Computer Writing and Research Lab

White Paper Series: #071105-1

Drupal Accessibility Modifications at the CWRL

Will Martin
wdmartin@mail.utexas.edu
The University of Texas at Austin

Date: 31 Oct. 2007

Keywords: Accessibility, Drupal

Abstract:
This document gives a technical overview of how the CWRL has modified Drupal (a content management system) to make it more accessible to users with disabilities.

Introduction

Over the summer of 2007, I was hired to work on the accessibility of the web sites maintained by the Computer Writing and Research Lab (CWRL), a research unit within the Department of Rhetoric and Writing (DRW) at the University of Texas in Austin. This article documents the modifications I have made during that time.

Background
The CWRL aims to support and create ways of teaching writing classes in computer-equipped classrooms. Among other things, instructors teaching in CWRL classrooms are required to create web sites for their courses. In past the lab attempted to teach HTML and CSS to all instructors so that they could create and maintain static sites for their classes; but in the long run, this approach proved unsustainable. The fairly rapid turnover of instructors (about 50% per year) meant that each fall the CWRL would need to spend significant amounts of time and effort teaching HTML to users of widely varying technical ability.

To remedy this system, the CWRL eventually began using a content-management system, Drupal, to provide course web sites for instructors. The Drupal installation here is set up in a multi-site configuration, so that all the sites are running from one individual installation of Drupal.

The CWRL is part of a publicly-funded institution, the University of Texas, and therefore falls under the requirements of Section 508 of

the Rehabilitation Act of 1973, as amended in 1998. Section 508 requires that web sites be made accessible to users with various disabilities, especially visual impairments. At the beginning of the summer, the CWRL was not fully in compliance with the Section 508 requirements.

Beyond our legal obligations, we at the CWRL feel a moral obligation to make our web sites accessible to as many people as possible. We therefore determined to meet not only the Section 508 requirements, but also try for compliance with the Web Content Accessibility Guidelines (WCAG). We chose to aim first for WCAG 1.0 compliance, and then move on to WCAG 2.0 compliance.

As of this writing, the CWRL is still using Drupal 4.7 rather than the newer 5.1, due to problems with flexinode in the newer version.

Identifying accessibility problems with Drupal

The scope of the summer project focused solidly on Drupal itself, as distinguished from the content hosted within the Drupal sites. The accessibility of the content is the responsibility of the individual staff members, including teachers, who create the content. As long as the Drupal framework generates accessible pages, any content issues can be remedied on a case-by-case basis.

The university runs accessibility scans using a program called WebXM. This program was used to identify recurring problems in Drupal-generated code. Between WebXM and the experience of the staff, we identified a number of problems:

- The "Event" module (for tracking upcoming events via calendar) routinely generated horribly malformed tables which make no sense whatsoever in screen readers.
- The "Flexinode" module improperly used <label> elements for visual formatting, causing problems in screen readers. (For example, the screen reader FireVox does not read label elements which are not associated with form elements at all.)
- Many modules and some of Drupal's core code generate tables which do not have summary attributes specified. (This problem was fixed, except in administrative pages which are not available to the general public)
- In many cases, the themes used separating characters such as a vertical bracket "|" or right angle-quote "»" between items in lists of links. Screen readers read these characters out loud, which is terribly annoying. These have been replaced with CSS equivalents to satisfy both sighted and visually impaired users.

In addition to correcting accessibility problems, we also added some new accessibility features to the Drupal themes we maintain.

- In addition to having "skip to main content" links, our themes now have "skip to header", "skip to left column", "skip to right column", and "skip to footer". Each of these links is equipped with an access key so that it can be triggered at any time. The header, main content, and footer links are always present; the left and right column links are only present when a corresponding column appears in the web page.
- The link targets embedded in the document now contain identifying text which is read by screen readers, e.g. "Now reading the header," or "Now reading the left column".
- We added access keys to some other standard links which appear on all pages.
- We created a new theme, "Accessibility", which contains additional enhancements. This theme can be activated for any site by any user with an account. Sadly, it is not technically feasible at this time to make this theme available to anonymous users.

Assumptions I've made in writing this document

I've made a number of assumptions in reading this document. First, I assume that you, the reader, have a basic working knowledge of HTML and CSS, which is utterly essential to making any kind of sense out of the following discussion. I have also assumed that you are reasonably familiar with the basics of PHP programming. If you meet those requirements, then you should be able to follow the discussion fairly easily.

