QNX® Neutrino® Realtime Operating System

Photon® microGUI
Programmer's Guide

For QNX® Neutrino® 6.5.0

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published 2010.

Contents

List of Figures

About This Guide

What you'll find in this guide

The Photon *Programmer's Guide* is intended for developers of Photon applications. It describes how to create applications and the widgets that make up their user interfaces, with and without using the Photon Application Builder (PhAB).

If you're familiar with earlier versions of Photon, you should read the What's New appendix. to find out how Photon and its widgets have changed in this release.

This table may help you find what you need in this book:

For information about:	**See:**
Photon, widgets, and PhAB	Introduction
Getting started with PhAB	Tutorials
PhAB's user interface	PhAB's Environment
Creating, opening, and saving applications in PhAB	Working with Applications
PhAB modules, such as windows, dialogs, and menus	Working with Modules
Adding, deleting, and modifying widgets in PhAB	Creating Widgets in PhAB
Initializing a widget's resources and callbacks	Editing Resources and Callbacks in PhAB
Setting the sizes of a widget and its children	Geometry Management
Getting PhAB to generate code	Generating, Compiling, and Running Code
Editing code generated by PhAB	Working with Code
Getting and setting widget resources	Manipulating Resources in Application Code
Adding or modifying widgets "on the fly" at runtime	Managing Widgets in Application Code
Building special areas into a widget	Control Surfaces
Using internal links to refer to PhAB modules	Accessing PhAB Modules from Code
Developing a multilingual application	International Language Support

continued...

For information about:	**See:**
Adding help information to your application	Context-Sensitive Help
Communicating with a Photon application	Interprocess Communication
Threads, work procedures, and background processing	Parallel Operations
Using `PtRaw` and Photon's low-level drawing routines	Raw Drawing and Animation
Encodings, fonts, languages and code tables	Understanding Encodings, Fonts, Languages and Code Tables
Photon's fonts	Fonts
Printing in a Photon application	Printing
Transferring data from one widget or application to another	Drag and Drop
Photon's regions	Regions
Interaction between applications, users, and the Photon server	Events
Working with windows and modal dialogs	Window Management
Developing applications "by hand" without PhAB	Programming Photon without PhAB
Photon's implementation	Photon Architecture
PhAB's widget icons	Widgets at a Glance
Handling international characters	Unicode Multilingual Support
Building an embedded system	Photon in Embedded Systems
Differences between the Windows and native QNX Neutrino versions of PhAB	Using PhAB under Microsoft Windows
Photon terminology	Glossary

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	**`-lR`**
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	**`/dev/null`**
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	**`something you type`**
Keyboard keys	Enter
Program output	`login:`
Programming constants	NULL
Programming data types	`unsigned short`
Programming literals	`0xFF`, `"message string"`
Variable names	*stdin*
User-interface components	**Cancel**

We use an arrow (→) in directions for accessing menu items, like this:

> You'll find the **Other...** menu item under **Perspective**→**Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (`/`) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

To obtain technical support for any QNX product, visit the **Support** area on our website (`www.qnx.com`). You'll find a wide range of support options, including community forums.

Chapter 1
Introduction

In this chapter. . .

By now, you've probably seen and tried various Photon applications—the window manager, Helpviewer, games, and so on—and you're ready to develop your own. This chapter introduces you to the basic terms and concepts you'll use when developing a Photon application.

Overview of the Photon architecture

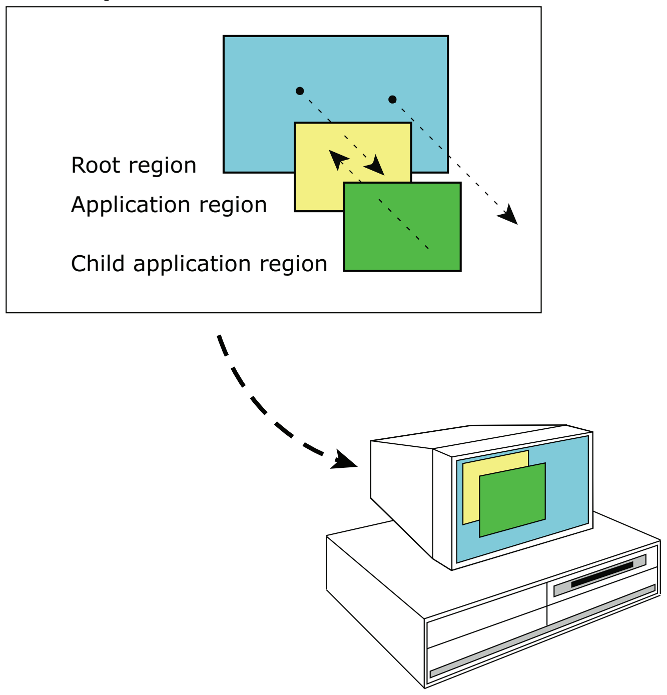
The Photon manager runs as a small server process, implementing only a few fundamental primitives. It creates a three-dimensional *event space* populated by *regions* and *events*. This manager can't draw anything or manage a mouse, keyboard, or pen.

External, optional processes — including device drivers and window and other managers — implement the higher-level functionality of the windowing system. They communicate by emitting events through the Photon event space.

A Photon application consists of one or more flat, rectangular regions that act as its "agents" in the event space. The application draws inside the regions. Regions are stacked on top of each other in the Photon event space. A region can have a parent region as well as siblings.

The user sits outside the event space, looking in from the front. The very back of the event space is a special region called the *root region*.

Event space



Photon's event space from the user's perspective.

When you run the application, you interact with it, and it interacts with other applications and Photon, in many ways: you press keys and mouse buttons, the application performs graphical operations, and so on.

These interactions are called *events*; they travel between regions in the event space like *photons* or particles of light. For example:

- When you press a mouse button, the device driver emits an event and sends it back through the event space (toward the root region). A region that's interested in the event can catch it and process it, activating a push button or other UI element.
- When your application wants to draw something, it emits an event and sends it toward the front of the event space (toward the user). A driver can catch the event and render the drawing on the screen.

Each region can determine which events it's interested in by setting its *sensitivity* and *opacity*:

- A region that's sensitive to a type of event notifies the application whenever such an event intersects it.
- A region that's opaque to a type of event blocks it by clipping its own area out of the event's area.

For more information, see the Photon Architecture appendix in this guide.

Photon uses a *draw buffer* to queue a series of draw commands (called the *draw stream*) for the application. Once the buffer is full or the application calls *PgFlush()*, the list of commands is sent to the Photon server. After that it's typically sent to **`io-graphics`** (see the *Utilities Reference*), which then interprets and renders the draw stream.

You can change the size of the draw buffer by calling *PgSetDrawBufferSize()*. The optimal size depends on what you're drawing and how often you flush the buffer.

Your application can work in *normal* or *direct mode*; buffering works in both modes.

Normal mode
: The application sends the draw stream to the Photon server, which then does some work on it, such as adding clipping to represent the regions or windows above the application and sending the draw stream to the regions that are sensitive to draw events (e.g. **`io-graphics`**, **`phrelay`**).

Direct mode
: The draw stream is sent directly to **`io-graphics`**. The Photon server doesn't see it or process it, so there are fewer context switches (switching from one process to another) and fewer operations done on the stream, which results in a significantly faster way of drawing.

  For more information, see "Direct mode" in the Raw Drawing and Animation chapter.

Photon Application Builder (PhAB)

The Photon microGUI includes a very powerful development tool called the *Photon Application Builder* (PhAB). It's a visual design tool that generates the underlying C and/or C++ code to implement your application's user interface. With PhAB, you can dramatically reduce the amount of programming required to build an application. You can save time not only in writing the user interface portion of your code, but also in debugging and testing. PhAB helps you get your applications to market sooner and with more professional results.

If you're using the Windows-hosted version of PhAB, you should read the appendix, Using PhAB under Microsoft Windows.

PhAB takes care of designing and creating *modules* (e.g. windows, menus, dialogs, and icons), and *widgets* (e.g. buttons and labels). It also helps you create *widget callbacks*, special resources that connect a widget to your application's code or link a widget to a PhAB module. For more information, see "Widget concepts" later in this chapter.

PhAB lets you access and create PhAB modules within your own code. It also provides a number of utility functions to set up databases of widgets that you can reuse as many times as you need, rather than create widgets from scratch.

Get immediate results

PhAB lets you bypass the trial-and-error process of creating a user interface by hand. Instead of having to write code for every button, window, or other widget, you just "point and click" to create the widgets you want.

As soon as you create a widget, PhAB displays it on the screen, along with all the resources that control how the widget looks and behaves. Changing any widget resource is easy—just click on the resource, choose a new value, and you're done. It's just as easy to move or resize a widget—simply click and drag the widget.

Concentrate on functionality

Like other GUI development environments, PhAB lets you attach code functions to a widget's callbacks so you can implement your application's main functionality. For example, you can attach a code function to a button so that the function is invoked whenever the user clicks the button.

In addition, PhAB doesn't force you to write and attach the code needed to "glue" the different parts of your interface together. Instead, you can attach a widget's callbacks directly to any window, dialog, or menu. The only code you have to worry about is the code specific to your application.

Create prototypes without writing code

Once you've completed any part of a user interface, you can have PhAB generate all the C and/or C++ code required to bring the interface to life. Which means you can create a complete prototype *without having to write a single line of code.*

After you've generated and compiled the code, you can run the prototype to see how the interface works. For example, if you link a button to a dialog, clicking on the button causes the dialog to pop up. You immediately get a sense of how the interface will "feel" to the user. In fact, PhAB makes the process of building and testing so efficient that you can even sit down with your users and design prototypes together.

After you've finished a prototype, you can build it into your working application. Or you can stop prototyping at any point, write some callback functions, experiment with your application to see how it works, and then return to prototyping. You're always free to fine-tune all aspects of your application until it looks and works just the way you want.

Cut code size

Your application may need to use the same widgets in several parts of its interface. With PhAB, you don't have to set up these widgets every time they're needed. Instead, you define the widgets just once, place them in a widget database, and then, using C functions provided by PhAB, reuse the widgets as often as you want. By taking this approach, you can reduce the code required to create a widget to a single line.

Create consistent applications

With PhAB, you rarely have to build an application from scratch. For example, if you've already created windows and dialogs for an existing application, you're free to drop these into a new application. You can also create a central database of widgets that you import into all your applications to create a consistent look and feel.

Create all kinds of applications

With PhAB, you can speed up development without compromising functionality, so you're free to create all kinds of applications. For example, we used PhAB to build almost all the applications that ship with Photon, including the Helpviewer, Terminal application, Desktop Manager, Snapshot, all games and demos—even PhAB itself!

The best introduction to PhAB is using it, so start by working through the tutorials. Within a very short time, you'll be able to put together very detailed prototypes. When you're ready to start programming your application, you can then read the sections pertaining to the widgets you're trying to use.

Widget concepts

When creating a new user interface (UI), you'll find it much simpler to compose the interface from a set of standard components, such as sliders, lists, menus, and buttons, than to implement each UI element from scratch. Each standard component included in the UI is an object called a *widget*.

Photon widgets implement a set of UI components that are more or less consistent with other windowing systems you may have seen.

The widget set is built on an object-oriented framework loosely based on the X Toolkit Intrinsics library (Xt). If you're already familiar with Xt, you'll see that many of the same concepts apply here.

A widget combines the data and operations required to implement a particular UI element. Grouping data and operations into an object like this is called *encapsulation*. A widget encapsulates the knowledge of how to:

- draw itself
- respond to user events (e.g. pressing a pointer button)
- repair itself by redrawing when it's *damaged* (for example, when a window that obscures it closes).

In addition, there are some widgets called *containers* that hold other widgets and manage their layout.

A widget also hides the details of how it performs these responsibilities from the outside world. This principle, known as *information hiding*, separates the widget's internal implementation from its public interface.

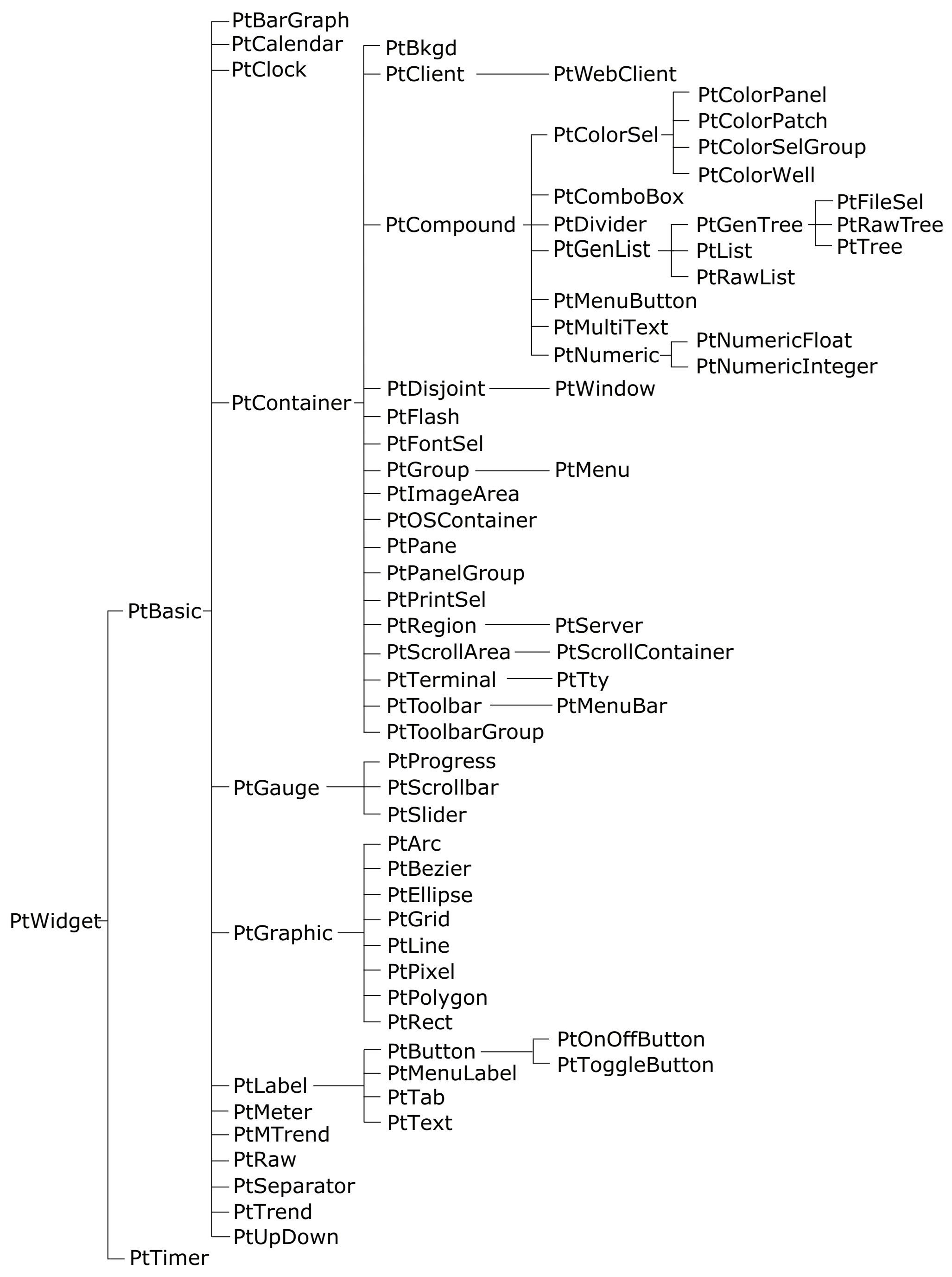
The public interface consists of all the attributes visible to other objects as well as the operations other objects may perform on the widget. The attributes in the widget's public interface are called *resources*.

The advantage to you as a programmer is that you don't have to know the implementation details of a widget to use it—you just need to know the public interface for the widget, how to create and destroy the widget, and how to manipulate its resources.

Not every object is unique. Objects that perform the same function and provide the same public interface belong to the same *class*. Widgets that provide the same UI component belong to the same *widget class*. The window's *class methods* implement the common functionality by the class.

Several widget classes may have attributes and operations in common. In such cases, these widget classes may be categorized as *subclasses* of the same *superclass* or *parent* class. The attributes and operations are encapsulated in the superclass; the subclasses *inherit* them from the parent class. The subclasses themselves are said to be inherited from the superclass.

The Photon library allows a widget class to be inherited from only one widget class. This relationship is known as *single inheritance*. The relationships between all of the widget classes can be drawn as a tree known as the *class hierarchy*.



The Photon widget hierarchy.

The nesting of widget instances in your application's GUI produces another widget hierarchy, called the *widget family* to distinguish it from the widget class hierarchy.

The Photon widget library acts like a widget factory. It provides a set of functions that let you to create a new widget of a particular widget class and then manipulate that widget. Once created, the widget has all the characteristics of the widget class. Because of inheritance, it also has all the characteristics of the superclasses of its widget class.

The new widget is an *instance* of the widget class. Creating a new widget of a particular class is thus also called *instantiating* the widget. This term isn't entirely accurate, however, because you're really instantiating the widget *class*. This reflects a tendency found throughout this guide to refer to *both* widgets and widget classes simply as "widgets."

The widget's resources are used to configure its appearance or behavior. You can edit resources in PhAB, and after the widget has been created you can change many of them with a call to *PtSetResource()* or *PtSetResources()*. Resources are used extensively to control the data displayed by a widget and to customize how it's displayed. For example:

- The *Pt_ARG_TEXT_STRING* resource for a **`PtLabel`** widget is the string that it displays.
- the resources for a **`PtButton`** widget specify whether the button displays a string and/or an image, its text, image, color, and what happens when the user selects the button.

How you get and set widget resources in your application code depends on the type of resource. For more information, see the Manipulating Resources in Application Code chapter.

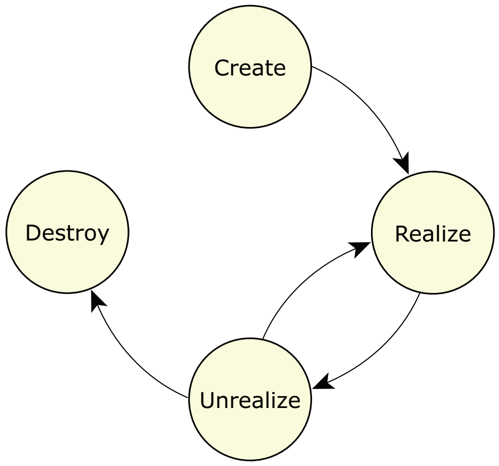
An important type of resource provided by widgets is the *callback list*, which is a list of functions that the widget invokes in response to some significant user event. For example, a text field widget calls the functions in one of its callback lists whenever the user enters a new value and presses Enter. When you develop an application, you can add *callbacks* to a widget's callback list in order to perform the appropriate action in response to a user event.

For more information about adding callbacks, see:

- "Callbacks" in the Editing Resources and Callbacks in PhAB chapter
- "Callbacks" in the Managing Widgets in Application Code chapter.

Widget life cycle

A widget has an inherent life cycle, as shown below.



Life cycle of a widget.

1. When the widget is required, it's *created* or *instantiated*. After being created, its attributes may be manipulated, or operations may be performed on it.

2. After a widget has been created, it's not immediately visible in the UI. It must be *realized*. If you're using PhAB, your widgets are realized automatically; if you're not using PhAB, you must realize them using *PtRealizeWidget()*.

   Realizing a widget automatically realizes all its descendants. Photon guarantees that all the descendants are realized before the widget itself, so the widget can calculate its initial size based on the sizes of its children. To have the application notified that the widget has been realized, you can register a callback on the *Pt_CB_REALIZED* callback list.

3. You can temporarily hide a widget from the UI by *unrealizing* it using *PtUnrealizeWidget()*. As with realization, you can notify the application, using the *Pt_CB_UNREALIZED* callback resource.

4. When the widget is no longer required, you can destroy it.

   You can destroy a widget by calling *PtDestroyWidget()*. The call doesn't actually destroy the widget immediately—it's marked to be deleted by the toolkit at an appropriate time and added to a list of widgets to be destroyed. These widgets are normally destroyed within the main loop of the application, after all the callbacks associated with an event have been invoked.

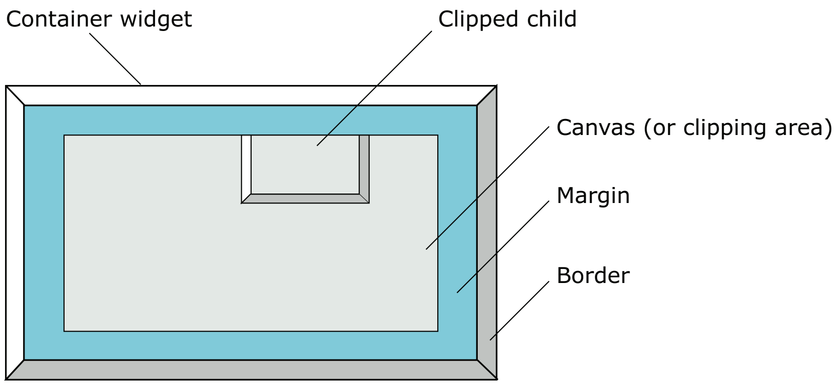
   Your application can define *Pt_CB_DESTROYED* callbacks for any widget. These callbacks are invoked when the widgets are marked for destruction.

   To have the application notified when the widget is actually destroyed, you can register a function with the destroy callback list (*Pt_CB_IS_DESTROYED*) for the widget. This is especially useful for cleaning up data structures associated with the widget.

Widget geometry

You can think of a widget as a painting or mounted photograph. The widget is held by a frame called a *border*. For a widget, the border is the set of outlines as well as the beveled edge that may be drawn around the outside.

The part of a widget that's used for drawing is called the *canvas*. For **`PtWidget`**, this is the area inside the widget's borders. For **`PtBasic`** and its descendants, the canvas is the area inside the widget's border and *margins*. Other widgets, such as **`PtLabel`**, define other margins. The margins form a matt and obscure any part of the canvas extending beyond the cut-out area. This cut-out region is sometimes referred to as the *clipping area*.



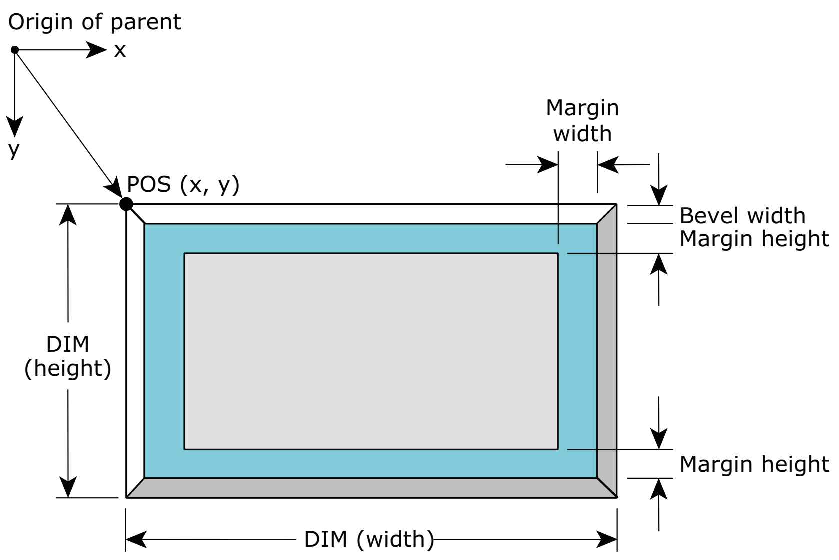
Anatomy of a **`PtBasic`** *widget.*

The canvas and margins are shown in different colors in the above diagram for the sake of clarity. In an actual widget, they're the same color.

For a widget, the border is optional. It's drawn only if the widget is highlighted (i.e. has Pt_HIGHLIGHTED set in its *Pt_ARG_FLAGS* resource). The border consists of various optional components, depending on the settings of the widget's *Pt_ARG_BASIC_FLAGS* resource. The components, from the outside in, are:

- a one-pixel etching line
- a one-pixel outline
- a bevel whose width is set by *Pt_ARG_BEVEL_WIDTH*
- a one-pixel inline.

A widget has several important attributes that define the geometry of these elements. The *dimension* of the widget, *Pt_ARG_DIM*, is the overall size of the widget, including its borders:

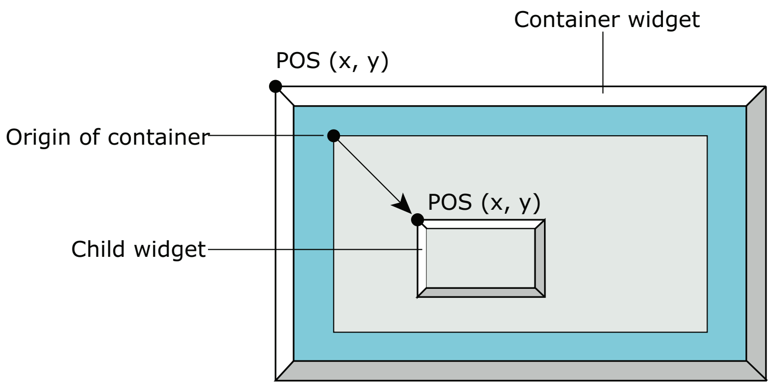


Widget position and dimensions.

Pt_ARG_MARGIN_WIDTH defines the width of the margins on the left and right of the canvas; *Pt_ARG_MARGIN_HEIGHT* defines the height of the margins above and below the canvas. These resources are defined by **`PtBasic`**.

Other widget classes define their own margin resources, which may be added to the basic margin width or height. For example, the label widget provides separate margins for the left, right, top, and bottom of the widget. These are added to the basic margin width and height to determine the amount of space to leave on each side of the canvas.

The origin of the widget (for the purposes of any drawing it performs or positioning of any children) is the *upper left corner* of the canvas. All coordinates specified for the widget are relative to this location, as are the coordinates of all events that the widget receives. For example, if the widget is a container, the positions of all the children are relative to this point:



Origin of a widget and the position of its children.

For positioning children, containers are concerned with only the outside edges of the widget's border. The position of the widget is maintained by the *Pt_ARG_POS* resource. This position is the point at the upper left corner of the *outside* of the widget's border. A container positions its children by adjusting this resource.

The position and dimensions of the widget can be accessed or modified simultaneously using the *Pt_ARG_AREA* resource provided by the widget.

The *extent* of a widget is a rectangle defined by the widget's position and dimensions. It isn't normally calculated until the widget is realized; you can force the widget to calculate its extent by calling *PtExtentWidget()*; to force a widget and its children to calculate their extents, call *PtExtentWidgetFamily()*. Once the extent is calculated, you can find out what it is by getting the *Pt_ARG_EXTENT* resource or by calling *PtWidgetExtent()*.

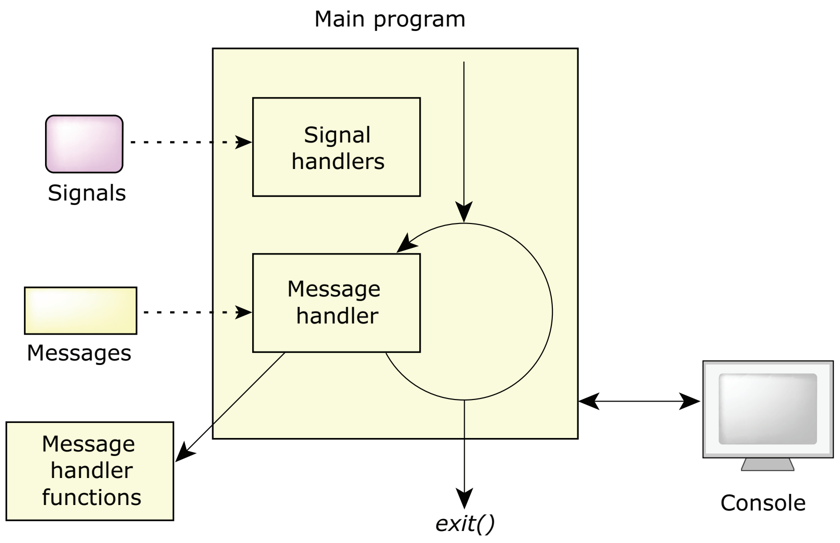
Programming paradigm

Let's compare how you write a text-mode application, a non-PhAB (Photon) application, and a PhAB application.

Text-mode application

When you write a non-Photon (text-mode) application, you generally concentrate on the main program, from which you do such things as:

- initialize the application
- set up signal handlers
- send and receive messages
- iterate
- call subroutines, as required
- communicate with the console
- and eventually exit.



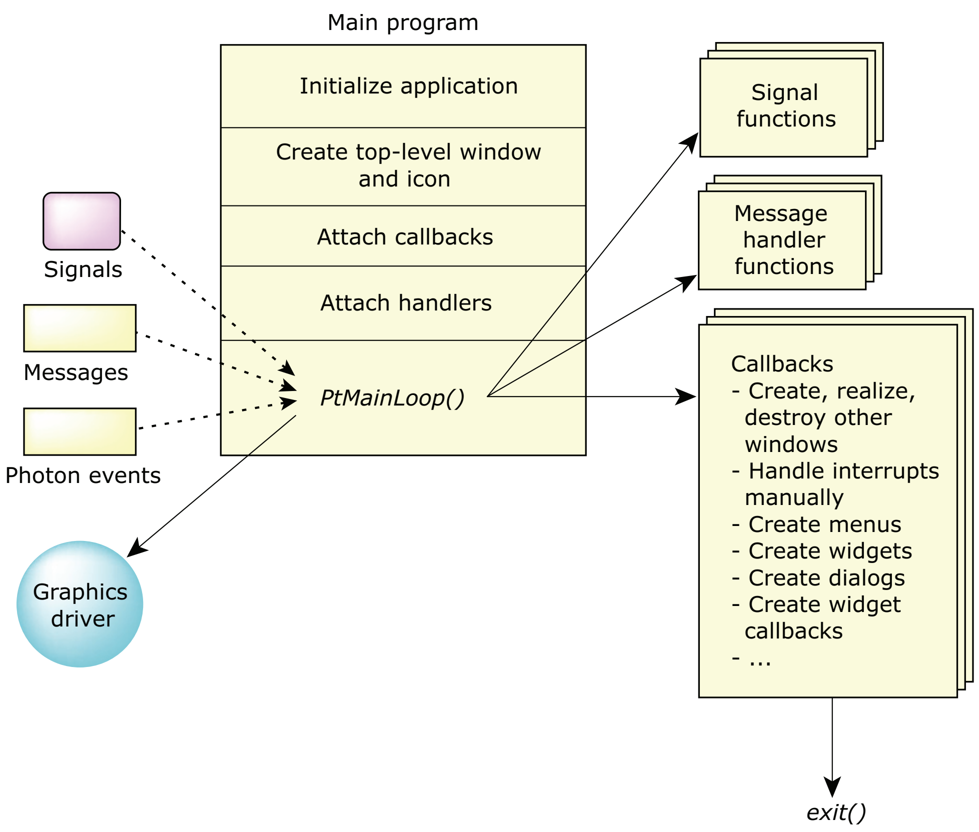
Structure of a text-mode application.

Non-PhAB application

A Photon application written without PhAB is similar to a text-mode application, except that you also:

- instantiate, initialize, and realize the application's widgets
- set the widget's resources, including those for:
  - size and position
  - anchoring
  - text
  - callback lists
  - etc.
- write callback routines to handle widget events. In these you may need to:
  - create windows and their widgets, set their resources, and realize them
  - create menus out of **`PtMenuButton`** widgets, set resources and callbacks, and realize the menus
  - destroy widgets
  - etc.
- call *PtMainLoop()* in your main program to handle events.

Usually one of your callbacks exits the application. Writing an application without PhAB means you'll be working directly with the widgets— a lot.



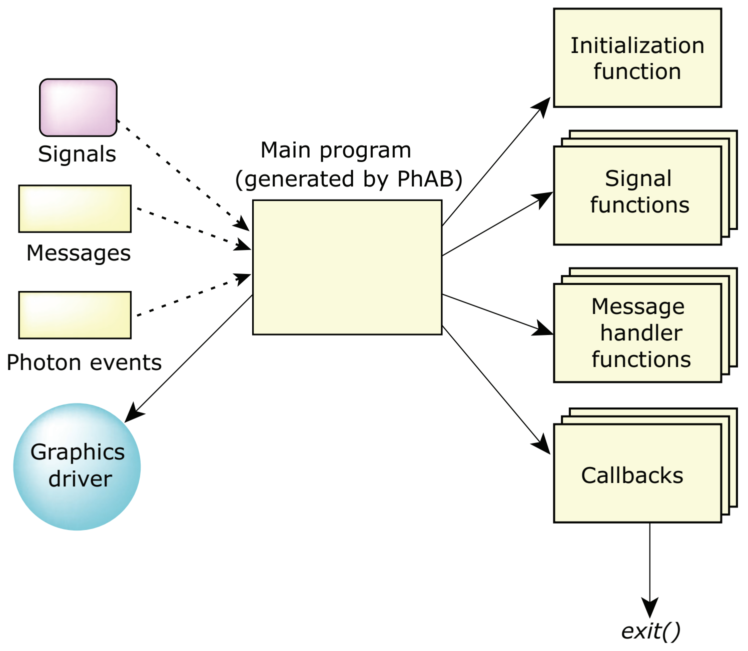
Structure of a Photon application written without PhAB.

PhAB application

When you develop a PhAB application, the main program is provided for you. Instead of worrying about the main program, you:

- provide a function that initializes the application
- set up signal handlers, which process the signals as they arrive and call signal-processing functions that you write
- set up input functions for messages
- write callbacks to handle events from the widgets.

The main program loops forever, processing events as they occur. Usually one of your callbacks ends the application. PhAB handles a lot of the details for you—you'll concentrate on your application's functionality, not the widgets'.



Structure of a Photon application written with PhAB.

In addition, you don't have to size and position widgets from your code; you do it visually in PhAB. PhAB also looks after instantiating, realizing, unrealizing, and destroying your widgets. PhAB even provides a menu module to make creating menus easy. You can see why we recommend using PhAB!

Photon libraries

API categories and libraries

The Photon application programming interface (API) is arranged into sets of functions, each distinguished by a two-character prefix:

`Al`	PhAB Translation functions that let you manipulate translation files (for PhAB applications or message databases) without using the translation editor. These routines aren't in the shared library; to use them, you'll need to link your application with the **`phexlib`** library.
`Ap`	PhAB functions that work with modules, widget databases, translation, and so on. These routines aren't in the shared library; to use them, you'll need to link your application with the **`Ap`** library.
`Pd`	Functions that manipulate the draw context.
`Pf`	Font services, including text metrics, and the generation of bitmaps of character strings. For more information, see the Fonts chapter.

`Pg`	Low-level graphics functions that access a rich set of primitives in the graphics drivers. These functions are used by the widget libraries and can also be called directly when using the **`PtRaw`** widget. See the Raw Drawing and Animation chapter or *Building Custom Widgets*.
`Ph`	Photon primitives that package up the draw requests and forward them to the Photon microkernel for steering and clipping until they arrive at the graphics driver ready to be rendered on screen. Although not commonly used by application programmers, these routines are heavily used by the graphics and widget libraries.
`Pi`	Image-manipulation functions. See "Manipulating images" in the Raw Drawing and Animation chapter.
`Pm`	Memory-context functions that can be used to reduce flickering. See "Animation" in the Raw Drawing and Animation chapter.
`Pp`	Printing functions that set up and control printing. See the Printing chapter.
`Pt`	Widget toolkit functions for creating, realizing, and destroying widgets, getting and setting resources, and so on. Besides using the widgets in the Photon widget library, you can use third-party widgets or your own custom widgets.
`Px`	Extended functions that deal with loading images, working with configuration files, and other useful routines. These routines aren't in the shared library; to use them, you'll need to link your application with the **`phexlib`** library.
`Rt`	Realtime timer functions. See "Timers" in the Working with Code chapter.
`utf8`	UTF-8 character string functions. See the appendix on Unicode multilingual support.
`wc`	Wide-character string functions. See the appendix on Unicode multilingual support.

The functions and data structures in these libraries are described in the Photon *Library Reference*.

The **`Pd`**, **`Pf`**, **`Pg`**, **`Ph`**, **`Pi`**, **`Pm`**, **`Pp`**, **`Pt`**, **`Rt`**, **`utf8`**, and **`wc`** routines are in Photon's main library, **`ph`**. Routines that are used to rasterize the Photon draw stream are in the **`phrender`** library. The **`ph`**, **`phrender`**, and **`Ap`** libraries are available in shared and static forms.

You may want to link your applications with the shared libraries; doing so makes your application much smaller. For more information, see "Choosing the libraries" in the Generating, Compiling, and Running Code chapter.

The **`Al`** and **`Px`** routines are included in the extended library, **`phexlib`**, which is available only in static form.

Versions and platforms

The Photon libraries currently support the following platforms:

- ARM little endian
- MIPS little endian
- PowerPC big endian
- SH-4 little endian
- x86 little endian

CAUTION:

The **`libphoton.so.1`** library is for applications created with version 1.14 of the Photon microGUI only. Don't combine this library with the current libraries or header files, or your application won't run properly.

The libraries in **`/usr/photon/lib`** are provided for runtime compatibility with Photon for QNX Neutrino 6.0 (x86 only). The current libraries are in **`/usr/lib`**.

If you need to determine the version number of the libraries, you can use:

- Ph_LIB_VERSION (defined in **`<PhT.h>`**) when you compile or run your application
- *PhLibVersion()* at runtime.

Both of these express the version number as:

```
major version * 100 + minor version
```

Building applications with PhAB—an overview

Step 1: Create modules

To construct an application UI in PhAB, you start with primary building blocks called *modules*. Modules look and work a lot like the windows you see in most Photon applications.

You could design a UI with just one module. But for most applications, you'll probably use several modules and assign each a different role. As a rule, each module groups together related information and lets the user interact with that information in a specific way. To help you handle the requirements of virtually any application, PhAB provides several module types:

- *window*—normally used for the application's major activities. A typical application has one main window that opens when the application starts up.
- *dialog*—allows the application to exchange information with the user

- *menu*—presents commands to the user
- *picture*—can be used in different ways. For example, you can use a picture to provide a convenient database of widgets or to change the contents of an existing module

For more information, see the Working with Modules chapter.

Step 2: Add widgets

Once you've created a module, you're ready to place widgets into it. To add a widget, just click on the appropriate icon in PhAB's widget palette, then click where you'd like the widget to go. PhAB lets you add any widget that comes with the Photon development environment. You can choose from widgets that:

- *display or edit values* (e.g. labels, text, and multiline text)
- *present choices* (e.g. lists, comboboxes, and groups)
- *display graphics* (e.g. bitmaps, images, lines, rectangles, ellipses, and polygons)
- *display scrolling areas* (e.g. scrollbars and scrolling containers)
- *initiate actions* (e.g. buttons that contain text or images)

To customize how a widget looks and works, you set its attributes or *resources*. PhAB's Control panels and Resource editors make it easy to do this. Just click on the resource you want to change, then select or enter a new value.

You can even customize a widget and then save it as a template to use to create similar widgets.

For more information, see the Editing Resources and Callbacks in PhAB chapter.

Step 3: Attach callbacks

You've created your modules and placed widgets into them. Now you're ready to define how the application works. To do this, you use *callbacks*.

Every Photon widget supports several callback types. To attach code functions to a callback, you set a resource or use a provided convenience function. The widget invokes the code function whenever the callback's conditions are met.

With PhAB, you're free to concentrate on writing application-specific code in your callbacks—you don't have to create code to "glue" interface components together. That's because PhAB provides *link callbacks*. Using link callbacks, you can attach a widget's callback resource directly to windows, dialogs, menus, and many other things besides application code.

Link callbacks also let you add functionality that isn't available when you attach callbacks "by hand." For example, if you link a dialog to a button widget, you can specify where the dialog is to appear. You can also specify a setup function that's automatically called before the dialog is realized, after the dialog is realized, or both.

The extended functionality provided by link callbacks makes it much easier to design a user interface. In fact, you can prototype an entire application *without having to write any code*.

For more information, see the Editing Resources and Callbacks in PhAB chapter.

Step 4: Generate code

You've created your application's modules and created the link callbacks to glue the various components together. Now you're ready to generate and compile code to turn your application design into a working executable.

The way you generate the code depends on whether you're using PhAB standalone, or through the IDE, and is described in the Generating, Compiling, and Running Code chapter.

When using PhAB from the IDE, you generate the application's user interface in PhAB, and then build the project from the IDE. In PhAB, select **Build→Generate UI**. If the application is new and no targets are added, the **Select New Platform** dialog is displayed and you're asked to add a target for your application. Then you switch to the IDE, and build the application. See Building projects in the Developing C/C++ Programs chapter of the IDE *User's Guide* for more information.

Using standalone PhAB, you use the **Build** menu. Select **Build→Build**. If the application is new and no targets are added, the **Select New Platform** dialog is displayed and you are asked to add a target for your application. Your application is then generated and built. A file manager is included in one of PhAB's palettes (under **Window→Show Project**) so you can edit the source code and manipulate files — without having to leave the PhAB environment.

For more information, see the Generating, Compiling, and Running Code chapter.

Step 5: Run your application

After you've generated, compiled, and linked your application, you can execute it. Again, how you do this depends on whether you're running PhAB standalone or from the IDE.

If you're running PhAB from the IDE, follow the instructions in Running projects in the Developing C/C++ Programs chapter of the IDE *User's Guide*.

If you're using PhAB standalone, run the application from the Build & Run dialog. Using this same dialog, you can even launch your application under a debugger for seamless debugging.

For more information, see the Generating, Compiling, and Running Code chapter.

Step 6: Repeat any previous step

After you've generated and compiled your application, you're free to change the interface, attach callbacks, and regenerate the code as often as you like.

Writing applications without PhAB

We recommend that you use PhAB to develop your application. However, even if you don't plan to use PhAB, you should read through this guide completely (especially the Programming Photon without PhAB chapter) in order to familiarize yourself with all the Photon fundamentals before you can start creating applications. You should then refer to the *Widget Reference*.

Chapter 2
Tutorials

In this chapter. . .

The best way to get to know PhAB is to use it. This chapter provides hands-on sessions to give you a jump start on creating applications. We'll take a closer look at using PhAB in the chapters that follow.

The first two tutorials cover the basics: creating widgets, changing how widgets look and behave, generating code, running your application, and so on.

The remaining tutorials go beyond the basics to show you how to create working menus, dialogs, and windows. When you've completed these tutorials, you'll be ready to start building almost any Photon application.

Before you start...

If you're developing Photon applications in the IDE, the way you use PhAB there is slightly different than from standalone PhAB. The differences are:

- New projects — When using the IDE, you create a QNX Photon Appbuilder project in the IDE, and then use PhAB to create the user interface. Using standalone PhAB, you create the project from within PhAB.
- Editing code — The IDE allows you to edit your project's code, and take advantage of features like syntax highlighting. When you use standalone PhAB, you use an external editor, such as **`vi`**, to edit code.
- Building and compiling — The IDE manages building and running your application, and you have to set up a target to run and debug the application on. Using standalone PhAB, you can build and run your application from within PhAB. Note that in both cases you need to configure targets within PhAB.

Creating a Photon project and starting PhAB

From the IDE:

To create a new PhAB project, see "Creating a QNX Photon Appbuilder project" in the Developing Photon Applications chapter of the *IDE User's Guide.* When you create a new project, the IDE opens PhAB, and you see the **New Window Style** dialog from which you can select the type of base window for your application.

From standalone PhAB:

You can start PhAB from the Launch menu in the lower-left corner of the screen; choose the Development submenu, and then choose Builder.

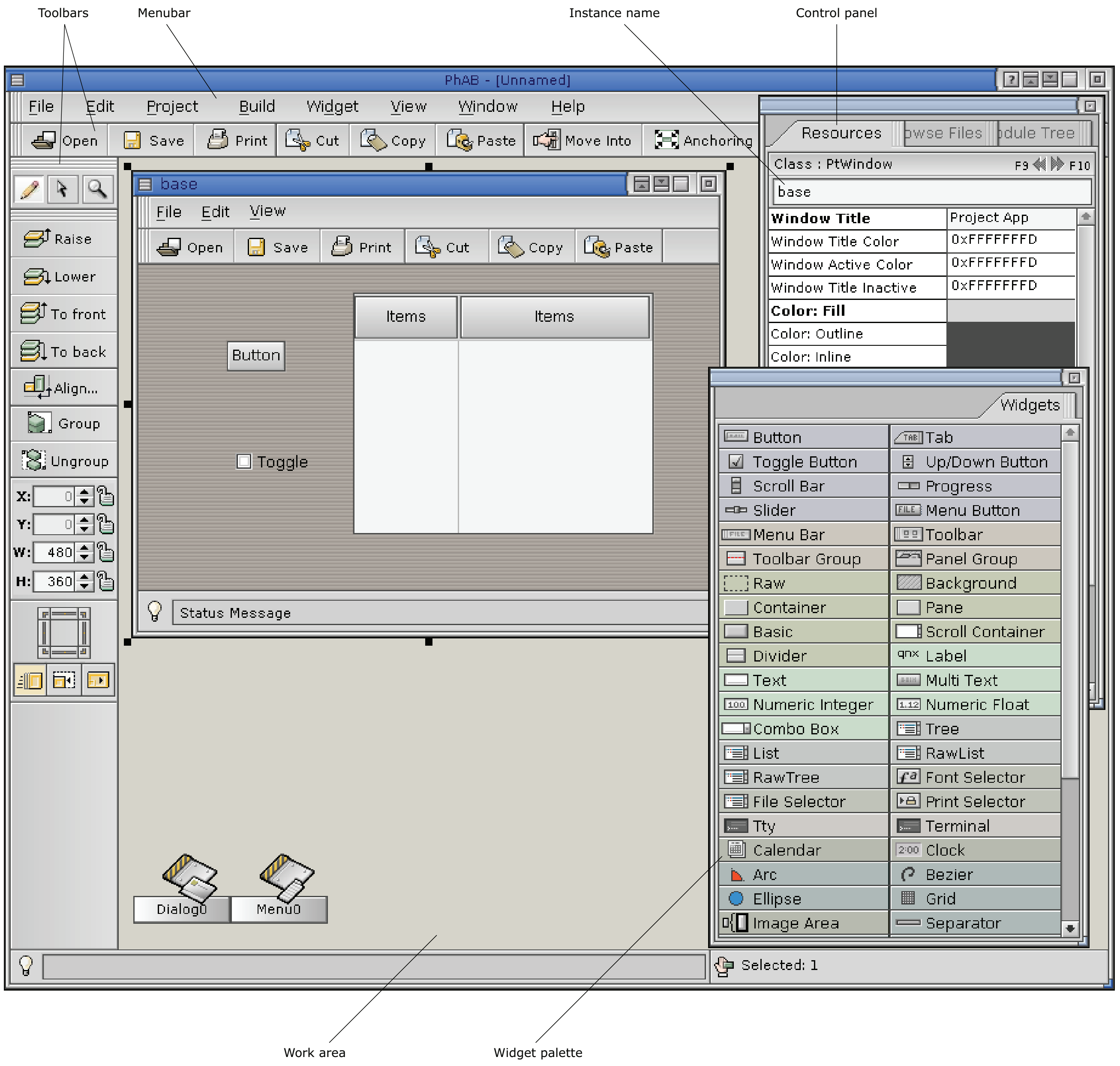
You can also start PhAB from a **`pterm`** window by typing:

```
appbuilder
```

For information about command-line options, see **`appbuilder`** in the QNX Neutrino *Utilities Reference*.

PhAB's Interface

Before you start the tutorials, take a moment to make yourself familiar with PhAB's user interface:



Overview of PhAB's user interface.

Menubar	Import graphics, create windows and dialogs, generate C and/or C++ code to implement your entire user interface, and more.
Toolbars	Save time with the toolbars—with a couple of mouse clicks you can duplicate, move, align, group, or resize any number of widgets.

Work area	Provides a flexible area where you can work on several application modules all at once.
Widget palette	Makes it easy to add widgets to your application. Just click the widget you want, then click where you want it.
Control panels	Let you fully customize your application's widgets. You can choose text fonts, modify colors, customize bitmaps, and attach callbacks that will pop up dialogs or invoke C and/or C++ code you've supplied.

The widget palette and control panels are initially in the same window, but you can drag any of them into a different window. To switch between panels in a window, click the tab at the top and choose a panel from the menu.

If you close a control panel, you can redisplay it by selecting the appropriate item from the View menu.

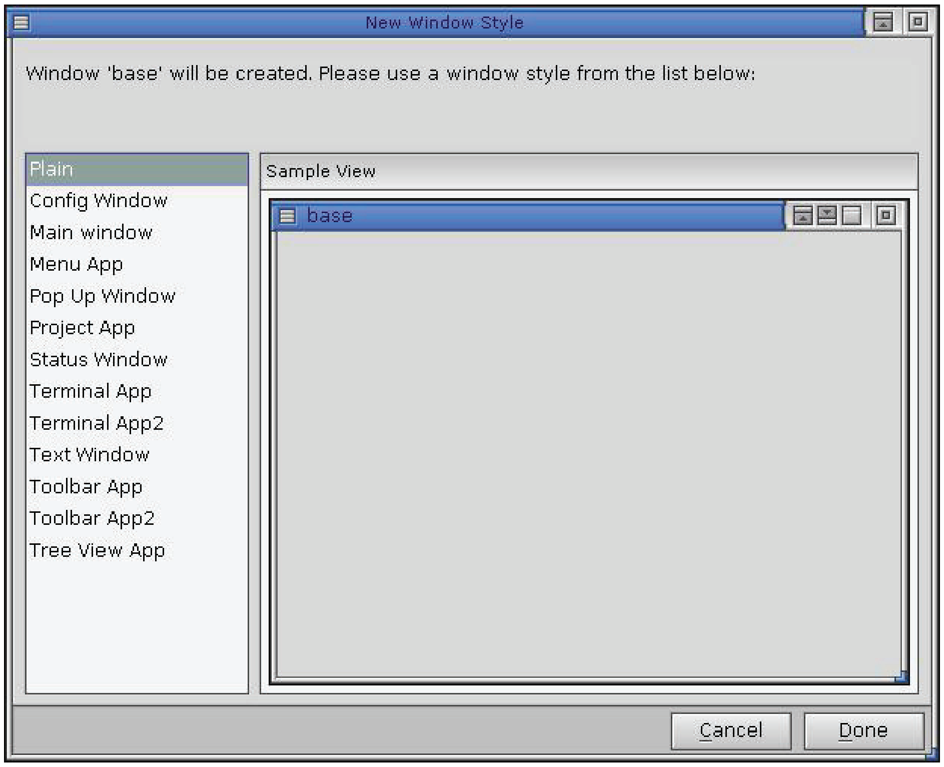
Tutorial 1 — Hello, world

In this tutorial you learn how to use PhAB to create and compile a simple application.

Creating the application

1 Create a new project. See "Creating a Photon project and starting PhAB" above.

2 PhAB displays a dialog to let you choose the style for the new application's default base window:



3 Choose a style and click Done; PhAB creates the base window and displays it.

4 Whenever you create a new application within standalone PhAB, it's a good idea to save the application and give it a name. (If you're running PhAB from the IDE, you've already saved the application when you created the project).

From the File menu, choose Save As to open the Application Selector dialog. Click the Application Name field, type **`tut1`**, then press Enter or click Save Application.

5 Look at PhAB's titlebar. It now indicates that the current application is named **`tut1`**.

6 If the widget palette isn't displayed, click the tab at the top of the current control panel and choose Widgets from the menu that appears.

7 Drag the widget palette away from the other control panels by pointing to the left of its tab, holding down the mouse button, and pointing to PhAB's work area.

8 If you wish, resize the widget palette and control panels.

9 Go to the widget palette and click the **`PtLabel`** widget icon:

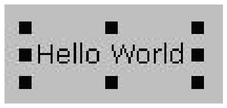
10 Move the pointer into the application's base window (the pointer changes to a crosshair) and click anywhere near the center of the window.

PhAB automatically:

- creates a new **`PtLabel`** widget
- selects the widget so you can edit its resources
- places resize handles around the widget
- displays the widget's resources in the Resources and Callbacks control panels.

11 Go to the Resources control panel and highlight the text **`Label`** beside the **Label Text** resource.

12 Change the text to **`Hello World`**. As you type, the text in the widget changes:



Generating code

Now you're ready to generate, compile, and execute the application. How you perform this step depends on whether you're using PhAB from the IDE or standalone.

From the IDE

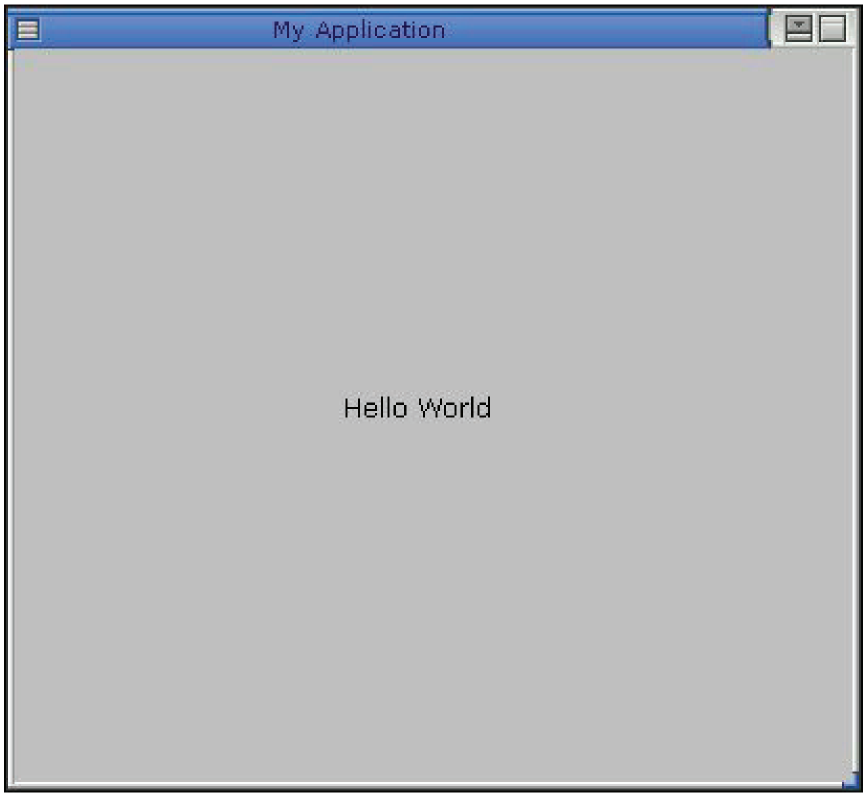
When running PhAB from the IDE, PhAB generates the code for your application's user interface, and the IDE compiles and executes the application. You'll notice that only the **Generate UI** item is available in PhAB's **Build** menu.

1. In PhAB, chose **Generate UI** from the **Build** menu to generate the user interface code.
2. Switch to the IDE.
3. To build the application, follow the instructions in Building projects in the Developing C/C++ Programs chapter of the *IDE User's Guide*.
4. To run the application, follow the instructions in Running projects in the Developing C/C++ Programs chapter of the *IDE User's Guide*.

From standalone PhAB

1. From the **Build** menu, choose **Build & Run**. PhAB displays a dialog for selecting a platform, which is a combination of the operating system, computer, compiler, and endian format. Choose the appropriate platform for your application. For example, if you're using the Neutrino OS on an Intel x86 machine and the `gcc` compiler, choose X86 (Little Endian).
2. Click **Done** once you've made your platform selection. Your application will be generated, compiled and linked. PhAB displays a dialog for entering run arguments. Click **OK**. Your application runs.

The application will appear in its own window, with the text "Hello World" in the center and the default title "My Application" in the title bar:



Congratulations! You've just created your first Photon application using PhAB.

To quit the application, click the window menu button in its top-left corner, then choose the Close item.

Want more info?

For more info on compiling, running, and debugging an application, see the Generating, Compiling, and Running Code chapter.

Tutorial 2 — editing resources

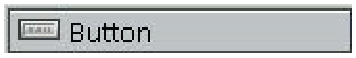
This tutorial introduces you to PhAB's resource editors, which let you change how widgets look and behave. You'll find out how to edit virtually any kind of resource a widget may have, including:

- numerical resources (e.g. border width)
- text fonts
- text strings
- flags
- colors
- pixmaps

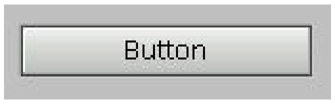
You'll also learn how to create a template so you can create other instances of an existing widget.

Adding a button widget

1. Create a new application called **`tut2`**. Choose the Plain window style.
2. Click **`PtButton`** in the widget palette:



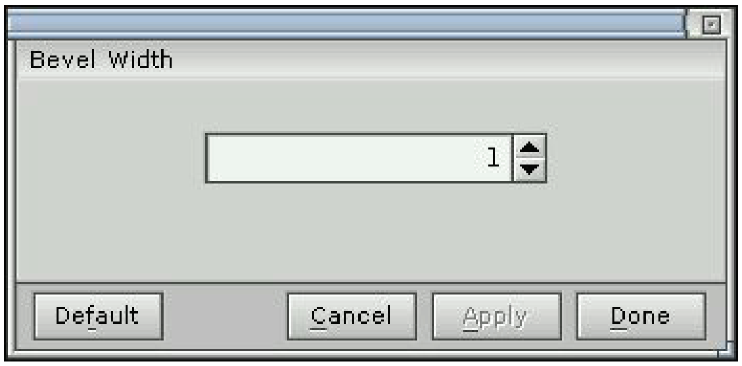
3. Click near the center of the application's window. You'll see a button widget.
4. Drag any of the button's resize handles until the button matches the following picture:



Changing the bevel width

Let's now edit a numerical resource—the button's bevel width.

1. Click the **Bevel Width** resource in the Control Panel. You'll see the number editor:



   This editor lets you change the value of any numerical widget resource.

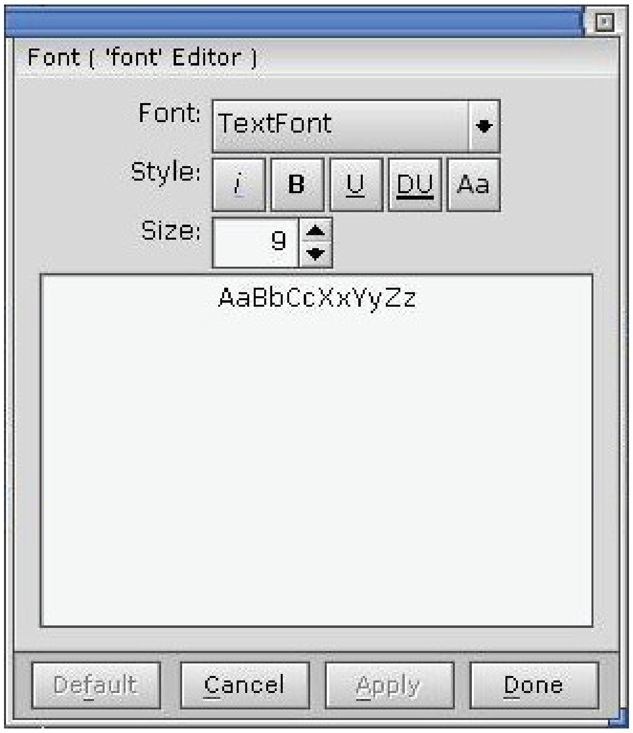
2. Change the value to 6. To do this, you can:
   - type in the new value
     or:
   - click the increment/decrement buttons.
3. To apply the new value and close the editor, press Enter or click **Done**.

You can also edit this resource (and most resources) right in the Resources control panel. Choose whichever method you like.

Changing the font

Let’s change the font of the button’s text:

1. Click the **Font** resource. You’ll see the font editor, which displays the button’s current font:



This editor lets you change the text font of any widget that has text.

2. Click the Font box or the Size box, select a typeface or size from the displayed list, and click Apply. The button displays the new font.

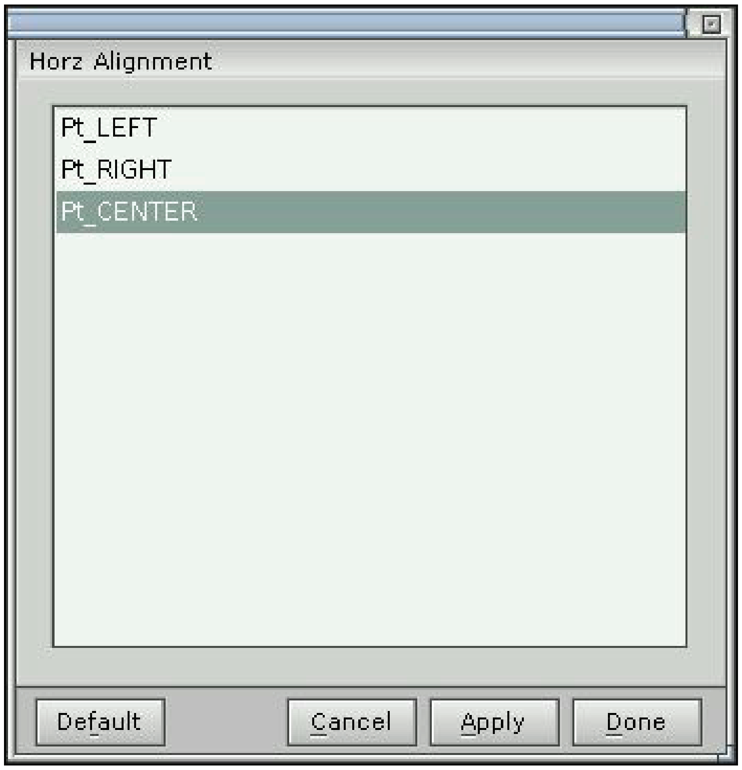
3. Click Default. The editor displays the widget’s default font, but doesn’t apply the font to the widget.

4. If you want to keep the new font that you selected, click Cancel to ignore the default. If you want to apply the default, click Done. Either way, the editor closes.

Changing the text alignment

Now let’s change the button’s horizontal text alignment:

1. Scroll through the Resources control panel to find the **Horz Alignment** resource, then click it. You'll see the flag/option editor, which displays the widget's current text alignment:



This editor serves a dual purpose in PhAB:

- It modifies any resource—such as text alignment—that can have one of several preset values.
- It selects one or more flags in any flag resource

2. Click Pt_LEFT or Pt_RIGHT, then click Apply. You'll see the button text move to the left or right edge of the button.

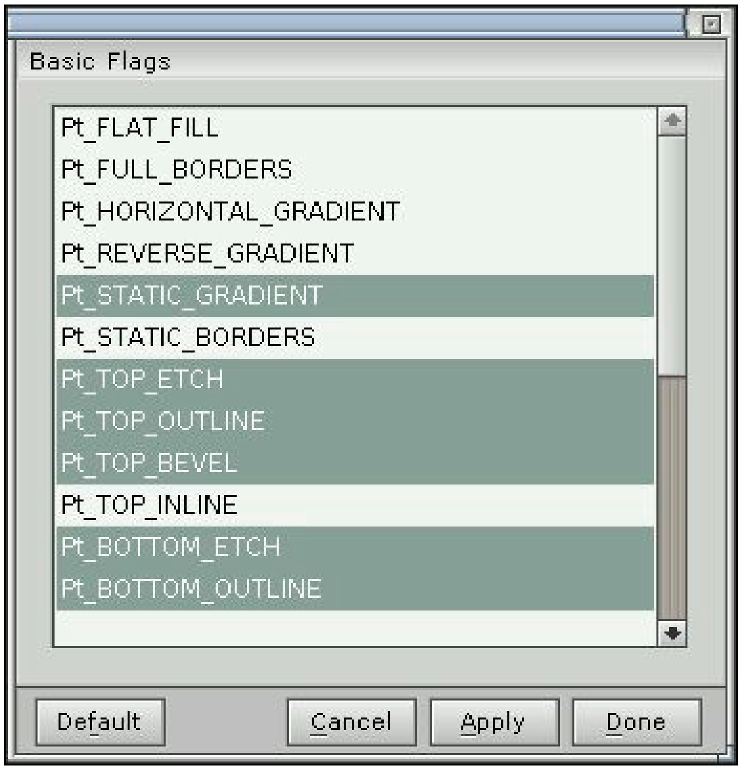
3. Click Done.

You can also set this resource right in the Resources control panel.

Setting flags

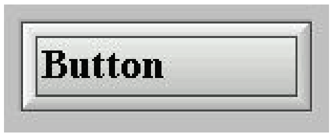
Let's now use the flag/option editor to set one of the widget's flags:

1. Scroll through the Resources control panel to find the **Basic Flags** resource, then click it. The flag/option editor reopens, but this time it shows the widget's current `PtBasic` flag settings:



The bits in this flag resource aren't mutually exclusive, so this time you can use the editor to select multiple options, if desired.

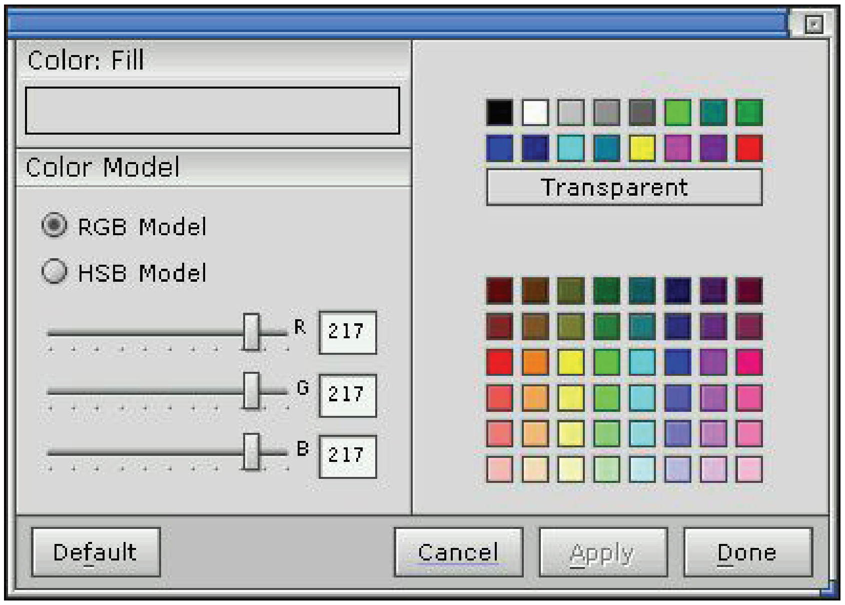
2 Set the Pt_TOP_INLINE, Pt_BOTTOM_INLINE, Pt_LEFT_INLINE, and Pt_RIGHT_INLINE flags, then click Done. PhAB draws the button with an inner border:



Changing the fill color

Let's change a color resource—the button's fill color.

1 Click the button's **Color: Fill** resource. You'll see the color editor, which displays the current fill color:



This editor lets you edit any color resource. It provides several preset base colors, which should work well with all graphic drivers, and 48 customizable colors for drivers that support 256 or more colors.

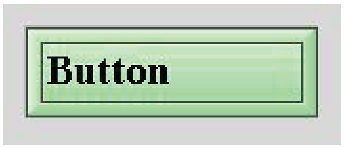
2 Click any color in the Base Colors set, then click on Apply. The button is filled with the color you selected.

3 Select a color from the Custom Colors set. The sliders will display the color's Red/Green/Blue (RGB) values. Change these values till you get a color you want, then apply your changes.

If you'd like to experiment with the Hue/Saturation/Brightness (HSB) color model, click the HSB Model button.

4 Click Done when you've finished experimenting with the editor.

Your button should now look something like this:



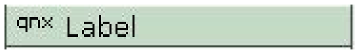
Don't delete this widget; we'll use it to create a template later on, so that you can create other widgets like it.

Editing a pixmap

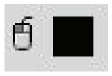
Let's now use the pixmap editor to edit a **`PtLabel`** widget. This editor is called "pixmap" instead of "bitmap" since it lets you edit many types of image resources besides bitmaps.

A **`PtLabel`** widget display text and/or an image.

1. Click **PtLabel** in the widget palette:

2. Move the pointer into the main window and click below the button widget you created. You'll see a **PtLabel** widget.
3. Click the Label Type resource in the Resources control panel, and set it to Pt_IMAGE.
4. Click the Label Image resource in the Resources control panel to bring up the pixmap editor.
5. Next, bring up the color editor to select a draw color. Just click the following button:

6. Select a color from the pixmap palette. You'll see that the draw color in the pixmap editor changes immediately.

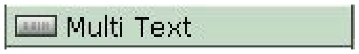
   If you click Edit Color, you'll see the Color Editor, as described earlier.
7. To draw a simple image, you can:
   - click the left mouse button to fill a cell with the draw color
   - click the right mouse button to fill a cell with the background color
   - hold down a mouse button and drag the pointer to draw freehand

   Feel free to try the other drawing tools.
8. When you're done, click the pixmap editor's Done button to apply your changes and close the editor.

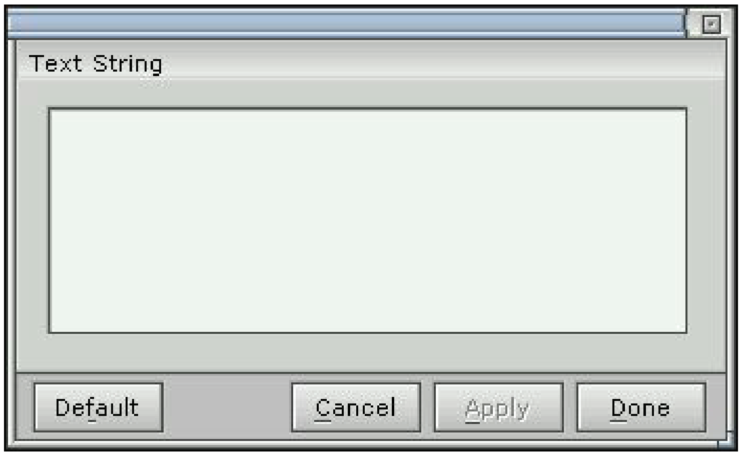
Editing multiline text

Next, we'll edit a multiline text resource—the text of a **PtMultiText** widget.

1. Click **PtMultiText** in the widget palette:

2. Move the pointer below the label widget you've just created, and drag until the new **PtMultiText** widget appears big enough to hold a few lines of text.
3. Click the **Text String** resource in the Resources control panel to bring up the multiline text editor:



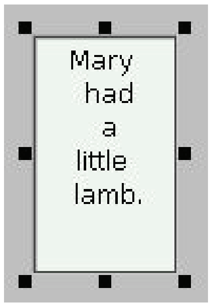
4 Type a few lines of text. To create a new line, press Enter. For example:

`Mary had`Enter
`a`Enter
`little lamb.`Enter

5 Click Done. Your text should appear exactly as you typed it. If it doesn't, try resizing the widget—the widget might not be wide enough or tall enough.

6 For a different effect, look for the **Horz Alignment** resource, click the arrow, and change the text alignment to Pt_CENTER. As you can see, each line is now centered individually.

7 If you haven't already, resize the widget by dragging on one of its resize handles. You'll see the text update automatically to adjust to the new size. For example:



You can edit the text right in the control panel, but it displays only the current line of text.

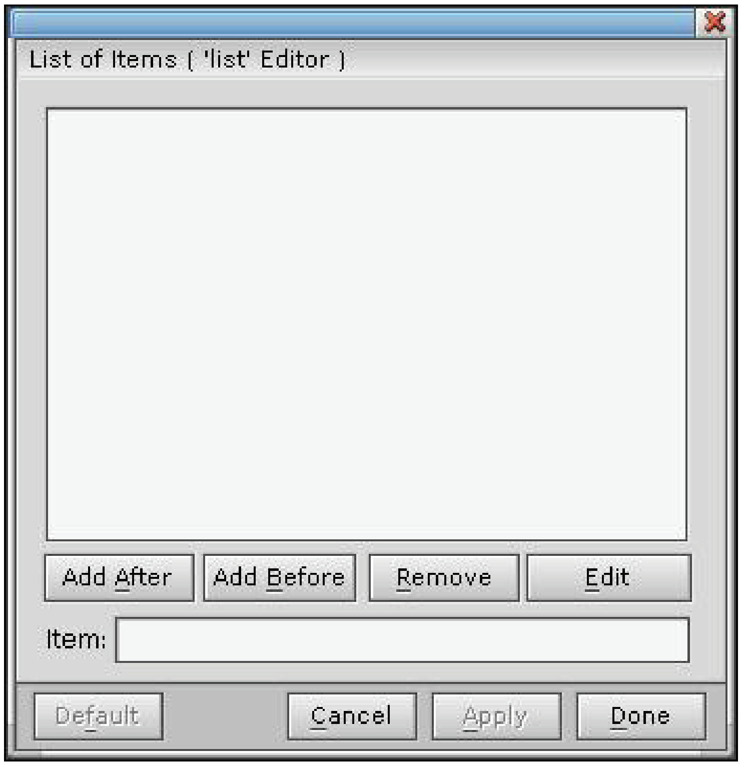
Editing a list of text items

Let's now create a **`PtList`** widget and add text to the widget using the list editor. This editor lets you add and edit text for any widget that provides a list of text items.

1 Click **`PtList`** in the widget palette:

2 Move the pointer into the application's base window, and drag the pointer until the new **`PtList`** widget appears big enough to hold a few lines of text.

3 Click the **List of Items** resource to bring up the list editor:



4 Click the text box at the bottom of the editor. You'll see the text-input cursor.

5 Type some text, then click Add After to place the first item in the list.

6 Now let's create the second item. Click in the text box, and type Ctrl-U to erase the text in the text box, then type some new text.

Click Add After to place this new item after the previous item.

7 Repeat the above step as often as you'd like.

8 Click Apply. The **`PtList`** widget should now display the list you've created.

9 Now try editing the list:

- To modify an existing item, click the item, edit the item's text, then click Edit.
- To delete an item, click the item, then click Remove.

10 When you're finished experimenting, click on Done to apply your changes and close the editor.

Creating a template

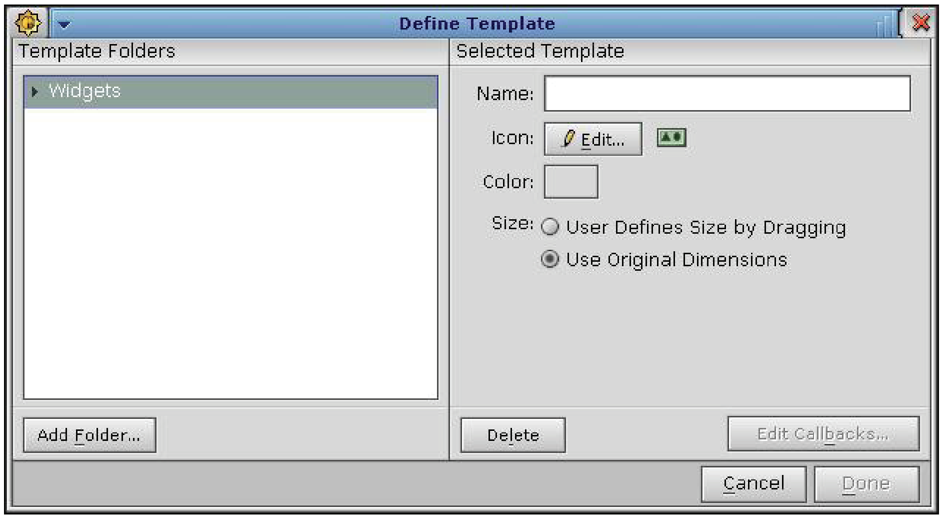
At times, you might want to create many widgets that look and behave alike. You can do this by creating a widget, editing its resources, and then copying and pasting it, but this isn't always very convenient, and doesn't copy certain important elements like callbacks.

PhAB makes it simpler by letting you create a *template* from an existing widget or widgets. PhAB creates a palette, similar to the widget palette, for your templates.

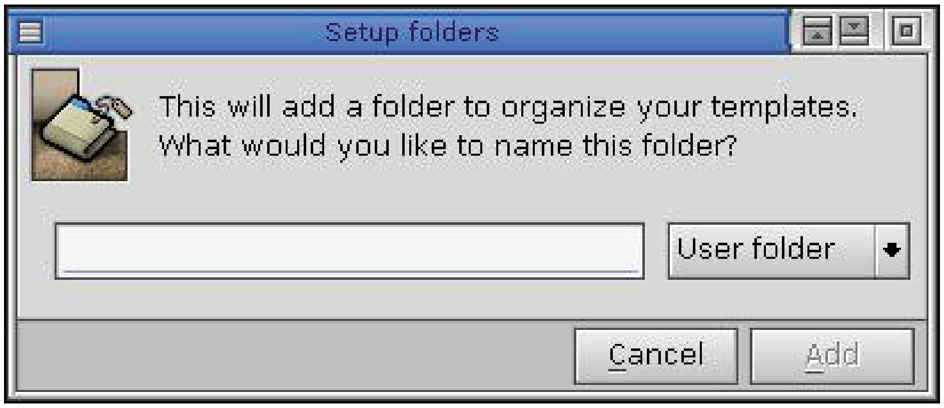
Let's create a template from the button that you created earlier in this tutorial.

1 Start by selecting the button.

2 Click the Widget menu, and then choose Define Template. The Define Template dialog appears.



3 You need to create a folder in which to store the template, so click on Add Folder. This dialog is displayed:

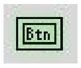

This will add a folder to organize your templates. What would you like to name this folder?

4 The new folder can be a user folder or a PhAB folder. A user folder is personal and can't be viewed by any other PhAB users logged on to the system. If you choose **PhAB folder**, the new folder can be shared between users; you must have the necessary permissions to create a PhAB folder.

Choose **User Folder**, type `My_templates` as the folder's name, and click **Add**. The dialog closes, and the folder's name is displayed in the **Define template** dialog.

5 Give the template a name, such as `Big green button`. This is the name that PhAB uses in the palette.

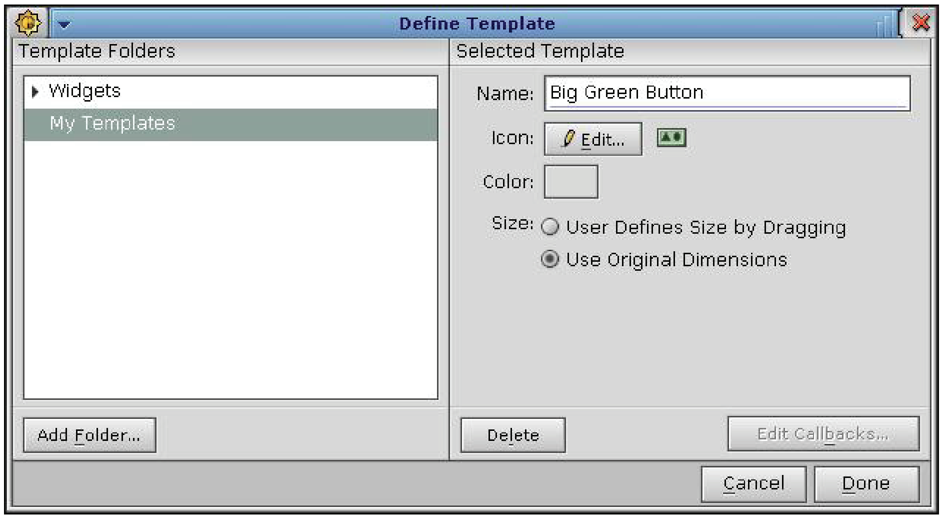
6 You can create an icon for the palette entry for the template. If you do not create an icon for the template entry, a default icon is used for it. To create the icon, click the icon **Edit** button, and then follow the instructions given earlier for editing pixmaps. You should make the icon look something like the widget:



7 Optionally, choose the background color for the palette entry by clicking on the Color box. You can use different background colors in a palette to distinguish widgets that are used for different purposes (e.g. buttons and text widgets).

8 Choose a resize method. This determines whether you drag or just click when you create instances of your template. For this button, choose the dragging method.

9 The dialog should now look something like this:



Selected Template

Name: Big Green Button

Icon: Edit...

Color:

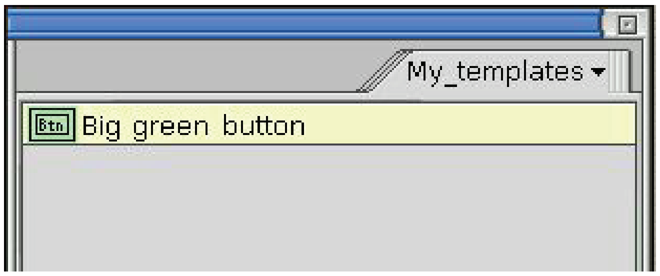
Size: User Defines Size by Dragging [✓ Use Original Dimensions]

Click Done.

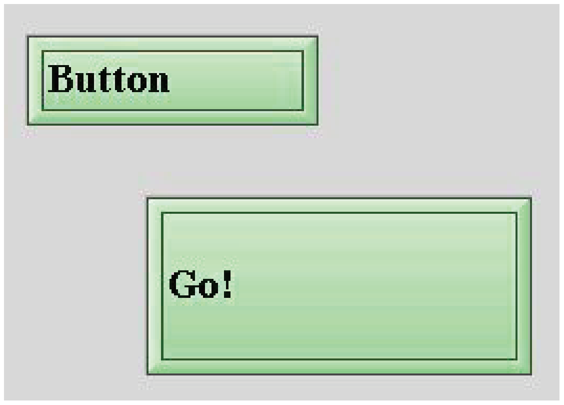
You've just created a template! Now, let's see how to use it.

1 Select **Window→Show Templates**. On the list of items, select **Show My Templates**. If the menu item is **Hide My Templates**, it means that My Templates is already displayed and visible on the screen.

2 Go to the control panels, and click the top tab. The popup menu now includes My_templates; choose it to display the palette.



3 Click the icon for your customized button, create an instance of it, and edit it as you wish:



If you wish, you can save, generate, make, and run the application.

Whenever you start PhAB, it automatically loads the palette for My_templates.

Want more info?

You now know the basics of editing any widget resource in PhAB. For more information, see the following sections in the Editing Resources and Callbacks in PhAB chapter:

To edit:	See this section:
Bitmaps or images	Pixmap editor
Colors	Color editor
Flags	Flag/option editor
Fonts	Font editor
Lists of text items	List editor
Numbers	Number editor or Flag/option editor
Single-line or multiline text strings	Text editors

For more information on templates, see “Templates” in the Creating Widgets in PhAB chapter.

Tutorial 3 — creating menus and menubars

This tutorial takes you through the steps required to create menus and menubars.

About link callbacks

In this tutorial, you'll learn how to set up a *link callback*, one of the key components of PhAB. To understand what a link callback is, let's start with some background info on widget callbacks.

Almost all widgets support a variety of callbacks. These callbacks enable your application's interface to interact with your application code. For example, let's say you want your application to perform an action when the user clicks on a button. In that case, you would attach a callback function to the button's "Activate" callback.

In some windowing environments, you can attach only code functions to widget callbacks. But whenever you use PhAB to create a callback, you can go one step further and attach entire windows, dialogs, menus, and much more. It's this extended functionality that we call a link callback.

PhAB provides two basic types of link callbacks:

Module-type link callback

Attaches an application module (such as a window, dialog, or menu) to any widget callback. The module opens whenever the callback's conditions are met. In this tutorial, you'll link a menu module to a button's "Arm" callback.

Code-type link callback

Attaches a code function to any widget callback. The widget invokes the function whenever the callback's conditions are met. Note that some code-type link callbacks let you close the parent module automatically. In this tutorial, you'll link a code function to a menu item's callback.

About instance names

To access a widget from your application code, you must first give the widget an *instance name*. Since all widget instance names reside in the same global namespace, no two widgets within an application can have the same instance name.

We recommend that you start every instance name with a module prefix. For example, if your base window has a **`PtButton`** widget that contains the label text "Blue," you could give this widget an instance name of **`base_blue`**.

Adopting a naming convention for your widgets will make it easier for you to work with large applications.

Creating a menubar

To learn about using link callbacks, let's create two functioning menus—File and Help—that you can later incorporate into your own applications.

In PhAB, menus are built in two pieces:

- a menu button, which you'll click to display the menu
- a menu module, which contains the menu items.

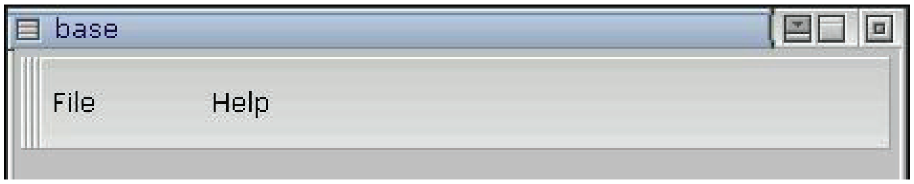
Using link callbacks, you'll link the menu modules to the File and Help buttons in a menubar. You'll also link a code-type callback to the Quit menu item in the File menu module. This callback will enable the Quit item to close the application.

1. Create a new application named **`tut3`**. Choose the Plain window style.
2. Select the **`PtMenuBar`** widget from the widget palette, point at the top left cornet of the main window's canvas, and drag until the menu bar is the width of the window.

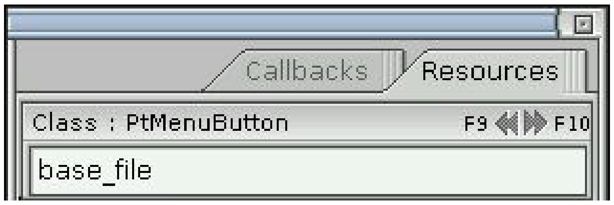
   The menubar grows and shrinks as you change the width of the window, and it always stays at the top of the window. You can see this by clicking in the titlebar of the window, then resizing the window by dragging on one of its resize handles.

If you accidentally click the Test button, the window won't resize or accept new widgets. If this happens, you just switched into Test Mode. To go back to Edit Mode, select **Project→Edit mode**.

