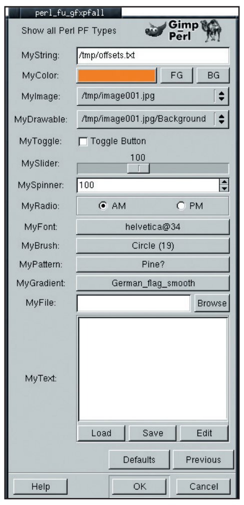
GIMP PROGRAMMING

Writing GIMP Plugins in Perl

PART 2 You can't get far with *GIMP Perl* without knowing how to access *The GIMP*'s internal functions, not to mention letting your user configure your script, says **Michael J Hammel**.

TIP

***GIMP1.2* and *GIMP 2.0* both provide four native programming interfaces: C, Perl, Python and *Script-FU* (a subset of *Scheme*). The design of *The GIMP* allows further language extensions to be added fairly easily. This series on *GIMP Perl* will not be the only language covered in this column. You can expect to see more on both the Perl and C interfaces in the coming months.**


perl_fu_gfxpfall

Show all Perl PF Types

MyString: /tmp/offsets.txt

MyColor: FG BG

MyImage: /tmp/image001.jpg

MyDrawable: /tmp/image001.jpg/Background

MyToggle: Toggle Button

MySlider: 100

MySpinner: 100

MyRadio: (●) AM ( ) PM

MyFont: helvetica@34

MyBrush: Circle (19)

MyPattern: Pine?

MyGradient: German_flag_smooth

MyFile: Browse

MyText:

Load Save Edit

Defaults Previous

Help OK Cancel

Fig1 **A *GIMP Perl* dialog with all possible PF_TYPEs used in the parameter list of the register() function call.**

When you cast your mind back to last month's tutorial, you'll recall that we introduced you to *GIMP Perl,* one of the four programming interfaces available from *The GIMP* for creating plugins. Nearly all *GIMP* plugins provide a user interface where users can set options before processing. *GIMP Perl* provides a simplified *GTK+* interface through the *GIMP::Fu* Perl module. This module offers a variety of input options, all of which are accessible through the register() function which we introduced briefly in the previous article in this series.

Aside from the *GIMP::Fu* module, *GIMP Perl* also provides both procedural and object-oriented interfaces into the *GIMP* function library. This library – known more commonly as the *Procedural Database* or *PDB* – is key to understanding how to making the most of *GIMP Perl.*

In this months tutorial, we'll take a detailed look at both *GIMP::Fu* and the *PDB* while using the second of our two example scripts – GFXLayerSave.pl - to demonstrate their use.

Building a better interface

GIMP::Fu is a wrapper around *The GIMP*'s C API that also provides access to the *GTK+* toolkit – the software which provides buttons, menus and text input fields for *GIMP* and the GNOME desktop. In a way, this is the closing a small circle of life for this software: *GTK+* began life as a tool specifically built for *The GIMP*, became a software entity of its own used by many applications and even GNOME, and now provides script language bindings that can be used to create plugins for its original parent – *The GIMP*.

In last months article, we discussed the use of the 9th argument to the **register()** call in a *GIMP Perl* script. This argument is a parameter array with each element being another array of values. The values define an input option for a user interface feature such as a text entry field, a color selection dialog or a set of radio buttons. The values also provide default settings, valid ranges and various other options.

The first part of the register function is laid out like this:

```

register(
   "function_name",
   "blurb", "help",
   "author", "copyright",
   "date",
   "menu path",
   "image types",
   [
     [PF_TYPE, name, desc, default, extra_args],
     [PF_TYPE, name, desc, default, extra_args],
     ...
   ],
   ....

```

GIMP SCRIPTS

On the *LXF* website at http://www.linuxformat.co.uk/gimp/55.zip, you will find the following files that are mentioned in the course of this tutorial. The two main files were included on last month's discs.

pf_all.pl
GFXLayerSave.pl.commented
GFXLayerSave.pl
GFXOffsets.pl.commented
GFXOffsets.pl

We will include these files on next month's discs as well, for those who do not have access to an Internet connection.

