_get] hook, this hook can also be used to store and retrieve some of the Phorum user fields using some external system.

Call time:

Just before user data is stored in the database.

Hook input:

An array containing user data that will be sent to the database.

Hook output:

The same array as the one that was used for the hook call argument, possibly with some updated fields in it.

3.6.2 user_register

This hook is called when a user registration is completed by setting the status for the user to PHORUM_USER_ACTIVE. This hook will not be called right after filling in the registration form (unless of course, the registration has been setup to require no verification at all in which case the user becomes active right away).

Call time:

Right after a new user registration becomes active.

Hook input:

An array containing user data for the registered user.

Hook output:

The same array as the one that was used for the hook call argument, possibly with some updated fields in it.

3.6.3 user_get

This hook can be used to handle the data that was retrieved from the database for a user. Modules can add and modify the user data. In combination with the [user_save] hook, this hook can also be used to store and retrieve some of the Phorum user fields in some external system

Call time:

Just after user data has been retrieved from the database.

Hook input:

An array of users. Each item in this array is an array containing data for a single user.

Hook output:

The same array as the one that was used for the hook call argument, possibly with some updated fields in it.

3.6.4 user_list

This hook can be used for reformatting the list of users that is returned by the `phorum_api_user_list()` function. Reformatting could mean things like changing the sort order or modifying the fields in the user arrays.

Call time:

Each time the `phorum_api_user_list()` function is called. The core Phorum code calls the function for creating user drop down lists (if those are enabled in the Phorum general settings) for the group moderation interface in the control center and for sending private messages.

Hook input:

An array of user info arrays. Each user info array contains the fields "user_id", "username" and "display_name". The hook function is allowed to update the "username" and "display_name" fields.

Hook output:

The same array as was used for the hook call argument, possibly with some updated fields in it.

3.6.5 user_delete

Modules can use this hook to run some additional user cleanup tasks or or to keep some external system in sync with the Phorum user data.

Call time:

Just before a user is deleted.

Hook input:

The `user_id` of the user that will be deleted.

Hook output:

The same `user_id` as the one that was used for the hook call argument.

3.7 User authentication and session handling

3.7.1 user_authenticate

This hooks gives modules a chance to handle the user authentication (for example to authenticate against an external source like an LDAP server).

Call time:

Just before Phorum runs its own user authentication.

Hook input:

An array containing the following fields:

- type: either `PHORUM_FORUM_SESSION` or `PHORUM_ADMIN_SESSION`;
- username: the username of the user to authenticate;
- password: the password of the user to authenticate;
- user_id: Always NULL on input. This field implements the authentication state.

Hook output:

The same array as the one that was used for the hook call argument, possibly with the `user_id` field updated. This field can be set to one of the following values by a module:

- `NULL`: let Phorum handle the authentication
- `FALSE`: the authentication credentials are rejected
- `1234`: the numerical `user_id` of the authenticated user