By the time you're finished the following steps, the menubar will look like this:



3. Place a **`PtMenuButton`** widget in the menubar you just created. The menu button is automatically centered vertically in the menubar.
4. Go to the Resources control panel and click the widget instance name just below the class name. Change the button's instance name to **`base_file`**:



Class : PtMenuButton

base_file

5. Change the button's Label Text resource to **`File`**.

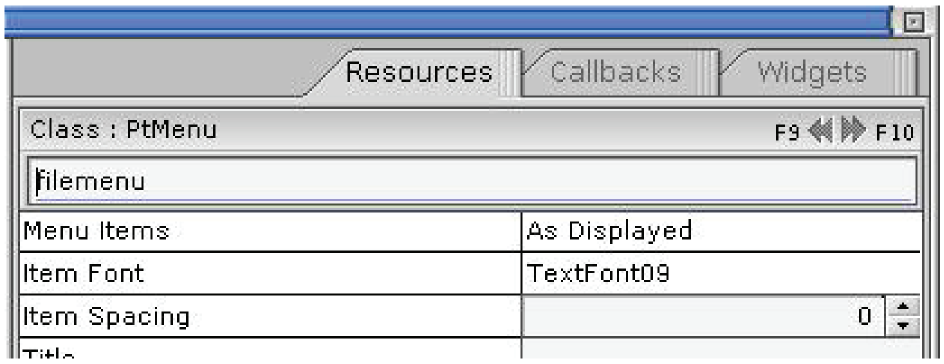
6 Place another **`PtMenuButton`** widget next to the first. Change its instance name to **`base_help`** and its text to **`Help`**.

Creating the File menu module

Now that you have menu buttons, you need to create your menu modules. Let's start with the File menu.

1 Select **Project**→**Add Menu**. A new menu module appears.

2 Change the name of the menu from **`Menu0`** to **`filemenu`**:



Class : PtMenu

filemenu

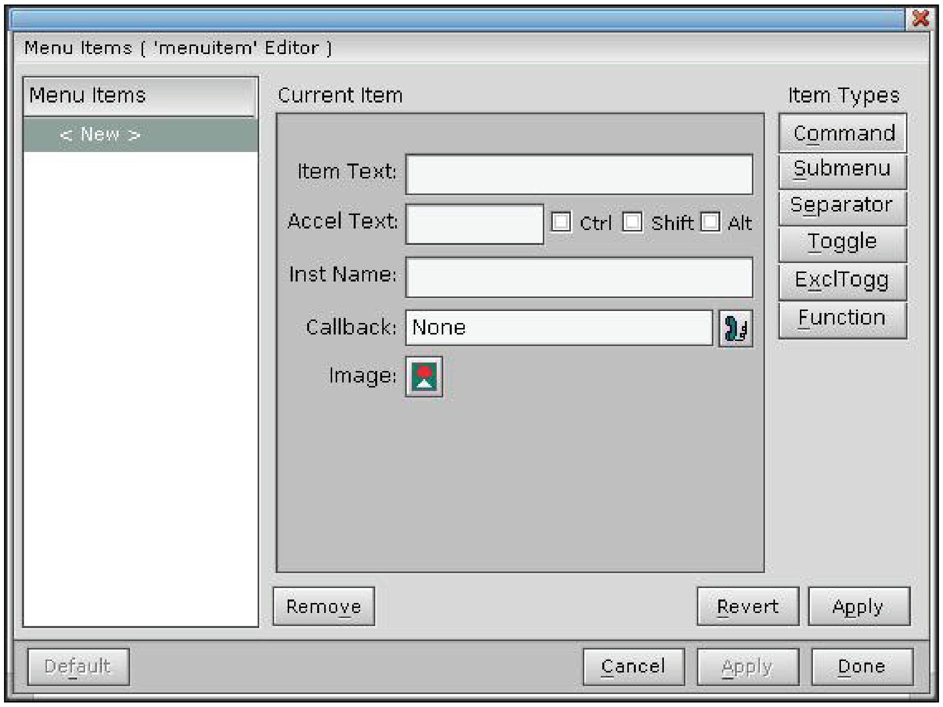
Menu Items	As Displayed
Item Font	TextFont09
Item Spacing	0
Title	

Adding menu items

Let's now add some menu items to the File menu.

If you click another module, the menu module becomes deselected, which means you can't work on it. To reselect the menu module, click its titlebar.

1 Click the Menu Items resource in the Resources control panel. You'll see the menu editor:



If you look at the Menu Items list, you'll see that the **<New>** item is selected. This special item lets you add menu items to the menu.

2 To add your first menu item—which also happens to be called "New"—click the Item Text field, then type **`New`**.

3 Now give the item an instance name. In the Inst Name field, type **`file_new`**.

4 Click Apply to add the item to the menu. You'll see the item's name in Menu Items list, prefixed by **`CMD`**. The **`CMD`** means this is a Command item; that is, an item that invokes a PhAB callback.

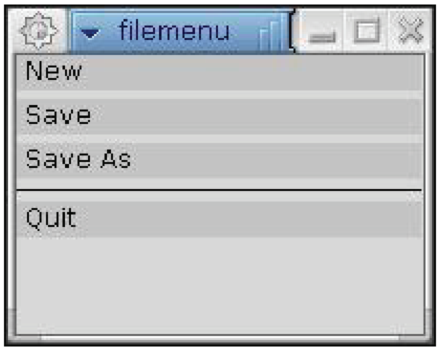
5 Repeat the above steps to create the two menu items labeled **Save** and **Save As.** Give these items the instance names **`file_save`** and **`file_as`**.

6 Up to now, you've added Command-type menu items. You'll now add a Separator item. Just click on the Separator button near the upper-right corner

7 Click **Apply** to get the default separator style, which is Etched - in.

8 Now let's add the Quit item. Click the Command button, then specify **`Quit`** as the item text and **`file_quit`** as the instance name.

9 You're finished with this menu module for now, so click Done. The module displays the items you just created:



10 You'll want to keep this module neatly out of the way while you work on your next task. So click the module's minimize button (the left button at the right side of the title bar), or select the Work menu button (upper-left corner) and choose Minimize.

Creating the Help menu module

Using what you just learned about creating a menu module, do the following:

1 Create your Help menu module and give it a name of **`helpmenu`**.

2 In this module, place a single command item called **`About Demo`** and give the item an instance name of **`help_about`**. When you're finished, minimize the module.

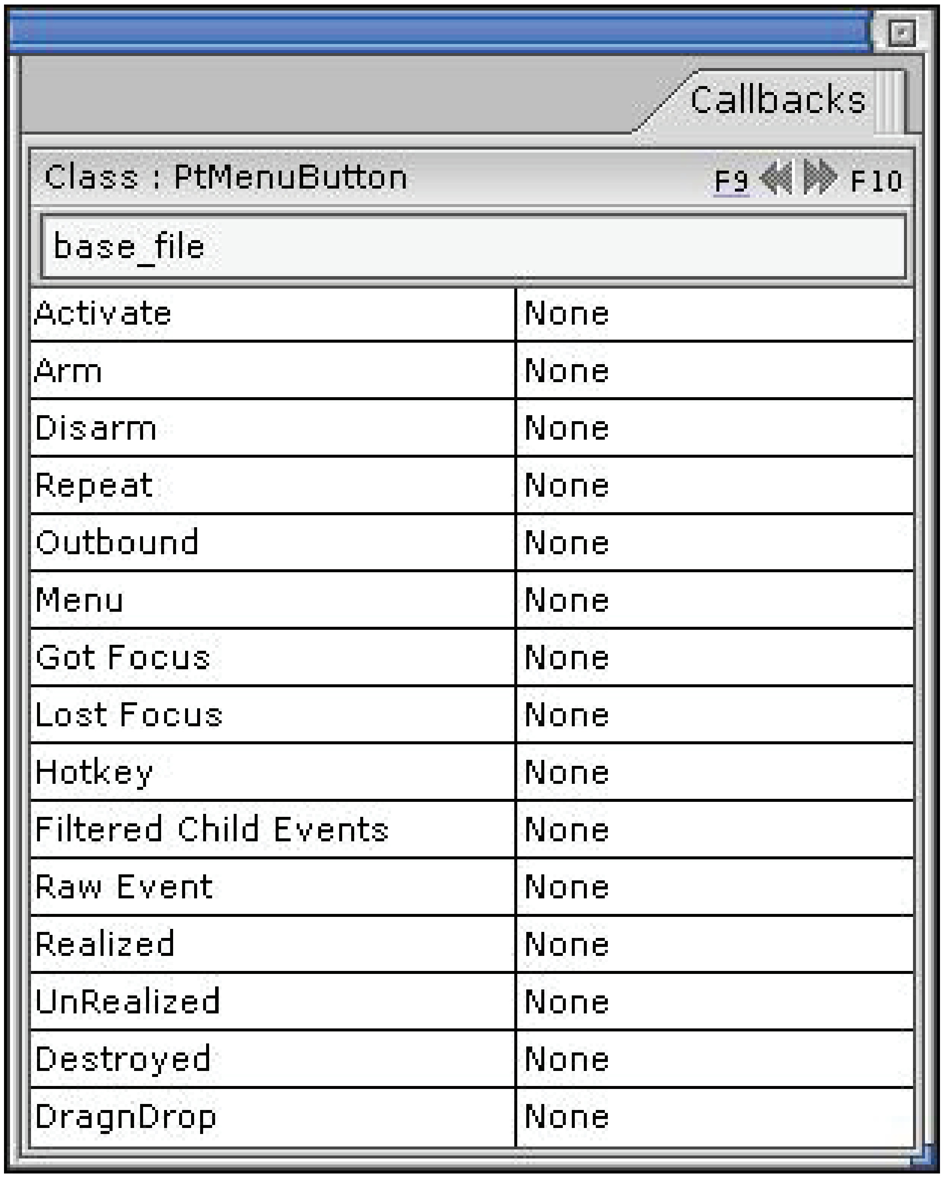
If one of your menu modules seems to "disappear" (you may have accidentally closed it or placed it behind another module), it's easy to bring the module back into view. See the "Finding lost modules and icons" in the Working with Modules chapter.

Attaching link callbacks

Let's return to the menu buttons you created earlier and attach link callbacks so that the buttons can pop up your menu modules.

Attaching a module-type link callback

1 Select the File menu button, then switch to the Callbacks control panel You'll see the File button's callback list:



Class : PtMenuButton

base_file

Activate	None
Arm	None
Disarm	None
Repeat	None
Outbound	None
Menu	None
Got Focus	None
Lost Focus	None
Hotkey	None
Filtered Child Events	None
Raw Event	None
Realized	None
UnRealized	None
Destroyed	None
DragnDrop	None

2 To have the File menu module pop up when you press the File button, you need to attach an Arm callback to the button. By attaching an Arm callback, you can open the menu using either *click-move-click* or *press-drag-release*.

Click Arm to bring up the callback editor.

3 The Module Types area of the editor let you choose the type of module you wish to link to. Because you want to link the File button to a menu module, click Menu.

4 Click the Name list and type `filemenu` (or select **filemenu** from the list) which is the name you gave your File menu module. This links the menu button to that module.

You can also select `filemenu` from a popup list of available modules. To bring up the list, click the icon to the right of the Name field.

5 Click Apply to add the link callback, then click Done to close the callback editor.

6 Repeat the above steps to link the Help menu button to the Help menu module.

Attaching a code-type link callback

Let’s now attach a code-type link callback to the File menu’s Quit item so that it can terminate the application.

1 Double-click the iconified **`filemenu`** module. This opens and selects the module.

2 Switch to the Resources control panel, then click the Menu Items resource.

3 Select the Quit item in the Menu Items list.

4 Click the icon next to the Callback field to open the callback editor:

5 When the editor opens, the default callback type is Code. Since this is the type you want, all you have to do is specify the name of the function you want to call.

The function should have a meaningful name. So type **`quit`** in the Function field.

6 Click Apply to update the Callbacks list, then click Done to close the editor.

7 Click Done again to close the menu editor.

Setting up the code

You’ll now generate the code for your application and edit a generated code stub so that the Quit item will cause your application to exit.

1 Select **Build→Generate UI**. This generates the necessary application files.

2 After the generation process is complete, open the **Browse Files** palette window by selection **Window→Show Project**.

Scroll through the list until you see the **`quit.c`** file. This is the generic code template that PhAB generated for your *quit()* function.

3 You need to make the function exit the program. To do this, select **`quit.c`** from the file list, click the Edit button, or double-click **`quit.c`**, then change the *quit()* function to the following:

```
int
quit( PtWidget_t *widget, ApInfo_t *apinfo,
      PtCallbackInfo_t *cbinfo )
{

     /* eliminate 'unreferenced' warnings */
     widget = widget,
     apinfo = apinfo,
     cbinfo = cbinfo;

     PtExit( EXIT_SUCCESS );

     /* This statement won't be reached, but it
        will keep the compiler happy. */

     return( Pt_CONTINUE );
}
```

PtExit() is a function that cleans up the Photon environment and then exits the application. It's described in the Photon *Library Reference*.

If you are using the IDE, you can also edit quit.c, or any other source code file, from the IDE editor by double-clicking the file in the project navigator tree.

4 After you've edited the code, saved your changes, and closed the editor, build and run your application.

5 Once your application is running, try clicking on the File button to bring up the File menu. Then choose Quit. Your application will immediately terminate and close.

Want more info?

For more info on:	See the section:	In the chapter:
Widget callbacks	Callbacks	Editing Resources and Callbacks in PhAB
	Editing callbacks	Editing Resources and Callbacks in PhAB
Instance names	Instance names	Creating Widgets in PhAB
Menu modules	Menu modules	Working with Modules

Tutorial 4 — creating dialogs

This tutorial describes how to create a dialog. It also provides a good example of how you can use setup code to modify a widget's resources before the widget appears onscreen.

This tutorial uses the application you created in Tutorial 3.

In this tutorial, you'll:

- link the **About Demo** item in the Help menu to a dialog
- add labels and a Done button to the new dialog
- define a setup function that changes the text of one of the labels to display a version number when the dialog is realized.

About dialogs

Dialog modules are designed to let you obtain additional information from the user. Typically, you use this information to carry out a particular command or task.

Since you don't usually need to get the same information twice, dialogs are single-instance modules. That is, you can't realize the same dialog more than once at the same time. If you try create a second instance of a dialog, PhAB simply brings the existing dialog to the front and gives it focus.

If you need to create a window that supports multiple instances, use a window module. You'll learn about window modules in the next tutorial.

More on instance names

To make it easier for you to access widgets from within your application code, PhAB generates a constant and a manifest. Both of these are based on the widget's instance name.

The constant, which has the prefix **`ABN_`**, represents the widget's name. The manifest, which has the prefix **`ABW_`**, represents the widget's instance pointer.

For example, let's say you have a widget named **`about_version`**. PhAB uses this name to generate a constant named ABN_about_version and a manifest named ABW_about_version.

In this tutorial you'll learn how to use these generated names.

The value of a widget's ABN_... constant is unique in the entire application.

Attaching a dialog module

1. Make a copy of the **`tut3`** application you created and name it **`tut4`**:
   - From the IDE — select the project, select **Edit→Copy**, and then select **Edit→Paste**. You can enter the new name for the project in the dialog that appears.
   - From standalone PhAB — open the **`tut3`** application and use the File menu's Save As item to save the application as **`tut4`**.
2. Open the Help menu module you created (it may still be iconified).
3. Click the Menu Items resource in the Resources control panel to open the menu editor.
4. Select the **About Demo** item, then click the icon next to the Callback field to open the callback editor:
5. When the editor opens, the default callback type is Code. Go to the Module Types group and change the callback type to Dialog.

6 In the Name field, type **`aboutdlg`** as the name of the dialog module you want to link to. (This dialog doesn't exist yet, but PhAB will ask you later to create it.)

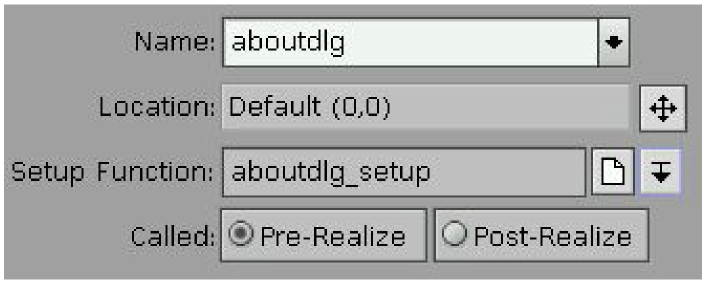
7 In the Setup Function field, type **`aboutdlg_setup`**. This is the name we're giving to the setup function that will be called before the dialog is realized.

Using this function, we'll change the content of a label widget within the dialog to display a version number.

8 Since you want the **`aboutdlg_setup`** function to be called before the dialog is realized, make sure the Prerealize button is enabled.

9 Click the Location icon to specify where you want the dialog to appear when it gets realized. (The **Center Screen** location is a good choice.) Click Done.

Your callback information should now look something like this (depending on the location you chose):


Name: aboutdlg
Location: Default (0,0)
Setup Function: aboutdlg_setup
Called: [✓ Pre-Realize] Post-Realize

10 Click on Apply in the Actions group to add the link callback. Since the dialog module you want to link to doesn't exist yet, PhAB asks you to choose a style; select Plain and click Done.

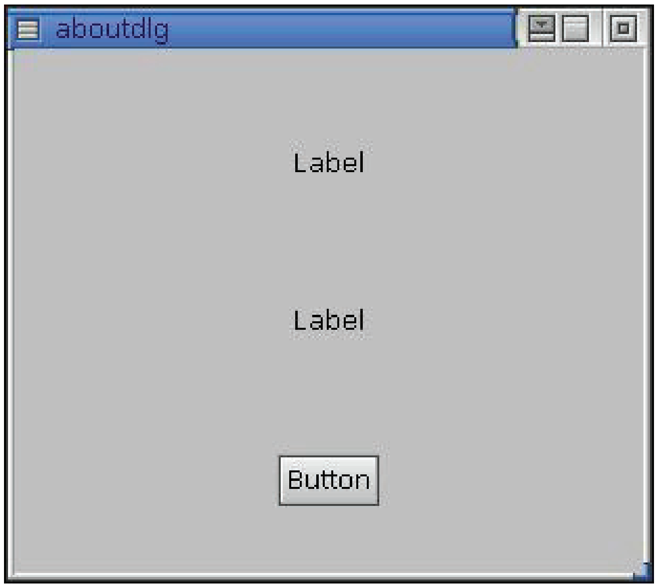
You'll see the new dialog in the work area. You'll also see the new callback in the Callbacks list in the callback editor.

11 Click Done to close the callback editor, then click Done again to close the menu editor.

Adding widgets to the dialog

1 Open the **`aboutdlg`** dialog module.

2 Place two **`PtLabel`** widgets in the top half of the dialog, and a **`PtButton`** near the bottom:



3 Select the top **`PtLabel`** widget and change its label text resource to **`About this Demo`**. Then change its horizontal alignment to Pt_CENTER.

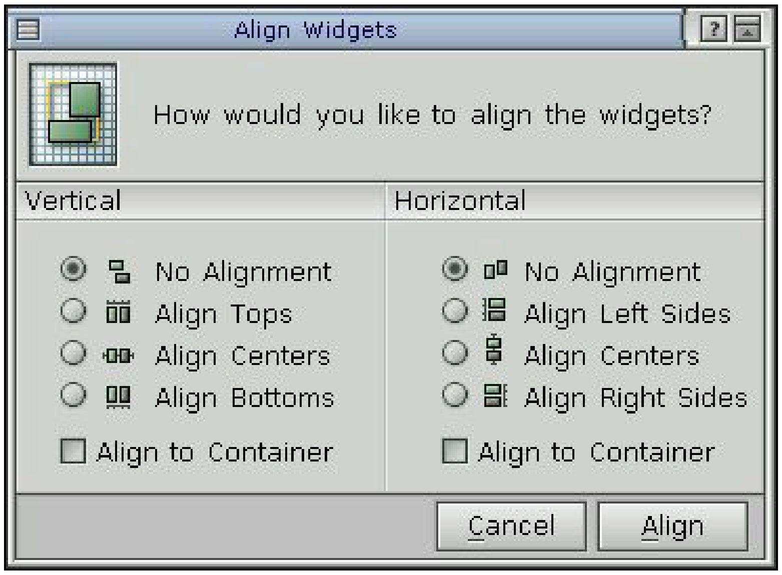
4 Select the other **`PtLabel`** widget and change its label text to a blank string. Then change its horizontal alignment to Pt_CENTER.

Later, you'll fill in the *aboutdlg_setup()* function so that it changes the blank text of this label to display a version number.

5 You must give this blank **`PtLabel`** widget an instance name since you'll be referring to it in code. So change its instance name to **`about_version`**.

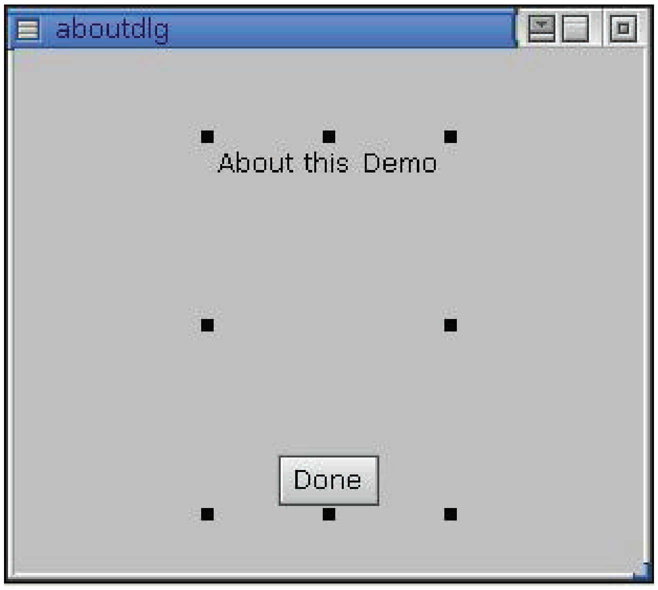
6 Select the **`PtButton`** widget and change its button text resource to **`Done`**. Then change its instance name to **`about_done`**.

7 Let's center the widgets horizontally in the dialog. Select both **`PtLabel`** widgets and the **`PtButton`** widget, choose Align from the Widget menu, and then choose Alignment Tool from the submenu. You'll see the Align Widgets dialog:



8 In the Horizontal column, click Align Centers and on Align to Container. Then click the Align button.

The two labels and the button should now be centered horizontally within your dialog. Your **aboutdlg** module should now look like this:



Adding a callback to the Done button

Now let's add a callback to the Done button so that the dialog closes when the user clicks on the button.

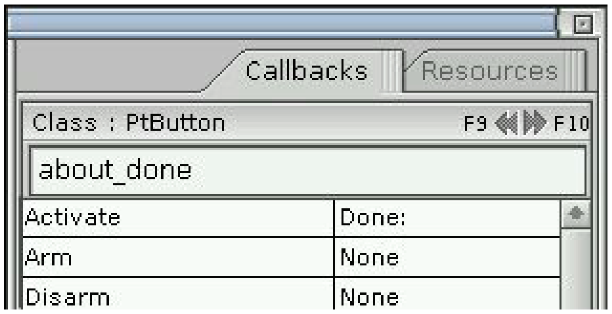
1 Select the Done button, then switch to the Callbacks control panel.

2 Click Activate to add an activate callback. You'll see the callback editor.

3 Select the Done code type, then click Apply. Don't enter anything in the Function field.

Selecting the Done code type tells the widget to perform a "Done" operation when the widget is activated. That is, the widget calls the function specified in the Function field (if one is specified) and then closes the dialog module.

4 Close the editor. The callback list now indicates that you've added an Activate callback called Done:



Activate	Done:
Arm	None
Disarm	None

Modifying a generated code function

You'll now modify the generated *aboutdlg_setup()* function so that it changes the text of the **`about_version`** label to show a version number.

1 Select **Build→Generate UI**. This save yours application and generates the necessary files.

2 When code generation is complete, choose **Window→Show Project** to bring the **Browse Files** palette window to front. Select **`aboutdlg_setup.c`** from the file list, and click **Edit**, or double-click the filename. If you are using PhAB from the IDE, you can open and edit this file in the IDE.

Change the code from:

```
int
aboutdlg_setup( PtWidget_t *link_instance,
                ApInfo_t *apinfo,
                PtCallbackInfo_t *cbinfo )
{

    /* eliminate 'unreferenced' warnings */
    link_instance = link_instance,
                    apinfo = apinfo,
                    cbinfo = cbinfo;

    return( Pt_CONTINUE );
}
```

to the following:

```
int
aboutdlg_setup( PtWidget_t *link_instance,
                ApInfo_t *apinfo,
                PtCallbackInfo_t *cbinfo )
{

    /* eliminate 'unreferenced' warnings */
    link_instance = link_instance, apinfo = apinfo,
                    cbinfo = cbinfo;

    PtSetResource( ABW_about_version, Pt_ARG_TEXT_STRING,
                   "1.00", 0);

    return( Pt_CONTINUE );

}
```

The code is placing the version number (1.00) into the text string resource for the **`about_version`** widget. To do this, the code calls *PtSetResource()* to set the resource for the **`about_version`** widget. The code uses the PhAB-generated manifest **`ABW_about_version`**, which provides access to the widget's instance pointer.

We can use this manifest safely since we're dealing with a dialog module—PhAB ensures that only one instance of the dialog will exist at any given time.

3 Save your changes and exit the text editor.

Compiling and Running

You're now ready to compile and run the program:

1 Build and run your application. If your program compiles and links without errors (which it should if you edited the function correctly), it will run.

2 From the running application, open the Help menu and choose About Demo. The dialog will open, and you'll see the version number (1.00) under the label **About this Demo.** Note that the dialog appears in the location you specified.

3 Now try to bring up a second instance of the dialog. As you see, it won't work. PhAB always ensures that there is only one instance of a Dialog widget.

4 Click **Done** to close the dialog, then quit the application by choosing **Quit** from its **File** menu.

Want more info?

For more info on:	See the section:	In the chapter:
Using dialogs	Dialog modules	Working with Modules
Instance names	Instance names	Creating Widgets in PhAB
	Variables and manifests	Working with Code
Callbacks	Callbacks	Editing Resources and Callbacks in PhAB

continued. . .

For more info on:	See the section:	In the chapter:
	Code-callback functions	Working with Code
Generating code	Generating application code	Generating, Compiling, and Running Code

Tutorial 5 — creating windows

In the previous tutorial, you learned how to handle dialog modules, which support just one instance. In this tutorial you'll learn how to handle window modules, which support multiple instances.

This tutorial uses the application you created in Tutorial 4.

By supporting multiple instances, window modules provide more flexibility than dialogs. But to take advantage of this flexibility, you must keep track of each window's instance pointer. Doing so ensures that you correctly reference the widgets within each instance of a window. You can't safely use the generated global ABW_*xxx* manifest since it refers only to the last instance created.

To simplify the task of working with multiple instances, PhAB provides API library functions that let you access any widget by means of its generated constant name (ABN_*xxx*).

Creating a window

To start, let's create a window module and attach it to the New menu item in the File menu in **`tut4`**. This window will contain buttons that change the color of another widget.

In the previous tutorial, you created a dialog module from within the callback editor. But this time you'll add the window from the **Project** menu. In the future, use whatever method you prefer.

1. Make a copy of the **`tut4`** application and call it **`tut5`**.
2. Iconify the **`aboutdlg`** dialog module.
3. From the **Project** menu, select **Add Window**. When PhAB prompts you for a window style, choose the Plain style.
4. Change the new window's instance name from **`Window0`** to **`newwin`**, by typing the new name in the instance name field in the control Panel.
5. The window module should now be the currently selected item.

Attaching callbacks

Because a window module supports multiple instances, you have to create code functions that will be called whenever the window opens or closes (i.e. whenever the window is created or destroyed). So let's first set up a callback to detect when the window closes:

1. Switch to the Callbacks control panel, if necessary.
2. From the list of callbacks, choose Window Manager. You want to use the Window Manager callback since it's invoked when the Photon Window Manager closes the window.
3. In the Function field, type **`newwin_close`**. You don't have to choose a callback type since the default, Code, is the one you want.

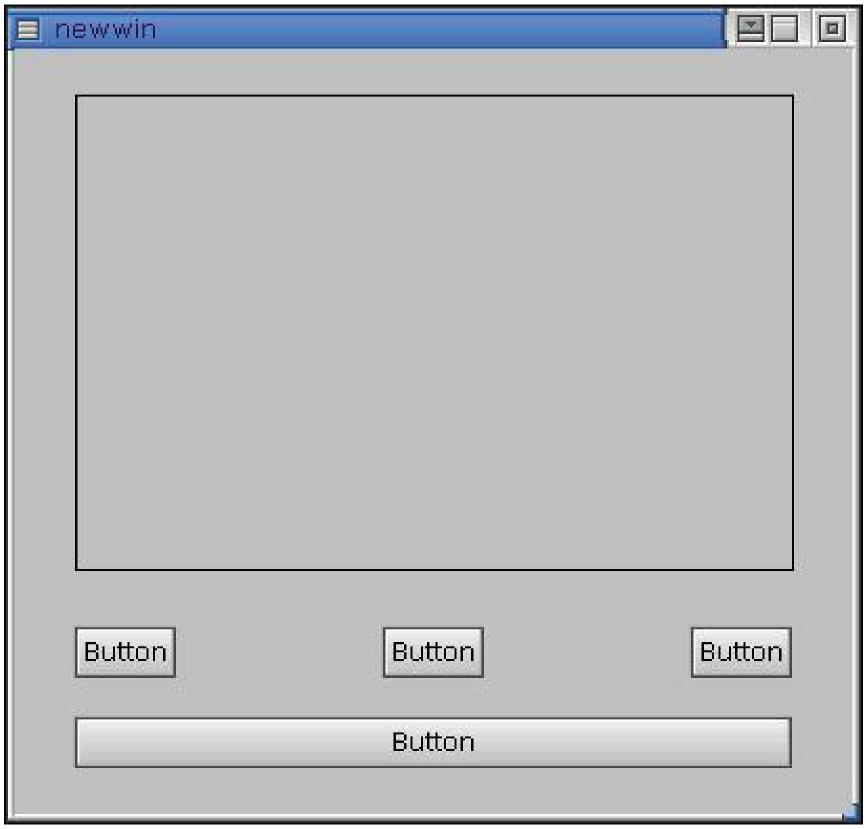
   Click Apply, then Done.
4. Switch to the Resources control panel and select the Flags: Notify resource. Make sure that the Ph_WM_CLOSE flag is set (i.e. highlighted), then click Done. This flag tells the Window Manager to notify your application when the window is closed.
5. Now let's set up a function that's invoked when the window opens.

   Open the **`filemenu`** menu module, then select the Menu Items resource in the Resources control panel. You'll see the menu editor.
6. Make sure the menu's New item is currently selected in the Menu Items list, then click the Callback icon to open the callback editor.
7. Choose the Window module type, then click the arrow next to the Name field. You'll see the list of existing window modules.
8. Choose **`newwin`**, which is the window you just created.
9. In the Setup Function field, enter **`newwin_setup`** as the name of the setup function. Later, you'll modify *newwin_setup()* to handle the window's multiple instances.
10. Click Apply, then on Done. Click Done again to close the menu editor.

Adding widgets

Let's now add some widgets to the **`newwin`** window module. Using these widgets, you'll learn how to update information in the current or other instances of a window module.

1. Add a **`PtRect`** widget and four **`PtButton`** widgets as follows:



2 Now modify the left button:

- Change the button's label text to **`Red`**.
- Give the button an instance name of **`btn_red`**.
- Attach an Activate callback, specifying a "Code" code type and a function name of **`color_change`**.

3 Modify the middle button:

- Change the label text to **`Green`**.
- Specify an instance name of **`btn_green`**.
- Attach an Activate/Code callback to the same function as above, **`color_change`**.

4 Modify the right button:

- Change the label text to **`Blue`**.
- Specify an instance name of **`btn_blue`**.
- Attach an Activate/Code callback to the same function as above.

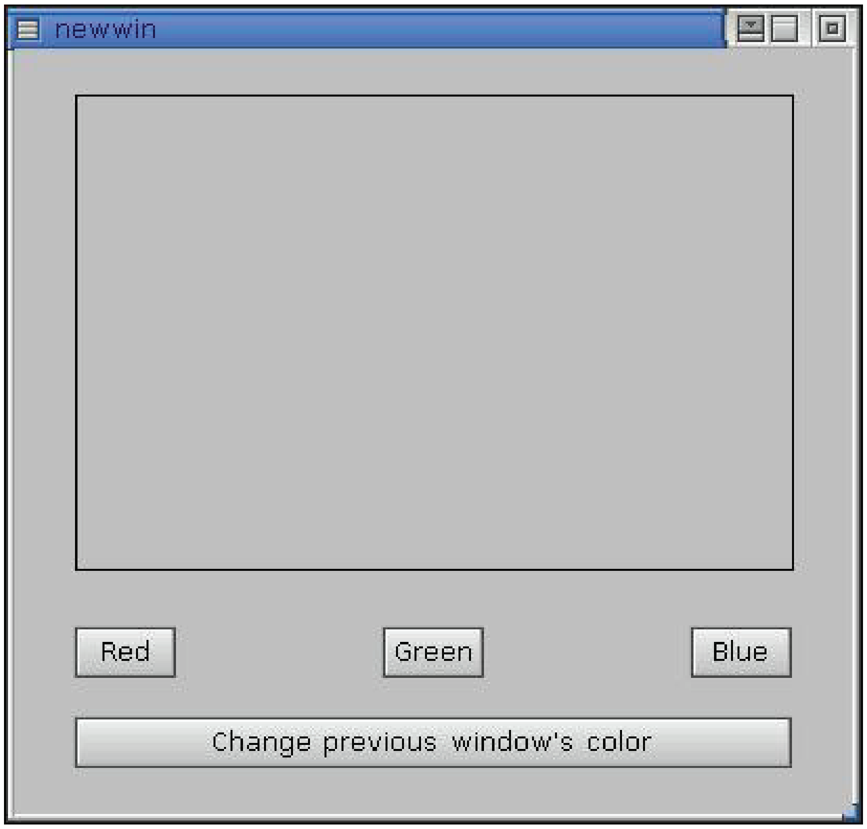
5 Modify the large button:

- Change the label text to **`Change previous window's color`**.
- Specify an instance name of **`btn_prev`**.

- Attach an Activate/Code callback to the same function as above.

6 Last of all, give the rectangle an instance name of **`color_rect`**. You need to specify this name so that the *color_change()* function can change the color of the rectangle.

Your window should now look something like this:



Generating and modifying the code

In the last tutorial, you used the generated ABW_*xxx* manifest to access a dialog's instance pointer. You can't use this manifest when dealing with multiple instances of a window module since it refers only to the last window created. Instead, you have to add code to the generated window-setup function so that it stores a copy of each window-instance pointer in a global widget array. In this tutorial, you'll need these pointers for the **Change Previous Window Color** button to work.

Generating the code

Open the **Build** menu and select **Generate UI**.

Modifying the setup function

Now let's modify the *newwin_setup()* function so that it:

- limits the number of possible instances to five

- stores a copy of each window pointer
- displays a window's instance number in that window's titlebar

Edit the **`newwin_setup.c`** file as follows:

```
int         win_ctr = 0;
PtWidget_t *win[5];

int
newwin_setup( PtWidget_t *link_instance,
              ApInfo_t *apinfo,
              PtCallbackInfo_t *cbinfo )
{
    char    buffer[40];

    /* eliminate 'unreferenced' warnings */
    link_instance = link_instance, apinfo = apinfo;
    cbinfo = cbinfo;

    /* Note: Returning Pt_END in a prerealize setup
       function tells PhAB to destroy the module
       without realizing it */

    /* allow only 5 windows max */
    if ( win_ctr == 5 ) {
        return( Pt_END );
    }

    /* save window-module instance pointer */
    win[win_ctr] = link_instance;

    sprintf( buffer, "Window %d", win_ctr + 1 );
    PtSetResource( win[win_ctr], Pt_ARG_WINDOW_TITLE,
                   buffer, 0 );
    win_ctr++;

    return( Pt_CONTINUE );

}
```

Modifying the color-change function

Now let's modify the *color_change()* function so that:

- pressing a Red, Green, or Blue button changes the rectangle color to the button color
- pressing the Change Previous Window Color button changes the background of the previous window to a color from an array.

If this were a dialog module you could use the **`ABW_color_rect`** manifest to update the color of the rectangle. However, because these are window modules, you must use the instance pointer for the window in which the button is being pressed.

To get the instance pointer of a widget in the current window, you need to call:

- *ApGetInstance()* to get a pointer to the window that contains the widget that invoked the callback
- *ApGetWidgetPtr()* to get a pointer to the widget with a given ABN_... manifest.

If only one instance of the window were guaranteed, the following would work:

```
PtSetResource( ABW_color_rect, Pt_ARG_FILL_COLOR,
               buffer, 0 );
```

But in this case *color_change()* has to use:

```
PtSetResource( ApGetWidgetPtr( ApGetInstance( widget ),
               ABN_color_rect ), Pt_ARG_FILL_COLOR,
               buffer, 0 );
```

So you need to change **`color_change.c`** to look like:

```
PgColor_t          colors[5] = {Pg_BLACK, Pg_YELLOW,
                                Pg_MAGENTA, Pg_CYAN,
                                Pg_GREEN};
int                base_clr = -1;
extern int         win_ctr;
extern PtWidget_t *win[5];

int
color_change( PtWidget_t *widget, ApInfo_t *apinfo,
              PtCallbackInfo_t *cbinfo )
{
  int     i, prev;
  PtWidget_t *this_window;

  /* eliminate 'unreferenced' warnings */
  widget = widget, apinfo = apinfo, cbinfo = cbinfo;

  /* Get a pointer to the current window. */
  this_window = ApGetInstance( widget );

  if ( ApName( widget ) == ABN_btn_red ) {
       PtSetResource(
          ApGetWidgetPtr( this_window, ABN_color_rect ),
          Pt_ARG_FILL_COLOR, Pg_RED, 0 );

  } else if ( ApName( widget ) == ABN_btn_green ) {
       PtSetResource(
          ApGetWidgetPtr( this_window, ABN_color_rect ),
          Pt_ARG_FILL_COLOR, Pg_GREEN, 0 );

  } else if ( ApName( widget ) == ABN_btn_blue ) {
       PtSetResource(
          ApGetWidgetPtr( this_window, ABN_color_rect ),
          Pt_ARG_FILL_COLOR, Pg_BLUE, 0 );

  } else if ( ApName( widget ) == ABN_btn_prev ) {

       /* Note: Here we use the window-module instance
                pointers saved in newwin_setup to update
                the window previous to the current window
                provided it hasn't been closed.

          Determine which window is previous to this window. */
```

```
        prev = -1;
        for ( i = 0; i < win_ctr; i++ ) {
            if ( win[i] == this_window ) {
                prev = i - 1;
                break;
            }
        }

        /* If the window still exists, update its background
           color. */

        if ( prev != -1 && win[prev] ) {
            base_clr++;
            if (base_clr >= 5) {
                base_clr = 0;
            }
            PtSetResource( win[prev], Pt_ARG_FILL_COLOR,
                           colors[base_clr], 0 );
        }
    }

    return( Pt_CONTINUE );
}
```

Modifying the window-close function

Last of all, you need to modify *newwin_close()* so that it sets the **win** array of instance pointers to NULL for a window when it's closed. That way, you can check for NULL in the **win** array to determine whether the window still exists.

Modify **newwin_close.c** as follows:

```
extern int         win_ctr;
extern PtWidget_t *win[5];

int
newwin_close( PtWidget_t *widget, ApInfo_t *apinfo,
              PtCallbackInfo_t *cbinfo )
{
    PhWindowEvent_t *we = cbinfo->cbdata;
    int              i;

    /* eliminate 'unreferenced' warnings */
    apinfo = apinfo;

    /* only process WM close events */
    if ( we->event_f != Ph_WM_CLOSE ) {
        return( Pt_CONTINUE );
    }

    /* okay it's a close so who is it? */
    for ( i = 0; i < win_ctr; i++ ) {
        if ( win[i] == widget ) {
            win[i] = NULL;
            break;
        }
    }

    return( Pt_CONTINUE );
}
```

Compiling and running

1. Make and run the application.
2. From the application's File menu, choose New several times to create multiple windows. You'll see each window's relative number in its titlebar.
3. Click a color button to make the rectangle change color. Then click the Change Previous Window Color button in any window to change the background color of the previous window.

Want more info?

For more info on:	See the section:	In the chapter:
Using windows	Window modules	Working with Modules
Instance names	Instance names	Creating Widgets in PhAB
	Variables and manifests	Working with Code chapter
Callbacks	Callbacks	Editing Resources and Callbacks in PhAB
	Code-callback functions	Working with Code
Generating code	Generating application code	Generating, Compiling, and Running Code
Window events	Window-management flags	Window Management

Chapter 3

PhAB's Environment

In this chapter. . .

This chapter describes PhAB's environment in more detail, and how you can customize it.

Menus

Across the top of PhAB's workspace you'll see the following menubar:

PhAB's menubar.

File menu

Commands that deal with your application and its files:

New	Standalone PhAB only*. Create a new application; see "Creating an application" in the Working with Applications chapter.
Open	Standalone PhAB only*. Open an existing application; see "Opening an application" in the Working with Applications chapter. This command is also available through PhAB's toolbars.
Close	Standalone PhAB only*. Close the current application; see "Closing an application" in the Working with Applications chapter.
Revert	Discard any changes and re-load the project from the last saved version.
Save Save As	Save the current application, under the same or a different name; see "Saving an application" in the Working with Applications chapter. The Save command is also available through PhAB's toolbars. **Save As** is available in Standalone PhAB only*.
Import	Import files created by other applications; see "Importing PhAB modules from other applications" in the Working with Applications chapter.
Export	Export the code used to create a module in a file called **`module.code`**; see "Export files" in the Working with Applications chapter.
Exit	End your current PhAB session. PhAB prompts you if there are any changes that you haven't yet saved.

This menu also lists the last few applications that you edited.

*When using PhAB in the IDE, projects are managed from the IDE, so these menu items are disabled. For more information see "Creating a QNX Photon Appbuilder Project", in the Developing Photon Applications chapter of the IDE *User's Guide*.

Edit menu

Commands for editing widgets:

Undo Redo	Undo and redo an operation, including: ● creating, deleting, moving, resizing, aligning, and selecting a widget ● changing the order of widgets ● changing a callback resource — PhAB can't undo/redo any changes you make to a callback function's code ● pasting ● joining and splitting a group ● editing a resource ● importing images, XBMs, and modules. ● changing a widget's class, matching widget resources and callbacks, and transferring widgets The text **Undo** and **Redo** menu items changes depending on the current operation.
Cut Copy Paste	Cut and copy widgets to the clipboard, and paste them from it; see "Clipboard" in the Creating Widgets in PhAB chapter. These operations also apply to module links.
Move Into	Move a widget from one container to another; see "Transferring widgets between containers" in the Creating Widgets in PhAB chapter.
Delete	Delete a widget without saving it on the clipboard; see "Deleting widgets" in the Creating Widgets in PhAB chapter.
Select All	Select all the widgets in the current module
Select All Children	Select all the children widgets in the current selected widget or module.
Deselect	Deselect the current widget or widgets. This option also selects the current module.
Find...	Display the **Find** dialog, which allows you to do find widgets in the current application by name, type, contained text, called function, or link.

Add Widget Class...	Add a brand new widget class to the application
Templates...	Customize PhAB's widget palettes. You can create new palette folders, add widgets to a folder, delete widgets, and change icons, colors and other properties.
Preferences	PhAB preferences, such as colors, editing commands, and styles of resource names.

Many of these commands also appear in PhAB's toolbars.

Project menu

Commands for building and configuring a project:

Add Window	Open the **New Window Style** dialog to add a new window to your project.
Add Dialog	Open the **New Dialog Style** dialog to add a new dialog to your project.
Add Menu	Add a new menu module to your project.
Add Picture Module	Add a new picture module to your project.
Edit Mode	Switch to edit mode, which means you can add widgets and modules to your projects, add callbacks, and so on. When PhAB is launched, it starts in this mode.
Test Mode	Switch to test mode, which means you can test the appearance of your GUI.
Zoom Mode	Allows you to zoom in, zoom out or move around, by using the mouse and keyboard. After zooming in or out on the desired area, you must return to Edit Mode or Test Mode to continue editing or testing.
Internal Links	An *internal link* is a PhAB mechanism that lets you access a PhAB module directly from your application's code; see the Accessing PhAB Modules from Code chapter.
Generate Report	Generate a report on the application's widgets and modules.
Language editor	A menu of commands used to create multilingual versions of your application; see the International Language Support chapter.

Properties	Information used for the application as a whole, including global headers, initialization function, and which modules to display at startup. For more information, see "Specifying your project properties" in the Working with Applications chapter.
Convert to Eclipse Project	Standalone PhAB only. Explicitly converts your project to the Eclipse PhAB Project format, if it hasn't already been converted. A project must be in this format to be opened by PhAB from the IDE. New projects are created in Eclipse PhAB project format by default, but you must explicitly convert multiplatform projects created in a previous version of PhAB.

Build menu

Commands for compiling and running an application:

Build & Run	Standalone PhAB only*. Builds the current project, and then runs it. If required, this command will save open files, generate support files, compile, and link the application. See the Generating, Compiling, and Running Code chapter.
Build & Debug	Standalone PhAB only*. Builds the current project, then launches it inside the preferred debugger. You can set breakpoints, step instruction by instruction, etc. See the Generating, Compiling, and Running Code chapter.
Rebuild All	Standalone PhAB only*. Rebuilds the current application by compiling all the files (equivalent to a "Make Clean" and then a "Build"). See the Generating, Compiling, and Running Code chapter.
Build	Standalone PhAB only*. Builds the current application. If required, this command will save open files, generate support files, compile, and link the application. See the Generating, Compiling, and Running Code chapter.
Make Clean	Standalone PhAB only*. Runs a "make clean" for all the current target platforms.
Generate UI	Generates just the supporting files for the current application. See the Generating, Compiling, and Running Code chapter.
Run	Standalone PhAB only*. Runs the last compiled version of the current application. See the Generating, Compiling, and Running Code chapter.
Targets	Standalone PhAB only*. Opens the **Manage Targets** dialog, which allows you to add and delete target platforms for the

current application. See the Generating, Compiling, and Running Code chapter.

*When using PhAB in the IDE, projects are built and run from the IDE, so these menu items are disabled. For more information see “Building a QNX Photon Appbuilder Project”, in the Developing Photon Applications chapter of the IDE *User’s Guide*.

Widget menu

Commands for manipulating widgets:

Arrange	Contains commands that let you change the order of selected widgets relative to one another. You can **Raise**, **Lower**, **Move To Front**, **Move To Back**.
Align	Contains commands for aligning the selected widgets; see “Aligning widgets” in the Creating Widgets in PhAB chapter.
Distribute	Contains commands to evenly distribute selected widgets horizontally or vertically, taking into account the size of each widget.
Match Height MatchWidth Match Resources Match Callbacks	These commands match the respective resources or callbacks of the selected widgets. The widget selected first is used as the source and subsequently selected widgets are modified. The source and destination widgets are highlighted with different colors (provided you have **Show Selection** set in the **View** menu).
Match Advanced	Opens the **Match resources and callbacks** dialog, which lets you match multiple resources and callbacks. The widget selected first is used as the source and subsequently selected widgets are modified. If the destination widget has the selected resource, then its value is set to the same value as the source widget’s corresponding resource.
Group Ungroup	Combine selected widgets into a group, or split up a selected group; see “Aligning widgets using groups” in the Geometry Management chapter.
Lock	Contains commands to lock the position or size of the selected widget.
Change Class	Change the class of the selected widgets; see “Changing a widget’s class” in the Creating Widgets in PhAB chapter.

Define Template	A template is a customized widget that you want to use as the basis for other widgets. This command opens the **Define template** dialog, which lets you create or edit a template; see "Templates" in the Creating Widgets in PhAB chapter.

View menu

Commands that change the way the modules in the work area are displayed:

Zoom In Zoom Out	Zoom the display in or out. Use these commands for doing precise alignments of your widgets. For example, if you zoom out so that the zoom factor is less than 100%, your workspace is larger, helping you work on a 1280x1024 application even if your display is set to 800x600.
Fit in Window	Fits the current selection to the window by adjusting the zoom factor.
Actual Size	Returns to 100% zoom, the default screen size.
Show Grid	Toggles the grid display. The settings for the grid can be adjusted on the **Grid** tab of the **AppBuilder Preference Settings** dialog. See the **Preferences** option of the Edit menu.
Snap to Grid	Toggles the snap to grid option. When this option is on, new or moved widgets are "snapped" to the grid—that is, they are aligned with the closest vertical and horizontal grid lines. The grid does not have to be visible for this option to work.
Show Selection	Toggles the widget selection option. When this option is on, selected widgets are highlighted with colored rectangles. When multiple widgets are selected, the first widget has a different selection color than subsequently selected widgets. This makes it easier to align or match widgets, where the first selected widget is the source. It also makes it easier to see what's exactly selected when widgets overlap.

Window menu

Commands that manipulate PhAB's windows:

Cascade	Arranges the currently open modules so that they're stacked from the upper left to lower right in PhAB's workspace.
Arrange Icons	Arranges iconified modules in PhAB's workspace in a grid along the bottom of the workspace.

Send To Back	Sends the currently selected module to the back of the workspace. Note that a module will always appear in front of the iconified modules.
Close	Iconifies the currently selected module. A closed module appears as an icon at the bottom of the workspace.
Close All	Iconifies all the application's modules.
Show Templates Show Resources Show Callbacks Show Module Links Show Module Tree Show Project	These commands show or hide control panels for defined templates, resources, callbacks, module links, module trees, and project files. If a control panel is shown, the corresponding menu command changes to **Hide**.

Help menu

Get online help information:

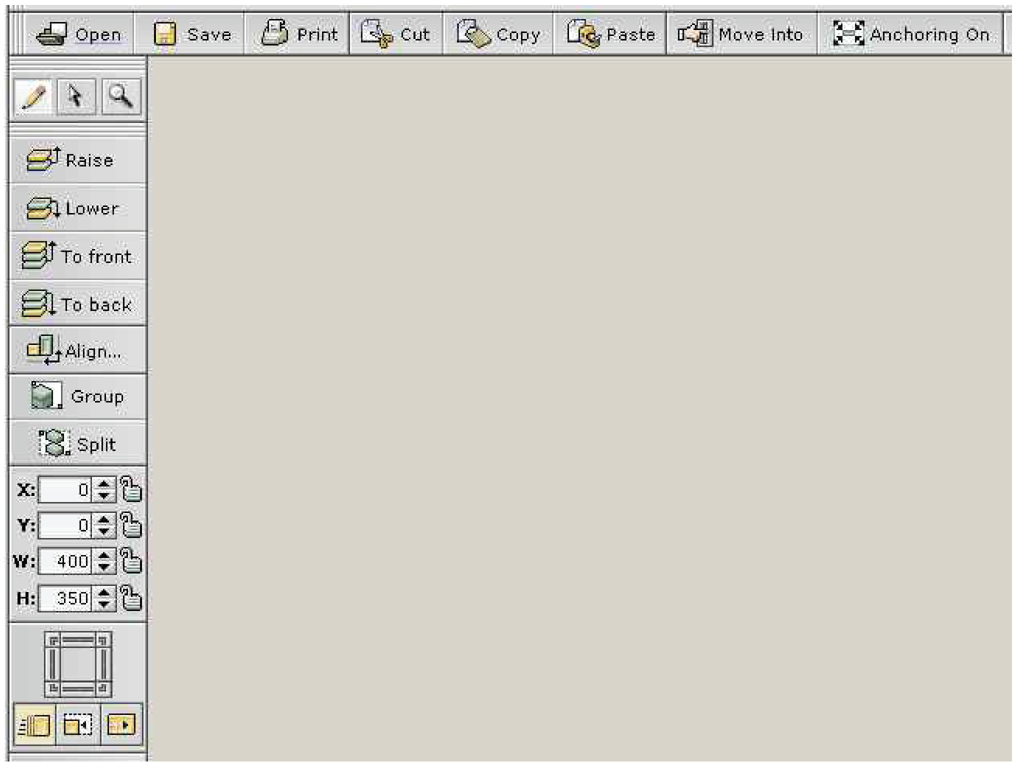
Welcome to PhAB Tutorials PhAB Concepts Tools + Techniques	Links to the appropriate section of this programmer's guide.
PhAB Library API	A link to the Photon *Library Reference*.
About PhAB	The version number and copyright information for PhAB.

There are other forms of help available in PhAB:

- Context-sensitive help — To get help on a part of PhAB's user interface, click on the question mark button, then click on the item in question. The Helpviewer displays the information on the selected item.
- Balloon help — To find out what a button in the widget palette or toolbars is for, pause the pointer over it. A descriptive balloon appears.

Toolbars

The toolbars give you quick access to frequently used commands from the menu bar:



PhAB's toolbars.

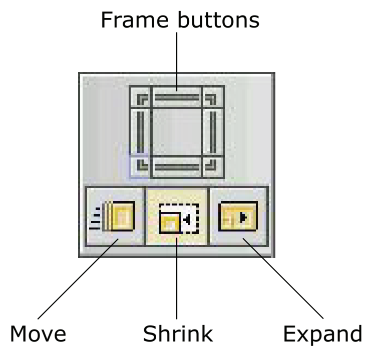
Open	Standalone PhAB only*. Open an existing application; see "Opening an application" in the Working with Applications chapter. This command is also available through the File menu.
Save	Save the current application; see "Saving an application" in the Working with Applications chapter. The Save command is also available through the File menu.
Print	Not implemented.
Cut Copy Paste	Delete and copy widgets to the clipboard, and paste them from it; see "Clipboard" in the Creating Widgets in PhAB chapter. These commands are also available through the Edit menu.
Move Into	Move a widget from one container to another; see "Transferring widgets between containers" in the Creating Widgets in PhAB chapter. This command corresponds to the Move Into command in the Edit menu.

Anchoring on/off	Turn the anchor flags for widgets on or off in the design workspace. This setting does not affect run-time anchoring in your application. For more information on anchoring widgets, see "Constraint management using anchors" in the Geometry Management chapter.
Edit Mode	Switch to edit mode, where you can add widgets and modules to your project, add callbacks, and so on. This is the default mode when PhAB is launched. This command corresponds to the Edit Mode command in the Project menu.
Test Mode	Switch to test mode, where you can test the appearance of your GUI. This command corresponds to the Test Mode command in the Project menu.
Zoom Mode	Switch to zoom mode, where you can increase or decrease the zoom by using the mouse and keyboard. After changing the zoom, you must return to edit or test mode to continue editing or testing. This command corresponds to the Zoom Mode command in the Project menu.
Raise Lower To Front To Back	Move the selected widgets forward or backward in, or to the front or back of the widgets in the container; see "Ordering widgets" in the Creating Widgets in PhAB chapter. The To Front and To Back commands are also available through the Arrange submenu of the Widget menu.
Align	The most frequently used commands for aligning the selected widgets; see "Aligning widgets" in the Creating Widgets in PhAB chapter. For more choices of alignment, see the Alignment item in the Widget menu.
Group Ungroup	Combine selected widgets into a group, or break up a selected group; see "Aligning widgets using groups" in the Geometry Management chapter. These commands are also available through the Widget menu.
X Y W H	The coordinates and size of the currently selected widget. To change them, type the new values and press Enter. To avoid changing a coordinate or dimension for the current widget, lock it by clicking on the padlock so that it closes. You can't change the field (either by typing or dragging) until you

unlock it, although you can change its position using the nudge tool. The locks are saved with your application.

Nudge tool

This tool lets you move, expand, or shrink a widget. Click on the button for the desired mode, and then click on the frame buttons above:



The nudge tool's components.

Every click on the frame buttons nudges, stretches, or shrinks the selected widget by one pixel. To nudge by multiple pixels, hold down the mouse button.

You can also use the Ctrl key and the numeric keypad to nudge, stretch, or shrink a widget. Each key corresponds to one of the nudge buttons. Pressing Ctrl-5 switches between modes, and Ctrl-↑ works like the tool's top frame button.

Control panels

PhAB includes a set of control panels (also referred to as "palette windows") that display information about the currently selected widget or widgets. They're displayed by default in PhAB, and you can move them anywhere you like. If you close a control panel, you can reopen it by choosing the appropriate item from the View menu.

The control panels include:

- Widget palette
- Resources panel
- Callbacks panel
- Module Tree panel
- Module Links panel
- Browse Files panel

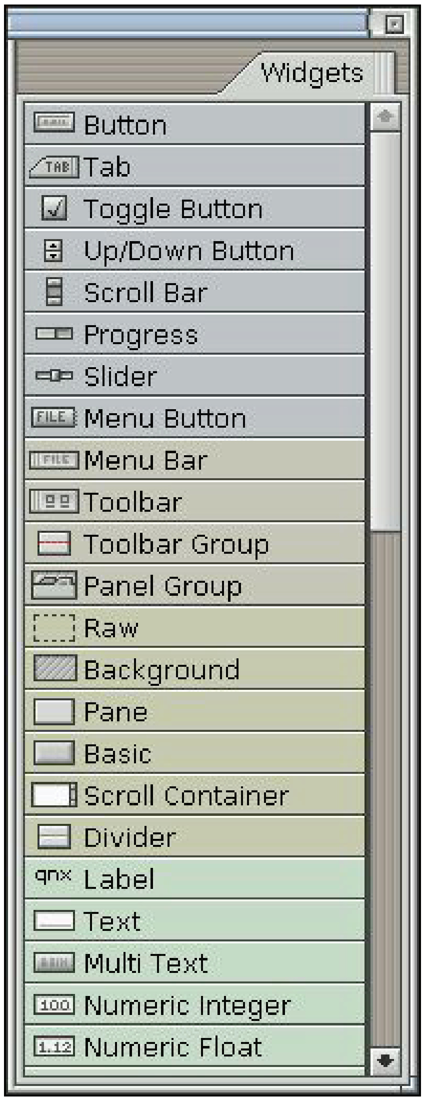
They're described in the sections that follow.

The control panels are originally displayed as a stack in a **`PtPanelGroup`** widget. If you click on the panel tab, a menu of the panels appears. If you expand the window enough, all the tabs are displayed in a line.

You can drag panels away from this group to customize the display. If you drop it on the background of PhAB's work area, it becomes a new panel group. If you drop it on another panel group, the panel joins that group. You're then free to resize the panel groups are you see fit. Depending on your choices in the AppBuilder Preferences Settings dialog, the arrangement of panels is saved with your application or for all your PhAB sessions.

Widget palette

The widget palette lets you add widgets to your application.



PhAB's widget palette.

If you close this panel, you can reopen it by choosing Show Templates from the Window menu, then selecting a pallet.

The predefined Widget palette is a collection of templates grouped into a folder called "Widgets". You can create your own templates by selecting **Widget→Define Template**. You can also customize the templates in the Widget palette by choosing Templates from the Edit menu. If a template is permanently removed from the Widget palette, you can always add it back by selecting Add Widget Class from the Edit menu. For more information on templates, see Adding a widget class in the Creating Widgets in PhAB chapter..

The widgets are arranged and color-coded by type. The names are optional; to hide or display them, right-click on the palette and choose the appropriate item from the pop-up menu.

To find out what widget a button represents if the widget names aren't displayed:

- Pause the pointer over it until a help balloon pops up.

  Or

- See the Widgets at a Glance appendix.

For information on using specific widget classes, see the Photon *Widget Reference*.

Modes (create vs select)

The widget palette has two modes:

Select mode	Lets you select existing widgets and modules in the work area.
Create mode	Lets you create new widgets.

Determining the mode

To find out which mode you're in:

- *Look at the widget palette*—If an icon button is pushed in, you're in create mode.
- *Look at the pointer*—If the pointer is a single-headed arrow when you move it into the work area, you're in select mode. If the pointer is anything else, you're in create mode.

Switching to create mode

To switch to create mode, click on any widget icon in the widget palette. You can now create one or more instances of that widget. For more information, see "Creating a widget" in the Creating Widgets in PhAB chapter.

Switching to select mode

To switch from create mode to select mode, do one of the following:

- Click the right mouse button in a module or on the background of the PhAB work area.

  Or:

- Click on the background of the PhAB work area. Note that this might not work if the selected template in the Widget palette creates a module. In this case, even if you click on the background of PhAB, the module specified by the template is created at the pointer position.

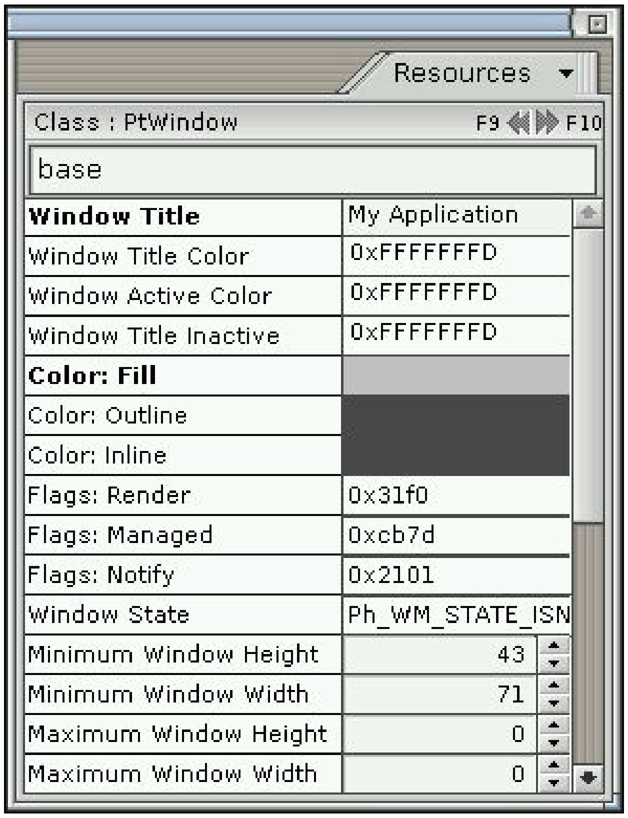
  Or:

- Click on the selected widget in the widget palette.

For most widgets, PhAB returns to select mode as soon as you've created a widget, but you can stay in create mode by pressing the Ctrl. This allows you to create multiple instances of the widget, one after another. Note that PhAB stays in create mode when you create some widgets (such as **`PtLine`**, **`PtPolygon`**, and **`PtPixel`**).

Resources panel

The Resources panel displays a list of resources for the selected widget or widgets. (If more than one widget is selected, this panel displays only the resources they have in common.) Here's an example:



Class : PtWindow

base

Window Title	My Application
Window Title Color	0xFFFFFFFD
Window Active Color	0xFFFFFFFD
Window Title Inactive	0xFFFFFFFD
Color: Fill	▨
Color: Outline	■
Color: Inline	■
Flags: Render	0x31f0
Flags: Managed	0xcb7d
Flags: Notify	0x2101
Window State	Ph_WM_STATE_ISN
Minimum Window Height	43
Minimum Window Width	71
Maximum Window Height	0
Maximum Window Width	0

The Resources panel.

If you close this panel, you can reopen it by choosing Show Resources from the Window menu.

It includes the following:

Widget class	The class of the selected widget.

Next and previous buttons	
	Let you navigate sequentially through widgets in the current module. These buttons also let you select multiple widgets or select widgets within a group. For more info, see the “Selecting widgets” section in the Creating Widgets in PhAB chapter.
Instance name	Lets you enter a unique instance name for the widget. For more information, see “Instance names” in the Creating Widgets in PhAB chapter.

You can change the value of a resource right in the control panel, or you can use the full-featured editor by clicking on the resource name. For more information, see the Editing Resources and Callbacks in PhAB chapter.

By default, the Resources and Callbacks control panels display resource labels descriptively. If you pause the pointer over a resource, the header manifest is displayed in a popup balloon.

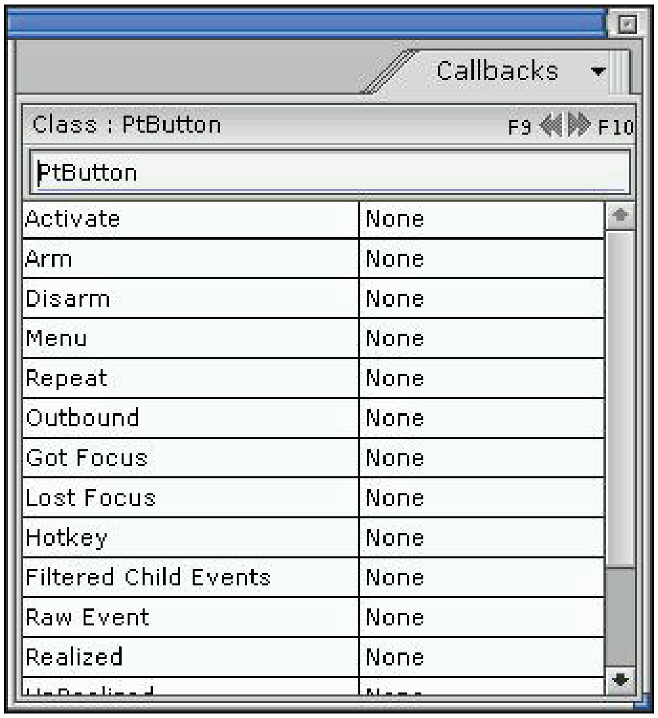
To have the labels displayed as the actual header manifests (convenient when writing code), open the Preferences dialog and change the setting in the Resource Names field. To open this dialog, choose Preferences from the Edit menu.

Now if you pause the pointer over a resource, the popup balloon displays the descriptive label. You can also copy a resource manifest or value by right clicking on the left column of the resources or callback panel. Select **Copy resource manifest to ph clipboard** to copy the resource manifest (for example, Pt_ARG_WINDOW_RENDER_FLAGS). Select **Copy resource value to ph clipboard** to copy the actual resource value.

The control panel doesn’t display all the resources for a widget. PhAB sets *Pt_ARG_AREA*, *Pt_ARG_DIM*, *Pt_ARG_EXTENT*, and *Pt_ARG_POS* automatically when you move or resize a widget. Some other resources are too complex to edit in PhAB.

Callbacks panel

The Callbacks panel displays a list of callback resources for the selected widget. You can use this panel only when you’ve selected a single widget. The widget must have a unique instance name. Here’s an example:



Class : PtButton

PtButton

Activate	None
Arm	None
Disarm	None
Menu	None
Repeat	None
Outbound	None
Got Focus	None
Lost Focus	None
Hotkey	None
Filtered Child Events	None
Raw Event	None
Realized	None

The Callbacks panel.

If you close this panel, you can reopen it by choosing Show Callbacks from the Window menu.

This panel, like the Resources panel, displays the widget class and instance names, and the next and previous buttons.

The left side of the list indicates the callback type. The right side displays:

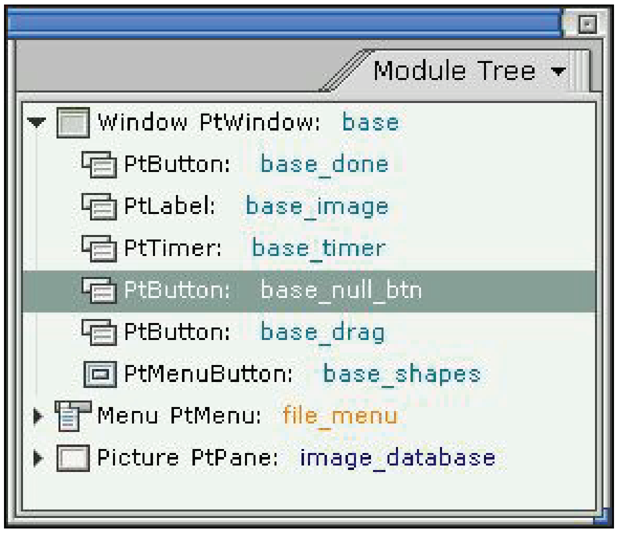
- "None" if there are no callbacks
- the callback type and name if there's one callback
- the number of callbacks if there's more than one.

You can right-click a callback name to copy its manifest or value. Pause the mouse over a callback name to view its manifest and a short description in a pop-up balloon.

To create a callback or to edit an existing one, click on the appropriate resource (for example, *Pt_CB_ACTIVATE*).

Module Tree panel

The Module Tree panel displays a hierarchical tree of the widgets in the current module. Here's an example:



The Module Tree panel.

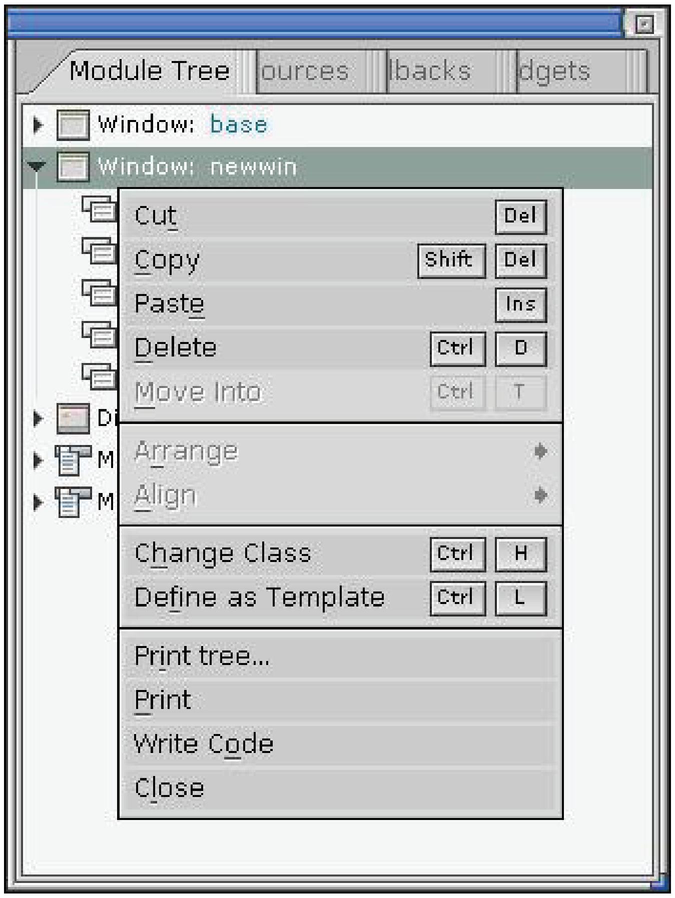
If you close this panel, you can reopen it by choosing Show Module Tree from the Window menu.

This panel makes it easy to:

- See the parent/child relationships of the module's widgets.
- Select a widget inside a group.
- Find a widget by name.
- Select a widget hidden underneath another widget.

To select a widget from the tree, click on the widget's name.

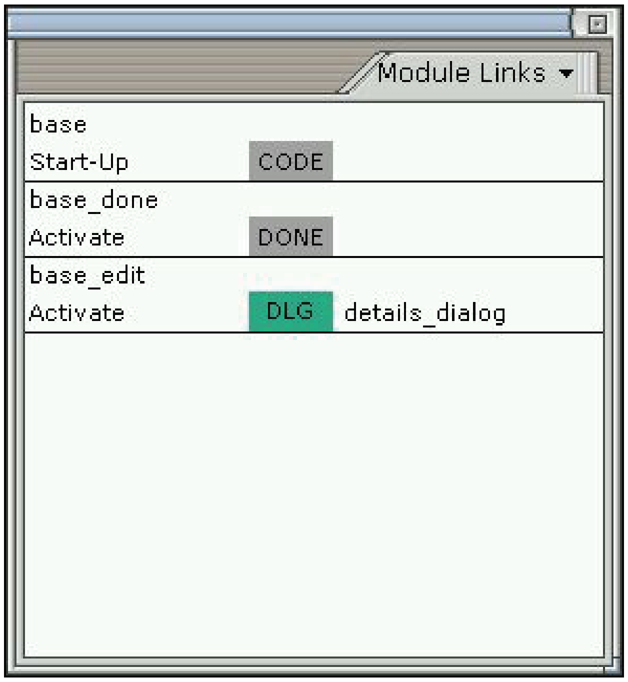
If you right-click on this panel, a menu appears:



The menu for the Module Tree panel.

Module Links panel

The Module Links panel displays a list of all link callbacks both to and from the current module. As you can see from the following example, the callbacks are displayed in a two-line format:



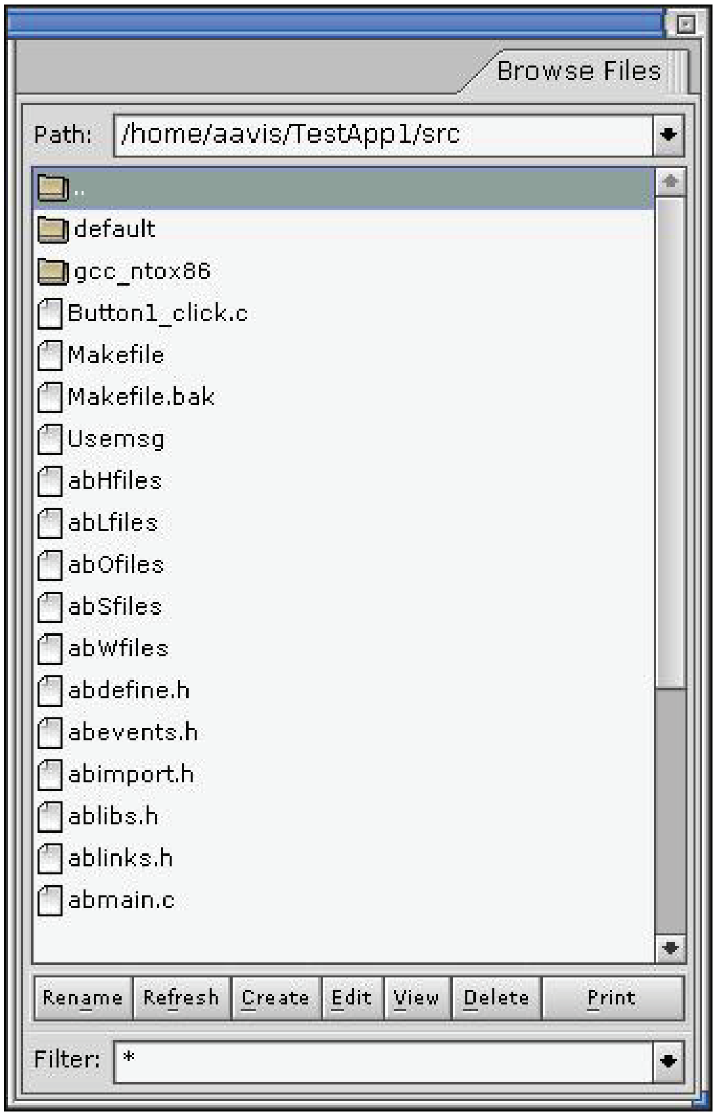
The Module Links panel.

To do this:	Click on the:
Select a widget	Instance name (e.g. **`base_file`**) in line 1
Edit a widget callback	Appropriate callback type (e.g. Arm) in line 2

If you close this panel, you can reopen it by choosing Module Links from the View menu.

Browse Files panel

The Browse Files panel is a file browser that you can use to view, edit, delete, print, create, and rename the files related to your application.

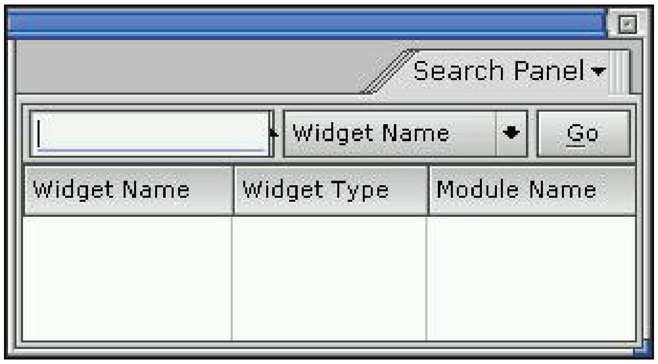


The Browse Files panel.

Use the **Filter** to select or enter a regular expression to filter the list of files. For example, **`*.c`** displays only files with a .c extension.

Search dialog

The Search dialog lets you search your application for widgets of a specified type, name, text resource, and so on.



The Search dialog.

Open this dialog by choosing Find from the Edit menu.

Just choose the category you want to find from the combo box and specify the pattern (which is case-sensitive) as appropriate:

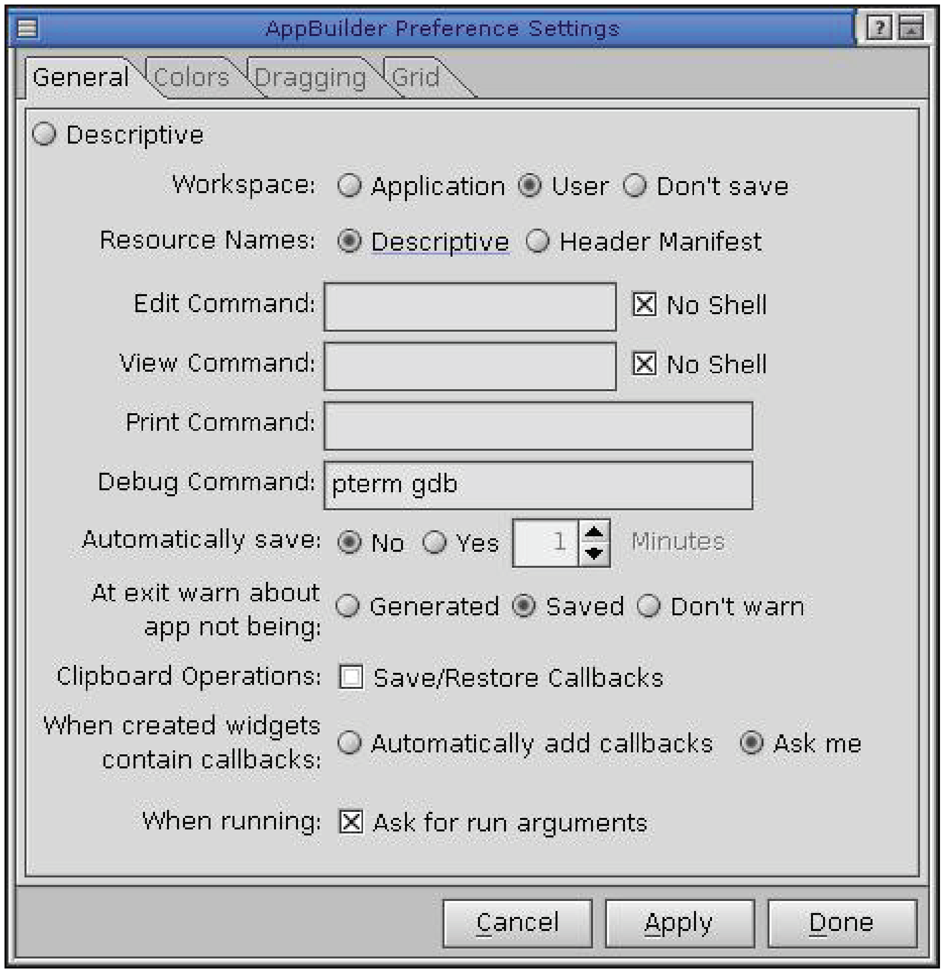
Widget Name	In the text field, type the exact name of the widget or a regular expression. For example, a value of **`my_button*`** matches all the widgets whose names begin with **`my_button`**.
Widget Type	Type a class name or a regular expression (e.g. **`PtScroll*`**), or use the combobox to select a widget class.
Widget Text	Type specific text or a regular expression to look for in the widgets' text resources.
Callback Type	Search for the widgets that have attached a callback of the type (Code, Done, and so on) selected from the pattern combo box.
Callback Function Name	Type a function name or a regular expression.
Callback Module Name	Type a module name or a regular expression. All the widgets that have a callback pointing to a module whose name matches the pattern are selected.

Next, press the Go button. The matching widgets are displayed in a list. Select entries from the list to select the actual widgets; the PhAB modules they're in are opened or made visible.

Customizing your PhAB environment

To customize PhAB to your preferences:

1 Choose Preferences from the Edit menu. You’ll see the Preference Settings dialog:



Descriptive

Workspace: Application [✓ User] Don't save

Resource Names: [✓ Descriptive] Header Manifest

Edit Command: [✓ No Shell]

View Command: [✓ No Shell]

Print Command:

Debug Command: pterm gdb

Automatically save: [✓ No] Yes 1 Minutes

At exit warn about app not being: Generated [✓ Saved] Don't warn

Clipboard Operations: Save/Restore Callbacks

When created widgets contain callbacks: Automatically add callbacks [✓ Ask me]

When running: [✓ Ask for run arguments]

Setting PhAB preferences.

2 Click on the tab for the group of settings you wish to change: General, Colors, Dragging, or Grid.

3 When you’re finished, click Done.

General preferences

You can set the following general preferences:

Workspace	Lets you decide whether to save your preferences on a per-application or per-user basis, or not at all.
Resource Names	By default, the Resources and Callbacks panels display resource labels descriptively. This field lets you display the labels as the

actual header manifests, which you may find helpful when writing code. Note, however, that manifests are long and take up more screen space.

If you pause the pointer over a resource, the label not displayed in the control panel is displayed in a popup balloon.

Edit Command	Lets you specify an external editor to edit source files.
View Command	Lets you specify the file viewer to use to view source files.
Print Command	Lets you specify the print command used to print a selected file (in the Project tab, for example).
Debug Command	Lets you specify the command used to debug your application.
Automatically save	Whether or not to save the application automatically, and how often.
Warnings on exiting	Whether or not to warn you when you exit PhAB without generating or saving your application.
Clipboard Operations	Lets you specify if callbacks should be saved along with widgets when copying from or cutting to the clipboard.
When created widgets contain callbacks	Lets you specify whether you're prompted to add callbacks when newly created widgets contain callbacks.
Ask for run arguments	Lets you specify whether PhAB prompts you to enter runtime arguments when you run an application from the Build menu.

Color preferences

You can set the following color preferences:

Resize Handle Non-Resizable Handle	If you choose a window background that makes it difficult to see resize handles, use these options to customize the color. (If you select a widget and the resize handles appear in the nonresize color, the widget can't be resized.)
Selection Color - First Item	The color for the first widget selected. Turn this feature on by selecting **View→Show Selection**.
Selection Color	The color for widgets selected after the first widget. Turn this feature on by selecting **View→Show Selection**.

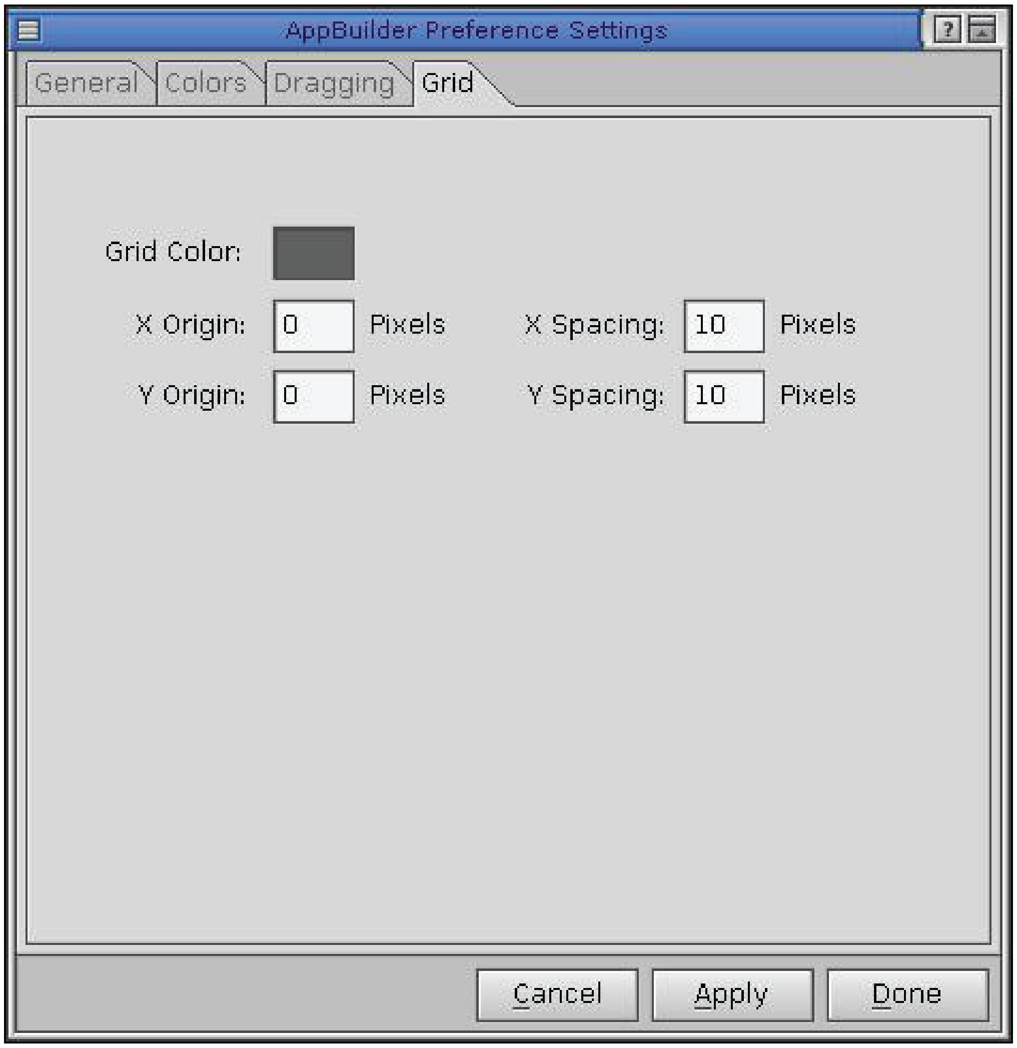
Dragging preferences

You can set the following dragging preferences:

Widget Module	Drag widgets and modules as outlines rather than as full objects.
Drag Damping Factor	The amount you must drag a widget or module before it moves. This factor helps avoid the annoyance of moving a widget when you really mean to select it.

Grid preferences

You can use a grid to position and size widgets.



Grid Color:

X Origin: 0 Pixels X Spacing: 10 Pixels

Y Origin: 0 Pixels Y Spacing: 10 Pixels

Grid Preferences.

You can:

- Choose the color of the grid.
- Specify the origin and spacing of the grid.

Chapter 4

Working with Applications

In this chapter. . .

This chapter describes working with an application as a whole in PhAB.

For information on running an application, see the Generating, Compiling, and Running Code chapter.

Creating an application

The way you create a new application depends on whether you're using PhAB from the IDE or standalone.