Note that argument nine of this function is enclosed in brackets, signifying an array. This example shows two elements in the parameter array. Each element specifies a parameter type (PF_TYPE in the example) followed by a name and description (both are required). After the description come optional default settings and any additional arguments specific to the parameter type.

The following list shows the set of possible parameter types and what kind of user interface option they provide. **Fig1** below left shows a *GIMP Perl* dialog window with all possible parameter types (except **PF_CUSTOM**) displayed.

```
File: scripts/pf_all.pl
```

shows the code that was used to generate **Fig1**.

As you can see from his simple **pf_all.pl** script, *GIMP:Fu* offers a wide variety user interface options: **PF_INT8, PF_INT16, PF_INT32, PF_INT, PF_FLOAT, PF_STRING, PF_VALUE**

These all provide text input fields, since Perl doesn't differentiate between strings and numeric values. Offering different types that map to a single user interface type allows other plugins to use the value in a language-appropriate way:

PF_COLOR

This provide users with the option of using a color selection dialog to choose a colour.

PF_IMAGE and PF_DRAWABLE

Provide menus of the currently open images and drawables (masks, channels or layer).

PF_TOGGLE, PF_BOOL

These provide a single button that will return either TRUE (if selected) or FALSE (if not selected). The default value for this can be TRUE, FALSE, 1, or 0. The description is also used for the toggle-button label.

PF_SLIDER

Displays a horizontal scale. To set the range and step size append an array in the form

```
[range_min, range_max, step_size, page_increment, page_size]
```

as an extra argument to the parameter array. Default values will be substituted for missing entries, for example:

[PF_SLIDER, "alpha value", "the alpha value", 100, [0, 255, 1]]

which sets the min, max and step size and uses *GTK+* defaults for page increment and page size in the slider.

PF_SPINNER

Provides a spinner widget. Ranges are specified in the same manner as the **PF_SLIDER** parameter type.

PF_RADIO

The extra argument field must refer to an array filled with Option-Name = Option-Value> pairs. *Gimp::Fu* translates this to a series of buttons, laid our horizontally, one for each pair. For example:

[PF_RADIO, "Hour", "AM or PM", 1, [AM => 1, PM => 2]]

draws two buttons: **AM** and **PM**. If **AM** is selected, the callback will receive a value of **1**. If PM is selected, it will receive a value of **2**.

PF_FONT

Provides a font selection option. This feature returns a X Logical Font Descriptor (XLFD) to the callback. The default argument, if specified, must also be a full XLFD specification or a warning will be printed.

PF_BRUSH, PF_PATTERN, PF_GRADIENT

These provide access to brush, pattern and gradient selection dialogs. The value returned can be used in the appropriate tool selection function.

PF_CUSTOM

For scripts requiring a non-standard-widget. See the pod documentation for details on how to use this.

PF_FILE

Provides a text field and button for browsing the file system. Its use is primarily for selecting files but is not limited to this.

PF_TEXT

Like PF_STRING, but presents a multi-line field for text entry along with buttons for saving the text, loading text from a file and editing the text with the user defined default text editor.

The name field for a parameter entry is used as the label displayed next to the widget (widgets are buttons, menus, and essentially any other part of a window). The description field is used in the *PDB*. A special case is the PF_TOGGLE type, which uses the description field as the text for the toggle button.

The order of the parameter elements defines the order they will be passed to your callback function. Remember that your callback function is the one that does the real work and is referenced as the last argument in the call to the **register()** function.

Some parameter element types include option arguments. These option arguments (which are not actually optional if the parameter type requires them) can be single values or arrays. For example, the PF_RADIO type requires an associative array in the option arguments field, with each element of that array describing the label and value associated with a single radio button. The PF_SLIDER parameter type also takes an array of values defining range settings.

The simplicity of *GIMP::Fu* is that the entire user interface is defined as a series of arrays inside a single array, all wrapped inside a single function call. This simplicity comes at a price, however. There is little flexibility in the layout of the user interface – all components are aligned vertically in the plugin dialog. There is little interactivity with the dialog as well. Callbacks for widgets are not configurable, so interactivity is restricted to what *GIMP Perl* has built in – you can't, for example, allow a user to draw shapes in a preview in a *GIMP Perl* plugin, while you can do this with the C API.

