TAILOR-MADE GUIs

Practical PHP Programming

This month, take your PHP to an all-new level by creating GUI applications. **Paul Hudson** shows you how...

Almost ten issues ago, in issue 30 of this very magazine, two momentous events took place: firstly, it was the first part of *Practical PHP Programming*, and secondly, Charlie Stross wrote an excellent tutorial on how to lever the Tk graphical toolkit to create applications using Perl.

Having covered command-line applications last issue, this issue we're going to move onto a fairly similar topic to Charlie's, with the important exception that we shall, of course, be working with PHP. Yep, you guessed it: the topic of this monster eight-page tutorial is how to create graphical applications using PHP. This is quite a jump from all other sorts of PHP programming, so you may find you need to read through this article a few times to get the hang of things!

Getting started

In order to create graphical PHP applications, you first need to install the PHP-GTK module – read the box titled *Installing PHP-GTK* for guidance.

PHP-GTK, as can be guessed from the name, is the combination of our favourite programming language and the GTK+ GUI toolkit. GTK, incidentally, stands for *GIMP Tool Kit*, as it was originally developed for use in the *GIMP* software. Since its creation, GTK+ has come a long way and is now used as a central part of GNOME and has been ported to Windows.

This cross-platform ability works perfectly with PHP's cross-platform nature, and the end result is that, as long as care is taken, you can create attractive and powerful applications that run on a wide variety of machines.

Important warning

Working with PHP-GTK, as already mentioned, is entirely unlike writing PHP in other situations; a solid grasp of object-oriented programming is a must, and also you should be prepared for quite a bit of theory before you get to dig in with the code. I'm not kidding about the OOP requirement!

That being said, I have specifically tried to simplify matters as much as possible, at least to begin with, so that you get to implement cool things using PHP-GTK as quickly as possible.

Graphical User Interfaces

GUIs form the core computing experience for the majority of users – even many Linux people today find themselves using KDE or GNOME for tasks that only a few years ago would have been done from a shell out of necessity. The key reason for this is that GUIs are, generally speaking, designed to be easy to use, with the goal of allowing users to spend more time thinking about what they wish to achieve and less about how it actually needs to be achieved.

In order to minimise the learning curve required for new users to get to grips with an environment, GUIs use shared code to implement the graphical components that make it up; for example, the code to generate a toolbar would be the same across all programs written using a given GUI toolkit in order that all the programs share the same look and feel. Several GUI toolkits, including GTK and Trolltech's Qt (used in KDE) are written using C++ classes and objects, which means each graphical element of a program, known in the *nix world as a *widget*, has its own class which inherits properties and methods from various ancestor classes.

For example, the GtkButton classes inherits from the GtkBin class, which in turn inherits from the GktContainer class, then the GtkWidget class, which inherits from the GtkObject class. Each sub-class adds new methods and properties to do a specific task for that widget, and if a programmer wished to create a specific kind of GtkButton – for example, a button that automatically played a sound when clicked – they would probably find it easiest to inherit from the GtkButton class.

After you've been using GTK for some time, you will likely come to appreciate its fine-grained class inheritance structure, as it allows you to create your own objects at all levels, and also provides you with some very powerful objects, like the GtkCalendar widget, which provides fairly good calendar functionality just by instantiating the class.

Beyond inheriting class variables ('properties') and class functions ('methods'), GTK widgets also inherit *signals*, which is a core topic in GTK. Simply put, signals are emitted when things happen in your GUI; usually this is the user interacting with your widgets. GtkButtons, to take the preceding example, emit a signal when the mouse moves over them, a signal when the mouse clicks them, a signal when the mouse is released from clicking them, and a signal when the mouse moves away from them, amongst other signals.

So each signal, as you can see, is sent out when a particular occurrence happens to a widget, and each widget has its own set of signals that it will emit as a result of user interaction. The magic comes when you tie a given signal, eg 'clicked' for a GtkButton, to a function you have written; this function is then called every time your user clicks the button you chose.

All clear on the theory aspect? Excellent; if so, then it's time to demonstrate the first piece of code...

A Basic GUI

```
<?php
function doshutdown() {
  gtk::main_quit();
}

function btnClick($button) {
  echo "Hello, console!\n";
}

$window =& new GtkWindow();
$window->connect("destroy", "doshutdown");

$button =& new GtkButton("Hello, GTK!");
$button->connect("clicked", "btnClick");

$window->add($button);
$window->show_all();

gtk::main();
?>
```

Our first program, which you should save as **gtk1.php** in your home directory, is simply the "Hello, World!" of PHP-GTK. Yes, I realise that it's twelve lines longer than a simple **echo 'Hello, World!';** would have been; this is because working with graphical interfaces requires a great deal more work: you must create and assign properties to widgets, set up signal handlers, and more. However, don't be put off by the length of the script – I want to go through it line by line in order to show quite how easy it is.

Ignore the two functions near the beginning for the time being; we will come to them shortly. Beyond them lies **$window =& new GtkWindow()**, which is probably something quite new to you, depending on your experience with the language.

If you didn't know already, **=&** means 'assign to equals' rather than 'copy to equals'. When **new GtkWindow()** is called, PHP creates a new object of the class GtkWindow. If we just had **$window = new GtkWindow()** then PHP *creates* the GtkWindow, then *copies* it into the $window variable. So, behind the scenes, this process involves *two* GtkWindows, one that is created using **new** and one that is created using **=**. Using **=&** rather than **=** circumvents this problem; **$window** becomes a *reference* to the same **GtkWindow()** created by the *new* operator, so the process involves only one GtkWindow being created. You're quite able to rewrite the line as **$window = &new GtkWindow()** if you prefer; it's all down to your coding style.