From the IDE:

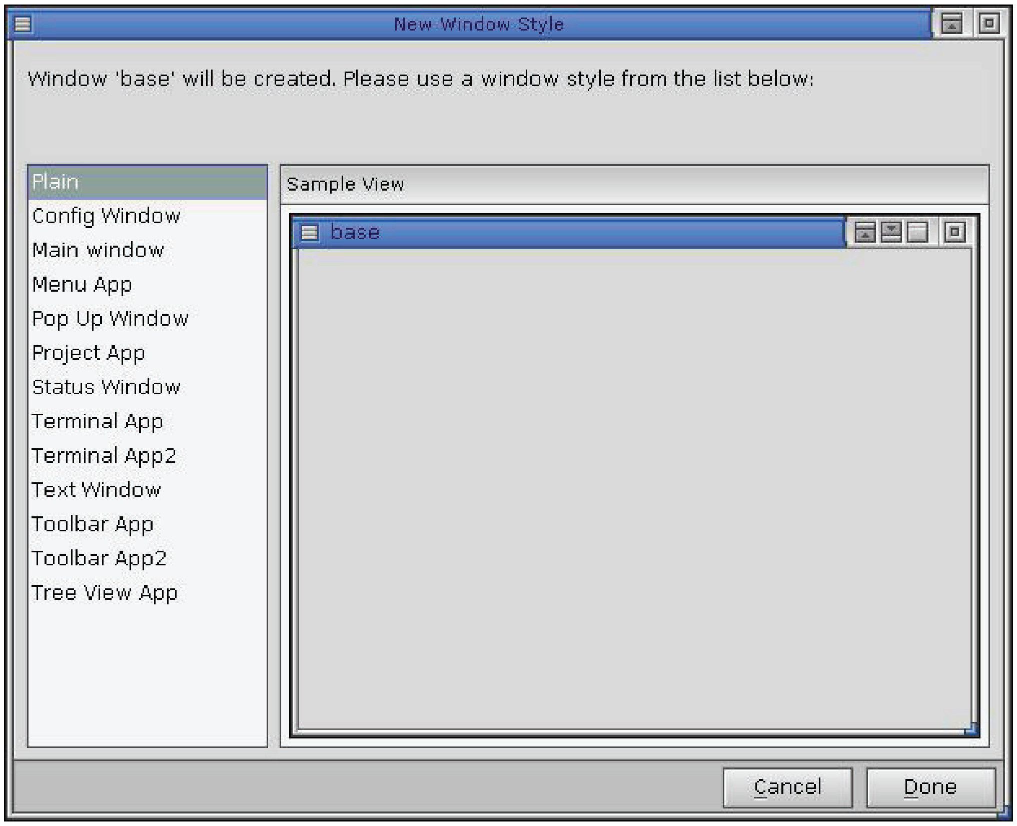
To create a new PhAB project, see "Creating a QNX Photon Appbuilder project" in the Developing Photon Applications chapter of the *IDE User's Guide.* When you create a new project, the IDE opens PhAB, and you see the **New Window Style** dialog where you can select the type of base window for your application.

Standalone PhAB:

To create a new application, choose **New** from the File menu or press Ctrl-N. If you're already working on an application, PhAB asks if you want to save any changes to that application before closing it.

Choosing a main window style

PhAB creates a new unnamed application that consists of a single main window named **`base`**. PhAB displays a dialog where you must choose the style of the base window:



Window 'base' will be created. Please use a window style from the list below:

Plain
Config Window
Main window
Menu App
Pop Up Window
Project App
Status Window
Terminal App
Terminal App2
Text Window
Toolbar App
Toolbar App2
Tree View App

Choosing the style of the base window.

After creating an application, you should:

- if you're using standalone PhAB, save it, giving it a name
- use the Project Properties dialog to:
  - specify a global header file
  - specify an initialization function
  - enable or disable command-line options
  - specify an icon for your application
  - specify your startup window, and other startup options

You should develop a naming convention for all the widgets, modules, functions, and so on. This will make managing your application easier.

Opening an application

The way you open an existing application depends on whether you're using PhAB in the IDE or standalone.

From the IDE

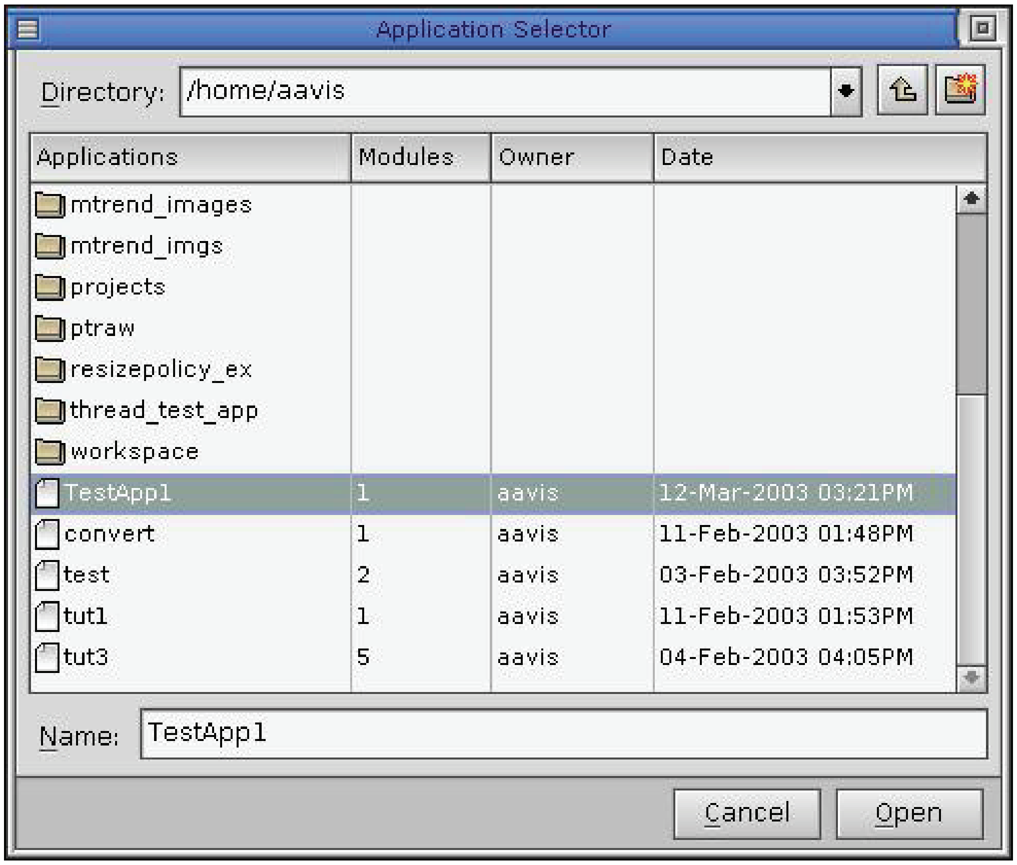
To open an existing PhAB project, see “Reopening PhAB” in the Developing Photon Applications chapter of the IDE *User’s Guide*.

Standalone PhAB

To open an existing application, choose **Open** from the File menu, press Ctrl-O, or choose **Open** from PhAB’s toolbar:

The Open button on PhAB’s toolbar.

You’ll see the application selector:



Directory: /home/aavis

Applications	Modules	Owner	Date
mtrend_images			
mtrend_imgs			
projects			
ptraw			
resizepolicy_ex			
thread_test_app			
workspace			
TestAppl	1	aavis	12-Mar-2003 03:21PM
convert	1	aavis	11-Feb-2003 01:48PM
test	2	aavis	03-Feb-2003 03:52PM
tut1	1	aavis	11-Feb-2003 01:53PM
tut3	5	aavis	04-Feb-2003 04:05PM

Name: TestAppl

Application Selector dialog.

If the application you want is in another directory, type the directory name in the Application Directory field, then press Enter, or browse to the directory. To enter a directory, double-click it. To go one level up in the directory structure, click the up arrow directory in the top-right corner of the dialog, or double-click the **..** folder in the file list.

To choose the application, do one of the following:

- Double-click the application.

  Or:

- Click the application, then press Enter or click Open Application.

  Or:

- Type the application's name, then press Enter or click Open Application.

If someone already has the application open, PhAB won't open it unless you started PhAB with the `-n` option.

If you're using NFS or SMB, you should start PhAB with the `-n` option because you can't lock files with either. For more information, see **`appbuilder`** in the QNX Neutrino *Utilities Reference*.

Saving an application

You can save your application in several ways, as described in the sections below.

To ensure the latest changes to your application are in effect, PhAB automatically saves your application whenever you regenerate or make your application code.

For information on using version-control software with PhAB applications, see "Version control" in the Generating, Compiling, and Running Code chapter.

How you save your application depends on whether you're running PhAB from the IDE or standalone.

From the IDE

When you run PhAB from the IDE, all file management is handled by the IDE. However, you can save the PhAB portion of the project by choosing **Save** from the File menu in PhAB. For information on managing projects in the IDE, see Developing C/C++ Programs in the IDE *User's Guide*.

From standalone PhAB

Naming or renaming an application

To save a new unnamed application or to save an application under another name or in a different directory:

1 Choose **Save As** from the File menu. You'll see the application selector dialog.

2 The dialog lists the contents of a directory. If you want to save your application in another directory, browse to the desired directory, or type the directory name in the **Application Directory** field and press Enter.

If you type a new directory name, it's saved. The next time you want to look in that directory, click the button to the right of the directory field and select the directory from the list.

3 Type the name of the application in the **Application Name** field.

4 Press Enter or click **Save Application**.

If you rename an application, you'll find that the name of the executable is also renamed if the project is an Eclipse-style project. However, if it's an older-style project, the application name isn't changed. This is because PhAB doesn't change the **`Makefile`**. In this case, to change the name of the executable:

- Edit the **`Makefile`** manually and change every occurrence of the executable's name.

  Or:

- If you haven't changed the **`Makefile`** since it was first generated, delete it and regenerate the application. See the Generating, Compiling, and Running Code chapter.

Saving an existing application

To save an existing application, choose Save from the File menu, press Ctrl-S, or select the Save button on PhAB's toolbar:

The Save button on PhAB's toolbar.

Overwriting an existing application

To overwrite an existing application:

1 Choose **Save As** from the File menu.

2 Do one of the following:

- Double-click the existing application.

  Or:

- Click the existing application, then press Enter or click **Save Application**.

Closing an application

When using PhAB from the IDE:

1. Close PhAB first using **Exit** from the File menu.
2. Close the project in the IDE.

To close an application in standalone PhAB, choose **Close** from the File menu. If you've made any changes but haven't saved your application, PhAB asks if you want to save it.

Specifying project properties

The Project Properties dialog lets you set up the typical operations that are performed when an application starts. You can:

- define startup windows and other initialization features
- set various project generation options
- define run options, such as command line arguments and language
- define build and debug options

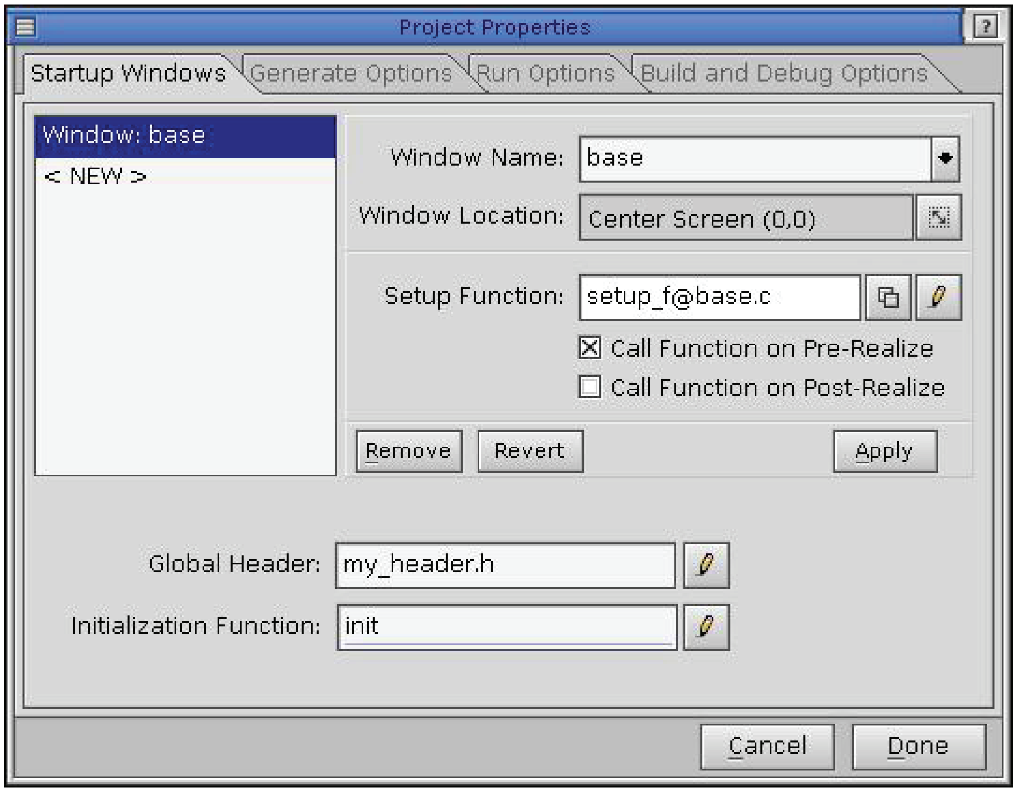
To open this dialog:

- Choose Project Properties from the Project menu.

  Or

- Press F2.

Here's the dialog, with some sample information filled in:



Window: base
< NEW >

Window Name: base
Window Location: Center Screen (0,0)
Setup Function: setup_f@base.c
☒ Call Function on Pre-Realize
☐ Call Function on Post-Realize

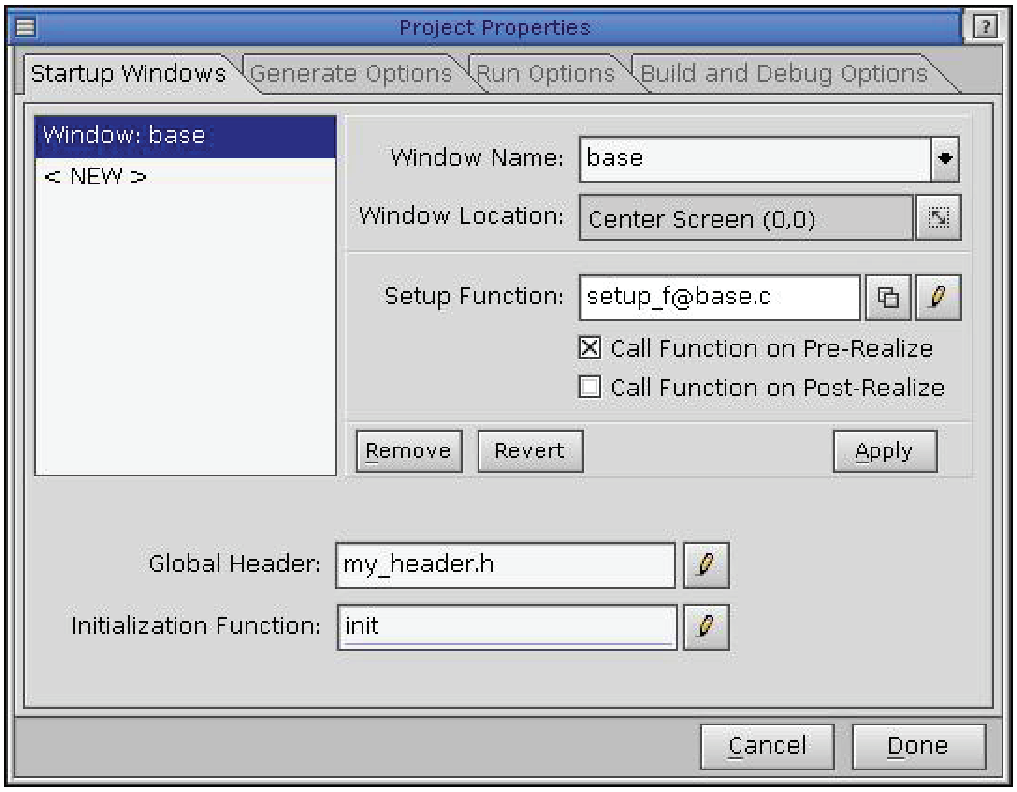
Global Header: my_header.h
Initialization Function: init

The Project Properties dialog.

Once you've made your changes, click Done.

Startup Windows tab

Here's the Startup Windows tab, with some sample information filled in:



Window: base
< NEW >

Window Name: base
Window Location: Center Screen (0,0)
Setup Function: setup_f@base.c
☒ Call Function on Pre-Realize
☐ Call Function on Post-Realize
Global Header: my_header.h
Initialization Function: init

The Project Properties dialog—Startup Windows tab.

You can use this tab to define:

- a startup window
- a global header
- an initialization function.

Define a startup window

When you first create an application, the mandatory base window is preset as the initial and only startup window. Using the Application Startup Information dialog, you can tell your application to:

- use another window as the initial startup window
- display several windows at startup
- use no startup windows.

The window that appears first in the Windows Opened/Startup list is the initial startup window:

- It's the first window to be displayed.

- It acts as the default parent window for all other windows and dialogs.
- Closing it causes the application to end.

Typically, the application's main window is the first created.

For each window in the startup list, you can specify information that's identical to the information used to create a module-type link callback, as described in the Editing Resources and Callbacks in PhAB chapter.

The information for each window includes:

Window Name	The name of the window module. To select from a list of existing windows, click the icon next to this field. If you specify the name of a module that doesn't exist, PhAB asks whether it should create that module.
Window Location	Where the window will appear; see "Positioning a module" in the Working with Modules chapter.
Setup Function	The function that's called when the window is realized (optional). To edit the function, click the icon next to this field. The buttons below the function name determine whether the setup function is called before the window is realized, after the window is realized, or both.
Apply	Applies any changes.
Revert	Restores the window information to its original state.
Remove	Deletes the selected window from the startup list.

Adding a startup window

To add a new window to the startup window list, click **<**NEW**>**, fill in the window information, and click Apply.

Modifying a startup window

To modify an existing startup window, select the window from the "Windows Opened/Startup" list, enter whatever changes are needed in the window information fields, and then click Apply.

Deleting a startup window

To delete an existing startup window, select the window from the "Windows Opened/Startup" list and click Remove.

Specifying a global header file

Most applications have a global header that's included in all source code files. If you plan to use a global header in your application, you should set up the header before PhAB generates any code. This lets PhAB automatically include the header in each file it generates.

To set up a global header:

1. Press F2 or choose Properties from the Project menu. You'll see the Project Properties dialog.
2. In the Global Header field, type the name of the header file you plan to use. You don't have to include the `.h` extension.

   For example, to set up a `globals.h` header file, you can simply enter: `globals`
3. To edit the header immediately, click the icon next to the Global Header field. You can edit the header only if you've named the application by saving it. The format of the header file is discussed in the Working with Code chapter.

If you specify the header after some code has been generated, you'll have to go back and manually add the header to the stub files that were previously generated.

Initialization function

Your application can include an initialization function that's called before any modules or widgets are created. In it you can initialize data, open widget databases, set up signal handlers, and so on. To set up an initialization function:

1. Press F2 or choose Properties from the Project menu. You'll see the Project Properties dialog.
2. In the Initialization Function field, type the name of the initialization function.

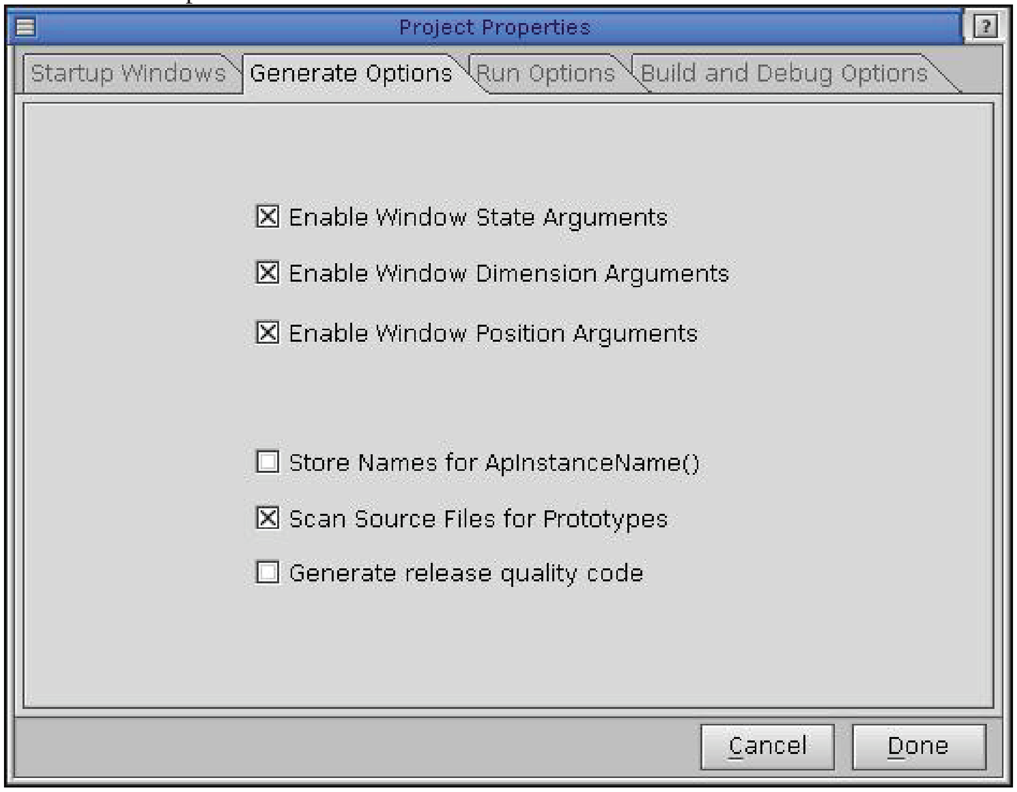
   When you specify a setup function, PhAB generates a stub function; for information on specifying the language (C or C++) and the filename, see "Function names and filenames" in the Working with Code chapter.
3. To edit the function immediately, click the icon next to the Initialization Function field. You can edit the function only if you've named the application by saving it. The prototype of this function is discussed in the Working with Code chapter.

Generate Options tab

The Generate Options tab lets you:

- set window arguments
- include instance names.

Here's an example of this tab:



[✓] Enable Window State Arguments
[✓] Enable Window Dimension Arguments
[✓] Enable Window Position Arguments
[ ] Store Names for ApInstanceName()
[✓] Scan Source Files for Prototypes
[ ] Generate release quality code

Build and Debug Options tab of the Project Properties dialog.

By default, all PhAB-generated applications have the following command-line options:

-h *height*[**%**]	The height of the window, in pixels, or as a percentage of the screen height if **%** is specified.
-s *server_name*	The name of the Photon server:

If server_name is:	This server is used:
node_path	*node_path*/**dev/photon**
fullpath	*fullpath*
relative_path	**/dev/***relative_path*

-w *width*[**%**]	The width of the window, in pixels, or as a percentage of the screen width if **%** is specified.

-x *position*[**%**][**r**]	The x coordinate of the upper-left corner of the window, in pixels, or as a percentage of screen width if **%** is specified. If **r** is specified, the coordinate is relative to the current console.
-y *position*[**%**][**r**]	The y coordinate of the upper-left corner of the window, in pixels, or as a percentage of screen height if **%** is specified. If **r** is specified, the coordinate is relative to the current console.
-Si|**m**|**n**	The initial state of the main window (iconified, maximized, or normal).

By default, all these options are enabled so that users can dynamically move or resize the application, or specify its initial state. For example, to start an application in console 4 (the center of the workspace), specify the command-line options:

```
-x100% -y100%
```

The PhAB API processes these options before it calls the initialization function; if you plan to add your own command-line options to an application, make sure you pick options that don't conflict with these. You should also code your option processing to handle and ignore these options. If you don't, you'll see errors on the console when you run the application. See the discussion on the initialization function in the Working with Code chapter.

If you don't want users to move or resize the application:

1. Press F2 or from the Project menu, choose Properties to open the Project Properties dialog.
2. Select the Generate Options tab.
3. Unset the toggle buttons for these options as required:
   - Enable Window State Arguments
   - Enable Window Dimension Arguments
   - Enable Window Position Arguments.

Other Generate options

You can set project-generation options for storing widget instance names, generating **proto.h**, and generating release-quality code.

Store Names for ApInstanceName()

PhAB converts your widgets' instance names into ABN_... manifests that you can use in your code to refer to your widgets by name. Check this option to include the instance-name text string in the widgets' memory.

Including instance names increases the amount of memory required to run your application.

Use *ApInstanceName()* to find this string for a widget—see the Photon *Library Reference* for more information.

Scan source files for prototypes

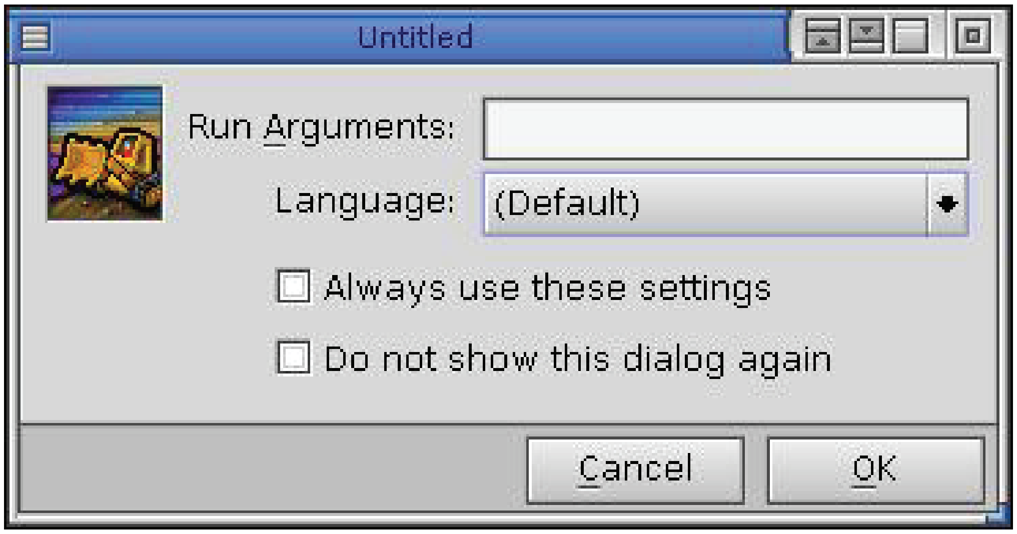
Indicates whether or not **`proto.h`** is to be generated — see “Generating function prototypes” in the Generating, Compiling, and Running Code chapter.

Generate release quality code

When your application has passed the development/debuging cycle and is ready for release, you can turn on this option and rebuild your executable. This will build an executable that requires less memory to run. The memory savings depend on the number of named widgets you have in your application, as PhAB optimizes the compiled code by turning ABN_ constants into macros.

Run options

Here’s an example of the Run Options tab:



Run Options tab on the Project Properties dialog.

Use the Run Options tab to specify:

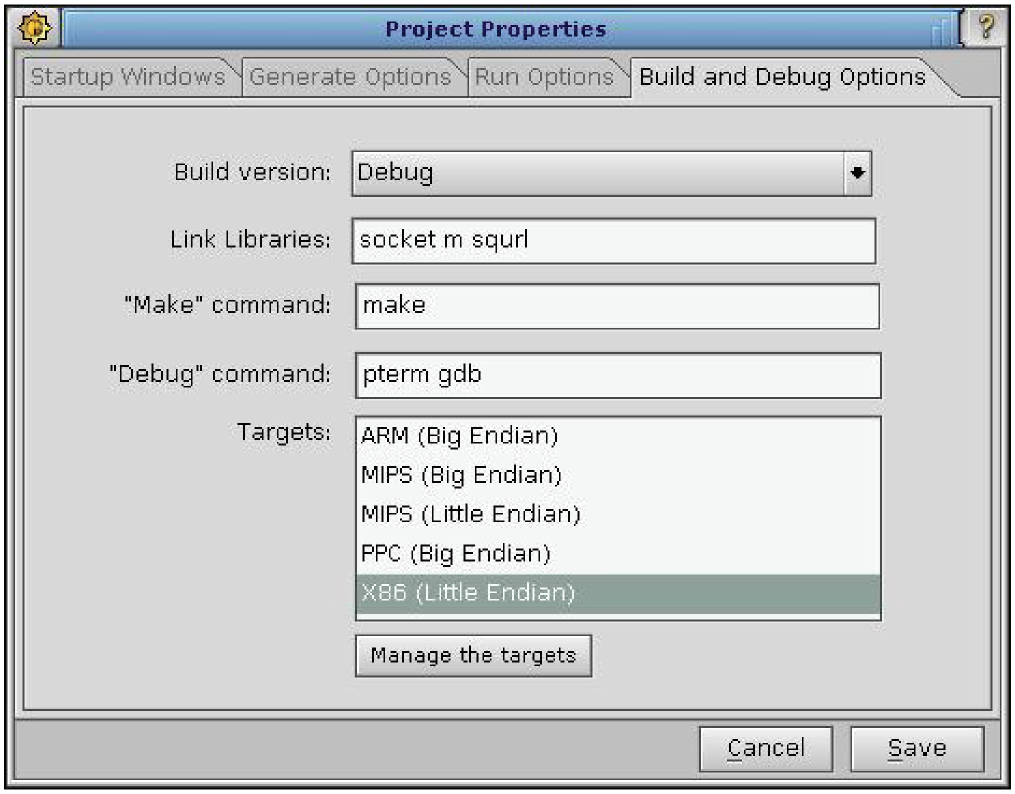
Run Arguments	Command line arguments used when PhAB runs your application
Language	The language PhAB uses when it runs your application

Project Large Icon
Project Small Icon

The large and small icons used for your project. Click the icon or the edit button to edit the icon in the pixmap editor. See the pixmap editor,as described in the Editing Resources and Callbacks in PhAB chapter.

Build and Debug options

This tab of the Project Properties dialog lets you specify options related to the build and debug process. Here is an example of this tab:



Build version: Debug
Link Libraries: socket m squrl
"Make" command: make
"Debug" command: pterm gdb
Targets:
ARM (Big Endian)
MIPS (Big Endian)
MIPS (Little Endian)
PPC (Big Endian)
X86 (Little Endian)

Build and Debug Options tab of the Project Properties dialog.

Build version — Set to Release to generate the executable without debug information, or Debug if you plan to debug your application.

Link Libraries — Use to specify additional libraries to be used when your application is linked. The **-l** is optional.

You can't specify link options in this field, except for the **-B** option, which specifies static or dynamic linking. For example, you could enter: **-Bstatic -lphexlib -Bdynamic**.

An exception to this rule is pre-6.3 projects which are not yet converted using **Project→Convert to Eclipse Project**. Those projects require the `-l` option, and take any linker command in this field.

The content of Link Libraries is saved in the **`indLfiles`** file and is included in the Makefile in the target directory.

Make command	Use to specify the command to build your application. The default is "make".
Debug Command	Use to specify the command to debug your application. The default is "pterm gdb", which launches a gdb session inside a new terminal window.
Targets	Use to select one or more targets for the current application. When you select Build or Rebuild All from the Build menu, only the selected targets are built. To add or delete targets, click **Manage Targets**.

Importing files

PhAB lets you import several types of files by using the Import Files item in the file menu:

- PhAB modules
- XBM Header File
- Graphics image files

The steps are the same for all types:

1. Choose **Import Files** from the File menu, then choose the appropriate type from the Import Files submenu. You'll see a file selector.
2. The file selector displays the available files of the specified type in the current directory.
3. To select a file, do one of the following:
   - double-click the file
   - click the file, then press Enter or click **Open**
   - type the file's name, then press Enter or click **Open**

Importing PhAB modules from other applications

When importing a PhAB module from another application, the file selector may display several modules. Each type of module has a different file extension; see "Module types" in the Working with Modules chapter.

Callbacks aren't imported, only the module and the widgets themselves. After importing the module, you can attach new application-specific callbacks.

You can copy and paste widgets or modules between two phab sessions in order to preserve the callbacks (you have to make sure the Clipboard Operations: Save/Restore callbacks option is set in the Preferences dialog). Or you can save the modules or widgets as templates, and choose to save the callbacks with the templates.

Normally, PhAB retains the instance name of each imported widget. However, if it detects a duplicate name, it changes that name to the widget-class name to avoid code-generation errors.

Importing XBM images

You can import X bitmap (XBM) files using the Import menu item. XBM image data is mostly found in header files (with a .h file extension) and in separate XBM bitmap files (with no file extension). To import an XBM image, use the Import menu item to open the header file containing the image data.

Importing graphics images

When importing graphics images, the file selector displays all files with the following extensions:

- **.bmp**
- **.tga**
- **.sgi**
- **.rgb**
- **.rgba**
- **.bw**
- **.png**
- **.gif**
- **.jpg**
- **.jpeg**
- **.pcx**

PhAB imports the graphic as a **`PtLabel`** into the currently selected module, and sets the widget's *Pt_ARG_LABEL_TYPE* resource to Pt_IMAGE. If you wish to edit the imported image, use the pixmap editor, as described in the Editing Resources and Callbacks in PhAB chapter.

The pixmap editor does not support 24-bit per pixel format JPEG images. If you wish to edit a JPEG using PhAB, you must first convert it to a palettized image (such as a .gif). You can do this by using an external image editing utility.

Exporting files

You can export the code used to create a module. To do this, select **File→Export**. A fragment of C code that creates the widgets in the currently selected module is written to the home directory in a file called **`module.code`**.

Chapter 5

Working with Modules

In this chapter. . .

Modules serve as containers to hold your application's widgets. Some modules, such as windows and dialogs, are actually Photon container-class widgets and let you place widgets directly inside them. Others, such as icons and menus, have either predefined widgets or a specialized editor for creating the widgets that they contain.

Module types

PhAB provides a number of types of modules, each with a specific usage. The module type is identified by:

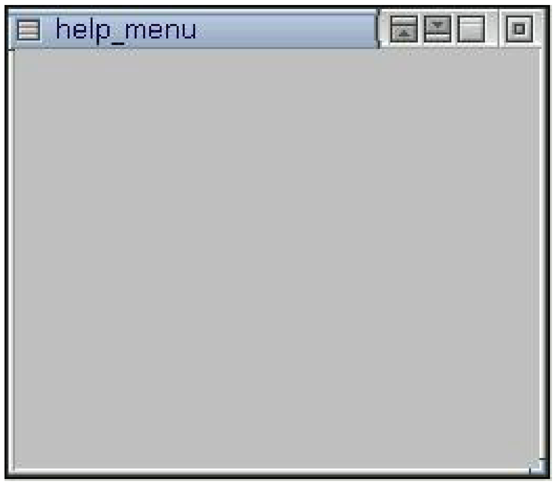
- the control panels if the module is selected
- the icon if the module is minimized
- the extension of the file that PhAB creates for the module when you generate your application's code.

CAUTION: Module files are binary; don't edit them with a text editor or you could damage them.

Module	Usage	Extension
Window	Major application activities	`.wgtw`
Dialog	Obtain additional information from the user	`.wgtd`
Menu	Multilevel text-only menus	`.wgtm`
Picture	Change the contents of an existing module, or create a widget database	`.wgtp`

Anatomy of a module

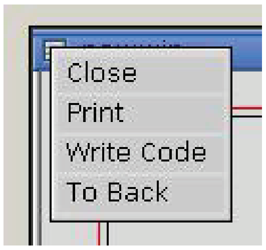
PhAB displays each module as a window in its work area. Like windows, modules have a set of controls in their frames.



Anatomy of a typical PhAB module.

Most modules include these elements:

Work menu button Brings up the module's Work menu:



The Work menu for a module.

The Work menu includes:

- Close — iconify the module.
- Print — print the module.
- Write Code — generate the code for the module.
- To Back — put this module behind all other modules in PhAB's work area.

Title bar Displays the module's instance name. To move a module, point to this bar and drag the pointer.

Collapse button
Minimize button
Close button These buttons iconify the module.

Test button (some modules only)

Like the Test item in the Work menu, this lets you switch the module into test mode.

When the render flags for a module's title bar are off, PhAB displays an area around the module that you can use to manipulate the module. This is useful if you are designing embedded applications, which usually have windows with no title bar and no decorations.

Selecting a module

To select a module that's in the PhAB work area:

- Click on the module's titlebar.

  Or

- If the module is iconified, double-click on its icon.

  Or

- Select the module in the Module Tree panel (this works for both iconified and noniconified modules).

Whichever method you choose, you'll see resize handles that indicate the module is selected.

How modules are saved

When you save your application, PhAB stores all the application's modules as files within the application's **`wgt`** directory. Each module is saved in its own file with a file extension based on the module's type. Later, when you "make" your application, PhAB binds all the modules into the binary executable. This makes the application a single free-standing program that you can distribute easily.

For more info, see "How application files are organized" in the Generating, Compiling, and Running Code chapter.

Changing module resources

When you select a module within PhAB, the Resources control panel changes to display the list of widget resources available for that module's class. Depending on which resources you change, you may see no immediate effect. All changes will take effect, however, when you run the application.

Because PhAB displays all modules as child windows within its work area, you can work with any number of modules at the same time.

Creating a new module

To create any new module, follow these simple steps:

1. From the Project menu, choose a command for the module type you want to create, which is one of:
   - **Add Window**
   - **Add Dialog**
   - **Add Menu**
   - **Add Picture Module**
2. For window and dialog modules, PhAB asks you to choose the style from a dialog that displays the available choices.

   For other types of modules, PhAB simply asks whether or not it should create the new module. Press Enter or click on Yes. You'll see the new module in PhAB's work area.
3. Click on Done.

PhAB creates the module for you. You can change the default instance name on the Resources control panel.

For more info on creating specific types of modules, see the sections on each type of module in this chapter.

You can also import modules from other PhAB applications. For more information, see "Importing files" in the Working with Applications chapter.

Deleting a module

To delete a module:

1. Select the module you want to delete.
2. Press the Delete key, or select **Edit→Delete**.

Deleting a module doesn't delete the module's file; it just removes the name from the list. Any callbacks belonging to the module or its children are deleted.

Iconifying modules

PhAB's work area lets you work on several application modules at once. You can iconify modules to organize your work area. To reduce any module in the work area to an icon:

- Double-click on the module's Work menu button (upper-left corner of the module).

  Or:

- Click on the Work menu button and choose Close.

  Or:

- Click on the Minimize button in the module's titlebar.

Once it's iconified, the module positions itself at the bottom of the work area. You can drag it anywhere in the work area, so (for example) you can group commonly used or related icons.

Displaying modules at run time

Your application needs a way to make modules appear when you run it. You can:

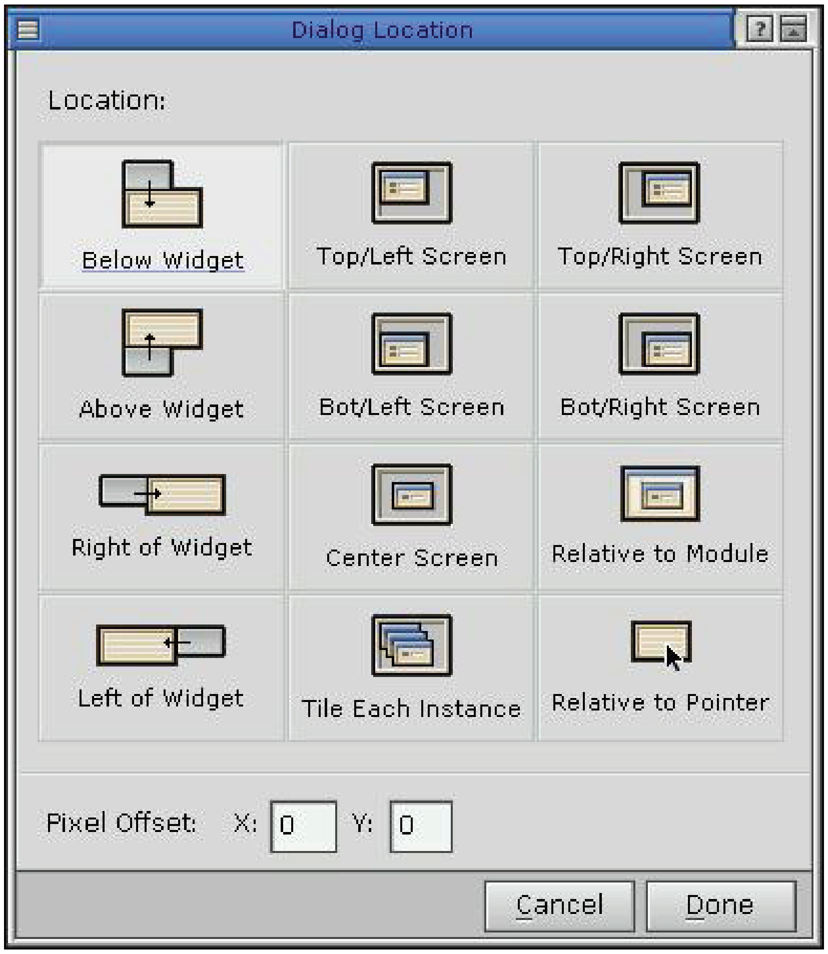
- Create a widget that uses a callback to display the module. For example, you can create a **`PtButton`** with a module-type link callback that displays the module. For more information, see "Editing callbacks" in the Editing Resources and Callbacks in PhAB chapter.
- Use an *internal link* to create the module in your application's code. See the Accessing PhAB Modules from Code chapter.

Positioning a module

You can specify where a module will display when you create a link callback from a widget to that module. To do this, you use the location dialog.

To open the Location dialog and select a module's location:

1. When creating or editing a link callback to a module, click on the Location field or on the icon to the right of the field. You'll see a list of locations:



Location dialog.

For windows and dialogs, the default location is Default (0,0), which places the window at the next available position defined by the Window Manager. The default location for a menu module is Below Widget.

2 Click on the location you want.

3 You can also specify x and y offsets. For example, if you set the location to the bottom-right corner and set the *x* offset to -100, the window will be displayed so that its bottom-right corner is 100 pixels to the left of the bottom-right corner of the screen.

If you choose Default as the location, the offsets are ignored.

4 Click on Done.

Finding lost modules and icons

To find a lost module or icon:

- Choose Cascade from the Window menu. PhAB cascades all open modules in the PhAB work area.
- Choose Arrange Icons from the Window menu. PhAB rearranges all existing icons along the bottom of the work area.

Window modules

Widget class	**File extension**	**Widget creation**
`PtWindow`	`.wgtw`	Directly from the widget palette

Typically, you use *window modules* for your application's major activities. Since most applications use a window module for their main window, PhAB automatically generates a window module named **`base`** when you first create any application. It also presets the application's startup information to make the base window open when the application starts up. (See "Specifying your project properties" in the Working with Applications chapter.)

The icon for a Window module.

Window modules can support multiple instances. That is, two or more copies of the same window module can be displayed at the same time. As a result, you should keep track of each window's instance pointer, which is generated when you create the window. That way, you'll always know which window you're dealing with when you process callbacks. For more information, see "Handling multiple instances of a window" in the Working with Code chapter.

Even though your application's base window is a window module, you usually display it only once, at startup. So unless your application needs to display more than one copy of the base window at the same time, you don't have to keep track of the base window's instance pointer.

For an example of code for handling multiple instances of window modules, see "Creating Windows" in the Tutorials chapter.

Resizing a window module

When you set a window module's size in PhAB, that's the size it will be when you run the application.

Dialog modules

Widget class	File extension	Widget creation
`PtWindow`	`.wgtd`	Directly from the widget palette

Dialog modules let you obtain additional information from the user. Typically, you use this information to carry out a particular command or task.

The icon for a Dialog module.

Most dialog modules include the following buttons:

- Done—allows users to indicate that they've finished entering information
- Cancel or Close—allows users to close the dialog without responding

From PhAB's perspective, dialog modules are almost identical to window modules, with one important difference—a dialog module can have only one active instance. So if you invoke a dialog that's already open, the PhAB API simply brings the existing instance of the dialog to the front of the screen. This behavior fits with the nature of a dialog—you rarely want to get the same information twice. If for any reason you need a dialog that can support multiple instances, use a window module.

Limiting a dialog to a single instance makes callback handling simpler since you can use the widget manifests that PhAB generates to access the widgets within the dialog. For more info, see the discussion on instance names in the Creating Widgets in PhAB chapter.

Resizing a dialog module

When you set a dialog module's size in PhAB, that's the size it will be when you run the application.

Predefined dialogs

The Photon libraries include convenience functions that define various handy dialogs:

ApError()	Display an error message dialog

PtAlert()	Display a message and request a response
PtFileSelection()	Create a file-selection dialog
PtFontSelection()	Create a font-selection dialog
PtMessageBox()	Pop up a message box
PtNotice()	Display a message and wait for acknowledgment
PtPassword()	Prompt for a password
PtPrintPropSelect()	Change the printing options for a selected printer via a modal dialog
PtPrintSelect()	Display a custom modal dialog for selecting print options
PtPrintSelection()	Display a modal dialog for selecting print options
PtPrompt()	Display a message and get textual input from the user

Menu modules

Widget class	File extension	Widget creation
`PtMenu`	`.wgtm`	Special editor

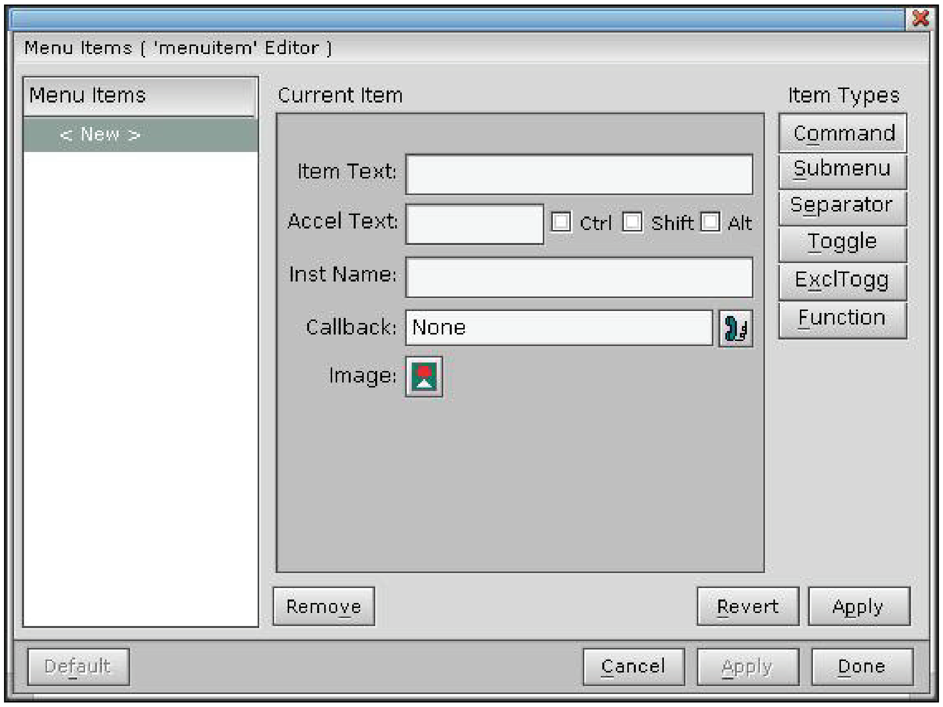
A *menu module* provides a multilevel text-only menu. Unlike most other modules, a menu module doesn't let you create widgets directly inside it. Instead, you use PhAB's menu editor to create the menu's items.

The icon for a Menu module.

Opening the menu editor

To open the menu editor:

1. Select a menu module.
2. Click on Menu Items in the Resources control panel. PhAB displays the menu editor:



PhAB's Menu editor.

In the upper-right corner you'll see buttons that represent the types of menu items you can create:

- Command—invokes a PhAB callback.
- Submenu—displays a child menu.
- Separator—provides a line between other menu items.
- Toggle or "ExclTogg" (exclusive toggle) —changes or displays an application state.
- Function—specifies an application function that can dynamically add menu items to the menu.

These buttons are at the bottom of the dialog:

When you want to:	Use this button:
Apply any changes and close the editor	Done
Apply any changes and continue editing the menu	Apply
Cancel any changes made since you opened the editor	Cancel

Specifying instance names

To create any command or toggle menu item (that is, any item that can invoke a callback), you must enter a unique instance name—PhAB enforces this. The instance name lets you access the menu item from within your application code.

When PhAB generates the code for your application, it generates an ABN_... constant for each menu item that requires it. You use this constant with the menu-item related API functions, *ApModifyItemAccel()*, *ApModifyItemState()*, and *ApModifyItemText()*.

For example, let's say a menu item isn't available when the user clicks on the widget that brings up the menu. Using the instance name, you can dim that item before displaying the menu. For more information, see "Initializing menus" in the Working with Code chapter.

Creating hotkeys and shortcuts

To help the user select a menu item more quickly, you can:

- provide a keyboard shortcut that selects the item
- provide a hotkey that directly invokes the command that the item represents, even when the menu isn't visible

A keyboard shortcut works only when the menu is currently visible. A hotkey, on the other hand, should work whether the menu is visible or not.

Creating a keyboard shortcut is easy. When you're entering the Item Text, simply place "**`&`**" in front of the character that will act as the shortcut. For example, let's say you're creating a "Save As" item. You could enter **`Save &As`**, which will underline the "A." When the menu opens, the user can press either **`A`** or **`a`** to invoke the callback associated with "Save As".

Creating a hotkey takes a bit more work, but it's still easy to do. First, you want to make sure that the hotkey accelerator appears next to the menu item when the menu is displayed. To do this, use the Accel Text field. For example, let's say the hotkey accelerator for a "Save" menu item will be Ctrl-S. In that case, you would type **`S`** in the Accel Text field and check the Ctrl toggle button.

Next, you need to create a hotkey callback for Ctrl-S. Since the menu might not be created when the user presses Ctrl-S, you can't attach the hotkey callback to the menu or to the menu item. Rather, you must attach the callback to the application's main module, which is usually the base window module. When you specify the hotkey callback's function, use the same function you defined for the menu item's callback.

If for some reason you need to differentiate between the two methods of invoking the callback, look at the callback's reason code. Hotkeys always have a reason code of Pt_CB_HOTKEY.

For more info on creating hotkey callbacks, see "Hotkey callbacks" in the Editing Resources and Callbacks in PhAB chapter.

Resizing a menu module

Feel free to resize a menu module to make it more readable or take up less space. When you run the application, the actual size of the **`PtMenu`** widget will be determined by the menu items.

Creating command items

A command item lets you invoke application code or display a module.

Field	**Description**
Item Text	The text that will be displayed
Accel Text	The hotkey to invoke the command
Inst Name	The name used within the application code
Callback	The function that will be called when the item is selected
Image	The icon to use for the menu item

To create a command item:

1. Click on <NEW>.
2. Click on the Command button in the upper-right corner.
3. In the Item Text field, enter the item's text. To create a shortcut key, place "**`&`**" in front of the character that will act as the shortcut.

   For example, let's say you enter **`&File`**. In that case, the user can select the item by pressing F.
4. In the Inst Name field, enter the instance name you'll use.
5. If you plan to have a hotkey callback for this item, enter the hotkey string and modifier key (for example, **`Ctrl-S`**) in the Accel Text field. The hotkey is displayed in the menu as a reminder to the user.
6. Add a PhAB callback by clicking on the Callback icon:

   For more info on creating a callback, see "Editing callbacks" in the Editing Resources and Callbacks in PhAB chapter.
7. Add an image, if appropriate.
8. Click on Apply to add the item to the menu.

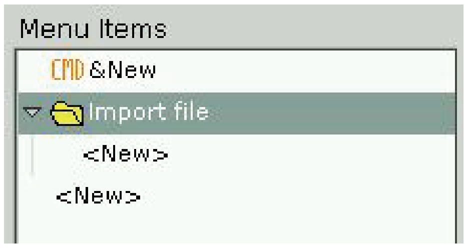
Creating submenu items

A submenu item lets you create another menu level.

Field	Description
Item Text	The text that will be displayed
Inst Name	The name used within the application code

To create a submenu item:

1. Click on <NEW>.
2. Click on the Submenu button in the upper-right corner.
3. In the Item Text field, type the name of the submenu. To create a keyboard shortcut, place "&" in front of the character that will act as the shortcut (just like command items, above).
4. Click on Apply.
5. The Menu Items list displays the submenu:



6. You can now add items to the submenu by selecting <NEW> in the submenu.

Creating separator items

A separator item lets you add a line between menu items. You'll find this item type handy for creating logical groupings of menu items.

To create a menu separator:

1. Click on <NEW>.
2. Click on the Separator button in the upper-right corner.
3. Click on Apply.

Creating toggle items

A toggle item lets you change or display an application state, which can be either on or off.

Field	Description
Item Text	The text that will be displayed
Accel Text	The hotkey to invoke the command
Inst Name	The name used within the application code
Callback	The function that will be called when the item is selected
Image	The icon to use for the menu item

To create a toggle item:

1. Click on <NEW>, then click on the Toggle button.
2. Follow the same procedure used to create command items.

Creating function items

A function item lets you specify an application function that dynamically adds menu items to the menu at runtime. For example, you could use a function item in a File menu to display the last three files the user worked on.

The PhAB library invokes the specified function as the menu is built. The dynamically created menu items appear where you've positioned the function item in the menu.

Field	Description
Function	The function that will be called

To create a function item:

1. Click on <NEW>, then click on the Function button.
2. In the Function field, enter the name of the application function that will dynamically add menu items to the menu.

   If you specify this function name, PhAB will generate a stub function; for information on specifying the language (C or C++) and the filename, see "Function names and filenames" in the Working with Code chapter.
3. You can edit the function right away by clicking on the button to the right of the function name.
4. Click on Apply.

For information on the application function, see "Generating menu items" in the Working with Code chapter.

Moving menu items

The Menu Items scrolling list lets you move a menu item to a new position in the menu.

Let's say you want to move an item named Browse so it appears just before an item named Edit. You would:

1. Drag the Browse item until its outline is directly over Edit.
2. Release the mouse button. The Browse item appears in its new position.

Using a menu module

Once you've created a menu module, you need a way to make your application display it. Typically, you do the following:

1. Create a **`PtMenuBar`** at the top of a window.
2. Add a **`PtMenuButton`** to the menu bar, giving it an appropriate instance name and text string.
3. Add a module-link callback to the menu button's *Pt_CB_ARM* callback list.

You could add the callback to the *Pt_CB_ACTIVATE* list, but adding it to *Pt_CB_ARM* allows the user to access it in two ways:

- by pressing the left mouse button on the menu button widget, dragging to highlight a menu item, and releasing to select it. This is known as the press-drag-release (PDR) method.
- by clicking on the menu, and then clicking on a menu item

If you use an Activate callback, the user can only use the second method.

4. Have the callback display the menu module. See "Module callbacks" in the Editing Resources and Callbacks in PhAB chapter.
5. If you need to initialize the menu whenever it's displayed, specify a setup function for it. See "Initializing menus" in the Working with Code chapter.

If you want your menu to appear when you press the right mouse button while pointing at a widget, you'll need to use an internal link. For more information, see the Accessing PhAB Modules from Code chapter — there's even an example.

Picture modules

Using a *picture module*, you can change the contents of an existing module or create a convenient database of widgets. You always display a picture inside a container-class widget or another module, such as a window or dialog.

The icon for a Picture module.

Like windows, picture modules support multiple instances. So you should keep track of the instance pointer of the container that each picture is placed into. That way, you'll always know which picture you're dealing with when you process callbacks.

If you're sure that your application will use only one instance of the picture at any given point, you don't have to keep track of instance pointers. Instead, you can use PhAB-generated manifests to access the picture's widgets.

Displaying a picture

You always access picture modules from within your application code. To access a picture, you must create an *internal link* to it. This tells PhAB to generate a manifest that you can use with PhAB's API functions, such as *ApCreateModule()*, to access the picture.

For more information, see the Accessing PhAB Modules from Code chapter.

Using pictures as widget databases

You can use a picture module as a *widget database*. A widget database contains predefined widgets that you can copy at any time into a window, dialog, or container.

When using a widget database, you don't have worry about handling multiple instances since the generated PhAB widget manifests don't apply to widget databases: each widget you create is a new instance. The instance pointer is returned to you when you create the widget using *ApCreateWidget()*. You'll need to keep track of this pointer manually if you need to access the widget in the future.

For more info, see "Widget databases" in the Accessing PhAB Modules from Code chapter.

Resizing a picture module

It doesn't matter how large or small you make a picture module. That's because it has no associated widget class. Only the widgets inside the module are used.

Chapter 6

Creating Widgets in PhAB

In this chapter. . .

Once you've created or opened an application, you'll probably want to add, delete, and modify widgets. This chapter describes how to work with widgets.

For information on using specific widget classes, see:

- the Widgets at a Glance appendix in this guide
- the Photon *Widget Reference*

Since widgets inherit a lot of behavior from their parent classes, you should make yourself familiar with the fundamental classes: **`PtWidget`**, **`PtBasic`**, **`PtContainer`**, and so on.

Types of widgets

There are two major types of widgets:

- *Container widgets*, such as **`PtWindow`** and **`PtScrollContainer`**
- *Noncontainer widgets*, such as **`PtButton`** and **`PtText`**.

Container-class widgets can contain other widgets—including other containers. Widgets placed inside a container are known as *child widgets*; the hierarchy resulting from this nesting is called the *widget family*. Container widgets can look after sizing and positioning their children, as described in the Geometry Management chapter.

When working with container-class widgets in PhAB, remember the following:

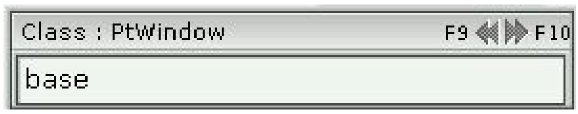
- If you move a container, all the container's child widgets also move.
- If you position the pointer inside a container when creating a new widget, that widget is placed hierarchically within the container.
- If you wish to use the bounding-box method to select widgets in a container, you must:
  - Press Alt before you start the bounding box.
  - Start the bounding box within the container.

  For more info, see "Selecting widgets" in this chapter.

Instance names

If your program has to interact with a widget, that widget must have a unique *instance name*. Using this name, PhAB generates a global variable and a manifest that let you easily access the widget from within your code.

To view or edit a widget's instance name, use the Widget Instance Name field at the top of the Resources or Callbacks control panel:



base

Editing a widget's instance name.

- A widget's instance name is used to make several C variables, so it can include only letters, digits and underscores. PhAB doesn't let you use any other characters. An instance name can be no longer than 64 characters.
- You should develop a naming convention for all the widgets in your application — it will make large applications more manageable.

You can optionally include the instance name in the widget's memory. See "Other Generate options" in the Working with Applications chapter.

Default instance name

When you create a widget, PhAB automatically gives it a default instance name. Typically, this default name is the widget's class name. For example, if you create a **`PtButton`**-class widget, the Resources and Callbacks control panels display **`PtButton`** as the instance name.

If a widget simply serves as a label or window decoration, it doesn't have to be accessed from within your application code. So you should tell PhAB to ignore the widget's instance name during code generation. To do this:

- Leave the instance name equivalent to the class name (that is, leave the default alone).

  Or:

- Provide a blank instance name.

When to assign a unique name

You should give a widget a unique name if:

- the widget needs to have a callback attached
- the application needs to change the widget by setting a resource
- the application needs to extract information from the widget

To keep the number of global variables to a minimum, don't give a widget a unique name unless you really need to access the widget from within your application. If you've given a widget a name and later decide you don't need the name, just change it back to the widget's class name or blank it out.

Instance names and translations

As described in the chapter on International Language Support, you'll need an instance name for every text string in your application's user interface. These instance names aren't needed in your code.

To indicate that an instance name isn't required for code generation, start the name with the **@** character. PhAB recognizes such a name when generating the text language database, but skips over it when generating code.

If you don't want to create a unique instance name for a string that's to be translated, specify a single **@** character for the instance name; PhAB appends an internal sequence number to the end.

If you don't want to create unique instance names, but you want to organize the text for translation (say by modules), you can give the strings the same instance name, and PhAB will append a sequence number to it. For example, if you assign an instance name of **`@label`** to several strings, PhAB generates **`@label`**, **`@label0`**, **`@label1`**, ... as instance names.

Duplicate names

PhAB resets the instance name of a widget back to the widget class name if it detects a duplicate name when you:

- copy and paste a widget (see "Clipboard")
- import a widget from another application (see "Importing PhAB modules from other applications" in the Working with Applications chapter)
- duplicate a widget (see "Duplicating widgets and containers").

Creating a widget

To create a widget:

1. Click on widget-palette icon for the type of widget you want to create (see the Widgets at a Glance appendix to identify the widget-palette icons).
2. Move the pointer to where you want to create the widget. The pointer changes to show you what to do next:
   - If the pointer is a crosshair and you're creating a **`PtPolygon`** or **`PtBezier`** widget, hold down the mouse button and drag the pointer until the line goes

where you want it to go. To add points, you must start the next point on top of the last. To close a polygon, place the last point on top of the first.

- If the pointer is a crosshair and you're creating any other type of widget, click the mouse button.
- If the pointer is a two-headed arrow, hold down the mouse button and drag the pointer until the widget is the size you want.

Widgets snap to the grid if it's enabled. See "Grid preferences" in the chapter on PhAB's environment.

To improve your application's performance, avoid overlapping widgets that are frequently updated.

You can also create a widget by dragging its icon from the widget palette to the Module Tree control panel. Where you drop the icon determines the widget's place in the family hierarchy.

Creating several widgets

Once you've created a widget, you're returned to select mode. To stay in create mode so you can create several widgets of the same type:

1. Press and hold down Ctrl.
2. Create as many widgets as you want.
3. Release Ctrl.

Canceling create mode

To cancel create mode without creating a widget:

- Click anywhere outside a module.

  Or:
- Click the right mouse button in a module.

Selecting widgets

When PhAB is in select mode, the pointer appears as an arrow. To put PhAB into select mode:

- Click anywhere outside a module.

  Or:
- Click the right mouse button in a module.

  Or:
- Click on the selected widget in the widget palette.

A single widget

To select a single widget, you can:

- Point and click

  Or:

- Use the next and previous buttons in the Resources or Callbacks control panel.
- Use the Module Tree control panel.

These methods are described below.

Point-and-click method

To select a single widget using point and click:

1. Make sure you're in select mode.
2. Click on the widget, using the left mouse button. Resize handles appear around the widget.

To select the parent of a widget, hold down Shift-Alt and click on the widget. This is a handy way to select a **`PtDivider`** or **`PtToolbar`**.

You must press Shift and then Alt for this method to work.

Control-panel methods

The Next and Previous buttons in the Resources and Callbacks control panels let you select any widget in the current module.

To select the:	**Click on:**	**Or press:**
Previous widget in the current module	⏪	F9
Next widget in the current module	⏩	F10

The Module Tree control panel displays a tree of all the widgets in the module. Using this tree, you can:

- select a widget inside a group
- find a widget by name
- select a widget hidden underneath another widget.

To select a widget from the tree, click on the widget's name.

Multiple widgets

To select multiple widgets, you can:

- Use a bounding box

  Or:

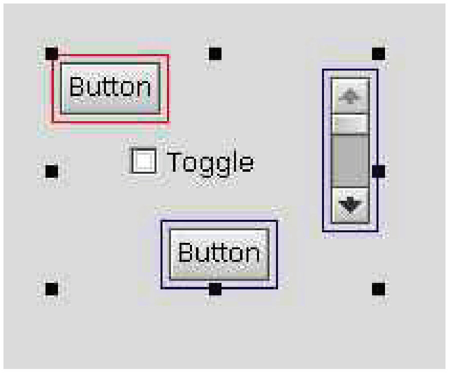
- Use "Shift and click"

  Or:

- Use the control panels.

When you select two or more widgets, the Resources control panel displays only the resources that those widgets have in common. Editing any of these resources affects *all* the selected widgets.

PhAB uses two colors to show selected items if the Show Selection option is selected in the View menu. The colors can be customized in the Preferences dialog.



Multiple selected widgets.

In the example above, the toggle widget is not selected, it just happens to be in the same area as the selected widgets. The widget highlighted by red is the first widget in the selection. The widgets highlighted by blue are the rest of the widgets in the selection. If you use an align or match command, the first selected widget is the source widget.

Using a bounding box

A bounding box lets you select several widgets all at once:

1. Position the pointer above and to the left of the widgets you want to select.
2. If the widgets belong to a container such as `PtBkgd`, make sure the pointer is within the container, then hold down the Alt key.
3. Hold down the left mouse button, then drag the pointer down to the right. You'll see an outline "grow" on the screen.

4 When all the widgets are within the outline, release the mouse button. You'll see resize handles appear around the area defined by the selected widgets.

Using "Shift and click"

To add or remove a widget from the current list of selected widgets, hold down Shift and click on the widget. This is also known as the *extended selection method*.

If the widget *isn't* already selected, it's added to the list. If the widget *is* already selected, it's removed from the list.

The above methods for selecting multiple widgets work only for widgets at the same hierarchical level. For example, let's say you've just selected two buttons inside a window. You can't extend that selection to include a button that's inside a pane.

Using the control panels

To select multiple widgets, using the Resources or Callbacks control panel's Next and Previous buttons:

1 Hold down Shift.

2 Click on the Next button.

Every time you click, PhAB adds the next widget in the current module to your selection.

To remove the last widget from the current list of selected widgets:

1 Hold down Shift.

2 Click on the Previous button.

Every time you click, PhAB removes another widget.

Widgets within a group

To select a widget inside a group, you can use the next and previous buttons in the Resources or Callbacks control panel, or use the Module Tree control panel.

Using the Module Tree panel

To select a single widget within a group, using the Module Tree control panel:

1 Switch to the Module Tree control panel.

2 Find the group in the tree and click on the widget's name.

3 Shift-click to select additional widgets, if you want.

4 To edit the widget, switch to the Resources or Callbacks control panel.

Using the Next and Previous buttons

To select one or more widgets within a group, using the Next and Previous buttons:

1. Click on any widget within the group to select the entire group.
2. Click on the Resources or Callbacks control panel's Next button (or press F10) until the widget you want is selected.
3. To select additional widgets, press Shift, then click again on the Next button.
4. You can now edit the widgets' resources or callbacks.

Hidden widgets

If you can't find a widget (it may be hidden behind another widget or is outside the boundaries of its container), do the following:

1. Use the Next and Previous buttons in the Resources or Callbacks control panel.
2. Select the widget from the Module Tree control panel.
3. Use the Search dialog. Select **Edit→Find** to open this dialog.
4. If the widget seems to be outside the current boundaries of its container, bring it back into view by using the X and Y fields in PhAB's toolbars.

For more information on the toolbars and control panels, see the chapter on PhAB's environment.

Aligning widgets

You can align several widgets to another widget or to their parent container.

For simple alignments, select the Align icon from PhAB's toolbar:

and then choose the alignment from the pop-up menu.

For more complicated alignment options, bring up the Align Widgets dialog by:

- Choosing the Align icon from PhAB's toolbar, and then choosing Alignment Tool from the menu

  Or:
- Choosing Align from the Widget menu, and then choosing Alignment Tool from the submenu

  Or:
- Pressing Ctrl-A.

To another widget

When you use this method to align widgets, the widgets are aligned to the first widget you select, which is highlighted differently from the other widgets in the selection if the Show Selection option is set in the View menu. To align to another widget:

1. Select the first widget.
2. Using the "Shift and click" selection method, select the remaining widgets. (This method is described in "Selecting widgets. ")
3. For simple alignments, select the Align icon from PhAB's toolbar and make a choice from the menu.
4. For more complicated alignment options, bring up the Align Widgets dialog. Choose one or more alignment options, then click on the Align button. *Don't* click on an Align to Container button.

To a parent container

To align widgets to their parent container:

1. Select one or more widgets in any order.
2. Bring up the Align Widgets dialog, choose your alignment options, then click on the appropriate Align to Container button.

If you choose both vertical and horizontal options, be sure to click on both Align to Container buttons.

3. Click on the Align button.

When aligning widgets to a container you may want the widgets to retain their relative positions to each other. To do this:

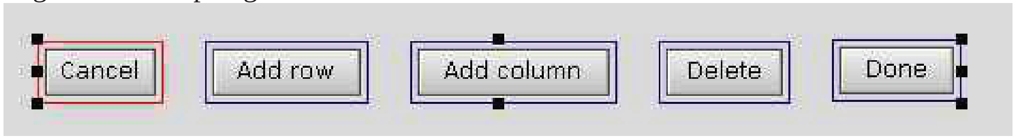
1. Group the widgets together (see the section "Aligning widgets using groups" in the Geometry Management chapter).
2. Align the widgets.
3. Optionally, break the group apart.

Distributing widgets

You can quickly and easily distribute widgets horizontally or vertically, to evenly space them out on the GUI. To do this:

1. Select the widgets you want to distribute.
2. From the Widget menu, select **Distribute→Horizontally** or **Distribute→Vertically**.

In the following example, several buttons have been distributed horizontally and aligned to the top edge:



Distributed widgets.

Common User Access (CUA) and handling focus

Common User Access (CUA) is a standard that defines how a user can change the keyboard focus. A widget is *focusable* if it can be given focus by pressing CUA keys or by calling a focus function.

Changing focus with the keyboard

The following keys move focus only to focusable widgets:

To go to the:	**Press:**
Next widget	Tab
Previous widget	Shift-Tab
First widget in the next container	Ctrl-Tab
Last widget in the previous container	Ctrl-Shift-Tab

For information on specifying the order in which the widgets are traversed, see the section "Ordering widgets" in this chapter.

Controlling focus

Use the following *Pt_ARG_FLAGS* flags to control focus for a widget:

Pt_GETS_FOCUS	Make the widget focusable.
Pt_FOCUS_RENDER	Make the widget give a visual indication that it has focus.

In addition, use the following *Pt_ARG_CONTAINER_FLAGS* flags to control focus for a container:

Pt_BLOCK_CUA_FOCUS	Prevent the CUA keys from being used to enter the container. However, if the user clicks inside the container, or a focus function gives it focus, the CUA keys can then be used.

Pt_ENABLE_CUA	Give the parent widget the chance to control whether or not a child container handles the CUA keys: • If this flag is set, the widget's code handles the CUA keys. • If it isn't set, the CUA keys are passed up the widget family until an ancestor is found with this flag set. This ancestor (if found) handles the keys.
Pt_ENABLE_CUA_ARROWS	The same as Pt_ENABLE_CUA, but it applies only to the arrow keys.

Focus callbacks

All descendants of the **`PtBasic`** widget have the following callback resources:

- *Pt_CB_GOT_FOCUS* —called when the widget gets focus
- *Pt_CB_LOST_FOCUS* —called when the widget loses focus. The widget can even refuse to relinquish focus (for example, if you type invalid data in a text widget).

`PtMultiText` and **`PtText`** have special versions of these callbacks.

For more information, see the *Widget Reference*.

Focus-handling functions

The functions listed below deal with focus. They're described in the Photon *Library Reference*.

These functions don't actually change which widget has focus; they tell you where focus can go:

PtFindFocusChild()
: Find the closest focusable child widget

PtFindFocusNextFrom()
: Find the next widget that can get focus

PtFindFocusPrevFrom()
: Find the previous widget that can get focus

You can use these routines to determine which widget has focus:

PtContainerFindFocus()
: Find the currently focused widget in the same family hierarchy as a widget

PtIsFocused()	Determine to what degree a widget is focused

You can use these routines to give focus to a widget:

PtContainerFocusNext()
: Give focus to the next Pt_GETS_FOCUS widget

PtContainerFocusPrev()
: Give focus to the previous Pt_GETS_FOCUS widget

PtContainerGiveFocus() or *PtGiveFocus()*
: Give focus to a widget — these routines are identical.

PtContainerNullFocus()
: Nullify the focus of a widget

PtGlobalFocusNext()
: Give focus to next widget

PtGlobalFocusNextContainer()
: Give focus to another container's widget

PtGlobalFocusNextFrom()
: Give focus to the next widget behind the specified widget

PtGlobalFocusPrev()
: Give focus to the previous widget

PtGlobalFocusPrevContainer()
: Give focus to a widget in the previous container

PtGlobalFocusPrevFrom()
: Give focus to widget previous to the specified widget

Ordering widgets

In PhAB, each widget exists in front of or behind other widgets. This is known as the *widget order*, and you can see it when you overlap several widgets. The order of the widgets dictates how you can use the CUA keys to move between widgets.

If you're not using PhAB, the widget order is the order in which the widgets are created. To change the order, see "Ordering widgets" in the Managing Widgets in Application Code chapter.

To view the widget order, do one of the following:

- Use the Module Tree control panel. The widgets for each container are listed from back to front.

  Or:

- Use Test mode and press Tab repeatedly to check the focus order.

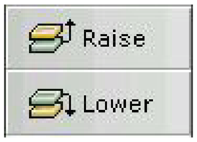
The easiest way to reorder the widgets is to use the Module Tree control panel — just drag the widgets around until they're in the order you want.

You can also use the Shift-select method to reorder the widgets:

1. Using the extended ("Shift and click") selection method, select the widgets in the order you want. (This selection method is described in "Selecting widgets.")
2. Do one of the following:
   - choose To Front or To Back from the Edit menu
   - press Ctrl-F or Ctrl-B
   - click on one of these icons:

   PhAB places the widgets in the order you selected them.

You can also select one or more widgets and then use the Raise and Lower icons to change the widget order:



Dragging widgets

Dragging a widget is the easiest way to move a widget in most situations since it's quick and fairly accurate:

1. Select the widgets.
2. Point to one of the selected widgets, press down the mouse button, then drag the widgets to the new position.

   If you want to drag the widgets horizontally, vertically, or diagonally, hold down the Alt while dragging.

To drag a container horizontally, vertically, or diagonally, press Alt *after* pressing the mouse button. (Pressing Alt before the mouse button selects widgets within the container.)

3 Release the mouse button. Widgets snap to the grid if it's enabled — see "Grid preferences " in the chapter on PhAB's environment.

To cancel a drag operation, press Esc before releasing the mouse button.

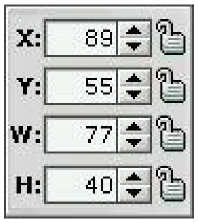
To move the parent container of a widget, hold down Shift-Alt and drag the child.

Another way to drag a widget is to hold down Shift while selecting and draging one of the widget's resize handles. This method may help when you're moving smaller widgets that are harder to select.

Widgets may "disappear" if you move them beyond the boundaries of their container. If this happens, use the Previous and Next buttons in the Resources or Callbacks control panel to select the widgets, then use the X and Y fields in PhAB's toolbar to bring the widgets back into view.

If you find that you're unintentionally dragging widgets when you're just trying to select them, consider:

- Setting the damping factor, as described in "Dragging preferences," below
- Locking the widgets' coordinates and/or size in PhAB's toolbar:



  For more information, see "Toolbars" in the chapter on PhAB's Environment.

Dragging preferences

There are several preferences that you can set for dragging (see the "Customizing your PhAB environment" section in the chapter on PhAB's environment):

- Dragging has a damping factor that determines how far you must drag before the widget moves. The default is 4 pixels.
- You can drag widgets either as an outline or as full widgets.
- You can drag modules either as an outline or as full modules.

Setting a widget’s x and y coordinates

To place one or more widgets at specific coordinates:

1. Select the widgets.
2. Type the coordinates in the x and y fields in PhAB’s toolbars, then press Enter. For more information, see the chapter on PhAB’s environment.

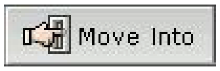
Transferring widgets between containers

To move one or more widgets directly from one container or module to another:

1. Select the widgets.
2. Do one of the following:
   - Choose Move Into from the Edit menu.

     Or:
   - Press Ctrl-T.

     Or:
   - Click on the Move Into icon in PhAB’s toolbar:

3. Move the pointer into the other container and click the mouse button.

Resizing widgets and modules

When you select a widget or module, you’ll see its height and width—including any borders and margins—displayed in the toolbar’s H and W fields. (These values are maintained by the *Pt_ARG_DIM* resource; see the description of **`PtWidget`** in the *Widget Reference*.)

To resize a selected widget, do one of the following:

- Drag one of the widget’s resize handles.

  Or:
- Click on the height or width field in PhAB’s toolbars, type in a new value, then press Enter. For more information, see the chapter on PhAB’s environment.

  Or:
- Use the nudge tool in the toolbar.

If a module is in Test mode, you can’t resize it or its widgets.

If you have trouble seeing a widget’s resize handles because of the background color you’ve chosen, you can change the resize-handle color. For more info, see “Customizing your PhAB environment” in the PhAB Environment chapter.

Clipboard

PhAB’s clipboard lets you cut, copy, and paste widgets and modules between PhAB instances. You can’t use this clipboard with other applications. To use the clipboard, open two PhAB instances, copy or cut something into clipboard in one instance, and then paste it into the other instance.

You’ll find the clipboard helpful for these reasons:

- It saves you from creating large numbers of widgets or modules from scratch.
- It helps you create applications whose widgets look and behave consistently with each other.
- You can also copy callbacks associated with copied widgets, saving you time. To include callbacks when you copy widgets, you need to set the Save/Restore Callbacks option on the General tab of the Preferences dialog.

Cutting and copying

A cut operation removes the currently selected widgets from their module and places them in the clipboard. A copy operation copies the currently selected widgets to the clipboard without removing them from their module.

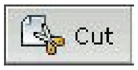
Whenever you cut or copy, PhAB deletes any widgets already in the clipboard.

To cut or copy one or more widgets:

1 Select the widgets.

2 To *cut*, do one of the following:

- Choose Cut from the Edit menu.
  Or:
- Press Ctrl-X.
  Or:
- Click on the Cut icon in PhAB’s toolbar:



3 To *copy*, do one of the following:

- Choose Copy from the Edit menu.

  Or:
- Press Ctrl-C.

  Or:
- Click on the Copy icon in the toolbar:

- If you want to move a widget to another container but retain its callbacks, you need to set the Save/Restore Callbacks option on the General tab of the Preferences dialog. See “Transferring widgets between containers” in this chapter.
- The Edit menu also contains a Delete command. This command permanently removes widgets without copying them to the clipboard.

Pasting

A paste operation copies widgets from the clipboard into a module.

To paste the contents of the clipboard:

1. Make sure you’re in Select mode.
2. Do one of the following:
   - Choose Paste from the Edit menu.

     Or:
   - Press Ctrl-V.

     Or:
   - Click on the Paste icon in the toolbar:

3. Point to where you’d like the clipboard objects to appear, then click the mouse.

- Instance names are normally maintained when you paste. But if PhAB detects a duplicate name, it ensures that the instance name is unique.
- Because the clipboard state is saved between PhAB applications, you can cut widgets from one PhAB application and paste them into another.

Duplicating widgets and containers

Here's a quick and easy way to duplicate a widget or container (it's much simpler than using the clipboard):

1. Press and hold down Ctrl.
2. Point to the widget or container, hold down the left mouse button, then drag the pointer to where you'd like the new widget to appear.
3. Release Ctrl and the mouse button.

If you want to duplicate many widgets at once:

1. Select the widgets you want to duplicate.
2. Press and hold down Ctrl.
3. Point to one of the widgets in the selection and drag to a new position.
4. Release Ctrl and the mouse button.

If you duplicate a container, all its children are duplicated as well.

Duplicating is achieved using a copy to clipboard operation and paste from clipboard, that are done internally. Therefore the rules for clipboard operations about instance names and callback are also valid for duplicating widgets.

- You can duplicate only one container or widget at a time. If you duplicate a container, all its children are duplicated as well.
- The instance names of the new widgets are reset to be the widget class name.
- Callbacks aren't duplicated.

Deleting widgets or modules

To permanently remove one or more selected widgets, or to delete a module:

1. Select the widgets or module.
2. Choose Delete from the Edit menu or press Delete.

If you want put the widgets or module somewhere else, you should cut them, not delete them. For more information, see the section on the clipboard in this chapter.

Matching widget resources and callbacks

You can copy the resources or callbacks from one widget to one or more other widgets by using the Matching feature. This features lets you quickly and easily create several widgets with similar appearances and behavior.

To use the matching feature:

1. Select the widget that is the source for the resources or callbacks.
2. Select one or more destination widgets.

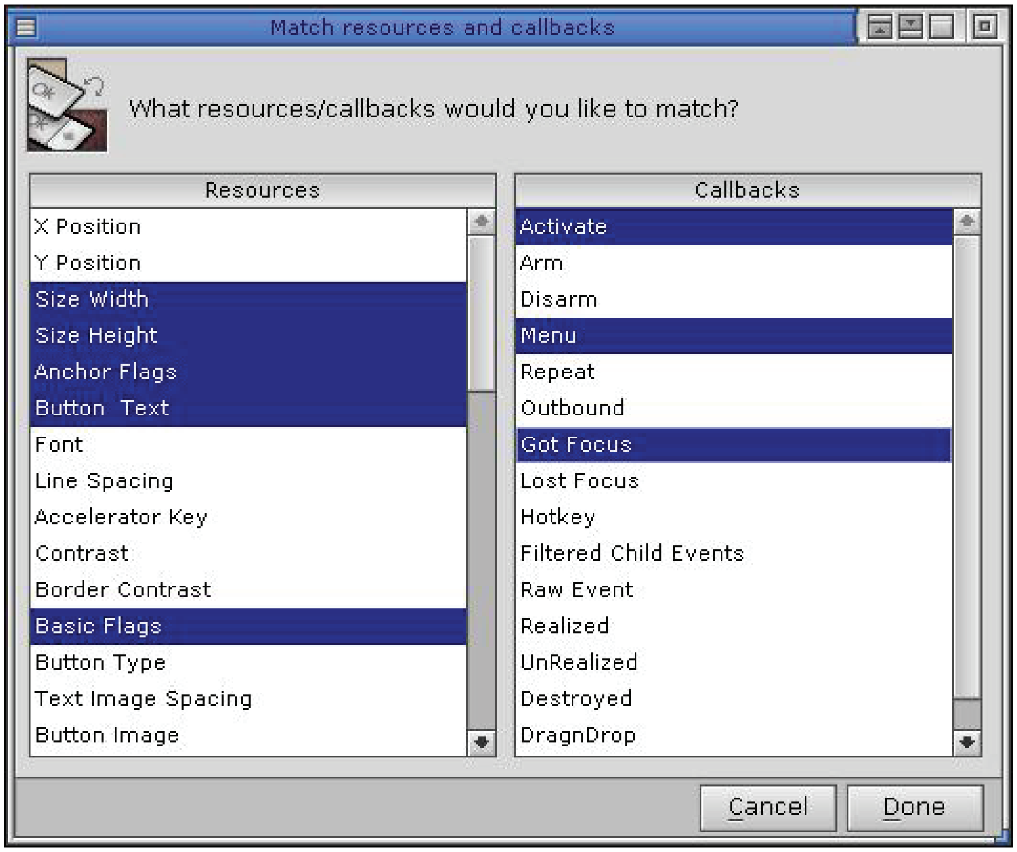
If you enable the Show Selection option in the View menu, the widget selected first highlighted with a different color than other selected widgets.

---

3. Select a match command from the Widget menu:
   - **Match Height** — sets the height of the destination widgets to the height of the source widget.
   - **Match Width** — sets the width of the destination widgets to the width of the source widget.
   - **Match Resources** — sets all the resources of the destination widgets to the source widget's. Only the resources that are common between the destination and the source are changed; unique resources are unaffected.

Geometry related resources — Pt_ARG_POS, Pt_ARG_DIM, Pt_ARG_AREA — are not copied.

   - **Match Callbacks** — sets all the callbacks of the destination widgets to the source widget's. Only the callbacks that are common between the destination and the source are changed; unique callbacks are unaffected
   - **Match Advanced** — displays the Match resources and callbacks dialog, which lets you choose exactly the resources and callbacks you want to copy from the source widget to the destination widgets.



Match resources and callbacks dialog.

Importing graphic files

PhAB lets you import several kinds of graphic files into your application. For more information, see “Importing files” in the Working with Applications chapter.

PhAB doesn’t export graphic files directly. That’s because any imported file is saved with the module in a PhAB-specific format.

The Pixmap editor (described in the Editing Resources and Callbacks in PhAB chapter) also lets you import graphics: select the widget you want to add the image to, edit its image, and choose the pixmap editor’s Import button.

Changing a widget’s class

You can change the class of a widget by selecting it and then choosing Change Class from the Widget menu. Choose the new class from the pop-up list, and then click the Change class button.

The resources and callbacks that are compatible with the new widget class are kept, along with their values. For example, if you decide that a **PtMultitext** better suits your needs than a **PtButton**, you can select the button, open the Change class dialog by right-clicking on the widget, or right-clicking in the Module tree or by choosing Change Class from the Widget menu. The widget's position, size, *Pt_ARG_TEXT_STRING*, and all the other resources common to the old and new classes are kept.

When you change a widget's class, some resources and callbacks might be deleted. Before proceeding, a dialog displays the number of resources and callbacks that will be removed. You have a chance to cancel the operation.

A container that has children (such as a **PtPanel** with some widgets inside it) can be converted this way, but the list of possible new classes you can choose from is restricted to compatible container classes. For instance a **PtPane** with a button inside can be changed into a **PtBkgd**, but not into a **PtList** or **PtTree**. An empty **PtTree** or any other empty container can be changed into anything, including into non-container widgets. A **PtTree** that has a child (a **PtDivider**) can be changed into a container widget.

Templates

A *template* is a customized widget, group or hierarchy of widgets that you want to use as the basis for other widgets. Templates are useful when you want to create many widgets that look and behave alike. PhAB automatically loads your templates, so you can easily create instances of your widgets in any application.

You can build and save your own collection of template widgets. Templates can be customized buttons, labels, sliders, backgrounds, standard dialogs and windows. You can include callbacks in templates.

Customized templates are not any different from the standard templates in the Widgets palette. In fact, when you create a template, you save it as a personal template (visible only to you) or as a global PhAB template (visible in all PhAB instances).