Despite such restrictions, *GIMP Perl* still provides adequate functionality for most scripts. Scripts are – after all – primarily quick methods of repeating a series of steps that users find they do often. *Adobe Photoshop* users might consider this similar to the use of Actions, but scripting offers more flexibility than Actions, while not being quite as flexible as compiled plugins.

Who to call and how to call 'em

With the dialog design firmly in mind, we can now look to how to talk to *The GIMP* itself. There is a core set of features provided by *GIMP* for plugins. This core set includes things like retrieving information about layers, accessing tools from the Toolbox and dealing with cut and paste. Beyond this core set, any plugin or script that calls **register()** provides a function that will be added to the *PDB*.

The GIMP's plugin API is the *Procedural Database*, or *PDB*. The *PDB* gives plugins written in any language access to both internal functions of *The GIMP* as well as features provided by other plugins. Any plugin that has registered its callback function can have that function called by any other plugin.

To find a registered function name, you need you start with the *DB Browser*. This is a dialog found via the Xtns>DB Browser menu option in *The GIMP*'s Toolbox. This dialog offers a searchable, scrolled list of functions on the left, and information about the currently selected function on the right.

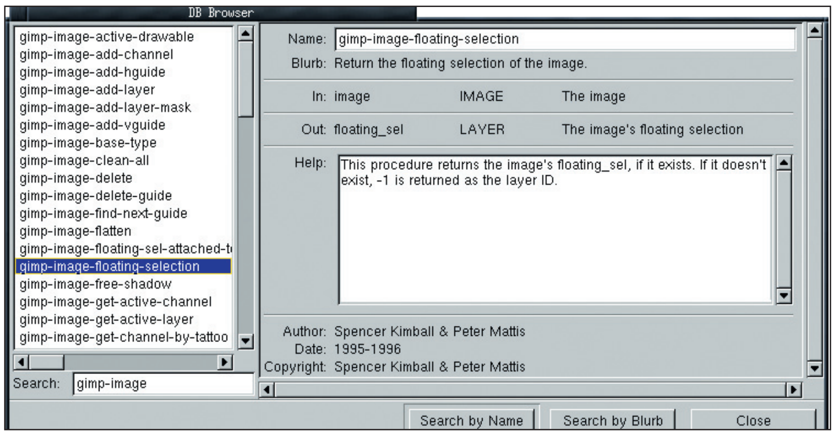
At the bottom left of this dialog is a text input field. Typing a word – or even a few letters – here and hitting **Enter** will limit the set of functions displayed in the scrolled list to any that contain that string of characters. If you want to see the whole list, clear the text field and hit **Enter**.

Clicking on a function in the list will bring up information about that function on the right side of the dialog. Functions can have input arguments – values a plugin will pass to it – and output arguments that are returned to the calling program. They are not required to have either, however. It possible to write a function that runs the entire list of layers of all open images, for

»

TIP

The *GIMP Perl* function name provided as the first argument to the call to register() is the name under which your plugin will be registered in the *GIMP* database. If this name doesn't start with 'perl_fu_', 'file_', 'plug_in_' or 'extension_' then the 'perl_fu_' prefix will be added. If you don't want this, prefix your function name with a single '+'.


DB Browser

```
gimp-image-active-drawable
gimp-image-add-channel
gimp-image-add-hguide
gimp-image-add-layer
gimp-image-add-layer-mask
gimp-image-add-vguide
gimp-image-base-type
gimp-image-clean-all
gimp-image-delete
gimp-image-delete-guide
gimp-image-find-next-guide
gimp-image-flatten
gimp-image-floating-sel-attached-t
gimp-image-floating-selection
gimp-image-free-shadow
gimp-image-get-active-channel
gimp-image-get-active-layer
gimp-image-get-channel-by-tattoo
```

Search: gimp-image

Name: gimp-image-floating-selection
Blurb: Return the floating selection of the image.
In: image IMAGE The image
Out: floating_sel LAYER The image's floating selection
Help: This procedure returns the image's floating_sel, if it exists. If it doesn't exist, -1 is returned as the layer ID.

