Maximizing Your Resourcefulness - Dealing With Widget Resources, Part 1

Authored by: **David W. LeBlanc**
Updated by: **Mikhail Nefedov**

One of the biggest hurdles in developing Photon applications, for beginners and seasoned developers alike, is manipulating widgets. Since widget manipulation involves determining and changing a widget's state, you need to deal with widget properties, known in Photon lingo as "resources". For those of you acquainted with PhAB, you know that changing widget properties in this environment is as simple as pointing and clicking (and in some cases, typing). Inevitably though, any useful application will have to do the same at runtime - outside of the PhAB environment. That means manipulating widget resources using code, and that's a different story altogether.

Fortunately, the Photon toolkit provides a very concise API for dealing with widget resources that remains consistent for all widgets, great and small. But before we take a look at the API itself, we need to consider the two types of resource interactions that can take place.

The first is resource setting, which involves modifying the value of a widget's resource. You do this when you need to change the state of a widget. For instance, in your application you may want to update a **PtLabel** that contains a status message for what your app is doing. This involves setting the text resource (**Pt_ARG_TEXT_STRING**) of the **PtLabel** to reflect a new value (i.e. the new status message).

The second resource interaction is resource getting, which involves discovering the value of a widget's resource. You do this when you need to know the value of a particular resource. For instance, your application may contain a text field that it needs to read to get input from the user. This involves getting the text resource (**Pt_ARG_TEXT_STRING**) of the **PtText** to find out what the user has typed.

To address these two types of interactions, the resource manipulation API contains two functions: **PtSetResources**() and **PtGetResources**(). That's it!

Obviously there must be a little more to it, or this would be a pretty short and unsatisfying article. Well, there is ... but many people find widget manipulation scary and mysterious, so it's important to underscore that it's really not that bad.

If we look at the prototypes for these two functions, we see that they are, in fact, identical. Both take three parameters, in the following order: `PtWidget_t *widget, int n`, and `PtArg_t *args`. Clearly the widget argument indicates the widget to operate on, but what about the other two parameters? And how is it that two functions that are effectively opposites can share the same parameter list?

The answer to both these questions lies in the central building block for all widget resource transactions - the concept of argument lists (more simply known as "arg lists" or just "args").

An argument list is specified by two parameters: an array of **PtArg_t** structures as well as the number of elements in that array. This explains the two arguments in our aforementioned functions, **PtSetResources**() and **PtGetResources**(). The **n** specifies the number of elements in the **args** array and **args** is a pointer to that array.

A good understanding of resource manipulation begins (and ends) with a good understanding of what a **PtArg_t** is, and how to fill out the structures to get the result you want.

PtArg_t is composed of the following pieces, which, when put together, specify all the information necessary for one complete resource transaction to take place:

long type - Every widget resource has an associated numerical value that uniquely identifies that resource. Resources are enumerated systematically. For example, if we refer to the header file for the **PtLabel** class, (*<PtLabel.h>* in */usr/include/photon*), we see that the identifier for **Pt_ARG_TEXT_STRING** is 3011. So, the type member of **PtArg_t** identifies the resource being operated on using these values. If you then want to operate on, say, a **PtLabel**'s text resource (**Pt_ARG_TEXT_STRING**), you would set this value to **Pt_ARG_TEXT_STRING** (which the preprocessor would map to 3011).

long value - While the meaning of this resource is dependent on the type of resource (and by type I'm referring to resource classifications, which we'll get into later), it generally refers to the new value when setting a resource, or an address where the result can be stored when getting a resource. (This concept will be mentioned again in part 2 of this series, when we discuss the different resource classifications, so don't worry if it doesn't seem clear yet.)

long len - Once again, the meaning of this member is resource-dependent, and often doesn't even apply. When used it's generally coupled with the value member in some way. This member is named as such because it was originally used exclusively to describe the length or size of the data referred to by the value member. But as the library grew, new resource types got introduced that required different information that could still be conveyed using this member. So even though it didn't make much sense semantically, the name for this multipurpose member stuck in order to provide source-level backward compatibility.

If **len** does not apply to the particular resource you're working with, you should set it to zero. Even if **len** is documented as ignored, future additions to the library may attach a meaning to it, which will break your application if you're setting **len** to something other than zero.