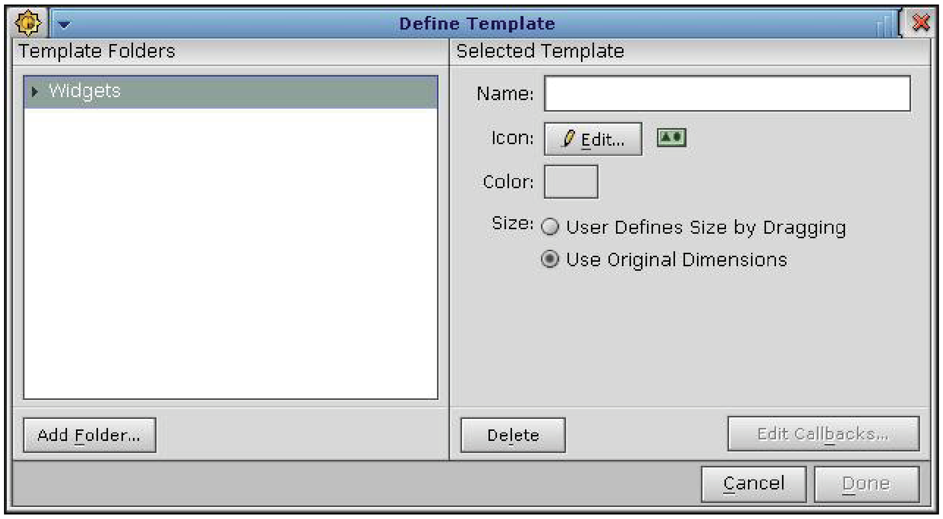
For an example of creating a template, see "Editing Resources" in the Tutorials chapter.

Creating templates

To create a template:

1. Create and edit the widget or widgets as required.
2. With the widget(s) selected, choose Define Template from the Widget menu or from the menu that appears when you right-click on the Module Tree control panel or on the module.

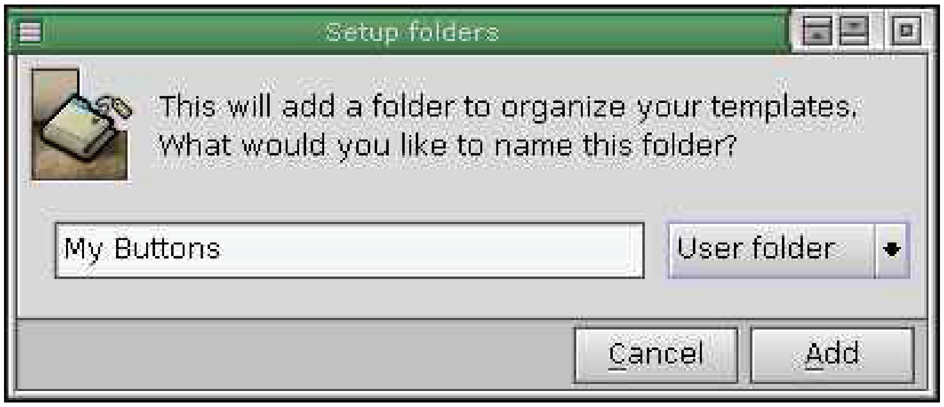
3 The Define template dialog appears.



The dialog for creating new templates.

4 Select the folder in which to place the new template. To replace an existing template, select the template instead of a folder.

To create a new folder, click on **Add Folder**. The Setup folders dialog appears:



This will add a folder to organize your templates. What would you like to name this folder?

My Buttons

User folder

Enter a folder name and select its type: User folder or PhAB folder. A User folder is visible only to you and cannot be shared with other PhAB users. A PhAB folder can be shared by several PhAB users. The predefined Widgets folder is a PhAB folder.

You need to have special permissions in order to create or to change a PhAB folder.

Each folder pops up as a palette, beside the widget palette. You can view or hide them using **Window→Show Templates**; this menu contains a list of all defined templates. When you launch PhAB, all the palettes pop up by default.

5 You must provide a name and an icon for the template.

You can create an icon by clicking **Icon Edit**.

6 Optionally, set the background color for the icon in the widget palette, and the resizing method (use original dimension or resize by dragging).

7 If the widgets that you're saving as a template have callbacks attached, you can click on the **Edit Callbacks** button and set the callbacks to be saved in the template. By default, all the callbacks are saved. If the widgets you are saving don't have any callbacks attached, the Edit Callbacks button is disabled.

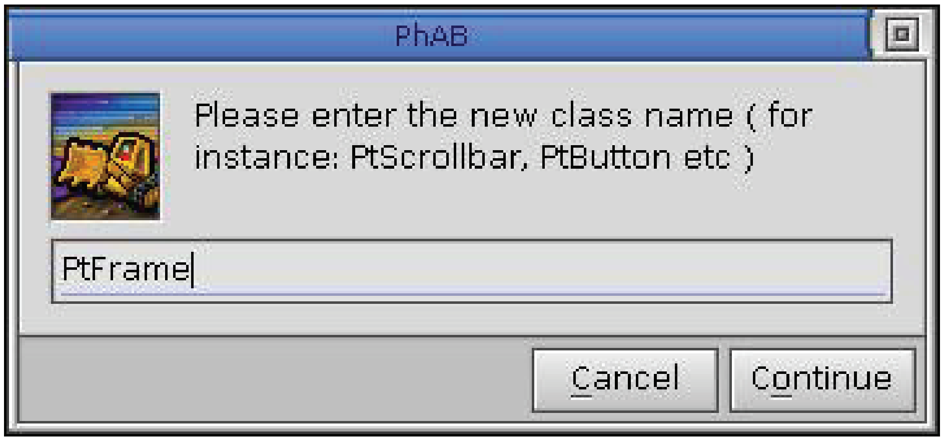
You can specify whether PhAB prompts you with a list of included callbacks when you instantiate a template widget that contains callbacks. This setting is set on the General tab of the Preferences dialog under the **When created widgets contain callbacks** option. If you select **Automatically add callbacks**, all callbacks are added. If you select **Ask me**, PhAB prompts you with a list of callbacks that you can select from.

Adding a widget class

You can create a new widget template from scratch, without starting from an existing template. If there is no template available for a widget class (for example, if you just created a brand new widget class), then you must instantiate the widget, then create the a template from the widget instance. See the *Building Custom Widgets* guide for information about building a widget and creating a widget description table.

To instantiate a widget class and then create a template:

1 Select **Edit→Add Widget Class**. The following dialog appears:



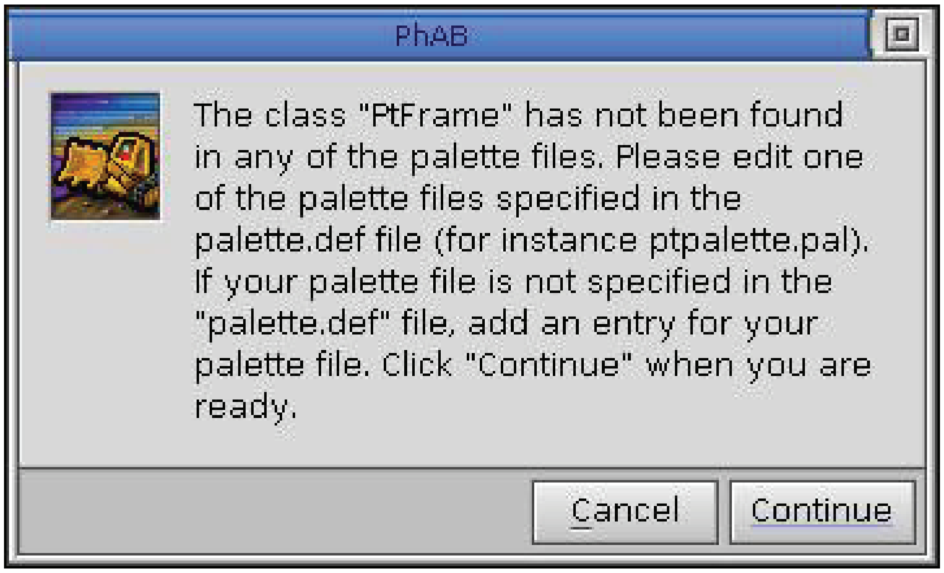
Please enter the new class name (for instance: PtScrollbar, PtButton etc)

PtFrame

2 Enter the name of the new widget class and click **Continue**.

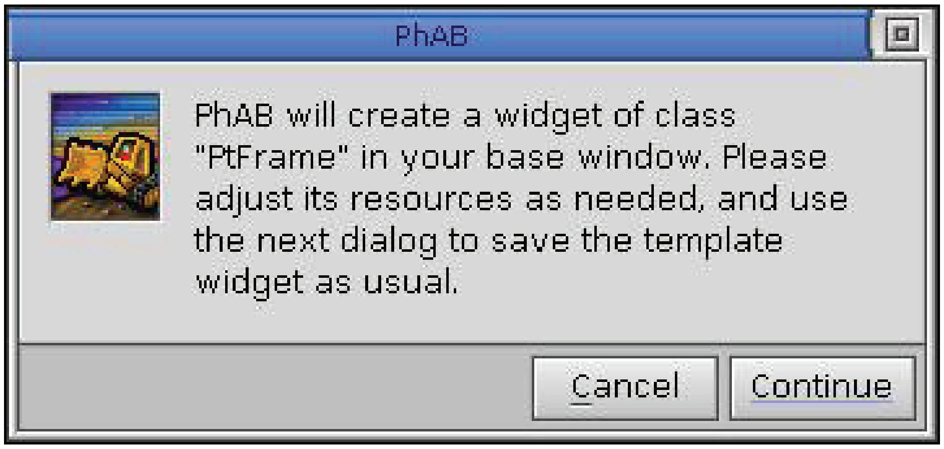
PhAB scans the palette definition files for the widget's description table. Palette definition files (`*.pal` are listed in `palette.def`. If you have written a new `*.pal` file containing your widget's description table, you should add it to `palette.def`.

If no widget definition is found, this dialog appears:


PhAB

The class "PtFrame" has not been found in any of the palette files. Please edit one of the palette files specified in the palette.def file (for instance ptpalette.pal). If your palette file is not specified in the "palette.def" file, add an entry for your palette file. Click "Continue" when you are ready.

3 Once the new widget class's description table is found in one of the palette files, the following dialog is displayed:


PhAB

PhAB will create a widget of class "PtFrame" in your base window. Please adjust its resources as needed, and use the next dialog to save the template widget as usual.

4 Customize the newly created widget, customize it, and use the Save Template to save the template.

Editing templates

You can change a template definition at any time by editing the templates. To edit an existing template:

1 Choose Templates from the Edit menu.

2 The Edit Templates dialog appears.

3 Edit the template as desired and then save the results.

You can change the template's name, the template's folder name, and the template's icon. Using drag and drop, you can move the templates between folders, and you can reorder the templates inside the same folder. Note that you can only move a template between folders of the same type (e.g. from a User folder to a User folder).

Deleting templates

To delete a template:

1. Choose Edit Templates from the Edit menu.
2. A dialog similar to that used to create a template is displayed.
3. Choose the template folder, and click Delete.

Chapter 7

Editing Resources and Callbacks in PhAB

In this chapter. . .

Editing widget resources

A widget typically has many resources that let you change its appearance and behavior. Each type of resource has its own editor.
To open any resource editor:

1 Select one or more widgets.

When you select two or more widgets, the Resources control panel displays only the resources that those widgets have in common. Editing any of these resources affects *all* the selected widgets.

2 Switch to the Resource control panel, if necessary.

If a resource's value has been changed from PhAB's default value for it, the resource's label is displayed in bold.

PhAB's default value for a resource isn't necessarily the default value assigned by the widget itself.

3 Click on a resource in the control panel. The appropriate resource editor pops up.

Every resource editor provides the following buttons:

Common buttons for resource editors.

When you want to:	Use this button:
Restore the resource to the default value (or values if more than one widget is selected)	Default
Cancel any changes made since you opened the editor or last clicked on Apply	Cancel
Apply any changes and continue editing	Apply
Apply any changes and close the editor	Done

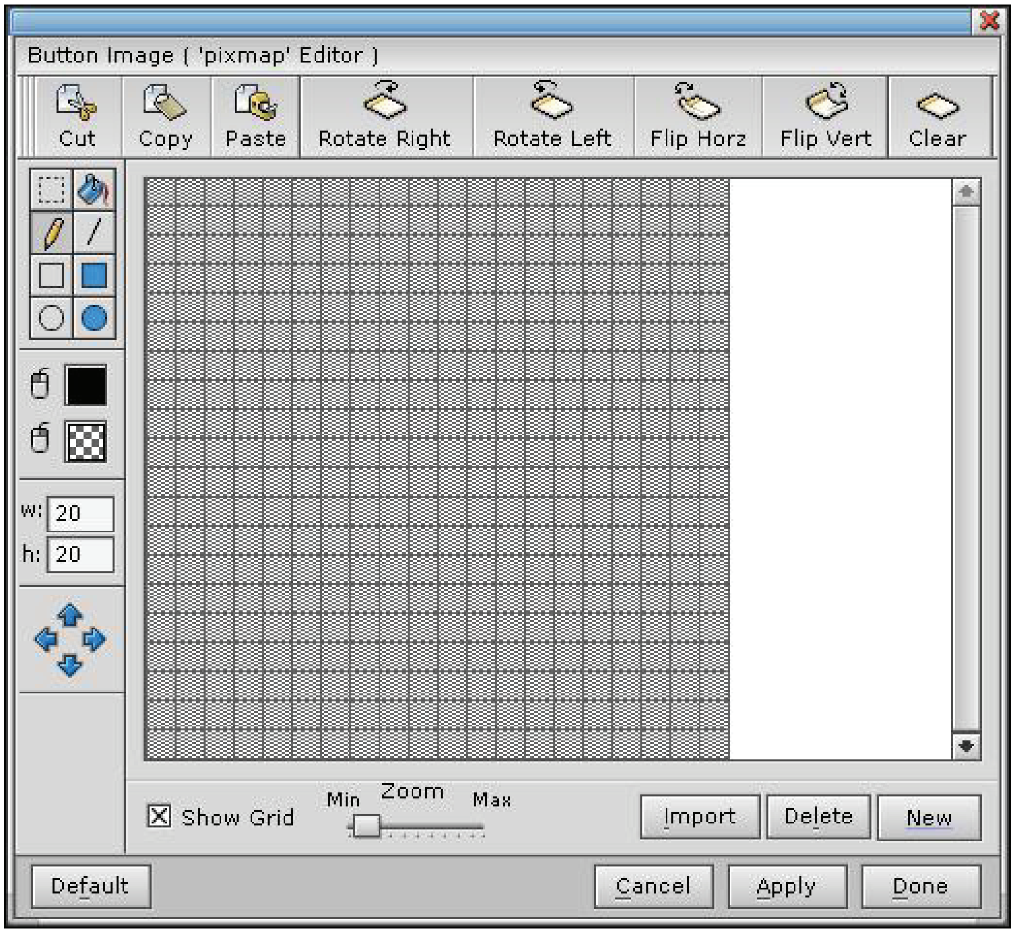
The editors for different types of resources are described in the sections that follow.

To edit:	**See this section:**
Images	Pixmap editor
Colors	Color editor
Flags	Flag/choice editor
Fonts	Font editor
Lists of text items	List editor
Numbers	Number editor
Single-line or multiline text strings	Text editors
Functions	Code editor
Layouts	Layout editors

Pixmap editor

The pixmap editor lets you customize a widget's pixmap. The editor provides a complete range of tools, so you can draw virtually any pixmap your application might need.

✷ To open the pixmap editor for any widget that can contain an image (for example, **`PtLabel`**, **`PtButton`**), click on the widget, then click on a Image resource in the Resources control panel.



Sample pixmap editor session.

The editor has several drawing modes and tools, which are described in the sections that follow. The default is freehand mode—you simply drag the pointer across the drawing grid.

The pixmap editor does not support 24-bit per pixel format JPEG images. If you wish to edit a JPEG using PhAB, you must first convert it to a palettized image (such as a .gif). You can do this by using an external image editing utility.

Setting the pixmap's size

The editor contains fields for the pixmap's height and width, both specified in pixels. To change a dimension, edit the field and press Enter.

If you reduce the size of a pixmap, part of the image may be cut off.

How to draw and erase

The following applies to all drawing tools:

In order to:	Use the:
Draw in the current color	Left mouse button
Erase a pixel or area (i.e. draw in the background color)	Right mouse button

Choosing colors

To choose the draw color:

1. Click on the following color selector:

2. The palette color selector is displayed. Click on the color of your choice. All drawing is done in that color until you select a new color.

Choosing a background color

The background (or erase) color is used when you draw with the right mouse button. To choose the background color:

1. Click on the following color selector:

2. Click on the color of your choice.

For more info, see the Color editor section.

Drawing freehand

The freehand tool lets you draw freeform lines and erase single pixels for quick fix-ups.

To draw in freehand mode:

1. Click on the freehand tool:

2. Point to where you'd like to start drawing.
3. Drag the pointer, moving it as if you were drawing with a pencil, then release the mouse button when you're done.

   You can repeat this step as often you'd like.

✷ To erase the pixel under the pointer, click the right mouse button.

Drawing lines, rectangles, and circles

To draw lines, rectangles, or circles, you use one standard method:

1. Click on the appropriate tool.
2. Point to where you'd like the object to begin.
3. Drag the pointer to where you'd like the object to end, then release the mouse button.

   You can repeat this step as often as you'd like.

Filling an enclosed area

To fill any enclosed area (i.e. any area made up of a single color):

1. Click on the fill tool:
2. Move the pointer inside the area you wish to fill, then click.

If an outline area has a break, the fill operation spills out of the hole and might fill the entire pixmap display.

Selecting an area

To use some tools, you first select an area of the pixmap.

To select an area:

1. Click on the Select tool:
2. Point to where you'd like the selection to begin.
3. Drag the pointer to where you'd like the selection to end, then release the mouse button.

   You can now "nudge" the area or perform any of the operations described in "Using the Pixmap toolbar," below.

Nudging an area

To nudge a selected area one pixel at a time:

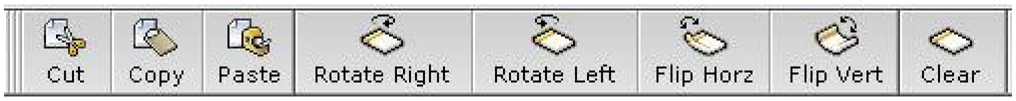
1. Select the area you wish to nudge.
2. Click a nudge arrow or press an arrow key:

Some things to note:

- PhAB overwrites any pixels in the direction of the nudge.
- A nudge can't push an area out of the pixmap.
- PhAB introduces blank space into the pixmap to fill the space left by the area you've nudged.

Using the Pixmap toolbar

The pixmap editor provides several other tools in its toolbar:



The Pixmap Editor's toolbar.

You can select an area and then use these tools to rotate, flip, cut, copy, clear, or paste the area.

Some things to note:

- If you rotate an area that isn't perfectly square, the area may overwrite some pixels.
- If part of the rotated area falls out of the pixmap, that part may be deleted.
- By using the flip tools with the copy tool, you can create mirror images.
- When you paste, point to the new location, then click. This position is the top-left corner of the pasted area.
- You can use the pixmap clipboard to copy images from one widget to another, or copy an image to its "set" version to make minor modifications.
- You can quickly clear the whole image by clicking the Clear command when nothing is selected.

Other pixmap controls

The pixmap editor also includes the following buttons:

When you want to:	**Use this control:**
Toggle the grid on and off	Show Grid
Examine smaller or larger areas of the pixmap	Zoom
Delete the pixmap	Delete

continued...

When you want to:	Use this control:
Import an image (for image-type resources only)	Import
Create a new, empty image	New

For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

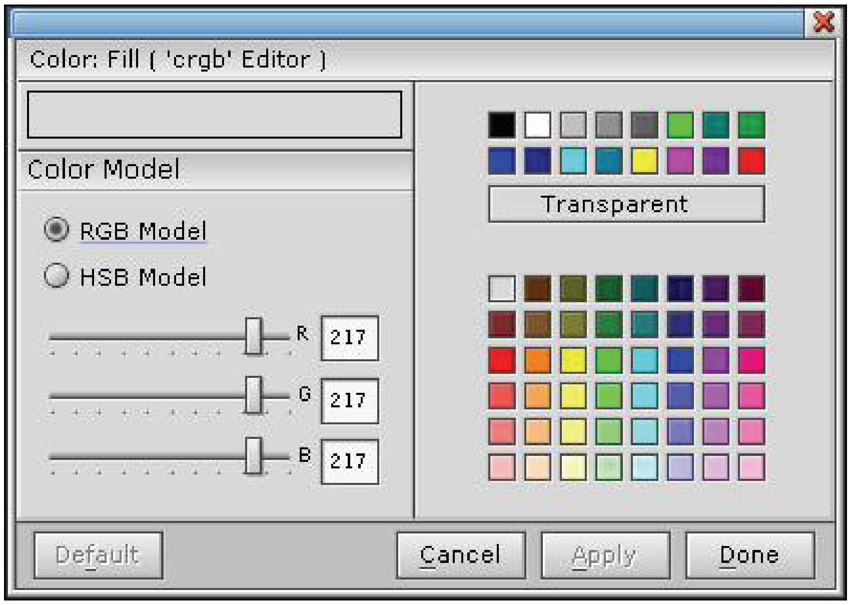
Color editor

The color editor lets you modify any color resource. Where you click in the Resource control panel determines which color editor you see:

To use the:	Click on the color resource's:
Full color editor	Name or description
Quick color editor	Current value

Full color editor

If you click on the resource name or description (i.e. the left side in the Resource control panel), the full color editor is displayed:



Color Model: [✓ RGB Model] HSB Model

R 217
G 217
B 217

Full color editor.

The full color editor gives you the choice of:

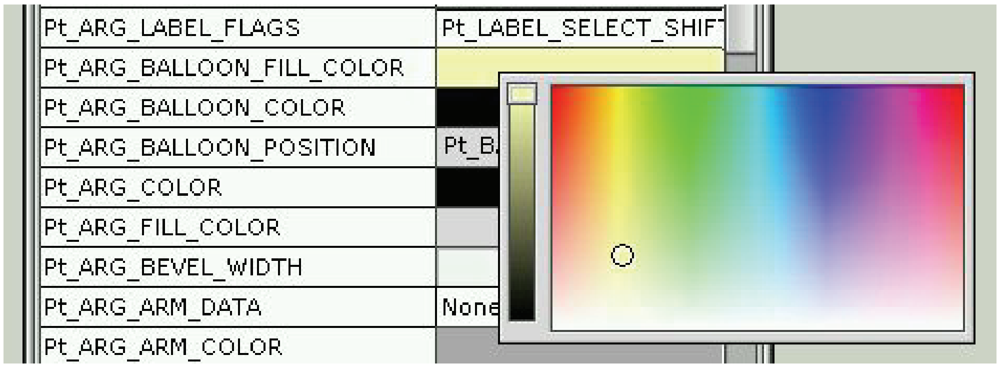
- 16 base colors that you can't change
- transparent
- 48 additional colors that you can customize using the sliders, in either the *RGB* (red/green/blue) or the *HSB* (hue/saturation/brightness) color model. PhAB maintains this custom palette between edits and sessions. Click **Reset** to return the palette to its default state.

- Using a transparent fill might introduce flickering and worsen your application's performance, especially if your application modifies the display a lot.
- The custom palette information is stored in `$HOME`**`/.ph/phab/abcpal.cfg`**. You can copy this file to other users' home directories if you want to share a default custom palette. If this file doesn't exist, PhAB looks for **`abcpal.cfg`** in the same directory the PhAB executable **`ab`** is running from.

For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

Quick color editor

If you click on the value of a color resource (i.e. the right side in the Resource control panel), the quick color editor is displayed:



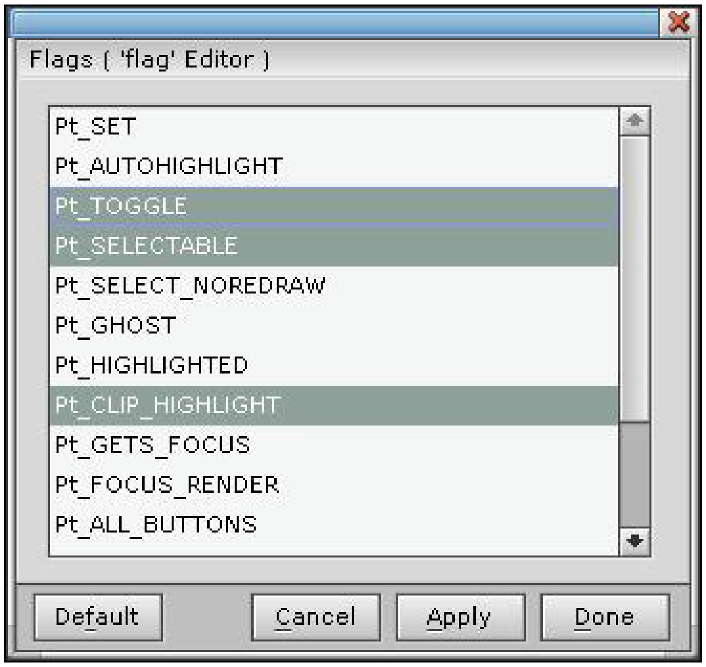
Pt_ARG_LABEL_FLAGS	Pt_LABEL_SELECT_SHIF
Pt_ARG_BALLOON_FILL_COLOR	🟨
Pt_ARG_BALLOON_COLOR	⬛
Pt_ARG_BALLOON_POSITION	Pt_B
Pt_ARG_COLOR	⬛
Pt_ARG_FILL_COLOR	▨
Pt_ARG_BEVEL_WIDTH	
Pt_ARG_ARM_DATA	None
Pt_ARG_ARM_COLOR	▨

Quick color editor.

To change the value, move the slider on the left. To change the hue, click or drag in the color patch on the right.

Flag/choice editor

Whenever you click on a flags resource or on a resource that lets you select only one of several preset values, you'll see the flag/choice editor. For example:



Flag/Choice editor.

Flag resources

If you click on a flag resource, this editor lets you make multiple selections from the displayed list.

To edit a flag list:

1 Select (or deselect) the flags you want. Since each flag is a separate toggle, you can select any number of flags or leave them all unselected. Some flags contain groups of mutually exclusive flag bits. Selecting one of these bits de-selects any corresponding mutually exclusive bits.

2 Apply your changes. The widget changes to reflect the new flags.

Option list resources

If you click on a resource that can have only one value, the flag/choice editor lets you make only one selection from the displayed list.

To choose an option from a list:

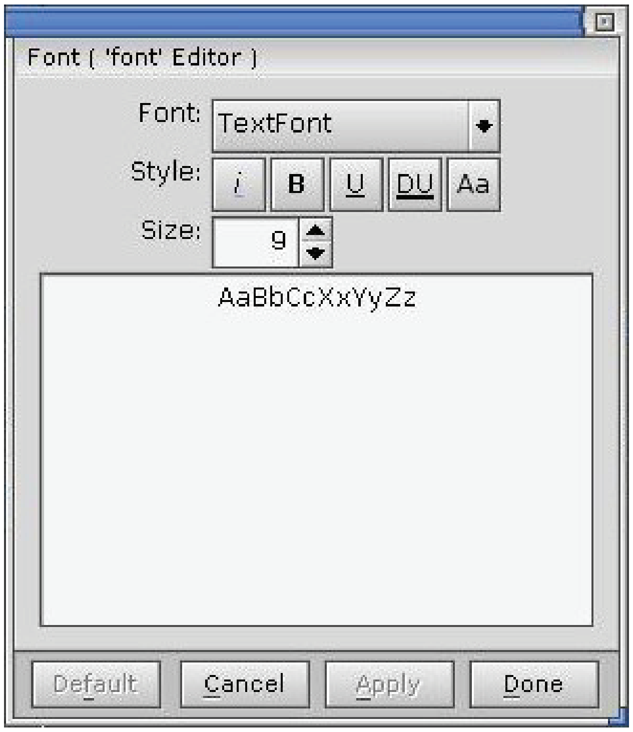
1 Click on the option. The previously selected option is deselected.

2 Apply your changes. The widget changes to reflect the new option.

For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

Font editor

Whenever you select any font resource in the Resources control panel you'll see the font editor:



Font editor.

The font editor includes these options:

Font	The typeface of the widget. Choose from the list of typefaces.
Style	The style, if applicable, of the font. Click a button to apply the style, or several buttons to apply a combination of styles (if available).
Size	The size of the font, in points.

If a typeface doesn't support an applied style at a given type size, the corresponding style becomes dimmed or unselectable.

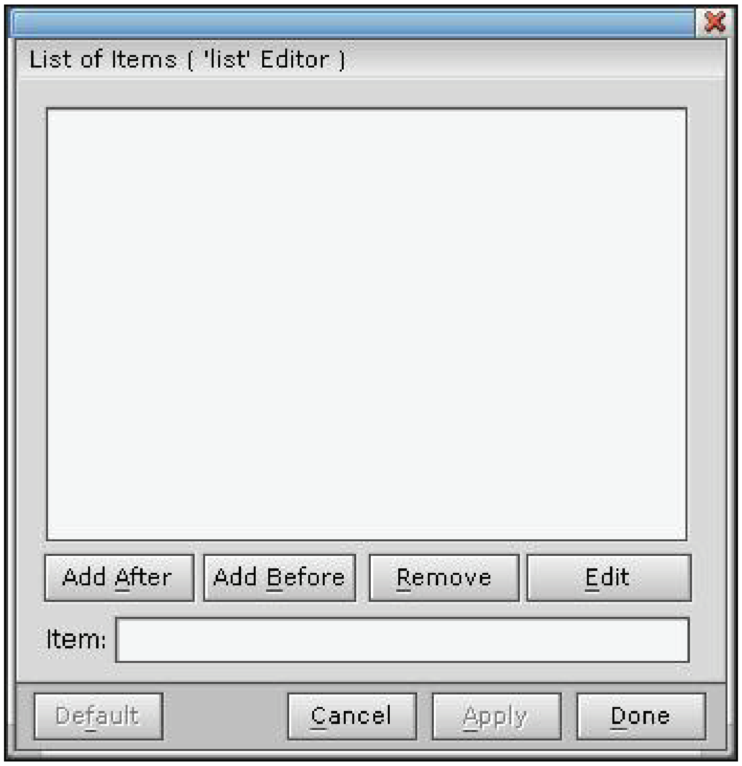
For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

List editor

Widgets such as **`PtList`** provide a list of text-based items. To edit the list, you use PhAB's list editor.

To open the editor and add the first item:

1 Select the widget, then click on the appropriate resource in the Resources control panel (usually "List of Items"). You'll see the editor:



List editor.

2 Click on the text field near the bottom of the dialog, then type the text you want.

If you need to type characters that don't appear on your keyboard, you can use the compose sequences listed in "Photon compose sequences" in the Unicode Multilingual Support appendix.

3 Press Enter or click on Insert After.

4 Click on Apply or Done.

To add more items to the list:

1 Click on an existing item, then click on Insert After or Insert Before. You'll see a new item added to the list.

2 Using the text field, edit the new item's text.

3 Click on Edit, then click on Apply or Done.

You can't create blank lines within the list of items.

Editing existing list items

To edit an existing item:

1 Click on the item.

2 Edit the item's text.

3 Click on Edit, then click on Apply or Done.

For text-editing shortcuts, see "Text editors."

Deleting list items

To delete an item:

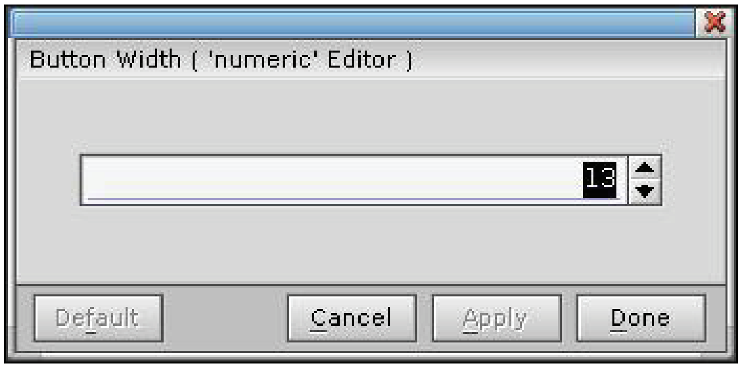
1 Click on the item, then click on Remove.

2 Click on Apply or Done.

For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

Number editor

You can edit the value of a numeric resource right in the Resources control panel, or you can click on the resource name to use the number editor:


Button Width ('numeric' Editor)

13

Number editor.

To change the value that the editor displays, you can:

- Use the text-editing techniques described in “Text editors.”

  Or:

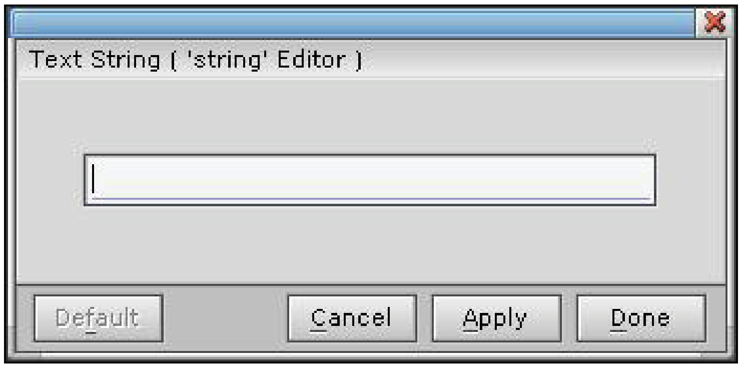
- Click on the editor’s increase/decrease buttons.

For a description of the standard editor buttons at the bottom of the editor, see the “Editing widget resources” section.

Text editors

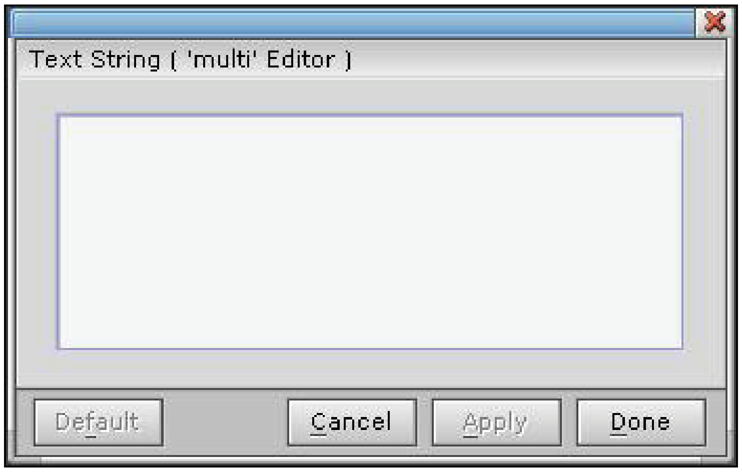
You can edit a text resource right in the Resources control panel, or you can click on the resource to display a text editor. There are two text editors: one for single-line text, and one for multiline.

Whenever you click on a single-line text resource in the Resources control panel (e.g. the Text String resource for **`PtText`**), you’ll see the text editor:



Text editor.

When you select any multiline text resource—such as the Text String resource of a **`PtLabel`** or **`PtMultiText`** widget—you’ll see the multiline text editor:



Multiline text editor.

The single-line and multiline editors are similar — here are the common operations:

In order to:	**Do this:**
Delete the character before the text cursor	Press Backspace
Delete the character after the cursor	Press Del
Delete several characters all at once	Drag the pointer across the characters, then press Del
Delete the entire line	Press Ctrl-U
"Jump" the cursor to any position in the line	Click on that position
Move the cursor character by character	Press ← or →
Move the cursor to the start or end of the line	Press Home or End

For the single-line text editor:

In order to:	**Do this:**
Process a text change	Press Enter, or click on Done or Apply

For the multiline text editor:

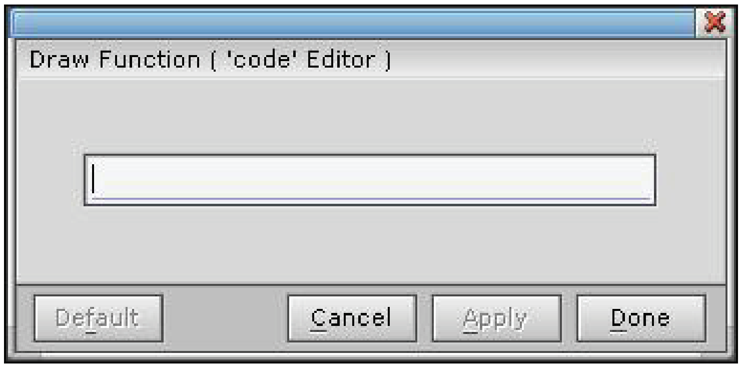
In order to:	**Do this:**
Enter a new line of text	Press Enter
Move the cursor to the start or end of a line	Press Home or End
Move the cursor to the start or end of the text string	Press Ctrl-Home or Ctrl-End
Apply your changes and dismiss the editor	Press Ctrl-Enter

If you need to type characters that don't appear on your keyboard, you can use the compose sequences listed in "Photon compose sequences" in the Unicode Multilingual Support appendix.

For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

Code editor

When you select a function resource, such as the Draw Function (*Pt_ARG_RAW_DRAW_F*) resource of a **`PtRaw`** widget, you'll see the Code editor:



Code editor.

The widget must have a unique instance name before you can edit its function resources.

Type the name of the function—see "Function names and filenames" in the Working with Code chapter. If you've already given your application a name by saving it (see "Saving an application" in the Working with Applications chapter), you can edit the function by clicking the button to the right of the text field.

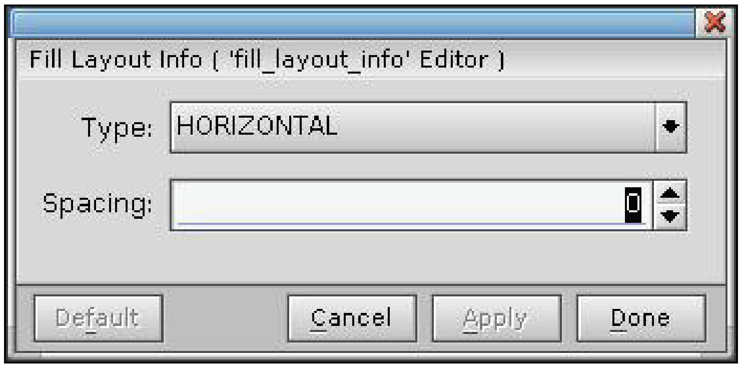
For a description of the standard editor buttons at the bottom of the editor, see the "Editing widget resources" section.

Layout editors

When you set the Layout Type resource *Pt_ARG_LAYOUT_TYPE* of a container widget to something other than the default *Pt_ANCHOR_LAYOUT*, you can then set the corresponding Layout Info resource. You can also set the corresponding Layout Data resource for widgets within the container widget. Each kind of Layout Info and Layout Data resource has an editor.

For more information about layouts, see Using layouts in the Geometry Management chapter.

Fill layout info editor


Fill Layout Info ('fill_layout_info' Editor)

Type: HORIZONTAL

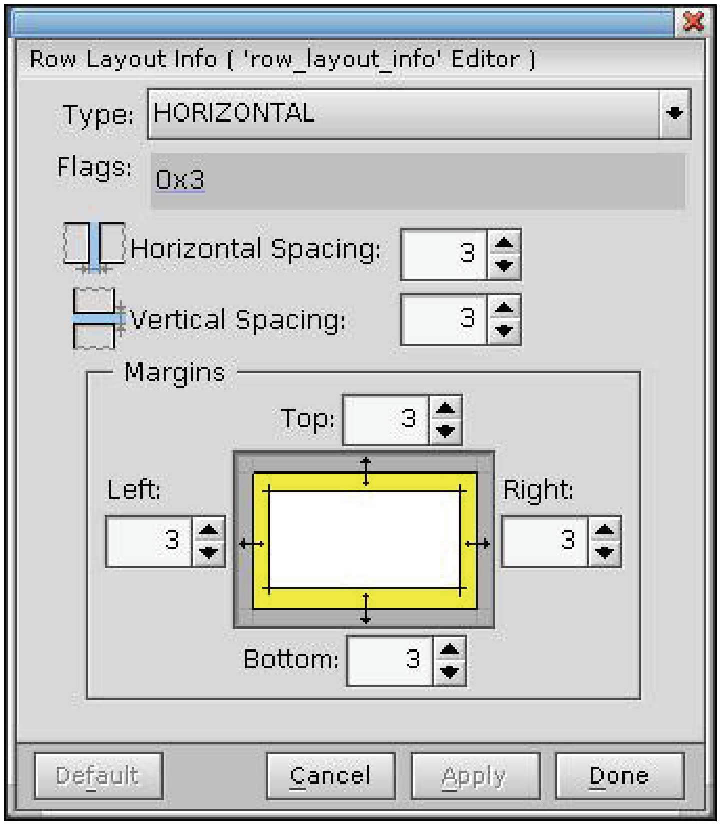
Spacing: 0

Fill layout info editor.

When you set *Pt_ARG_LAYOUT_TYPE* to Pt_FILL_LAYOUT, you can open this editor by clicking on the Fill Layout Info resource (*Pt_ARG_FILL_LAYOUT_INFO*). Using this editor you can set the:

- type of fill layout
- spacing between widgets.

Row layout info editor


Row Layout Info ('row_layout_info' Editor)

Type: HORIZONTAL

Flags: 0x3

Horizontal Spacing: 3

Vertical Spacing: 3

Margins

Top: 3

Left: 3

Right: 3

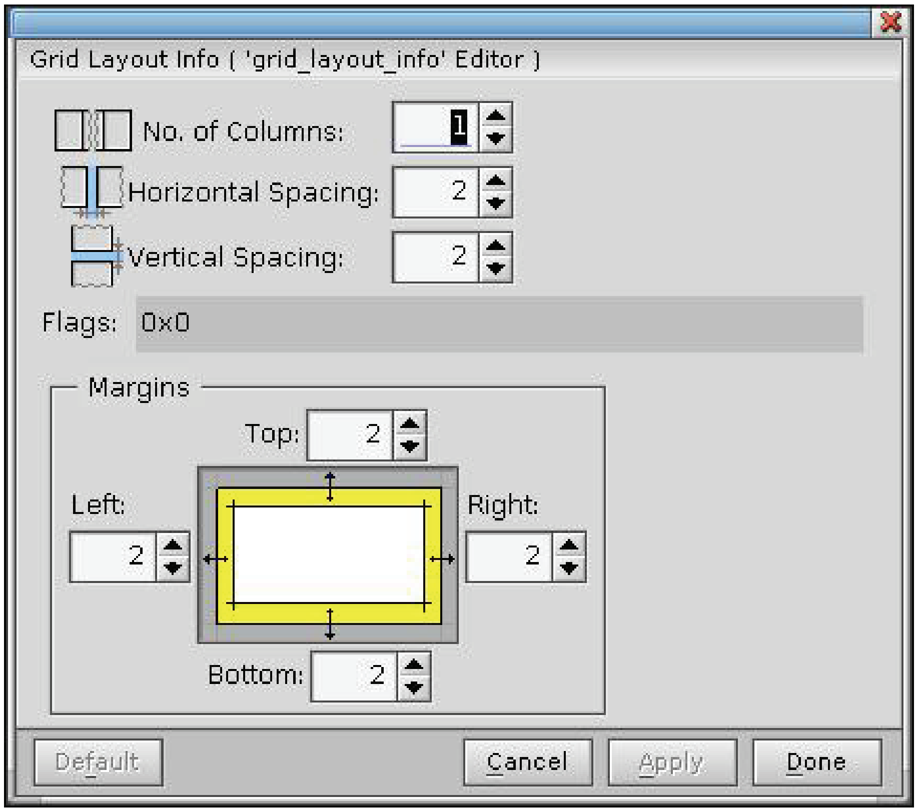
Bottom: 3

Row layout info editor.

When you set *Pt_ARG_LAYOUT_TYPE* to Pt_ROW_LAYOUT, you can open this editor by clicking on the Row Layout Info resource (*Pt_ARG_ROW_LAYOUT_INFO*. Using this editor you can set:

- the row layout type
- additional flags
- horizontal and vertical spacing between widgets
- the outside margin between widgets and the parent container.

Grid layout info editor:


Grid Layout Info ('grid_layout_info' Editor)

No. of Columns: 1

Horizontal Spacing: 2

Vertical Spacing: 2

Flags: 0x0

Margins

Top: 2

Left: 2

Right: 2

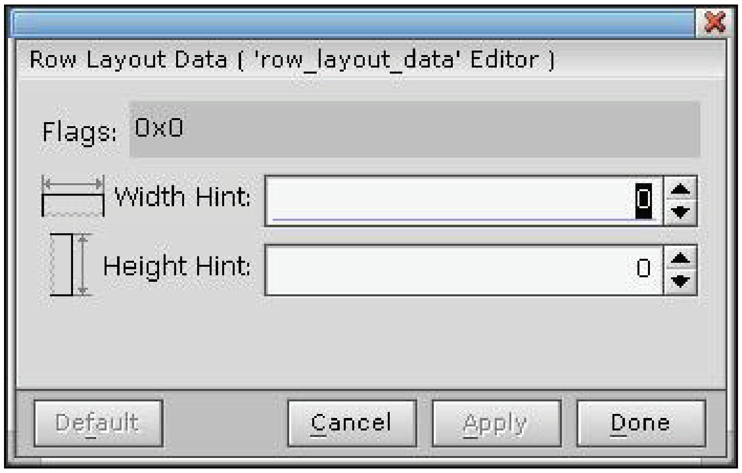
Bottom: 2

Grid layout info editor.

When you set *Pt_ARG_LAYOUT_TYPE* to Pt_GRID_LAYOUT, you can open this editor by clicking on the Grid Layout Info resource (*Pt_ARG_GRID_LAYOUT_INFO*. Using this editor you can set:

- the number of columns in the grid
- additional flags
- horizontal and vertical spacing between widgets
- the outside margin between widgets and the parent container.

Row layout data editor


Row Layout Data ('row_layout_data' Editor)

Flags: 0x0

Width Hint: 0

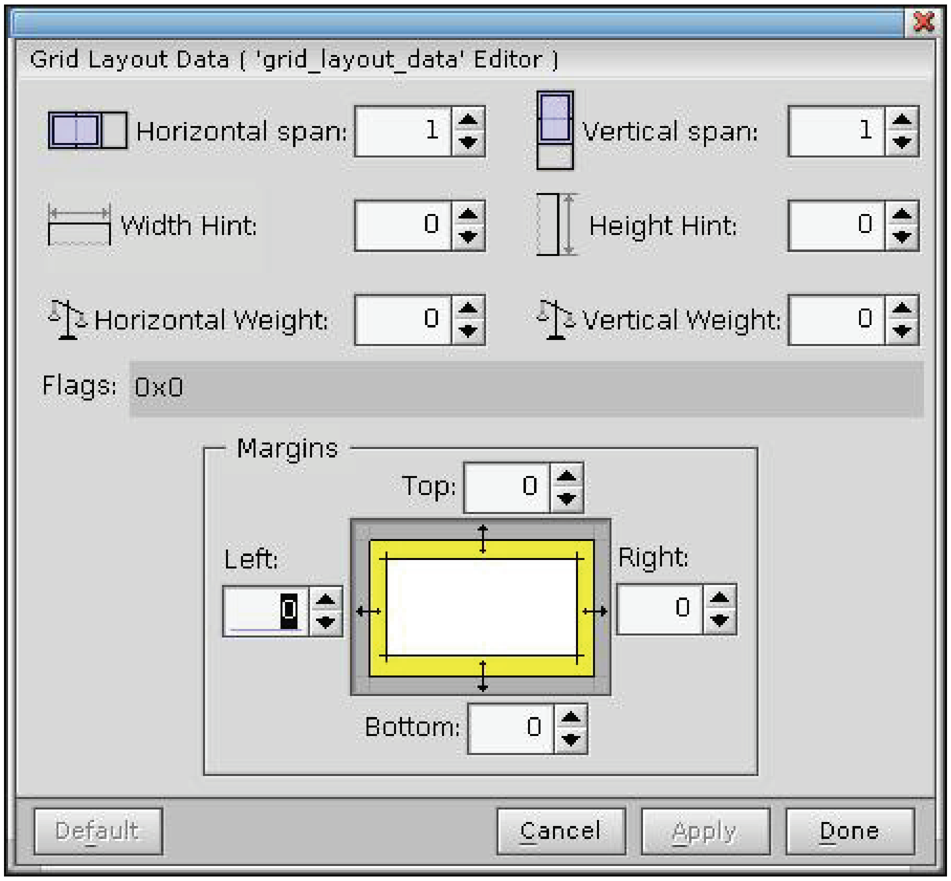
Height Hint: 0

Row layout data editor.

When you set *Pt_ARG_LAYOUT_TYPE* to Pt_ROW_LAYOUT of a container widget, you can open this editor by clicking on the Row Layout Data resource (*Pt_ARG_Row_LAYOUT_DATA* of any child widget. Using this editor you can set the:

- width spacing hint
- height spacing hint.

Grid layout data editor


Grid Layout Data ('grid_layout_data' Editor)

Horizontal span: 1
Vertical span: 1
Width Hint: 0
Height Hint: 0
Horizontal Weight: 0
Vertical Weight: 0
Flags: 0x0

Margins
Top: 0
Left: 0
Right: 0
Bottom: 0

Grid layout data editor.

When you set *Pt_ARG_LAYOUT_TYPE* to Pt_GRID_LAYOUT of a container widget, you can open this editor by clicking on the Grid Layout Data resource (*Pt_ARG_GRID_LAYOUT_DATA* of any child widget. Using this editor you can set the:

- horizontal and vertical span
- width and height hints
- horizontal and vertical weights
- grid margins.

For a description of the standard editor buttons at the bottom of these editors, see the "Editing widget resources" section.

Callbacks

Callbacks form the link between your application interface and your application code. For example, let's say you want your application to perform an action when the user selects a certain button. In that case, you would attach a callback function to that button's Activate callback. When the user selects the button, the widget invokes the callback function, and your application takes the appropriate action in the callback code.

Almost all widgets support several types of callbacks. These callbacks can be specific to the widget or inherited from its parent classes. Some of these types (defined in the **`PtBasic`** widget) are defined in the following table:

Type	Resource	Typically invoked when the user:
Activate	*Pt_CB_ACTIVATE*	Presses and releases the left mouse button
Arm	*Pt_CB_ARM*	Presses the left mouse button
Disarm	*Pt_CB_DISARM*	Releases the left mouse button
Repeat	*Pt_CB_REPEAT*	Holds down the left mouse button
Menu	*Pt_CB_MENU*	Presses the right mouse button

For more information about these callbacks, see the *Widget Reference*. If you're interested in using *Pt_CB_MENU* to display a menu module, see the Accessing PhAB Modules from Code chapter.

All Photon widgets inherit two other types of callbacks:

Hotkey callbacks
: Attach callback code to a key or keychord. When the application window gets focus, the hotkeys become active. Pressing one invokes the appropriate hotkey link callback.

Event handlers (*Raw* or *Filter* callbacks)
: Attach callbacks directly to Photon events

In the development environments for some windowing systems, you can attach only callback code functions to a widget's callbacks. But whenever you use PhAB to create a callback, you can go one step further and attach windows, dialogs, menus, and much more. As we mentioned earlier, this extended functionality is provided by PhAB's special form of callback, called the *link callback*.

Link callbacks also let you add functionality that isn't available when you attach callbacks "by hand." For example, if you link a dialog to a button widget, you can specify where the dialog appears. You can also specify a setup function that's automatically called before the dialog is realized, after the dialog is realized, or both.

PhAB provides two general categories of link callbacks:

module-type link callbacks

let you attach an application module to any widget callback. PhAB provides the following categories of module-type link callbacks:

- Dialog
- Window
- Menu
- Picture

For more information, see "Module callbacks" later in this chapter.

code-type link callbacks

let you run a code function when the widget's callback is invoked. PhAB provides the following categories of code-type link callbacks:

- Code
- Done
- Cancel

The Done and Cancel types provide an additional feature: they'll automatically close or destroy the widget's parent module after the callback function is called. You'll find these types useful for creating any button that closes a dialog window.

A Done callback in the base window exits the application. A Cancel callback in the base window closes the application's windows but *doesn't* exit the application.

For more information, see "Code callbacks" later in this chapter.

Editing callbacks

The callback editor lets you add, change, delete, or view a widget's list of callbacks.

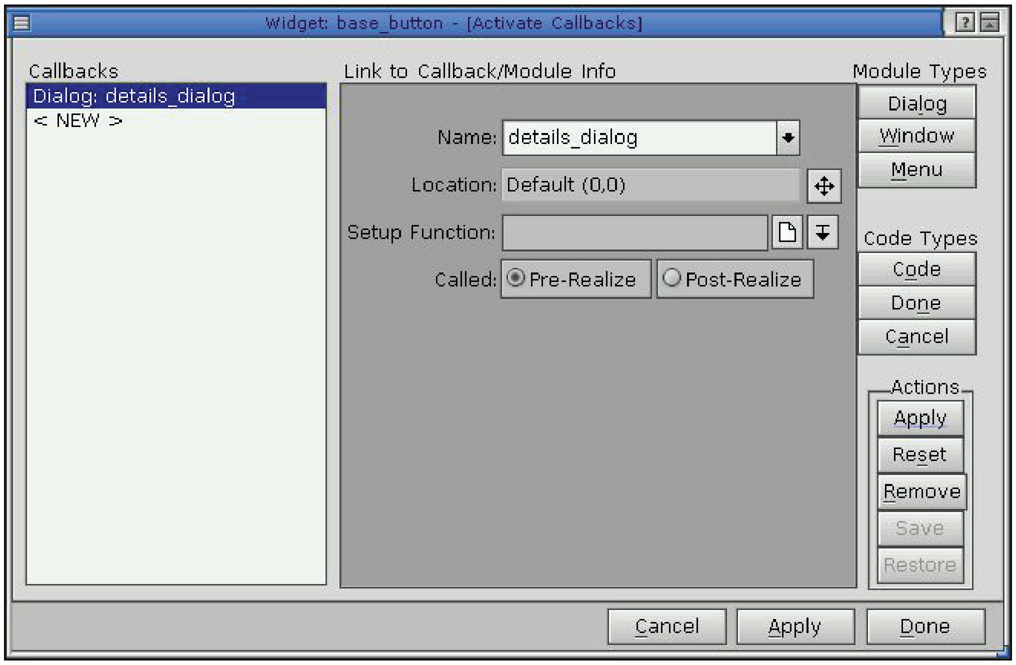
To add a callback to a command item or toggle item in a menu, see "Menu modules" in the Working with Modules chapter.

If you're adding a link callback to a widget, the widget's instance name must be unique. If PhAB tells you the name isn't unique, use the Resources or Callbacks control panel's Widget Instance Name field to edit the name.

To open the callback editor and edit a widget's list of callbacks:

1. Select the widget, then switch to the Callbacks control panel, if necessary.
2. Choose the callback type from the widget's callback list. (For example, to add an *Pt_CB_ACTIVATE* callback, click on Activate.)

Here’s a sample callback-editor session:



Name: details_dialog
Location: Default (0,0)
Setup Function:
Called: [✓ Pre-Realize] Post-Realize

Callback editor.

1 To add a new callback, click on <NEW>. To edit an existing callback, click on that callback in the Callbacks list.

2 If you’re adding a new callback, choose the type of callback you want to add. To do this, choose from either “Module Types” or “Code Types.”

3 Fill in the information in the “Link to Callback/Module Info” section. The fields in this section depend on the type of callback chosen. For more information, see the sections in this chapter on specifying:

- module callbacks
- code callbacks
- hotkey callbacks
- event handlers (raw and filter callbacks)

4 After you’ve added or edited any callback, click on the appropriate button:

- Apply—Apply any changes; be sure to do this before you start working on other callbacks.
- Reset—Restore the callback information to the original values.
- Remove—Delete the callback from the callback list.

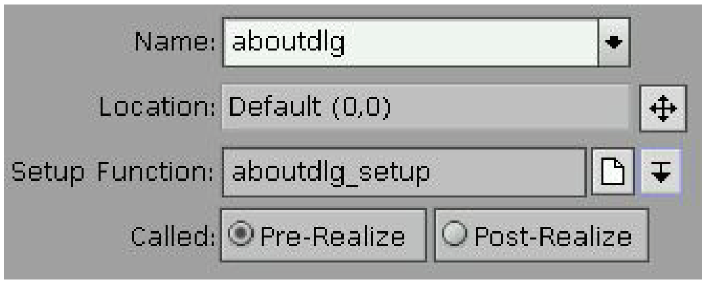
Module callbacks

A module-type link callback can be used to connect a widget to a module. For example, selecting a button could create a module.

When you use a module-type link callback to create a module, the module becomes a child of your application's base window, not a child of the module that contains the widget that the link callback is defined for.

If you want the new module's parent to be something other than the base window, you need to use an *internal link* to create the module in your application's code. For more information about internal links and other cases where you'd use them, see the Accessing PhAB Modules from Code chapter.

Depending on the kind of module-type link callback you're creating, PhAB's callback editor displays some or all of these fields:


Name: aboutdlg
Location: Default (0,0)
Setup Function: aboutdlg_setup
Called: (●) Pre-Realize ( ) Post-Realize

Callback editor fields for module-type link callbacks.

Name	The name of the module. If you click on the icon next to this field, you'll see a list of existing modules. Either choose from this list or enter the name of a module that doesn't exist (PhAB creates the module for you when you add the callback).
Location	Lets you specify where the module is displayed. By default, a menu module is located below the widget that invokes it. For all other modules, the default location is determined by the Window Manager. For more information, see "Positioning a module" in the Working with Modules chapter.
Setup Function	Lets you specify a function that can be called at two different times (as specified by the Called field): • before the module is displayed (prerealize) • after the module is displayed (postrealize) You can specify only one setup function—the PhAB API calls the same function for both pre- and postrealization of the module. To

distinguish which pass is being invoked, check the PhAB reason code.

Click on the icons beside the Setup Function field to edit the function or select from the existing callbacks.

Hotkey—(hotkey callbacks only)

The keyboard key and modifier (such as Alt or Ctrl) that trigger the callback. See the section " Specifying hotkey callbacks."

Event Mask—(event handlers only)

Lets you specify which Photon events the widget is sensitive to. See "Event handlers — raw and filter callbacks."

Prerealize setup function

The prerealize setup function lets you preset a module. For example, let's say your application needs to "fill in the blanks" of a dialog before displaying that dialog. In the setup function, you would use PhAB-generated manifest names to preset the resources of the dialog's various widgets.

After the setup function runs, it returns Pt_CONTINUE. The dialog is then realized and displayed on the screen, using all the preset values.

Postrealize setup function

The postrealize function works much like the prerealize function, except that it's called after the dialog is displayed on the screen. You typically use this type of function when you need to update the module after it's become visible. PhAB's code-generation dialog provides a good example. It displays on the screen and then, using a postrealize function, updates a progress bar continuously as the application code is generated.

The setup function for a menu module is called only before the menu is displayed. For most applications, you would use this function to set the initial states of the menu items. For example, you could use it to disable certain menu items before the menu is displayed.

Setup functions are stored in stub files

When you specify a setup function, PhAB generates a stub function; for information on specifying the language (C or C++) and the filename, see "Function names and filenames" in the Working with Code chapter.

Code callbacks

This type of callback lets you run a code function when the widget's callback is invoked.

You can add code callbacks from your application's code, but it's easier to do in PhAB. For more information, see "Callbacks" in the Managing Widgets in Application Code chapter.

When you're creating a code-type link callback, the callback editor asks you to specify the following:

Function
: This is the function that's called when the widget invokes the callback. For the Done and Cancel types, this function is optional, so you can attach the callback just to close the module.

  As described above, Done and Cancel are similar, except that a Done callback in the base window exits the application, while a Cancel callback closes the window but doesn't exit the application. There's no real difference between the Done and Cancel callback functions — they simply provide different reason codes in the callback.

  For example, let's say you have a dialog with a Done button and a Cancel button. If you attach a Done-type callback to the Done button and a Cancel-type callback to the Cancel button, you could use the same code function in both and just look at the reason code to determine which button the user selected.

Hotkey—(Hotkey callbacks only)
: The keyboard key and modifier (such as Alt or Ctrl) that trigger the callback. See the section "Hotkey callbacks."

Event Mask—(event handlers only)
: Lets you specify which Photon events the widget is sensitive to. See "Event handlers — raw and filter callbacks."

Callback functions are stored in stub files

When you specify a callback function, PhAB generates a stub function; for information on specifying the language (C or C++) and the filename, see "Function names and filenames" in the Working with Code chapter.

Hotkey callbacks

Widgets support *hotkey callbacks*. These callbacks let you attach keyboard keys to specific callback functions. When the application window gets focus, the hotkeys become active. Pressing one invokes the appropriate hotkey link callback.

Hotkeys — the basics

Here's some basic information about hotkeys:

- Hotkeys are a combination of a character key and a modifier key (Alt, Shift or Ctrl). Most of the time, Alt is used for hotkeys.
- You can use a modifier on its own as a hotkey, but it's probably not a good idea.
- A hotkey isn't invoked if any ancestor of the widget that owns it is blocked.
- A hotkey is processed *after* the widgets have been given the key event. If a widget consumes the event, no hotkey callback is called. So when a text field has focus, the Enter key, arrow keys, Space, and all displayable characters won't work as hotkeys because the widget consumes those events. This is usually the desired behavior (consider editing in an application which has hotkeys defined for all the arrow keys).

  You can force the hotkey processing to be done first by setting Pt_HOTKEYS_FIRST in the *Pt_ARG_CONTAINER_FLAGS* resource of the container widget (window, pane, . . .) that contains the widgets that would normally consume your would-be hotkey events. Setting this flag on the window guarantees all hotkey processing is done before any widgets get the key event. For more information, see "Processing hotkeys," below.
- Widgets *must* be selectable for their hotkeys to be active (with the exception of disjoint widgets such as windows and menus). Make sure the widget's *Pt_ARG_FLAGS* has Pt_SELECTABLE and Pt_GETS_FOCUS set.

  If the widget isn't normally selectable and you don't want its appearance to change when selected, you'll also want to set the Pt_SELECT_NOREDRAW widget flag.
- Often it doesn't matter which widget a callback is connected to. In those cases, just attach the hotkey to the window.

Specifying the hotkey label

Setting up a hotkey isn't enough—you need to tell your user about it! You should display a hotkey label in the widget invoked by the hotkey:

- For most widgets, edit the Accelerator Key (*Pt_ARG_ACCEL_KEY*) resource. Specify the character in the widget's label that you want underlined. You can't include any modifier keys in the label.
- For menu items, the underlined character is the shortcut that you can use to select an item when the menu's displayed. The hotkey label is displayed separately, to the right of the menu item's label. Specify the hotkey (including the modifier keys) in the menu editor's Accel Text field.

Specifying the callback

In PhAB, each widget's callback list displays an entry called "Hotkey" or *Pt_CB_HOTKEY* that you use to define hotkeys. Before you define the hotkey, you need to determine *where* to do so. Where you define the hotkey callback depends on:

- where you want a module (such as a menu) to appear

- what widget you need in the callback function
- where the user is going to type the hotkey.

Where you want a module to appear

When you define the hotkey, you can specify where the module is to appear. For example, if the hotkey is meant to display the menu module associated with a **`PtMenuButton`** widget in your window's **`PtMenuBar`**, define the hotkey in the menu button. Use the Location dialog to have the menu appear under the menu button. For more information, see "Positioning a module" in the Working with Modules chapter.

What widget you need in the callback function

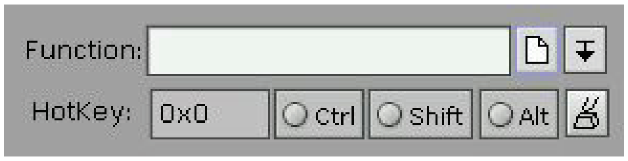
The widget that owns the callback is the one passed to the callback function.

Where the user is going to type the hotkey

For example, if the hotkey is an accelerator for a menu item, add the hotkey to the window in which the menu is used, not to the menu module.

- The hotkeys in a given module should be unique. If you define the same hotkey more than once, the last one is used.
- If you're developing a multilingual application, you'll need to choose hotkeys carefully so they'll be relevant in each language, or you'll need a different set of hotkeys for each language. See the International Language Support chapter for more information.

When you select the *Pt_CB_HOTKEY* callback, the callback editor pops up with a Hotkey field in the link-information area:



Hotkey field in the callback editor.

You must fill in the Hotkey field when creating a hotkey callback. There are two ways to set up the hotkey: one easy, the other not so easy.

- **the not-so-easy way**—you can type the hotkey value, in hex, in the Hotkey field. To find the value for the keycap you want to use, see the header file **`<photon/PkKeyDef.h>`**, and search for the name of the keycap, prefixed by **`Pk_`**.

Use lowercase letters for hotkeys; uppercase letters won't work. For example, for a hotkey Alt-F, look up the hex value for `Pk_f`, not `Pk_F`.

The field also has 3 toggle buttons—Ctrl, Shift, and Alt—to let you specify modifiers for the hotkey value.

- **the easy way**—press the button to the right of the Alt toggle button, then press the keychord you want to use for the hotkey. PhAB automatically determines the key and modifiers you pressed.

Processing hotkeys

Here's how a hotkey works:

- When a key event arrives at a window, the window passes the event to its child widgets.
- If a child consumes the event, nothing further happens.
- Otherwise, the event is checked against the window's list of hotkeys. If the hotkey is found, its callback is invoked.
- If the hotkey isn't found, the parent window's hotkey list is searched, and so on up the hierarchy of windows.

The *Pt_ARG_CONTAINER_FLAGS* resource of container-class widgets includes some flags that affect the processing of hotkeys:

Pt_HOTKEY_TERMINATOR

Prevent the hotkey search from going up to the parent container.

The Pt_HOTKEY_TERMINATOR flag works only if it's set in a *disjoint* container-class widget.

Pt_HOTKEYS_FIRST

Process key events that reach this container as hotkeys before passing them to the container's children. If the event is a hotkey, it's consumed, so it isn't passed to the children.

Disabling hotkeys

Giving the user a visual indication that a hotkey is disabled is different from actually disabling the hotkey.

To give the visual indication, use the technique appropriate to the widget:

- If the hotkey is associated with a button, set the Pt_GHOST flag and remove the Pt_SELECTABLE and Pt_GETS_FOCUS flags in the button's *Pt_ARG_FLAGS* resource.

- If the hotkey is associated with a menu item created in PhAB, call *ApModifyItemState()*.
- ...

To disable the hotkey, use one of the following techniques:

- Don't disable the hotkey. Instead, as the first thing you do in your hotkey's callback code, check to see if anything should be done. If not, just return from the callback. For example, if the hotkey callback is the one for pasting text, check to see if there's anything to paste. If there isn't, simply return.

  Or

- With the exception of disjoint widgets, if the widget that the hotkey callback is attached to isn't selectable, the hotkey is treated as if it didn't exist. For a widget to be selectable, the Pt_SELECTABLE flag must be set in the *Pt_ARG_FLAGS* resource.

  One good reason for this approach is that it works even if your application has the same hotkey defined in more than one window. For example, we might have an Edit menu in the base window and an Erase button in a child window, both with Alt-E as a hotkey. If the child window currently has focus and the user presses Alt-E, the child window's Erase button callback is called.

  Now, if we disable the Erase button in the child window, we would want Alt-E to cause the base window's Edit menu to appear. In this scenario, as long as the Erase button is selectable, its callback would be called. So we simply make the Erase button unselectable. Now when the user presses Alt-E, the base window's Edit menu appears, even though the child window still has focus.

  Or

- You could call *PtRemoveHotkeyHandler()* to remove the hotkey and later call *PtAddHotkeyHandler()* to enable it again.

Event handlers — raw and filter callbacks

Event handlers let you respond directly to Photon events. You can attach event handlers to any widget; they're like other widget callbacks, but with the addition of an event mask. Using this mask, you can choose which events your callbacks receive.

You'll find them particularly useful for getting the Ph_EV_DRAG events for a particular window. For more information on dragging, see " Dragging" in the Events chapter.

`PtWidget` defines the following event-handler resources:

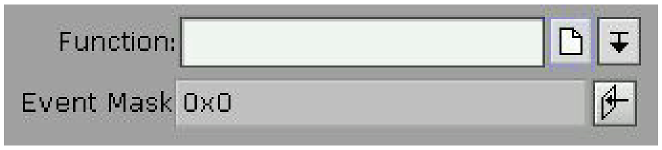
Pt_CB_FILTER	Invoked *before* the event is passed to the widget.
Pt_CB_RAW	Invoked after the widget has processed the event (even if the widget consumes the event).

For a description of raw and filter event handlers and how they're used, see "Event handlers — raw and filter callbacks" in the Events chapter.

For information on adding event handlers in application code, see "Event handlers" in the Managing Widgets in Application Code chapter.

To attach a raw or filter callback:

1. Select the widget, then switch to the Callbacks control panel, if necessary.
2. Click on the *Pt_CB_RAW* (Raw Event) or *Pt_CB_FILTER* (Filter) resource to open the callback editor.
3. The editor pops up with an Event Mask field in the link information area:



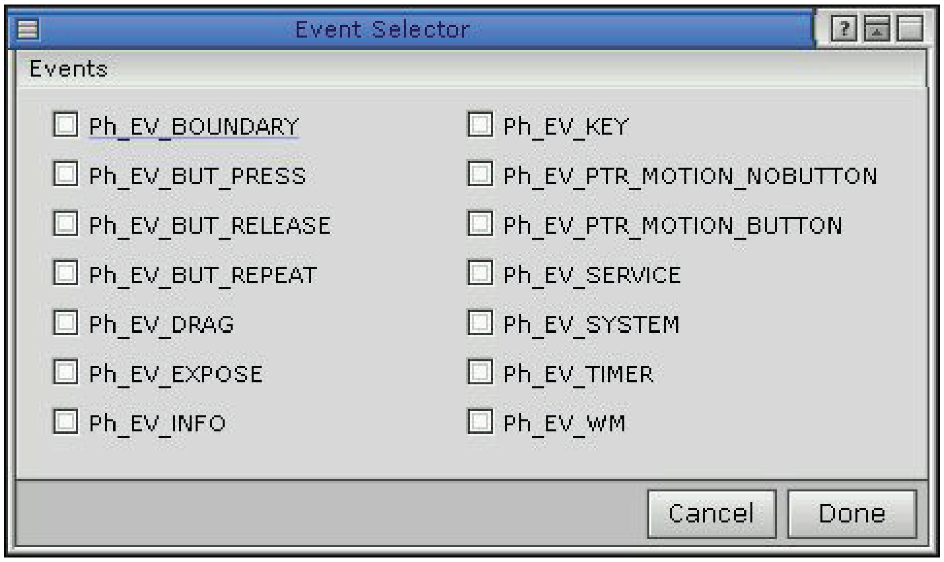
Function:

Event Mask 0x0

Event Mask field in the callback editor.

The Event Mask field lets you specify which Photon events you want the widget to be sensitive to. When any of those low-level events occur, the widget invokes the callback.

Click on the icon next to this field to open the event selector:



Event Selector

Events

☐ Ph_EV_BOUNDARY	☐ Ph_EV_KEY
☐ Ph_EV_BUT_PRESS	☐ Ph_EV_PTR_MOTION_NOBUTTON
☐ Ph_EV_BUT_RELEASE	☐ Ph_EV_PTR_MOTION_BUTTON
☐ Ph_EV_BUT_REPEAT	☐ Ph_EV_SERVICE
☐ Ph_EV_DRAG	☐ Ph_EV_SYSTEM
☐ Ph_EV_EXPOSE	☐ Ph_EV_TIMER
☐ Ph_EV_INFO	☐ Ph_EV_WM

Event selector.

4. Select the events you want the widget to be sensitive to, then close the selector.

For more info, see the event types described for the **`PhEvent_t`** structure in the Photon *Library Reference*.

Chapter 8

Geometry Management

In this chapter. . .

This chapter discusses how to set up or fine-tune your widgets' geometry.

Container widgets

A *container* widget is a child of the **`PtContainer`** widget class. Container widgets are the only widgets that can have children. Any widget that doesn't have a window of its own is always rendered within the boundaries of its parent. Only widgets belonging to a subclass of the **`PtWindow`** widget class get a window of their own.

Container widgets are responsible for performing geometry management. A container widget's primary responsibility is to position each child and size itself appropriately to accommodate all its children at their desired locations. The container may also impose size constraints on its children (for example, forcing them all to be the same size). The container must also constrain the children so that they don't appear outside the container's boundaries. This is normally done by clipping the child.

To understand how different containers handle geometry management, it's important to understand the *geometry* of a widget. See "Widget Geometry" in the Introduction to this guide.

Geometry negotiation

When a widget is realized, a geometry negotiation process is initiated in all the widgets in the widget family hierarchy. Each child of the widget is given the opportunity to calculate its size. This ripples down through all the widgets in the family, resulting in a calculation of the size of each of the descendants first.

Once each child has calculated its desired size, the parent widget may attempt to determine the layout for its children. The layout that the widget performs depends on:

- the widget's layout policy
- any size that has been set for the widget
- the dimensions and desired position of each of the children.

If the application has specified a size for the widget, then it may choose to lay out the children using only that available space. This is influenced by the resize policy set for the widget. The *Pt_ARG_RESIZE_FLAGS* resource is a set of flags that determine the resizing policy for the widget. The flags specify a separate resizing policy for the width and height of the widget. If no policy is specified for either dimension, the widget doesn't attempt to resize itself in that dimension when performing the layout. Any other resize policy allows the widget to grow in that dimension to accommodate its children. For more details, see "Resize policy," below.

If the widget doesn't have a predetermined size, it tries to size itself to accommodate all the children using the appropriate layout policy. It does so by first attempting to determine a correct layout and then determining the space required to accommodate it.

The layout process determines the desired location of each child. The layout policy used by the widget controls how the layout attempts to position the children relative to each other. It must take into account the dimensions of the children. The container is responsible for fixing the position of each child, so the layout policy may choose whether or not to take into account the position attributes of the children.

In performing the layout, the widget may also take the resize policy into account. Based on this policy, it determines whether it must adjust its width or height, or change the layout to account for space restrictions. The widget tries to choose a layout that best meets the constraints imposed by any size restrictions and by the layout policy.

After determining the desired position of each of its children, the widget calculates the width and height it needs to accommodate the children at these locations. It changes its dimensions, if necessary, to fit each of the children at the desired position. If this isn't possible because the resize policy doesn't allow it, the widget recalculates the positions to fit the children within the space available.

Once the layout is successfully established, the widget sets the position of each of the children by altering the child's position attribute.

Resize policy

Any change to a widget that may affect the amount of space required to display its contents can result in the widget's resizing itself to fit its contents. This is governed by the resize policy enforced by the widget.

The resize policy affects both basic widgets and containers. A container checks its resize policy when it lays out its children to determine whether it should resize itself to accommodate all the children at their desired locations. Through the geometry negotiation process, this effect is propagated up the widget family until the size of the window widget is determined.

The *Pt_ARG_RESIZE_FLAGS* resource controls the resize policy. This resource consists of a separate set of flags for the width and the height. The values of the flags determine the conditions under which the widget recalculates the corresponding dimension. The values are checked whenever the widget is realized or its contents change.

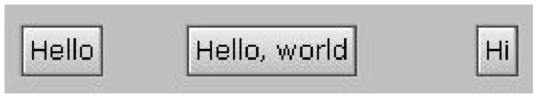
If the resize policy conflicts with the anchors, the *Pt_ARG_RESIZE_FLAGS* override *Pt_ARG_ANCHOR_OFFSETS* and *Pt_ARG_ANCHOR_FLAGS*.

The resize flags are as follows:

Pt_RESIZE_X_ALWAYS

Recalculates the widget's size whenever the value of the *x* dimension changes. The widget grows or shrinks horizontally as its contents change.

For example, the following figure shows a button with the Pt_RESIZE_X_ALWAYS flag set as the label changes from **`Hello`** to **`Hello, world`** to **`Hi`**:



Pt_RESIZE_Y_ALWAYS

Recalculates the widget's size whenever the value of the *y* dimension changes. The widget grows or shrinks vertically as its contents change.

Pt_RESIZE_XY_ALWAYS

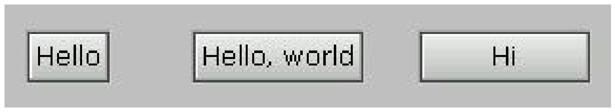
Recalculates the widget's size whenever the value of the *x* or *y* dimension changes. The widget grows or shrinks in both directions as its contents change.

The Pt_RESIZE_XY_ALWAYS flag isn't defined in PhAB. It's provided for your convenience when setting resize flags from your code.

Pt_RESIZE_X_AS_REQUIRED

Recalculates the widget's size whenever the *x* dimension changes *and* no longer fits within the current space available.

For example, the following figure shows a button with the Pt_RESIZE_X_AS_REQUIRED flag set as the label changes from **`Hello`** to **`Hello, world`** to **`Hi`**.



Pt_RESIZE_Y_AS_REQUIRED

Recalculates the widget's size whenever the *y* dimension changes *and* no longer fits within the current space available.

Pt_RESIZE_XY_AS_REQUIRED

Recalculates the widget's size whenever the *x* or *y* dimension changes *and* no longer fits within the current space available.

The Pt_RESIZE_XY_AS_REQUIRED flag isn't defined in PhAB. It's provided for your convenience when setting resize flags from your code.

These flags may also be modified by the values of another set of flags, namely:

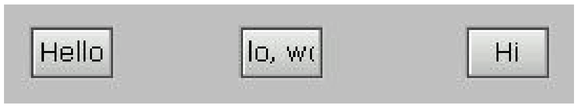
- Pt_RESIZE_X_INITIAL
- Pt_RESIZE_Y_INITIAL
- Pt_RESIZE_XY_INITIAL

The Pt_RESIZE_XY_INITIAL flag isn't defined in PhAB. It's provided for your convenience when setting resize flags from your code.

If you set any of these "initial" flags, the widget doesn't resize in response to a change in the data — it changes its size only during the geometry negotiation process *whenever it's realized*. The widget either makes itself exactly the right size for its contents, or grows to fit its contents if the dimensions it has at the time aren't large enough.

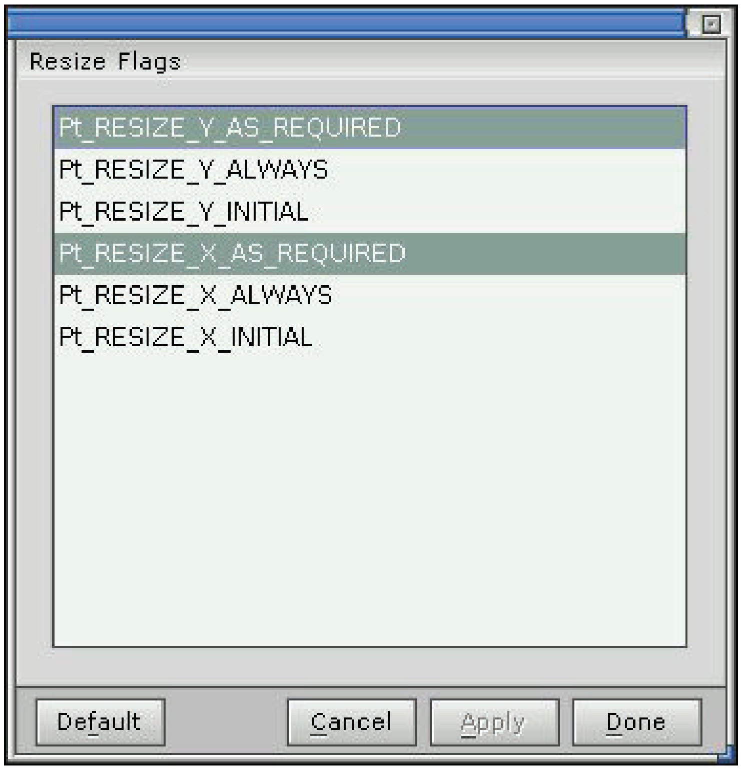
If none of the resize flags is set, the widget doesn't try to calculate its own dimensions, but uses whatever dimensions have been set by the application (thereby possibly clipping the widget's contents as a result).

For example, the following figure shows a button with no resize flags set as the label changes from **`Hello`** to **`Hello, world`** to **`Hi`**.



Setting the resize policy in PhAB

You can set these flags in PhAB by editing the container's resize flags, *Pt_ARG_RESIZE_FLAGS*, as shown below:


Resize Flags

Pt_RESIZE_Y_AS_REQUIRED (selected)
Pt_RESIZE_Y_ALWAYS
Pt_RESIZE_Y_INITIAL
Pt_RESIZE_X_AS_REQUIRED (selected)
Pt_RESIZE_X_ALWAYS
Pt_RESIZE_X_INITIAL

Setting the resize policy in your application's code

You can also set the container's resize flags in your code, if required, using the method described in the Manipulating Resources in Application Code chapter.

Bit masks are provided for controlling which bits are set. There's one bit mask for each of the x and y resize policies:

- Pt_RESIZE_X_BITS
- Pt_RESIZE_Y_BITS
- Pt_RESIZE_XY_BITS

For example, to make a container grow to fit all its children if it isn't large enough when it's realized, set both the *initial* and *required* resize flags for *x* and *y*:

```
PtSetResource (ABW_my_container, Pt_ARG_RESIZE_FLAGS,
   (Pt_RESIZE_XY_INITIAL|Pt_RESIZE_XY_AS_REQUIRED),
   Pt_RESIZE_X_BITS|Pt_RESIZE_Y_BITS);
```

To set up the argument list to clear the x resize policy:

```
PtSetResource (ABW_my_container, Pt_ARG_RESIZE_FLAGS,
   Pt_FALSE, Pt_RESIZE_X_BITS);
```

There are also some constants that simplify the setting of these flags. For example, there's a constant representing the bitmask for setting both the x and y flags simultaneously, and there are constants for each flag with the x or y shift applied. All these constants are defined in the **`<photon/PtWidget.h>`** header file.

Absolute positioning

The most basic form of layout a container can provide is to position its children without imposing any size or positioning constraints on them. In such a situation, a child widget is pinned to a particular location within the container, and the container doesn't change its size.

A widget employing this layout policy is somewhat analogous to a bulletin board. You can pin items to the bulletin board, and they stay wherever they're pinned. All container widgets can perform absolute positioning.

The easiest way to position and size each child is to use the mouse in PhAB.

To position each of the children from your application's code, you must set the *Pt_ARG_POS* resource for each child. If the widgets must be a consistent or predetermined size, you must also set the *Pt_ARG_DIM* resource for each child. The position you specify is relative to the upper left corner of the parent's canvas, so you can disregard the parent's margins when positioning children.

By default, all widgets that perform absolute positioning use a resize policy of Pt_AS_REQUIRED and Pt_INITIAL. In other words, the container's initial size is chosen when it's realized. The container is made large enough to fit all the children at their specified locations, given their size after they've been realized.

The simplest way to do absolute positioning is to place and position widgets within the main **`PtWindow`** widget of the application. If you need to create a container widget that does absolute positioning as part of another container, you can use a **`PtContainer`** widget.

Aligning widgets using groups

`PtGroup` widgets are container-class widgets that can manage the geometry of their children. You'll find them useful for aligning widgets horizontally, vertically, or in a matrix. They also have the unique ability to stretch child widgets.

PhAB extends the usefulness of this widget class by turning it into an action-oriented "Group" command. Using this command, you can select several widgets within a module and group them together to form a single group widget. If you try to select any widget in the group by clicking on it, the entire group is selected instead.

When you select a group, the Resources control panel shows the resources available to the **`PtGroup`** widget class. These include resources that allow you to align the widgets inside the group and to set up exclusive-selection behavior.

The **`PtGroup`** widget can be used to arrange a number of widgets in a row, column, or matrix. Several resources are used to control this, and they're interpreted slightly differently depending on the desired arrangement of the children.

Joining widgets into a group

To join widgets into a group:

1 Select the widgets using either a bounding box or the "Shift and click" method (as described in the Creating Widgets in PhAB chapter).

You should use "Shift–click" if you plan to align the widgets in order using the Orientation resource. The first widget you select becomes first within the group. If order isn't important or alignment isn't required, the bounding box method works just fine.

2 Do one of the following:

- Choose Group from the Widget menu.
- Press Ctrl-G.
- Click on the Group icon on PhAB's toolbar:

PhAB groups the widgets and selects the group.

Accessing widgets in a group

Although PhAB treats a group as a single widget, you can still access any of the individual widgets that make up the group. To do this, use the next and previous buttons in the Resources or Callbacks control panel, or select the widget directly in the

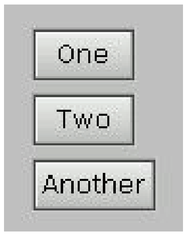
Module Tree panel. For more info, see “Selecting widgets” in the Creating Widgets in PhAB chapter.

Aligning widgets horizontally or vertically

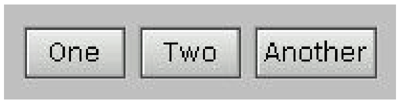
The orientation resource, *Pt_ARG_GROUP_ORIENTATION*, controls whether the group widget’s children are arranged as rows or columns. Pt_GROUP_VERTICAL causes the children to be arranged vertically, while Pt_GROUP_HORIZONTAL causes them to be arranged horizontally.

You can control the amount of space to be left between the widgets arranged in the group widget by using the *Pt_ARG_GROUP_SPACING* resource. The value of the resource gives the number of pixels to be left between widgets.

The following example shows how several children are laid out if the group uses vertical orientation with a space of five pixels between children:



If the orientation is changed to horizontal, the group appears like this:



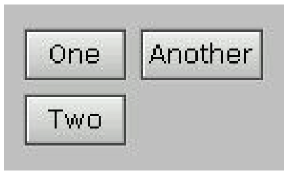
When first realized, the group widget initially sizes itself to be large enough to hold all the children after they’ve been arranged.

Aligning widgets in rows and columns

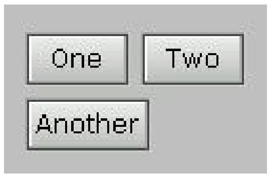
The group widget may also be used to layout children in both rows and columns for creating tables or spreadsheets by setting the value of *Pt_ARG_GROUP_ROWS_COLS* resource to some value other than one.

The interpretation of this resource depends on the orientation:

- When the orientation is vertical, this resource specifies the number of rows to be displayed; the number of columns is calculated based on the number of widgets to yield the correct number of rows.



- Otherwise, the value specifies the number of columns, and the widget calculates the number of rows.



The last row or column may have fewer widgets than the others.

When the elements of a group are laid out in rows and columns, the widgets themselves may either be tightly packed or they may be spaced out equally as rows and/or columns. This is controlled by the *Pt_ARG_GROUP_SPACING* resource.