I have also made assumptions about the technical environment. I assume:

1. That you're using *nix operating system such as Unix, Solaris, FreeBSD, or any variant of Linux;
2. That you're using Drupal, which itself requires:
    a. A working installation of PHP
    b. A working database server
    c. A working web server
    d. An internet connection and appropriate web browser
3. That you have multiple Drupal themes installed;
4. That you're probably running multiple web sites off of one installation of Drupal;
5. That all of your themes are standardized on an XHTML 1.0 transitional doctype (or higher).

6. If any of these assumptions do not apply, you'll need to adapt anything you find here to your environment, which can be easy or simple depending on all kinds of factors.

Document Conventions
I have adopted the following conventions:

File name
`example_file.php`

Folder name
`themes/example_folder`

HTML code block

```
<element attribute="value">Human readable text</element>
```

PHP variable
`$variable`

PHP function
`example_function()`

PHP code block

```
<?php

     "// This is a comment"

     $variable = function('some data');

?>
```

Types of modifications

There are two types of modifications I have made to Drupal in the course of making it more accessible:

- Theme overrides
- Modifications to modules

I will describe each of these in detail.

Theme Overrides

Almost all of the modifications I have made take the form of theme overrides. In order to understand how these work, you need to know how Drupal's theming sub-system works. Note that in the course of

completing this project we removed or ported all themes which were not written in the PHP Template theming dialect that Drupal uses. The following discussion will therefore be specific to PHP Template. Other theming systems also work with Drupal, but they are beyond the scope of this article.

PHP Template Basics

At the most basic level, a PHP Template theme for Drupal consists of a collection of files stored in the `themes/name_of_the_theme` directory, where "name_of_the_theme" is just that—the name of the theme. For my theme Nefertari, the directory is therefore `themes/Nefertari`. In any given theme directory, you will find some or all of the following files:

`page.tpl.php`
The basic framework of the page as a whole. Technically, this is the only file required for a new theme.

`page-front.tpl.php`
An alternative page template which only applies to the front page for the whole site.

`node.tpl.php`
Defines the markup of nodes. A "node" in this context means "A piece of content." (*Pro Drupal Development*, p. 83) These are the blog posts, book pages and forum topics that make up the site's content.

`block.tpl.php`
Defines the markup of navigation blocks that sit to the left or right of the main content. These can also contain other things besides navigations, such as lists of recent posts, links to blogs, log-in boxes, and so on. And, technically, you can assign blocks to appear in any defined region on the page—so you could, say, stick one in the footer. In practice, it's rare to see a block anywhere besides the left or right column.

`comment.tpl.php`
Defines the markup of comments on blog posts, or responses to forum topics.

`box.tpl.php`
Obscure and rarely used. See details in *Pro Drupal Development*, p. 122.

`style.css`
Defines the appearance of basically everything.

There are likely to be other files in each theme directory also, such as image files used by the theme, extra style sheets, javascript files, and so on. In some themes you may also encounter non-standard template files like `page-admin.tpl.php` (more on that below).

Editing template files is fairly simple. They consist of standard HTML mixed with some very basic PHP. Here's a basic `page.tpl.php`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html lang="en">
  <head>
    <title><?php print $head_title; ?></title>
    <?php print $head; ?>
  </head>
  <body>
    <div id="main_frame">

      <div id="header"><?php print $site_name; ?></div>

      <div id="left_column"><?php print $sidebar_left; ?></div>

      <div id="content">
        <?php print $breadcrumb; ?><br />
        <?php print $content; ?>
      </div>

      <div id="right_column"><?php print $sidebar_right; ?></div>

      <div id="footer"><?php print $footer_message; ?></div>

    </div>
  </body>
</html>
```

The values of the PHP variables like `$breadcrumb` are set by Drupal. The variables I've used in this example are not comprehensive; there are plenty of other available variables. Drupal.org has a more complete list of `page.tpl.php` variables. See also Chapter 8 of *Pro Drupal Development*.

How the Drupal Theme System Sets Variables

Let's look at how Drupal sets the `$links` variable, which prints out several links on one line, for example the "Delete", "Edit", and "Reply" links for a given comment. As you might expect, Drupal has a function which puts together the `$links` variable. Here is the default links function, as defined in `theme.inc`:

```
/**
 * Return a themed set of links.
 *
 * @param $links
 *   An array of links to be themed.
 * @param $delimiter
 *   A string used to separate the links.
 * @return
 *   A string containing the themed links.
 */
function theme_links($links, $delimiter = ' | ') {
  if (!is_array($links)) {
    return '';
  }
  return implode($delimiter, $links);
}
```

This takes an array of links (`$links`), and turns them into a single line of text with a vertical lines in between each link. These vertical bars serve as a divider, helping the user's eye separate the links from one another. That's fine for sighted users. For a visually impaired user, however, the signs don't make much sense when read aloud. Suppose you have this code:

```
<a href="/comment/delete/2886">delete</a>|
<a href="/comment/edit/2886">edit</a>|
<a href="/comment/reply/1124/2886">reply</a>
```

Screen readers will pronounce the dividers, so that it comes out sounding like "link: delete, vertical bracket, link: edit, vertical bracket, link: reply." A visually impaired user doesn't really need to hear "vertical bracket" read out in between each link, particularly not when the screen reader prefaces each one with the phrase "link."

Overriding Default Theme Functions

Fortunately, we can override the `theme_links()` function with one of our own. It is never called directly; instead, it is called by a wrapper function called `theme()`, like this:

```
$output .= theme('links', $links);
```

The first argument for `theme()` specifies what theming function should be used—in this case, the argument "links" tells it to look for the `_links()` function. Any subsequent arguments, like `$links` in this example, get passed on to the function that `theme()` chooses to use.

The `theme()` function tests for the existence of three functions and uses the first one it finds. If it's looking for the *_links()* function, it checks for three possible variants of that function:

1. *theme_name*`_links()`
2. *theming_engine*`_links()`
3. `theme_links()`

First, it gets the name of the current theme, checks whether the theme has a `template.php` file, and looks in that file for a theme-specific `_links()` function. So, for example, if the current theme is "Nefertari", it tries to use the function `Nefertari_links()` from the file `themes/Nefertari/template.php`.

But if there is no such function, it checks to see whether `template.php` contains a function called `phptemplate_links()`. If so, then it'll use that function to theme the links. And lastly, if all else fails, it'll use just plain `theme_links()` from `theme_links()` from `/include/theme.inc`.

The upshot of all this is that we can fix the vertical bars issue fairly easily. We just have to:

1. Copy `theme_links()` from theme.inc;
2. Paste it into template.php and re-name it `phptemplate_links()`;
3. Re-write the function's code to do whatever we want.
4. Here is my `phptemplate_links()` function:

```
function phptemplate_links($links){
  // $links has to be an array.
  if (!is_array($links)) { return ''; }

  // $links has to have at least one link in it.
  if (count($links) == 0){ return ''; }
```

```
  // If there's only one link, return it unaltered.
  if (count($links) == 1){ return $links[0]; }

  $output = '<span class="link_list">';

  for($i = 0; $i < count($links); $i++){
    // For the very first link, chop off the "<a" and
replace it with '<a class="first"'.
    // This is so we can easily apply a different style to
the first link to prevent it
    // from having a vertical divider at the left.
    if($i == 0){
      $output .= '<a class="first"'.substr($links[0], 2);
    } else { $output .= $links[$i]; }
  }

  $output .= '</span>';

  return $output;
}
```

It outputs code like this:

```
<span class="link_list">
<a class="first" href="/comment/delete/2886">delete</a>
<a href="/comment/edit/2886">edit</a>
<a href="/comment/reply/1124/2886">reply</a>
</span>
```

Then we add some CSS to the theme's stylesheet to make it look as though there's a vertical bar separating the links.