Author: Spencer Kimball & Peter Mattis
Date: 1995-1996
Copyright: Spencer Kimball & Peter Mattis

Fig2 **The *DB Browser* shows what functions are available to plugins. Note the input and output arguments, and the help text that describes these arguments.**

TIP

There is an additional browser for *PDB* that comes with *GIMP Perl* called the *PDB Explorer*. This dialog is essentially the same as the *DB Browser* but is Perl based. It is little more informative than the the *DB Browser* though it does show function names with underscores instead of dashes.

« example. Such a function would not need to be passed any arguments as the set of images and set of layers for each image are available from *The GIMP*'s core set of functions.

The *DB Browser* lists function names the way they are called using the procedural interface. The only difference is that the *PDB* lists function names with dashes between words in the name while *GIMP Perl* requires you to use underscores instead. The procedural method requires all arguments listed in the *DB Browser* to be passed in the function call while the object-oriented method allows you to leave out arguments that are handled by the object. For example, the procedural call to retrieve the **id** of the current floating selection would be

```
$float_id = gimp_image_floating_selection($image_id);
```

while the object-oriented method would attach this call to an image object, allowing you to leave off the **gimp_image_** prefix:

```
$float_id = $image_id->floating_selection();
```

where **$image_id** is the current image **id** and was passed to your callback function as the first argument (and the current drawable as the second argument) as long as you specified your plugins menu location to be headed by **<Image>**. *GIMP Perl* is smart enough to know that if the method requested doesn't exist for that object it can call the procedural version of that function instead. Note that if you specify your plugin to fall under **<Xtns>** instead, you will not be passed any image or drawable arguments. Remember that the menu location for your plugin is a parameter in your call to **register()**, as we discussed last month.

The real world, part II

With UI and API in hand, we can return to a real-world example with more confidence. Remember last month we said there were two scripts in this project: GFXOffsets.pl and GFXLayerSave.pl. GFXOffsets was the more simple of the two, and we breezed through it fairly quickly. Now, it's time to look at the slightly more complex GFXLayerSave.pl.

```
register (
   "gfxlayersave",
   "Save each layer to a file",
   "Save each layer to a file",
   "Michael J. Hammel <mjhammel\@graphics-muse.org>",
   "Private License - Copyright 2004 Michael J. Hammel",
   "First release - February 2004",
   "<Image>/Filters/GFXMuse/GFXLayerSave",
   "*",
   [
      [ PF_STRING, "path", "Where should the output be saved?",
"/tmp" ]
   ],
   \&LayerSaveGFX_Run
);
```

The register function again places our plugin in the Image section under the Filters/GFXMuse menu. Remember that plugins placed under the Image menus get the active image **id** and active drawable **id** passed as the first argument to the callback routine. Our callback routine is called **LayerSaveGFX_Run** and we've requested a single UI element – a text input field that we will use as the directory to store the layers as separate images.

We've also given our plugin the name **"gfxlayersave"**, which will be translated to **perl-fu-gfxlayersave** in the *PDB*. Other plugins could call our plugin with this name, passing in the image, drawable and path values. If called this way, our single callback, **LayerSave GFX_Run**, would be be run with those arguments *sans* the UI.

The entrance into the callback routine assigns the input values, in the order they are passed, to the variables **$img, $drawable,** and **$path**. We need the image value for this script – our goal is to find each layer in that image and save it as a separate image file.

```
sub LayerSaveGFX_Run {

   # Grab the input parameters.
   my ($img, $drawable, $path) = @_;
```

Next we do some error-checking on the supplied pathname. If the directory does not exist, we use *The GIMP*'s internal message dialog to tell the user that they made a mistake, then let the plugin exit gracefully. The user will be able to see the message even if our plugins dialog has closed.

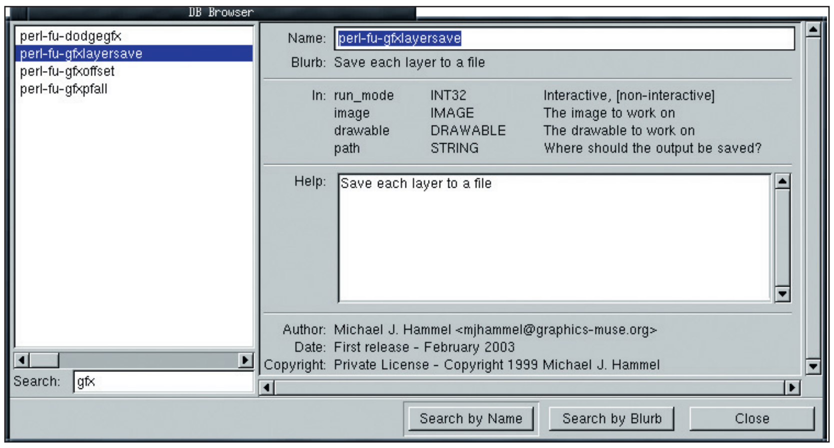
```
   if ( ! -d $path )
   {
      gimp_message("$path does not exist.\nCreate it and try
again.");
      return;
            }
```