Using the Group flags

The **`PtGroup`** widget includes a set of flags, *Pt_ARG_GROUP_FLAGS*, that can be used to control how the child widgets can be selected, sized, and stretched:

Pt_GROUP_EXCLUSIVE

: Allow only one child to be set at a time. This flag can be used to make a group of toggle buttons into radio buttons (that is, a set of mutually exclusive choices).

Pt_GROUP_EQUAL_SIZE

: Lay out the widgets in a grid, using a cell size chosen by the group based on the width of the widest child and the height of the tallest child. The dimensions of all the children are set to this size when they're laid out.

Pt_GROUP_EQUAL_SIZE_HORIZONTAL

: Make all the children the width of the widest one.

Pt_GROUP_EQUAL_SIZE_VERTICAL

: Make all the children the height of the tallest one.

Pt_GROUP_NO_SELECT_ALLOWED

: Set this flag for an exclusive group if it's valid not to have any child set. The user can unselect the currently set child by clicking on it again.

Pt_GROUP_NO_KEYS

: Don't allow the user to move inside the group by pressing the arrow keys.

Pt_GROUP_NO_KEY_WRAP_HORIZONTAL

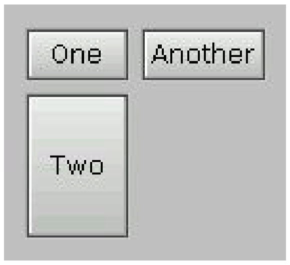
: Don't wrap around to the other side of the group when using the left and right arrow keys.

Pt_GROUP_NO_KEY_WRAP_VERTICAL

: Don't wrap around to the top or bottom of the group when using the up and down arrow keys.

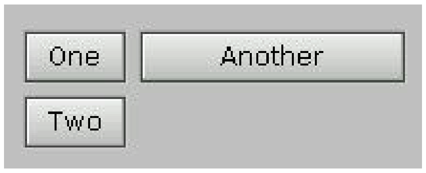
Pt_GROUP_STRETCH_VERTICAL

: Stretch the bottom row of widgets as the group expands.



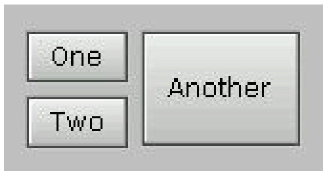
Pt_GROUP_STRETCH_HORIZONTAL

Stretch the right column of widgets as the group expands.



Pt_GROUP_STRETCH_FILL

Stretch the last widget(s) to fill the available space in the direction indicated by the orientation.



Don't set the Pt_GROUP_EQUAL_SIZE_... and Pt_GROUP_STRETCH_... flags for the same direction — the group will expand every time its extent is calculated.

For more information, see the description of the **`PtGroup`** widget in the *Widget Reference*.

Splitting apart a group

To split a group into its individual widgets:

1. Select the group.
2. Do one of the following:
   - Choose Split Apart from the Edit menu.
   - Press Ctrl-P.
   - Click on the Split icon in PhAB's toolbar:

   PhAB splits apart the widgets and removes the **`PtGroup`** container.

Constraint management using anchors

Here's a common layout situation that's not handled by any of the layout policies we've examined. Suppose a container is divided into a number of panes that have constraints on their size and position. Normally, we don't want the panes to overlap, and we want control over how the panes are resized if the container itself grows or shrinks. A constraint mechanism provides this control.

Anchors are provided as a constraint mechanism on the position and size of any widget used within any container. The position attribute and the anchors of each of the children are always used to determine their positions.

In the current version of the Photon microGUI, widgets are anchored immediately upon creation. In earlier versions, anchoring is done when the widgets are realized.

An anchor may be specified for any side of a child widget. The anchor is attached to one of the sides of the parent. It maintains the corresponding side of the child at a fixed distance, the *anchor offset* from the anchoring side of the parent. The anchor offset may also be expressed as a proportion of the canvas width or height of the parent.

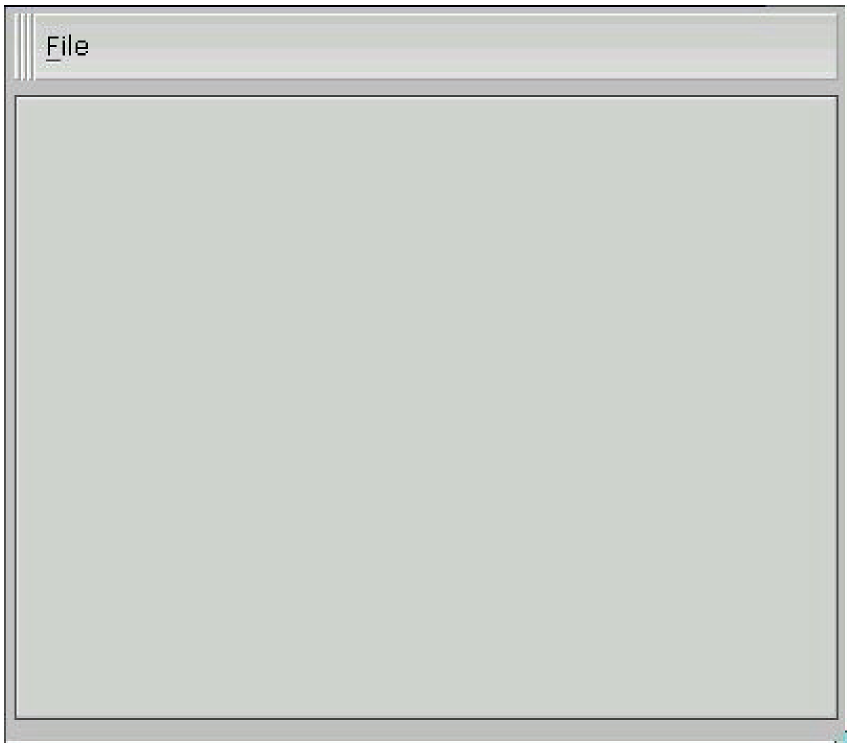
It's possible — but not always desirable — to anchor a widget's edges beyond its parent's canvas.

Any time the parent is resized, the child's position (and possibly size) is altered to maintain this relationship. If any side of the child is not anchored to the parent, it's allowed to float freely. If you explicitly set the size and/or position of an anchored widget, its anchor offsets are recalculated automatically.

When using PhAB, you don't specify anchor offsets. Instead you position the widgets at the desired offset by setting the position (*Pt_ARG_POS*) and dimension (*Pt_ARG_DIM*) resources. PhAB calculates the anchor offsets automatically, based on the relative sizes and positions of the parent and the anchored child.

You can turn anchoring on or off using the **Anchoring on/off** button on the PhAB toolbar. If a parent widget has children that are anchored, and you want to change the size of the parent widget without affecting size of the children, turn anchoring off.

The width of the child widget is influenced by the anchors for its left and right sides; the height is influenced by the anchors for the top and bottom. If either of an opposing pair of edges is allowed to float, the constraints are met by altering only the position of the widget in the corresponding dimension. This means that the widget may slide in any of the four directions to satisfy the anchor constraints. If both edges are anchored, the widget must be resized in that dimension as well.



Example of anchoring.

If the resize policy conflicts with the anchors, the *Pt_ARG_RESIZE_FLAGS* override *Pt_ARG_ANCHOR_OFFSETS* and *Pt_ARG_ANCHOR_FLAGS*.

Creating an application's main window provides a simple example of using anchor resources. The window commonly has at least two parts to it: the menu bar and the work area. If we consider an application that has a group widget for the work area, we can identify the types of anchors necessary to make it resize correctly in response to a change in the size of the window widget.

Each edge of the work area is anchored to the corresponding edge of the window. The left and top anchor offsets are set to be the same as the position attribute for the widget. This must be calculated to sit below the menu bar. The dimensions of the widget are set to the desired amount of work space.

When realized, the window positions the work area at the location specified by its position attribute. The window's size is set to be large enough to contain the work area.

If the window is resized, the width and height of the work area are resized accordingly, since all the edges are anchored. If the anchor offsets were specified correctly, the position of the widget aren't altered.

We don't have to do anything for the menu bar, because it's automatically anchored to the top and sides of the window.

Anchor resources

The *Pt_ARG_ANCHOR_FLAGS* resource (defined by **`PtWidget`**) controls the anchor attachments. Within the anchor flags, there are three flags associated with each edge of the widget; these three flags allow each edge to be anchored in one of two possible ways:

- anchored to the corresponding edge of its parent
- anchored to the opposite edge of its parent

These flags use this naming scheme:

`Pt_`*edge***`_ANCHORED_`***anchor*

where:

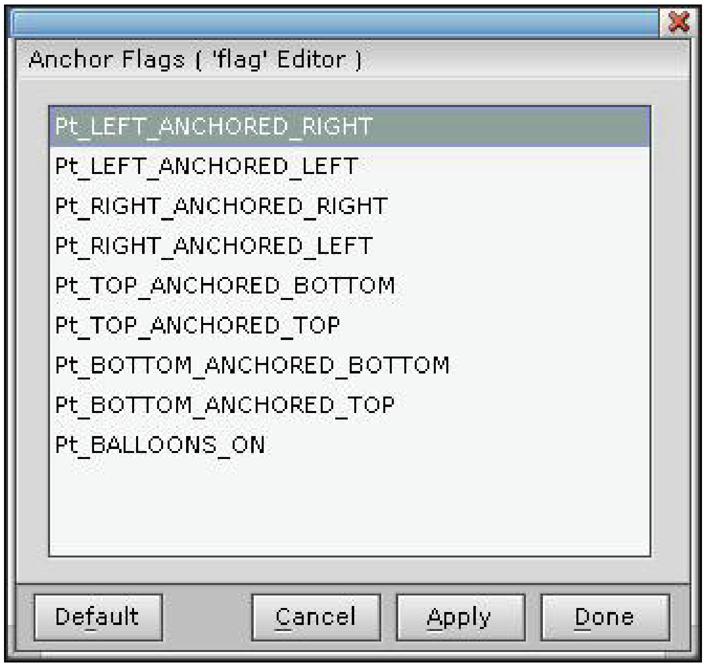
edge	is the name of the edge to be anchored, and must be TOP , LEFT, RIGHT, or BOTTOM.
anchor	is the name of the parent *edge* it's to be anchored to.

Thus, the following flags are defined:

- Pt_LEFT_ANCHORED_LEFT
- Pt_LEFT_ANCHORED_RIGHT
- Pt_RIGHT_ANCHORED_LEFT
- Pt_RIGHT_ANCHORED_RIGHT
- Pt_TOP_ANCHORED_BOTTOM
- Pt_TOP_ANCHORED_TOP
- Pt_BOTTOM_ANCHORED_BOTTOM
- Pt_BOTTOM_ANCHORED_TOP

Setting anchor flags in PhAB

To set the anchor flags, click on the anchor flags (*Pt_ARG_ANCHOR_FLAGS*) resource and use PhAB's flag editor:


Anchor Flags ('flag' Editor)

Pt_LEFT_ANCHORED_RIGHT
Pt_LEFT_ANCHORED_LEFT
Pt_RIGHT_ANCHORED_RIGHT
Pt_RIGHT_ANCHORED_LEFT
Pt_TOP_ANCHORED_BOTTOM
Pt_TOP_ANCHORED_TOP
Pt_BOTTOM_ANCHORED_BOTTOM
Pt_BOTTOM_ANCHORED_TOP
Pt_BALLOONS_ON

Setting anchor flags in your application's code

You can also set these flags from your code, using the method described in the Manipulating Resources in Application Code chapter. For convenience, each set of flags has an associated bit mask:

- Pt_LEFT_IS_ANCHORED — isolates the bits responsible for specifying an anchor for the *left* edge.
- Pt_RIGHT_IS_ANCHORED — isolates the bits responsible for specifying an anchor for the *right* edge.
- Pt_TOP_IS_ANCHORED — isolates the bits responsible for specifying an anchor for the *top* edge.
- Pt_BOTTOM_IS_ANCHORED — isolates the bits responsible for specifying an anchor for the *bottom* edge.

So to set the left and right edge for our menu bar in the example above, the argument list element would be initialized as follows:

```
PtSetArg(&arg[n], Pt_ARG_ANCHOR_FLAGS,
    Pt_LEFT_ANCHORED_LEFT|Pt_RIGHT_ANCHORED_RIGHT|
    Pt_TOP_ANCHORED_TOP,
    Pt_LEFT_IS_ANCHORED|Pt_RIGHT_IS_ANCHORED|
    Pt_TOP_IS_ANCHORED);
```

When setting anchor flags from your application's code, all the anchor offsets are specified using the *Pt_ARG_ANCHOR_OFFSETS* resource. This resource takes a **PhRect_t** structure (see the Photon *Library Reference*) as a value. The upper left corner of the rectangle is used to specify the anchor offset for the top and left edges of the widget, and the lower right corner of the rectangle indicates the anchor offset for the right and bottom edges.

Using layouts

If you wish to maintain more complex relationships among the positions of children relative to the container, or relative to each other, consider using layouts. A *layout* is a property of container widgets that sizes and organizes its children when the container or its children change size.

Layouts are an alternative to the direct geometry management of the widgets using *Pt_ARG_POS*, *Pt_ARG_DIM*, and *Pt_ARG_AREA*. Instead of calculating positions and sizes of the widgets in your code, your code describes the rules of how the widgets should be laid out. This proves to be very efficient and doesn't require additional code to handle resize operations. Layouts usually guarantee that widgets are not overlapping, which is an important feature for many multilingual applications. Each type of layout has its own method of sizing and positioning the children in its container. Each child widget can have detailed info of how it should be sized and positioned in its parent. If a child is a container, it can have its own layout for its children.

These are the layout types in the Photon Widget Library:

- PtFillLayout — a simple layout that "fills" widgets into a container.
- PtRowLayout — a layout that organizes widgets into rows.
- PtGridLayout — a layout that organizes widgets into a grid pattern.
- PtAnchorLayout — the default layout using widget anchors.

You set layouts using resources in the container widget. You can refine the way layouts place widgets by setting resources on the child widgets. Below are the resources involved in using layouts.

`PtContainer` has these layout-related resources:

Pt_ARG_LAYOUT
: This is a generic resource that lets you set the active layout of the container and optionally the layout information structure. Here's an example of setting both for a fill layout:

```
PtArg_t               args[20];
int                     i = 0;
PtFillLayoutInfo_t      info;

info.type = Pt_LAYOUT_HORIZONTAL;
info.spacing = 2;
PtSetArg( &args[i++], Pt_ARG_LAYOUT, PtFillLayout, &info );
```

*Pt_ARG_LAYOUT_INFO**
: This is a generic resource to specify layout info, which you can use to set or get any of the *Pt_ARG_*_LAYOUT_INFO* resources.

Pt_CB_LAYOUT
: This callback is called when the layout is about to start laying out children and when it's finished. The *cbinfo*->*reason_subtype* indicates which situation called it, and is one of:

- Pt_LAYOUT_INIT — layout is about to start laying out
- Pt_LAYOUT_DONE — layout has finished laying out children.

You can use this callback to fine-tune the layout procedure.

Pt_ARG_LAYOUT_TYPE

This is an alternative method of setting or getting the active layout type and optionally the layout information structure (*Pt_ARG_*_LAYOUT_INFO*). Here's an example of setting both for a fill layout:

```
PtArg_t              args[20];
int                    i = 0;
PtFillLayoutInfo_t     info;

info.type = Pt_LAYOUT_HORIZONTAL;
info.spacing = 2;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_TYPE, Pt_FILL_LAYOUT,
          &info );
```

*Pt_ARG_FILL_LAYOUT_INFO**

Used only to set the **`PtFillLayoutInfo_t`** structure for the **`PtFillLayout`** layout type.

*Pt_ARG_ROW_LAYOUT_INFO**

Used only to set the **`PtRowLayoutInfo_t`** structure for the **`PtRowLayout`** layout type.

*Pt_ARG_GRID_LAYOUT_INFO**

Used only to set the **`PtGridLayoutInfo_t`** structure for the **`PtGridLayout`** layout type.

* You can set all the *Pt_ARG_*_LAYOUT_INFO* resources at the same time. The active layout will look for its type of info structure to perform the layout.

`PtWidget` has these layout-related resources:

*Pt_ARG_LAYOUT_DATA**

This is the generic method of setting/getting layout data for a widget.

*Pt_ARG_ROW_LAYOUT_DATA**

Used only to set the **`PtRowLayoutData_t`** structure for the **`PtRowLayout`** layout type.

*Pt_ARG_GRID_LAYOUT_DATA**

Used only to set the **`PtGridLayoutData_t`** structure for the **`PtGridLayout`** layout type.

Pt_ARG_EFLAGS

If this resource has Pt_SKIP_LAYOUT set, the container doesn't apply the layout to the widget. This is useful if you have widgets that you want to place by hand.

* You can set all the *Pt_ARG_*_LAYOUT_DATA* resources at the same time. The active layout of the widget's parent will look for its type of data structure to perform the layout.

These are the structures used to set layout-related resources:

Layout type (*Pt_ARG_LAYOUT* in the container)	**Layout Info Structure (*Pt_ARG_LAYOUT_INFO* or *Pt_ARG_*_LAYOUT_INFO* in the container)**	**Layout Data Structure (*Pt_ARG_LAYOUT_DATA* or *Pt_ARG_*_LAYOUT_DATA* in child widgets)**
`PtFillLayout`	**`PtFillLayoutInfo_t`**	N/A
`PtRowLayout`	**`PtRowLayoutInfo_t`**	**`PtRowLayoutData_t`**
`PtGridLayout`	**`PtGridLayoutInfo_t`**	**`PtGridLayoutData_t`**

PtFillLayout

This is a simple type of a layout. It organizes children in one row or one column and makes them the same size. The width of each child will be at least as wide as the widest child's width. The same rule applies to the height.

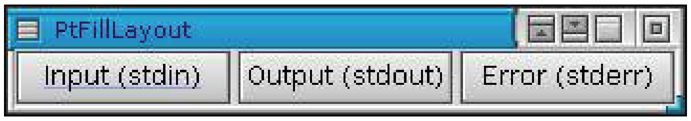
You can set the layout's options using the layout information structure, **`PtFillLayoutInfo_t`**. See the **`PtContainer`** resource *Pt_ARG_FILL_LAYOUT_INFO* for a description of the **`PtFillLayoutInfo_t`** structure.

There is no layout data structure for the *Pt_ARG_LAYOUT_DATA* resource of children for this layout type.

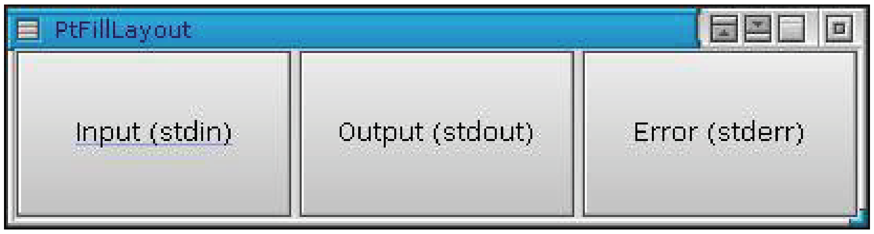
Let's take a look at this example:

```
/* fill_layout.c example */

#include <Pt.h>

int
main( int argc, char *argv[] )
{
PtWidget_t              *window;
PtArg_t                  args[20];
int                      i = 0;
PtFillLayoutInfo_t       info;
/* Set layout type and spacing between its children */
info.type = Pt_LAYOUT_HORIZONTAL;
info.spacing = 2;

/* Create a window */
i = 0;
PtSetArg( &args[i++], Pt_ARG_WINDOW_TITLE, "PtFillLayout",
0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT, PtFillLayout, &info
);
if( NULL == ( window = PtAppInit( NULL, &argc, argv, i,
args ) ) ) {
        perror( "PtAppInit()" );
        return 1;
}

/* Create buttons */
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Input (stdin)",
0 );
PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Output (stdout)", 0
);
PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Error (stderr)", 0
);
PtCreateWidget( PtButton, window, i, args );

PtRealizeWidget( window );
PtMainLoop();
return 0;
}
```

Build and run the application:

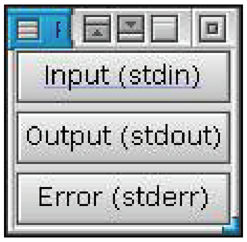


Fill layout initial.

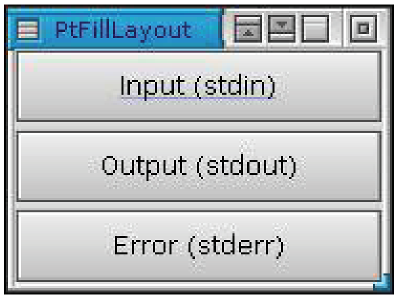


Fill layout after resizing.

Now change the type from *Pt_LAYOUT_HORIZONTAL* to Pt_LAYOUT_VERTICAL and we get this:



Vertical fill layout.



Vertical fill layout after resizing.

PtRowLayout

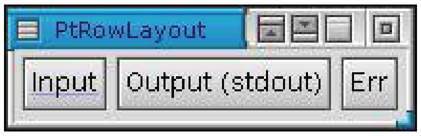
The row layout is similar to the fill layout but it has a very important difference — it can wrap, so it can place children in more than one row or column. When the *flags* member of the **PtRowLayoutInfo_t** structure has the Pt_ROW_WRAP flag set, and there is not enough space in the row to fit the next child, the child is moved to the beginning of the next row. The row layout also has margins (the space around all the widgets) and the children of the container can have their own data (in their *Pt_ARG_LAYOUT_DATA* resources) which can be used to fine-tune the layout.

See the **PtContainer** resource *Pt_ARG_ROW_LAYOUT_INFO* for a description of the **PtRowLayoutInfo_t** structure. The **PtRowLayoutData_t** structure is described in the **PtWidget** resource *Pt_ARG_ROW_LAYOUT_DATA*.

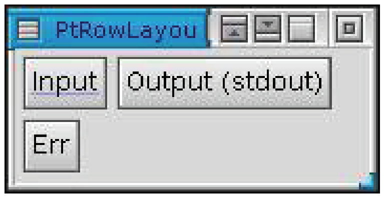
Let's take a look at an example:

```
#include <Pt.h>

int
main( int argc, char *argv[] )
{
PtWidget_t     *window, *b1, *b2, *b3;
PtArg_t         args[20];
int             i = 0;
PtRowLayoutInfo_t       info;
/* Set layout type and layout info */
info.type = Pt_LAYOUT_HORIZONTAL;
info.flags = Pt_ROW_PACK | Pt_ROW_WRAP;
info.margin.ul.x = 3;
info.margin.ul.y = 3;
info.margin.lr.x = 3;
info.margin.lr.y = 3;
info.h_spacing = 3;
info.v_spacing = 3;

/* Create a window */
i = 0;
PtSetArg( &args[i++], Pt_ARG_WINDOW_TITLE, "PtRowLayout",
0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT, PtRowLayout, &info
);
if( NULL == ( window = PtAppInit( NULL, &argc, argv, i,
args ) ) ) {
        perror( "PtAppInit()" );
        return 1;
}

/* Create buttons */
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Input", 0 );
b1 = PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Output (stdout)", 0
);
b2 = PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Err", 0 );
b3 = PtCreateWidget( PtButton, window, i, args );

PtRealizeWidget( window );
PtMainLoop();
return 0;
}
```

When you build and run this application, you'll see:



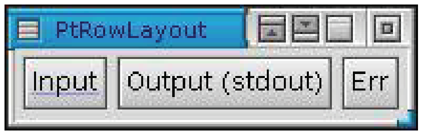
Initial row layout.



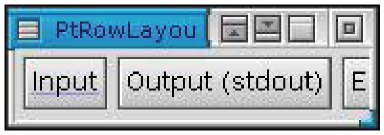
Row layout after resizing.

Let’s look at what happens when we set and unset some **`PtRowLayoutInfo_t`** *flags*.

When the Pt_ROW_WRAP is not set, content is clipped to the size of the window. To test this case, you need to clear resize flags for *Pt_RESIZE_X_BITS* on the window:

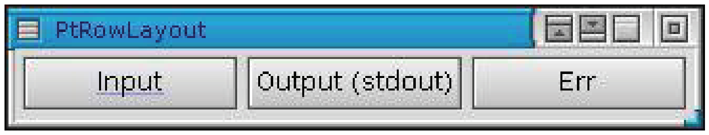


Initial window without Pt_ROW_WRAP *set.*

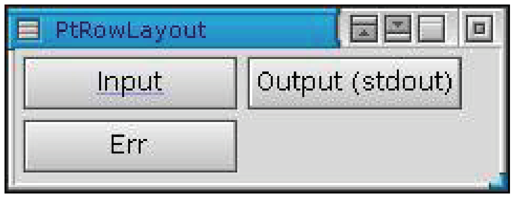


After shrinking without Pt_ROW_WRAP *set.*

When Pt_ROW_PACK is not set, all the buttons are the same size:

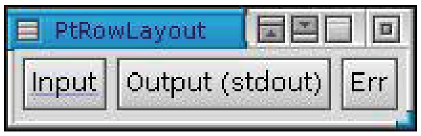


Initial window without Pt_ROW_PACK *set.*

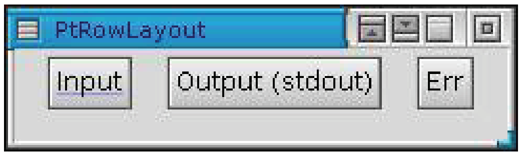


After resizing without Pt_ROW_PACK *set.*

When Pt_ROW_JUSTIFY is set, extra space is distributed evenly between the buttons:

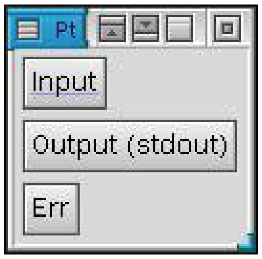


Initial window with Pt_ROW_JUSTIFY set.

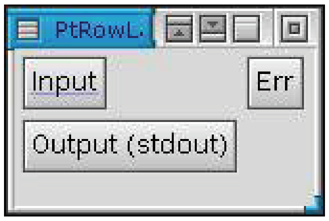


After resizing with Pt_ROW_JUSTIFY set.

Now see what happens when we set the layout type to Pt_LAYOUT_VERTICAL:



Initial window with Pt_LAYOUT_VERTICAL set.



After resizing with Pt_LAYOUT_VERTICAL set.

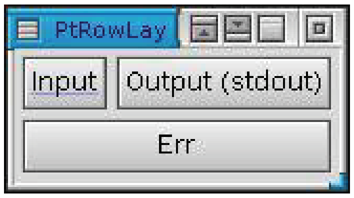
If we replace this code:

```
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Err", 0 );
b3 = PtCreateWidget( PtButton, window, i, args );
```

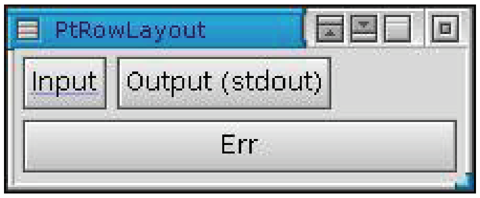
with:

```
PtRowLayoutData_t data = { {0,0}, Pt_ROW_FILL | Pt_ROW_WRAP_BEFORE
};
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Err", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data,
PtRowLayout );
b3 = PtCreateWidget( PtButton, window, i, args );
```

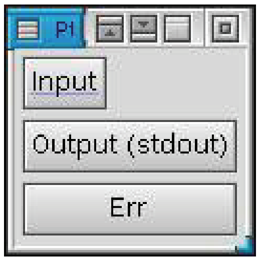
The results are:



Initial window.



After stretching.



After shrinking.

What happened? The **Err** button started its own row because the Pt_ROW_WRAP_BEFORE flag was set. Then the **Err** button was stretched to fill all the extra space because the Pt_ROW_FILL flag was set.

PtGridLayout

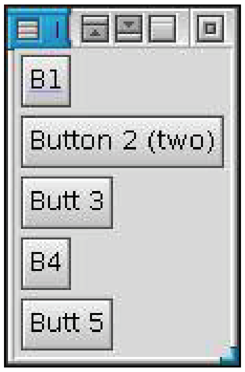
The Grid Layout is a very powerful and widely used layout. This layout arranges children in a grid, which is why it's sometimes called a "table layout." The layout info structure **PtGridLayoutInfo_t** contains a number of members that adjust the layout operation. Children of the layout container can have layout data (**PtGridLayoutData_t**) attached to them to fine-tune the layout.

See the **PtContainer** resource *Pt_ARG_GRID_LAYOUT_INFO* for a description of the **PtGridLayoutInfo_t** structure. The **PtGridLayoutData_t** structure is described in the **PtWidget** resource *Pt_ARG_GRID_LAYOUT_DATA*.

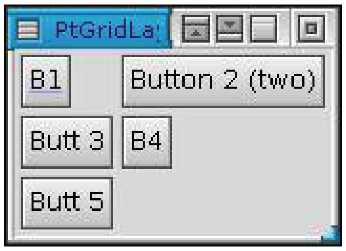
Let’s look at some examples that use a grid layout:

```
#include <Pt.h>

int
main( int argc, char *argv[] )
{
PtWidget_t              *window;
PtArg_t                  args[20];
int                      i = 0;
PtGridLayoutInfo_t       info = PtGridLayoutInfoDflts;
PtGridLayoutData_t       data = PtGridLayoutDataDflts;

info.n_cols = 3;
info.flags = 0;
data.flags = Pt_H_ALIGN_BEGINNING;

/* Create a window */
i = 0;
PtSetArg( &args[i++], Pt_ARG_WINDOW_TITLE, "PtGridLayout",
0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT, PtGridLayout, &info
);
if( NULL == ( window = PtAppInit( NULL, &argc, argv, i,
args ) ) ) {
        perror( "PtAppInit()" );
        return 1;
}

/* Create buttons */
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "B1", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Button 2 (two)", 0
);
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Butt 3", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "B4", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
PtCreateWidget( PtButton, window, i, args );

i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Butt 5", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
PtCreateWidget( PtButton, window, i, args );

PtRealizeWidget( window );
PtMainLoop();
return 0;
}
```

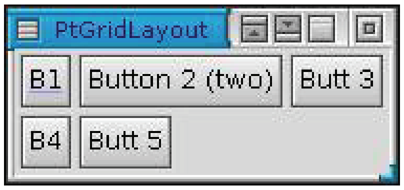
Build and run the sample, and you'll see:



One column (n_cols=1).

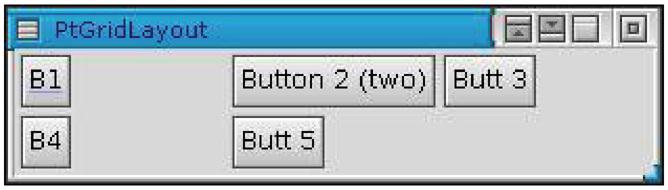


Two columns (n_cols=2).



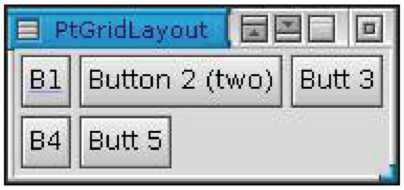
Three columns (n_cols=3).

Change the code `info.flags = 0;` to `info.flags |= Pt_EQUAL_COLS;`, and we get this:

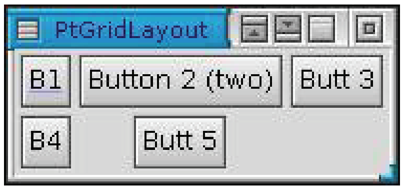


Now let's see how different cell alignments work.

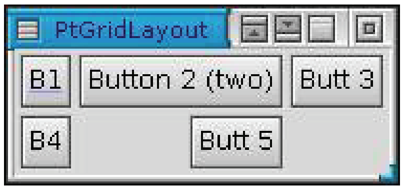
Set *data.flags* to Pt_H_ALIGN_BEGINNING:



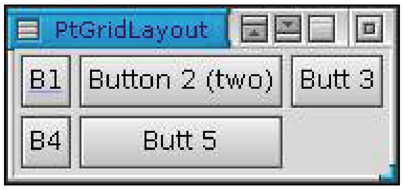
Set *data.flags* to Pt_H_ALIGN_CENTER:



Set *data.flags* to Pt_H_ALIGN_END:



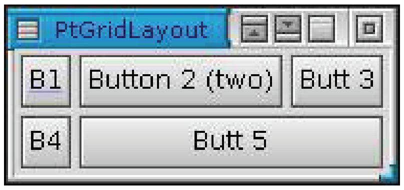
Set *data.flags* to Pt_H_ALIGN_FILL:



Let's see how spans work. Set the horizontal span of **Butt 5** to 2 columns. To do that, add this code before **Butt 5** is created:

```
data.flags = Pt_H_ALIGN_FILL;
data.h_span = 2;
```

The result is:



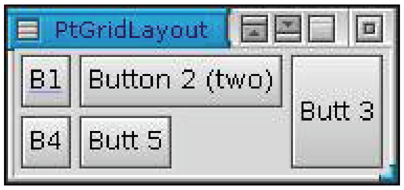
Let's test the vertical span. Undo the changes for the previous example and add the following code before the **Butt 3** widget is created:

```
data.flags = Pt_V_ALIGN_FILL;
data.v_span = 2;
```

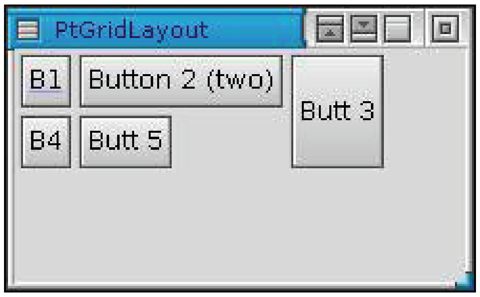
and add the following code after the widget creation call:

```
data = PtGridLayoutDataDflts;
```

The result:



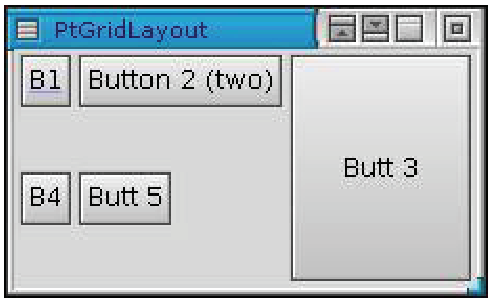
Finally, let's see how the grab flags work. If we resize the window we get this:



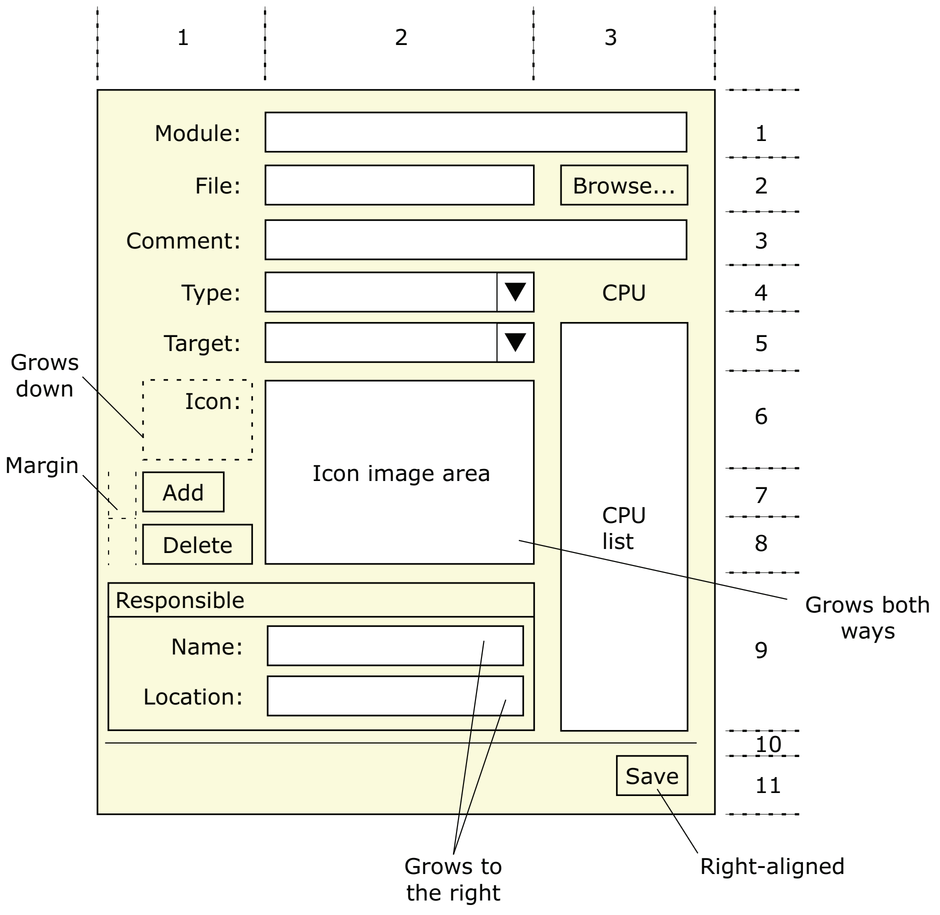
If we change the flags for the **Butt 3** widget to:

```
data.flags = Pt_ALIGN_FILL_BOTH | Pt_GRAB_BOTH;
```

We get this:



Let's take a look at a more complicated example. Here is the draft drawing of what we need to program



Sketch of grid layout.

The code example below implements this grid-layout plan. Please note that this example is for demonstration purposes only and there are no optimizations in the code.

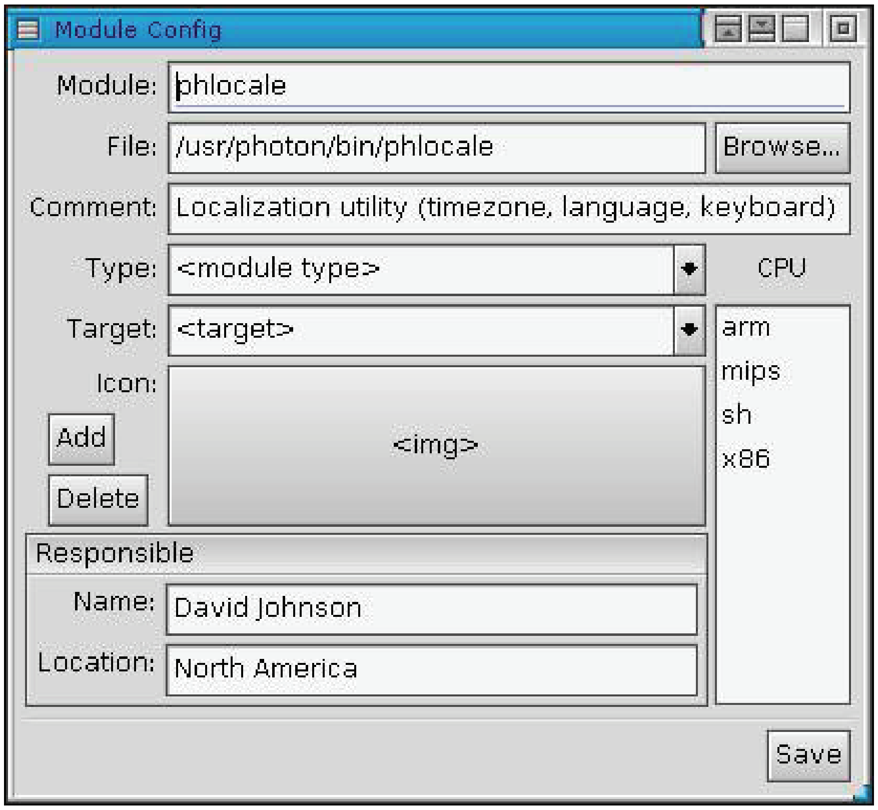
```
#include <Pt.h>

int
main( int argc, char *argv[] )
{
PtWidget_t              *window, *w, *ctnr;
PtArg_t                         args[20];
int                                     i = 0;
PtGridLayoutInfo_t      info;
PtGridLayoutData_t      data;

info = PtGridLayoutInfoDflts;
info.n_cols = 3;
info.margin.ul.x = info.margin.ul.y = 5;
info.margin.lr.x = info.margin.lr.y = 5;

/* Create a window */
i = 0;
PtSetArg( &args[i++], Pt_ARG_WINDOW_TITLE, "Module Config", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT, PtGridLayout, &info );
if( NULL == ( window = PtAppInit( NULL, &argc, argv, i, args ) ) ) {
        perror( "PtAppInit()" );
        return 1;
}
```

```
data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Module:", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL | Pt_H_GRAB;
data.h_span = 2;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "phlocale", 0 );
PtCreateWidget( PtText, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "File:", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL | Pt_H_GRAB;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "/usr/photon/bin/phlocale", 0 );
PtCreateWidget( PtText, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Browse...", 0 );
PtCreateWidget( PtButton, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Comment:", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL | Pt_H_GRAB;
data.h_span = 2;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Localization utility
(timezone, language, keyboard)", 0 );
PtCreateWidget( PtText, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Type:", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "<module
type>", 0 );
w = PtCreateWidget( PtComboBox, window, i, args );
{
        const char *list[] = { "Binary", "Config", "Development", "Support" };
        PtListAddItems( w, list, sizeof(list)/sizeof(list[0]), 1
);
}

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_CENTER | Pt_V_ALIGN_CENTER;
i = 0;
```

```
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "CPU", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Target:", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL | Pt_V_ALIGN_CENTER;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "<target>",
0 );
w = PtCreateWidget( PtComboBox, window, i, args );
{
        const char *list[] = { "OS targets", "GUI target", "DEV target" };
        PtListAddItems( w, list, sizeof(list)/sizeof(list[0]), 1 );
}

data = PtGridLayoutDataDflts;
data.flags = Pt_V_ALIGN_FILL | Pt_H_ALIGN_FILL;
data.v_span = 5;
data.hint.h = 30;       // This is important to keep the list "in bounds"
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, 0 );
w = PtCreateWidget( PtList, window, i, args );
{
        const char *list[] = { "arm", "mips", "sh", "x86" };
        PtListAddItems( w, list, sizeof(list)/sizeof(list[0]), 1 );
}

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END | Pt_V_ALIGN_BEGINNING | Pt_V_GRAB;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Icon:", 0 );
PtCreateWidget( PtLabel, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_ALIGN_FILL_BOTH | Pt_GRAB_BOTH;
data.v_span = 3;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "<img>", 0 );
PtCreateWidget( PtButton, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_BEGINNING | Pt_V_ALIGN_BEGINNING;
data.margin.ul.x = 10;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Add", 0 );
PtCreateWidget( PtButton, window, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_BEGINNING | Pt_V_ALIGN_BEGINNING;
data.margin.ul.x = 10;
i = 0;
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Delete", 0 );
PtCreateWidget( PtButton, window, i, args );

data = PtGridLayoutDataDflts;
data.h_span = 2;
data.flags = Pt_H_ALIGN_FILL;
i = 0;

PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, 0 );
PtSetArg( &args[i++], Pt_ARG_CONTAINER_FLAGS,
        Pt_SHOW_TITLE|Pt_ETCH_TITLE_AREA | Pt_GRADIENT_TITLE_AREA,
        Pt_SHOW_TITLE|Pt_ETCH_TITLE_AREA | Pt_GRADIENT_TITLE_AREA
);
```

```
PtSetArg( &args[i++], Pt_ARG_TITLE, "Responsible", 0 );
ctnr = PtCreateWidget( PtPane, window, i, args );
{
        PtGridLayoutInfo_t info = PtGridLayoutInfoDflts;
        info.n_cols = 2;

        PtSetResource( ctnr, Pt_ARG_LAYOUT, PtGridLayout, &info );

        i = 0;
        data = PtGridLayoutDataDflts;
        data.flags = Pt_H_ALIGN_END;
        PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, 0 );
        PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Name:", 0 );
        PtCreateWidget( PtLabel, ctnr, i, args );

        i = 0;
        data = PtGridLayoutDataDflts;
        data.flags = Pt_H_GRAB | Pt_H_ALIGN_FILL;
        PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, 0 );
        PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "David Johnson", 0 );
        PtCreateWidget( PtText, ctnr, i, args );

        i = 0;
        data = PtGridLayoutDataDflts;
        data.flags = Pt_H_ALIGN_END;
        PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, 0 );
        PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Location:", 0 );
        PtCreateWidget( PtLabel, ctnr, i, args );

        i = 0;
        data = PtGridLayoutDataDflts;
        data.flags = Pt_H_GRAB | Pt_H_ALIGN_FILL;
        PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, 0 );
        PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "North America", 0 );
        PtCreateWidget( PtText, ctnr, i, args );

}

i = 0;
data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_FILL;
data.h_span = 3;
data.hint.h = 6;
PtSetArg( &args[i++], Pt_ARG_SEP_FLAGS, Pt_SEP_HORIZONTAL, 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtCreateWidget( PtSeparator, window, i, args );

i = 0;
data = PtGridLayoutDataDflts;
data.flags = Pt_H_ALIGN_END;
data.h_span = 3;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "Save", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout );
PtCreateWidget( PtButton, window, i, args );

PtRealizeWidget( window );
PtMainLoop();
return 0;
}
```

Here’s what the resulting interface looks like:



Module: phlocale

File: /usr/photon/bin/phlocale

Comment: Localization utility (timezone, language, keyboard)

Type: <module type>

Target: <target>

Icon: <img>

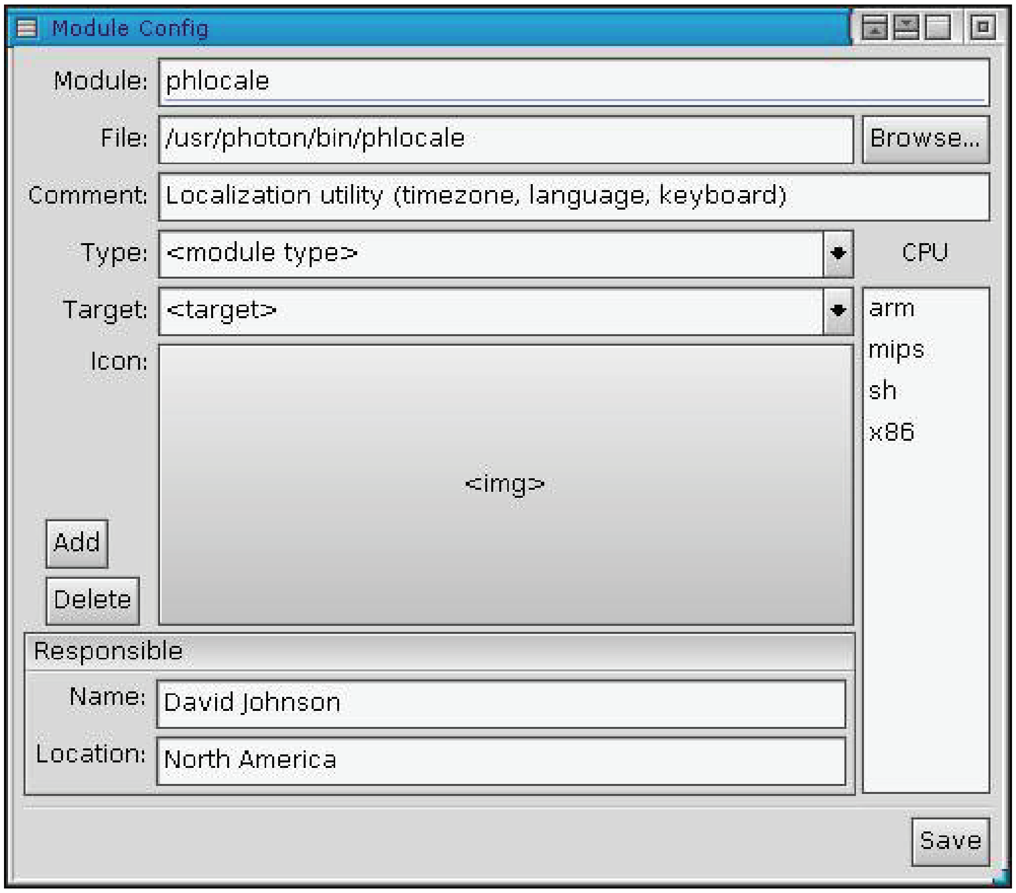
arm
mips
sh
x86

Responsible

Name: David Johnson

Location: North America

Complex grid layout — initial.


Module Config

Module: phlocale

File: /usr/photon/bin/phlocale

Browse...

Comment: Localization utility (timezone, language, keyboard)

Type: <module type>

CPU

Target: <target>

arm
mips
sh
x86

Icon:

<img>

Add

Delete

Responsible

Name: David Johnson

Location: North America

Save

Complex grid layout — resize.

Using hints

Let's take a look at an interesting way of using hints with the grid layout. We will use a "draggable" separator to obtain the user's input and adjust the width hint of one of the widgets in the layout.

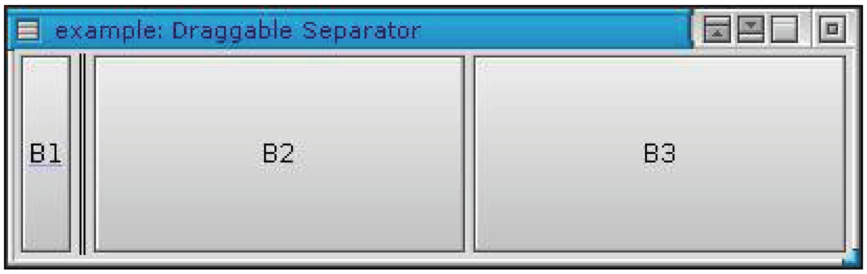
This is what we'll do:

1. The grid layout contains four columns and one row.
2. Buttons are placed in columns 1, 3, and 4.
3. A separator is placed in column 2.
4. The separator is draggable (the Pt_SEP_DRAGGABLE bit is set in *Pt_ARG_SEP_FLAGS*.
5. When the separator is dragged, the user's callback (*Pt_CB_SEP_DRAG*) checks that the separator position is in the acceptable range, and adjusts the hint of the button in column 1.

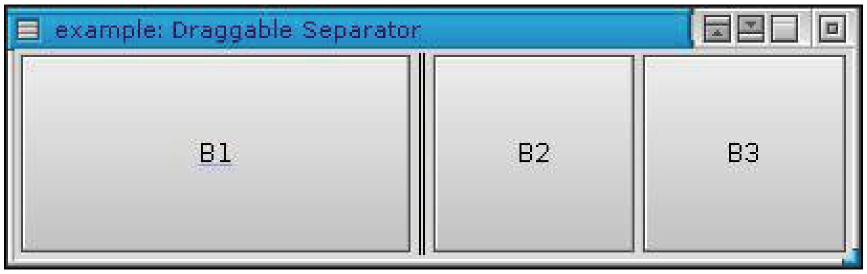
Here is the actual implementation:

```
#include <Pt.h>

#define MARGIN_W 2
#define SPACING_W        2
#define SEP_WIDTH        5

/* These variables contain the limits for the separator movement
*/
static short col1_min_width, col2to4_width;

/*****************************************************************
DRAG_CB() ***/
int
drag_cb( PtWidget_t *widget, void *data, PtCallbackInfo_t *cbinfo)
{
PtSeparatorCallback_t   *cb = (PtSeparatorCallback_t*)cbinfo->cbdata;
PtWidget_t              *b1 = (PtWidget_t*)data;
short                   pos_x = cb->rect.ul.x;
PtGridLayoutData_t      *_grid_data, grid_data;
const PhRect_t          *p_canvas = PtGetCanvas( widget->parent
);
short                   parent_width = p_canvas->lr.x - p_canvas->ul.x&
#SPACE
+ 1;

/* Validate the pos.x of the separator, so that it stays in the
range */
pos_x = max( col1_min_width, pos_x );
pos_x = min( pos_x, parent_width - col2to4_width );

PtGetResource( b1, Pt_ARG_LAYOUT_DATA, &_grid_data, PtGridLayout
);
grid_data = *_grid_data;
grid_data.hint.w = pos_x - (MARGIN_W + SPACING_W);
PtSetResource( b1, Pt_ARG_LAYOUT_DATA, &grid_data, PtGridLayout
);

return Pt_CONTINUE;
}

/*******************************************************************
MAIN() ***/
int
main( int argc, char *argv[] )
{
int                     i;
PtWidget_t      *window, *sep, *b1, *b2, *b3;
PtArg_t         args[20];
PhDim_t                 dim;
PtGridLayoutData_t      data;
PtGridLayoutInfo_t      info = PtGridLayoutInfoDflts;

info.n_cols = 4;
info.margin.ul.x = info.margin.lr.x = MARGIN_W;
info.h_spacing = SPACING_W;

i = 0;
PtSetArg( &args[i++], Pt_ARG_WINDOW_TITLE, "example: Draggable
Separator", 0 );
PtSetArg( &args[i++], Pt_ARG_HEIGHT, 100, 0 );
PtSetArg( &args[i++], Pt_ARG_WIDTH,  400, 0 );
PtSetArg( &args[i++], Pt_ARG_CONTAINER_FLAGS, Pt_AUTO_EXTENT,
Pt_AUTO_EXTENT );
PtSetArg( &args[i++], Pt_ARG_LAYOUT, PtGridLayout, &info
);

if( NULL == ( window = PtAppInit( NULL, &argc, argv, i,
args ) ) ) {
        perror( "PtAppInit()" );
        return 1;
}

data = PtGridLayoutDataDflts;
data.flags = Pt_V_ALIGN_FILL | Pt_H_ALIGN_FILL;
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "B1", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
b1 = PtCreateWidget( PtButton, NULL, i, args );

data = PtGridLayoutDataDflts;
data.hint.w = SEP_WIDTH;
data.flags = Pt_V_ALIGN_FILL;
i = 0;
PtSetArg( &args[i++], Pt_ARG_SEP_ORIENTATION, Pt_VERTICAL,
0 );
PtSetArg( &args[i++], Pt_ARG_SEP_TYPE, Pt_DOUBLE_LINE,
0 );
PtSetArg( &args[i++], Pt_ARG_SEP_FLAGS, Pt_SEP_DRAGGABLE, Pt_SEP_DRAGGABLE&
#SPACE
);
PtSetArg( &args[i++], Pt_ARG_CURSOR_TYPE, Ph_CURSOR_DRAG_HORIZONTAL,
0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
sep = PtCreateWidget( PtSeparator, NULL, i, args );
PtAddCallback( sep, Pt_CB_SEP_DRAG, drag_cb, b1 );

i = 0;
data = PtGridLayoutDataDflts;
data.flags = Pt_V_ALIGN_FILL | Pt_H_ALIGN_FILL | Pt_GRAB_BOTH;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "B2", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
b2 = PtCreateWidget( PtButton, NULL, i, args );

data = PtGridLayoutDataDflts;
data.flags = Pt_V_ALIGN_FILL | Pt_H_ALIGN_FILL | Pt_GRAB_BOTH;
i = 0;
PtSetArg( &args[i++], Pt_ARG_TEXT_STRING, "B3", 0 );
PtSetArg( &args[i++], Pt_ARG_LAYOUT_DATA, &data, PtGridLayout
);
b3 = PtCreateWidget( PtButton, NULL, i, args );

/* Calculate the limits of the dragging */

/* Column 1: width of the b1 + left margin + spacing */
PtGetPreferredDim( b1, 0, 0, &dim );
col1_min_width = dim.w + MARGIN_W + SPACING_W;

/* Column 2 to 4:
 *      separator width + spacing + b2 width + spacing + b3 width
+ right margin
 */
col2to4_width = SEP_WIDTH + SPACING_W;

PtGetPreferredDim( b2, 0, 0, &dim );
col2to4_width += dim.w + info.h_spacing;

PtGetPreferredDim( b3, 0, 0, &dim );
col2to4_width += dim.w + info.margin.lr.x;

PtRealizeWidget( window );

PtMainLoop(  );

return 0;
}
```

This is how the application behaves when resized:



Initial Hints example.



Hints example after resizing.

Enforcing position or size constraints without anchors or layouts

If you wish to maintain more complex relationships among the positions of children relative to the container, or relative to each other, and you don't want to use layouts, you *must* catch resize events for the container. The **`PtContainer`** widget class provides a resize callback, *Pt_CB_RESIZE*, that you can use for this purpose.

The *cbdata* member of the **`PtCallbackInfo_t`** structure (see the Photon *Widget Reference*) is a pointer to a **`PtContainerCallback_t`** structure that contains at least the following:

`PhRect_t` *old_size*

A **`PhRect_t`** structure (see the Photon *Library Reference*) that defines the former size of the container.

`PhRect_t` *new_size*

A **`PhRect_t`** structure that defines the new size.

Chapter 9
Generating, Compiling, and Running Code

In this chapter. . .

PhAB automatically generates everything that's required to turn your application into a working executable, including:

- code to handle the UI portion of your application
- stub C and/or C++ files for application-specific callbacks, module-setup functions, application-initialization functions, and so on
- all the files required to compile and link the application—Makefile, global header, main-line file, and prototype file.

By doing all this, PhAB lets you get on with the job of writing the code that implements your application's main functionality.

The way you build PhAB projects depends on whether you're running PhAB standalone or from the IDE. This chapter describes building and generating code using PhAB standalone, and indicates where there are differences when using the IDE. For more information on building and generating code in the IDE, see the "Building Projects" section in the Developing Programs chapter of the IDE *User's Guide*.

For most code generation, you can use the Build menu. However, you can also generate some C and C++ stub files on the spot when using various dialogs to develop your application; use the icons located next to function or filename fields:

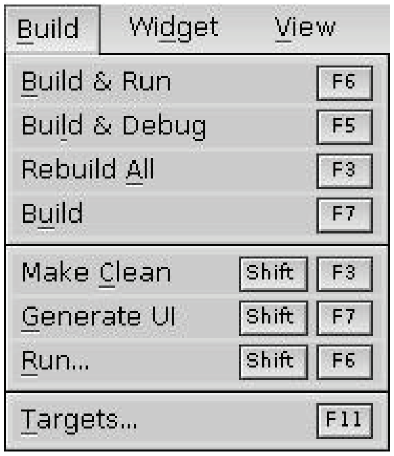
This means you're free to edit a callback function while still in the process of attaching it to the widget. You don't have to go into the Build menu, generate code from there, and then come back to write the function.

Using the Build menu

If you are using PhAB from the IDE, you can use PhAB's Build menu to generate the application's user interface code. All other build and run functions are handled by the IDE.

In standalone PhAB, the Build menu is the development center for building your applications. From this menu you can:

- generate the application code
- make (compile and link) your application
- debug your application
- run your application
- configure your targets.



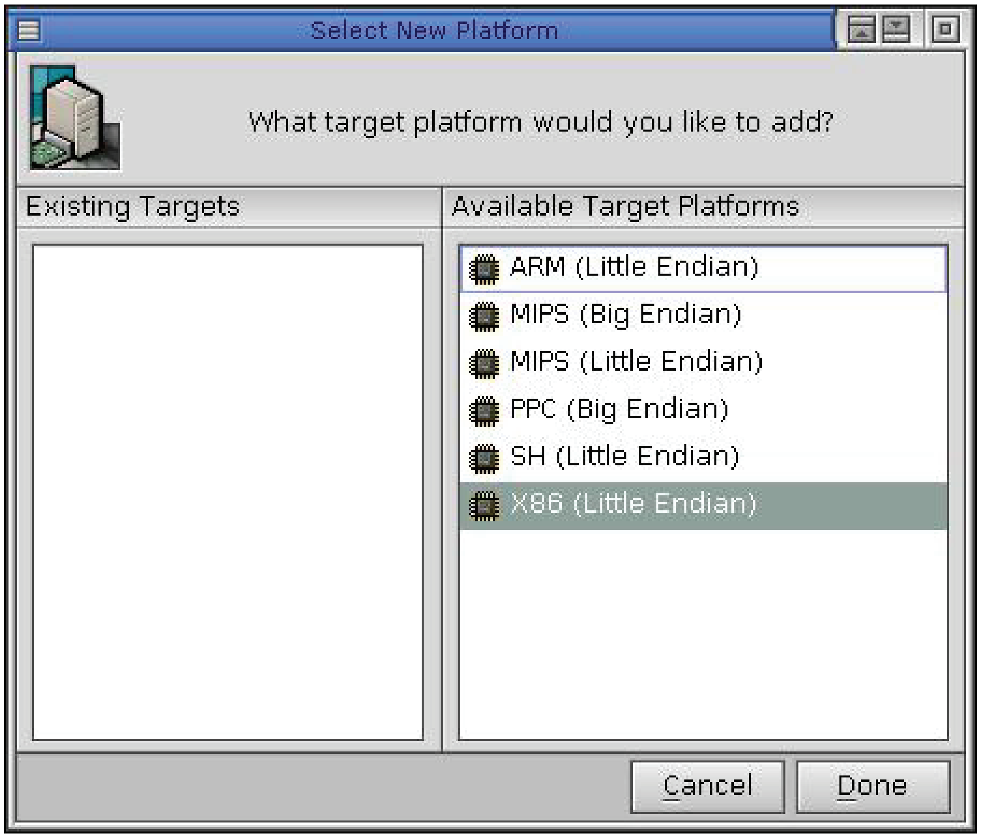
Building your application

To build your application in standalone PhAB:

- Select **Build→Build**.

  When you build your application, PhAB creates an executable that you can run on the target system.

  If you haven't already added targets to your application, the Select New Platform dialog is displayed:



This dialog lets you select a target platform. For example, if you plan to run your application on an SH processor, you should select SH (Little Endian).

You can add more targets later using the Select New Platform dialog. To open this dialog, select **Build→Targets**, then click **Add target**. See Managing targets below.

Select your target and click Done.

- Generate the application by selecting **Build→Generate UI**. PhAB saves your application and then generates the required files. See Generating application code.
- PhAB opens the Make Application dialog and compiles and links your application, generating an executable that can be run on your target system. PhAB runs **`make`** in all of the selected target directories. The **`make`** utility links and compiles the source files; PhAB then runs binders to build the widget files into your executable.

  If any errors or warnings are detected during the make process and it can determine which source file contains errors, PhAB enables the **Edit** button.

  To edit the first file that contains errors, click **Edit**. After fixing the problems, click **Restart** to run **`make`** again.

  To stop **`make`** at any time, click **Cancel**.

Depending on the state of the application, PhAB may skip some of the above steps. For instance, if you've already built your application, but just changed a resource for one of the widgets, PhAB may only save your application to update the changed widget and then rebind the widget files into your application; it will skip code generation, compilation, and linking.

Generating application code

When you make changes to your application, even within your own source files, you may need to generate the application code. Doing so ensures that the prototype header file, **`proto.h`**, is up to date. You can safely generate the code at any time — PhAB won't overwrite any code you've added to the stubs it generated earlier.

When you use PhAB from the IDE, PhAB regenerates the application code every time you save the project.

Before generating the code, PhAB saves your application if you've modified any modules. To minimize compile time, PhAB compiles only the files that have changed.

If you plan to use a global header in your application, you should set up the header before generating any code. For more information, see "Setting project properties" in the Working with Applications chapter, and "Global header file" in the Working with Code chapter.

To generate your application code, select Generate UI from the Build menu. PhAB generates any required files any time you build your application.

What PhAB generates

PhAB generates various files and stores them in the application's **`src`** directory.

CAUTION: Any filename that starts with **`ab`** is a PhAB file and shouldn't be modified at any time. If you try to edit an **`ab`** file, you could lose work (when PhAB overwrites the file) or experience incorrect behavior (when files get out of sync).

You can modify any other files that PhAB generates, with few conditions. These conditions are described in the following sections.

Here are the files that PhAB generates:

`appname/Makefile`	Used to compile and link the application
`appname/common.mk`	Contains traditional makefile options, such as compiler options and link options.
`src/Usemsg`	A usage message for the application
`src/abHfiles`	Contains a list of header files that PhAB knows about.
`src/abLfiles`	External PhAB references in the **`Makefile`**
`src/abSfiles`	Contains a list of source files that PhAB knows about, for example source files with callbacks used in the PhAB project.
`src/abWfiles`	Contains a list of PhAB modules that are part of your application.
`src/abdefine.h`	Contains all the PhAB-generated manifests. PhAB includes this header in all C files.
`src/abevents.h`	Contains all the application's callbacks
`src/abimport.h`	The **`extern`** reference header that's included in all C files. See "Function prototypes", below.
`src/ablinks.h`	Contains all the application's module definitions

src/abmain.c	The application's main-line C file. This file starts with **ab**, so don't modify it.
src/abmain.cc	If PhAB detects any C++ functions in your application, it generates **abmain.cc** instead of **abmain.c**. This file also starts with **ab**, so don't modify it.
src/abplatform	Contains a list of the platform directories for the application
src/abvars.h	Contains all the PhAB-generated global variables
src/abwidgets.h	Contains all the PhAB data references
src/indHfiles	Contains additional header files not contained in **abHfiles**.
src/indLfiles	Contains additional options for the linker.
src/indSfiles	Contains additional source files not contained in **abSfiles**.
proto.h	Contains the application's prototypes — see "Function prototypes", below. *Don't rename this file.*

Version control

Here are the files you need to save if you're using version-control software (PhAB can generate some of them, but it's a good idea to save them all):

abapp.dfn	Callbacks and other information — this is a binary file.
abapp.wsp	Widget lock status — this is a binary file.
wgt/*	Widget resources — these might look like text, but they're binary.
src/*.{c,cc,cpp,C,h}	Source files and headers.
src/*files, **src/Usemsg**	Files that list non-PhAB source files. Be sure to save the non-PhAB source, too.
Makefile, **common.mk**	The **make** files.
target_directory/**Makefile**, *target_directory*/***/Makefile**	The target files and their content.
application_name.**ldb**	Your application's language database. Save any translation files as well.

You'll need to keep a matched set of all the files that PhAB generates; save the same version of the **abapp.dfn**, **src/ab***, and **wgt/*.wgt?** files.

Tips on using CVS

It's easier to save a PhAB application in CVS than RCS. Here are some things to keep in mind:

- Flag the `*.wgt?` and `abapp.dfn` files as binary (`-kb`).
- Since binary files can't be merged, try to prevent multiple people from modifying the binary files at the same time. CVS doesn't support locking; the closest you can get is to set a "watch" on `abapp.dfn` (`cvs watch on abapp.dfn`).

  This way, if you just check out an application, your copy of `abapp.dfn` is read-only and PhAB doesn't let you load the application. If you do want to modify the application, you have to run `cvs edit abapp.dfn`, which makes the file writable. Even though this doesn't prevent other people from doing the same, it at least adds you to a list of "editors" on the CVS server that other people can query.

Function prototypes

PhAB generates function prototypes used by the compiler to check that your functions are called correctly. These prototypes are placed in `abimport.h` and optionally in `proto.h`. Here's how these files compare:

`proto.h`	`abimport.h`
Generated by parsing your source code.	Generated by looking at your application's settings.
Prototypes for all functions are generated.	Only prototypes known to PhAB (callbacks, setup functions, pointer-to-function resources) are generated.
You can have problems with preprocessor directives (see "Potential problems with generating `proto.h`"), unusual C constructs, syntax errors, and C++ code.	Prototypes don't depend on the source code.
Doesn't work with C++.	Contains the appropriate `#ifdef`s and `extern "C"` declarations required by C++.
Prototypes match what the functions look like.	Prototypes match what the functions are *supposed* to look like—if the source code is different, the compiler can detect it.

To suppress the generation of prototypes in `proto.h`:

1. Press F2 or from the Project menu, choose Properties to open the Project Properties dialog.

2 Click the Generate Options tab.

3 Disable the Scan Source Files for Prototypes option.

Potential problems with generating `proto.h`

In the interests of speed, the program that scans your source files for function prototypes ignores preprocessor directives. This can cause some problems in **`proto.h`**.

For example, say we have the following code:

```
#ifdef DOUBLE
    for (i = 0; i < 18; i++, i++) {
#else
    for (i = 0; i < 18; i++) {
#endif
        x += 2 * (i + x);
        y += x;
    }
```

Since preprocessor directives are ignored, the prototype generator sees:

```
    for (i = 0; i < 18; i++, i++) {
    for (i = 0; i < 18; i++) {
        x += 2 * (i + x);
        y += x;
    }
```

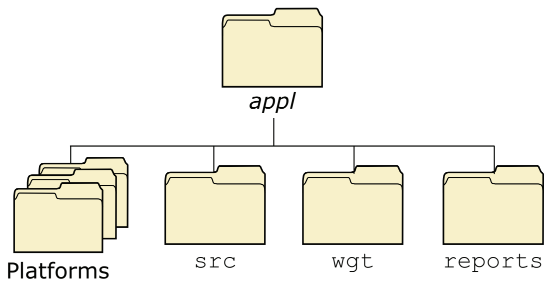
The two opening braces cause it some confusion, and an incorrect prototype is generated. Look for this kind of thing if the prototype generator is creating incorrect prototypes.

To fix the example above, we could remove the opening braces and put an opening brace on the line after the **`#endif`**. Or we could do the following:

```
#ifdef DOUBLE
#define ADDAMOUNT   2
#else
#define ADDAMOUNT   1
#endif

    for (i = 0; i < 18; i += ADDAMOUNT) {
        x += 2 * (i + x);
        y += x;
    }
```

How application files are organized

PhAB stores each application as a directory structure. This structure consists of a main directory that holds the application-definition file, two subdirectories that store the module files and application source code, and, potentially, directories for different development platforms:



Directories for a PhAB application.

At the top of the application's directory structure are the **wgt** and **src** directories. The **wgt** directory contains the application's widget files (files with extension **.wgt*** that PhAB uses to store information about your modules that are part of your application). The **src** directory contains header and source files needed to compile your application. At the same level as **src** and **wgt** are the platform directories (e.g. **x86**, **mips**, and **arm**). Their structure is the same as the structure generated by the **addvariant** command. The **reports** directory contains the files created when you choose the Generate Report command from the Project menu.

This directory structure is called the "Eclipse PhAB project" structure, because it is allows you to create an application in PhAB and then load it in Eclipse as an "Eclipse PhAB project". Conversely, an application created in Eclipse can be loaded in PhAB.

If you first generated your application with an earlier version of Photon, it might have been created as a single-platform application. In this case, the placement of files is slightly different, as described in the sections that follow.

You can choose the target platforms on which to compile your application. You can edit your target configuration by choosing Targets from the Build menu.

Multiplatform applications

Here's what each directory contains for a multiplatform application:

appl	The name of this directory is the same as your application. It contains the application-definition file, **abapp.dfn**. Because this file is proprietary to PhAB, you shouldn't try to modify it.
appl/**src**	The **src** directory contains the source code, headers, and a **Makefile** generated by PhAB, as well as any code you create yourself. For detailed information on the files stored in this directory, see "What PhAB generates" in this chapter.

appl/**src**/*platforms*
: These directories contain the **Makefile**, object files, and executables for the chosen platforms. The **Makefile** in the **src** directory runs those in the platform directories.

appl/**wgt**
: The **wgt** directory contains your application modules. Because each type of module has a different file extension, it's fairly easy to recognize the exact modules you want when importing modules from another application. For more info, see the handy tables in the "Module types" section in the Working with Modules chapter.

CAUTION: Always use PhAB to edit the module files in the **wgt** directory. Don't try to edit these binary files with another editor.

Never modify any files prefixed by **ab**.

Single-platform applications

Here's what each directory contains for a single-platform application:

appl
: The name of this directory is the same as your application. It contains the application-definition file, **abdefn.app**. Because this file is proprietary to PhAB, you shouldn't try to modify it. After you've compiled and linked the application, the executable is also placed in this directory.

appl/**src**
: The **src** directory contains the source code, headers, object files, and **Makefile** generated by PhAB, as well as any code you create yourself.

For detailed information on the files stored in this directory, see "What PhAB generates" in this chapter.

appl/**wgt**
: The **wgt** directory contains your application modules. Because each type of module has a different file extension, it's fairly easy to recognize the exact modules you want when importing modules from another application. For more info, see the handy tables in the "Module types" section in the Working with Modules chapter.

CAUTION: Always use PhAB to edit the module files in the **wgt** directory. Don't try to edit these binary files with another editor.

Never modify any files prefixed by **ab**.

Converting to Eclipse

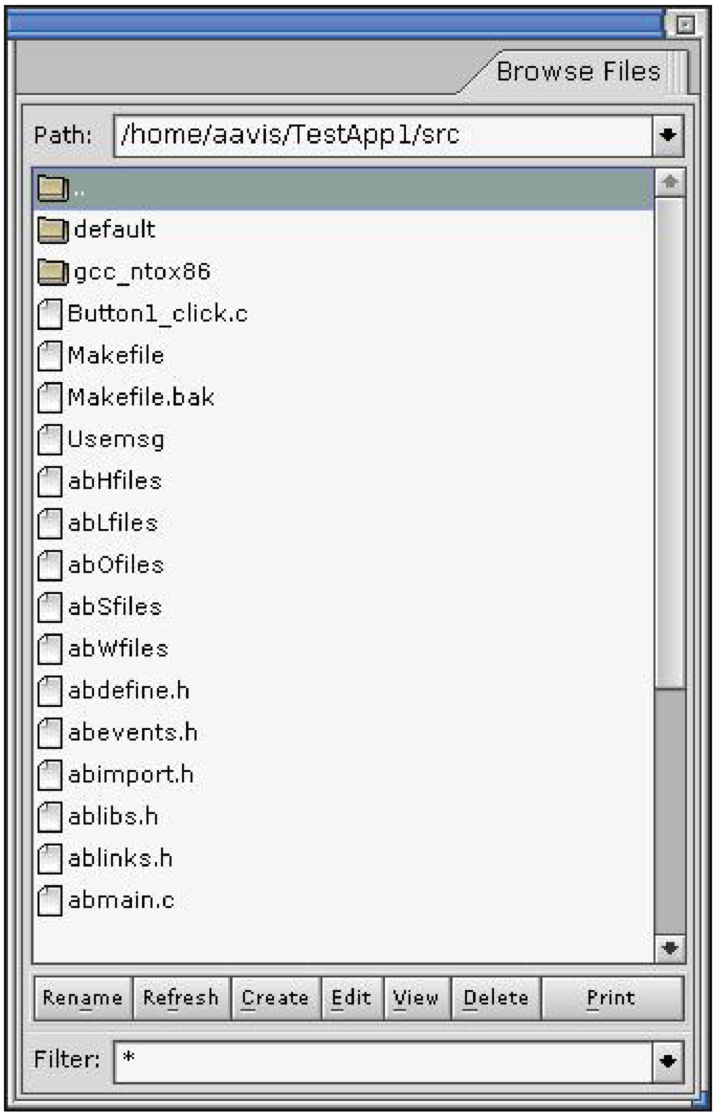
If you have a single-platform application built with an earlier version of Photon, it is converted to “Eclipse Project” format when you load it for the first time. When converting an application, PhAB moves any existing Makefile to **`src/default/Makefile.old`**.

If you have a multiplatform application built with an earlier version of Photon, it is loaded by PhAB as is. You can choose to convert it to Eclipse Project format by selecting **Project→Convert to Eclipse Project**.

Editing source

You can edit source files from an external editor, which you can configure in PhAB. If you’re using PhAB from the IDE, you can use an editor in the IDE to work with source files as well.

You can open source files for editing within PhAB from the Browse Files palette. If the Browse Files palette isn’t visible, select **Window→Show Project**. The Browse Files palette looks like this:



Browse Files palette.

To edit, view, or delete source code from within PhAB:

1 Click the source-code file.

2 Click the appropriate action button (**Edit**, **View**, **Delete**, ...).

You can also edit a file by double-clicking its name.

Choosing an editor or browser

To choose which external editor or browser the Edit and View buttons invoke, see “Customizing your PhAB environment” in the PhAB’s Environment chapter.

Creating a source module

To create a new source-code module:

1. Click **Create** to open the Create File dialog, then type the name of the new file.
2. If you wish to create the file using a template (for a header file or widget callback), select a format from the Template combobox.
3. Click **Create**. You'll see a terminal window that displays either an empty file or a file that contains the template.

If you create any files, click **Refresh** to reread the application directory and refresh the list of files in the Browse Files palette.

Changing the file display

To control which files are displayed in the Browse Files palette, use the following:

- Refresh—forces a reread of the application source directory to ensure that your file list is current
- Filter—lets you specify a filename pattern

The Filter line at the bottom of the palette allows you to filter the list of the files the Browse Files palette displays. For instance **`*.[ch]`** displays subdirectories and filenames ending in **`.c`** and **`.h`**.

Compiling and linking

This section applies to standalone PhAB. For information about compiling and linking applications using the IDE, see the "Building projects" section in the Developing Programs chapter of the *IDE User's Guide*.

After generating the application code, you need to:

- specify additional libraries your application needs
- use **`make`** to compile and link your application.

Specifying additional libraries

By default all PhAB applications link against **`libAp`** and **`libph`**. You can specify addtional libraries libraries for your application in the **Link Libraries** field of the **Build and Debug Options** dialog.

You can also specify how libraries are linked against your application:

- Static Lib—link the Photon and PhAB libraries into the application executable. The application is larger than if you used the shared library, but runs without the shared libraries. This might be useful in an embedded environment.
- Shared Lib—don't include the libraries in the application. The application is much smaller, but requires the Photon shared libraries in order to run.

The default is shared libraries. To specify how libraries are linked, use the `-B` option in the **Link Libraries** field of the **Build and Debug Options** dialog. For example, consider the following **Link Libraries** line: `-Bstatic -lphexlib -Bdynamic -lmylib -Bstatic`. This links against `libphexlib.a`, `mylib.so`, and since the default libraries are considered being after the end, `libAp.a` and `libph.a`.

You can also specify a list of library callback functions when you start PhAB. For more information, see `appbuilder` in the *Utilities Reference*.

Running `make`

Once you've chosen the library type, you're ready to compile and link.

The first time you generate your application, PhAB creates `Makefile` and `common.mk` files in the project directory (plus a `Makefile` for each platform selected for multiplatform development) so you can make the application. Subsequent generations don't modify the file directly; instead, they update external files referenced in the `common.mk`.

Once the `common.mk` is generated you can modify it, though you should only do so when there's no way to set options from within PhAB.

By default, the `Makefile` is compatible with the installed `make` command. You can convert the file to a format that suits the `make` command you prefer—just ensure the external file reference method is still compatible.

For more information, see "Including non-PhAB files in your application," later in this chapter.

To make your application:

1. Select **Build**→**Build** to open the Make Application dialog and start the make process.
2. If any errors or warnings are detected during the make process, PhAB enables the **Edit** and **Restart** buttons.

   To edit the first file that contains errors, click **Edit**. After fixing the problems, click **Restart** to run `make` again.

   To stop `make` at any time, click **Cancel**.
3. Once the application has been compiled and linked, PhAB enables the Make dialog's **Done** button. Click **Done** to close the dialog.

   The **Done** button is also enabled when you click **Cancel**.

Modifying the make command

By default, PhAB uses the installed `make` command to make your application. If you need to change this command in any way, click **Build Preferences**.

Any changes you make to Build Preferences settings are saved with the application itself rather than as a global preference.

Customizing the build process

You can customize the build options on the Build and Debug Options tab on the Project Properties dialog. Open the Project Properties dialog by selecting **Project→Properties**. For a description of this dialog, see Build and Debug options in the Working with Applications chapter.

When you use PhAB from the IDE, PhAB handles the code generation for the user interface, while the IDE manages the build process. Therefore, only items in this dialog that affect generation are available when you run PhAB from the IDE.

The first time you generate your application, PhAB creates a **`Makefile`** and a **`common.mk`** in the top level directory. Subsequent generations don't modify the file directly. You can customize your build process by changing or adding compile and link flags in **`common.mk`**.

Once the **`Makefile`** is generated you can modify it, though you should only do so when there's no way to set options from within PhAB.

By default, the **`Makefile`** is compatible with the installed **`make`** command. You can convert the file to a format that suits the **`make`** command you prefer—just ensure the external file reference method is still compatible.

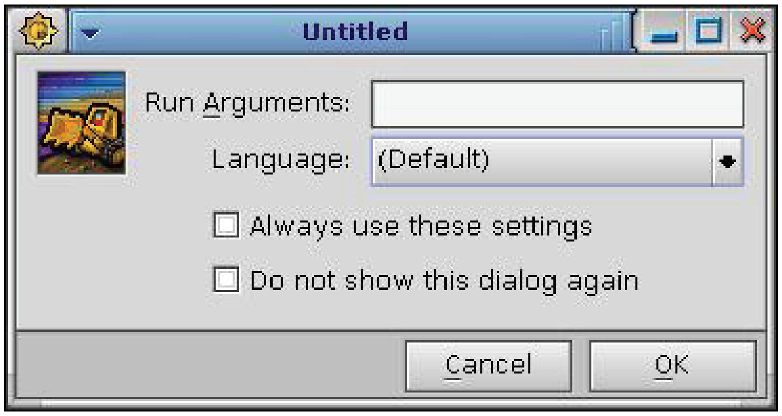
For more information, see "Including non-PhAB files in your application," later in this chapter.

Running the application

How you run an application developed in PhAB depends on whether you're running PhAB standalone or from the IDE. When running PhAB from the IDE, you set up launch configurations for targets through the IDE. For more information, see the Preparing Your Target chapter and "Running projects" section of the Developing C/C++ Programs chapter in the IDE *User's Guide*.

From standalone PhAB, once your application has been compiled and linked without errors, it's ready to run. Just follow these steps:

1 Select Run from the Build menu. The following dialog appears:



2 If you've used PhAB to create a multilingual application, you can choose the language from the **Language** list. For more information, see the International Language Support chapter.

3 If your application needs command-line arguments, type them into **Run Arguments**.

4 Click **OK**.

When your application runs, its working directory is the platform directory (i.e. the directory containing the executable).

If you check **Do not show this dialog again**, the Run Options dialog doesn't appear the next time you run the application. To reenable the dialog, check the **Ask for run arguments** option on the General tab of the AppBuilder Preferences Settings dialog.

If you check **Always use these settings**, then the settings are saved and used any time you run this application. You can change the run arguments or language on the Run Options tab of the Project Properties dialog.

If you use functions such as *printf()* in your application, the output goes to your console if you run your application from PhAB. To see this output, open a `pterm` and run the application from there instead of from PhAB.

PhAB is still active while your application is running. To switch between the two, use the Window Manager's taskbar.

If the target you chose to build your application for is not the same as the one you run PhAB on, you'll have to transfer your application's executable to your target in order to run it.

Debugging

You can run your application with a debugger, which can be handy if your application crashes or behaves incorrectly.

When running PhAB from the IDE, you use the debug features in the IDE. For more information, see the Debugging Programs chapter in the IDE *User's Guide*.

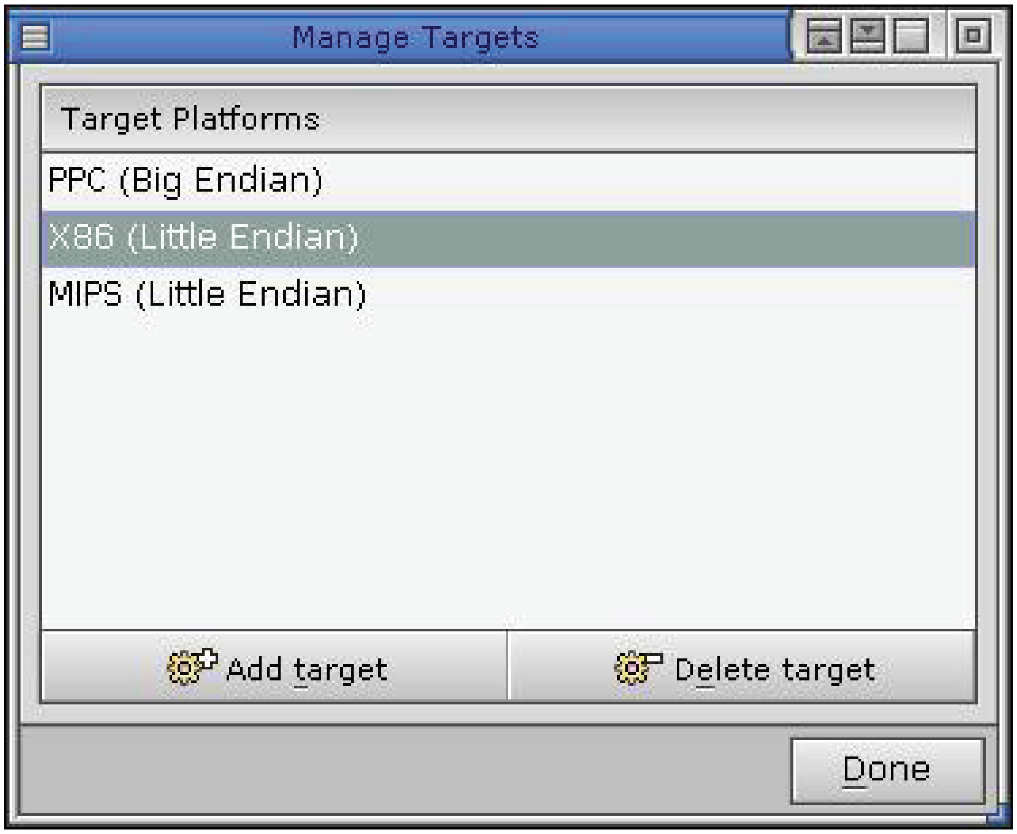
- To debug your application in standalone PhAB, choose **Build & Debug** from the **Build** menu. This launches the application in your preferred debugger.

In order to debug an application, you have to compile a debug-build version. To do this, select **Build & Debug** from the **Build** menu.

To switch between the debugger and the application, use the Window Manager's taskbar.

Managing targets

You can add or delete target platforms using the **Manage Targets** dialog. To open this dialog, select **Build→Targets**.



Manage Targets dialog.

To add a target, click Add target.

- If you select a target that is already added to your application, the **`Makefile`** in the target directory is overwritten.
- Targets on which Photon doesn't run are greyed out and can't be selected. The targets which don't support Photon are defined in a configuration file, `$QNX_HOST`**`/usr/photon/appbuilder/nonphoton_targets.def`**.

To delete a target, select it and click **Delete target**.

Only the targets selected in this dialog are built.

The Build menu

Here's a description of all the items in the Build menu:

Build & Run	Standalone PhAB only. Builds the current application and then runs it. This command saves any changed files before building the application.
Build & Debug	Standalone PhAB only. Builds your application and then debugs it. This command saves any changed files before building the application.
Rebuild All	Standalone PhAB only. Rebuilds all files for all selected targets.
Build	Standalone PhAB only. Builds the application for selected targets.
Make Clean	Standalone PhAB only. Removes the executable and object files from the target directories before running **`Make`**.
Generate UI	Generates files related to the user interface, but doesn't build the application's executable.
Run	Standalone PhAB only. Runs the last built executable.
Targets	Standalone PhAB only. Opens the Manage Targets dialog.