The class **GtkWindow()** itself is a general-purpose application window, and it descends from the class GtkBin. I mention this specifically because descendants of the GtkBin container widget are only allowed to have one child widget inside them. For GtkWindow, this means that you are only allowed one widget (eg one button, one combo box) in your window, which may sound a little restrictive at first, but all will be revealed later on.

GtkBin itself descends from GtkContainer – it is a specific kind of container in that it allows only one child widget – and this provides it with GtkContainer's automagic resizing of child widgets. This means that any widget created inside a GtkBin will take up all the free space inside the GtkBin, and will resize as the GtkBin is resized.

On the next line, we again use **&= new** to create a new widget: this time it's a GtkButton. When you create your button, you can pass in as a parameter the caption you wish to give the button; that is, what text you wish to appear upon it. In the above example, "Hello, GTK!" is passed for the caption of the button, and internally this is used to create a GtkLabel inside the button, to which the caption is assigned – this will become important if you later wish to change the caption of the button to something different.

Again, the **connect()** function is called, this time we connect the 'clicked' signal of our new GtkButton to the PHP function btnClick. This works in the same way as our previous called to **connect()**.

The **add()** method of our GtkWindow is native to all descendants of GtkContainer, and works as you would expect: the widget passed in as the first parameter is added to the container and placed where available. In the situation of GtkWindow, which, remember, is a descendant of GtkBin, this means that the widget being passed (our new GtkButton) will automatically take up all free space in the window.

The next line, **$window->show_all()** translates to "Show this window, and all its child widgets". An alternate method here is **show()**, which would have displayed the GtkWindow but not the GtkButton.

Finally, we come to the call to **gtk::main();**. If this line looks a little odd to you, don't worry: it's a remnant of PHP's mantra of "If we want to do something complicated, at least make it look like C++; that way at least *some* people will understand it". Technically speaking, **gtk::main()** is a call to the static member function **main** of class **gtk**. In normal circumstances, one creates an object of a class before calling a function of that class. However, sometimes it's not necessary or indeed it is counter-productive to use an object on some occasions, and so these functions are 'static': always in the same place.

»

Installing PHP-GTK

The easiest part of it all

Debian users, type **apt-get install php-gtk** as root, and you're done. For everyone else, here's a step-by-step guide:

1) Pop over to www.gtk.org/download/ to download and install GTK+ 1.2. You *may* have this already installed for you by your system, depending on your particular distribution

2) Download the PHP-GTK source from http://gtk.php.net/download.php and extract it locally

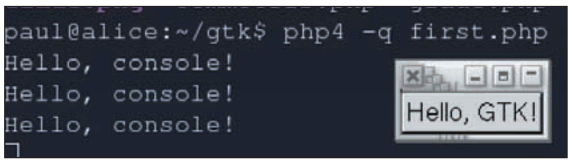
3) Run from the directory you extracted PHP-GTK into, run **./buildconf**, then **./configure**

4) Run **make**, then **make install** to install the extension into your default PHP extension directory

5) Make sure you have the line **extension=php_gtk.so** in your php.ini file

To test your installation, try some of the scripts from the **test** directory. My favourite is **gtk.php**, which shows off a wide selection of widgets to get you started. If you're interested in installing the *Glade* interface builder, then, if you're a Debian user, type **apt-get install glade** as root. Note that the glade-2 package is for GTK+ 2 development, which is as yet unsupported.

For other distros, grab the source code from http://glade.gnome.org/download.html, then run **./configure**, **make**, and **make install** to compile and install the program. You can run *Glade* by simply typing **glade** from the command prompt.



```
paul@alice:~/gtk$ php4 -q first.php
Hello, console!
Hello, console!
Hello, console!
```

All the hard work is over, and we finally have a simple GUI application. It's all easy from here – sort of...

If this all seems complicated, don't worry about it – it really *is* quite complicated, but you needn't understand any more about how it works other than that it's a function call you can use like any other.

What it *does*, though, is quite important, and I'm sorry but it includes *even more* theory! Put simply, owing to the fact that GUIs are signal-based – that is, they wait until told what to do – your control over the program ceases once you've finished creating your GUI and performing any startup tasks. Instead of running everything yourself, control gets passed onto GTK, which enters what's known as its *message loop*, which internally looks something like this:

```
while (1) {
  if user moves over widget {
     if widget has signal handler set {
        send signal to program that mouse is over widget
     } else {
           do nothing
     }
  }

  if user terminates program {
     send signal to program that it is being killed
     break out of while(1) loop
  }

  ...
}
```

Granted, that's quite a big simplification, however you should get the gist: GTK gets control of your GUI and does all the processing for it such as resizing buttons, dropping down combo boxes, highlighting buttons when the mouse is over, etc, and only passes control back to you by sending a signal which in turn calls your signal handler functions. While these functions are running, you are back in control, and you may run all the PHP code you like, including calling other functions. However control will eventually be passed back to GTK, again leaving you waiting for a signal to be passed.