```
/* Give the links padding on the left and right, and a
black line background at the left. */
.link_list a {
     padding-right: 0.2em;
     padding-left: 0.4em;
     background: transparent
url("/themes/Link_List_Line.png") top left repeat-y;
}

/* Cancel the left padding and the background on the first
link in the list. */
```

```
.link_list a.first {
     padding-left: 0;
     background: transparent;
}
```

The file `Link_List_Line.png` is just a single black pixel, which gets repeated vertically down the left edge of the link. Thus, a sighted user gets the visual cue separating the links; but screen readers are oblivious to background images, and so they just read off the links. There—problem solved.

The bulk of my modifications to the Drupal output take exactly this form—I've identified the themeable functions, copied them, renamed them to `phptemplate_whatever()`, and modified them to output more accessible code.

Centralization

But there's a problem with this scenario. In the example above, I've put the override function `phptemplate_links()` into one theme's `template.php` file, and modified that theme's style sheet to give format the HTML nicely. That's great as far as it goes—but the CWRL has about 47 themes. The same code will work in all of them, but we have to paste it into all 47 copies of `template.php` and `style.css`, which is a pain because it takes ages to finish and it's completely tedious. Furthermore, we'll have to do it all over again if we ever decide to change the code. Yick.

Fortunately, we can centralize everything for easy maintenance.

Centralizing the Theme Override Functions

Ordinarily, `template.php` holds nothing but functions needed for the theme it's part of; but it's fully capable of containing and executing any PHP statement. The one we're interested in here is the `include()` function. This lets us "include" an external file as if it were part of `template.php`. To that end, I created a file called `/themes/universal-overrides.php`. This contains all the override functions I've written. Then I added an include statement to each `template.php`:

```
include('/usr/local/www/data-dist/themes/universal-
overrides.php');
```

This needs to be the first line in the `template.php` file. Also, you cannot use a relative path in this statement, due to security limitations on the server. It must be the complete path name. When those two conditions are met, the `universal-overrides.php` file will be included when Drupal opens any particular `template.php`, and the override functions will be available to the theme.

This way, if we decide to alter an override function for some reason, we can edit `universal-overrides.php`, upload the new version, and the changes will instantaneously take effect for all 47 themes.

Centralizing the Theme Override CSS

We can accomplish much the same thing for the CSS in our example, though using a somewhat different technique. To do so, I first put the CSS into a file called `/themes/accessibility.css`.

The key to including `accessibility.css` into every file is the Drupal function `_phptemplate_variables()` (*Pro Drupal Development*, p. 126), which lets us modify the variables that Drupal is working with before they are ever sent to the theming engine for final rendering. Here is a simplified version of the `_phptemplate_variables()` function from `universal-overrides.php`:

```
function _phptemplate_variables($hook, $vars = array()){
     switch($hook){

          "// Stuff to do when we're loading a whole page."
          case "page":
               "// Add a link to the accessibility style
sheet."
               $vars["head"] .= '<style type="text/css"
media="all">@import "/themes/accessibility.css";</style>';
          break;
     }

     return $vars;
}
```

Lets look at the two arguments to this function. The first, `$hook`, contains a string identifying where Drupal is in the process of building the page. There are several possible values for `$hook`, but the only one we care about at the moment is "page", which lets us "hook into" the page-building process at the highest level.

The other argument, `$vars`, is an array containing all of the possible variables that will be available to a template file. So, for example, at this stage in the process we have a variable called `$vars["head"]`, which contains all of the HTML for linking in Drupal's default style sheets and javascript files. By the time we reach `page.tpl.php`, that gets turned into just plain `$head`. So all we need to do is append a link to our `accessibility.css` file to

`$vars["head"]`, and our accessibility styles will automatically be included in every theme.

At this point, we have a working system for overriding themable functions centrally, and all that needs to be done is identify themeable functions that need customization for accessibility and do with them as we will.

Creating New Template Variables

But we can take it further than that; we can also define custom variables for use in our templates. Doing so requires two things:

1. A new template file.
2. A callback function.

Let's look at an example template file. Here is the contents of `themes/cwrl_footer.tpl.php`:

```
<p>Computer Writing and Research Lab, The University of Texas at Austin<br />
<span class="accessibility_elements">
<a href="<?php print $base_path; ?>" accesskey="1">Main home page for <?php print $site_name; ?>.</a>
<a href="http://www.cwrl.utexas.edu/accesskeys" accesskey="2">Access key details.</a>
<a href="http://www.cwrl.utexas.edu/" accesskey="3">Computer Writing and Resarch Lab main site.</a>
<a href="http://www.utexas.edu/" accesskey="4">University of Texas main site.</a>
<a href="http://www.utexas.edu/directory/" accesskey="5">University of Texas Student, Faculty, and Staff directory.</a>
</span>
<span class="link_list">
<a class="first" href="http://www.utexas.edu/web/guidelines/accessibility.html" accesskey="6">Accessibility</a>
<a href="http://www.adobe.com/products/acrobat/readstep2.html" accesskey="7">Adobe Acrobat Reader</a>
<a href="http://www.macromedia.com/software/flashplayer/" accesskey="8">Flash Player</a>
<a href="http://www.apple.com/quicktime/download/win.html" accesskey="9">QuickTime Player</a>
<a href="http://www.real.com/" accesskey="0">Real Player</a>
```