* For these commands, "all your selected targets" means the building process will take place only for selected targets. To change the currently selected targets simply choose **Targets** from the Build menu and change the selection in the target list. You can also select targets from the Build and Debug Options tab of the Project Properties dialog.

Including non-PhAB files in your application

Your application can include files that aren't created with PhAB, but you need to tell PhAB how to find them.

Eclipse Project applications

To add non-PhAB source files to an application, place them in the project's **src** directory.

Multiplatform applications

PhAB generates empty lists in the following files in the **src** directory, and you can edit them:

indHfiles Non-PhAB header files. For example:

```
MYHDR = ../header1.h ../header2.h
```

indOfiles Non-PhAB object files. For example:

```
MYOBJ = file1.o file2.o
```

indSfiles Non-PhAB source files. For example:

```
MYSRC = file1.c file2.c
```

Remember to specify the filenames relative to where the **Makefile** is found. For a multiplatform application, that's relative to the platform directory:

- Header files and source files are usually in the parent directory, **src**, so their names start with **../** .
- Object files are usually in the same directories as the **Makefile**s.

Single-platform applications

A single-platform application from an earlier version of Photon doesn't have the **indHfiles**, **indOfiles**, and **indSfiles** files. Instead, you'll find *MYHDR*, *MYOBJ*, and *MYSRC* in your **Makefile**, and you can specify filenames there.

Remember to specify the filenames *relative to where the Makefile is found*. For a single-platform application, that's relative to the **src** directory.

Adding libraries

If your application uses a library that isn't included by default in the `Makefile`, you can add it with the **Link Libraries** line on the Build and Debug Options tab of the Project Properties dialog. Open this dialog by selecting **Project→Properties**.

For instance, if you want to link against the socket library, put the following in the **Link Libraries** line:

`-l socket`.

You can also add it by editing the *LDFLAGS* variable in `common.mk`:

Making a DLL out of a PhAB application

You can make a PhAB application into a DLL, but there isn't a PhAB option that will do it for you. PhAB doesn't know anything about building DLLs; it's the PhAB *library* that lets you turn a PhAB application into a DLL.

The application that loads your DLL must be a PhAB application so that the PhAB library is properly initialized.

Even though PhAB lets you set up an initialization function and windows to open at startup, they're ignored when your application is a DLL. That's because this regular PhAB startup is done by *main()*, and the *main()* function of a DLL isn't called. (Don't attempt to call it from your own code, either.)

Compiling and linking

In general, you can turn any application (whether created by PhAB or not) into a DLL by adding `-shared` to the compiler and linker flags (and most likely adding a `dll` extension to the filename). You should also give the `-Bsymbolic` option to the linker to make sure that locally defined symbols that your DLL uses aren't overridden by any symbols in the executable with the same name.

To make these changes for a PhAB application, run the `addvariant` command at the top level directory for the application. For example:

```
addvariant x86 dll
```

For more information, see `addvariant` in the *Utilities Reference*, and the Conventions for Recursive Makefiles and Directories appendix of the QNX Neutrino *Programmer's Guide*.

Initializing your DLL

Typically, a DLL defines an initialization function that an application calls after it calls *dlopen()* to load the DLL. Your DLL's initialization function needs the full path to the DLL.

Before calling any PhAB code, the initialization function must call *ApAddContext()*, like this:

```
ApAddContext( &AbContext, fullpath );
```

The arguments are:

AbContext	A global data structure that PhAB puts in **abmain.c**.

This structure has the same name in every PhAB application or DLL, so you must link your DLL with the **-Bsymbolic** option mentioned above.

fullpath	The full path of your DLL, suitable for passing to *open()*.

You can call *ApAddContext()* more than once, but you need to keep track of how many times you called it.

ApAddContext() returns zero on success, or -1 on failure. Don't call any *Ap** functions if *ApAddContext()* fails.

AbContext holds, among other things, the location of the executable or DLL that it's contained in, and the language translation files currently loaded for that file. At any time, one of the registered contexts may be "current"; a few **libAp** functions implicitly refer to the current context, and some require a current context when you call them. At program startup, the program's context is made current. You can unset it or change it to a different context by calling *ApSetContext()*:

```
ApContext_t *ApSetContext( ApContext_t *context );
```

This makes the given *context* current, and returns the previous current context. Both can be NULL; but if you pass a non-NULL pointer, it must point to a registered context.

So, for example, your initialization code would look something like this in your main application (with error checking removed for clarity):

```
void *handle = dlopen( path, 0 )
int (*entrypnt)() = ((int(*)()) dlsym( handle, "myentrypoint" );
(*entrypnt)( path, ABW_dll_pane );
```

And the initialization function in the DLL would look something like this:

```
int myentrypoint( const char *path, PtWidget_t *panewgt ) {
    ApAddContext( &AbContext, path );
    ApCreateModule( ABM_dll_contents, panewgt, NULL );
}
```

Unloading your DLL

When an application is about to unload a DLL, it typically calls a cleanup function in the DLL. In your DLL's cleanup function, you must:

- close any widget databases that your DLL opened:

    - that was opened using *ApOpenDBase()* with an argument pointing to a module defined in the DLL

      or

    - that was opened using *ApOpenDBaseFile()* if the DLL's *AbContext* was the current context when it was called

- destroy any PhAB widgets that "belong" to your DLL; this means you have to destroy any widgets:
    - created by PhAB link callbacks defined in the DLL
    - *ApCreateModule()* using a module in the DLL
    - that are using any of the widget databases that must be closed
- if your DLL defined widget classes by calling *ApAddClass()*, remove them by calling *ApRemoveClass()*
- call *ApRemoveContext()*, like this:

  ```
  ApRemoveContext( &AbContext );
  ```

You must call *ApRemoveContext()* the number of times that you successfully called *ApAddContext()*. After you've called *ApRemoveContext()*, your DLL must not call any PhAB functions.

Chapter 10

Working with Code

In this chapter. . .

PhAB makes it easy to create the user interface for an application. Once PhAB has generated code stubs for your application, you'll need to write the parts that make the application *do* something. This chapter describes how to work with the code for a PhAB application.

For information about using threads in a Photon program, see the Parallel Operations chapter.

Variables and manifests

Widget variables and manifests

PhAB creates global variables and manifests for every module you create, and every widget with a unique instance name. This makes it easier to access the widgets from your application code.

The global variable represents the widget's name, and is defined in the **abvars.h** file. Each global variable takes this form:

- ABN_*widget_name*—where *widget_name* is the widget name or module-instance name that you defined in the Resources or Callbacks control panel. The value of this variable is unique in the entire application.

The manifest represents the widget's instance pointer, and is defined in the **abdefine.h** file. This file, which is included in all application C files, also defines an external reference to the global variables. Each manifest takes this form:

- ABW_*widget_name*—where *widget_name* is the widget name or module-instance name that you defined in the Resources or Callbacks control panel.

PhAB doesn't create **ABW_**... manifests for menu modules or menu items. Menus typically don't exist for very long, so manifests for them aren't very useful. If you need to change the resources of the **PtMenu**, create a setup function for the menu module and do the work there. See "Module setup functions," below.

The manifest for a window module refers to the last created instance of the module. See "Handling multiple instances of a window," below.

When PhAB detects a unique instance name it generates a global variable name and a widget manifest. For example, if you change the instance name for a **PtButton**-class widget to **done**, PhAB will generate the following:

- ABN_done
- ABW_done

A widget's global variable and manifest are valid *only* after the widget has been created, and before it has been destroyed.

Using the global variable and widget manifest

Let's now look at some examples of how you can use the global name and widget manifest within application code. First, here's an example of using the ABN_done constant and the *ApName()* function to check for a specific widget in a callback:

```
int
mycallback( PtWidget_t *widget, ... )

    {

    /* check for specific widget */
    if ( ApName( widget ) == ABN_done ) {
        /* done button processing */
        }

    return( Pt_CONTINUE );
    }
```

The next example uses ABW_done to change the **done** widget's background color to red (for more information, see the Manipulating Resources in Application Code chapter):

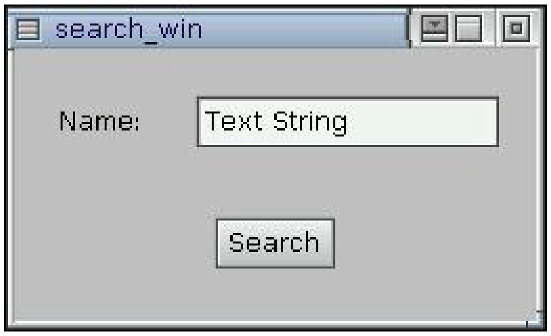
```
int
mycallback( PtWidget_t *widget, ... )

    {
    PtSetResource( ABW_done, Pt_ARG_FILL_COLOR, Pg_RED, 0 );

    return( Pt_CONTINUE );
    }
```

Remember that the global variable and the manifest are valid *only* after the widget has been created and before it has been destroyed.

Handling multiple instances of a window

If you have more than one instance of a window module displayed at a time, then you'll have a problem accessing the widgets in the window. The problem stems from the fact that **ABW_***instance_name* for any widget in a window module points to the last created instance of that widget. If you have more than one instance of the window, then you have more than one instance of the widgets within the window created.

Let's say you have the following window module:



A sample search window.

Let's assume that the instance names are:

- **`search_win`** for the window
- **`name_txt`** for the text field
- **`search_btn`** for the button.

If you have two instances of the window on the screen at the same time and the user clicks on the Search button, how can you get the value in the Name text widget? Since two instances of the window exist, two instances of the text widget exist. ABW_name_txt points to the last instance of the text widget that was created.

The solution lies in the fact that ABN_name_txt can be used to refer to both instances of **`name_txt`**, provided you have the widget pointer to the window that contains the desired text widget. This is done using the *ApGetWidgetPtr()* function:

```
PtWidget_t  *window_wgt, *text_wgt;

text_wgt = ApGetWidgetPtr(window_wgt, ABN_name_txt);
```

Where do you get *window_wgt*? In the above case, you'd have a code callback on the Search button. The first parameter passed to that code callback is the widget pointer to the Search button. You can use *ApGetInstance()* to get a pointer to the window that contains the Search button.

So the callback would become:

```
int search_callback( PtWidget_t *widget, ApInfo_t *apinfo,
                     PtCallbackInfo_t *cbinfo)
{
    char        *name;
    PtWidget_t  *window_wgt, *text_wgt;

    /* Get the window that the Search button is in. */

    window_wgt = ApGetInstance( widget );

    /* Given the window, find the text widget. */

    text_wgt = ApGetWidgetPtr( window_wgt, ABN_name_txt );
    /* Now get the text. */

    PtGetResource( text_wgt, Pt_ARG_TEXT_STRING, &name, 0 );

    /* The 'name' variable now points to the correct name text.
       Process the text as appropriate. */
    ⋮

    return( Pt_CONTINUE );
}
```

Internal link manifests

PhAB generates a manifest for each internal link defined in your application:

- ABM_*internal_link_name*—where *internal_link_name* is a pointer to the module's internal definition.

For more information about using internal links, see the Accessing PhAB Modules from Code chapter.

Global header file

PhAB lets you define one global header file for each application. PhAB generates this file only once, the first time you generate the application's code.

Once you've defined the header, PhAB automatically includes it in any generated C or C++ stub file. So it's best to define the header when you first create the application. See "Startup Windows tab" in the Working with Applications chapter. You can modify the header whenever you need to.

Here's a handy way of using this single header file to simultaneously define all your global variables and the **extern** references to those variables:

```
/* Header "globals.h" for my_appl Application */

#include <Pt.h>

#ifdef DEFINE_GLOBALS

#define GLOBAL
#define INIT(x) = x

#else

#define GLOBAL extern
#define INIT(x)

#endif

/* global variables */
GLOBAL int           variable1        INIT(1);
```

If **DEFINE_GLOBALS** *is* defined, then the last line in the above example looks like:

```
int variable1 = 1;
```

If **DEFINE_GLOBALS** *isn't* defined, then the last line in the above example looks like:

```
extern int variable1;
```

Remember to define all your application's global variables with the **GLOBAL** prefix, as shown above. Also make sure to include the following line in one (and only one) of your code files:

```
#define DEFINE_GLOBALS
```

Including this line ensures that the global variables are defined in this code file and used as **extern** declarations in all other code files.

In the **Makefile**, make the code files dependent on the header file. That way, if you make any changes to the header, all the code will be recompiled when you make your application.

Function names and filenames

PhAB generates a function for every initialization function, module setup function, callback, function menu item, and so on you've specified in your application. If you don't need a function, leave its name blank.

After a function has been generated, you're free to modify it. There's just one condition: if you change the function's name, you must also change the name you specified in the link callback or internal link definition. Otherwise, PhAB will continue to regenerate the old name every time it generates the application.

The way you specify the function name in PhAB determines the name of the file the stub is put into:

function_name Create a C stub file called *function_name*.**c**

function_name@filename.ext

Create a stub function and put it in *filename.ext*. This file will include the headers and function structure required to compile in the Photon environment.

PhAB recognizes **.cc**, **.cpp**, and **.C** as C++ extensions.

If this file already exists, the stub function is added to it. You can use this technique to reduce the number of code files for your application. You can place any number of functions in the same file. We recommend you put all functions related to a module in the same file.

function_name.ext Short form for *function_name@function_name.ext*

class`::`*function_name*`@`*filename*`.cc`
: Generate a stub C++ static member function, but no prototype.

class`::`*function_name*`@`
: Don't create a stub function or a prototype. Instead, invoke a C++ static class member function. Prototypes aren't generated for class member functions; your application must have the necessary declarations in its global header file.

function_name`@`
: Generate a prototype for a C function, but not a stub. This is useful if you're using a library of C functions.

`::`*function_name*`@`
: Generate a prototype for a C++ function, but not a stub. This is useful if you're using a library of C++ functions.

You can use C and C++ in the same PhAB application. See "What PhAB generates" in the Generating, Compiling, and Running Code chapter.

Initialization function

PhAB lets you define an application-level *initialization function*. The PhAB API calls this function once at startup, before any windows or other widgets are created. For information on setting up this function, see "Startup Windows tab" in the Working with Applications chapter.

The initialization function includes the standard *argc* and *argv* arguments so that your application can parse command-line options if needed (see below). You can also use this function to open any widget databases (see the Accessing PhAB Modules from Code chapter) or other application-specific data files.

Here's a sample initialization function generated by PhAB:

```
/* Y o u r   D e s c r i p t i o n                       */
/*                            AppBuilder Photon Code Lib */
/*                                         Version 2.01A */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "abimport.h"
#include "proto.h"

/* Application Options string */
const char ApOptions[] =
    AB_OPTIONS ""; /* Add your options in the "" */
```

```
int
init( int argc, char *argv[] )

    {

    /* eliminate 'unreferenced' warnings */
    argc = argc, argv = argv;

    /* Handle command-line options - if required.
       Remember to ignore the ones handled by Photon. */

    /* Typical spot to open widget databases */

    /* Any other application-specific initialization */

    return( Pt_CONTINUE );

    }
```

Processing command-line options

PhAB applications have several command-line options by default:

-h *height*[**%**]	The height of the window, in pixels, or as a percentage of the screen height if **%** is specified.
-s *server_name*	The name of the Photon server:

If server_name is:	This server is used:
node_number	**//***node_number***/dev/photon**
fullpath	*fullpath*
relative_path	**/dev/***relative_path*

-w *width*[**%**]	The width of the window, in pixels, or as a percentage of the screen width if **%** is specified.
-x *position*[**%**][**r**]	The x coordinate of the upper-left corner of the window, in pixels, or as a percentage of screen width if **%** is specified. If **r** is specified, the coordinate is relative to the current console.
-y *position*[**%**][**r**]	The y coordinate of the upper-left corner of the window, in pixels, or as a percentage of screen height if **%** is specified. If **r** is specified, the coordinate is relative to the current console.
-Si|**m**|**n**	The initial state of the main window (iconified, maximized, or normal).

You can suppress the options for the application's size and position—see "Command-line options" in the Working with Applications chapter. You can also define additional options.

Edit the application's usage message, which you'll find in the **`Usemsg`** file in your application's **`src`** directory, to include any additional options. For details about the usage message syntax, see **`usemsg`** in the QNX Neutrino *Utilities Reference*.

Use the *getopt()* function (described in the *C Library Reference*) to process the command-line options. The following example shows how you could process several options (three of which take arguments):

```
const char ApOptions[] = AB_OPTIONS "a:b:c:pqr";

int init( int argc, char *argv[] ) {
    int opt;
    while ( ( opt = getopt( argc, argv, ApOptions ) ) != -1 )
        switch ( opt ) {
            case 'a' : ...
            case 'b' : ...
            ...
            case '?' : ...
            }
    ...
    return Pt_CONTINUE;
    }
```

AB_OPTIONS is a macro that defines the default options added by PhAB. It's generated by PhAB, based on your the settings from the Generate Options tab of the Project Properties dialog. For example, if you disable the **Enable Window Position Arguments** option, the AB_OPTIONS macro won't contain **`x:`** or **`y:`**. You can process the options in AB_OPTIONS in two ways:

- include a **`case`** branch for each option, but do nothing in it. You could also include a **`default`** that prints a message if an invalid option is given.

  or

- don't include **`case`** branches for them. If you do this, you won't be able to have a **`default`** branch.

The PhAB library also looks at the *ApOptions* array to take into account the options you've added. For example, for the above code the library recognizes that **`-px100`** specifies the X position (along with **`-p`**), while **`-ax100`** doesn't.

Module setup functions

A module setup function is generated if you specify a function name in a module-type link callback, as described in "Module callback" in the Editing Resources and Callbacks in PhAB chapter.

All PhAB setup functions have three main arguments:

```
int
base_setup( PtWidget_t *link_instance,
            ApInfo_t *apinfo,
            PtCallbackInfo_t *cbinfo )

    {

    /* eliminate unreferenced warnings */
    link_instance = link_instance,
                    apinfo = apinfo,
                    cbinfo = cbinfo;

    /* your code here */

    return( Pt_CONTINUE );
    }
```

where:

link_instance an instance pointer for the PhAB module being created. You'll need to save this pointer if it points to a window module that supports multiple instances.

apinfo A pointer to a **`ApInfo_t`** structure that provides:

- A pointer to the widget that caused the setup function to be invoked (that is, the widget that caused the module to be displayed). For an internal link, this pointer is a copy of the widget pointer passed to *ApCreateModule()*; this pointer is useful for positioning the module.

  You can also determine the widget that caused the setup function to be invoked by calling *ApWidget()*.
- Reason codes related to the type of setup function being invoked:

  ABR_PRE_REALIZE This setup function is being called before the module is realized.

  ABR_POST_REALIZE

  This setup function is being called after the module is displayed on the screen.

cbinfo a pointer to a common Photon callback structure. The structure provides information related to the widget callback being invoked, the Photon event, and some widget-specific callback data. The format of the data varies with the widget class and callback type. For more info, see **`PtCallbackInfo_t`** in the *Widget Reference*.

Normally, a setup function returns the value Pt_CONTINUE. This tells the PhAB API to continue processing and to display the module being linked. For window and dialog modules, you can return Pt_END to terminate the link callback and destroy the module without displaying it. For window modules, you can return Pt_HALT to neither realize nor destroy the window. This is useful if you want to realize the window later.

Code-callback functions

A code-callback function is generated if you specify a code-type link callback, as described in “Code callbacks” in the Editing Resources and Callbacks in PhAB chapter.

All code-type callback functions have three main arguments:

```
int
mycallback( PtWidget_t *widget,
            ApInfo_t *apinfo,
            PtCallbackInfo_t *cbinfo )

    {

    /* eliminate unreferenced warnings */
    widget = widget,
    apinfo = apinfo,
    cbinfo = cbinfo;

    /* your code here */

    return( Pt_CONTINUE );
    }
```

where:

widget A pointer to the widget instance that invoked the callback. This is a pointer to a **PtWidget_t** structure, but you should treat it as a widget identifier; don’t manipulate the members of the structure.

apinfo A pointer to a **ApInfo_t** structure that includes reason codes related to the type of callback function being invoked:

ABR_CANCEL	This callback function is being called by a Cancel link.
ABR_CODE	This callback function is being called by a Code link.
ABR_DONE	This callback function is being called by a Done link.

cbinfo a pointer to a common Photon callback structure. The structure provides information related to the widget callback being invoked, the Photon event, and some widget-specific callback data. The format of the data varies with the widget class and callback type. For more information, see **PtCallbackInfo_t** in the *Widget Reference*.

Your callback should return Pt_CONTINUE unless the description of the callback gives you a reason to return some thing else. ABR_CANCEL and ABR_DONE callbacks can return Pt_HALT to prevent the module from being closed.

Geometry data types

Here are the data structures that you'll use a lot when specifying positions, sizes, and areas:

`PhPoint_t`	The *x* and *y* coordinates of a single point. You'll use it to specify locations on the screen, in widget canvasses, and so on.
`PhDim_t`.	A width (*w*) and a height (*h*), usually in Photon coordinates. You'll use it to specify dimensions.
`PhArea_t`	A rectangular area, expressed as a **`PhPoint_t`** for the area's upper left corner, and a **`PhDim_t`** that defines the area's size.
`PhRect_t`	A rectangle, expressed as two **`PhPoint_t`** structures, one for the upper left corner, and one for the lower right.
`PhTile_t`	A list of rectangles. This structure is used mostly in *damage lists* that define the areas of the screen or a widget that need to be refreshed.

Photon maintains an internal pool of tiles because they're frequently used, and using a pool reduces the amount of time spent allocating and freeing the tiles. Use *PhGetTile()* to get a tile from the pool, and *PhFreeTiles()* to return a list of tiles to the pool.

You probably won't use the **`PhTile_t`** structure unless you're using a **`PtRaw`** widget or creating a custom widget. For more information, see "**`PtRaw`** widget" in the Raw Drawing and Animation chapter, and *Building Custom Widgets*.

The Photon libraries provide various functions that work with these data types:

PhAreaToRect()	Convert an area into a rectangle
PhDeTranslateRect()	Detranslate a rectangle (subtract offset)
PhRectIntersect()	Find the intersection of two rectangles
PhRectToArea()	Convert a rectangle into an area
PhRectUnion()	Determine a bounding box for two rectangles
PhTranslateRect()	Translate a rectangle (add offset)

Timers

If you wish to schedule your own operations at particular time intervals, or if you just want to implement a time-out or trigger an event at a particular time, you may want to have a timer-based callback function. There are several ways to do this, with varying amounts of difficulty and accuracy:

- **`PtTimer`** widget — easy, but not very accurate.
- *RtTimer** functions — a bit more work, a bit more accurate.
- Timers in a separate process from the GUI — necessary for hard realtime. For more information, see "Threads" in the Parallel Operations chapter.

The Photon libraries also include some low-level timer routines, but you need to be careful using them:

PhTimerArm()	Arm a timer event. Don't use this function in an application that uses widgets.
PtTimerArm()	Arm a timer event on a widget. This routine is typically used when building custom widgets. Some widgets (such as **`PtTerminal`**) already use this type of timer, so calling *PtTimerArm()* may have unexpected results.

Using `PtTimer`

The easiest way to implement a timer is to use a **`PtTimer`** widget. It defines these resources:

Pt_ARG_TIMER_INITIAL
Initial expiration time.

Pt_ARG_TIMER_REPEAT
Optional repeat interval.

Pt_CB_TIMER_ACTIVATE
Expiration callbacks.

For more information, see the *Widget Reference*.

When you create a **`PtTimer`** widget in PhAB, it appears as a black box. The box doesn't appear when you run the application; it's just a placeholder.

`PtTimer` is easy to use, but doesn't give accurate timer events. In particular, it doesn't guarantee a constant repeat rate; since the repetition is handled by rearming the timer for each event, any delays in handling the events accumulate. Kernel timers guarantee an accurate repeat rate even if your application can't keep up with them.

*RtTimer** functions

The *RtTimer** functions (described in the Photon *Library Reference*) give more accurate timing than **`PtTimer`**, but still not hard realtime. They're cover functions for the POSIX functions that manipulate the kernel timers:

RtTimerCreate()	Create a realtime timer
RtTimerDelete()	Delete a realtime timer
RtTimerGetTime()	Get the time remaining on a realtime timer
RtTimerSetTime()	Set the expiration time for a realtime timer

These functions are more accurate than **`PtTimer`** because the timer is rearmed by the kernel, not by Photon. However, if Photon is busy handling events, there could still be delays in processing the expiration events.

Initializing menus

You may want to do various things to a menu before it's displayed. You can use the menu's setup function to:

- enable, disable, or toggle items
- change the text for an item

You can also use a function menu item to generate new items at runtime.

The methods for doing these things are discussed in the sections that follow.

Enabling, disabling, or toggling menu items

If a menu item isn't currently a valid choice, it's a good idea to disable it so the user won't try to select it. Of course, you'll need to enable it when appropriate, too. If your menu has any toggle items, you'll also need to set them before the menu is displayed. To do these things, use the *ApModifyItemState()* function.

ApModifyItemState() takes a variable number of arguments:

- The first argument is a pointer to the menu module. For example, if the instance name of the menu module is **`draw_menu`**, pass **`&draw_menu`** as the first parameter.
- The second argument is the desired state:

AB_ITEM_DIM	to disable the item
AB_ITEM_NORMAL	to enable and unset the item
AB_ITEM_SET	to set a toggle item

- The rest of the arguments form a NULL-terminated list of the menu items to be set to the given state. This list consists of the **`ABN_...`** constants of the items.

For example, suppose our application has a menu module whose name is **`draw_menu`**, which includes items with the instance names **`draw_group`** and **`draw_align`**. We can disable these items with the following call:

```
ApModifyItemState (&draw_menu, AB_ITEM_DIM,
                   ABN_draw_group, ABN_draw_align, NULL);
```

Changing menu-item text

You can use the *ApModifyItemText()* function to change the text for a menu item, for example, to replace a command by its opposite. The arguments are as follows:

- a pointer to the menu module. For example, if the instance name of the menu module is **`draw_menu`**, pass **`&draw_menu`** as the first parameter.
- the **`ABN_...`** constant for the menu item
- the new text

For example, our Draw menu might have an item that's either **`Group`** or **`Split`**, depending on what objects the user chooses. We could change the text of the **`draw_group`** item in the **`draw_menu`** with the following code:

```
ApModifyItemText (&draw_menu, ABN_draw_group, "Split");
```

To get the current item text, call *ApGetItemText()*.

If you change the item's text, you probably need to change the shortcut, too, by calling *ApModifyItemAccel()*.

Generating menu items

Sometimes you may need to generate the menu items at runtime. For example, PhAB's Window menu includes a list of the modules in your application. To generate menu items, add a function item to your menu module (as described in "Creating function items" of the Working with Modules chapter), and edit the stub function PhAB generates.

For example, if our **`draw_menu`** module includes a function item that calls *add_shapes()*, PhAB generates the following code:

```
int add_shapes (PtWidget_t *widget, ApInfo_t *apinfo,
                PtCallbackInfo_t *cbinfo)
{

    /* eliminate 'unreferenced' warnings */
    widget=widget, apinfo=apinfo, cbinfo=cbinfo;

    return (Pt_CONTINUE);
}
```

The parameters passed to this function are of no use.

We use the *PtCreateWidget()* function to create the menu items, which are usually **`PtMenuButton`** widgets. As discussed in the Manipulating Resources in Application Code chapter, we can use the same sort of argument list to set initial values for the resources as we use with *PtSetResources()*. For example, to add an item **`Rectangle`** with a keyboard shortcut of **`R`**:

```
PtArg_t    args[2];
PtWidget_t *new_item;

PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Rectangle", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "R", 0);
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           2, args);
```

The second parameter in the call to *PtCreateWidget()* is the parent of the widget; when you're generating menu items, this should be set to Pt_DEFAULT_PARENT. This makes the new item a child of the current menu or submenu. Don't call *PtSetParentWidget()* in this case.

Next, we need a callback function for the new item. We have to create this manually; PhAB doesn't create a stub function for it. For example, the callback for our new item could be:

```
int rect_callback( PtWidget_t *widget,
                   void *client_data,
                   PtCallbackInfo_t *cbinfo )
{
    ...
}
```

This callback is similar to a code callback generated by PhAB. Its arguments are:

widget	A pointer to the menu item selected.
client_data	Arbitrary data passed to the callback.

This is different from a PhAB code callback, which receives *apinfo* as its second argument.

cbinfo	a pointer to a common Photon callback structure. The structure provides information related to the widget callback being invoked, the Photon event, and some widget-specific callback data. The format of the data varies with the widget class and callback type. For more info, see **PtCallbackInfo_t** in the *Widget Reference*.

The last thing we need to do is add the callback to the menu item's *Pt_CB_ACTIVATE* callback list, using the *PtAddCallback()* function:

```
PtAddCallback (new_item, Pt_CB_ACTIVATE,
               rect_callback, NULL);
```

The last argument to *PtAddCallback()* specifies what's to be passed as the *client_data* argument of the callback. For more information, see "Callbacks" in the Managing Widgets in Application Code chapter.

Let's put all this together:

```
int rect_callback( PtWidget_t *widget,
                   void *client_data,
                   PtCallbackInfo_t *cbinfo)
```

```
{
    ...
}

int
add_shapes (PtWidget_t *widget, ApInfo_t *apinfo,
            PtCallbackInfo_t *cbinfo)
{
    PtArg_t    args[2];
    PtWidget_t *new_item;

    /* eliminate 'unreferenced' warnings */
    widget=widget, apinfo-apinfo, cbinfo=cbinfo;

    PtSetArg (&args[0], Pt_ARG_TEXT_STRING,
              "Rectangle", 0);
    PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "R", 0);
    new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                               2, args);
    PtAddCallback (new_item, Pt_CB_ACTIVATE,
                   rect_callback, NULL);

    /* Repeat the above for other shapes... */

    return (Pt_CONTINUE);
}
```

Creating submenus

You can create submenus in the menu created for a menu function item as follows:

1. Create a menu button for the cascade menu, setting the *Pt_ARG_BUTTON_TYPE* to Pt_MENU_RIGHT or Pt_MENU_DOWN, as required.
2. Save a pointer to the current parent widget by calling *PtGetParent()*:

   ```
   menu = PtGetParentWidget ();
   ```

3. Create a new **`PtMenu`** widget and set Pt_MENU_CHILD in the new menu's *Pt_ARG_MENU_FLAGS* resource.

`PtMenu` is a container, so this new menu becomes the current default parent.

4. Create submenu items, as described above.
5. Reset the default parent from the saved value by calling *PtSetParentWidget()*:

   ```
   PtSetParentWidget( menu );
   ```

6. Continue adding items to the top menu, if desired.

This example shows how to generate a submenu, as well as one way the *client_data* can be used in a generic callback to identify the item chosen from the menu:

```
/* A menu with a submenu                                    */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "abimport.h"
#include "proto.h"

/* Constants for the shapes in the menu */
#define RECTANGLE  1
#define CIRCLE     2
#define DOT        3
#define BLOB       4
#define POLYGON    5

int
ShapeMenuCB( PtWidget_t *widget, void *client_data,
             PtCallbackInfo_t *cbinfo )
{
  int shape_chosen = (int) client_data;

  widget=widget, client_data=client_data, cbinfo=cbinfo;

  /* This callback uses the client data to determine
     which shape was chosen. */

  switch (shape_chosen) {

    case RECTANGLE: ...
                    break;
    case CIRCLE   : ...
                    break;
    case DOT      : ...
                    break;
    case BLOB     : ...
                    break;
    case POLYGON  : ...
                    break;
    default       : printf ("Unknown shape: %d\n",
                            shape_chosen);
  }

  return (Pt_CONTINUE);
}

int
add_shapes( PtWidget_t *widget, ApInfo_t *apinfo,
            PtCallbackInfo_t *cbinfo )
{
  PtArg_t args[3];
  PtWidget_t *menu, *new_item;

  /* eliminate 'unreferenced' warnings */
  widget = widget, apinfo = apinfo, cbinfo = cbinfo;
```

```
PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Rectangle", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "R", 0);
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           2, args);
PtAddCallback ( new_item, Pt_CB_ACTIVATE, ShapeMenuCB,
                (void *)RECTANGLE );

PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Circle", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "C", 0);
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           2, args);
PtAddCallback ( new_item, Pt_CB_ACTIVATE, ShapeMenuCB,
                (void *)CIRCLE );

/* Create a menu button for the submenu. */

PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Miscellaneous", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "M", 0);
PtSetArg (&args[2], Pt_ARG_BUTTON_TYPE, Pt_MENU_RIGHT, 0 );
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           3, args);

/* Save the current default parent. */

menu = PtGetParentWidget();

/* Create a submenu. It becomes the new default parent. */

PtSetArg (&args[0], Pt_ARG_MENU_FLAGS,
    Pt_MENU_CHILD, Pt_MENU_CHILD);
new_item = PtCreateWidget( PtMenu, Pt_DEFAULT_PARENT, 1, args);

/* Add items to the submenu. */

PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Dot", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "D", 0);
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           2, args);
PtAddCallback ( new_item, Pt_CB_ACTIVATE, ShapeMenuCB,
                (void *)DOT );

PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Blob", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "B", 0);
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           2, args);
PtAddCallback ( new_item, Pt_CB_ACTIVATE, ShapeMenuCB,
                (void *)BLOB);

/* Restore the current default parent. */

PtSetParentWidget (menu);

/* Continue adding items to the top menu. */

PtSetArg (&args[0], Pt_ARG_TEXT_STRING, "Polygon", 0);
PtSetArg (&args[1], Pt_ARG_ACCEL_KEY, "P", 0);
new_item = PtCreateWidget( PtMenuButton, Pt_DEFAULT_PARENT,
                           2, args);
PtAddCallback ( new_item, Pt_CB_ACTIVATE, ShapeMenuCB,
                (void *)POLYGON);

return( Pt_CONTINUE );
```

```
}
```

Delaying and forcing updates to the display

If your application is making changes to a lot of widgets at once, you might want to delay updating the display until you're finished making the changes. Doing this can reduce flickering, and, in some cases, improve your application's performance.

You can delay updating:

- all of your application's widgets — the Photon libraries record any damage, but don't redraw the widgets.
- a specific container and its children — the libraries don't even record the damage.

Globally

The Photon libraries use a *hold count* to let you delay updating the display for your entire application:

- When the hold count is nonzero, the display isn't updated. To increment the hold count, call *PtHold()*.
- When you modify a widget, the libraries mark it as damaged.
- When the hold count is 0, the libraries repair damaged widgets as normal. To decrement the hold count, call *PtRelease()* or *PtUpdate()* (these two functions are identical).

For more information about these functions, see the Photon *Library Reference*.

For a specific container

The Photon libraries use *flux counts* to let you delay updating the display for a specific container. When the flux count is nonzero, and you modify the container or its children, the widgets aren't marked as damaged. What happens when the flux count returns to zero depends on which functions you use:

PtStartFlux()
PtEndFlux()
	When the container's flux count goes to zero, you must explicitly damage the areas you want to repair.

PtContainerHold()
PtContainerRelease()
	When the container's flux count goes to zero, the entire container is marked as damaged.

PtContainerHold() and *PtContainerRelease()* are easier to use, because you don't need to determine which widgets or areas you need to damage. However, there might be more flickering than if you use *PtStartFlux()* and *PtEndFlux()*.

If you need to determine if a container or any of its parents is currently in flux, call *PtIsFluxing()*.

For more information about these functions, see the Photon *Library Reference*.

Forcing updates

You can call *PtFlush()* at any time to immediately update the damaged areas of the display. *PtFlush()* ignores the hold count and doesn't change it.

If a container is in flux, and you modify it or its children, the Photon libraries don't mark the widgets as damaged, so *PtFlush()* doesn't repair them.

Combining holds on the whole application, holds on containers, and calls to *PtFlush()* can give you unexpected results. For example, if you hold the entire application, damage part of a container, hold the container, modify it, and then call *PtFlush()*, the libraries repair the damage — displaying whatever portion of the modifications that affect the damaged area.

Chapter 11

Manipulating Resources in Application Code

In this chapter. . .

This chapter describes how you can set and get the values of a widget's resources inside your application.

Although you can set the initial values of a widget's resources in PhAB, you'll probably need to access them from your code. For example:

- when a dialog appears, you may need to initialize some of the data it displays by setting resources beforehand
- when the user types a value in a **`PtText`** widget, you may need the value in your program, so you'll have to get resources.

In addition, if you use *PtCreateWidget()* to instantiate a widget in your code, you can give an initial value to its resources.

The value for the resource is specified or retrieved using an *argument list*.

There are *two* steps involved in specifying or retrieving more than one resource value:

- Setting up the argument list, using the *PtSetArg()* macro.
- Setting the value, using *PtSetResources()*, or retrieving the value, using *PtGetResources()*.

If you're getting or setting one resource, it's easier to use *PtGetResource()* or *PtSetResource()* — you don't need to set up the argument list.

Argument lists

An argument list is an array of **`PtArg_t`** structures (see the Photon *Library Reference*). Each of these elements identifies a widget resource and a new value for the resource (or the address of a variable that will be set to the resource's current value).

You can use the *PtSetArg()* macro to initialize each element of the argument list:

```
PtSetArg( PtArg_t *arg,
          long type,
          long value,
          long len );
```

If the values don't need to be calculated at runtime, you might be able to use *Pt_ARG()* instead to initialize the argument list. For more information, see the Photon *Library Reference*.

The first two arguments to *PtSetArg()* are the address of the argument list element, and the name of the resource. The third and fourth arguments vary, depending on the type of the resource, and on whether a set or a get operation is being applied. When setting a resource, the third argument is always used to hold a resource value or a pointer to a resource's value.

The fourth argument is used as either a size indicator or a mask, depending on the type of the value being specified. The possible resource types are given in the table below:

Type:	**Description:**
Alloc	An arbitrarily sized memory object
Array	An array
Boolean	A bit that's either on or off
Color	A color
Complex	A resource that's handled in a special way; see below.
Flag	A value in which each bit has a different meaning
Function	A pointer to a function
Image	A pointer to a **PhImage_t** structure
Link	A linked list
Pointer	A pointer to an address that you specify
Scalar	A value that can be represented within a single **long**
String	A null-terminated string
Struct	A fixed-size data type, usually a structure, **float**, or **double**

For information about the resources defined for each widget, see the Photon *Widget Reference*.

Complex resources are special; see their descriptions in the *Widget Reference* for instructions for setting and getting them. Widgets that have complex resources usually have convenience functions to make it easier to work with them.

Setting resources

Remember that there are *two* steps involved in setting more than one resource value:

- Setting up the argument list, using the *PtSetArg()* macro.
- Setting the value, using *PtSetResources()*.

If you're setting one resource, it's easier to use *PtSetResource()* — you don't need to set up the argument list. See "Setting one resource," below.

Argument lists for setting resources

Many of the sections that follow demonstrate setting some resources for a **PtComboBox** widget. Note that you can set more than one resource at a time. To do so, define an argument list of the appropriate length:

```
PtArg_t  args[5];
```

After initializing the argument list, you'll actually set the resources.

Scalar and color resources

When setting a scalar value, you should specify the value as the third argument to *PtSetArg()*. The fourth argument isn't used and should be set to 0.

For example, to set the bevel width of the combo box, pass the new value as the third argument:

```
PtSetArg(&args[0], Pt_ARG_BEVEL_WIDTH, 5, 0);
```

When you call *PtSetResources()*, the widget copies the scalar value into its own internal data structure.

String resources

Setting a string value is similar to setting a scalar value; you specify the string as the third argument to the *PtSetArg()* macro. The fourth argument is the number of bytes to copy; if it's 0, *strlen()* is used to determine the length of the string.

For example, to set the default text for the combo box, you could specify a value for the *Pt_ARG_TEXT_STRING* resource in one element of the argument list:

```
PtSetArg(&args[1], Pt_ARG_TEXT_STRING,
         "Rectangle", 0);
```

When you call *PtSetResources()*, the widget copies the string into its own internal data structure.

If you need to use international (non-ASCII) characters in a string, do one of the following:

- Define the string in a widget database and use the language editor to translate the string. See the International Language Support chapter.
- Use **`ped`** or some other UTF-compatible editor to edit the application's C code. You can then use the compose sequences described in "Photon compose sequences" in the Unicode Multilingual Support appendix.

  Most **`pterm`**-based editors, such as **`elvis`** and **`vedit`**, aren't UTF-compatible.

  For more information on **`ped`**, see the QNX Neutrino *Utilities Reference*.
- Look up the desired symbol in **`<photon/PkKeyDef.h>`**, use *wctomb()* to convert the character from Unicode to UTF-8, and then code the hexadecimal digits in your string. For example, the French word **`résumé`** would be coded as **`"r\xC3\xA9sum\xC3\xA9"`** — difficult to read, but it works with all editors.

For more information on Unicode and UTF-8, see the appendix on Unicode Multilingual Support.

Alloc resources

Some resources are designed to store an allocated block of memory. For example, every widget includes a *Pt_ARG_USER_DATA* resource that you can use to store any data you want in the widget's internal memory. To set this resource, pass a pointer to the data as the third argument to *PtSetArg()*. The fourth argument is the size of the block of memory, in bytes:

```
my_struct user_data;

/* Initialize the data */

PtSetArg(&args[2], Pt_ARG_USER_DATA, &user_data,
         sizeof (user_data));
```

The widget copies the number of bytes given into its internal memory when you call *PtSetResources()*.

Image resources

Image resources are designed to store a **`PhImage_t`** structure. For example, a **`PtLabel`** has a *Pt_ARG_LABEL_IMAGE* resource that you can use to store an image. To set this resource, create and initialize the **`PhImage_t`** structure, and pass a pointer to it as the third argument to *PtSetArg()*. The fourth argument is 0:

```
PhImage_t *my_image;

/* Create and initialize the image. */

PtSetArg(&args[2], Pt_ARG_LABEL_IMAGE, my_image, 0);
```

The widget copies the image structure (but not any memory pointed to by the **`PhImage_t`** members) into its internal memory when you call *PtSetResources()*.

Array resources

When setting an array value, the third argument to *PtSetArg()* is the address of the array. The fourth argument is the number of elements in the array.

For example, the following entry in the argument list can be used to set up *Pt_ARG_ITEMS*, the list of choices for the combo box:

```
char *cbox_items[3] = {"Circle", "Rectangle", "Polygon"};

PtSetArg(&args[3], Pt_ARG_ITEMS, cbox_items, 3);
```

The widget copies the contents of the array into its own internal data structure when you call *PtSetResources()*.

Flag resources

When setting a flag, the third argument to *PtSetArg()* is a bit field specifying the *value* of the bits to be set. The fourth argument is a *bit mask* indicating which elements of the bit field should be used.

For the value, use Pt_TRUE, Pt_FALSE, or a combination of specific bits and their complements. Don't use a value of 1, since it contains just one bit that's on; that bit might not correspond to the bit you're trying to set.

For example, the following argument list specification turns on the combo box widget's Pt_COMBOBOX_STATIC flag (so that the combo box always displays the list of items):

```
PtSetArg(&args[4], Pt_ARG_CBOX_FLAGS,
         Pt_TRUE, Pt_COMBOBOX_STATIC);
```

When you call *PtSetResources()*, the widget uses the bit mask to determine which bits of its internal flag resource representation to alter. It takes the bit values from the value specified.

Function resources

When setting a function resource, pass a pointer to the function as the third argument to *PtSetArg()*. The fourth argument is ignored; set it to 0.

For example, to specify a drawing function for a **PtRaw** widget, set the *Pt_ARG_RAW_DRAW_F* resource as follows:

```
PtSetArg( &args[0], Pt_ARG_RAW_DRAW_F,
          &my_raw_draw_fn, 0);
```

When you call *PtSetResources()*, the widget copies the pointer into the resource.

Pointer resources

When setting a pointer resource, the pointer must be given as the third argument to *PtSetArg()*. The fourth argument is ignored and should be set to 0.

When you call *PtSetResources()*, the widget simply does a shallow copy of the pointer into the resource.

The widget doesn't make a copy of the memory referenced by the pointer; don't free the memory while the widget is still referencing it.

For example, every widget includes a *Pt_ARG_POINTER* resource that you can use to store in the widget's internal memory a pointer to arbitrary data. The widget never refers to this data; it's just for you to use. To set this resource, allocate the desired memory, and pass a pointer to it as the third argument to *PtSetArg()*. The fourth argument is set to 0:

```
my_struct *user_data;

/* Allocate and initialize the data */

PtSetArg( &args[0], Pt_ARG_POINTER, user_data, 0 );
```

The widget copies the value of the pointer into its internal memory when you call *PtSetResources()*.

Link resources

When setting a Link, pass the address of an array of data as the third argument to *PtSetArg()*. The fourth argument has some special meanings:

num	append *num* items (if *num* is 0, one item is appended)
Pt_LINK_INSERT	insert the first array element at the beginning of the linked list
Pt_LINK_DELETE	remove the first list element that matches the first array element

The widget copies the data into its internal memory when you call *PtSetResources()*.

Struct resources

When setting a struct resource, pass the address of the data as the third argument to *PtSetArg()*. The fourth argument isn't used and should be set to 0.

The widget copies the data into its internal memory when you call *PtSetResources()*.

Boolean resources

When setting a Boolean value, you should specify the value as the third argument to *PtSetArg()*, using 0 for false, and a nonzero value for true. The fourth argument isn't used, and should be set to 0.

For example, to set the protocol for a **PtTerminal** to ANSI, pass a nonzero value as the third argument:

```
PtSetArg(&args[1], Pt_ARG_TERM_ANSI_PROTOCOL, 1, 0);
```

When you call *PtSetResources()*, the widget clears or sets one bit in its own internal data structure depending on whether or not the value is zero.

Calling *PtSetResources()*

Once you've set up the argument list, you're ready to set the resources. Remember that *PtSetArg()* doesn't set the resources; it just sets up the argument list.

You can use *PtSetResources()* to set the new values for resources:

```
int PtSetResources( PtWidget_t *widget,
                    int n_args,
                    PtArg_t *args );
```

The arguments to this function are a pointer to the widget, the number of entries in the argument list, and the argument list itself.

You can also set resources by passing an argument list to *PtCreateWidget()*. The rules for specifying values in argument list elements are the same. For more information, see "Creating widgets" in the Managing Widgets in Application Code chapter.

For example, you could set the resources of a combo box, using the argument list created above. Call *PtSetResources()* as follows:

```
PtSetResources (ABW_shapes_cbox, 5, args);
```

In response to a change to its resources, a widget may have to redisplay itself. The *PtSetResources()* call triggers this change. Any changes to the appearance of the widget, however, don't take effect until control is restored to the Photon event-handling loop. Therefore, if *PtSetResources()* is called from within a callback function or an event-handling function, the change to the widget won't be visible until all the callbacks in the callback list and all event handlers have been executed.

Setting one resource

If you're setting one resource, it's easier to use *PtSetResource()* than *PtSetResources()*. With *PtSetResource()*, you don't need to set up the argument list.

The arguments to *PtSetResource()* are a combination of those for *PtSetArg()* and *PtSetResources()*:

```
int PtSetResource( PtWidget_t *widget,
                   long type,
                   long value,
                   long len );
```

The *widget* is a pointer to the widget whose resource you're setting. The other arguments are set just as they are for *PtSetArg()* when setting more than one resource. See "Argument lists for setting resources," above.

For example, setting one resource with *PtSetResources()* requires code like this:

```
PtArg_t args[1];

PtSetArg(&args[0], Pt_ARG_BEVEL_WIDTH, 5, 0);
PtSetResources (ABW_shapes_cbox, 1, args);
```

Setting the same resource with *PtSetResource()* is like this:

```
PtSetResource (ABW_shapes_cbox,
               Pt_ARG_BEVEL_WIDTH, 5, 0);
```

It takes just one function call, and there's no need for an *args* array.

Getting resources

There are *two* steps involved in retrieving more than one resource value:

- Setting up the argument list, using the *PtSetArg()* macro.
- Getting the value, using *PtGetResources()*.

If you're getting one resource, it's easier to use *PtGetResource()* — you don't need to set up the argument list. See "Getting one resource," below.

There are two methods of getting resources: one that involves pointers, and one that doesn't. The nonpointer method is usually easier and safer:

- Since you're getting a copy of the value, the chances of overwriting the original by accident are smaller.

- You don't need to worry about the type of the value (**`short`** versus **`long`**).
- You have fewer local variables and don't use pointers to them, which makes your code easier to read and helps the compiler generate better code.

The pointer method may be less confusing if you're getting the values of several resources at once; you'll have named pointers to the values instead of having to remember which element in the argument list corresponds to which resource.

Not using pointers

If you set the *value* and *len* arguments to *PtSetArg()* to zero, *PtGetResources()* returns the resource's value (converted to **`long`**) as follows:

Resource type	***value***	***len***
Flags (any C type)	Value of the resource	N/A
Scalar (any C type)	Value of the resource	N/A
Pointer (any C type)	Value of the resource	N/A
String	Address of the string	N/A
Struct	Address of the data	N/A
Array	Address of the first array item	Number of items in the array
Alloc	Address of where the resource is stored	N/A
Boolean	0 (false) or 1 (true)	N/A

Scalar and flag resources (nonpointer method)

To get a scalar or flag resource (of any C type) with the nonpointer method:

```
unsigned long getscalar( PtWidget_t *widget, long type ) {
    /* Get any kind of scalar */
    PtArg_t arg;
    PtSetArg( &arg, type, 0, 0 );
    PtGetResources( widget, 1, &arg );
    return arg.value;
    }
```

String resources (nonpointer method)

Here's how to use the nonpointer method to get the value of a string resource:

```
const char *getstr2( PtWidget_t *widget, long type ) {
    PtArg_t arg;

    PtSetArg( &arg, type, 0, 0 );
    PtGetResources( widget, 1, &arg );
    return (char*) arg.value;
    }
```

Boolean resources (nonpointer method)

In the nonpointer method to get a boolean, the value (0 or 1) is returned in *value* argument to *PtSetArg()*:

```
int getbool( PtWidget_t *widget, long type ) {
    PtArg_t arg;

    PtSetArg( &arg, type, 0, 0 );
    PtGetResources( widget, 1, &arg );
    return arg.value;
    }
```

Using pointers

When using the pointer method to get a scalar, array, or flag resource, the widget always gives a pointer to an internal widget data structure. In the argument list element you set up using *PtSetArg()*, you must provide the address of a variable to which the internal data pointer can be assigned.

The fourth argument isn't used for most types of resources. For arrays, it's the address of a pointer that on return from *PtGetResources()* points to the number of entries.

For example, to obtain the contents of the *Pt_ARG_FLAGS* resource (which is a **long**) for a widget, you must pass the *address of a pointer* to a **long**:

```
const long *flags;
PtArg_t arg[1];

PtSetArg(&arg[0], Pt_ARG_FLAGS, &flags, 0);
PtGetResources(ABW_label, 1, arg);
```

CAUTION: *PtGetResources()* returns pointers directly into the widget's internal memory. Don't attempt to modify the resources directly using these pointers. Such a modification won't have the desired effect and will likely corrupt the widget's behavior. *Never free these pointers either* — this will certainly result in a memory violation or some other fault.

Using **const** pointers will help avoid these problems.

Changes to the widget's state may invalidate these pointers; use them promptly.

If you wish to retrieve the value of a given resource and then modify that value:

1. Get the resource.
2. Copy the resource to a temporary variable.
3. Modify the temporary variable.
4. Using the modified copy, set the resource.

You *can* use the value obtained to set the value of another resource of this or any other widget, as long as you don't change the original value.

For example, you can use the following code to obtain *Pt_ARG_TEXT_STRING*, the text string displayed in the label widget named **label**:

```
char *str;
PtArg_t args[1];

PtSetArg(&args[0], Pt_ARG_TEXT_STRING, &str, 0);
PtGetResources(ABW_label, 1, args);
```

You can then assign this text string to another label named **label2**:

```
PtSetArg(&args[0], Pt_ARG_TEXT_STRING, str, 0);
PtSetResources(ABW_label2, 1, args);
```

Scalar and flag resources (pointer method)

If you're getting scalar or flag resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer to the appropriate C type.
- *len* isn't used.

When *PtGetResources()* is called, the pointer is set to point to the widget's internal storage for that resource.

Here are some functions that get a scalar or flag resource, using the pointer method:

```
unsigned long getlong( PtWidget_t *widget, long type ) {
    /* Get a long or long flags */
    PtArg_t arg; unsigned long const *result;

    PtSetArg( &arg, type, &result, 0 );
    PtGetResources( widget, 1, &arg );
    return *result;
    }

unsigned getshort( PtWidget_t *widget, long type ) {
    /* Get a short or short flags */
    PtArg_t arg; unsigned short const *result;

    PtSetArg( &arg, type, &result, 0 );
    PtGetResources( widget, 1, &arg );
    return *result;
    }

unsigned getbyte( PtWidget_t *widget, long type ) {
    /* Get a char or char flags */
    PtArg_t arg; unsigned char const *result;

    PtSetArg( &arg, type, &result, 0 );
    PtGetResources( widget, 1, &arg );
    return *result;
    }
```

String resources (pointer method)

If you're getting string resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a **`char`** pointer.
- *len* isn't used.

When *PtGetResources()* is called, the pointer specified is set to point to the widget's internal storage for the string resource. For example:

```
const char *getstr1( PtWidget_t *widget, long type ) {
    PtArg_t arg; const char *str;

    PtSetArg( &arg, type, &str, 0 );
    PtGetResources( widget, 1, &arg );
    return str;
    }
```

Alloc resources (pointer method)

If you're getting alloc resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer of the appropriate type (the type is determined by the data given to the widget when this resource is set).
- The *len* isn't used.

When *PtGetResources()* is called, the pointer specified is set to point to the widget's internal data.

Image resources (pointer method)

If you're getting Image resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer to a **`PhImage_t`** structure.
- The *len* isn't used.

When *PtGetResources()* is called, the pointer specified is set to point to the widget's internal data.

Array resources (pointer method)

If you're getting array resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer of the appropriate C type (the first of the two C types given in the "New Resources" table).
- *len* is the address of a pointer of the second C type given.

When *PtGetResources()* is called:

- The pointer given by *value* is set to point to the beginning of the array in the widget's internal storage.

- The pointer given by *len* is set to point to the array-item count in the widget's internal storage.

Pointer resources (pointer method)

If you're getting pointer resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer of the appropriate C type.
- *len* isn't used.

When *PtGetResources()* is called, the pointer specified is set to point to the same data as the widget's internal pointer. The data is external to the widget; you might be able to modify it, depending on the resource.

Link resources (pointer method)

If you're getting link resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer to a **`PtLinkedList_t`** list structure. This structure contains at least:

  `struct Pt_linked_list *`*next*

  A pointer to the next item in the list.

  `char` *data[1]* The address of the data stored in the list.

- *len* isn't used.

When *PtGetResources()* is called, The pointer given by *value* is set to point to the first node of the widget's internal linked list.

If you get a callback resource, the *value* argument to *PtSetArg()* is the address of a pointer to a **`PtCallbackList_t`** structure. For more information, see "Examining callbacks" in the Managing Widgets in Application Code chapter.

Struct resources (pointer method)

If you're getting struct resources using the pointer method:

- The *value* argument to *PtSetArg()* is the address of a pointer of the appropriate C type.
- *len* isn't used.

When *PtGetResources()* is called, the pointer specified is set to point to the widget's internal storage for the struct resource.

Boolean resources (pointer method)

If you're getting boolean resources using the pointer method:

- The *value* argument to *PtSetArg()* is a pointer to an **`int`**.
- *len* isn't used.

When *PtGetResources()* is called, the **`int`** is set to 1 if the Boolean is true, or 0 if it's false.

For example, to get the value of the *Pt_ARG_CURSOR_OVERRIDE* resource of a **`PtContainer`**:

```
PtArg_t arg;
int     bool_value;

PtSetArg( &arg[0], Pt_ARG_CURSOR_OVERRIDE, &bool_value, 0 );
PtGetResources (ABW_container, 1, arg);

if ( bool_value ) {
  /* The container's cursor overrides that of its children. */
}
```

Calling *PtGetResources()*

Use *PtGetResources()* to obtain the values of each of the resources identified in an argument list:

```
int PtGetResources( PtWidget_t *widget,
                    int n_args,
                    PtArg_t *args );
```

The arguments to this function are the identifier for the widget, the number of entries in the argument list, and the argument list itself.

PtGetResources() returns 0 on success, or -1 if an error occurs. A return code of -1 might indicate that you've tried to get the value of a resource that isn't defined for the widget.

Getting one resource

If you're getting the value of one resource, it's easier to use *PtGetResource()* than *PtGetResources()*. With *PtGetResource()*, you don't need to set up the argument list. The arguments to *PtGetResource()* are:

```
int PtGetResource( PtWidget_t *widget,
                   long type,
                   long value,
                   long len );
```

The *widget* is a pointer to the widget whose resource you're getting. The other arguments are set just as they are for *PtSetArg()* when getting more than one resource using the pointer method.

Here's an example of getting one resource with *PtGetResources()* and the pointer method:

```
unsigned short *width;
PtArg_t arg;

PtSetArg( &arg, Pt_ARG_BEVEL_WIDTH, &width, 0 );
PtGetResources( widget, 1, &arg );
```

With *PtGetResource()*, the code is like this:

```
unsigned short *width;

PtGetResource( widget, Pt_ARG_BEVEL_WIDTH, &width, 0 );
```

CAUTION: *PtGetResource()* returns a pointer directly into the widget's internal memory. Don't attempt to modify the resource directly using this pointer. Such a modification won't have the desired effect and will likely corrupt the widget's behavior. *Never free the pointer either* — this will certainly result in a memory violation or some other fault.

Using a **`const`** pointer will help avoid these problems.

Changes to the widget's state may invalidate the pointer; use it promptly.

Application-level resources

Applications have callback resources that you can set and get, just like widgets, except the resources apply to the application as a whole instead of individual widget instances. These resources apply to applications:

- Pt_CB_APP_EXIT
- Pt_CB_APP_WCLASS_CREATED
- Pt_CB_FILTER
- Pt_CB_RAW
- Pt_CB_HOTKEY

At this time, application-level resources are all callback resources. There may be other resource types in a future version of the Photon library.

To manipulate application-level resources, you can use these functions:

- *PtAppAddCallback()*
- *PtAppGetResource()*
- *PtAppGetResources()*
- *PtAppRemoveCallback()*
- *PtAppSetResource()*

- *PtAppSetResources()*

The application-level resource functions are similar to their widget counterparts, except you don't specify a widget.

Setting resources

You can set application-level resources using these functions:

- *PtAppAddCallback()*
- *PtAppSetResource()*
- *PtAppSetResources()*

If you are adding a single callback to an application's callback list, *PtAppAddCallback()* is the easiest method. For example, here is an application exit callback function that prints a message to the standard output when the application exits:

```
int exit_cb(void *data,
            PtCallbackInfo_t *cbinfo)
{
  printf( "I\'m exiting\n" );
  return( Pt_CONTINUE );
};
```

To add this callback to the application's Pt_CB_APP_EXIT callback list using *PtAppAddCallback()*, you would put this in the application's initialization code:

```
PtAppAddCallback(Pt_CB_APP_EXIT, exit_cb, NULL);
```

You can also set a single callback using *PtAppSetResource()*, but instead of passing it a pointer to the callback, you need to pass it the address of a **PtAppCallback_t**:

```
PtAppCallback_t exit_callback = {exit_cb, NULL};
PtAppSetResource(Pt_CB_APP_EXIT, &exit_callback, 0);
```

To use the *PtAppSetResources()* function, you'll need to create an argument list using *PtSetArg()*. For example:

```
PtAppCallback_t exit_callbacks[] = {{exit_cb, NULL}};
PtArg_t args[1];

PtSetArg( &args[0], Pt_CB_APP_EXIT, exit_callbacks,
          sizeof(exit_callbacks)/sizeof(exit_callbacks[0]));
PtAppSetResources( 1, args );
```

Removing callbacks

You can remove a callback using *PtAppRemoveCallback()*. It takes the same arguments as *PtAppAddCallback()*. For example, to remove the callback added in the examples above:

```
PtAppRemoveCallback( Pt_CB_APP_EXIT, exit_cb, NULL );
```

Getting callbacks

You can retrieve a pointer to an application callback to examine it. You can use *PtAppGetResource()* to get a single callback, or *PtAppGetResources()* to get one or more.

For example, to retrieve a pointer to the application exit callback added in the previous example, you would use:

```
PtAppCallback_t *my_exit_callback;

PtAppGetResource(Pt_CB_APP_EXIT, &my_exit_callback, 0 );
```

See the section on Getting Resources for more information.

Chapter 12

Managing Widgets in Application Code

In this chapter. . .

We recommend that you create your application's UI in PhAB — it's easier than doing it in your code. However, if the interface is dynamic, you'll probably have to create parts of it "on the fly."

Creating widgets

Creating a widget in your application code is a bit more work than creating it in PhAB. That's because PhAB looks after a lot of the physical attributes for you, including size, location, and so on. If you create the widget in your code, you'll have to set these resources yourself.

To create a widget in your code, call *PtCreateWidget()*. The syntax is as follows:

```
PtWidget_t *PtCreateWidget(
                PtWidgetClassRef_t *class,
                PtWidget_t *parent,
                unsigned n_args,
                PtArg_t *args );
```

The arguments are:

class	The type of widget to create (e.g. **PtButton**)
parent	The parent of the new widget. If this is Pt_DEFAULT_PARENT, the new widget is made a child of the default parent, which is the most recently created container-class widget. If *parent* is Pt_NO_PARENT, the widget has no parent.
n_args	The number of elements in the *args* array.
args	An array of **PtArg_t** structures (see the Photon *Library Reference*) that store your settings for the widget's resources. These settings are like the ones used for *PtSetResources()*; see the Manipulating Resources in Application Code chapter.

You can specify the default parent (used if the *parent* argument to *PtCreateWidget()* is Pt_DEFAULT_PARENT) by calling *PtSetParentWidget()*. To assign a widget to a different container, call *PtReparentWidget()*.

Here are a few things to note about widgets created in application code:

- The widget isn't realized until the container widget is realized. If the container is already realized, you can call *PtRealizeWidget()* to realize the new widget.
- If you create a widget in a PhAB module and then destroy the module, the widget is destroyed, too. The next time the module is created, it will appear as it was specified in PhAB.
- If you save a global pointer to the widget, make sure you reset it to NULL when the widget is destroyed. This can easily be done in the widget's *Pt_CB_DESTROYED* callback. Failing to reset the global pointer (and check it before using it) is a frequent source of problems with widgets created in code.

Ordering widgets

The order in which widgets are given focus depends on the order in which they were created or on the widget order specified in PhAB (see "Ordering widgets" in the Creating Widgets in PhAB chapter). The backmost widget is the first in the tab order; the frontmost widget is the last.

If you're creating widgets programmatically, you can create them in the order in which you want them to get focus, or you can use these functions to change the order:

PtWidgetInsert()	Insert a widget in the widget family hierarchy
PtWidgetToBack()	Move a widget behind all its brothers
PtWidgetToFront()	Move a widget in front of all its brothers

Alternatively, you can use a widget's *Pt_CB_LOST_FOCUS* callback (defined by **`PtBasic`**) to override the tab order by giving focus to another widget.

In the lost-focus callback, use *PtContainerGiveFocus()* to give focus to the desired widget, and return Pt_END from the callback to prevent focus from being given to the original target of the focus change.

The *Pt_CB_LOST_FOCUS* callback is called a second time as focus is removed from the widget to go to the new target. To avoid an endless loop, use a static variable to indicate that this callback has already redirected focus.

Working in the widget family

The following functions can be used to work with the widget family hierarchy, and may be useful in setting the focus order:

PtChildType()	Determine the relationship between two widgets
PtFindDisjoint()	Return the nearest disjoint parent widget
PtFindFocusChild()	Find the closest focusable child widget
PtFindGuardian()	Find the widget responsible for another widget's actions
PtGetParent()	Find the nearest parent widget that matches the specified class
PtGetParentWidget()	Return the current default widget parent
PtNextTopLevelWidget()	Get a pointer to the next top-level widget
PtValidParent()	Identify a valid parent for a widget

PtWidgetBrotherBehind()	Get the brother behind a widget
PtWidgetBrotherInFront()	Get the brother in front of a widget
PtWidgetChildBack()	Get the child that's farthest back in a container
PtWidgetChildFront()	Get the child at the very front of a container
PtWidgetFamily()	Traverse the widget hierarchy from back to front
PtWidgetParent()	Get a widget's parent
PtWidgetSkip()	Skip to a widget in the next hierarchy
PtWidgetTree()	Walk the widget tree from front to back
PtWidgetTreeTraverse()	Walk the widget family hierarchy from front to back

Callbacks

You can add and remove callbacks in your code as well as from PhAB — just watch for differences between the two types!

Adding callbacks

An application registers callbacks by manipulating the widget's callback resources. The Photon widget classes employ a naming convention for these resources — they all begin with **`Pt_CB_`**.

Callbacks can be added to the callback list kept by these resources using *PtAddCallbacks()* to add several callback functions to the list or *PtAddCallback()* to add just one. In either case, the first two arguments to the function are the widget and the name of the callback resource to be augmented. The remaining arguments depend on which function is used.

The third argument to *PtAddCallbacks()* is an array of callback records. Each record contains a pointer to a callback function and the associated client data pointer that will be passed to the callback function when it's invoked. Each of these callback records is copied to the widget's internal callback list.

For example, we might want to have the application perform some action when the user selects (i.e. presses) a button. The **`PtButton`** widget class provides the *Pt_CB_ACTIVATE* callback resource for notifying the application when the button has been pressed. To create the widget and attach a callback function to this callback resource, we'd have to use code like this:

```
{
   PtWidget_t *button;
   int push_button_cb( PtWidget_t *, void *,
                       PtCallbackInfo_t *);
   PtCallback_t callbacks[] = { {push_button_cb, NULL} };

   ...

   button = PtCreateWidget(PtButton, window, 0, NULL);
   PtAddCallbacks(button, Pt_CB_ACTIVATE, callbacks, 1);
}
```

where *push_button_cb* is the name of the application function that would be called when the user presses the button. The **PtCallback_t** structure is used to define lists of callbacks; for details, see the Photon *Widget Reference*.

When adding only one callback function to the callback list (as in this case), it's simpler to use *PtAddCallback()*. This function takes the pointer to the callback function as the third argument, and the client data pointer as the final argument. The above code fragment could be written more concisely as:

```
{
   PtWidget_t *button;
   int push_button_cb( PtWidget_t *, void *,
                       PtCallbackInfo_t *);
   button = PtCreateWidget(PtButton, window, 0, NULL);
   PtAddCallback(button, Pt_CB_ACTIVATE, push_button_cb,
                 NULL);
}
```

You can also give an array of callback records as the value for the callback resource when using argument lists in conjunction with *PtCreateWidget()* or *PtSetResources()*. Since the callback list is an array, you should specify the array's base address as the third argument to *PtSetArg()*, and the number of elements as the final argument. In this case, the callback records are added to the current callback list, if there is one. This gives us another way to specify the callback for the above example:

```
{
   PtArg_t arg[5];
   int push_button_cb( PtWidget_t *, void *,
                       PtCallbackInfo_t *);
   PtCallback_t callbacks[] = { {push_button_cb, NULL} };
...
   PtSetArg(&args[0], Pt_CB_ACTIVATE, callbacks, 1);
   PtCreateWidget(PtButton, window, 1, arg);
}
```

Each of these methods has its advantages. *PtAddCallback()* is of course simple. *PtAddCallbacks()* is more efficient when there are several callbacks. Using *PtSetArg()* and passing the result to *PtCreateWidget()* allows the widget creation and callback list attachment to be performed atomically.

Callback invocation

When called, the callback function is invoked with the following parameters:

PtWidget_t **widget*

The widget that caused the callback function to be called, i.e. the one on which the action took place.

void **client_data*

Application-specific data that was associated with the callback when it was registered with the widget.

The client data that's passed to a callback you add from your code isn't the same as the *apinfo* data passed to a PhAB callback.

PtCallbackInfo_t **call_data*

A pointer to a **PtCallbackInfo_t** structure (see the Photon *Widget Reference*) that holds data specific to this invocation of the callback. It relates to the reason the callback was called and may have data specific to the callback's behavior.

The **PtCallbackInfo_t** structure is defined as:

```
typedef struct Pt_callback_info {
        unsigned long     reason;
        unsigned long     reason_subtype;
        PhEvent_t         *event;
        void              *cbdata;
} PtCallbackInfo_t;
```

The elements of **PtCallbackInfo_t** have the following meaning:

- *reason* — indicates the reason the callback was called; this is normally set to the name of the callback resource whose callback list has been called.
- *reason_subtype* — indicates a particular callback type associated with the reason; for most callbacks, this value is zero.
- *event* — a pointer to a **PhEvent_t** structure (see the Photon *Library Reference*) that describes the Photon event that caused the callback to be invoked.
- *cbdata* — call data that is specific to the callback resource that caused the callback function to be called.

For more information, see the descriptions of the callbacks defined for each widget in the *Widget Reference*.

Removing callbacks

You can remove one or more callbacks from a callback list associated with a widget resource using the *PtRemoveCallbacks()* and *PtRemoveCallback()* functions.

CAUTION: Don't try to remove a callback that was added through PhAB; unexpected behavior may result.

PtRemoveCallbacks() takes an array of callback records as an argument and removes all the callbacks specified by it from the callback list. *PtRemoveCallback()* removes just one callback function from the callback list. Both functions take the widget as the first argument and the widget resource as the second argument.

To remove the callback from the button we've created above, we could do this:

```
int push_button_cb( PtWidget_t *, void *,
                    PtCallbackInfo_t *);
PtCallback_t callbacks[] = { {push_button_cb, NULL} };
PtRemoveCallbacks(button, Pt_CB_ACTIVATE, callbacks, 1);
```

or this:

```
int push_button_cb( PtWidget_t *, void *,
                    PtCallbackInfo_t *);
PtRemoveCallback(button, Pt_CB_ACTIVATE, push_button_cb,
```

Both the callback function pointer and the client data pointer are important when removing callbacks. Only the first element of the callback list that has both the same callback function *and* the same client data pointer will be removed from the callback list.

Examining callbacks

You can examine the callback list by getting the value of the appropriate callback list resource. The type of value you get from a callback list resource is different from the value used to set the resource. Although this resource is set with an array of callback records, the value obtained by getting the resource is a *pointer* to a list of callback records. The type of the list is **PtCallbackList_t**. Each element of the list contains a *cb* member (i.e. the callback record) and a *next* pointer (which points to the next element of the list).

The following example shows how you can traverse through the *Pt_CB_ACTIVATE* callback list for *widget* to find all instances of a particular callback function, *cb*:

```
...
PtCallbackList_t *cl;

PtGetResources(widget, Pt_CB_ACTIVATE, &cl, 0);
for ( ; cl; cl = cl->next )
{
    if ( cl->cb.func == cb )
        break;
}
```

Event handlers

You can add and remove event handlers (raw and filter callbacks) in your application code as well as in PhAB — however, there are some differences between the two types.

For a description of raw and filter callbacks and how they're used, see "Event handlers — raw and filter callbacks" in the Events chapter.

For information on adding event handlers in PhAB, see "Event handlers — raw and filter callbacks" in the Editing Resources and Callbacks in PhAB chapter.

Adding event handlers

As with callbacks, you can also set or examine event handlers by performing a set or get directly on the event handler resource. The following resources of **`PtWidget`** let you specify handlers for Photon events:

- *Pt_CB_FILTER*
- *Pt_CB_RAW*

For more information about these callback resources, see the Photon *Widget Reference*.

The set operation requires an array of event handler records of type **`PtRawCallback_t`**. These are similar to the callback records mentioned above, having *event_mask*, *event_f*, and *data* fields.

The *event mask* is a mask of Photon event types (see **`PhEvent_t`** in the Photon *Library Reference*) indicating which events will cause the callback function to be invoked. The *event_f* and *data* members are the event handler function and client data, respectively.

If you add an event handler to a realized widget and the widget's region isn't sensitive to one or more of the events contained in the event mask, then the region is made sensitive to them.

If you add the event handler *before* realizing the widget, you have to adjust the region's sensitivity yourself after realizing the widget. See *PhRegionChange()* in the Photon *Library Reference*.

A get operation yields a **`PtRawCallbackList_t *`** list of event handler records. As with callback lists, the list contains two members: *next* and *cb*. The *cb* member is an event handler record.

You can add *Pt_CB_RAW* event handlers using either the *PtAddEventHandler()* or *PtAddEventHandlers()* function.

You can add *Pt_CB_FILTER* event handlers using either the *PtAddFilterCallback()* or *PtAddFilterCallbacks()* function.

The arguments to *PtAddEventHandler()* and *PtAddFilterCallback()* are:

widget	Widget to which the event handler should be added.
event_mask	Event mask specifying which events should cause the event handler to be called.

event_f	Event-handling function.
data	A pointer to pass to the event handler as client data.

The arguments to *PtAddEventHandlers()* and *PtAddFilterCallbacks()* are:

widget	Widget to which the event handlers should be added.
handlers	Array of event handler records.
nhandlers	Number of event handlers defined in the array.

Removing event handlers

You can remove *Pt_CB_RAW* event handlers by calling either *PtRemoveEventHandler()* or *PtRemoveEventHandlers()*.

You can remove *Pt_CB_FILTER* event handlers by calling either *PtRemoveFilterCallback()* or *PtRemoveFilterCallbacks()*

CAUTION: Don't remove event handlers that were added through PhAB; unexpected behavior may result.

The parameters to *PtRemoveEventHandler()* and *PtRemoveFilterCallback()* are:

widget	Widget from which the event handler should be removed.
event_mask	Event mask specifying the events the handler is responsible for.
event_f	Event-handling function.
data	Client data associated with the handler.

This looks for an event handler with the same signature — i.e. the same *event_mask*, *data* and *event_f* — in the widget and removes one if it's found.

The parameters to *PtRemoveEventHandlers()* and *PtRemoveFilterCallbacks()* are:

widget	Widget from which the event handlers should be removed.
handlers	Array of event-handler records.
nhandlers	Number of event handlers defined in the array.

As with *PtRemoveEventHandler()* and *PtRemoveFilterCallback()*, an event handler is removed only if it has the exact same signature as one of the event handler specifications in the array of event handler records.

Event handler invocation

When invoked, event handlers receive the same arguments as callback functions, i.e. the parameters are:

- the widget that received the event (*widget*)
- the client data associated with the event handler (*client_data*)

The client data passed to this event handler isn't the same as the *apinfo* data passed to an event handler added through PhAB.

- the callback information associated with the particular event (*info*).

Event handlers return an integer value that the event handler must use to indicate whether or not further processing should be performed on the event. If the event handler returns the value Pt_END, this indicates that no further processing is to be performed on the Photon event, and the event is consumed.

The *event* member of the *info* parameter contains a pointer to the event that caused the event handler to be invoked. You should check the *type* member of this event to determine how to deal with the event. It will be one of the event types specified in the *event_mask* given when the event handler was added to the widget.

To retrieve the data associated with the particular event, call the *PhGetData()* with the pointer to the event as a parameter. This will return a pointer to a structure with the data specific to that particular event type. This structure's type depends on the event type.

Widget styles

Widget class styles let you customize or modify a widget's appearance, size, and behavior at runtime. They also let multiple looks for a single type of widget exist at the same time. Essentially, a widget class style is a collection of methods and data that define the look and feel of instances of the widget class.

Each widget class has a default style, but you can add or modify an arbitrary number of additional styles at any time. You can even modify the default style for a class, changing the look and feel of any instances of that class that are using the default style.

Each instance of a widget can reference a specific style provided by its class. You can change the style that any widget is using whenever you want.

Each style has a set of *members*, including a name for the style and functions that replace or augment some of the widget class's *methods*. Methods are class-level functions that define how the widget initializes itself, draws itself, calculates its extent, and so on. For more information about methods, see the *Building Custom Widgets* guide.

The members of a style are identified by the following manifests:

Pt_STYLE_DRAW	The address of a function that's called whenever any widget that's using this style needs to draw itself.
Pt_STYLE_EXTENT or Pt_STYLE_SIZING	The address of a function that whenever a widget that's using this style is moved, resized, or modified in some fashion that may require the widget to move or resize (change in widget data). This function is responsible for setting the widget's dimension to the appropriate values.
Pt_STYLE_ACTIVATE	The address of a function that's called whenever a widget is created that defaults to this style, and whenever a widget's style is changed from some other style to this one. This function is the place to put manipulation of a widget's control surfaces, the addition of callbacks, or the setting of resources (to override the widget's defaults).
Pt_STYLE_CALC_BORDER	The address of a function that's responsible for reporting how much space is required to render the widget's edge decorations and margins.
Pt_STYLE_CALC_OPAQUE	The address of a function that's responsible for calculating the list of tiles that represents the opaque areas of a widget. This list is used to determine what needs to be damaged *below* this widget when it's modified.
Pt_STYLE_DEACTIVATE	The address of a function that's called whenever a widget using this style is either being destroyed or is switching to a different style.
Pt_STYLE_NAME	The name of the style.
Pt_STYLE_DATA	A pointer to an arbitrary data block for the style's use.

For details about the members, see *PtSetStyleMember()*.

The following functions let you create and manipulate the widget class styles:

PtAddClassStyle()	Add a style to a widget class
PtCreateClassStyle()	Create a class style
PtDupClassStyle()	Get a copy of a widget class style
PtFindClassStyle()	Find the style with a given name

PtGetStyleMember()	Get a member of a style
PtGetWidgetStyle()	Get the style that a widget is currently using
PtSetClassStyleMethods()	Set multiple members of a style from an array
PtSetStyleMember()	Set a member of a style
PtSetStyleMembers()	Set multiple members of a style from a variable-length argument list
PtSetWidgetStyle()	Set the current style for a widget

Some of these functions require or return a pointer to a **`PtWidgetClassStyle_t`** structure. Don't access the members of this structure directly; call *PtGetStyleMember()* instead.

You can also set the style for a widget instance by setting its *Pt_ARG_STYLE* resource (see **`PtBasic`** in the *Widget Reference*). Setting this resource has the same effect as calling *PtSetWidgetStyle()*.

This example creates a style called **`blue`** and some buttons. Note that your widgets can use a style before you've added the style to the class or even before you've created the style. When you do create the style and add it to the class, any widgets that use the style are updated immediately.

```
#include <Pt.h>

PtWidget_t *win, *but;
PtWidgetClassStyle_t *b;

void blue_draw (PtWidget_t *widget, PhTile_t *damage)
{

    /* This is the drawing function for the blue style.
       It draws a blue rectangle (without a label) for
       the widget. */

    PgSetFillColor( Pg_BLUE);
    PgDrawRect( PtWidgetExtent (widget,NULL),
                Pg_DRAW_FILL);
}

int use_blue_style( PtWidget_t *widget, void *data,
                    PtCallbackInfo_t *cbinfo)
{
    /* This callback sets the current style for the given
       widget instance. If you haven't attached the blue
       style to the class, there shouldn't be any change
       in the widget's appearance. */

    PtSetWidgetStyle (widget, "blue");
    return Pt_CONTINUE;
}
```

```
int attach_blue_style( PtWidget_t *widget, void *data,
                       PtCallbackInfo_t *cbinfo)
{

    /* This callback adds the style to the widget class.
       If you've clicked on one of the "Use blue style"
       buttons, the style of all buttons should change. */

    PtAddClassStyle (PtButton, b);
    return Pt_CONTINUE;
}

int main()
{
    PhArea_t area = {{0,50},{100,100}};
    PtArg_t argt[10];
    PtStyleMethods_t meth;
    PtCallback_t cb = {use_blue_style, NULL};
    PtCallback_t cb2 = {attach_blue_style, NULL};
    int unsigned n;

    /* Initialize the methods for the style. */
    meth.method_index = Pt_STYLE_DRAW;
    meth.func = blue_draw;

    PtInit(NULL);

    /* Create the window. */
    PtSetArg (&argt[0], Pt_ARG_DIM, &area.size, 0);
    win = PtCreateWidget (PtWindow, NULL, 1, argt);

    /* Create some buttons.  When you click on one of these
       buttons, the callback makes the widget instance use
       the blue style. */
    n = 0;
    PtSetArg (&argt[n++], Pt_ARG_TEXT_STRING,
              "Use blue style", 0);
    PtSetArg (&argt[n++], Pt_CB_ACTIVATE, &cb, 1);
    but = PtCreateWidget (PtButton, NULL, n, argt);

    PtSetArg (&argt[0], Pt_ARG_TEXT_STRING,
              "Use blue style also", 0);
    PtSetArg (&argt[n++], Pt_ARG_POS, &area.pos, 0);
    but = PtCreateWidget (PtButton, NULL, n, argt);

    /* Create another button.  When you click on it, the
       callback attaches the blue style to the widget class. */

    n = 0;
    PtSetArg (&argt[n++], Pt_ARG_TEXT_STRING,
              "Attach blue style", 0);
    PtSetArg (&argt[n++], Pt_CB_ACTIVATE, &cb2, 1);
    PtSetArg (&argt[n++], Pt_ARG_POS, &area.pos, 0);
    area.pos.y = 85;
    but = PtCreateWidget (PtButton, NULL, n, argt);

    /* Copy the default style to make the blue style.
       Replace the drawing member of the new style. */
    b = PtDupClassStyle (PtButton, NULL, "blue");
    PtSetClassStyleMethods (b,1,&meth);

    PtRealizeWidget (win);
    PtMainLoop();

    return EXIT_SUCCESS;
}
```

Photon hook

Photon provides a mechanism for you to allow a block of user code to be pulled in and executed during the initialization of Photon applications. This functionality is most frequently used to customize widget styles, allowing you to change the appearance and behavior of widgets without having to re-compile, re-link, or otherwise reconstruct executables.

The Photon hook can be used for many other things besides widget styles. For example, it can be used to log application usage information, or for more complicated situations such as remote control of an application.

PtInit() looks for a DLL, **`PtHook.so`**, in the search path, and executes the symbol for *PtHook()* in the DLL.

Multi-hook

You can use the **`pt_multihook.so`** DLL and rename it as **`PtHook.so`** to load one or several DLLs, pointed to by the **PHOTON_HOOK** environment variable. If **PHOTON_HOOK** points to a DLL, that DLL is loaded and its *PtHook()* function is executed. If **PHOTON_HOOK** points to a directory, each DLL in it is loaded and its *PtHook()* function executed.

The **`PtHook.so`** feature may introduce security holes if the DLL code is insecure. If you use the **`pt_multihook.so`**, you may wish to modify its code to add your own security features. See the code listing below.

Example **`PtHook.so`** - the pt_multihook:

```
#include <stdio.h>
#include <stdlib.h>

#include <dlfcn.h>
#include <dirent.h>

#include <photon/PtHook.h>

static int hookit( const char *hookname, PtHookData_t *data ) {
  void *handle;
  if ( ( handle = dlopen( hookname, 0 ) ) == NULL )
    return -1;
  else {
    PtHookF_t *hook;
    if  ( ( hook = (PtHookF_t*) dlsym( handle, "PtHook" ) ) == NULL
      ||  (*hook)( data ) == 0
        )
      dlclose( handle );
    return 0;
  } }
```

```
int PtHook( PtHookData_t *data ) {
  const char *hookname;
  DIR *dir;
  if  ( ( hookname = getenv( "PHOTON_HOOK" ) ) != NULL
    &&  hookit( hookname, data ) != 0
    &&  ( dir = opendir( hookname ) ) != NULL
      ) {
    struct dirent *de;
    while ( ( de = readdir( dir ) ) != NULL)
      if ( de->d_name[0] != '.' ) {
        char path[512];
        if ( (unsigned) snprintf( path, sizeof(path), "%s/%s",
             hookname, de->d_name ) < sizeof(path) )
          hookit( path, data );
        }
    closedir( dir );
    }
  return Pt_CONTINUE;
  }
```

The PtHook function, declared in **Photon/PtHook.h**, looks like this:

```
int PtHook( PtHookData_t *data );
```

PtHookData_t has at least these members:

int *size*	The size of the **PtHookData_t** structure.
int *version*	The version of the Photon library that loaded the DLL.

The function can return Pt_END to ensure the DLL is not unloaded by *PtInit()*, or return Pt_CONTINUE to ensure DLL is unloaded.

Setting widget styles using the Photon Hook

Here is a simple example of changing widget styles using the Photon Hook. The following code changes the fill for all buttons to blue, based on the previous widget style example.

To compile this code, use:

```
cc -shared button_sample.c -o PtHook.so
```

Place the **PtHook.so** in the search path to change the button style for all Photon applications. You can get the search path with **getconf _CS_LIBPATH**.

```
#include <Pt.h>

static void (*button_draw)(PtWidget_t *widget, PhTile_t const *damage ) = NULL;

void blue_draw (PtWidget_t *widget, PhTile_t *damage)
{

    /* This is the drawing function for the blue style.
       It draws a blue rectangle (without a label) for
       the widget. */

    PgSetFillColor( Pg_BLUE);
    PgDrawRect( PtWidgetExtent (widget,NULL),
                Pg_DRAW_FILL);
}


int
```

```
PtHook (void *data)
{
    PtStyleMethods_t button_meth        = { Pt_STYLE_DRAW, blue_draw };
    PtWidgetClassStyle_t        *button_style   = PtFindClassStyle( PtButton, NULL );

    button_draw = button_style->draw_f;
    PtSetClassStyleMethods( button_style, 1, &button_meth );
    return( Pt_END );
}
```

Chapter 13

Control Surfaces

In this chapter. . .

What's a control surface?

Control surfaces are geometrical regions within a widget that can position, size and draw themselves. Additionally, they can define their own behavior. They do all this via callbacks and event-handling flags that are supplied when the surface is created.

Control surfaces let you redefine the behavior for any area within a widget's drawable extent. Additionally, they can draw themselves as well as calculate their own geometry. Conceptually, they can be considered as lightweight "widgets within widgets."

For example, consider a scroll bar. You get different actions, depending on where you click on it: the arrow buttons step up and down; clicking in the trough pages up and down; dragging on the handle scrolls as you move. **`PtScrollbar`** is implemented as a single widget with several control surfaces on it.

You could also use control surfaces for:

- keyboard emulations, for example to add a Shift-lock to a Shift key
- panels of pushbuttons
- and so on.