Note that we can search for **message** in the *DB Browser* to find out how to use this function. It takes a single text string as its only argument. And because this function is not associated with any specific *GIMP* image, it doesn't have an image or drawable input argument (see the *DB Browser* entry for **gimp_message**).

The image **id** passed in to the callback can now be used as on object identifier. We first use it to retrieve the type of image – RGB, INDEXED, and so forth. This is used to choose an appropriate filename extension later on in the script. The extension is pulled from an array we defined at the top of the script, but which we include in the code shown next below for clarity. The base type is returned as an integer so the array of filename extensions is an ordinary array. If **base_type()** returned a string, we could have used an associative array instead.

```
   @EXTS   = ( "png", "png", "gif" );
   ...
   my $imgtype = $img->base_type();
   $ext = $EXTS[$imgtype];
   my @layers = $img->get_layers();
   my $count = scalar(@layers);
```

After the filename extension is set, we use the image object again to retrieve an array of layers in the image. We'll be running through that array in a moment. We also save a count of the number of layers retrieved.



```
perl-fu-dodgegfx
perl-fu-gfxlayersave
perl-fu-gfxoffset
perl-fu-gfxpfall

Search: gfx

     Name: perl-fu-gfxlayersave
    Blurb: Save each layer to a file

       In: run_mode    INT32        Interactive, [non-interactive]
           image       IMAGE        The image to work on
           drawable    DRAWABLE     The drawable to work on
           path        STRING       Where should the output be saved?

     Help: Save each layer to a file

   Author: Michael J. Hammel <mjhammel@graphics-muse.org>
     Date: First release - February 2003
Copyright: Private License - Copyright 1999 Michael J. Hammel
```

GFXLayerSave.pl registered its function as gfxlayersave, shown here as it is displayed in the *DB Browser*, with its translated name and input arguments.

```
  Gimp->progress_init("GFXLayerSave is working...");
  my $progress_increment = 1 / $count;
  my $progress = 0.0;
```

Before we begin our loop, we let the user know something is about to happen. The call to **Gimp->progress_init()** uses the global object **Gimp**. It is similar to the procedural version of this call except we don't have to specify the **gdisplay** argument (seed the *PDB* entry for **gimp-progress-init**) since we just update the active image window. A couple of progress-related variables are set up, and we're ready to enter our loop.

```
  foreach (@layers)
  {
    $height    = $_->drawable_height();
    $width     = $_->drawable_width();
    $layername = $_->layer_get_name();
    $hasalpha  = $_->drawable_has_alpha();
    $_->edit_copy();
```

The loop iterates over all the layers in the array filled by the call to **$img->get_layers()**. Each layer object is assigned to the special Perl variable **$_**, and we use that to retrieve the height, width, and layer name as well as check to see if the layer has an alpha channel (*ie* transparency). Finally, we make a copy of the layer in *The GIMP*'s primary copy buffer.

Have you noticed that the object calls use the function names from the *PDB* but without the usual **gimp_** or **image_** prefix? This is just a shortcut that the object-based *GIMP Perl* interface provides. If you use the procedural interface instead, you will need to use the function names just as they are in the *PDB*, with dashes changed to underscores. Our next few calls use the procedural interface – we can mix both methods within a single *GIMP Perl* script if we choose.

```
    $newimage = gimp_image_new($width, $height, 0);
    $layer = gimp_layer_new($newimage,$width,$height,0,
 $layername,100,0);
    if ( $hasalpha ) { gimp_layer_add_alpha($layer); }
  gimp_image_add_layer($newimage,$layer,-1);
  gimp_image_set_active_layer($newimage,$layer);
    gimp_edit_clear(gimp_image_active_drawable
 ($newimage));
```