So, calling **gtk::main()** instructs GTK that you're done setting up, and that you're ready for it to take control. Now, onto our two functions: **doshutdown()** and **btnClick()**.

doshutdown() was passed to our first call to **connect()** to tie it to the signal **destroy**. The end result of this is that when the **GtkWindow $window** is being destroyed, that is, closed by the user, it will be sent the event **destroy** by GTK causing it to emit the signal **destroy**, which in turn will call the **doshutdown()** function. **doshutdown()** has just one line inside it: a call to another static member function, **gtk::main_quit()**.

gtk::main_quit() only has any use when **gtk::main()** has been called, because it instructs GTK that you're ready to exit its message loop and resume control of the application. Generally speaking, this is the end of your application, as control is passed back to you after **gtk::main_quit()** has been called, PHP continues to execute any code that lies after your original **gtk::main()** call.

Our **btnClick()** function was tied to the 'clicked' signal of our GtkButton, and so it will be called every time a user clicks **Hello, GTK!**. The 'clicked' signal is often sent part-way between two other signals, 'pressed' and 'released', which correspond to a mouse button being pressed on a button and a mouse button being released on a button. The 'clicked' signal is also sent when the button is activated by way of the keyboard (pressing **Enter**, etc), as is common in many programs.

Inside **btnClick()**, we again only have one line, which calls some relatively simple PHP. However, it's important to realise that calling **echo** from inside your PHP-GTK scripts allows you to write to the console just as easily as if this were a standard PHP script. A helpful bonus is that one signal can execute several functions simply by calling connect() multiple times with varying second parameters.

There you have it! Although it may seem like an awful lot of explanation is required for what is actually quite a short piece of script, the large percentage has been the theory behind how the script works. Go ahead and run your script with the following command line:

```
php4 -q gtk1.php
```

If your PHP CGI is named differently, you shall need to make the appropriate change. However, as long as you have installed PHP-GTK correctly, the end result is the same - all being well, you should see something similar to screenshot 1, above left. **-q**, as you may know, is 'quiet mode' for the PHP CGI, and it forces PHP not to emit its standard HTTP headers. You can exclude this parameter if you are using the PHP CLI SAPI.

Multiple Windows

Once you are feeling confident with the whole situation so far, you can move onto a slightly more complicated script – here's the source code:

```
<?php

function doshutdown() {
  gtk::main_quit();
}

function btnClick($button) {
  $window =& new GtkWindow();
  $window->set_title("Spawning Windows 2");
  $window->set_default_size(300, 100);

  $label =& new GtkLabel("This is a new window.");

  $window->add($label);
  $window->connect("destroy","doshutdown");
  $window->show_all();

  return false;
}

$window =& new GtkWindow();
$window->set_title("Spawning Windows 1");
$window->set_default_size(300, 100);
$window->set_border_width(10);
$window->connect("destroy", "doshutdown");

```

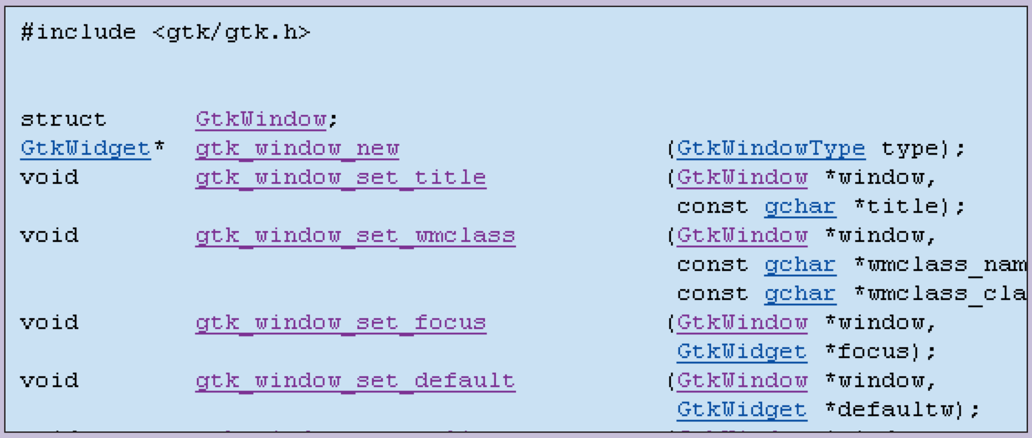
Get more out of GTK

Online resources to make your GTK code fly

The PHP-GTK documentation itself is a good enough place for some pieces of information, but, to be honest, it's rather weak on the whole; large chunks of code are left unexplained, various function calls are entirely undocumented, etc. However, it's a good place to start, so take a look at http://gtk.php.net/manual/en/.

http://gtk.miester.org is a fairly good website with regards to PHP-GTK, but there isn't much there, yet. This isn't terribly surprising because the GTK interaction with PHP is still working its way into the community. However, we anticipate this site will continue to grow healthily over time.

There is a long-running PHP-GTK mailing list available online at http://marc.theaimsgroup.com/?l=php-gtk. There's a lot of worthwhile information to be had here, but a great deal of it is repeated time and time again. However, you *can* search for particular terms, which increases its usefulness somewhat.



```
#include <gtk/gtk.h>


struct        GtkWindow;
GtkWidget*    gtk_window_new                         (GtkWindowType type);
void          gtk_window_set_title                   (GtkWindow *window,
                                                      const gchar *title);
void          gtk_window_set_wmclass                 (GtkWindow *window,
                                                      const gchar *wmclass_nam
                                                      const gchar *wmclass_cla
void          gtk_window_set_focus                   (GtkWindow *window,
                                                      GtkWidget *focus);
void          gtk_window_set_default                 (GtkWindow *window,
                                                      GtkWidget *defaultw);
```

The GTK+ C documentation. "It's documentation, Jim, but not as we know it."