It's important to note that control surfaces are a property of a widget; they require a widget in order to exist. However, a widget can possess any number of control surfaces, making it possible to implement a whole user interface using only one widget (say a **`PtWindow`**) at a fraction of the runtime data size (8% being a reasonable upper bound) as opposed to implementing the same UI using widgets.

Limitations

There are a few limitations to control surfaces:

- The widget library provides services to widgets that can't, for reasons of economy, be provided to control surfaces. For instance, widgets have the concept of opacity, which the library uses when drawing to reduce flicker. Control surfaces are simply drawn from the back to the front without any regard to opacity.
- Control surfaces can't contain other control surfaces, and don't include the concept of focus.
- Control surfaces are very raw elements and can provide only the behavior that you implement when you create them. It isn't difficult to implement a button as a control surface, but building **`PtMultitext`** as one would require more effort.

Binding actions to control surfaces

You can bind control surfaces to any of a widget's predefined actions or to user-defined actions.

The types of control surfaces are:

Regular surfaces	Let you define an event mask and callback function for the control surface.
Action surfaces	Let you automatically bind a control surface to one of a widget's predefined actions.

Referring to control surfaces

You can refer to a control surface via:

- a pointer to the control surface structure (**`PtSurface_t *`**).
- a numerical identifier (16-bit **`unsigned PtSurfaceId_t`**). This ID uniquely identifies a control surface within its associated widget. Valid values for the surface ID are in the range of 1 through 255, inclusive.

While the pointer method is more direct and therefore quicker, it's not as safe as the ID method. To understand why, consider how control surfaces are organized and stored in memory.

Unlike the widget hierarchy, which is implemented as a linked list, control surfaces are stored as an array of surface structures (**`PtSurface_t`**). The major reasons for storing them this way are:

- The array allows for quick traversal in both directions (which is a requirement, since drawing is handled from back to front and events are processed from front to back).
- The array reduces the memory requirement per surface. To satisfy the quick-traversal requirement, a doubly linked list would have to be used, adding significantly to the amount of memory required.
- You aren't likely to add or remove control surfaces very often, so using an array doesn't cause much of a penalty in performance.

As you physically move control surfaces around in the stacking order, their placement in the array changes, affecting their address in memory. In addition, as you add or remove control surfaces to or from a widget, the array needs to be reallocated, which also may cause the array itself to move around in memory. With all this possibility of memory movement, numerical identifiers are the only reliable way of locating a surface.

If you're pretty certain that a widget's surface configuration isn't going to change, then the pointer method is safe (and quicker, since the ID method needs to do a linear lookup in the surface array).

Control-surface API

The functions listed below are described in the Photon *Library Reference*.

Creating and destroying control surfaces

The following functions create and destroy control surfaces:

PtCreateActionSurface()	Create a control surface within a widget, bound to a widget action
PtCreateSurface()	Create a regular control surface within a widget
PtDestroyAllSurfaces()	Destroy all of a widget's control surfaces
PtDestroySurface()	Destroy a control surface
PtDestroySurfaceById()	Destroy the control surface with a given ID

Finding IDs for control surfaces

To find surface and action IDs, use these functions:

PtSurfaceActionId()	Get the action ID for a surface
PtSurfaceId()	Get the ID of a control surface

Calculating geometry for control surfaces

You must provide a function that calculates the control surface's geometry. Control surfaces are asked to calculate their geometry twice when the widget that owns them is asked to calculate its geometry:

- once before the widget's geometry calculation (which allows a widget to size itself according to the requirements of its surfaces if it cares — and some widgets do)
- once after (allowing surfaces to position and size themselves according to the size of the widget).

The *post* argument that's passed to the geometry function tells you which case is in progress.

A surface may also calculate its geometry based on the geometry of other surfaces. Using *PtCalcSurface()* or *PtCalcSurfaceById()*, you can ensure that the surface you're interested in has calculated its geometry prior to examining it.

The actual recording of the surface's geometry is simply a matter of directly modifying the surface's points array. Be sure you know how this array is organized before proceeding. This organization is detailed in the documentation for *PtCreateSurface()*.

These functions deal with a control surface's geometry:

PtCalcSurface()	Force a surface to calculate its geometry
PtCalcSurfaceByAction()	Force all surfaces associated with an action to calculate their geometry
PtCalcSurfaceById()	Force the control surface with a given ID to calculate its geometry
PtSurfaceCalcBoundingBox(), *PtSurfaceCalcBoundingBoxById()*	Calculate the bounding box for a control surface
PtSurfaceExtent(), *PtSurfaceExtentById()*	Calculate the extent of a control surface
PtSurfaceHit()	Find the control surface hit by a given point
PtSurfaceRect(), *PtSurfaceRectById()*	Get the bounding box of a control surface
PtSurfaceTestPoint()	Test whether or not a point is inside a control surface

Drawing control surfaces

Control surfaces are asked to draw themselves from back to front, after the widget itself has drawn. No clipping is done for you. If you want clipping, you have to implement the necessary logic to adjust the clipping list as surfaces are traversed, and then reinstate the clipping stack after the last surface is drawn. Otherwise, you'll get some unexpected results.

The following functions damage control surfaces so they'll be redrawn:

PtDamageSurface(), *PtDamageSurfaceById()*	Mark a surface as damaged so that it will be redrawn
PtDamageSurfaceByAction()	Damage all surfaces that are associated with an action

Activating control surfaces

This function activates a control surface:

PtCheckSurfaces()	Match an event with the control surfaces belonging to a widget

Enabling and disabling control surfaces

You can enable and disable control surfaces, like widgets:

PtDisableSurface(), PtDisableSurfaceById()
: Disable a control surface

PtDisableSurfaceByAction()
: Disable all control surfaces associated with an action

PtEnableSurface(), PtEnableSurfaceById()
: Enable a control surface

PtEnableSurfaceByAction()
: Enable all control surfaces associated with an action

PtSurfaceIsDisabled()
: Determine if a control surface is disabled

PtSurfaceIsEnabled()
: Determine if a control surface is enabled

Finding control surfaces

To find a control surface, use these functions:

PtFindSurface()	Find the control surface with a given ID
PtFindSurfaceByAction()	Find the control surface associated with a given action
PtWidgetActiveSurface()	Get a widget's currently active control surface

Hiding and showing control surfaces

You can hide and show control surfaces, too:

PtHideSurface(), PtHideSurfaceById()
: Hide a control surface

PtHideSurfaceByAction()
: Hide all control surfaces associated with an action

PtShowSurface(), PtShowSurfaceById()
: Show a hidden control surface

PtShowSurfaceByAction()
: Show all hidden control surfaces associated with an action

PtSurfaceIsHidden()
: Determine if a control surface is hidden

PtSurfaceIsShown()
: Determine if a control surface is shown

Ordering control surfaces

Like widgets, you can stack control surfaces:

PtInsertSurface(), *PtInsertSurfaceById()*
: Insert a control surface in front of or behind another

PtSurfaceBrotherBehind()
: Get the control surface behind a given one

PtSurfaceBrotherInFront()
: Get the control surface in front of a given one

PtSurfaceInBack()
: Get the backmost control surface belonging to a widget

PtSurfaceInFront()
: Get the frontmost control surface belonging to a widget

PtSurfaceToBack(), *PtSurfaceToBackById()*
: Move a control surface behind all other control surfaces belonging to a widget

PtSurfaceToFront(), *PtSurfaceToFrontById()*
: Move a control surface in front of all other control surfaces belonging to a widget

Storing user data with control surfaces

There's no callback data in the function associated with control surfaces; you can store user data with control surfaces by calling:

PtSurfaceAddData(), *PtSurfaceAddDataById()*
: Add data to a control surface

PtSurfaceGetData(), *PtSurfaceGetDataById()*
: Get data associated with a control surface

PtSurfaceRemoveData(), *PtSurfaceRemoveDataById()*
: Remove data from a control surface

Example

Here's a program that creates some control surfaces:

```
#include <Pt.h>

/* This is the function that's called when an event occurs
   for our rectangular control surface. When a user clicks
   on this surface, we'll tally up the clicks and print how
   many have occurred. */

static int rect_surface_callback( PtWidget_t *widget,
                                  PtSurface_t *surface,
                                  PhEvent_t *event)
{
 static int rclicks = 1;
 printf("Rectangle clicks:  %d\n", rclicks++);
 return(Pt_END);
}

/* This is the function which draws the contents of our
   rectangular control surface. This is a very simple
   example; it draws a red rectangle. */

static void rect_surface_draw( PtWidget_t *widget,
                               PtSurface_t *surface,
                               PhTile_t *damage)
{
 PgSetFillColor(Pg_RED);
 PgDrawRect(PtSurfaceRect(surface, NULL), Pg_DRAW_FILL);
}

/* This is the function keeps the size of the control
   surface in sync with the size of the widget.
   PtWidgetExtent() returns a rect containing the current size
   of the widget.

   PtSurfaceRect() is a macro; this means that you have direct
   access to the data within your control surface. You do not
   need to call any functions to change its size. Change the
   data directly. */

static void rect_surface_calc( PtWidget_t *widget,
                               PtSurface_t *surface,
                               uint8_t post)
{
 /* Do this only after widget has extented. */
 if(post)
 {
  /* The rect occupies the top left quadrant of the window. */
  PhRect_t *extent;
  PhRect_t *srect;

  extent = PtWidgetExtent(widget, NULL);
  srect = PtSurfaceRect(surface, NULL);

  srect->ul = extent->ul;
  srect->lr.x = (extent->ul.x + extent->lr.x) / 2;
  srect->lr.y = (extent->ul.y + extent->lr.y) / 2;
 }
}

/* This is the function that's called when an event occurs
```

```
   for our elliptical control surface. When a user clicks on
   this surface, we'll tally up the clicks and print how
   many have occurred. */

static int ell_surface_callback( PtWidget_t *widget,
                                 PtSurface_t *surface,
                                 PhEvent_t *event)
{
 static int eclicks = 1;
 printf("Ellipse clicks:  %d\n", eclicks++);
 return(Pt_END);
}

/* This is the function that draws the contents of our
   elliptical control surface. This is a very simple
   example; it draws a green ellipse. */

static void ell_surface_draw( PtWidget_t *widget,
                              PtSurface_t *surface,
                              PhTile_t *damage)
{
 PhRect_t *s = PtSurfaceRect(surface, NULL);
 PgSetFillColor(Pg_GREEN);
 PgDrawEllipse(&(s->ul), &(s->lr),
              Pg_DRAW_FILL | Pg_EXTENT_BASED);
}

/* This is our main function. We create a window, initialize
   our application with the Photon server and create two
   control surfaces.

   Notice that the second surface doesn't supply the last
   parameter, the extent calculation function. This isn't
   needed because of the fifth parameter, the height and
   width stored in a point structure. This is a pointer
   to the actual point structure within the window widget.
   Thus, if the window's extent changes, changing the
   extent point structure, the control surface is
   automatically updated with the new values! */

int main(int argc, char **argv)
{
 PtArg_t args[1];
 PtWidget_t *window;
 const PhDim_t dim = { 200, 200 };

 PtSetArg(&args[0], Pt_ARG_DIM, &dim, 0);
 window = PtAppInit(NULL, &argc, argv, 1, args);

 /* Create a rectangular control surface. */
 PtCreateSurface( window, 0, 0, Pt_SURFACE_RECT, NULL,
                  Ph_EV_BUT_PRESS, rect_surface_callback,
                  rect_surface_draw, rect_surface_calc);

 /* Create an elliptical control surface to fill the window. */
 PtCreateSurface( window, 0, 0, Pt_SURFACE_ELLIPSE,
                  (PhPoint_t*)PtWidgetExtent(window, NULL),
                  Ph_EV_BUT_PRESS, ell_surface_callback,
                  ell_surface_draw, NULL);

 PtRealizeWidget(window);
 PtMainLoop();
 return(EXIT_SUCCESS);
}
```

Chapter 14

Accessing PhAB Modules from Code

In this chapter. . .

You can access any module directly from your application code by creating an *internal link* to that module.

An internal link is like a link callback—it lets you specify the module type, a setup function, and, where appropriate, a location. But unlike a link callback, which is always associated directly with a widget callback, an internal link has no association with any widget. Instead, PhAB will generate a manifest that you use in your application code to specify which internal link you want to use. PhAB provides several functions to help you use internal links (discussed below).

You can use internal links to:

- Create a PhAB module within application code.

  Using a link callback, you can directly link a widget to a PhAB application module. But sometimes you need to create the module from your application code instead. To do that, use an internal link.

  Here are some common situations where you should use an internal link to create a module:

  - when your application can display one of two different modules based on some condition inside the application
  - when you need to control the parentage of a module instead of using the PhAB defaults (by default a new module is a child of the base window)
  - when you want to display a menu when the user presses the right mouse button.

- Access and display picture modules.

  You use picture modules primarily to replace the contents of existing container widgets, such as **`PtWindow`** or **`PtPanelGroup`**.

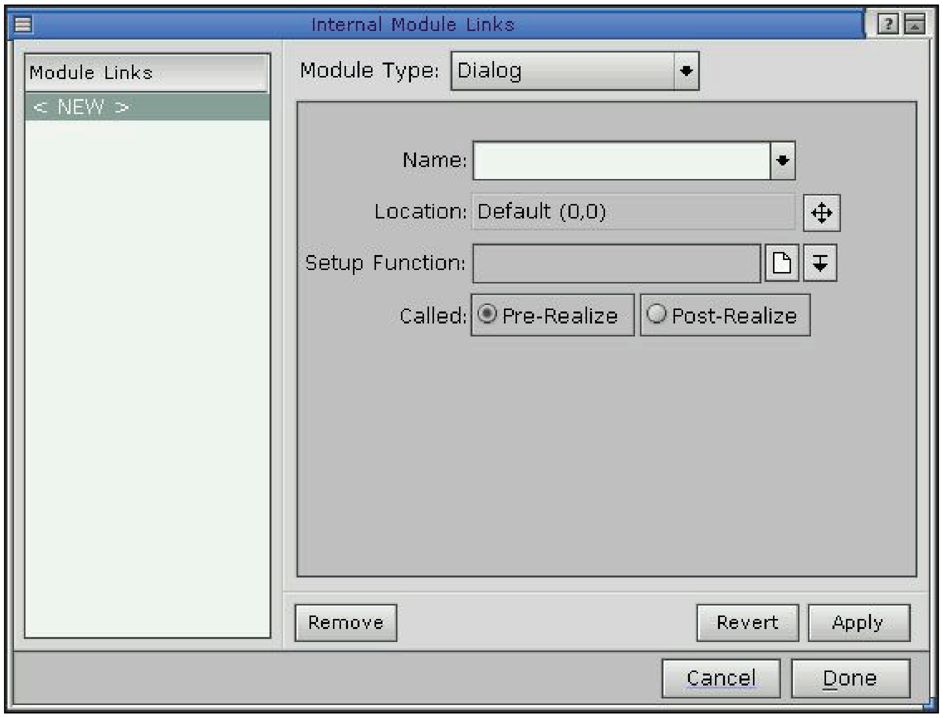
  Note that when you create a picture module using *ApCreateModule()*, you must specify the parent container widget.

- Open widget databases. For more information, see “Using widget databases.”

Creating internal links

To create an internal link:

1. Choose Internal Links from the Application menu or press F4. You’ll see the Internal Module Links dialog:



Internal Module Links dialog.

2 Click on the <NEW> option if it isn't already selected.

3 Choose the type of module you want.

4 Fill in the fields in the Module Link Info section — see below.

5 Click on Apply, then click on Done. If the module you specified in the Name field doesn't exist, PhAB will ask whether it should create that module.

You can create only one internal link per module.

The fields in the Internal Module Links dialog include:

- Name—Contains the name of the module. To select from a list of existing modules, click on the icon next to this field.
- Location—Determines where the module will appear; see "Positioning a module" in the Working with Modules chapter.
- Setup Function—Specifies the function that will be called when the module is realized (optional). To edit the function, click on the icon next to this field.

  For more information, see "Module setup functions" in the Working with Code chapter.

- Called—Determines whether the setup function is called before the module is realized, after the module is realized, or both.
- Apply—Applies any changes.
- Reset—Restores the internal link information to its original state.
- Remove—Deletes the selected internal link from the Module Links list.

Using internal links in your code

Manifests

For every internal link defined in your application, PhAB generates a manifest so you can identify and access the link.

Since PhAB derives the manifest name from the module name, each module can have only one internal link. This may appear limiting, but PhAB provides module-related functions (see below) that let you customize a module's setup function and location from within your application code.

To create the manifest name, PhAB takes the module's name and adds ABM_ as a prefix. So, for example, if you create an internal link to a module named **mydialog**, PhAB creates the manifest ABM_mydialog.

Internal-link functions

The manifest is used by the following PhAB API functions:

ApCreateModule()	Lets you manually create modules designed within PhAB. A module created with this function behaves exactly as if it were linked directly with a link callback. For example, if you define a location and a setup function for the internal link, the module will appear at that location and the setup function will be called. Furthermore, widget callbacks, hotkeys, and so on will become active.
ApModuleFunction()	Lets you change the setup function associated with an internal link.
ApModuleLocation()	Lets you change the display location associated with an internal link.
ApModuleParent()	Lets you change the parent of a window or dialog module associated with an internal link. This function applies only to internal links for window and dialog modules.
ApOpenDBase()	Lets you open the module associated with an internal link as a widget database.

For more info on the above functions, see the Photon *Library Reference*.

Example — displaying a menu

Here's how you can display a menu module when the user presses the right mouse button while pointing at a widget:

1. In PhAB, create the menu module. Give it a name, such as **`my_menu`**.
2. Create an internal link to the menu module, as described above. For a popup menu, you'll usually want the module to be positioned relative to the widget or relative to the pointer.
3. Select the widget to be associated with the menu. Make sure it has Pt_MENUABLE set and Pt_ALL_BUTTONS cleared in its *Pt_ARG_FLAGS*.
4. Generate the code for your application. PhAB creates a manifest for the internal link. In this example, it's called ABM_my_menu.
5. Every widget that's a descendant of **`PtBasic`** has a *Pt_CB_MENU* resource that's a list of callbacks invoked when you press the right mouse button while pointing at the widget. Edit this resource, and create a callback function like this:

   ```
   int
   text_menu_cb( PtWidget_t *widget, ApInfo_t *apinfo,
                 PtCallbackInfo_t *cbinfo )
       {

       /* eliminate 'unreferenced' warnings */
       widget = widget, apinfo = apinfo, cbinfo = cbinfo;

       ApCreateModule (ABM_my_menu, widget, cbinfo);

       return( Pt_CONTINUE );

       }
   ```

   The *widget* passed to *ApCreateModule()* is used if the menu is to be positioned relative to the widget; the *cbinfo* argument is used if the menu is to be positioned relative to the pointer.
6. Compile, link, and run your application. When you press the right mouse button over the widget, your menu should appear.

Using widget databases

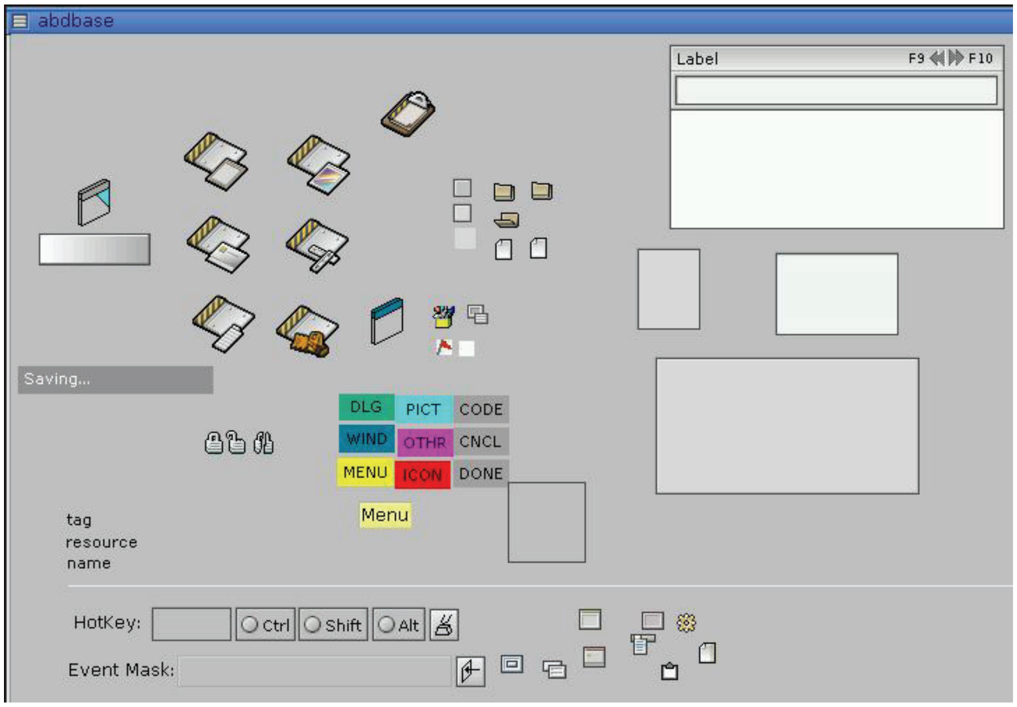
Picture modules have two purposes:

- to let an application replace the contents of any container widget
- to serve as widget databases.

If you plan to use a widget several times within your application, a widget database lets you design the widget just once. It also saves you from a lot of coding. All you have to do is preset the widget's resources and then, using PhAB's widget-database

API functions, create a copy of the widget wherever you'd normally create the widget within your code.

Here's an example of a widget database—it's part of the one that PhAB uses for its own interface:



Widget database used for PhAB's interface.

Creating a database

To create a widget database:

1 Create a picture module within your application.

2 Create an internal link to the picture module.

3 Create the widgets that you'll need to access in your application code.

For example, let's say you need to create a certain icon many times in your application. By creating the icon inside the picture module, you can create as many copies of the icon as you need at run time.

Preattaching callbacks

Besides being able to preset all of a widget's resources in the database module, you can also preattach its callbacks. When you create the widget dynamically, any callbacks you attached will also be created.

By presetting the resources and callbacks of a database widget, you can easily reduce the code required to dynamically create the widget to a single line.

Preattached callbacks work only with modules and functions that are part of your executable. If your application opens an external file as a widget database, the PhAB library won't be able to find the code to attach to the callback.

Assigning unique instance names

Assign each widget in a widget database an instance name—this lets you refer to the widgets when using database-related API functions.

Creating a dynamic database

You can also create a widget database that you can change dynamically. To do this, open an external widget database—that is, one that isn't bound into your executable—with *ApOpenDBaseFile()* instead of *ApOpenDBase()*. *ApOpenDBaseFile()* lets you access a module file directly and open it as a database.

Once you've opened the module file, you can copy the widgets from that file to your application's internal database and save the resulting database to a new file that you can reopen later.

Widget-database functions

PhAB provides several support functions to let you open a widget database and copy its widgets into modules—you can copy the widgets as often as needed. PhAB also provides convenience functions to let you copy widgets between databases, create widgets, delete widgets, and save widget databases.

ApOpenDBase() *ApCloseDBase()*	These let you open and close a widget database. To ensure that the database is always available, you typically use *ApOpenDBase()* in the application's initialization function.
ApOpenDBaseFile() *ApSaveDBaseFile()*	These let you open and save external module files as databases within your application.
ApAddClass()	This function lets you indicate which widget classes you're likely to encounter when you call *ApOpenDBaseFile()*. When you link your application, only those widgets it needs are linked into it. If you access widgets that aren't in your application because they're in an external database, you must add them to your internal class table so that they can be linked in at compile time.

ApCreateDBWidget()
ApCreateDBWidgetFamily()
ApCreateWidget()
ApCreateWidgetFamily()

These create widgets from the widget database. *ApCreateWidget()* and *ApCreateDBWidget()* create a single widget only, regardless of the widget's class.

For a noncontainer-class widget, *ApCreateWidgetFamily()* and *ApCreateDBWidgetFamily()* create a single widget; for a container-class widget, they create all the widgets within the container.

These functions differ in the parent used for the widgets:

- *ApCreateDBWidget()* and *ApCreateDBWidgetFamily()* include a *parent* argument; if this is NULL, the widget has no parent.
- *ApCreateWidget()* and *ApCreateWidgetFamily()* put the new widget(s) in the current parent. To make sure the correct widget is the current parent, call *PtSetParentWidget()* before calling either of these functions.

Don't use the manifests generated for the widget database's picture module. Instead, use the widget pointers returned by *ApCreateWidget()* or *ApCreateDBWidget()*.

ApCopyDBWidget() Lets you copy a widget from one widget database to another. Typically, you use this only when you're dynamically creating and saving widget databases within your application.

ApDeleteDBWidget()

Deletes a widget from a widget database.

ApGetDBWidgetInfo()

Gets information about a widget in a widget database, including its name, class, parent, and level in the hierarchy.

ApGetImageRes() Pull out image-resource data from a widget and use this data to set resources of a widget already displayed in your application. This function lets you achieve very basic animation.

If you use a widget database to create widgets that have **`PhImage_t`** data attached to them, don't close the database with *ApCloseDBase()* until after those widgets are destroyed. (Closing the database frees the memory used by the image.) If you must close the database, make sure to copy the image data within your application code and to reset the image data resource to point to your new copy.

	For more information, see the "Animation" section in the chapter on Drawing.
ApGetTextRes()	This lets you extract text strings from a widget database. It's useful for multilingual applications, as the text is automatically translated if the language support is enabled. For more information, see the International Language Support chapter.
ApRemoveClass()	Remove a widget class. If you've loaded a DLL that defines widget classes, you should remove them before unloading the DLL. For more information, see "Making a DLL out of a PhAB application" in the Generating, Compiling, and Running Code chapter.

For more info on widget database functions, see the Photon *Library Reference*.

Chapter 15

International Language Support

In this chapter. . .

PhAB has builtin support for applications that need to be translated into other languages.

By keeping a few design considerations in mind, and then following a few simple steps, your application can very easily be translated into other languages without the need to recompile or rebuild your application:

1. PhAB generates a database of all the text strings used by your application.
2. This text database is used by PhAB's language editor to allow you to translate each text string into another language.
3. The translated text strings are saved in a translation file, and are shipped with your application.
4. To run your application in another language, simply set an environment variable before starting the application. When PhAB's API builds the application windows, dialogs and other modules, it replaces the text strings with the new translations.

It's that simple.

Application design considerations

This section provides a few design considerations to assist you in creating a language-independent application. You should keep these ideas in mind as you are designing and implementing your application, since modifying the application after it's complete is more difficult.

Size of text-based widgets

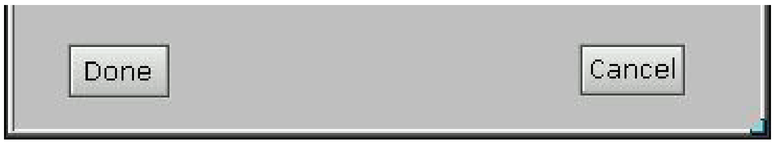
Typically, when you design an application, you lay out the window using widgets that have the default application text already preset. For example, if you had a Done button at the bottom of a dialog window, the button itself would be only large enough to hold the text string "Done". You would also place the Done button based on its current size. This works well in an application that doesn't require translation, but causes many problems for a language-independent application. What would happen if the translated text were 12 characters instead of the default of 4 characters?

- The translated button could become much larger. In this case, the button may be so wide that it writes on top of other widgets in the window. This would cause the application to look ugly or poorly designed.

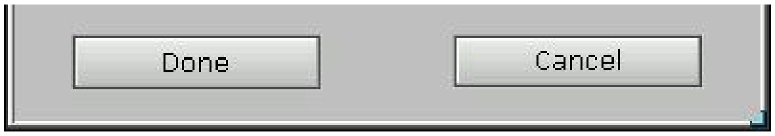
  or

- The text could be truncated within the default size. In this case, the translated text would be unreadable, and the user wouldn't know what the button does.

For example, these buttons are too small to accommodate translated text:

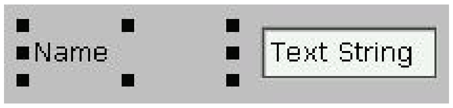


The solution is simple. Make the button larger to accommodate longer translated text strings. Here's an example:

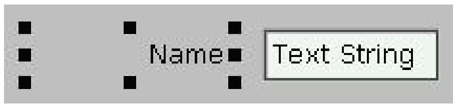


Justification

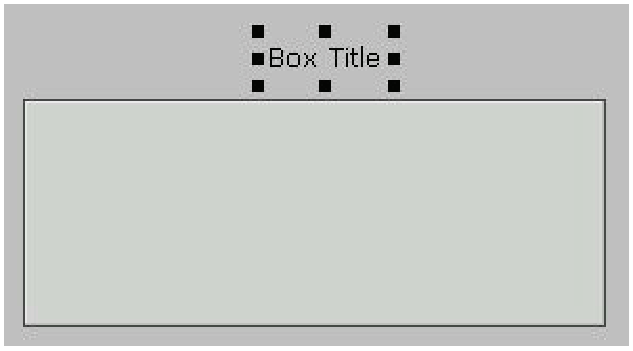
In addition to making text-based widgets wider to accommodate translated text, you should give some thought to the justification of text, based on the widget's usage. For example, in a simple text entry field, it's quite common to place a label to the left side of the field. If you make the label wider to allow for translation, the label itself moves to the far left:



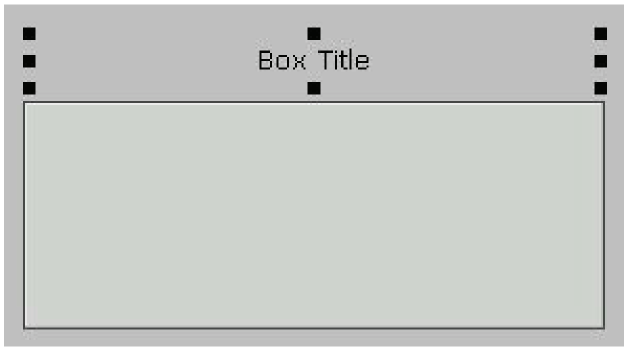
This problem is easily solved by setting the label's horizontal alignment to be right-justified. This allows for longer translated text strings, and still keeps a tight alignment with the text entry field:



Another common labeling method is to place a label centered above or within the border of a box. Usually the text is centered by placing it in the desired position based on its current text:



When the text is later translated, it's either too short or too long, and the box label looks lopsided. The simple solution is to make the box title much wider than necessary, and set the horizontal alignment to be centered.



There are probably many other cases similar to this but the important point is to think about how the translated text will effect the look of the application. A lot of aesthetics can be maintained simply by making text-based widgets wider and setting an appropriate justification.

Font height

The fonts for some languages, such as Japanese or Chinese, are only readable at large point sizes. For these fonts, the minimum size may be 14 points or even larger. If you've designed your entire application using a 10-point Helvetica font, you'll have lots of problems when *all* your text-based widgets are stretched 4 or more pixels taller to accommodate the larger fonts. If your application needs to be translated to other languages, look into the font requirements before you begin, and use this minimum font size in the default language built into the application.

If you really want to use the smaller font sizes for your default application text, you can borrow a tip from the previous section. You can make the height of widget larger and set the vertical alignment to center. However, this may not work well for text input fields, and you should keep this consideration in mind.

Hard-coded strings

Another major area for consideration is with informational, warning, error or any textual messages that are displayed in popup dialog windows or other points within the application. Examples include calls to *PtAlert()*, *PtNotice()*, and *PtPrompt()*. The most common way to handle text messages is to embed the text strings in the application code. For example:

```
char *btns[] = { "&Yes", "&No", "&Cancel" };

answer = PtAlert( base_wgt, NULL, NULL, NULL,
                  "File has changed. Save it?",
                  NULL, 3, btns, NULL, 1, 3,
                  Pt_MODAL );
```

While this is quick to code, it's impossible to translate without rewriting the application code, recompiling, and so on. Essentially, you need to create a complete new version of the application for each language supported.

A much better method is to take advantage of PhAB's message databases. Using a message database, you can put all your text messages into a single file and give each message a unique name. To retrieve the text at runtime, call *ApGetMessage()*.

```
char *btns[3];
ApMsgDBase_t *textdb;
ApLoadMessageDB(textdb, "MyMessageDb");

btns[0] = ApGetMessage( textdb, "msgyes" );
btns[1] = ApGetMessage( textdb, "msgno" );
btns[2] = ApGetMessage( textdb, "msgcancel" );
answer = PtAlert( base_wgt, NULL, NULL, NULL,
                  ApGetMessage( textdb, "file_save_msg" ),
                  NULL, 3, btns, NULL, 1, 3,
                  Pt_MODAL );

ApCloseMessageDB(textdb);
```

An advantage of message databases is that they can be shared between applications. If several applications use the same types of strings, this can make translation easier.

See the section Message Databases below for more information.

Another method of storing strings is to use PhAB's widget databases. You can put all your text messages in a single (or multiple) widget database and give each message a unique name. To retrieve the text at runtime, call *ApGetTextRes()* (see the Photon *Library Reference* for details). In the *PtAlert()* example above, it would become:

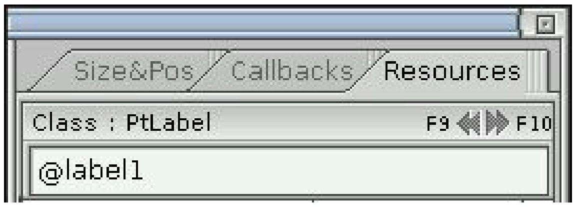
```
char *btns[3];

btns[0] = ApGetTextRes( textdb, "@msgyes" );
btns[1] = ApGetTextRes( textdb, "@msgno" );
btns[2] = ApGetTextRes( textdb, "@msgcancel" );
answer = PtAlert( base_wgt, NULL, NULL, NULL,
                  ApGetTextRes( textdb, "@msg001"),
                  NULL, 3, btns, NULL, 1, 3,
                  Pt_MODAL );
```

This method allows the application to have no predefined text-based messages within it, and it can be easily translated. In addition, because the text strings are put into a widget database, PhAB automatically takes care of including the message texts when it generates the application's text string database. This may be more convenient than using a separate message database, especially if you only have a few strings, they're only used in one application, and the application has a widget database anyway.

Use of @ in instance names

By default, PhAB ignores widgets that have no instance name or have the instance set to the class name. This means if you place a label within a window and change the text to something appropriate, PhAB skips this widget when it generates code for your application. This is because PhAB assumes the label is constant and the application doesn't require access to it. However, when it comes to translating your application to another language, this label becomes very important.

To differentiate between widgets that are important for translation but not for code generation, PhAB recognizes a special character when placed in the first position of the instance name. This special character is the `@` character. This means you can give a label the instance name of **`@label1`**, and PhAB will recognize this label when generating the text language database, but skip over it when generating code.



This sounds fine, except PhAB also requires that all instance names be unique. This rule must be adhered to so that PhAB knows which text string to replace at run time. Unfortunately, dreaming up potentially hundreds of unique instance names that you don't really care about can be a lot of work. To simplify this task, PhAB lets you specify a single `@` character for the instance name, and PhAB appends an internal sequence number to the end. This eliminates the need to keep track of all constant text strings that require instance names just for translation.

If you want to group translation text strings (say, by module), you can give them all the same instance name, and PhAB will append a sequence number to make the name unique. For example, if you assign the name **`@base`** to several widgets, PhAB generates **`@base`**, **`@base0`**, **`@base1`**, ... as instance names.

Bilingual applications

Sometimes it's necessary to design an application to be bilingual. This means two different languages are displayed in every text string. While this can be done, it's usually difficult for the user to read and understand.

PhAB allows you to use another approach. You can create the application in one language and provide the ability to flip to the other language within application control. This is done via a PhAB API function named *ApSetTranslation()*. This function (which is described in the Photon *Library Reference*) changes the current translation file for the application immediately, such that all *future* dialogs, windows, and so on are drawn using the new translation file.

Any existing modules and widgets aren't translated, only new ones. If you want immediate feedback, you need to recreate the modules. This is easy for dialogs, but more difficult for the base window; remember that destroying the base window exits the application. One way to translate the contents of the base window is to put them in a picture module, which can be recreated.

Common strings

If you have several applications to translate, you can reduce the work by sharing the common text strings and translating them separately. To do this:

1 Create a message database populated with the common text strings.

2. Use the PhAB Language Editor to translate the strings.
3. Once the database is created and translated, you can open it and use the strings in other applications using *ApLoadMessageDB()* and *ApGetMessage()*.

Generating a language database

This is the easy part. The most important aspect to this step is to know *when* to generate the text string database. Ideally, you want to do this when all application development is complete. This is because the run-time translation mechanism is hinged on the widget's instance name. If you generate your database mid-way through the development and do the translations, it's quite likely that a lot of widgets will be changed or deleted, and translations may be deleted or the time wasted.

One exception to this would be bilingual applications. In this case, you might want to generate and translate the application continuously so that the application's translations can be tested throughout the development.

To generate an application's language database:

1. Select the **Project→Language Editor→Generate Language Database**.
2. A progress dialog appears, and the language database is generated.
3. Click Done.

The Languages item in the Application menu is disabled if you haven't saved your application for the first time and given it a name.

The database has now been generated and is ready for use with the PhAB Language Editor. The name of the database is *app*`.ldb`, where *app* is the name of the executable file for the application (which is typically the same as the name of the application, unless you've used the Save As command to rename the application). The language database is placed in the application's directory (where the **`abapp.dfn`** file is found).

Message databases

A message database is a file that contains textual messages. Each message is identified by a tag name.

To load a message database, call *ApLoadMessageDB()*. This function does the usual file search based on the **ABLPATH** environment variable and the current language:

- If no language is defined, the message database is loaded. It must have a name of *name*`.mdb`.
- If a language is defined, the function looks for a translation file called *name.language*. Translation files can be created using the PhAB translation editor — it can handle message databases.

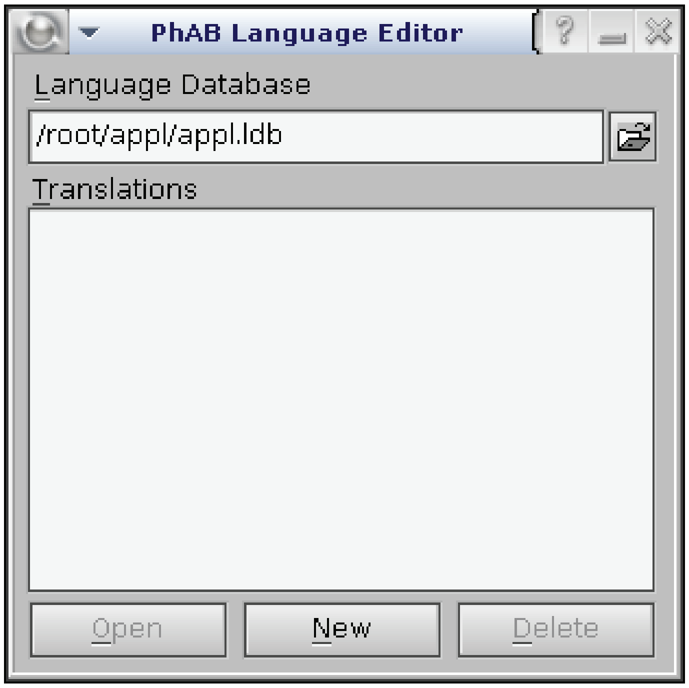
To retrieve a message, given its tag, call *ApGetMessage()*.

To close the message database, call *ApCloseMessageDB()*.

You can create message databases using the PhAB message database utility **`phabmsg`**, located in the PhAB application directory. This utility allows you to create new message databases, and edit existing message databases.

Language editor

After the database has been generated, you can use PhAB's Language Editor to translate the default text strings to another language. The Language Editor is designed to work both as a stand-alone application that you can distribute with your application, or as an integrated part of PhAB itself.



PhAB Language Editor.

Starting the Language Editor within PhAB

When you are developing an application within PhAB, you can run the Language Editor using the current application's language database quite easily:

- Select **Project→Language Editor→Run Language Editor**.

This starts the Language Editor using the current application's language database. At this point, you can proceed to create a new translation file or edit an existing one.

Starting the Language Editor as a stand-alone application

If you plan to allow your application to be translated at a customer site, you'll need to include the following files with your application:

- **/usr/photon/appbuilder/phablang**
- **/usr/photon/appbuilder/languages.def**

The **languages.def** file must be in the same directory as the **phablang** editor.

- your application's language database file, *xxx*.**ldb**

To start at the client site, you can:

- type **/usr/photon/appbuilder/phablang &**

  or
- create an entry in the Desktop Manager to run **/usr/photon/appbuilder/phablang** (assuming the customer is running the full desktop environment).

Once **phablang** is started:

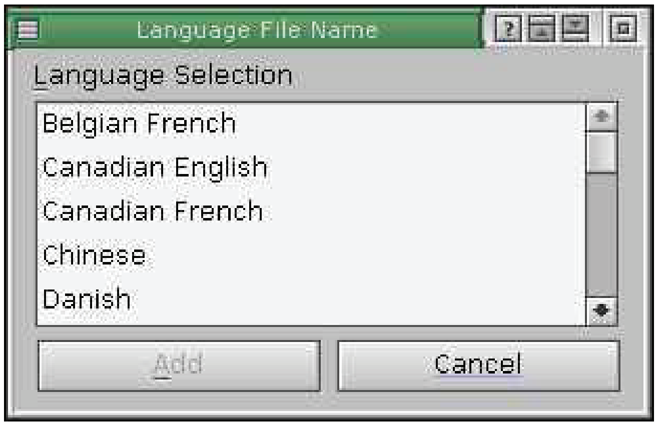
1. Click on the Open Folder icon

   to bring up the file selector.
2. Using the file selector, find the application's *xxx*.**ldb** file.
3. Open the *xxx*.**ldb** file.

Creating a new translation file

To create a new translation file:

1. Click on the New button located at the bottom of the window. The Language Selection dialog is displayed:



Language Selection dialog.

2 Choose the desired language from the list of supported language file types.

3 Click on Add.

4 At this point you're asked to reconfirm your selection. Click on Yes.

The Language Selection dialog closes, and you should now see the newly created translation file in the list of available translations.

Editing an existing translation file

To edit a translation file in the Translations list:

1 Click on the desired language from the list.

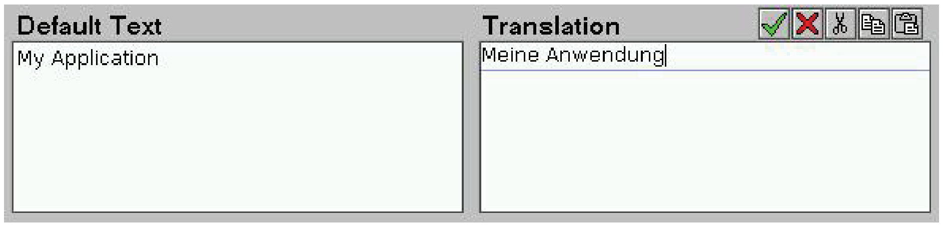
2 Click on the Open button.

The Text Translation Editor dialog appears. This editor displays all the text strings available for translation in the current language database.

Translating the text

To translate a text string:

1 Click on the text string you want to translate. The selected text string is displayed in the text areas at the bottom of the window:



My Application

Meine Anwendung

Default Text	the default text bound into the application

Translation	the current translation (if any) for the text string. This is the area you use to type in the new translation.

- If you need to type characters that don't appear on your keyboard, you can use the compose sequences listed in "Photon compose sequences" in the Unicode Multilingual Support appendix.
- You don't have to translate every string. If a translation isn't available, the default text is used.

You can use the cut, copy, and paste buttons that are above the Translation area when editing the translations.

2. Once you change the translated string, a green check mark and red X appear above the Translation area.
3. Click on the green check mark to accept your changes (the shortcut is Ctrl-Enter).
4. Click on the red X to cancel your changes.

Repeat the above steps for all the text strings you need to translate. When you're finished, click on the Save & Close button.

Hotkeys

One problem with translating an application is that the hotkey assignments no longer match up if the translated string doesn't include the accelerator key value. For this reason, PhAB adds the accelerator key strings to the language database too.

When translating the text string, the translator can also change the accelerator key. If the key used in the hotkey isn't a function key (i.e. the key code is less than **`0xF000`**), PhAB automatically changes the hotkey to match the accelerator key.

For example, suppose your application has a button labeled **`Cancel`**. You'd set the button's *Pt_ARG_ACCEL_KEY* to be **`C`**, and arrange for Alt-C to invoke *Pt_CB_HOTKEY*.

When you generate the language database, you'll find that it includes the button's label and its accelerator key. If you translate the application into French, the button's label would become **`Annuler`**, so the hotkey Alt-C is no longer appropriate. Just translate the button's *Pt_ARG_ACCEL_KEY* to be **`A`**, and the hotkey automatically becomes Alt-A when you run the application in French.

You'll need to make sure there are no duplicate accelerator keys. If it does happen by accident, only the first key defined is accepted.

Help resources

If you use the Photon Helpviewer for your application help and you plan on providing multiple language help files for your application, the translator can also translate the help topic paths to point to the correct positions within the corresponding help files.

Translation functions

You can build a custom language editor if the default one doesn't meet your needs. You'll find these functions (described in the Photon *Library Reference*) useful:

AlClearTranslation()	Clear all the translations in a language or message database
AlCloseDBase()	Close a language or message database
AlGetEntry()	Get an entry from a language or message database
AlGetSize()	Get the number of records in a language or message database
AlOpenDBase()	Load a language or message database
AlReadTranslation()	Read a translation file into a database
AlSaveTranslation()	Save translations from a language or message database
AlSetEntry()	Set the translated string for a database entry

You can use these functions to create your own language editor, or to convert a language database to a different file format (for example, so you can send the file to a non-Photon or non-QNX system for translation).

Running your application

After the language database is fully translated, the last step is to run the application.

When you create the translation files, they're placed in the same directory as your application's **`abapp.dfn`** file. You can think of these as the working versions of the files. When you run your application from PhAB, these are the versions you'll use.

When you run your application outside of PhAB, it looks for the translation files as follows:

1. In the directories listed in the **ABLPATH** environment variable, if defined. This list takes the form:

   dir`:`*dir*`:`*dir*`:`*dir*

   Unlike the **PATH** environment variable, the current directory must be indicated by a period, not a space. A space indicates the directory where the executable is.

2 In the same directory as the executable, if the **ABLPATH** environment variable isn't defined.

You can think of these as the production versions of the translation files.

In order for the PhAB API to know which translation file you want to use, you must set the **ABLANG** environment variable to one of the values below:

Language:	**Value:**
Belgian French	`fr_BE`
Canadian English	`en_CA`
Canadian French	`fr_CA`
Chinese	`zh_CN`
Danish	`da_DK`
Dutch	`nl_NL`
French	`fr_FR`
German	`de_DE`
Italian	`it_IT`
Japanese	`ja_JP`
Korean (North)	`ko_KP`
Korean (South)	`ko_KR`
Norwegian	`no_NO`
Polish	`pl_PL`
Portuguese	`pt_PT`
Slovak	`sk_SK`
Spanish	`es_ES`
Swedish	`se_SE`
Swiss French	`fr_CH`
Swiss German	`de_CH`
UK English	`en_GB`
USA English	`en_US`

This list is current at the time this document was written, but may have since been updated. For the latest version, see the file:

`/usr/photon/appbuilder/languages.def`

For example, to run an application in German (as spoken in Germany), you would do the following:

```
$ export ABLANG=de_DE
$ myapplication
```

The application looks for the best match. For example, if the language extension specified is **`fr_CA`**, the search is as follows:

1. Exact match (e.g. **`fr_CA`**).
2. General match (e.g. **`fr`**).
3. Wildcard match (e.g. **`fr*`**).

If no translation is found, the original text in the application is used.

The **`export`** command could be put in the user's login profile so that the application will run in each user's preferred language.

Distributing your application

When you ship your application to the customer site, you must make sure to include the translation files in your distribution list. For example, if your application is named **`myapp`**, and you have translation files for French and German, you would need to include the **`myapp.fr_FR`** and **`myapp.de_DE`** files with the application. These files must be located:

- in the directories listed in the **ABLPATH** environment variable, if defined. This list takes the form:

  dir:*dir*:*dir*:*dir*

  Unlike the **PATH** environment variable, the current directory must be indicated by a period, not a space. A space indicates the directory where the executable is.

- in the same directory as the executable, if **ABLPATH** isn't defined

If you want each customer to be able to translate the application, you'll also need to distribute:

- the language editor (**`phablang`**), which can be placed in the **`/usr/bin/photon`** directory
- the language definition file (**`languages.def`**), which must be installed in the same directory as the editor

- the application's language database (`myapp.ldb`)

The language database and the translation files that the customer creates should be in:

- one of the directories listed in the **ABLPATH** environment variable, if defined
- the same directory as the executable, if **ABLPATH** isn't defined

Chapter 16
Context-Sensitive Help

In this chapter. . .

For information about creating help files for use with the helpviewer, see “Creating Help Files” in the helpviewer topic in the *QNX Neutrino Utilities Reference*.

Referring to help topics

The Helpviewer understands two distinct ways of specifying the location of the HTML help text to be displayed:

- Universal Resource Locator (URL)
- topic path

Universal Resource Locator (URL)

A URL specifies the filesystem path of the help-text file. It specifies this path in the standard HTML way, except that all files must reside on the local network. Here’s a sample URL:

```
$QNX_TARGET/usr/help/product/photon/prog_guide/window_mgmt.html
```

URLs are case-sensitive. These URLs are restricted in scope to the help files; they can’t be used to access the web.

Topic path

A topic path is a group of concatenated topic titles that are defined in the current topic tree. For example, here’s the equivalent topic path to the above URL:

```
/Photon microGUI/Programmer’s Guide/Window Management
```

For the Helpviewer, the topic path is case-insensitive (unlike other HTML browsers) and may contain the wildcard characters `*` or `?`, where `*` matches a string and `?` matches a character. The first matching topic is selected.

A topic tree used by the Helpviewer must have at least three hierarchical levels: the top level is known as the bookshelf, the second level as the book set, and the third level as the book. A book may contain further levels of chapters and sections.

Entries in a bookshelf or book set should not contain any HTML, only `.toc` entries for the next level; help text should only be found in books.

Connecting help to widgets

You can display help information for a widget in the Helpviewer or in a help balloon. You can even use both methods in an application. For example, you could use a balloon for short explanations and the Helpviewer for more detailed help.

No matter which method you choose, you need to do the following in each of your application’s windows:

1. Set Ph_WM_HELP in the Flags: Managed (*Pt_ARG_WINDOW_MANAGED_FLAGS*) resource.
2. Set Ph_WM_RENDER_HELP in the Flags: Render (*Pt_ARG_WINDOW_RENDER_FLAGS*) resource. This will add a ? icon to the window frame. The user can click on it, and then click on a widget, and the help information will be displayed.

   If using the ? icon isn't suitable for your application, see "Help without the ? icon" later in this chapter.

For more information, see the Window Management chapter.

Displaying help in the Helpviewer

To use the Helpviewer to display help information for a widget, do the following:

1. Optionally, specify the Help Root (*Pt_ARG_WINDOW_HELP_ROOT*) resource for each window in your application. This allows you to specify relative topic paths for widgets inside the window.

Use a topic path, not a URL.

   The topic root should start with a slash (`/`), and should be the top of all topics for the window, usually taken from a TOC file in the **/usr/help/product** directory. For example:

   ```
   /Photon microGUI/User's Guide
   ```

2. For each widget in the window, fill in the Help Topic (*Pt_ARG_HELP_TOPIC*) resource. If you specified a topic root for the window, this topic path can be relative to the window's topic root. For example:

   ```
   Introduction
   ```

When the user clicks on the ? icon and selects the widget, the help information is displayed in the Helpviewer.

If you get an error message about a bad link when you ask for help for a widget, make sure that the topic path is correct.

Displaying help in a balloon

To use a balloon to display help information for a widget:

1. Put the text you want displayed in the balloon into the widget's Help Topic (*Pt_ARG_HELP_TOPIC*) resource.
2. Set the Pt_INTERNAL_HELP flag in the widget's Extended Flags (*Pt_ARG_EFLAGS*) resource.

When the user clicks on the ? icon, and selects the widget, the help information appears in a balloon.

Help without the ? icon

In many applications the ? icon in the window frame isn't suitable. However, you may still want to use the Photon Helpviewer for displaying help. For example:

- For touch screens, the window's ? icon may be too small.
- You may want the mouse pointer to change to the Help pointer when a key is pressed.

To get the mouse pointer to change to the Help pointer, forward the Ph_WM_HELP event to the window manager. The following code would be in a callback attached to a **`PtButton`** widget labeled **`Help`**:

```
int
help_cb( PtWidget_t *widget, ApInfo_t *apinfo,
         PtCallbackInfo_t *cbinfo )
{
    PhWindowEvent_t winev;

    memset( &winev, 0, sizeof(winev) );
    winev.event_f = Ph_WM_HELP;
    winev.rid = PtWidgetRid( window );
    PtForwardWindowEvent( &winev );

    return( Pt_CONTINUE );
}
```

The window must have Ph_WM_HELP set in the Managed Flags (*Pt_ARG_WINDOW_MANAGED_FLAGS*) resource. You must also fill in the Help Topic (*Pt_ARG_HELP_TOPIC*) resource for the widgets that have help, as outlined above.

Accessing help from your code

Use the following functions (described in the Photon *Library Reference*) to access help from your application's code—you don't need them if you're using the method described in "Connecting help to widgets":

PtHelpUrl()	Display help for the URL.
PtHelpUrlRoot()	Set a URL root.
PtHelpTopic()	Display help for a topic path.
PtHelpTopicRoot()	Set a topic root.
PtHelpTopicTree()	Display help for the topic tree.
PtHelpSearch()	Search for a string.
PtHelpQuit()	Exit the Helpviewer.

PtHelpUrlRoot() and *PtHelpTopicRoot()* don't save the passed string, so don't free it until you're finished using the help root.

Chapter 17
Interprocess Communication

In this chapter. . .

A Photon application can't always work in isolation — sometimes it needs to communicate with other processes.

The QNX Neutrino operating system supports various methods of interprocess communication (IPC), including:

- messages
- pulses
- signals

These methods can be used in a Photon application, as long as you're careful. However, it's best to use Photon connections:

- Connectors let the two communicating processes find each other easily. Photon connectors are registered with Photon, and therefore there's no chance of namespace conflicts between multiple Photon sessions running on the same machine.
- Photon connections know how to direct messages even if you have multiple connections between the same pair of processes. If you use raw Neutrino messages and input processes, you might need to handle that possibility yourself.

On the other hand, here's why raw Neutrino messages and/or pulses might sometimes be better:

- If one of the two communicating processes is not a Photon application, it can't use the Photon library. In this case both processes should use Neutrino messages or pulses.
- If the two processes don't necessarily belong to the same Photon session, they will need some other way of finding each other.
- If all you need is pulses, using a Photon connection is overkill.

The Photon main event-handling loop that your application calls is responsible for handling Photon events so that widgets update themselves and your callback functions are called.

This simple event-driven model of programming used with the Photon widget library presents some challenges for the application developer because the event-handling loop performs an unconditional *MsgReceive()* to obtain events from Photon. This means your application has to be careful if it needs to call *MsgReceive()*, or Photon events might go astray and the user interface might not be updated.

If you need to:

- respond to other messages in your application
- perform I/O using another mechanism (such as reading from a pipe)
- process signals

- respond to pulses

you'll need a way to hook your application code into the event-handling loop. Similarly, you may want to be able to add time-outs to your application and associate callback functions with them.

Connections

The process of establishing a connection uses an object called a *connector*. The connector is a name that the server creates and owns, and the client attaches its connection to. The connector is used only for establishing a connection.

The connector has a numeric ID and may also have a name associated with it. Both the name and the ID are unique in their Photon session. Here are a few examples of how the name can be used:

- When the server is started, it creates a connector with a well known name (such as **`Helpviewer`**). Clients connect to it, send requests, and then disconnect. If a client fails to find the name, it spawns the server and retries.
- A server creates a nameless connector and somehow sends the connector's ID to a potential client (drag-and-drop events use this scheme). The client then uses the ID to connect to the server.
- If a client always needs to spawn a new server for itself even if a copy of the server is already running, the client can call *PtConnectionTmpName()*. to create a "temporary connector name," pass the name to the server in a command-line argument or an environment variable, and then connect to the server.

Naming conventions

You can define unique names for your connectors by following these naming conventions:

- Names that don't contain a slash are reserved for QNX Software Systems.
- Third-party products should register only names that start with a unique string (e.g. an email address or a domain name) followed by a slash. For example, **`www.acme.com/dbmanager`**.

Typical scenario

Here's how you typically use connections:

1. The server calls *PtConnectorCreate()* to set up a connector, specifying a function to call whenever a client connects to the connector.
2. If the client needs a connector ID to find the connector, the server calls *PtConnectorGetId()* to determine the ID. The server then needs to give the ID to the client.

3 The client looks for a connector by calling *PtConnectionFindName()* or *PtConnectionFindId()*. If these functions succeed, they return a client connection object of type **`PtConnectionClient_t`**.

   The client can make repeated attempts (within a specified time limit or until the server terminates) to find the server by calling *PtConnectionWaitForName()*.

4 If the client finds the connector, the library sets up a connection to it and invokes the callback that the server specified when it created the connector, passing a **`PtConnectionServer_t`** server connection object to the routine.

5 The server's callback uses *PtConnectionAddMsgHandlers()* to set up a handler for any messages from the client.

6 The client uses *PtConnectionSend()* or *PtConnectionSendmx()* to send a message to the server. The client blocks until the server replies.

7 The library invokes the server's message handler, which calls *PtConnectionReply()* or *PtConnectionReplymx()* to reply to the client.

8 The client and server continue to exchange messages.

9 If the client wants to break the connection, it calls *PtConnectionClientDestroy()*; if the server wants to break the connection, it calls *PtConnectionServerDestroy()*.

10 When the server no longer needs the connector, it destroys it by calling *PtConnectorDestroy()*.

You can pass user data with the connection. The server calls *PtConnectionServerSetUserData()* to specify the data that the client can retrieve by calling *PtConnectionClientGetUserData()*. Similarly, the client calls *PtConnectionClientSetUserData()* to specify the data that the server can retrieve by calling *PtConnectionServerGetUserData()*.

You can set up functions to handler errors; the server does this by calling *PtConnectionServerSetError()*, and the client by calling *PtConnectionClientSetError()*.

The server can also use events to communicate with the client:

1 The client sets up one or more event handlers by calling *PtConnectionAddEventHandlers()*. You can set up different types of messages, and handlers for each type.

2 To send an event to the client, the server calls *PtConnectionNotify()*, which in turn can call *PtConnectionFlush()* if there isn't enough room for the notification in the server's buffer.

3 The server can change the size of the buffer by calling *PtConnectionResizeEventBuffer()*.

Local connections

It's possible for a process to create a connection to itself. The behavior of such a connection differs a little bit from a normal connection:

- For a normal connection, the *PtConnectionSend()*, *PtConnectionSendmx()*, and *PtConnectionNotify()* functions simply send a message or a pulse, and never execute any user code (except when *PtConnectionNotify()* needs to flush the buffer, which may cause Photon events to be processed in the normal way).

  If the connection is local, *PtConnectionSend()*, *PtConnectionSendmx()*, and *PtConnectionNotify()* invoke the handler directly, and both the calling code and the handler must take into account any side effect of that.

- Another difference is that the handlers are called from a different context. Normally, an event handler or message handler callback is called from within an input function. Since input functions are not invoked unless you either return to the main loop or call *PtBkgdHandlerProcess()* or *PtProcessEvent()*, it's often safe to assume that a handler won't be invoked while a callback is running. But if the connection is local, the handler is invoked directly by *PtConnectionSend()*, *PtConnectionSendmx()*, or *PtConnectionNotify()*, and the assumptions need to be modified accordingly.

- Another side effect of this is that if a message handler calls *PtConnectionNotify()*, the client gets the notification before the reply. In other words, the client's event handler is invoked before the *PtConnectionSend()* or *PtConnectionSendmx()* call returns. If the handler calls *PtConnectionSend()* or *PtConnectionSendmx()* again, the call fails with *errno* set to EBUSY (that's how the library protects itself from infinite recursion).

  The simplest way around this is to avoid sending notifications from within a message handler — instead, the notification can be placed in the reply.

Example

This application uses a connector to determine if there's already another instance of the application running. The program takes two command-line options:

-e	If another instance of the application is already running, tell it to exit.
-f *file*	If another instance of the application is already running, tell it to open the given *file*; otherwise just open the file.

Here's the code:

```
/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
```

```
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "abimport.h"
#include "proto.h"

enum MyMsgType {
    MY_MSGTYPE_EXIT, MY_MSGTYPE_OPEN_DOC, MY_MSGTYPE_TOFRONT
    };

enum MyReplyType {
    MY_REPTYPE_SUCCESS, MY_REPTYPE_BADMSG
    };

struct MyMsg {
    char docname[ PATH_MAX ];
    };

struct MyReply {
    enum MyReplyType status;
    };

/* Handle a message from a client: */
static PtConnectionMsgFunc_t msghandler;

static void const *msghandler(
        PtConnectionServer_t *connection, void *data,
        unsigned long type, void const *msgptr,
        unsigned msglen, unsigned *reply_len
        )
{
    struct MyMsg const *msg = (struct MyMsg const*) msgptr;
    static struct MyReply reply;

    reply.status = MY_REPTYPE_SUCCESS;
    switch ( type ) {
        case MY_MSGTYPE_EXIT :
            PtConnectionReply( connection, sizeof(reply),
                               &reply );
            PtExit( EXIT_SUCCESS );
            break;

        case MY_MSGTYPE_OPEN_DOC :
            reply.status = OpenNewDocument( msg->docname );
            break;

        case MY_MSGTYPE_TOFRONT :
            break;

        default :
            reply.status = MY_REPTYPE_BADMSG;
        }

    PtWindowToFront( ABW_base );
    *reply_len = sizeof(reply);
    return &reply;
}

/* Set up a new connection: */
static PtConnectorCallbackFunc_t connector_callback;
static void connector_callback(
               PtConnector_t *connector,
               PtConnectionServer_t *connection,
               void *data )
{
    static const PtConnectionMsgHandler_t
        handlers = { 0, msghandler };
    if ( PtConnectionAddMsgHandlers( connection,
                                     &handlers, 1 ) != 0 ) {
        fputs( "Unable to set up connection handler\n", stderr );
        PtConnectionServerDestroy( connection );
    }
}

/* Application Options string */
const char ApOptions[] =
    AB_OPTIONS "ef:"; /* Add your options in the "" */

/* Application initialization function */
int init( int argc, char *argv[] )
{
    struct MyMsg msg;
    int opt;
    long msgtype = MY_MSGTYPE_TOFRONT;
    const char *document = NULL;
    static const char name[] = "me@myself.com/ConnectionExample";

    while ( ( opt = getopt( argc, argv, ApOptions ) ) != -1 )
        switch ( opt ) {
            case '?' :
                PtExit( EXIT_FAILURE );

            case 'e' :
                msgtype = MY_MSGTYPE_EXIT;
                break;

            case 'f' :
                document = optarg;
            }

    if ( document )
        if ( msgtype == MY_MSGTYPE_EXIT ) {
            fputs(
               "You can't specify both the -e and -f options\n",
               stderr );
            PtExit( EXIT_FAILURE );
        } else {
            msgtype = MY_MSGTYPE_OPEN_DOC;
            strncpy( msg.docname, document,
                     sizeof(msg.docname)-1 );
        }

    while ( PtConnectorCreate( name, connector_callback, 0 )
            == NULL ) {

            /* If this failed, another instance of the app must
               be already running */

            PtConnectionClient_t *clnt;
            if ( ( clnt = PtConnectionFindName( name, 0, 0 ) )
                 != 0 ) {
                struct MyReply reply;
                int result = PtConnectionSend( clnt, msgtype,
                            &msg, &reply, sizeof(msg),
                            sizeof(reply) );
                PtConnectionClientDestroy( clnt );
                if ( result == 0 )
                    PtExit( reply.status );
            }
    }

    /* Since PtConnectorCreate() has succeeded, we're the only
       instance of the app running */

    if ( msgtype == MY_MSGTYPE_EXIT ) {
        fputs( "Can't tell it to exit; it's not running\n",
               stderr );
        PtExit( EXIT_FAILURE );
        }

    if ( document )
        OpenNewDocument( document );

    return Pt_CONTINUE;
}
```

Sending QNX messages

A Photon application can use *MsgSend()* to pass messages to another process, but the other process needs to *MsgReply()* promptly, as Photon events aren't processed while the application is blocked. (Promptness isn't an issue if your application has multiple threads that process events, and you call *PtLeave()* before *MsgSend()*, and *PtEnter()* after. For a discussion of writing applications that use multiple threads, see the Parallel Operations chapter.)

As an example, here's a callback that extracts a string from a text widget, sends it to another process, and displays the reply in the same text widget:

```
/* Callback that sends a message to another process      */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/neutrino.h> /* Needed for MsgSend() */

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "globals.h"
#include "abimport.h"
#include "proto.h"

extern int coid;

int
send_msg_to_b( PtWidget_t *widget,
               ApInfo_t *apinfo,
               PtCallbackInfo_t *cbinfo )

{
  char *a_message;

  /* eliminate 'unreferenced' warnings */
  widget = widget, apinfo = apinfo, cbinfo = cbinfo;

  /* Get the string from the text widget. */

  PtGetResource (ABW_msg_text, Pt_ARG_TEXT_STRING, 0, 0);

  /* Send the string to another process. */

  a_message = (char *)args[0].value;
  if ( MsgSend (coid, a_message, msg_size,
             rcv_msg, msg_size) == -1)
  {
    perror ("Send to B failed");
    PtExit (-1);
  }

  /* Remember the UI is "hung" until the other
     process replies! */

  /* Display the reply in the same text widget. */

  PtSetResource (ABW_msg_text, Pt_ARG_TEXT_STRING,
                 rcv_msg, 0);

  return( Pt_CONTINUE );

}
```

For more information on messages, see the QNX Neutrino *System Architecture* guide.

Receiving QNX messages

To handle non-Photon events in a Photon application, you need to register an *input handling procedure* (or *input handler*), otherwise the events are lost. This is because the unconditional *MsgReceive()* performed by the Photon event handling loop ignores non-Photon events.

You can create your own channel and call *MsgReceive()* on it, but remember that your application and its interface will be blocked until that process sends a message. It's better to use an input handler as described in this section.

An input handler is responsible for handling messages received by the application from a particular process. When you register the handler with the widget library, you identify the pid the input handler is associated with.

You can define more than one input handler in your application for a pid. When a message is received from that process, the widget library starts calling the handlers in the list, starting with the last one registered, and continuing up the list in reverse order until a handler recognizes the message (that is, it doesn't return Pt_CONTINUE). See the description of Pt_CONTINUE below.

You can register a nonspecific input handler by specifying a value of zero for the pid. This handler is called when the application receives:

- *any* non-Photon messages that don't have an input handler specifically associated with the sender's pid
- *any* non-Photon messages that have input handlers associated with the sender's pid, but which were not handled (that is, the input handlers returned PT_CONTINUE)
- a user pulse (i.e. a pulse with a nonnegative code)

Adding an input handler

To register an input handler, call *PtAppAddInput()* when you initialize the application. The syntax is given below; for more information, see the Photon *Library Reference*.

```
PtInputId_t *PtAppAddInput(
                 PtAppContext_t app_context,
                 pid_t pid,
                 PtInputCallbackProc_t input_func,
                 void *data );
```

The arguments are:

app_context	The address of the application context, a **PtAppContext_t** structure that manages all the data associated with this application. Pass NULL for this argument, so that the default context is used.
pid	The process ID of the process whose messages this handler is to deal with, or 0 if the handler is for messages from all processes.
input_func	Your input handler, of type **PtInputCallbackProc_t**. For details, see the Photon *Library Reference*.
data	Extra data to be passed to the input handler.

PtAppAddInput() returns a pointer to an input-handler ID, which you'll need if you want to remove the input handler later.

The prototype for an input handler is as follows:

```
int input_proc( void *data,
                int rcvid,
                void *msg,
                size_t msglen );
```

The arguments are:

data	A pointer to any extra data you want to pass to the input handler.
rcvid	The rcvid of the process that sent the message.

msg	A pointer to the message sent.
msglen	The size of the message buffer. If the actual message is longer than the buffer, load the rest of the message by calling *MsgRead()*.

You can also declare the input handler to be of type **`PtInputCallbackProcF_t`** to take advantage of the prototype checking done by the compiler.

If your input handler changes the display, it should call *PtFlush()* to make sure the display is updated.

An input handler must return one of the following:

Pt_CONTINUE	The input handler doesn't recognize the message. If there are other input handlers attached to the same process ID, they're called. If there are no input handlers attached specifically to this process ID, or if all input handlers attached specifically to this process ID return Pt_CONTINUE, the library looks for input handlers attached to rcvid 0. If all the input handlers return Pt_CONTINUE, the library replies to the message with an ENOSYS.
Pt_END	The message has been recognized and processed and the input handler needs to be removed from the list. No other input handlers are called for this message.
Pt_HALT	The message has been recognized and processed but the input handler needs to stay on the list. No other input handlers are called for this message.

name_attach() and *PtAppAddInput()*

If possible, you should use a Photon connection instead of *name_attach()* to establish a connection with another process. However, you can't use a Photon connection in these cases:

- The connecting client isn't a Photon application.
- The connecting processes belong to a different Photon session. This applies even if you're running multiple Photon sessions on one machine, or if your Photon session consists of applications running on several machines and the two connecting processes happen to be on different machines.

PtAppAddInput() and *name_attach()* both try to create a channel with _NTO_CHF_COID_DISCONNECT and _NTO_CHF_DISCONNECT set (see the QNX Neutrino *Library Reference*). If your application calls both functions, you need to let Photon use the same channel as *name_attach()*, by calling *PhChannelAttach()* first, like this:

```
PhChannelAttach( chid, -1, NULL );
```

before calling *name_attach()* or *PtAppAddInput()*.

If you want to create a separate channel for Photon, it doesn't matter whether you create it and give it to *PhChannelAttach()* before or after calling *name_attach()*. But keep in mind that since certain mechanisms in Photon library expect the Photon channel to have the two DISCONNECT flags, they might not work properly if it doesn't. One such mechanism is the detection of broken connections (see *PtConnectionClientSetError()* and *PtConnectionServerSetError()*) and anything that relies on it.

Removing an input handler

To remove an input handler:

- Have it return Pt_END.

  Or:

- Call *PtAppRemoveInput()*, passing it the ID returned by *PtAppAddInput()*.

Message buffer size

As described above, arguments to your input function include:

msg	A pointer to an event buffer that was used to receive the message.
msglen	The size of the buffer.

This buffer might not be large enough to hold the entire message. One way of handling this is to have the first few bytes of the message indicate the message type and from that determine how big the message should be. Once you know the message size, you can:

- Reread the entire message by calling *MsgReadv()*.

  Or:

- Copy the part you've already received into a new buffer. Get the rest of the message by calling *MsgReadv()*. Add the rest of the message to the first part.

Alternatively, you can set the event buffer to be the size of the largest message your application will receive (if known). This can be done by calling *PtResizeEventMsg()*. You'd typically call this before you expect to receive any messages.

PtResizeEventMsg() won't reduce the message buffer beyond a certain minimum size. This is so that the widget library will continue to function.

Example — logging error messages

The following code fragment shows how a nonspecific input handler may be used to respond to error-logging messages from any process. When one of these messages is received, the application displays the message's contents in a multiline text widget. (This example assumes **`log_message`** is declared elsewhere.)

```
int input_proc(void *client_data, int rcvid, void *msg,
               size_t msglen)
{
   struct log_message *log = (struct log_message *)msg;

   /* Only process log messages */
   if (log->type == LOG_MSG)
   {
      PtWidget_t *text = (PtWidget_t *)client_data;
      struct log_message header;
      int msg_offset = offsetof(struct log_message, msg);
      int log_msglen;
      int status;

      /* See if our entire header is in the buffer --
         it should be */
      if (msglen < msg_offset)
      {
         /* Read in the whole header */
         if (MsgRead(rcvid, &header, msg_offset, 0)  == -1)
         {
            status = errno;
            MsgError( rcvid, status);
            return Pt_HALT;    /* bail out */
         }
         log = &header;
      }

      log_msglen = msg_offset+log->msg_len;

      /* See if the whole message is in the buffer */

      if (msglen < log_msglen)
      {
         struct log_message *log_msg =
            (struct log_message *)alloca(log_msglen);

         /* Read the remainder of the message into
            space on the stack */

         if (log_msg == NULL ||
             MsgRead( rcvid, log_msg, log_msglen, 0) == -1)
         {
            status = errno;
            MsgError( rcvid, status);
            return Pt_HALT;    /* bail out */
         }
         log = log_msg;
      }

      add_msg(text, log);
      status = 0;
      MspReply( rcvid, 0, 0, 0);
   }
```

```
    return Pt_HALT;
}
```

This application registers the *input_proc()* function as an input handler for handling non-Photon messages from any other process.

The *input_proc()* function first checks the message type of the incoming message. If the input handler isn't responsible for this type of message, it returns immediately. This is important because any other nonspecific input handlers that were registered will be called as well, and only one of them should respond to a given message.

If the type of message received is a log message, the function makes sure that Photon has read the entire message into the Photon event buffer. This can be determined by looking at the message length provided as the *msglen* to the input handler. If part of the message isn't in the event buffer, a message buffer is allocated and *MsgRead()* is called to get the whole message. The *input_proc()* function then calls *add_msg()* to add the message to the text widget and replies to the message.

When *input_proc()* is complete, it returns the value Pt_HALT. This instructs the Photon widget library not to remove the input handler.

Photon pulses

In addition to synchronous message-passing, Photon supports pulses. A process that wants to notify another process but doesn't want to wait for a reply can use a Photon pulse. For example, a server can use a pulse to contact a client process in a situation where sending a message would leave both SEND-blocked (and hence deadlocked).

A Photon pulse is identified by a negative PID that can be used as the *pid* argument to *PtAppAddInput()*. This PID is local to your application. If you want another process to send pulses to you, you must "arm" the pulse using *PtPulseArm()*. This creates a **`PtPulseMsg_t`** object that can be sent to the other process in a message. The other process will then be able to send pulses by calling *MsgDeliverEvent()* function.

Under the QNX Neutrino OS version 6, **`PtPulseMsg_t`** is a **`sigevent`** structure. The bits in *msg.sigev_value.sival_int* that correspond to _NOTIFY_COND_MASK are clear, but can be set by the application that sends the pulse. For more information, see *ionotify()* in the QNX Neutrino *Library Reference*.

PtPulseArm() (described in the Photon *Library Reference*) simply takes a **`sigevent`** structure. *PtPulseArmFd()* and *PtPulseArmPid()* are for compatibility with earlier version of the QNX Neutrino OS and the Photon microGUI.

Let's look at the code you'll need to write to support pulses in a:

- Photon application that receives pulses
- Photon application that delivers pulses

Photon application that receives a pulse

It's the recipient of a Photon pulse that has to do the most preparation. It has to:

1. Create the pulse.
2. Arm the pulse.
3. Send the pulse message to the process that will deliver it.
4. Register an input handler for the pulse message.
5. Deliver the pulse to itself, if necessary.
6. Destroy the pulse when it's no longer needed.

The sections below discuss each step, followed by an example.

Before exiting, the recipient process should tell the delivering process to stop sending pulses.

Creating a pulse

To create a Photon pulse, call *PtAppCreatePulse()*:

```
pid_t PtAppCreatePulse( PtAppContext_t app,
                        int priority );
```

The arguments are:

app	The address of the application context, a **PtAppContext_t** structure that manages all the data associated with this application. You should pass NULL for this argument, so that the default context is used.
priority	The priority of the pulse. If this is -1, the priority of the calling program is used.

PtAppCreatePulse() returns a pulse process ID, which is negative but never -1. This is the receiver's end of the pulse.

Arming a pulse

Arming the pulse fills in the **sigevent** structure, which can be used for most of the QNX Neutrino calls that take this type of argument.

There's nothing wrong with having more than one process deliver the same pulse, although the recipient won't be able to tell which process sent it.

To arm a pulse, call *PtPulseArm()*. The prototype is:

```
int PtPulseArm( PtAppContext_t app,
                pid_t pulse,
                struct sigevent *msg );
```

The arguments are:

app	A pointer to the **`PtAppContext_t`** structure that defines the current application context (typically NULL).
pulse	The pulse created by *PtAppCreatePulse()*.
msg	A pointer to a pulse message that this function creates. This is the deliverer's end of the pulse, and we'll need to send it to that process, as described below.

This function returns a pointer to a pulse message ID, which you'll need later.

Sending the pulse message to the deliverer

The method you use to send the pulse message depends on the process that will deliver the pulse. For example,

- For a resource manager, use *ionotify()*:

```
ionotify (fd, _NOTIFY_ACTION_POLLARM, _NOTIFY_COND_INPUT,
          &pulsemsg);
```

- For other process types, you could use *MsgSendv()*:

```
/* Create your own message format: */
msg.pulsemsg = pulsemsg;
MsgSendv (connection_id, &msg, msg_parts, &rmsg, rmsg_parts);
```

Registering an input handler

Registering an input handler for the pulse is similar to registering one for a message; see "Adding an input handler" earlier in this chapter. Pass the pulse ID returned by *PtAppCreatePulse()* as the *pid* parameter to *PtAppAddInput()*.

The *rcvid* argument for the input handler won't necessarily have the same value as the pulse ID: it matches the pulse ID on the bits defined by _NOTIFY_DATA_MASK (see *ionotify()* in the QNX Neutrino *Library Reference*), but the other bits are taken from the Neutrino pulse that was received.

Delivering a pulse to yourself

If the application needs to send a pulse to itself, it can call *PtAppPulseTrigger()*:

```
int PtAppPulseTrigger( PtAppContext_t app,
                       pid_t pulse );
```

The parameters for this function are the **`PtAppContext_t`** structure that defines the application context (typically NULL) and the pulse ID returned by *PtAppCreatePulse()*.

Destroying a pulse

When your application no longer needs the pulse, it can be destroyed by calling *PtAppDeletePulse()*:

```
int PtAppDeletePulse( PtAppContext_t app,
                      pid_t pulse_pid );
```

The parameters are the **PtAppContext_t** structure that defines the application context (typically NULL) and the pulse ID returned by *PtAppCreatePulse()*.

Example — message queues

Here's an application that receives Photon pulses. It opens a message queue (**/dev/mqueue/testqueue** by default), sets up a pulse, and uses *mq_notify()* to give itself a pulse when there's something to read from the message queue:

```
/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <mqueue.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "abimport.h"
#include "proto.h"

mqd_t mqd = -1;
struct sigevent sigev;

static void readqueue( void ) {
    static unsigned counter;
    unsigned mprio;
    ssize_t msize;
    char mbuf[ 4096 ];
    while ( ( msize = mq_receive( mqd, mbuf, sizeof(mbuf),
                                  &mprio ) ) >= 0 ) {
        char hbuf[ 40 ];
        PtTextModifyText( ABW_mtext, 0, 0, -1, hbuf,
            sprintf( hbuf, "Msg #%u (prio %d):\n", ++counter,
                     mprio )
            );
        PtTextModifyText( ABW_mtext, 0, 0, -1, mbuf, msize );
        }
    if ( errno != EAGAIN )
        perror( "mq_receive" );
    }

static int input_fun( void *data, int rcvid, void *message,
                      size_t mbsize ) {

    if ( mq_notify( mqd, &sigev ) == -1 )
```

```
        perror( "mq_notify" );
    readqueue();
    return Pt_HALT;
    }

pid_t pulse;

/* Application Options string */
const char ApOptions[] =
    AB_OPTIONS ""; /* Add your options in the "" */


int init( int argc, char *argv[] ) {
    if  (   ( pulse = PtAppCreatePulse( NULL, -1 ) ) == 0
        ||  PtAppAddInput( NULL, pulse, input_fun, NULL ) ==
            NULL ) {
        fputs( "Initialization failed\n", stderr );
        exit( EXIT_FAILURE );
        }
    PtPulseArm( NULL, pulse, &sigev );
    /* eliminate 'unreferenced' warnings */
    argc = argc, argv = argv;
    return( Pt_CONTINUE );
    }

int open_queue( PtWidget_t *link_instance, ApInfo_t *apinfo,
                PtCallbackInfo_t *cbinfo ) {
    const char *name;
    PtArg_t arg;

    if ( mqd >= 0 )
        mq_close( mqd );

    PtSetArg( &arg, Pt_ARG_TEXT_STRING, &name, 0 );
    PtGetResources( ABW_qname, 1, &arg );
    if ( ( mqd = mq_open( name, O_RDONLY | O_CREAT | O_NONBLOCK,
                          S_IRUSR | S_IWUSR, NULL ) ) < 0 )
        perror( name );
    else
        if ( mq_notify( mqd, &sigev ) == -1 ) {
            perror( "mq_notify" );
            mq_close( mqd );
            mqd = -1;
            }
        else
            readqueue();

    /* eliminate 'unreferenced' warnings */
    link_instance = link_instance, apinfo = apinfo;
    cbinfo = cbinfo;

    return( Pt_CONTINUE );
    }
```

Photon application that delivers a pulse

A Photon application that's going to deliver a pulse must:

- Have an input handler for messages from the application that's going to receive the pulses. This input handler is created as described in "Receiving QNX messages" earlier in this chapter. It will need to handle messages that contain the pulse message, and tell the deliverer to stop sending pulses.

Save the *rcvid* from the message that contains the pulse message — you'll need it to deliver the pulse.

- Deliver the pulse by calling *MsgDeliverEvent()*.

Processing signals

If your application needs to process signals, you'll need to set up a signal handler. The problem is that you can't call Photon functions from a signal handler because the widget library isn't signal-safe or reentrant.

To get around this problem, the Photon library includes a signal handler. You register a signal-processing function, and Photon calls it *after*

- Photon's signal handler returns

  AND

- all processing for the current widget is complete

CAUTION: By handling signals in this way, you're not getting strict realtime performance, since your signal-processing function isn't called right away.

Adding a signal-processing function

To add a signal-processing function, use the *PtAppAddSignalProc()* function. You typically call this in

- your application's initialization function

  Or

- the setup function for a window

You'll need to include **<signal.h>**.

The syntax for *PtAppAddSignalProc()* is as follows:

```
int PtAppAddSignalProc( PtAppContext_t app,
                        sigset_t const *set,
                        PtSignalProc_t func,
                        void *data);
```

The arguments are as follows:

app The address of the application context, a **PtAppContext_t** structure that manages all the data associated with this application. Specify NULL for this argument, so that the default context is used.

set	A pointer to the set of signals that should cause the signal-processing function to be called. Use the *sigemptyset()* and *sigaddset()* functions to build this set. See the QNX Neutrino *Library Reference* for more information.
func	The signal-processing function. See **`PtSignalProc_t`** in the Photon *Library Reference*.
data	Any data to be passed to the function.

PtAppAddSignalProc() returns 0 on success, or -1 if an error occurs.

Your signal-processing function has the following prototype:

```
int signalProcFunctions (int signum
                         void *data);
```

The arguments are:

signum	The number of the signal to be processed.
data	The *data* parameter specified in the call to *PtAppAddSignalProc()*.

If you want the signal handler to remain installed, return Pt_CONTINUE. To remove it for the current signal, return Pt_END (if the function was registered for other signals, it's still called if they're raised).

Removing a signal-processing function

To remove a signal-processing function:

- Call *PtAppRemoveSignal()* to remove one or all occurrences of a (signal-processing function, data) pair.
- Return Pt_END from the signal-processing function. If the function was registered for more than one signal, it remains installed for signals other than the one just processed.

Other I/O mechanisms

If your application needs to perform I/O such as reading from or writing to a pipe, you should add an *fd handler*. An fd handler is a function that's called by the main event loop when a given file descriptor (fd) is ready for input or output:

- To add an fd handler to your application, call *PtAppAddFd()* or *PtAppAddFdPri()*.
- For details about the prototype of the fd handler, see **`PtFdProc_t`**.
- To change the mode that's of interest to the fd handler, call *PtAppSetFdMode()*
- To remove an fd handler, return Pt_END from it or call *PtAppRemoveFd()*.

These functions are described in the Photon *Library Reference*.

If your fd handler changes the display, it should call *PtFlush()* to make sure the display is updated.

Chapter 18

Parallel Operations

In this chapter...

Overview

When you have to perform an operation that takes a long time to execute, it's not a good idea to implement it as a simple callback. During the time the callback is executing, the widgets in your application can't repair damage and they won't respond to user input at all. You should develop a strategy for handling lengthy operations within your application that involves returning from your callback as quickly as possible.

Returning from your callback allows the widgets to continue to update themselves visually. It also gives some visual feedback if the user attempts to do anything. If you don't want the user to be able to perform any UI operations during this time, you should deactivate the menu and command buttons. You can do this by setting the Pt_BLOCKED flag in the application window widget's *Pt_ARG_FLAGS* resource.

You might consider one of several different mechanisms for dealing with parallel operations:

- If you can't break the operation into pieces, process Photon events while the operation continues; see "Background processing," below.
- If you can break the operation into small chunks, you may wish to have a function that keeps track of the current state and executes one small chunk of the operation at a time. You can then set up a timer widget and attach it to a callback that invokes the function whenever the timer goes off. Or you may call the function from within a *work procedure*. These methods are especially effective for iterative operations where the function may be executed once per iteration. See "Work procedures," below.
- Use multiple threads. This requires some special handling, because the Photon libraries aren't thread-safe; see "Threads," below.
- Spawn another process in the callback, and have the other process return its results to the application by sending it messages. In this case, it's very important to be able to monitor the operation's progress and give the user visual feedback.

Background processing

If a lengthy operation can't be easily decomposed, and you don't want to use multiple threads, you should at least call *PtBkgdHandlerProcess()* to process Photon events so that the GUI doesn't appear to be frozen.

If the operation is very lengthy, you can call *PtBkgdHandlerProcess()* within a loop. How often you need to call *PtBkgdHandlerProcess()* depends on what your application is doing. You should also find a way to let the user know what progress the operation is making.

For example, if you're reading a large directory, you could call the background handler after reading a few files. If you're opening and processing every file in a directory, you could call *PtBkgdHandlerProcess()* after each file.