The above shows that we created a new image with the same height and width as our layer and that is of type **RGB** (the **0** value). We added a layer to this image (**gimp_image_new()** doesn't add any layers!) that is fully opaque. If the layer from the original image had transparency, we add it to our just created layer. We then add the layer to the new image, make it the active layer and clear it. Clearing it is necessary, because a layer is a block of memory, and unless you specifically request it, (with **gimp_edit_clear**) that block of memory might have garbage image data in it.

```
    $floatsel = $layer->edit_paste(1);
  gimp_floating_sel_to_layer($floatsel);
    $newimage->merge_visible_layers(0);
```

Remember the copy we made of the original layer a while back? We now paste it using the new layer as the destination object. This will create a floating selection, which we first need to make into a new layer, and then merge with the first layer we created in the new image.

```
    $filename = $layername;
    $filename =~ s/\s/_/g;
    $filename =~ s/\//-/g;
    $filename =~ s/\"//g;
    $filename =~ s/\'//g;
    $filename =~ s/\,/_/g;
    Gimp::Fu::save_image($newimage, "$path/$filename.$ext");
```

With an original image's current layer copied into a layer in a new image, we're now ready to save it to a file. We clean up the filename a bit, which was based on the layername. The layername can have spaces and other cruft that we don't want, so all those **=~** that pervade the above code are just a way of changing the cruft to something more filename-friendly. Then we use *GIMP::Fu* to save the image. This is a convenience function in *GIMP::Fu* that makes it very easy to save images based on their filename extension. Remember: we determined the filename extension earlier – if the extension is **.png**, we save the file and **PNG**, and so forth.

```
    $progress += $progress_increment;
    Gimp->progress_update ($progress);
    $newimage->delete();
    gimp_displays_flush();
  }
  gimp_message("GFXLayerSave completed successfully.");
```

The loop ends with an update to the progress bar in our current image window (image windows are also known as Canvas windows to avoid confusion with the often overused term "image"). After that update we delete the newly created image window which was used as a temporary holder for the copy of our current layer. Then the *GIMP* windows are 'flushed', which causes them to be updated to reflect any changes (there should only be the progress update in this case).

After the loop completes, a final message window is displayed letting the user know the layers have been saved to individual files. All plugins should return **0** (zero) when they are complete so that *GIMP Perl* will know to close the window and let *The GIMP* clean up after itself internally.

This particular script has only the most simplistic user interface. It makes no changes to any Canvas windows either. So from a user perspective, it looks like it actually does very little. This is why the progress updates and calls to **gimp_message()** are so important. Other scripts will produce visible changes, so the use of **gimp_message()** may not be as important.

One important note to remember for plugins that do make changes to images: be sure to include a call to **gimp_undo_push_group_start()** at the start of your callback function and a call to **gimp_undo_push_group_end()** at the end of this function. Doing so will allow the user to use **Ctrl-Z** just once to undo whatever your plugin does, even if multiple changes are made by the plugin. Since no changes were made to any layers in this script, we didn't waste any undo levels with calls to these two functions.

There isn't much to this script and it would be easy – especially after reviewing the myriad of functions available in the *PDB* – to extend this script to do much more clever things. But this is a good start, and an easy way to see how to make use of *GIMP::Fu* and the *PDB* (via the *DB Browser*) to get the most out of scripting in *The GIMP*. LXF

FURTHER INFORMATION

To find more information on *GIMP Perl*, try Marc Lehman's original documentation at www.goof.com/pcg/marc/gimp.html. Information on *GIMP Perl* for *GIMP 2.0* is being laid out by Seth Burgess at www.gimp.org/~sjburges/perl/gimp-perl-faq.html

TIP

A word about modifying your script on the fly: You can update your script while *The GIMP* is running as long as its calling parameters (such as the user interface) don't change. Just make your updates to the callback function and drop the script back in $HOME/.gimp-1.2/plugins (for GIMP 1.2, GIMP 2.0 uses .gimp-2.0 instead). You can then select it from the menus immediately to test your changes. However, if the script has not been registered yet you must restart GIMP before you can use it.