```
</span>
</p>
```

This is plain HTML, with the exception of the first link, which uses PHP to print out the path to the main page of the current site. (Note that this code block is slightly simplified—I have omitted the title attributes to prevent the lines from getting too long.) You need to have a copy of `cwrl_footer.tpl.php` in each theme directory. Since the file is identical, however, we can use symbolic links instead. Symbolic links act like files, but actually point to a file in some other place. So we put a copy of `cwrl_footer.tpl.php` in the themes directory, and then create symbolic links in each theme directory. Symbolic links for this file can be created by entering the following commands on the server's command line, assuming the path to the themes directory is `/usr/local/www/data-dist/themes`:

```
cd /usr/local/www/data-dist/themes
ln -s /usr/local/www/data-dist/themes/cwrl_footer.tpl.php
Theme_Directory_Goes_Here/cwrl_footer.tpl.php
```

This creates a symbolic link in the specified theme directory. Note that you cannot use relative paths in these symbolic links; for security reasons, Drupal will refuse to load any symbolic link which uses a relative path like "`../cwrl_footer.tpl.php`". You must specify the complete path. Also, this command only creates one link at a time. In order to create symbolic links in every theme directory at once, use this command:

```
cd /usr/local/www/data-dist/themes
find . -type d -maxdepth 1 -exec ln -s /usr/local/www/data-
dist/themes/cwrl_footer.tpl.php {}/cwrl_footer.tpl.php \;
```

Now for the callback function, defined in `universal-overrides.php`:

```
function phptemplate_cwrl_footer($vars = array()){
      return _phptemplate_callback("cwrl_footer", $vars);
}
```

The `_phptemplate_callback()` function here looks for a file called "`cwrl_footer.tpl.php`" in the directory of the currently selected theme, loads it, and returns the result for assignment to a variable or printing. In our case, it will find the symbolic link in the theme directory, follow the link to the real file, and load it. By editing that one copy of `cwrl_footer.tpl.php`,

we can apply changes instantly to every site regardless of which theme it's using. At our option, we can also pass it an array of variables. The items in the array will be turned into normal variables and made available inside the template file.

Next, we need to add a line to `_phptemplate_variables()`:

```
function _phptemplate_variables($hook, $vars = array()){
     switch($hook){

          // Stuff to do when we're loading a whole page.
          case "page":
               // Add a link to the accessibility style
sheet.
               $vars["head"] .= '<style type="text/css"
media="all">@import "/themes/accessibility.css";</style>';

               // Load the CWRL footer template.
               $vars["cwrl_footer"] = phptemplate_cwrl_footer($vars);
          break;
     }

     return $vars;
}
```

Passing the `$vars` variable to `phptemplate_cwrl_footer()` gives us access to all the default `page.tpl.php` variables in `cwrl_footer.tpl.php`. And adding the results of `phptemplate_cwrl_footer()` to a new item in the `$vars` array means that we'll have our CWRL footer defined in every `page.tpl.php`.

The last step is to add the following line to the footer of each `page.tpl.php`:

```
<?php print $cwrl_footer; ?>
```

The actual `_phptemplate_variables()` function in `universal-overrides.php` is considerably longer than this, but each of the things it does follows the same basic pattern as the cases I've described here.

Theming sub-sites

It is possible to use separate themes for sub-sites within a Drupal site. I am not using this technique extensively, but I have used it once. The theme "Clementine" uses a fixed-width layout that makes it difficult to use the administrative pages properly. The tables in the administrative areas are too wide for the fixed-width column, and so portions of them get obscured under the navigation blocks. To remedy this, I've created a separate template file called `page-admin.tpl.php`. It can be found in `themes/clementine`. The only difference between the `page.tpl.php` and `page-admin.tpl.php` for clementine is that the latter specifies a wider column. In order to make clementine use this template file for the admin areas, we need to modify our `_phptemplate_variables()` function:

```
function _phptemplate_variables($hook, $vars = array()){
     switch($hook){

          // Stuff to do when we're loading a whole page.
          case "page":
               // Get the current theme.
               global $theme_key;

               // Add a link to the accessibility style
sheet.
               $vars["head"] .= '<style type="text/css"
media="all">@import "/themes/accessibility.css";</style>';

               // Load the CWRL footer template.
               $vars["cwrl_footer"] = phptemplate_cwrl_footer($vars);

               switch($theme_key){

                    case "clementine":
                         // Clementine's fixed width is too
narrow for the administration pages.
                         // Make it use page-admin.tpl.php
instead for more width on admin pages.
                         if((arg(0) == "admin")){
$vars['template_file'] = 'page-admin'; }
                    break;

               }
          break;
```

```
    }

    return $vars;
}
```

Switching according to `$theme_key` lets us pick whichever theme is currently being used and undertake theme-specific alterations to the `$vars` array. We test to see if the first argument from the Drupal query string is "admin", and if so, then we specify that `$vars['template_file']` should point at the template file beginning with "page-admin" instead of just plain "page".

Modifications to modules

Very few of my modifications required me to alter module code directly, which his good, because these modifications will have to be re-applied to any later version of the module which comes out. If the modules change significantly, the modifications may need to be rewritten. I've made direct modifications to only three modules:

1. Filter
2. Organic Groups
3. Signup

To illustrate the changes, let's examine the changes I made to Filter.

The unmodified filter.module contains a function, `filter_filter_tips()` that outputs a table of basic HTML tags. The table is assembled as a collection of header information, stored in the variable `$header`, and rows, stored in `$rows`. Then the table is made like this:

```
$output .= theme('table', $header, $rows);
```

The `theme()` function here works exactly as described above. Note that this lacks a summary attribute. In order to add a summary attribute, we need to rewrite the line like this:

```
$output .= theme('table', $header, $rows, array("summary" => t("A
list of basic HTML tags.")));
```

Ordinarily, it would be simple enough to copy the function to `universal-overrides.php`, rename it, and edit away. But unfortunately, this table is created inside a function which is not themeable. In order to be themeable, a function MUST begin with

"theme_", which `filter_filter_tips()` does not. We could theoretically override `theme_table()`, the function that `theme()` calls here. But `theme()` doesn't know what the purpose of the table is. In order to override it at this level, our modified `theme()` function would have to parse the variables it was passed looking for identifying information and then pick the text for a summary attribute based on what it finds there. That approach is awkward, and likely to fail in the case of fairly minor changes to the content of the variables. Therefore, the only sane way to make this modification is by editing `filter.module` directly.

Avoid doing this when possible, because it will need to be re-done every time the function changes (as, for example, it might when we upgrade to a newer version of Drupal).

Conclusion

You should now have a solid working overview of the Drupal theming system and how I went about using it to improve Drupal's accessibility. The exact changes I've made will be summarized in the Appendix. But the best way to proceed is to open universal-overrides.php and study it—I've left copious comments in place for each change.

Appendix

List of modified and new variables in the CWRL's Drupal installation
All of these are available in any `page.tpl.php`:

`$logged_in`
TRUE if the user is logged in, otherwise false

`$admin`
TRUE if the user is an administrator, otherwise false

`$cwrl_footer`
The CWRL footer, loaded from `cwrl_footer.tpl.php`

`$access_links`
The access links, loaded from `access_links.tpl.php`

`$search_box`
The search box, loaded from `search-box.tpl.php`. Note that this is a per-theme template file, not a centralized one, because some themes need a slightly different version than others in order to work properly.

`$header_target`

The target for the header, `#HeaderAccessibility`

`$content_target`

The target for the main content, `#Content`

Note that this is not called `#ContentAccessibility` because there's a good chance that someone may try to guess the URL. Some of the themes had markup tagged as `id="content"`, which causes validation errors when this is used; those themes have been fixed (by replacing their local `#Content` with `#Main` or `#ContentTable` or whatever).

`$footer_target`

The target for the footer, `#FooterAccessibility`

`Sidebar targets`

These are automatically loaded into `$sidebar_left` and `$sidebar_right`.