What I believe will be the best resource for PHP-GTK in six months or so is the PHP-GTK Wiki. If you're new to the wiki phenomenon, then you've been missing out! "Wiki wiki" is Hawaiian for "quick", and it's basically a set of online documents that anyone – by default, absolutely everyone – can add to, edit, and delete from. The PHP-GTK Wiki is an online FAQ where visitors can easily add and amend entries to make the best documentation available, and it is picking up pace as this is being written.

If you find yourself confounded to get the information you're looking for about a particular widget or other GTK-related item, then you may find you have to bite the bullet and read the GTK C documentation available online at http://developer.gnome.org/doc/API/gtk/. This documentation is *very* thorough, however it's clarity leaves a little to be desired in places.

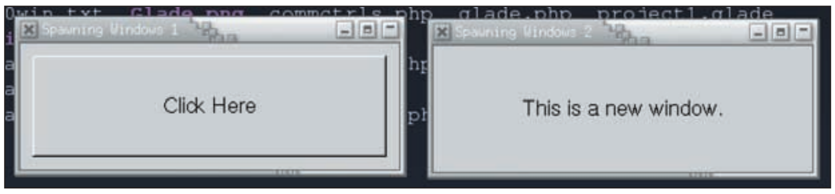
```
$button =& new GtkButton("Click Here");
$button->connect("clicked", "btnClick");
$button->set_relief(GTK_RELIEF_NONE);
$window->add($button);

$window->show_all();

gtk::main();
?>
```

You should recognise about half of the code from the previous script. For now, again, ignore the two functions near the top and concentrate on the main body of code.

We create our GtkWindow in the same way as last time, however this time we follow up with three new methods: **set_title()**, **set_default_size()**, and **set_border_width()**. These methods are all named quite clearly, but just to make sure we're on exactly the same wavelength, **set_title()** sets the titlebar caption for this window, **set_default_size()** sets the initial width and height of this window, and **set_border_width()** sets the amount of margin space in pixels on each edge of the window that is unavailable to child widgets.

If **set_border_width()** had been used with a positive value in the prior example, the GtkButton being used would not have taken up *all* of the space in the window, just what was left after the border.

For all intents and purposes, the rest of the script is pretty much similar, with the most notable exception being the function btnClick(). This time the function is much longer, and also has a return value. If your signal handlers send back **false** as their return value, PHP-GTK fires the default signal handler as soon as your function finishes, whereas if you return **true**, it is assumed that you wish no more processing to take place for this signal beyond any other signal handlers you have defined.



A slightly more complicated PHP-GTK script, this time with multiple windows being created.

At the start of **btnClick()**, a new GtkWindow is created with a new caption and a default size. Also, a new GtkLabel is created, which is a basic widget that allows you to display short amounts of text. As with GtkButtons, GtkLabels take the string they should display as a parameter when being created, and you can change this string at a later date by using the method **set_text()**.

Continuing on in the function, the label is added to the window, our shutdown function is connected to the 'destroy' signal, and the new window is shown. Attaching doshutdown() to the destroy signal of each window being created means that if the user closes *any* window, the application will terminate – you may want a different situation in your own programs.

The other change that you'll notice in this script is the call to the **set_relief()** method of our GtkButton. Like the **set_title()**, **set_default_size()**, and **set_border_width()** calls in this script, this method isn't *necessary*, but I've included it to show you more of the GTK functionality. This method takes one of three special constants: GTK_RELIEF_NORMAL (the default setting), GTK_RELIEF_HALF (much lighter shading for buttons), and GTK_RELIEF_NORMAL (no shading for buttons unless mouse is over the button).

Now you understand what's going on in the script, go ahead and save it as **gtk2.php** and run it. I've included another screenshot at the top of this column of how this should look when your mouse is over the button, although my mouse cursor is invisible in the screenshot.

»

Handling popup menus

With over a hundred different widgets supported, GTK+ makes for a very rich programming environment. However, there is a generally accepted set of 'standard' widgets that are most commonly used in applications, one of which is the popup menu.

If you're a long-time shell person who really doesn't know much about GUIs, then you should understand that popup menus are also known as context-sensitive menus and generally appear close to the mouse pointer when the right mouse button is pressed. Anyway, here's the next script:

```
<?php

function doshutdown() {
  gtk::main_quit();
}

function show_popup($event, $menu) {
  if ($event->button == 3)
    $menu->popup(null, null, null, $event->button, $event->time);
}

function mnunew_click($new) {
  echo "New clicked!\n";
}

function mnuopen_click($new) {
  echo "Open clicked!\n";
}

function mnuexit_click($new) {
  echo "Exit clicked!\n\n";
  doshutdown();
}

$menu =& new GtkMenu();
$new =& new GtkMenuItem("New");
$open =& new GtkMenuItem("Open");
$sep =& new GtkMenuItem("");
$sep->set_sensitive(false);
$exit =& new GtkMenuItem("Exit");

$menu->append($new);
$menu->append($open);
$menu->append($sep);
$menu->append($exit);
$menu->show_all();

$window =& new GtkWindow();
$window->set_title("Using menus");
$window->set_default_size(300, 100);
$window->connect("destroy", "doshutdown");

$window->add_events(GDK_BUTTON_PRESS_MASK |
GDK_BUTTON_RELEASE_MASK);
$window->connect_object('button-press-event', 'show_popup',
$menu);

$new->connect("activate", 'mnunew_click');
$open->connect("activate", 'mnuopen_click');
$exit->connect("activate", 'mnuexit_click');

$window->show_all();

gtk::main();

?>
```

If you're wondering at the length of that script, don't worry – it's the longest in this article! You already know quite a bit of what goes on in there also, so it's probably not all that fearsome.