There are several methods of setting up an argument list. One method involves assigning each member in each **PtArg_t** directly, as follows:

```
PtArg_t args[10];
...
arg[0].type = Pt_ARG_TEXT_STRING;
arg[0].value = (long)"New string";
arg[0].len = 0;
...
```

Another method can be used to statically initialize an array of **PtArg_t** structures like this:

```
PtArg_t args[] = { { Pt_ARG_TEXT_STRING,
(long)"New string",0 },{ ... } };
...
```

The two methods mentioned above work and both are valid C code. However, you won't find them documented in the Photon manual. The two methods you will find are:

```
PtArg_t args[2];
...
PtSetArg(&args[0],Pt_ARG_TEXT_STRING,"New string",0);
...
```

Or, for statically initializing an argument list:

```
PtArg_t args[] = { Pt_ARG(Pt_ARG_TEXT_STRING,
"New string",0),Pt_ARG(...) };
```

Photon provides the macros **PtSetArg**() and **Pt_ARG**() and I recommend using them because they're more concise and readable. They're also easier to use since they perform all of the necessary casting for you, allowing you to be lazy and not have to put up with compiler warnings!

Avoiding array overruns by super-sizing your fries

From the viewpoint of human error, statically initializing your **arg** lists is usually pretty safe. You'll probably do something like:

```
PtArg_t args[] = { { Pt_ARG_TEXT_STRING,
"New string",0},{...} };
```

and then later do:

```
PtSetResources(widget,sizeof(args) / sizeof(args[0]),args);
```

Using **sizeof** in this manner protects you from having to update the number of **args** as you go back and add or remove arguments from your list.

If, however, you need to use **PtSetArg**() to initialize your **arg** lists (since statically initializing them is not always possible), you'll need to explicitly specify the size of the arg list when declaring it. This is inherently more dangerous, because then the compiler can't protect you from overrunning your array. Array overrun occurs when you **PtSetArg**() a member of your array that goes outside of the bounds you've declared.

If, for instance, you add a few **args** to an **arg** list that was already a tight fit and forget to go back and bump up the size accordingly in your declaration for the array, you will fall victim to this pitfall. Usually, if you're lucky, this will cause your program to segment violate at once. Note that I said "usually" and not "always." If you're unlucky you'll merely corrupt memory, which will cause problems down the road that will very likely be difficult to debug. So make life easier on yourself and be generous when declaring your **args** arrays. Make them a bit larger than you think you'll need, particularly in the initial stages of development. Setting or getting, say, six resources? Bump up your **args** array to a size of 12. This will give you a little more elbow room, lending a bit of forgiveness down the road to your already taxed memory. As your product solidifies and you approach the optimization phase of development, then you can go back and tighten up your code.

Ensuring a large and/or constantly changing arg list is always "in sync"

If you're using the **PtSetArg**() method for constructing argument lists, it's often easier and safer (particularly early in the development cycle) to use a counter variable in your initializations.

Consider the following code:

```
PtArg_t args[10];
...
PtSetArg(&args[0],yada,yada,yada);
PtSetArg(&args[1],yada,yada,yada);
PtSetArg(&args[2],yada,yada,yada);
PtSetResources(wgt,3,args);
```

This is perfectly valid as is (except, of course, for the "yadas"). However, if you're making a lot of modifications to the code (usually the case early in the development cycle), you might be adding and removing **PtSetArg**()s frequently, making this area of your software particularly vulnerable to bugs.

What if, for instance, you removed the first **PtSetArg**() and forgot to collapse the indices in the remaining two? **arg**[0] will remain uninitialized and thus contain garbage when you call **PtSetResources**(), and that will yield unpredictable results. Or, what if you did remember to adjust the remaining two indices but forgot to adjust the number in the **PtSetResources**()? Similar mischief will ensue.

The same code is much less susceptible to problems if we introduce a counter variable and use it as follows:

```
PtArg_t args[10];
int n;
...
n = 0;
PtSetArg(&args[n++],yada,yada,yada);
PtSetArg(&args[n++],yada,yada,yada);
PtSetArg(&args[n++],yada,yada,yada);
PtSetResources(wgt,n,args);
```

This way, you can add and remove lines as you please, resting assured that there will never be "holes" in your **arg** list, and that you will always be setting or getting the right number of arguments. There are only two potential problems in this case: forgetting to initialize your counter variable, which will usually cause your program to segment violate at once; or overrunning your array, which, as mentioned above, can be a bit trickier. Fortunately this risk is easy to reduce by following the guidelines discussed previously.

Getting or setting one resource at a time

When I said that there are only two API calls for manipulating resources, I was lying a little. There are actually two additional macros, **PtGetResource**() and **PtSetResource**(), to make getting or setting a single resource at a time less cumbersome. While they both share the same prototype, that prototype differs from its multiple-resource counterparts (`PtWidget_t *widget, long type, long value, long len`). Note that the macros do not use a **PtArg_t**, which means you won't have to declare and initialize one. Rather, they take type, value and **len** directly. The meanings for the parameters are the same as their corresponding meanings within the **PtArg_t** structure.

Conclusion

We've touched on the fundamentals of using the resource manipulation API. Next, we need to consider how to manage the vast collection of resources that span the widget hierarchy. Luckily, they can be classified into a small number of types, with fixed rules for dealing with them. In Part 2 of this series, we'll explore those types and their associated rules.