Starting at the line **$menu =& new GtkMenu(),** a new class is introduced: GtkMenu. This widget is solely designed to host a popup menu, but it intertwines cleverly with GtkMenuBar, the widget designed to host the horizontal-style menu bar, and we'll discuss that later.

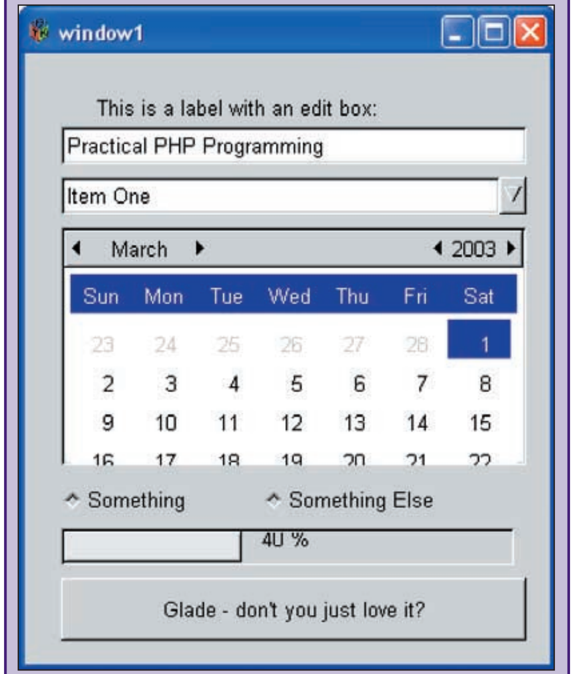
Distributing your applications

Platform-independent code?

PHP is cross-platform. GTK is cross-platform. So, surely it's easy to transfer your code from one platform to another, right? Well, sort of. In order to get PHP-GTK working, you need, unsurprisingly, PHP and GTK installed. Furthermore, if you wish to use *Glade*, you will also need *libglade* installed. Naturally, very few Windows users fit this criteria, so you have two options: either you provide very (very!) detailed installation instructions, or you package up the PHP interpreter, any DLLs you want to use, a pre-written php.ini file, and include it all in your installation. This is actually quite a good option, as you can be sure it will work with no hassle for your users, and it only adds around 2MB to your package once compressed.

For Unix users, generally speaking we're used to software having a few installation requirements, so generally noting on your website and in the docs "Requires PHP4.x, GTK 1.x" etc should be fine.

However, as you can see in the screenshot below, our Glade script runs perfectly well on Windows. Skins are unavailable in the existing Windows port of GTK, however, so sadly Windows users have to live with a less-than-beautiful interface.



Cross-platform coding in action. Yes, it's Microsoft Windows. Yes, you're still reading *Linux Format*...

Each GtkMenu contains several GtkMenuItems, and these are created similarly to GtkButtons and GtkLabels in that you passed the string you wish them to display.

Just before creating the last **GtkMenuItem**, I slipped a blank GtkMenuItem in there that some of you may not have noticed. This item, upon which I call the **set_sensitive()** method that is universal to all GTK widgets, is there to act as a separator between New/Open and Exit. Creating a GtkMenuItem with no text results in a blank menu item that may still be selected by users. Calling **set_sensitive()** on that widget and passing in *false* disables the widget.

Through the use of the **append()** method of GtkMenu we add our GtkMenuItems to our popup menu then create and set-up the GtkWindow itself. Two new methods are called here: **add_events()** and **connect_object()**.

add_events() is a peculiar but very helpful function that allows you to modify which events a given object captures. In essence, you can make a widget listen to an event it ignores by default. The method takes one parameter (we OR two parameters into one in the example), which is a bit mask of constants from the GdkEventMask list. In the long piece of script on the preceding page, GDK_BUTTON_PRESS_MASK and GDK_BUTTON_RELEASE_MASK are combined into one bitmask before being passed in, which makes the widget calling add_events, our GtkWindow, respond to mouse buttons being pressed and released. In turn, our GtkWindow will emit the signal **button-press-event**, which we bind a function to in the next line.

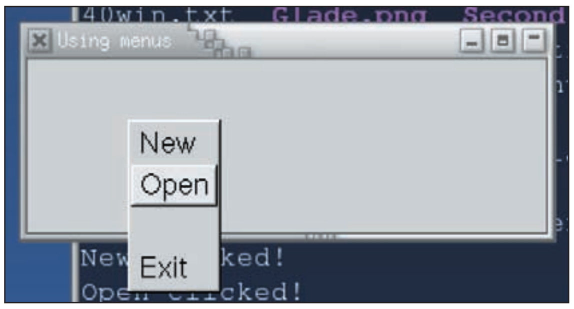
Similar to the **connect()** method we've been using so far, the **connect_object()** method also connects signals to functions, with the key difference that the object passed into the handler function isn't the object you used to call the method on. Instead, the object passed in is the one you set as parameter three to **connect_object()**.

You might not think this is particularly helpful, but in the example we set **$menu** as the object to be passed to our **show_popup()** function. If **connect()** had been used as opposed to **connect_object()** we would have had to try to get a handle to **$menu**, because a pretty useless reference to the GtkWindow would have been passed in. Whilst this can be overcome with custom parameters (see the box about *Using Custom Parameters* overpage for more information), it's more logical to use **connect_object()**.

So, at the end of the day, we tie our GtkWindow's button-press-event signal to our **show_popup** function. Note that the first parameter in **show_popup()** is of the type GdkEvent, which at the time of writing sadly seems entirely undocumented in the PHP-GTK documentation (please correct me if I'm wrong). However, it *is* documented in the GDK developer documentation, albeit in C++, which can be found at: **http://developer.gnome.org/doc/API/gdk/gdk-event-structures.html#GDKEVENT.**

The particular GdkEvent type we're interested in in this situation is GdkEventButton, documented at **http://developer.gnome.org/doc/API/gdk/gdk-event-structures.html#GDKEVENTBUTTON**. This event is sent when buttons are clicked and released, which is what we're looking to handle. If your C is sketchy, never fear – here's a quick breakdown of some of the data included in this event:

button – the mouse button press (left=1, middle=2, right=3)
time – the time, in milliseconds, that the event occurred



Popup menus are easy to create, and very powerful.

x, y – the x and y coordinate of the mouse
state – a bitmask of GdkModifierTypes (see the main PHP-GTK documentation) that describes whether Control was held down, etc
pressure – generally only used for graphics tablets, this is a floating point value from 0 to 1 describing how "hard" the button was clicked. This defaults to 0.5 for mouse clicks

So, the first line of the function checks which button was pressed to generate the event, and, if it was button 3 (the right mouse button), we call the **popup()** method of our GtkMenu.

popup() takes a total of five parameters: the first two are generally null as they are only used when tying menus to GtkMenuItems. Parameter three is null in the example, but can be the name of a function to call to return the x and y coordinates at which you wish your menu to appear as an array. So, for example:

```
function mnupos() {
  return array(50, 200);
}

$menu->popup(null, null, 'mnupos', $event->button, $event-
>time);
```

When the third parameter is null, the current mouse coordinates are used, which is usually the desired result. Parameter four is the button that was pressed to generate the event, and finally parameter five is the time the event took place, in milliseconds. As seen above, even though there are five parameters for the method, you'll see that it's actually quite straightforward to use.

The **mnuexit_click()** function could have been removed entirely because, as well the fact that multiple functions can be connected to a single signal, multiple *signals* can be connected to a single *function*. If there were no special processing to be run when Exit was clicked (in our example we echo to the console), then the activate GtkMenuItem signal could have been connected to **doshutdown()** as well as the GtkWindow destroy signal.

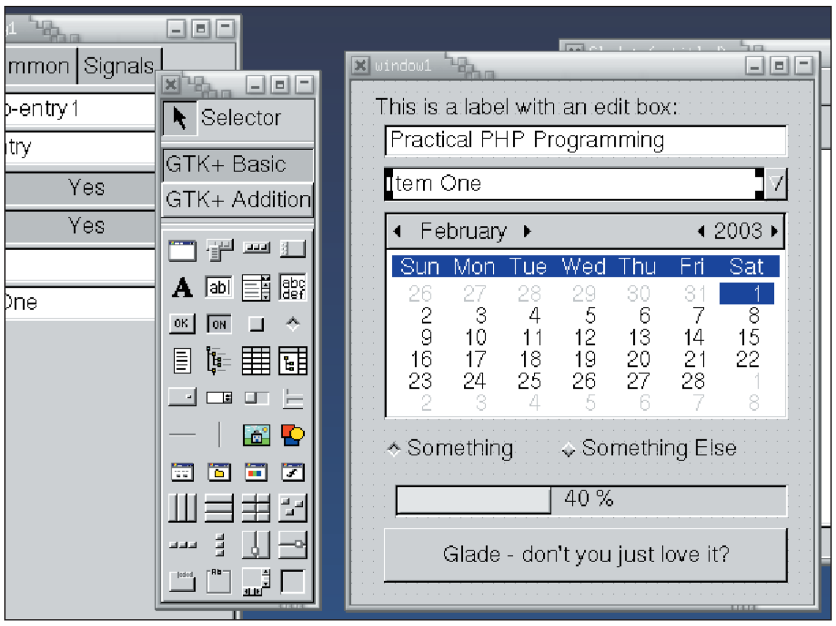
Save this script as **gtk3.php** and run it as before. Again, check my screenshot at the top of this column to make sure you got everything right.

As promised earlier in this tutorial, I want to briefly mention how GtkMenu bar works. A menu created with GtkMenu can be used as seen above, where it is activated in a floating space. It can also be used to provide the contents of a horizontal-style menu bar item, for example 'New, Open, Save, Exit' would be the GtkMenu that was attached to the 'File' GtkMenuItem of a GtkMenuBar widget.

Does this make sense? If not, here's a quick piece of code to demonstrate what I mean:

```
$mainmenu =& new GtkMenuBar();
$filemenu =& new GtkMenuItem("File");
$mainmenu->append($filemenu);
```

»


Common
Signals
Selector
GTK+ Basic
GTK+ Additional
window1
This is a label with an edit box:
Practical PHP Programming
Item One
February
2003
Sun Mon Tue Wed Thu Fri Sat
Something
Something Else
40 %
Glade - don't you just love it?

With *Glade*, interface design is a snap.

«

```
$filemenuoptions = &new GtkMenu();
$open =& new GtkMenuItem("Open");
$filemenuoptions->append($open);
$save =& new GtkMenuItem("Save");
$filemenuoptions->append($save);
$filemenu->set_submenu($filemenuoptions);
```

So, a GtkMenuBar is the menu strip along the top of your window. The top level items (eg: File, Edit, Document, Bookmarks, etc, in KDE's Kate 2.1) are GtkMenuItems, which each contain a GtkMenu of their contents. The 'Document' GtkMenuItem in *Kate* would contain a GtkMenu which itself contained GtkMenuItems for Back, Forward, and any open files.

Advanced GUIs

There are so many possibilities using PHP-GTK that, sadly, I've had to pick and choose what I can cover here owing to space reasons – and that's despite the fact that this tutorial is extra long! So far we've looked at windows, buttons, labels, menus, and menuitems.

What we're going to look at now is an easy way to use all sorts of GTK widgets, perfectly lined up where you want them to be, with many widgets in the same window, and, surprisingly enough, with almost no work. This is the power of *Glade*.

Available from **http://glade.gnome.org**, *Glade* is a GPLed GTK+ user interface builder designed to allow you to design and build your GUI, including defining signal handler functions, with little work.

Take a look at the screenshot at the top of this column to see *Glade* in action. As you can see in the picture, you have a big toolkit available to you under the 'GTK+ Basic', and another large toolkit available under 'GTK+ Additional'. When you want to make use of a particular widget, you simply have to select from the toolbox and 'draw' on your window. Properties can be set from a property editor which is partly offscreen to the left. Once you're finished designing the parts of your GUI, you can even instruct *Glade* to generate source code for you, although sadly, this is not yet available in PHP.

However, there is still a way *Glade* can be used with PHP. Take a look at this final script:

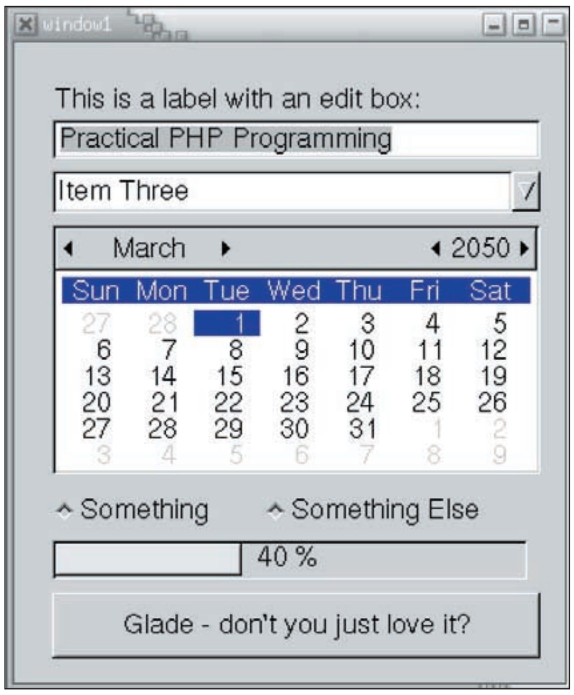
```
<?php

function doshutdown() {
  gtk::main_quit();
}

$layout = &new GladeXML('complex_interface.glade');
$layout->signal_autoconnect();

$window = $layout->get_widget('window1');
$window->connect("destroy", "doshutdown");

Gtk::main();
?>
```

Here, there is a new class available if you have *libglade* installed. That takes the .glade project file that *Glade* saves for its own purposes and translates that into a GUI. This GUI, stored in **$layout** in the example above, can then have its signals connected using the GladeXML method **signal_autoconnect()**.

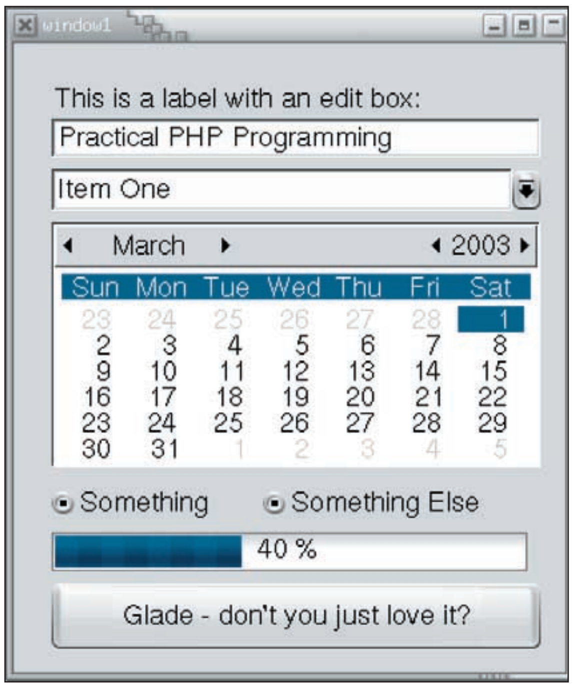
In order to provide a clean shutdown of the script, I have used the GladeXML method **get_widget()** to grab the main window. Note that you may need to change this line if you have used a particular name for your window in *Glade*. **get_widget()** takes just the one parameter, which is the name of the widget you wish to get from the layout, and returns the widget for you to use.

With our GtkWindow reference, I have connected the destroy signal to our usual **doshutdown()** function, and that's the end of the script.

If you save that as **gtk4.php**, you can then go experiment with *Glade* to see what you can make. You can see my interface live in action in the screenshot below, though you'll see that some bits are still labelled 'something'. As you can see, using *Glade* takes all the hard work away from designing a GUI. All that's left to do now is to write handlers for all the signals you wish to work with, and your interface is done.


window1
This is a label with an edit box:
Practical PHP Programming
Item Three
March
2050
Sun Mon Tue Wed Thu Fri Sat
Something
Something Else
40 %
Glade - don't you just love it?

Here is the script running, and it looks just like it did when it was being built in *Glade*. Magic.


window1
This is a label with an edit box:
Practical PHP Programming
Item One
March
2003
Sun Mon Tue Wed Thu Fri Sat
Something
Something Else
40 %
Glade - don't you just love it?

Our *Glade*-built GUI doing a KDE impression.

Using Custom Parameters

Get connected

When you're connecting signals to functions, it is possible to add one or more custom parameters to the signal handler. As seen earlier, **connect_object()** can be used to pass a particular widget into a function, however an alternative is to use **connect()** with extra parameters for the information you wish to use inside the function.

So, **connect()** could have been called like this:

```
$window->connect('button-press-event',
'show_popup', $menu);
```

The handler function would then need to have accepted three parameters - the event, the GtkWindow object that emitted the signal, and the custom parameter $menu. The only difference here is that behind the scenes a little more data needs to be passed around for the handler function to be called with the extra parameter.

Here's a complete code example you can try out to get the idea:

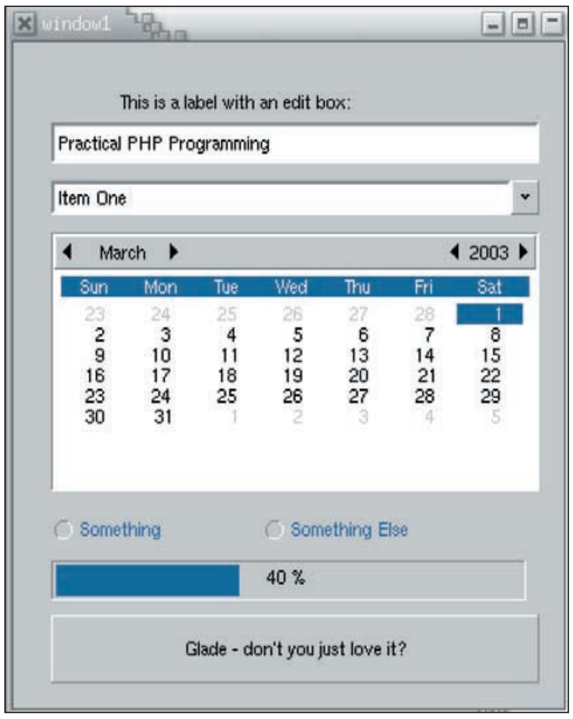
```
<?php

function btnClick($button, $window) {
  $window->destroy();
  gtk::main_quit();
}

$window =& new GtkWindow();
$btnquit =& new GtkButton("Quit");
$btnquit->connect("clicked", "btnClick",
$window);
$window->add($btnquit);
$window->show_all();
gtk::main();

?>
```

As you can see, the parameter **$button** isn't being used inside btnClick(), however that's hardly a major speed hit.

Themes

Some people say that unthemed GNOME looks a little unsightly, and they might be right. However, luckily all applications made using PHP-GTK happily work with GTK+ 1.x themes with no additional work. I myself have GNOME 2.2 installed, and so I needed to specifically install a GTK1 theme as GTK2 themes do not work with PHP-GTK or other GTK1 apps.

Anyhow, themes are all transparent to your scripts – you can't tell whether a particular theme is enabled or not, and neither should you need to as your scripts will adapt to whatever the user has selected.


window1
This is a label with an edit box:
Practical PHP Programming
Item One
March
2003
Sun Mon Tue Wed Thu Fri Sat
Something
Something Else
40 %
Glade - don't you just love it?

And again, this time doing a Microsoft Windows impression.

Take a look at screenshots on this page to see the same *Glade* GUI interface themed in different ways.

Conclusion

If you've made it this far, the chances are the sun will be rising outside shortly and you had best get some sleep before morning! However, hopefully you will have learned a great deal about the coolest – and probably least-exploited – alternative use for PHP.

Creating graphical applications for PHP may, at first, not seem "right", which is quite true to some extent. After all, PHP was designed to be a language for general web use, and not for GUIs. However, once you get over the initial, and indeed inevitable shock of switching to signal-based programming, it is normally a pleasant experience.

The GTK version used with PHP-GTK is quite old, and recent releases have been far superior, with a lot of work done to GTK to make it more flexible, with more intuitive interaction between objects. While GTK2 support is not currently on the cards for PHP-GTK, I shan't imagine it will be long. Until then, there's more than enough information to help you get to grips with using the current release of PHP-GTK - good luck! LXF

Make your mark

Brainstorms 'R' Us

Would you like to get your name in the mag and learn about stuff you're most interested in?
We're looking out for ideas for new *Linux Format* Practical PHP tutorials, and where better to look than to you, the reader? If, while reading past issues of Practical PHP, you've thought ***"I wish they'd covered XYZ in more depth..."***, or ***"I really want to know how to use..."***, then now's the time to get your voice heard!
Send an email to **paul.hudson@futurenet.co.uk** with your ideas – all the good ideas that you send in will be covered in future issues. So far, the topics we have covered in some depth include MySQL, XML, CLI, GUIs, media generation, templates, and more.
If you're short of ideas, you're certainly welcome to write in with comments about prior issues – we're always looking to improve the overall quality of tutorials.

NEXT MONTH

Next month we'll be looking at potentially the most complicated aspect of PHP, and that is creating your own modules for the language. If you thought this month's topic was tricky, you've got another think coming – next month, knowledge of C is strongly recommended.

If you have any comments or suggestions about this series, please be sure to write in.