It's safe to call *PtBkgdHandlerProcess()* in callbacks, work procedures, and input procedures, but *not* in a widget's Draw method (see *Building Custom Widgets*) or a **`PtRaw`** widget's drawing function.

If a callback calls *PtBkgdHandlerProcess()*, be careful if the application can invoke the callback more than once simultaneously. If you don't want to handle this recursion, you should block the widget(s) associated with the callback.

The following functions process Photon events:

- *PtBkgdHandlerProcess()*
- *PtFileSelection()*
- *PtProcessEvent()*
- *PtSpawnWait()*

Work procedures

A work procedure is run whenever there are no messages for your application to respond to. In every iteration of the Photon event-handling loop, this procedure is called if no messages have arrived (rather than block on a *MsgReceive()* waiting for more messages). This procedure will be run very frequently, so keep it as short as possible.

If your work procedure changes the display, call *PtFlush()* to make sure that it's updated.

See "Threads and work procedures," below, if you're writing a work procedure for a multithreaded program.

Work procedures are stacked; when you register a work procedure, it's placed on the top of the stack. Only the work procedure at the top of the stack is called. When you remove the work procedure that's at the top of the stack, the one below it is called.

There is one exception to this rule. If the work procedure that's at the top of the stack is running already, the next one is called. This is only possible if the already running procedure allows the Photon library to start another one, perhaps by calling a modal function like *PtModalBlock()*, *PtFileSelection()* or *PtAlert()*, or calling *PtLeave()* while you have other threads ready to process events.

The work procedure itself is a callback function that takes a single **`void *`** parameter, *client_data*. This *client_data* is data that you associate with the work procedure when you register it with the widget library. You should create a data structure for the work procedure that contains all its state information and provide this as the *client_data*.

To register, or add, a work procedure, call *PtAppAddWorkProc()*:

```
PtWorkProcId_t *PtAppAddWorkProc(
                   PtAppContext_t app_context,
                   PtWorkProc_t work_func,
                   void *data );
```

The parameters are:

- *app_context*—the address of the application context, a **PtAppContext_t** structure that manages all the data associated with this application. This must be specified as NULL, so that the default context is used.
- *work_func*—the work procedure callback function. See **PtWorkProc_t** in the Photon *Library Reference*.
- *client_data*—the data to be passed to the function when it's invoked.

PtAppAddWorkProc() returns a pointer to a **PtWorkProcId_t** structure that identifies the work procedure.

To remove a work procedure when it's no longer needed, call *PtAppRemoveWorkProc()*:

```
void PtAppRemoveWorkProc(
                PtAppContext_t app_context,
                PtWorkProcId_t *WorkProc_id );
```

passing it the same application context and the pointer returned by *PtAppAddWorkProc()*.

A practical example of the use of work procedures would be too long to cover here, so here's a simple iterative example. The work procedure counts to a large number, updating a label to reflect its progress on a periodic basis.

```
#include <Pt.h>
#include <stdlib.h>

typedef struct workDialog {
   PtWidget_t *widget;
   PtWidget_t *label;
   PtWidget_t *ok_button;
} WorkDialog_t;

typedef struct countdownClosure {
   WorkDialog_t *dialog;
   int value;
   int maxvalue;
   int done;
   PtWorkProcId_t *work_id;
} CountdownClosure_t;

WorkDialog_t *create_working_dialog(PtWidget_t *parent)
{
   PhDim_t      dim;
   PtArg_t      args[3];
   int          nargs;
   PtWidget_t   *window, *group;
   WorkDialog_t *dialog =
      (WorkDialog_t *)malloc(sizeof(WorkDialog_t));
   if (dialog)
   {
      dialog->widget = window =
         PtCreateWidget(PtWindow, parent, 0, NULL);

      nargs = 0;
      PtSetArg(&args[nargs], Pt_ARG_GROUP_ORIENTATION,
               Pt_GROUP_VERTICAL, 0); nargs++;
      PtSetArg(&args[nargs], Pt_ARG_GROUP_VERT_ALIGN,
               Pt_GROUP_VERT_CENTER, 0); nargs++;
      group = PtCreateWidget(PtGroup, window, nargs, args);

      nargs = 0;
      dim.w = 200;
      dim.h = 100;
      PtSetArg(&args[nargs], Pt_ARG_DIM, &dim, 0); nargs++;
      PtSetArg(&args[nargs], Pt_ARG_TEXT_STRING,
               "Counter:        ", 0); nargs++;
      dialog->label = PtCreateWidget(PtLabel, group,
                                     nargs, args);

      PtCreateWidget(PtSeparator, group, 0, NULL);

      nargs = 0;
      PtSetArg(&args[nargs], Pt_ARG_TEXT_STRING, "Stop", 0);
               nargs++;
      dialog->ok_button = PtCreateWidget(PtButton, group,
                                         1, args);
   }
   return dialog;
}

int done(PtWidget_t *w, void *client,
          PtCallbackInfo_t *call)
{
   CountdownClosure_t *closure =
      (CountdownClosure_t *)client;

   call = call;

   if (!closure->done) {
      PtAppRemoveWorkProc(NULL, closure->work_id);
   }
   PtDestroyWidget(closure->dialog->widget);
   free(closure->dialog);
   free(closure);
   return (Pt_CONTINUE);
}

int
count_cb(void *data)
{
   CountdownClosure_t *closure =
      (CountdownClosure_t *)data;
   char    buf[64];
   int     finished = 0;

   if ( closure->value++ == 0 || closure->value %
        1000 == 0 )
   {
      sprintf(buf, "Counter: %d", closure->value);
      PtSetResource( closure->dialog->label,
                     Pt_ARG_TEXT_STRING, buf, 0);
   }

   if ( closure->value == closure->maxvalue )
   {
      closure->done = finished = 1;
      PtSetResource( closure->dialog->ok_button,
                     Pt_ARG_TEXT_STRING, "Done", 0);
   }

   return finished ? Pt_END : Pt_CONTINUE;
}

int push_button_cb(PtWidget_t *w, void *client,
                   PtCallbackInfo_t *call)
{
   PtWidget_t   *parent = (PtWidget_t *)client;
   WorkDialog_t *dialog;

   w = w; call = call;

   dialog = create_working_dialog(parent);

   if (dialog)
   {
      CountdownClosure_t *closure =
          (CountdownClosure_t *)
           malloc(sizeof(CountdownClosure_t));

      if (closure)
      {
         PtWorkProcId_t *id;

         closure->dialog = dialog;
         closure->value = 0;
         closure->maxvalue = 200000;
         closure->done = 0;
         closure->work_id = id =
            PtAppAddWorkProc(NULL, count_cb, closure);

         PtAddCallback(dialog->ok_button, Pt_CB_ACTIVATE,
                       done, closure);
         PtRealizeWidget(dialog->widget);
      }
   }
   return (Pt_CONTINUE);
}

int main(int argc, char *argv[])
{
   PhDim_t      dim;
   PtArg_t      args[3];
   int          n;
   PtWidget_t   *window;
   PtCallback_t callbacks[] = {{push_button_cb, NULL}};
   char Helvetica14b[MAX_FONT_TAG];

   if (PtInit(NULL) == -1)
      exit(EXIT_FAILURE);

   dim.w = 200;
   dim.h = 100;
   PtSetArg(&args[0], Pt_ARG_DIM, &dim, 0);
   if ((window = PtCreateWidget(PtWindow, Pt_NO_PARENT,
                                1, args)) == NULL)
      PtExit(EXIT_FAILURE);

   callbacks[0].data = window;
   n = 0;
   PtSetArg(&args[n++], Pt_ARG_TEXT_STRING, "Count Down...", 0);

   /* Use 14-point, bold Helvetica if it's available. */

   if(PfGenerateFontName("Helvetica", PF_STYLE_BOLD, 14,
                         Helvetica14b) == NULL) {
       perror("Unable to generate font name");
   } else {
       PtSetArg(&args[n++], Pt_ARG_TEXT_FONT, Helvetica14b, 0);
   }
   PtSetArg(&args[n++], Pt_CB_ACTIVATE, callbacks,
            sizeof(callbacks)/sizeof(PtCallback_t));
   PtCreateWidget(PtButton, window, n, args);

   PtRealizeWidget(window);

   PtMainLoop();
   return (EXIT_SUCCESS);
}
```

When the pushbutton is pressed, the callback attached to it creates a working dialog and adds a work procedure, passing a closure containing all the information needed to perform the countdown and to clean up when it's done.

The closure contains a pointer to the dialog, the current counter, and the value to count to. When the value is reached, the work procedure changes the label on the dialog's button and attaches a callback that will tear down the entire dialog when the button is pressed. Upon such completion, the work procedure returns Pt_END in order to get removed.

The *done()* function is called if the user stops the work procedure, or if it has completed. This function destroys the dialog associated with the work procedure and removes the work procedure if it was stopped by the user (i.e. it didn't run to completion).

If you run this example, you may discover one of the other features of work procedures — they preempt one another. When you add a new work procedure, it preempts all others. The new work procedure will be the only one run until it has completed or is removed. After that, the work procedure that was previously running resumes. This is illustrated in the above example if the user presses the *Count Down...* button before a countdown is finished. A new countdown dialog is created, and that countdown runs to the exclusion of the first until it's done.

The granularity for this preemption is at the function call level. When the callback function for a work procedure returns, that work procedure may be preempted by another work procedure.

Threads

Photon applications are event-driven and callback-based; whenever an event arrives, the appropriate callback is invoked to handle it, and then the control returns to the event loop to wait for the next event. Because of this structure, most Photon applications are single-threaded.

The Photon library lets you use threads, but in a way that minimizes the overhead for single-threaded applications. The Photon library is "thread-friendly," rather than completely thread-safe the way *printf()* and *malloc()* are thread-safe.

Don't cancel a thread that might be executing a Photon library function or a callback (because the library might need to do some cleanup when the callback returns).

Locking the Photon library

You can use multiple threads by arranging your program so that only the thread that called *PtInit()* calls Photon functions, but you might find this approach restrictive.

The Photon library is mostly single-threaded, but has a mechanism that lets multiple threads use it in a safe way. This mechanism is a *library lock*, implemented by the *PtEnter()* and *PtLeave()* functions.

This lock is like a big mutex protecting the Photon library: only one thread can own the lock at a time, and only that thread is allowed to make Photon calls. Any other thread that wants to call a Photon function must call *PtEnter()* first, which blocks until the lock is available. When a thread no longer needs the lock, it calls *PtLeave()* to let other threads use the Photon library.

To write your non-Photon threads:

- Put calls to *PtEnter()* and *PtLeave()* around any Photon calls in them.
- Group as much Photon code together as you can and wrap it in a single enter-leave pair, because this minimizes the number of potentially blocking calls to *PtEnter()* in your code.
- Try to leave any non-Photon code that can take a while to complete outside of the enter-leave section — otherwise it may unnecessarily prevent other threads from doing their job.

Don't call *PtLeave()* if your thread hasn't called *PtEnter()*, because your application could crash or misbehave.

Remember that if you're in a callback function, something must have called *PtEnter()* to let you get there.

PtLeave() doesn't atomically give the library lock to another thread blocked inside *PtEnter()*; the other thread gets unblocked, but then it must compete with any other threads as if it just called *PtEnter()*.

You should use *PtEnter()* and *PtLeave()* instead of using your own mutex because when *PtProcessEvent()* (which *PtMainLoop()* calls) is about to wait for an event, it unlocks the library. Once *PtProcessEvent()* has an event that it can process, it locks the library again. This way, your non-Photon threads can freely access Photon functions when you don't have any events to process.

If you use your own mutex that *PtProcessEvent()* doesn't know about, it's unlocked only when your code unlocks it. This means that the only time that your non-Photon threads can lock the mutex is when your application is processing an event that invokes one of your callbacks. The non-Photon threads can't lock the mutex when the application is idle.

Multiple event-processing threads

If you need to have a lengthy callback in your application, you can have your callback invoke *PtBkgdHandlerProcess()* as described earlier in this chapter. You can also spawn a new thread to do the job instead of doing it in the callback.

Another choice is to have more than one Photon thread that processes Photon events in your application. Here's how:

- Spawn one or more extra threads that call *PtEnter()* followed by *PtMainLoop()*. If one of your Photon threads receives an event that invokes your lengthy callback, the remaining threads can take over processing Photon events.
- Call *PtLeave()* from the callback to give the other threads access to the Photon library.
- Don't forget to call *PtEnter()* before returning from the callback; the code that invokes your callback expects to own the Photon lock when the callback returns.

Unlocking the library lets other threads modify your widgets and global variables while you're not looking, so be careful.

If your callback allows other threads to process events while it's doing its lengthy operation, there's a chance that the person holding the mouse may press the same button again, invoking your callback before its first invocation is complete.

You have to make sure that your application either handles this situation properly, or prevents it from happening. Here are several ways to do this:

- Block your button before the callback calls *PtLeave()*, and unblock the button after calling *PtEnter()*.

  Or:
- Use a flag to tell the second invocation of the callback that it's already running.

  Or:
- Use a counter if you want to count rather than just ignore any extra button presses.

  Or:

- Use your own mutex or another synchronization mechanism to make sure that you're not stepping on your own toes (but watch out for potential deadlocks).

Realtime threads

Don't make Photon calls from threads that must have deterministic realtime behavior. It's hard to predict how long *PtEnter()* will block for; it can take a while for the thread that owns the lock to finish processing the current event or call *PtLeave()*, especially if it involves sending to other processes (like the window manager).

It's better to have a "worker thread" that accepts requests from your realtime threads and executes them in its own enter-leave section. A condition variable — and possibly a queue of requests — is a good way of sending these requests between threads.

If you're using worker threads, and you need to use a condition variable, call *PtCondWait()* instead of *pthread_cond_wait()* and a separate mutex. *PtCondWait()* uses the Photon library lock as the mutex and makes an implicit call to *PtLeave()* when you block, and to *PtEnter()* when you unblock.

The threads block until:

- They get access to the library.
- They get hit by a signal.
- Another thread raises a signal or broadcasts on the condition variable.
- Another thread calls *PtExit()*, *exit()*, or *_exit()*.

PtCondTimedWait() is similar to *PtCondWait()*, but the blocking is limited by a timeout.

Non-Photon and Photon threads

The library keeps track of which of your threads are Photon threads (event readers) and which are non-Photon threads (nonreaders). This way, the library always knows how many of your threads are available to receive and process events. This information is currently used only by the *PtModalBlock()* function (see "Modal operations and threads," below).

By default, the thread that called *PtInit()* is an event reader, and any other thread isn't. But if a nonreader thread calls *PtProcessEvent()* or *PtMainLoop()*, it automatically becomes an event reader.

Photon doesn't start new threads for you if you run out of Photon threads.

You can also turn a nonreader into a reader and back by passing a flag to *PtEnter()* or *PtLeave()*:

Pt_EVENT_PROCESS_ALLOW
: Turn the calling thread into an event reader.

Pt_EVENT_PROCESS_PREVENT

Turn the calling thread into a nonreader.

If you don't need to change the thread's status (e.g. for a non-Photon thread that never processes any events), don't set either of these bits in the flags.

If you're calling *PtLeave()* in a callback because you're about to do something lengthy, pass Pt_EVENT_PROCESS_PREVENT to *PtLeave()*. This tells the library that this thread isn't going to process events for a significant amount of time. Make sure to pass Pt_EVENT_PROCESS_ALLOW to *PtEnter()* before returning from the callback.

Modal operations and threads

A modal operation is one where you need to wait until a particular event happens before you can proceed — for example, when you want the user to make a decision and push a Yes or a No button. Since other events usually arrive before the one you're waiting for, you need to make sure that they're processed.

In a single-threaded application, attach a callback to the Yes and No buttons. In this callback, call *PtModalUnblock()*. When you display the dialog, call *PtModalBlock()*. This function runs an event-processing loop similar to *PtMainLoop()*, except that *PtModalBlock()* returns when something (e.g. the callback attached to the Yes and No buttons) calls *PtModalUnblock()*.

In a multithreaded application, *PtModalBlock()* may either:

- Do the same as in a single-threaded application.

  Or:

- Block on a condition variable and let other Photon threads process events.

By default, *PtModalBlock()* uses a condition variable if you have any other Photon threads. This removes the thread from the pool of event-processing threads, but prevents a situation where starting a second modal operation in a thread that's running the event loop in *PtModalBlock()* makes it impossible for the first *PtModalBlock()* to return until after the second modal operation has completed.

In most applications, there's no chance of this happening; usually, you either don't want to allow another modal operation until the current one has completed, or you actually want the stacking behavior where the second modal operation prevents completion of the first one. For example, if the first modal operation is a file selector and the second one is an "Are you sure you want to overwrite this file?" question, you don't want to let the user dismiss the file selector before answering the question.

If you know that your application doesn't have two unrelated modal operations that may happen at the same time but can be completed in any order, you can pass Pt_EVENT_PROCESS_ALLOW to *PtModalBlock()*. This tells *PtModalBlock()* to run an event loop even if you have other Photon threads available, and may reduce the total number of Photon threads that your application needs.

Exiting a multithreaded program

Terminating a multithreaded application can be tricky; calling *exit()* makes all your threads just disappear, so you have to make sure that you don't exit while another thread is doing something that shouldn't be interrupted, such as saving a file.

Don't call *pthread_exit()* in any kind of callback (such as a widget callback, an input function, a work procedure, and so on). It is likely that the library code that invoked your callback needs to do some cleanup when the callback returns. If it doesn't return, your application may leak memory.

Remember that all callbacks are run by a thread that has locked the libraries.

In a Photon application, the library may call *PtExit()* when your application's last window is closed. If you don't want that to happen while a thread is doing something important, turn off Ph_WM_CLOSE in your base window's *Pt_ARG_WINDOW_MANAGED_FLAGS* resource and handle the close message yourself. You also need to find all the calls to *exit()* or *PtExit()* in your code and make sure that you don't exit until it's safe to exit. If a widget in your base window has a Done or Cancel type callback, you have to handle that, too.

The Photon library provides some mechanisms to make handling this type of situation easier and safer:

- There's a simple counter that tells *PtExit()* to block until the counter goes to zero.

  The functions that implement this counter, *PtPreventExit()* and *PtAllowExit()*, are not only thread-safe, but also realtime-friendly: they're guaranteed to run a bound amount of code and never generate priority inversion.

  This mechanism is considered relatively low-level and meant primarily for threads that don't have anything to do with Photon (perhaps temporarily — i.e. while in a *PtLeave()* - *PtEnter()* section of a Photon callback).

  The reason is that certain Photon calls that normally are blocking cause the calling thread to go to sleep (blocked indefinitely) if *PtExit()* is pending (otherwise *PtExit()* would potentially block for a long time). This also happens when a thread blocks *before* another thread calls *PtExit()*; the blocked thread stays blocked without returning from the blocking call. The sleeping threads behave as if the scheduler didn't give them any CPU cycles until the entire process terminates. This allows the thread(s) that called *PtPreventExit()* to finish their job as quickly as possible.

  The list of Photon calls that make their calling threads sleep after another thread has called *PtExit()* includes attempts to process events, do anything modal, block on a condvar using *PtCondWait()* or *PtCondTimedWait()*, calling *PtEnter()* or *PtLeave()*, and calling *PtExit()*.

- It may sometimes be difficult to make sure that your thread doesn't call any of those after calling *PtPreventExit()* — and if it does and stays blocked without having a chance to call *PtAllowExit()*, your process will lock up and you'll have to kill it manually.

To prevent such situations, there's a Pt_DELAY_EXIT flag that you can pass to *PtEnter()* and *PtLeave()*. Doing it not only prevents *PtEnter()* and *PtLeave()* from blocking indefinitely if another thread has called *PtExit()*, but also implicitly calls *PtPreventExit()*. If your thread is put to sleep by a "sleep inducing" call, the library knows to call *PtAllowExit()* for you. The only way to keep Pt_DELAY_EXIT turned on is by making sure that you don't call any of the "sleep inducing" calls and pass Pt_DELAY_EXIT to *PtEnter()* and *PtLeave()* each time you call them. The Pt_DELAY_EXIT flag makes your "save file" callback as simple as this:

```
my_callback( ... )
{
    PtLeave( Pt_DELAY_EXIT );
    save_file();  /* You're safe here... */
    PtEnter( 0 ); /* But this may never return
                     -- and that's OK! */
}
```

You still have to make sure that *save_file()* doesn't attempt any of the "sleep inducing" calls. In particular, you can't pop up a dialog with an error message if something goes wrong. If you want to pop up a dialog that will potentially sit on the screen for minutes or hours, you have to do it before calling *PtExit()*, for example, by using the *Pt_ARG_WINDOW_MANAGED_FLAGS* trick mentioned above.

The protection that Pt_DELAY_EXIT gives your thread is disabled not only when the thread is put to sleep, but also when it dies for any reason.

Threads and work procedures

Note the following concerning threads and work procedures:

- If you attach a work procedure and you have more than one reader thread, there's a very narrow window where the work procedure can be invoked right away instead of after you run out of events.
- Mixing threads and work procedures might cause a minor problem; if one of the other threads adds a workproc while another thread is already waiting for an event, the workproc might not be invoked until you receive an event.

Chapter 19

Raw Drawing and Animation

In this chapter. . .

PtRaw widget

The **Pg** routines in the Photon library are the lowest-level drawing functions. They're used by the widget library to draw the widgets. You can use the Pg functions in a Photon application, but your application has to:

- handle any interaction with the user
- determine when the drawing is damaged (for example, when it's uncovered, or when the user moves the window)
- repair the drawing whenever it's damaged.

You should use widgets whenever possible because they do all of the above themselves.

If your application must do its own drawing, you should use the **PtRaw** widget. It does the following:

1. Tells the application what has been damaged.
2. Flushes the draw buffer almost whenever necessary. (You should flush the buffer explicitly; for example, before a blitting operation. Blitting shifts a rectangular area of your drawing by some distance; you want your drawing to be up-to-date before this happens.)

To create a **PtRaw** widget in PhAB, click on its icon in the widget palette:

Position it where you want your drawing to appear.

You can provide various functions for the **PtRaw** widget; they're called in the order given below when the widget is realized, and are then called as necessary:

Pt_ARG_RAW_INIT_F
: An initialization function that's called before the widget's extent is calculated.

Pt_ARG_RAW_EXTENT_F
: If provided, calculates the widget's extent when the widget is moved or resized.

Pt_ARG_RAW_CALC_OPAQUE_F
: Calculates the raw widget's opacity tile list.

Pt_ARG_RAW_CONNECT_F
: Called as the last stage in realizing the widget, just before any required regions are created.

Pt_ARG_RAW_DRAW_F
: Does the drawing.

Most of the time you'll need to specify only the drawing function (see below). You can use PhAB's function editor (described in the Editing Resources and Callbacks in PhAB chapter) to edit these resources — but you must give the raw widget a unique instance name first. You can also set these resources in your application's code; for more information, see "Function resources" in the Manipulating Resources in Application Code chapter.

For information on **PtRaw**'s resources, see the Photon *Widget Reference*.

Raw drawing function

When you create a **PtRaw** widget in PhAB and edit its *Pt_ARG_RAW_DRAW_F* function, you'll see default code like this:

```
void my_raw_draw_fn( PtWidget_t *widget,
                     PhTile_t *damage )
{
    PtSuperClassDraw( PtBasic, widget, damage );
}
```

The call to *PtSuperClassDraw()* (described in the *Building Custom Widgets* guide) invokes **PtBasic**'s draw function, which draws the raw widget's borders, fills the widget, and so on, as specified by its resources. The raw widget can do all this by itself, but using *PtSuperClassDraw()* reduces the complexity of the raw drawing function.

There are several things to consider in the raw drawing function:

- You'll need to know the raw widget's canvas.
- The origin for the drawing primitives is the top left corner of the canvas of the raw widget's *parent*, not the raw widget itself. You'll need to translate the coordinates.
- The raw widget can draw beyond its canvas, but it's not a good idea. You should set up clipping in the drawing function.
- The drawing function is passed a list of damaged areas that can be used to speed up repairs.
- For raw widgets whose contents change dynamically, you can define a model that describes what to draw.

These are described below, followed by some examples of simple drawing functions.

Don’t call *PtBkgdHandlerProcess()* in a **`PtRaw`** widget’s drawing function.

Don’t change any other widget in any way (creating, destroying, setting resources, and so on) in a raw widget’s drawing function. It’s safe to get resources from other widgets.

Don’t call the drawing function directly from your program. Instead, damage the widget by calling *PtDamageWidget()*, and let the library call the drawing function.

Determining the raw widget canvas

You can determine the raw widget’s canvas by calling *PtCalcCanvas()* as follows:

```
PhRect_t  raw_canvas;

PtCalcCanvas (widget, &raw_canvas);
```

You’ll need this canvas to perform any required translations and clipping.

Translating coordinates

The origin for the drawing primitives is the upper left corner of the raw widget’s parent’s canvas. You’ll probably find it easier to use the upper left corner of the raw widget’s canvas as the origin.

Once you’ve determined the raw widget’s canvas, you can do one of the following:

- Add the coordinates of the upper left corner of the raw widget’s canvas to any coordinates passed to the drawing primitives. For example, to draw an ellipse centered at (80, 60) relative to the raw widget’s canvas:

  ```
  PhPoint_t    c1 = { 80, 60 };
  PhPoint_t    r = { 72, 52 };

  c1.x += raw_canvas.ul.x;
  c1.y += raw_canvas.ul.y;
  PgSetFillColor(Pg_YELLOW);
  PgDrawEllipse ( &c1, &r, Pg_DRAW_FILL );
  ```

  This is the preferred method.

- You can set the translation by calling *PgSetTranslation()*, passing to it the upper left corner of the raw widget’s canvas:

  ```
  PhPoint_t    c1 = { 80, 60 };
  PhPoint_t    r = { 72, 52 };

  PgSetTranslation (&raw_canvas.ul, Pg_RELATIVE);

  PgSetFillColor(Pg_YELLOW);
  PgDrawEllipse ( &c1, &r, Pg_DRAW_FILL );
  ```

Be sure to restore the old translation before leaving the raw widget's drawing function. Here's one way to do it:

```
/* Restore the translation by subtracting the
   coordinates of the raw widget's canvas. */

raw_canvas.ul.x *= -1;
raw_canvas.ul.y *= -1;
PgSetTranslation (&raw_canvas.ul, Pg_RELATIVE);
```

Clipping

As mentioned above, it's possible to draw beyond the raw widget's extent in its drawing function, but it's not a good thing to do:

- It can mess up the rest of your application's interface.
- If the raw drawing beyond the raw widget's extent is damaged but the raw widget itself isn't, the raw widget's drawing function isn't called, so the damage won't be repaired.

It's possible to write the drawing function so that clipping isn't needed, but it can make your code more complicated. For example, if you try to write text that extends beyond the raw widget's canvas, you might need to draw partial letters. You'll also have to consider what happens if the user changes the size of the raw widget.

It's much easier to use *PtClipAdd()* to set the clipping area to be the raw widget's canvas and let the graphics driver restrict the drawing:

```
PtClipAdd ( widget, &raw_canvas);
```

Before leaving the drawing function, call *PtClipRemove()* to reset the clipping area:

```
PtClipRemove ();
```

Using damage tiles

If your raw widget's drawing function takes a lot of time, you might not want to redraw the entire canvas when a small portion of it has been damaged. You can speed up the repairs by using the drawing function's *damage* argument.

The *damage* argument is a pointer to a linked list of **PhTile_t** structures (see the Photon *Library Reference*), each of which includes these members:

rect	A **PhRect_t** structure that defines the damaged area.
next	A pointer to the next tile in the list.

The damaged areas are relative to the raw widget's disjoint parent (usually a **PtWindow** widget). Use *PtWidgetOffset()* to obtain the offset.

If there's more than one tile in the linked list, the first one covers the entire area covered by the rest. Either use the first tile and ignore the rest, or ignore the first and use the rest:

```
void rawDrawFunction (PtWidget_t *widget,
                      PhTile_t *damage)
{
  if (damage->next != NULL) {

    /* If there's more than one tile, skip the first. */

    damage = damage->next;
  }

  while (damage != NULL) {

    /* Examine 'damage' to see if any drawing
       needs doing:
         damage->rect.ul.x, damage->rect.ul.y,
         damage->rect.lr.x, damage->rect.lr.y */

    ⋮
    damage = damage->next;  /* Go on to the next tile. */
  }
}
```

The following functions (described in the Photon *Library Reference*) work with tiles:

PhAddMergeTiles()	Merge two list tiles, eliminating overlap
PhClipTilings()	Clip one list of tiles from another
PhCoalesceTiles()	Combine a list of tiles
PhCopyTiles()	Copy a list of tiles
PhDeTranslateTiles()	Subtract x and y offsets from the vertices of a list of tiles
PhFreeTiles()	Return a list of tiles to the internal tile pool
PhGetTile()	Retrieve a tile from the internal tile pool
PhIntersectTilings()	Determine the intersection of two lists of tiles
PhMergeTiles()	Remove all overlap from a list of tiles
PhRectsToTiles()	Create a list of tiles from an array of rectangles
PhSortTiles()	Sort a list of tiles
PhTilesToRects()	Create an array of rectangles from a list of tiles
PhTranslateTiles()	Add x and y offsets to the vertices of a list of tiles

Using a model for more complex drawing

If the contents of the raw widget are static, you can call the Pg drawing primitives directly from the raw drawing function. If the contents are dynamic, you'll need to define a data structure or *model* that describes them.

The structure of the model depends on your application; the raw drawing function must be able to traverse the model and draw the required graphics. Use the raw widget's *Pt_ARG_USER_DATA* or *Pt_ARG_POINTER* resource to save a pointer to the model.

Examples of simple PtRaw drawing functions

This drawing function draws a couple of ellipses, one of which is clipped:

```
void my_raw_draw_fn( PtWidget_t *widget,
                     PhTile_t *damage )
{
   PhRect_t     raw_canvas;
   PhPoint_t    c1 = { 80, 60 };
   PhPoint_t    c2 = { 30, 210 };
   PhPoint_t    r = { 72, 52 };

   PtSuperClassDraw( PtBasic, widget, damage );
   PtCalcCanvas(widget, &raw_canvas);

   /* Set the clipping area to be the raw widget's
      canvas. */

   PtClipAdd ( widget, &raw_canvas);

   /* Draw the ellipses. */

   c1.x += raw_canvas.ul.x;
   c1.y += raw_canvas.ul.y;
   PgSetFillColor(Pg_YELLOW);
   PgDrawEllipse ( &c1, &r, Pg_DRAW_FILL );

   c2.x += raw_canvas.ul.x;
   c2.y += raw_canvas.ul.y;
   PgSetFillColor(Pg_RED);
   PgDrawEllipse ( &c2, &r, Pg_DRAW_FILL );

   /* Reset the clipping area. */

   PtClipRemove ();
}
```

This function is the same, but it sets the translation:

```
void my_raw_draw_fn( PtWidget_t *widget,
                     PhTile_t *damage )
{
   PhRect_t     raw_canvas;
   PhPoint_t    c1 = { 80, 60 };
   PhPoint_t    c2 = { 30, 210 };
   PhPoint_t    r = { 72, 52 };

   PtSuperClassDraw( PtBasic, widget, damage );
   PtCalcCanvas(widget, &raw_canvas);
```

```
    /* Set the clipping area to be the raw widget's
       canvas. */

    PtClipAdd ( widget, &raw_canvas);

    /* Set the translation so that drawing operations
       are relative to the raw widget's canvas. */

    PgSetTranslation (&raw_canvas.ul, Pg_RELATIVE);

    /* Draw the ellipses. */

    PgSetFillColor(Pg_YELLOW);
    PgDrawEllipse ( &c1, &r, Pg_DRAW_FILL );

    PgSetFillColor(Pg_RED);
    PgDrawEllipse ( &c2, &r, Pg_DRAW_FILL );

    /* Restore the translation by subtracting the
       coordinates of the raw widget's canvas. */

    raw_canvas.ul.x *= -1;
    raw_canvas.ul.y *= -1;
    PgSetTranslation (&raw_canvas.ul, Pg_RELATIVE);

    /* Reset the clipping area. */

    PtClipRemove ();
}
```

Color

Colors are specified in the Photon microGUI with the **`PgColor_t`** type. The library and graphics drivers interpret this data type according to the current color model (described in the documentation for **`PgColor_t`**).

The default color model, Pg_CM_PRGB, uses a 32-bit Red-Green-Blue (RGB) representation:

Reserved	Red	Green	Blue
`0000 0000`	`rrrr rrrr`	`gggg gggg`	`bbbb bbbb`

Macros for the most commonly used colors are defined in **`<photon/Pg.h>`**.

Although **`PgColor_t`** uses 32 bits, only 24 bits are used per color. This representation is called *true color*. The Photon microGUI is a true-color windowing system; it uses this 24-bit RGB representation internally.

Most graphics cards currently use true color (24 bits) or high color (16 bits). However, some graphics drivers take advantage of the palette on older palette-based cards.

The following datatypes and functions that deal with color are described in the Photon *Library Reference*:

PgAlphaValue()	Extract the alpha component from a color value
PgARGB()	Convert alpha, red, green, and blue values to composite color format
PgBackgroundShadings()	Calculate top and bottom shading colors
PgBlueValue()	Extract the blue component from a color value
PgCMY()	Convert cyan, magenta, and yellow values to composite color format
`PgColorHSV_t`	Hue-Saturation-Value color value
PgColorMatch()	Query for best color matches
PgGetColorModel()	Get the current color model
PgGetPalette()	Query for current color palette
PgGray()	Generate a shade of gray
PgGrayValue()	Extract color brightness
PgGreenValue()	Extract the green component from a color value
PgHSV()	Convert hue, saturation, and value to composite color format
PgHSV2RGB()	Convert HSV colors to RGB
PgRedValue()	Extract the red component from a color
PgRGB()	Convert red, green, and blue values to composite color format
PgRGB2HSV()	Convert RGB colors to HSV
PgSetColorModel()	Set the current color model
PgSetPalette()	Set the color palette

Drawing attributes

When doing raw drawing, you can set various attributes, including fonts, palettes, fill colors, line colors and styles, and text colors. The attributes that you set affect all raw drawing operations until you set them again. For example, the line color affects all lines, pixels, and bitmaps that you draw using the drawing primitives.

You don't need to set these attributes if you're using widgets; the drawing attributes are set based on the widgets' definitions and resources.

However, in all other cases you should set these attributes before you begin drawing. The defaults are undefined and drawing before setting the relevant attributes may produce unexpected results.

General attributes

The functions that set general drawing attributes are:

PgDefaultMode()	Reset draw mode and plane mask to their default values
PgSetDrawMode()	Set draw mode
PgSetPlaneMask()	Protect video memory from being modified

Text attributes

The text attributes affect all the text that you draw by calling the drawing primitives described in "Text," below. The functions that set text attributes are:

PgDefaultText()	Reset the text attribute to its system default
PgSetFont()	Set text font
PgSetTextColor()	Set text color
PgSetTextDither()	Set text dither pattern
PgSetTextTransPat()	Set draw transparency
PgSetTextXORColor()	Set a color for XOR drawing
PgSetUnderline()	Set colors for underlining text

Fill attributes

The fill attributes affect all the drawing that you do by calling the primitive functions described in

- Arcs, ellipses, polygons, and rectangles
- Text
- Bitmaps

The functions that set fill attributes are:

PgDefaultFill()	Reset the fill attribute to its default value
PgSetFillColor()	Set exact fill color
PgSetFillDither()	Set specific dither pattern and colors
PgSetFillTransPat()	Set draw transparency
PgSetFillXORColor()	Set a color for XOR drawing

Stroke (line) attributes

The stroke attributes affect all the drawing that you do by calling the primitive functions described in

- Arcs, ellipses, polygons, and rectangles
- Lines, pixels, and pixel arrays
- Text
- Bitmaps

PgDrawEllipse()* does not support stroke joins

The functions that set stroke attributes are:

PgDefaultStroke()	Reset the stroke attribute to its system default
PgSetStrokeCap()	Set what the ends of lines look like
PgSetStrokeColor()	Set the color of subsequent outlines
PgSetStrokeDither()	Apply a color pattern to outlines
PgSetStrokeTransPat()	Use a masking pattern to set draw transparency on outlines
PgSetStrokeXORColor()	Use the XOR of a color to draw outlines
PgSetStrokeDash()	Set dashed lines
PgSetStrokeWidth()	Set line thickness
PgSetStrokeFWidth()	Set line thickness

Arcs, ellipses, polygons, and rectangles

The Photon libraries include a number of primitive functions that you can use to draw shapes, including:

- rectangles
- rounded rectangles
- beveled boxes, rectangles, and arrows
- polygons
- arcs, circles, chords, and pies
- spans — complex shapes

Don't use these drawing primitives in an interface that uses widgets; widgets redisplay themselves when damaged, so anything drawn on top of them disappears. To display arcs, lines, etc. in an interface:

- Create a **`PtRaw`** widget and call the primitives in its draw function. See the section on the **`PtRaw`** widget earlier in this chapter.

  Or:

- Use the corresponding graphical widget. For more information, see **`PtGraphic`** in the Photon *Widget Reference*.

By using each primitive's *flags*, you can easily draw an outline (stroke), draw the filled "inside" (fill), or draw both as a filled outline.

The current fill and stroke attributes are used. For more information, see "Drawing attributes," earlier in this chapter.

To:	**Set *flags* to:**
Fill the primitive, using the current fill attributes	Pg_DRAW_FILL
Outline the primitive, using the current stroke attributes	Pg_DRAW_STROKE
Fill the primitive and outline it, using the current fill and stroke attributes	Pg_DRAW_FILL_STROKE

The **`mx`** versions of these functions place the address of the primitive into the draw buffer in your application's data space. When the draw buffer is flushed, the primitive is copied to the graphics driver. The non-**`mx`** versions copy the primitive itself into the draw buffer.

Rectangles

You can draw rectangles, using the current drawing attributes, by calling *PgDrawIRect()* or *PgDrawRect()*.

PgDrawRect() uses a **PhRect_t** structure (see the Photon *Library Reference*) for the rectangle coordinates, while *PgDrawIRect()* lets you specify the coordinates individually. Use whichever method you want.

The following example draws a rectangle that's filled, but not stroked (i.e. it has no border):

```
void DrawFillRect( void )
{
    PgSetFillColor( Pg_CYAN );
    PgDrawIRect( 8, 8, 152, 112, Pg_DRAW_FILL );
}
```

If you wish, you can call *PgDrawRect()* instead:

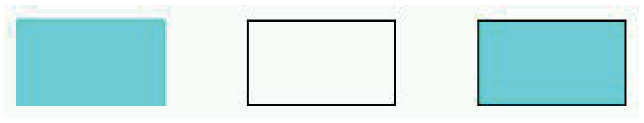
```
void DrawFillRect( void )
{
    PhRect_t rect = { {8, 8}, {152, 112} };

    PgSetFillColor( Pg_CYAN );
    PgDrawRect( &rect, Pg_DRAW_FILL );
}
```

The following example draws a stroked, unfilled rectangle:

```
void DrawStrokeRect( void )
{
    PgSetStrokeColor( Pg_BLACK );
    PgDrawIRect( 8, 8, 152, 112, Pg_DRAW_STROKE );
}
```

This code draw a stroked, filled rectangle:

```
void DrawFillStrokeRect( void )
{
    PgSetFillColor( Pg_CYAN );
    PgSetStrokeColor( Pg_BLACK );
    PgDrawIRect( 8, 8, 152, 112, Pg_DRAW_FILL_STROKE );
}
```



Filled and stroked rectangles.

Rounded rectangles

Rounded rectangles are programmed almost the same way as rectangles — just call *PgDrawRoundRect()* with a **PhPoint_t** parameter to indicate, in pixels, the roundness of the rectangle corners. The radii are truncated to the rectangle's sides.

The following example draws a black rounded rectangle with five pixels worth of rounding at the corners:

```
void DrawStrokeRoundRect( void )
{
    PhRect_t rect = { {20, 20}, {100, 100} };
    PhPoint_t radii = { 5, 5 };

    PgSetStrokeColor( Pg_BLACK );
    PgDrawRoundRect( &rect, &radii, Pg_DRAW_STROKE );
}
```

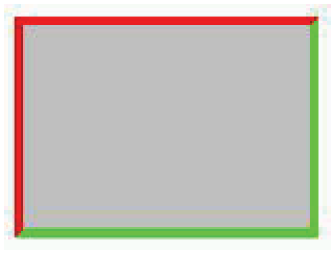
Beveled boxes, rectangles, and arrows

PgDrawBevelBox() draws a beveled box, which is a special type of rectangle:

- If you set Pg_DRAW_FILL or Pg_DRAW_FILL_STROKE in the *flags* argument, the area of the beveled box is filled according to the current fill attributes.
- If you set Pg_DRAW_STROKE OR Pg_DRAW_FILL_STROKE in the *flags*, the top and left edges are drawn according to the current stroke attributes, and the bottom and left edges are drawn with an extra color that's passed as one of the parameters.
- There's also a parameter to let you set the with or "depth" of the bevel.

This code draws a dark grey beveled box with a green and red bevel that's four pixels wide:

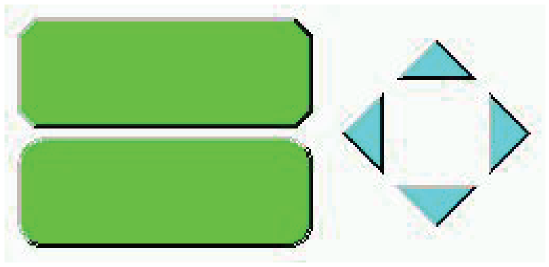
```
void DrawBevelBox( void )
{
    PhRect_t r = { 8, 8, 152, 112 };
    PgSetFillColor( Pg_DGREY );
    PgSetStrokeColor( Pg_RED );
    PgDrawBevelBox( &r, Pg_GREEN, 4, Pg_DRAW_FILL_STROKE );
}
```



A beveled box.

You can call *PgDrawBeveled()* to draw a beveled rectangle (optionally with clipped or rounded corners) or a beveled arrow. If you draw a rectangle with square corners, the results are the same as for *PgDrawBevelBox()*. Here's some code that draws clipped and rounded rectangles, and a set of arrows:

```
void DrawBeveled() {

    PhRect_t clipped_rect = { {10, 10}, {150, 62} };
    PhRect_t rounded_rect = { {10, 67}, {150, 119} };
    PhPoint_t clipping = { 8, 8 };
    PhPoint_t rounding = { 12, 12 };

    PhRect_t rup    = { {190, 20}, {230, 40} };
    PhRect_t rdown  = { {190, 90}, {230, 110} };
    PhRect_t rleft  = { {165, 45}, {185, 85} };
    PhRect_t rright = { {235, 45}, {255, 85} };

    /* Draw beveled rectangles: one clipped, one rounded. */

    PgSetFillColor( Pg_GREEN );
    PgSetStrokeColor( Pg_GREY );
    PgDrawBeveled( &clipped_rect, &clipping, Pg_BLACK, 2,
                   Pg_DRAW_FILL_STROKE | Pg_BEVEL_CLIP );
    PgDrawBeveled( &rounded_rect, &rounding, Pg_BLACK, 2,
                   Pg_DRAW_FILL_STROKE | Pg_BEVEL_ROUND );

    /* Draw beveled arrows. */

    PgSetFillColor( Pg_CYAN );
    PgSetStrokeColor( Pg_GREY );
    PgDrawBeveled( &rup, NULL, Pg_BLACK, 2,
                   Pg_DRAW_FILL_STROKE | Pg_BEVEL_AUP );
    PgDrawBeveled( &rdown, NULL, Pg_BLACK, 2,
                   Pg_DRAW_FILL_STROKE | Pg_BEVEL_ADOWN );
    PgDrawBeveled( &rleft, NULL, Pg_BLACK, 2,
                   Pg_DRAW_FILL_STROKE | Pg_BEVEL_ALEFT );
    PgDrawBeveled( &rright, NULL, Pg_BLACK, 2,
                   Pg_DRAW_FILL_STROKE | Pg_BEVEL_ARIGHT );
}
```



Beveled rectangles and arrows.

If you want to draw an arrow that fits inside a given rectangle (for example, the arrow for a scrollbar), call *PgDrawArrow()*.

Polygons

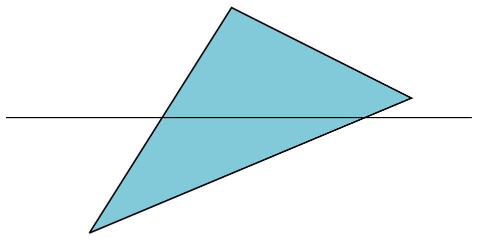
You can create polygons by specifying an array of **`PhPoint_t`** points. If you use Pg_CLOSED as part of the *flags*, the last point is automatically connected to the first point, closing the polygon. You can also specify points relative to the first point (using Pg_POLY_RELATIVE).

The following example draws a blue-filled hexagon with a white outline:

```
void DrawFillStrokePoly( void )
{
    PhPoint_t start_point = { 0, 0 };
    int num_points = 6;
    PhPoint_t points[6] = {
        { 32,21 }, { 50,30 }, { 50,50 },
        { 32,59 }, { 15,50 }, { 15,30 }
    };

    PgSetFillColor( Pg_BLUE );
    PgSetStrokeColor( Pg_WHITE );
    PgDrawPolygon( points, num_points, start_point,
        Pg_DRAW_FILL_STROKE | Pg_CLOSED );
}
```

Overlapping polygons

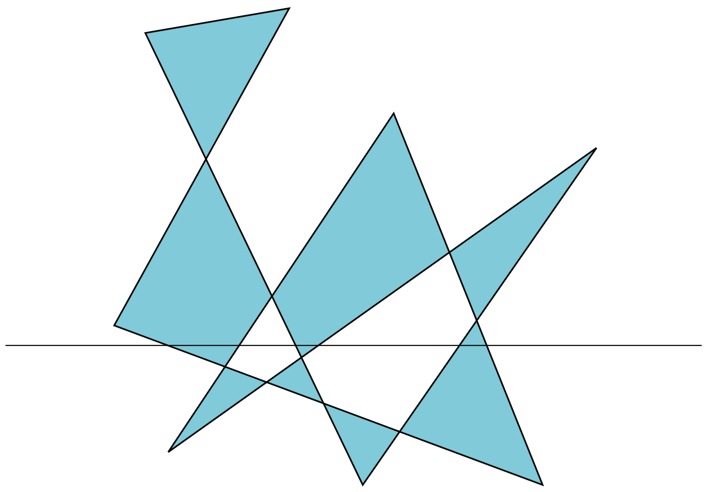
Polygons that overlap themselves are filled using the so-called *even-odd* rule: if an area overlaps an odd number of times, it isn't filled. Another way of looking at this is to draw a horizontal line across the polygon. As you travel along this line and cross the first line, you're inside the polygon; as you cross the second line, you're outside. As an example, consider a simple polygon:



Filling a simple polygon.

This rule can be extended for more complicated polygons:

- When you cross an odd number of lines, you're inside the polygon, so the area is filled.
- When you cross an even number of lines, you're outside the polygon, so the area isn't filled



Filling an overlapping polygon.

The even-odd rule applies to both the *PgDrawPolygon()* and *PgDrawPolygonmx()* functions.

Arcs, circles, chords, and pies

The *PgDrawArc()* function can be used for drawing:

- arcs
- circles
- ellipses
- elliptical arcs
- chords
- pie slices

You can also call *PgDrawEllipse()* to draw an ellipse.

The start and end angles of arc segments are specified in binary gradations (bi-grads), with 65536 bi-grads in a complete circle.

To draw a full circle or ellipse, specify the same value in bi-grads for the start and end angles. For example:

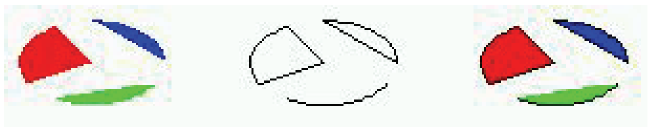
```
void DrawFullCurves( void )
{
    PhPoint_t circle_center  = { 150, 150 },
              ellipse_center = { 150, 300 };
    PhPoint_t circle_radii  = { 100, 100 },
              ellipse_radii = { 100, 50 };
    /* Draw a white, unfilled circle. */
    PgSetStrokeColor( Pg_WHITE );
    PgDrawArc( &circle_center, &circle_radii, 0, 0,
        Pg_DRAW_STROKE | Pg_ARC );

    /* Draw an ellipse with a white outline, filled
       with black. */
    PgSetFillColor( Pg_BLACK );
    PgDrawArc( &ellipse_center, &ellipse_radii, 0, 0,
        Pg_DRAW_FILL_STROKE | Pg_ARC );
}
```

To draw a chord (a curve with the end points connected by a straight line), add Pg_ARC_CHORD to the *flags* parameter. For example:

```
void DrawChord( void )
{
    PhPoint_t center  = { 150, 150 };
    PhPoint_t radii   = { 100, 50 };

    /* Draw an elliptical chord with a white outline,
       filled with black.  The arc is drawn from 0 degrees
       through to 45 degrees (0x2000 bi-grads). */

    PgSetStrokeColor( Pg_WHITE );
    PgSetFillColor( Pg_BLACK );
    PgDrawArc( &center, &radii, 0, 0x2000,
        Pg_DRAW_FILL_STROKE | Pg_ARC_CHORD );
}
```

Similarly, to draw a pie section or curve, add Pg_ARC_PIE or Pg_ARC to the *flags*. For example:

```
void DrawPieCurve( void )
{
    PhPoint_t pie_center = { 150, 150 },
              arc_center = { 150, 300 };
    PhPoint_t pie_radii  = { 100, 50 },
              arc_radii  = { 50, 100 };

    /* Draw an elliptical pie with a white outline,
       filled with black. */

    PgSetStrokeColor( Pg_WHITE );
    PgSetFillColor( Pg_BLACK );
    PgDrawArc( &pie_center, &pie_radii, 0, 0x2000,
        Pg_DRAW_FILL_STROKE | Pg_ARC_PIE );

    /* Draw a black arc. */
    PgSetStrokeColor( Pg_BLACK );
    PgDrawArc( &arc_center, &arc_radii, 0, 0x2000,
        Pg_DRAW_STROKE | Pg_ARC );
}
```



Filled and stroked arcs.

Spans — complex shapes

If the shape you want to draw can't be expressed as any of the other shapes that the Photon microGUI supports, you can draw it as a series of *spans* by calling *PgDrawSpan()*.

This function takes as one of its arguments an array of **`PgSpan_t`** records. The members are:

`short` *x1*	Starting x position.
`short` *x2*	Last x position.
`short` *y*	Y position.

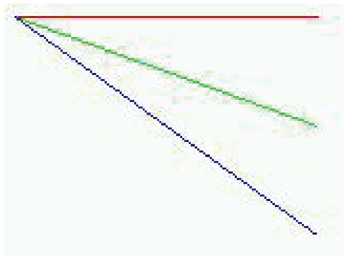
Lines, pixels, and pixel arrays

Lines and pixels are drawn using the current stroke state (color, thickness, etc.). The drawing primitives are:

PgDrawBezier(), *PgDrawBeziermx()*	Draw a stroked and/or filled Bézier curve
PgDrawGrid()	Draw a grid
PgDrawLine(), *PgDrawILine()*	Draw a single line
PgDrawPixel(), *PgDrawIPixel()*	Draw a single point
PgDrawPixelArray(), *PgDrawPixelArraymx()*	Draw multiple points
PgDrawTrend(), *PgDrawTrendmx()*	Draw a trend graph

The following example draws red, green, and blue lines:

```
void DrawLines( void )
{
    PgSetStrokeColor( Pg_RED );
    PgDrawILine( 8, 8, 152, 8 );
    PgSetStrokeColor( Pg_GREEN );
    PgDrawILine( 8, 8, 152, 60 );
    PgSetStrokeColor( Pg_BLUE );
    PgDrawILine( 8, 8, 152, 112 );
}
```



Lines created by the drawing primitives.

Text

There are various routines that draw text, depending on your requirements:

PgDrawMultiTextArea()	Draw multiline text in an area
PgDrawString(), *PgDrawStringmx()*	Draw a string of characters
PgDrawText(), *PgDrawTextmx()*	Draw text
PgDrawTextArea()	Draw text within an area
PgDrawTextChars()	Draw the specified number of text characters
PgExtentMultiText()	Calculate the extent of a multiline text string
PgExtentText()	Calculate the extent of a string of text

Text is drawn using the current text attributes; for more information, see "Text attributes," above. If you set *flags* to Pg_BACK_FILL, the text's extent is filled according to the current fill attributes (see "Fill attributes"). If you define an underline with *PgSetUnderline()*, the underline is drawn under the text and on top of the backfill.

For example, to print black text in 18-point Helvetica:

```
void DrawSimpleText( void )
{
    char *s = "Hello World!";
    PhPoint_t p = { 8, 30 };
    char Helvetica18[MAX_FONT_TAG];

    if(PfGenerateFontName("Helvetica", 0, 18,
                          Helvetica18) == NULL) {
        perror("Unable to generate font name");
    } else {
        PgSetFont( Helvetica18 );
    }
    PgSetTextColor( Pg_BLACK );
    PgDrawText( s, strlen( s ), &p, 0 );
}
```

To print black text on a cyan background:

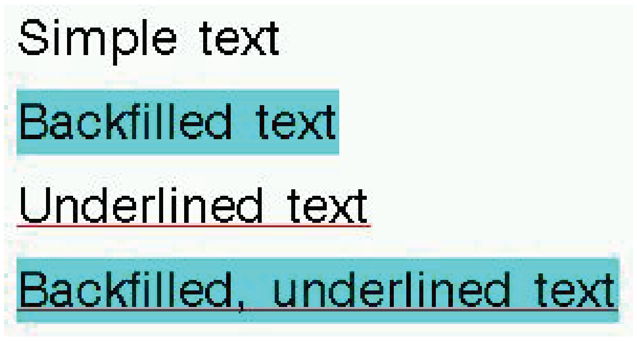
```
void DrawBackFillText( void )
{
    char *s = "Hello World!";
    PhPoint_t p = { 8, 30 };
    char Helvetica18[MAX_FONT_TAG];

    if(PfGenerateFontName("Helvetica", 0, 18,
                          Helvetica18) == NULL) {
        perror("Unable to generate font name");
    } else {
        PgSetFont( Helvetica18 );
    }
    PgSetTextColor( Pg_BLACK );
    PgSetFillColor( Pg_CYAN );
    PgDrawText( s, strlen( s ), &p, Pg_BACK_FILL );
}
```

To print black text with a red underline:

```
void DrawUnderlineText( void )
{
    char *s = "Hello World!";
    PhPoint_t p = { 8, 30 };
    char Helvetica18[MAX_FONT_TAG];

    if(PfGenerateFontName("Helvetica", 0, 18,
                          Helvetica18) == NULL) {
        perror("Unable to generate font name");
    } else {
        PgSetFont( Helvetica18 );
    }
    PgSetTextColor( Pg_BLACK );
    PgSetUnderline( Pg_RED, Pg_TRANSPARENT, 0 );
    PgDrawText( s, strlen( s ), &p, 0 );
    PgSetUnderline( Pg_TRANSPARENT, Pg_TRANSPARENT, 0 );
}
```

To print black text with a red underline on a cyan background:

```
void DrawBackFillUnderlineText( void )
{
    char *s = "Hello World!";
    PhPoint_t p = { 8, 30 };
    char Helvetica18[MAX_FONT_TAG];

    if(PfGenerateFontName("Helvetica", 0, 18,
                          Helvetica18) == NULL) {
        perror("Unable to generate font name");
    } else {
        PgSetFont( Helvetica18 );
    }
    PgSetTextColor( Pg_BLACK );
    PgSetFillColor( Pg_CYAN );
    PgSetUnderline( Pg_RED, Pg_TRANSPARENT, 0 );
    PgDrawText( s, strlen( s ), &p, Pg_BACK_FILL );
    PgSetUnderline( Pg_TRANSPARENT, Pg_TRANSPARENT, 0 );
}
```



Text created by the drawing primitives.

Bitmaps

Bitmaps are drawn using the current text state. If you set *flags* to Pg_BACK_FILL, the blank pixels in the image are drawn using the current fill state. The drawing primitives for bitmaps are:

PgDrawBitmap(), PgDrawBitmapmx()

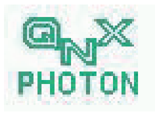
Draw a bitmap

PgDrawRepBitmap(), PgDrawRepBitmapmx()

Draw a bitmap several times

This example draws the bitmap with a transparent background:

```
void DrawSimpleBitmap( void )
{
    PhPoint_t p = { 8, 8 };

    PgSetTextColor( Pg_CELIDON );
    PgDrawBitmap( TestBitmap, 0, &p, &TestBitmapSize,
                  TestBitmapBPL, 0 );
}
```



A bitmap with a transparent background.

This example draws the bitmap against a yellow background:

```
void DrawBackFillBitmap( void )
{
    PhPoint_t p = { 8, 8 };

    PgSetTextColor( Pg_CELIDON );
    PgSetFillColor( Pg_YELLOW );
    PgDrawBitmap( TestBitmap, Pg_BACK_FILL, &p,
                  &TestBitmapSize, TestBitmapBPL, 0 );
}
```

A backfilled bitmap.

Images

The Photon microGUI supports these main types of images:

direct color	Consisting of: • image data — a matrix of colors (but not necessarily of type **PgColor_t**). Each element in the matrix is the color of a single pixel. Direct-color images have a type that starts with **Pg_IMAGE_DIRECT_**.
palette-based	Consisting of: • a palette — an array of type **PgColor_t** • image data — a matrix whose elements are offsets into the palette. Palette-based images have a type that starts with **Pg_IMAGE_PALETTE_**.
gradient color	Colors are algorithmically generated as a gradient between two given colors.

You can define any image by its pixel size, bytes per line, image data, and format. Images can be stored in structures of type **PhImage_t** (described in the Photon *Library Reference*). The *type* field of this data structure identifies the type of image.

Palette-based images

Palette-based images provide a fast, compact method for drawing images. Before drawing a palette-based image, you must set either a *hard palette* or *soft palette* to define the colors for the image.

Setting a hard palette changes the physical palette. All colors set with a *PgSetFillColor()* function are chosen from this palette. Other processes continue to choose colors from the Photon microGUI's *global palette* and may appear incorrect.

When you release the hard palette, the other processes return to normal without being redrawn. You should always release the hard palette when your window loses focus.

Setting a soft palette lets you redefine how colors are interpreted for the given draw context *without* changing the physical palette. All colors in the soft palette are mapped to the physical palette.

If your physical palette uses more colors than your graphics card supports, some colors are dropped, and the image won't look as nice.

The image data (either bytes or nibbles) is an index into the current palette. For example:

```
PgColor_t   ImagePalette[256];
char       *ImageData;
PhPoint_t   ImageSize;
int         ImageBPL;

void DrawYourImage( PhPoint_t pos )
{
     PgSetPalette( ImagePalette, 0, 0, 256,
                   Pg_PALSET_SOFT );
     PgDrawImage( ImageData, Pg_IMAGE_PALETTE_BYTE, pos,
         ImageSize, ImageBPL, 0 );
}
```

Direct-color images

With direct-color images, every pixel can be any color. But compared to palette-based images, the image data is larger and the image may take longer to draw. You can choose from several types of direct-color images, listed in the description of **PhImage_t** in the Photon *Library Reference*; each has a different image-pixel size and color accuracy.

Gradient-color images

With gradient-color images, colors are algorithmically generated as a gradient between two given colors.

Creating images

To create a **PhImage_t** structure:

- Call *PhCreateImage()*

  Or:

- Call *PxLoadImage()* to load an image from disk.

  Or:

- Call *ApGetImageRes()* to load an image from a PhAB widget database.

  Or:

- Get the value of the *Pt_ARG_LABEL_IMAGE* resource of a **PtLabel** or **PtButton** widget (provided the widget's *Pt_ARG_LABEL_TYPE* is Pt_IMAGE or Pt_TEXT_IMAGE).

  Or:

- Allocate it and fill in the members by hand.

It's better to call *PhCreateImage()* than to allocate the structure and fill it in by hand. Not only does *PhCreateImage()* provide the convenience of setting up a blank image, but it also observes the restrictions that the graphics drivers impose on image alignment, and so on.

Caching images

The *image_tag* and *palette_tag* members of the **PhImage_t** structure are used for caching images when dealing with remote processes via **phrelay** (see the QNX Neutrino *Utilities Reference*) for example, when using **phindows**.

These tags are cyclic-redundancy checks (CRCs) for the image data and the palette, and can be computed by *PtCRC()* or *PtCRCValue()* If these tags are nonzero, **phindows** and **phditto** cache the images. Before sending an image, **phrelay** sends its tag. If **phindows** finds the same tag in its cache, it uses the image in the cache. This scheme reduces the amount of data transmitted.

You don't need to fill in the tags if the images don't need to be saved in the cache. For example, set the tags to 0 if you're displaying animation by displaying images, and the images never repeat.

PxLoadImage() and *ApGetImageRes()* set the tags automatically. PhAB generates the tags for any images generated through it (for example, in the pixmap editor).

Transparency in images

If you want parts of an image to be transparent, you can:

- Use a chroma key.

  Or:

- Create a *transparency mask* for the image.

Chroma is accelerated by most hardware, whereas transparency bitmaps are always implemented in software.

Using chroma

To make a given color transparent in an image, using chroma if possible, call *PhMakeTransparent()*, passing it the image and the RGB color that you want to be made transparent.

Using a transparency mask

The transparency mask is stored in the *mask_bm* member of the **`PhImage_t`** structure. It's a bitmap that corresponds to the image data; each bit represents a pixel:

If the bit is:	The corresponding pixel is:
0	Transparent
1	Whatever color is specified in the image data

The *mask_bpl* member of the **`PhImage_t`** structure specifies the number of bytes per line for the transparency mask.

You can build a transparency mask by calling *PhMakeTransBitmap()*.

If you use *PxLoadImage()* to load a transparent image, set PX_TRANSPARENT in the *flags* member of the **`PxMethods_t`** structure. If you do this, the function automatically makes the image transparent; you don't need to create a transparency mask.

Displaying images

There are various ways to display an image:

- If the image is stored in a **`PhImage_t`** structure, call *PgDrawPhImage()* or *PgDrawPhImagemx()*. These functions automatically handle chroma key, alpha operations, ghosting, transparency, and so on.

  To draw the image repeatedly, call *PgDrawRepPhImage()* or *PgDrawRepPhImagemx()*.

  To draw a rectangular portion of the image, call *PgDrawPhImageRectmx()*.

- If the image isn't stored in a **`PhImage_t`** data structure, call *PgDrawImage()* or *PgDrawImagemx()*.

  To draw the image repeatedly, call *PgDrawRepImage()* or *PgDrawRepImagemx()*

- If the image isn't stored in a **`PhImage_t`** structure and has a transparency mask, call *PgDrawTImage()* or *PgDrawTImagemx()*.

- Set the *Pt_ARG_LABEL_IMAGE* resource for a **`PtLabel`** or **`PtButton`** widget (which use *PgDrawPhImagemx()* internally). The widget's *Pt_ARG_LABEL_TYPE* must be Pt_IMAGE or Pt_TEXT_IMAGE.

The **`mx`** versions of these functions place the address of the image into the draw buffer in your application's data space. When the draw buffer is flushed, the entire image is copied to the graphics driver. The non-**`mx`** versions copy the image itself into the draw buffer.

You can speed up the drawing by using shared memory. Call *PgShmemCreate()* to allocate the image data buffer:

```
my_image->image = PgShmemCreate( size, NULL );
```

If you do this, the image data isn't copied to the graphics driver.

The images created and returned by *ApGetImageRes()* and *PxLoadImage()* aren't in shared memory.

Manipulating images

The following functions let you manipulate images:

PiConvertImage()	Convert an image to another type
PiCropImage()	Crop an image to the specified boundary
PiDuplicateImage()	Duplicate an image
PiFlipImage()	Flip all or part of an image
PiGetPixel()	Retrieve the value of a pixel within an image
PiGetPixelFromData()	Retrieve a value from a run of pixels
PiGetPixelRGB()	Retrieve the RGB value of a pixel within an image
PiResizeImage()	Resize an image
PiSetPixel()	Alter the value of a pixel within an image
PiSetPixelInData()	Set the value of a pixel in a run of pixels
PxRotateImage()	Rotate an image

Releasing images

The **`PhImage_t`** structure includes a *flags* member that can make it easier to release the memory used by an image. These flags indicate which members you would like to release:

- Ph_RELEASE_IMAGE
- Ph_RELEASE_PALETTE
- Ph_RELEASE_TRANSPARENCY_MASK
- Ph_RELEASE_GHOST_BITMAP

Calling *PhReleaseImage()* with an image frees any resources that have the corresponding bit set in the image flags.

- *PhReleaseImage()* doesn't free the **`PhImage_t`** structure itself, just the allocated members of it.
- *PhReleaseImage()* correctly frees memory allocated with *PgShmemCreate()*.

The *flags* for images created by *ApGetImageRes()*, *PiCropImage()*, *PiDuplicateImage()*, *PiFlipImage()*, and *PxLoadImage()* aren't set. If you want *PhReleaseImage()* to free the allocated members, you'll have to set the flags yourself:

```
my_image->flags = Ph_RELEASE_IMAGE |
                 Ph_RELEASE_PALETTE |
                 Ph_RELEASE_TRANSPARENCY_MASK |
                 Ph_RELEASE_GHOST_BITMAP;
```

When should you set the release flags? When you know that the image is referred to only by one entity. For example, if one widget will be using an image, then it should free the image once it's done with it. If you set the release flags appropriately, prior to setting the image resource, then this will happen automatically — that is, the widget will free the image and data when it's destroyed, or you apply a new setting for the resource.

If multiple widgets use the same image (they have their own copies of the image structure but share the data to conserve memory), then you need to be a little more clever and make sure the image is freed only when all the widgets are done with it, and never before. There are a number of ways to accomplish this. For example, you could:

- Release the image in a *Pt_CB_DESTROYED*, but you would need to be sure that no other widgets are using it. If you know that one widget will survive the rest, then release the image in its *Pt_CB_DESTROYED*. Otherwise you need a more sophisticated approach, like your own reference count.
- Alternatively, if you know one widget will outlast all the others using the image, then set the release flags in the structure prior to setting the image resource of that widget. All the rest should have the flags clear. Note that if you change the image resource on that widget, however, the image will be freed, thus invalidating all the other widgets' references to it!

The approach you take will depend on your situation and requirements.

If the image is stored in a widget, the allocated members of images are automatically freed when an new image is specified or the widget is destroyed, provided that the appropriate bits in the *flags* member of the **`PhImage_t`** structure are set before the image is added to the widget.

Animation

This section describes how you can create simple animation. There are two basic steps:

- creating a series of "snapshots" of the object in motion
- cycling through the snapshots

It's better to use images for animation than bitmaps, as images aren't transparent (provided you haven't created a transparency mask). This means that the background doesn't need to be redrawn when replacing one image with another. As a result, there's no flicker when you use images. For other methods of eliminating flicker, see "Flickerless animation", below.

It's also possible to create animation by using a **PtRaw** widget and the Photon drawing primitives. See "**PtRaw** widget", earlier in this chapter.

Creating a series of snapshots

To animate an image you'll need a series of snapshots of it in motion. For example, you can use a **PtLabel** widget (with a *Pt_ARG_LABEL_TYPE* of Pt_IMAGE or Pt_TEXT_IMAGE) for animation. Create one **PtLabel** widget where you want the animation to appear, and create another **PtLabel** widget for each snapshot. You can store these snapshots in a widget database or a file.

Using a widget database

As described in "Widget databases" in the Accessing PhAB Modules from Code chapter, you can use a picture module as a widget database. To use one for animation, do the following in PhAB:

1. Create a picture module to use as widget database.
2. Create an internal link to the picture module.
3. Create the snapshots of the object in motion. Use the same widget type as you use where the animation is to appear. Give each snapshot a unique instance name.

In your application's initialization function, open the database by calling *ApOpenDBase()* or *ApOpenDBaseFile()*. Then, load the images with the *ApGetImageRes()* function. For example:

```
/* global data */
PhImage_t *images[4];
ApDBase_t *database;
int cur_image = -1,
    num_images = 4;

int
app_init( int argc, char *argv[])
```

```
{
  int       i;
  char      image_name[15];

  /* eliminate 'unreferenced' warnings */
  argc =  argc, argv = argv;

  database = ApOpenDBase (ABM_image_db);

  for (i = 0; i < num_images; i++)
  {
    sprintf (image_name, "image%d", i);
    images[i] = ApGetImageRes (database, image_name);
  }

  return (PT_CONTINUE);
}
```

ApGetImageRes() returns a pointer into the widget database. Don't close the database while you're still using the images in it.

Using a file

You can also load the snapshots from a file into a **PhImage_t** structure, by using the *PxLoadImage()* function. This function supports a number of formats, including GIF, PCX, JPG, BMP, and PNG. For a complete list, see **/usr/photon/dll/pi_io_***.

Cycling through the snapshots

No matter where you get the images, the animation is the same:

1. Create a **PtTimer** widget in your application. PhAB displays it as a black box; it won't appear when you run your application.
2. Specify the initial (*Pt_ARG_TIMER_INITIAL*) and repeat (*Pt_ARG_TIMER_REPEAT*) timer intervals.
3. Create an activate (*Pt_CB_TIMER_ACTIVATE*) callback for the timer. In the callback, determine the next image to be displayed, and copy it into the destination widget.

For example, the callback for the timer could be as follows:

```
/* Display the next image for our animation example.     */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>
```

```
/* Local headers */
#include "globals.h"
#include "abimport.h"
#include "proto.h"

int
display_image( PtWidget_t *widget,
               ApInfo_t *apinfo,
               PtCallbackInfo_t *cbinfo )

    {

    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo, cbinfo = cbinfo;

    cur_image++;
    if (cur_image >= num_images)
    {
       cur_image=0;
    }

    PtSetResource (ABW_base_image, Pt_ARG_LABEL_IMAGE,
                   images[cur_image], 0 );
    PtFlush ();

    return( Pt_CONTINUE );

    }
```

ABW_base_image is the widget name of the **PtLabel** widget in which the animation appears.

Flickerless animation

There are two ways to eliminate flicker in animation:

- Create a **PtOSContainer** (an offscreen-context container) and make it the parent of all the widgets in the area being animated.

  Or:

- Use the *PmMem...()* memory-context functions to build the image in memory, and display it when complete.

PtOSContainer

When you do animation in a child of an offscreen-context container, the **PtOSContainer** renders the draw stream into offscreen video memory, taking advantage of any hardware-acceleration features that the graphics driver supports. The graphics hardware can then blit the image directly onto the screen, resulting in flicker-free widgets and/or animation.

PtRaw (like any other widget) can be a child of **PtOSContainer**. This means that you can have flicker-free animation even when using the Photon drawing primitives.

Memory-context functions

You can call these functions to use a memory context to reduce flickering:

PmMemCreateMC()	Create a memory context
PmMemFlush()	Flush a memory context to its bitmap
PmMemReleaseMC()	Release a memory context
PmMemSetChunkSize()	Set the increment for growing a memory context's draw buffer
PmMemSetMaxBufSize()	Set the maximum size of a memory context's draw buffer
PmMemSetType()	Set the type of a memory context
PmMemStart()	Make a memory context active
PmMemStop()	Deactivate a memory context

Start by creating a memory context:

```
PmMemoryContext_t * PmMemCreateMC(
                        PhImage_t *image,
                        PhDim_t *dim,
                        PhPoint_t *translation );
```

The *image* structure must at least specify the *type* and *size* members. The image data buffer is optional, but if you want it in shared memory, you must provide it. The image *type* must be either Pg_IMAGE_DIRECT_888 or Pg_IMAGE_PALETTE_BYTE.

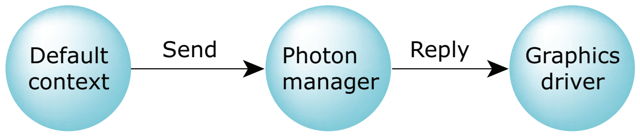
Once you've created the memory context:

- Call *PmMemStart()* to set the current draw context to the memory context.
- Call *PmMemStop()* when finished your drawing, to return to the default draw context.
- Call *PmMemFlush()* to get the resulting image.

When you no longer need the memory context, call *PmMemReleaseMC()*.

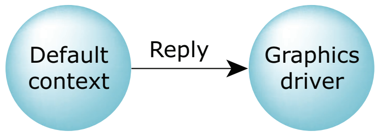
Direct mode

In normal (nondirect) mode, an application sends drawing requests to the Photon manager. The graphics driver blocks on the Photon manager.



Communication in normal (nondirect) mode.

When an application enters direct mode, it's requesting that the graphics driver receive draw streams and service messages directly from the application, instead of from the Photon manager. The driver blocks on the application, which is now responsible for telling the graphics driver what to do.



Communication in direct mode.

While in direct mode, the application has complete control over the display, since no other applications are able to be serviced by the graphics driver. The graphics driver's region is also no longer sensitive to draw events (this way the Photon manager discards all other applications' requests for rendering services to this driver). The other benefit with this mode is that graphical services are no longer sent through the Photon event space, so performance is improved. The drawback for this mode is that applications that expect to capture draw events can't record the application's view.

For convenience, a new context type, called a **`PdDirectContext_t`**, has been created. This context, when activated, becomes the default context for the application, so all other Photon *Pg** calls work correctly while in this mode.

While in this mode, the origin of all drawing operations is the upper left corner of the display, since the requests are no longer clipped or translated by the Photon event space. Your application can still translate and clip the events by calling *PgSetTranslation()* and *PgSetClipping()* if necessary.

The following functions deal with direct mode:

PdCreateDirectContext()	Create a direct-mode context
PdDirectStart()	Enter direct mode
PdDirectStop()	Leave direct mode
PdGetDevices()	Get region IDs for the currently available draw devices

PdReleaseDirectContext()	Leave direct mode and release the direct-mode context
PdSetTargetDevice()	Set the target device
PgWaitVSync()	Wait for vertical synchronization

Here are some things to keep in mind:

- When you enter or leave direct mode, all video RAM contexts (except the display), are destroyed on the driver side (an OSINFO event is shot out by the driver so applications are notified and can reinitialize any video memory contexts). This includes video RAM used by **`PdOffscreenContext_t`** structures and anything used by the video overlay API.
- When you leave direct mode, an expose event is also sent out by the driver, so all other applications redraw themselves.
- When you're in direct mode, the graphics driver region is no longer sensitive to draw events (so the Photon manager doesn't build up a massive list of draw events to be processed from other applications).
- If you have automatic double buffering turned on (e.g. **`devg-banshee -B ...`**), it's turned off while you're in direct mode (to let applications control the double buffering themselves).

Example

Here's how to get the address of any video memory context (including the display, which is considered to be one).

If you create a direct context by calling *PdCreateDirectContext()*, and then enter direct mode by calling *PdDirectStart()*, your application "owns" the graphics driver (*PgFlush()* goes to the video driver directly, instead of to the Photon server). You don't need to be in direct mode to get a pointer to offscreen RAM, but you do need to be to get a pointer to the primary display.

Here's some pseudo-code:

```
/* Create the direct context. */
direct_context = PdCreateDirectContext();

/* Start Direct Mode. */
PdDirectStart(direct_context);

/* Get the primary display. */
primary_display = PdCreateOffscreenContext( 0, 0, 0,
                                            Pg_OSC_MAIN_DISPLAY);

/* Get a pointer to the display. */
vidptr = PdGetOffscreenContextPtr(primary_display);
```

```
/* Make sure the Photon driver isn't doing anything
   (it shouldn't be at this point but this is just to
   be sure that we haven't gotten ahead of the video
   card's draw engine). */
PgWaitHWIdle();

/* Do what ever you do to the memory. */
Do_something(vidptr);

/* Leave direct mode, and destroy the direct context
   (an alternative would be PdDirectStop if you don't
   want to destroy the context). */
PdReleaseDirectContext(direct_context);
```

Video memory offscreen

These API calls allow you to use the leftover memory on a video card. When a video card is in a video mode, there's usually video RAM leftover that isn't being used by the display area. These areas of RAM can be used to do a variety of graphical operations while still using the accelerator on the video card. They're treated in the Photon microGUI basically the same way that a memory context is used, but should be much faster because there's hardware acceleration for these areas.

The functions and data structures include:

PdCreateOffscreenContext()	Create an offscreen context in video RAM
PdDupOffscreenContext()	Duplicate an offscreen context
PdGetOffscreenContextPtr()	Create a shared memory object reference to an offscreen context
`PdOffscreenContext_t`	Data structure that describes an offscreen context
PdSetOffscreenTranslation()	Set the translation for an offscreen context
PdSetTargetDevice()	Set the target device
PgContextBlit()	Copy data from a rectangle in one context to another context
PgContextBlitArea()	Copy data from an area in one context to another context
PgSwapDisplay()	Point the CRT of the video display at a given context
PgWaitHWIdle()	Wait until the video driver is idle

PhDCRelease()	Release a draw context

Here's an example that loads an image, creates an offscreen context for the image, and then blits the image data to the screen. It creates a window that contains a PtRaw, and uses *PgContextBlit()* in the PtRaw's raw drawing callback to redraw the image whenever the window is damaged or resized. You can specify a starting size for the window by passing **-h** and **-w** commandline options, followed by the path to an image (the format must be supported by *PxLoadImage()*).

- You need to link against the **libphexlib.so** library when you build this sample:
  qcc offscreen_ex.c -o offscreen_ex -lph -lphexlib
- You need to have all the **/usr/photon/dll/pi_io_*** dlls and **libphimg.so** on your target to run this example.

```
#include <Pt.h>
#include <photon/PxImage.h>

static PdOffscreenContext_t *context;
static void *my_alloc(long nbytes,int type);
static void raw_draw(PtWidget_t *widget,PhTile_t *damage);


int main(int argc,char *argv[])
{
    int c;
    PhDim_t dim = { 0,0 };

    if(PtInit(NULL))
        return(-1);

    while((c = getopt(argc,argv,"h:w:")) != -1)
    {
        switch(c)
        {
            case 'h':
                dim.h = atoi(optarg);
                break;
            case 'w':
                dim.w = atoi(optarg);
                break;
        }
    }

    if(argv[optind])
    {
        PxMethods_t methods;
        PhImage_t *image;

        memset(&methods,0,sizeof(methods));
        methods.px_alloc = my_alloc;
        methods.flags = PX_DIRECT_COLOR;

        if((image = PxLoadImage(argv[optind],&methods)) != NULL)
        {
            /* Create a context to render the image into.  The context will be
```

```
               created to be the size of the image and will store an exact copy
               of the original.  Note:  if you are short on video RAM, you
               might want to enable the Pg_OSC_MEM_SYS_ONLY flag to force
               the context to go to system RAM.  This will result in a slower
               1:1 blit though because the video h/w will not be able to access
               the image data directly - the data will have to be transferred
               from system memory (over the PCI bus) to video memory.  However
               if using a s/w scaled blit (ie scaled blit not supported in your
               h/w) it's better for the original image to be in system
               RAM because otherwise the CPU has to read the original,
               unscaled image from video RAM (over the PCI bus) to
               scale, then put it back into video RAM (over the PCI bus).
               The round trip (particularly the read) is expensive.  */

            if((context = PdCreateOffscreenContext(image->type,
                            image->size.w,image->size.h,0)) != NULL)
            {
                PtArg_t args[4];
                PtWidget_t *window;
                PhDrawContext_t *dc = PhDCSetCurrent(context);

                if(!dim.w || !dim.h)
                    dim = image->size;

                PgSetFillColor(Pg_WHITE);
                PgDrawIRect(0,0,image->size.w - 1,image->size.h - 1,Pg_DRAW_FILL);
                PgDrawPhImagemx(NULL,image,0);
                PgFlush();
                PgWaitHWIdle();
                PhDCSetCurrent(dc);

                image->flags |= Ph_RELEASE_IMAGE_ALL;
                PhReleaseImage(image);
                free(image);

                /* create a PtWindow with a PtRaw inside to draw the image */

                PtSetArg(&args[0],Pt_ARG_DIM,&dim,0);
                PtSetArg(&args[1],Pt_ARG_WINDOW_TITLE,argv[optind],0);
                if((window = PtCreateWidget(PtWindow,Pt_NO_PARENT,2,args)) != NULL)
                {
                    PhRect_t arect = { { 0,0 },{ 0,0 } };

                    PtSetArg(&args[1],Pt_ARG_RAW_DRAW_F,raw_draw,0);
                    PtSetArg(&args[2],Pt_ARG_ANCHOR_FLAGS,
                        Pt_LEFT_ANCHORED_LEFT | Pt_RIGHT_ANCHORED_RIGHT |
                        Pt_TOP_ANCHORED_TOP | Pt_BOTTOM_ANCHORED_BOTTOM,
                        Pt_LEFT_ANCHORED_LEFT | Pt_RIGHT_ANCHORED_RIGHT |
                        Pt_TOP_ANCHORED_TOP | Pt_BOTTOM_ANCHORED_BOTTOM);
                    PtSetArg(&args[3],Pt_ARG_ANCHOR_OFFSETS,&arect,0);
                    if(PtCreateWidget(PtRaw,Pt_DFLT_PARENT,4,args) != NULL)
                    {
                        PtRealizeWidget(window);
                        PtMainLoop();
                        return(0);
                    }
                }
            }
        }
    }

    return(-1);
}
```

```
static void *my_alloc(long nbytes,int type)
{
    return(type == PX_IMAGE ? PgShmemCreate(nbytes,NULL) : malloc(nbytes));
}


static void raw_draw(PtWidget_t *widget,PhTile_t *damage)
{
    /* raw widget draw function; simply blit the context onto the screen.
       PgContextBlit() will take care of scaling */

    PhRect_t src;

    src.ul.x = src.ul.y = 0;
    src.lr.x = context->dim.w - 1;
    src.lr.y = context->dim.h - 1;

    PgContextBlit(context,&src,PhDCGetCurrent(),PtCalcCanvas(widget,NULL));
}
```

Offscreen contexts can be invalidated by the graphics driver for a number of reasons. When this happens, the graphics driver sends to the Photon manager a Ph_EV_INFO event with a subtype of Ph_OFFSCREEN_INVALID. The event data is a single **long** describing why the offscreen areas have been invalidated. The possible reasons are as follows:

Pg_VIDEO_MODE_SWITCHED
: The graphics driver has changed video modes.

Pg_ENTERED_DIRECT
: An application has entered direct mode.

Pg_EXITED_DIRECT
: An application has left direct mode.

Pg_DRIVER_STARTED
: The video driver has just started execution.

Applications planning on using offscreen contexts should be sensitive to this event and reinitialize their off screen contexts accordingly.

Offscreen locks

You generally use offscreen locks with pointers that you gained via *PdGetOffscreenContextPtr()*. The locks ensure that:

- Draw stream commands don't draw while the offscreen context is locked.
- The memory is valid while the application is using it.

Your application should lock offscreen memory for as little time as possible. If the graphics driver needs to do something with the offscreen memory, it tries to gain a lock itself, potentially blocking **`io-graphics`** for a long period of time (the result being that the display may not get updated, and the user thinks that the computer has locked up).

The locks are implemented as semaphores in shared memory between **`io-graphics`** and the application.

The basic steps for using offscreen locks are:

1. Create a lock for an offscreen context by calling *PdCreateOffscreenLock()*. You can arrange for a signal to be dropped on the application if a request is made to remove the offscreen context while it's locked.
2. Lock the offscreen context, when required, by calling *PdLockOffscreen()*. You can optionally specify a timeout for the blocking.
3. Unlock the offscreen context by calling *PdUnlockOffscreen()*.
4. When you no longer need to lock the offscreen context, destroy the lock by calling *PdDestroyOffscreenLock()*.

When you're debugging, you can call *PdIsOffscreenLocked()* to determine whether or not the offscreen context is currently locked.

If you've locked the context, call *PdUnlockLockOffscreen()* to unlock it before destroying the lock or releasing the offscreen context.

Alpha blending support

Alpha blending is a technique of portraying transparency when drawing an object. It combines the color of an object to be drawn (the source) and the color of whatever the object is to be drawn on top of (the destination). The higher the portion of source color, the more opaque the object looks.

Alpha blending can be applied in three ways:

- As a global factor that applies to every pixel of the source
- With a map that indicates the alpha blending to be applied to each individual pixel. Alpha maps are "pinned" to the origin of your draw command and are tiled if the dimensions of the map are smaller than the dimension of the drawing operation.
- On a per-pixel basis

A 32-bit color is made up of four 8-bit *channels*: alpha, red, green, and blue. These channels are represented as (A, R, G, B). When referring to the source, the channels are denoted as As, Rs, Gs, and Bs; for the destination, they're Ad, Rd, Gd, and Bd.

The basic formula for alpha blending is:

```
Sm = source pixel        *   source multiplier
Dm = destination pixel   *   destination multiplier
destination pixel = Sm + Dm
```

There are several options for multipliers to achieve different blending effects. Flags are defined for source and destination multipliers in *PgSetAlpha()*.

You can also perform an "alpha test", which tests for certain conditions in the alpha channel before writing the source pixel to the destination. In an alpha test, pixels aren't blended — the source pixel is either written to the destination or it's not. For example, you can set the operation to only write the source pixel to the destination if the source alpha is less than the destination alpha.

The functions include:

PgAlphaOff()	Turn alpha blending operations off
PgAlphaOn()	Turn alpha blending operations on
PgAlphaValue()	Extract the alpha component from a color value
PgARGB()	Convert alpha, red, green, and blue values to composite color format
PgSetAlpha()	Set the parameters for alpha blending in detail
PgSetAlphaBlend()	Set the parameters for alpha blending simply

Chroma key support

Chroma-key operations are a method of masking out pixel data during a rendering operation (copies, image rendering, rectangles, etc.) based on a chroma color value. The basic modes of operation are:

- Masking on the source key color
- Masking on the destination key color
- Masking on everything but the source key color
- Masking on everything but the destination key color.

The functions include:

PgChromaOff()	Turn chroma key operations off
PgChromaOn()	Turn chroma operations on
PgSetChroma()	Set the chroma color and operation

Extended raster operations

The Photon microGUI supports 256 raster operations. Operations can be done using a combination of source pixel data, destination pixel data, and color expanded monochrome pattern pixel data. Extended raster operations are set the same way the normal raster operations are set, using *PgSetDrawMode()*.

The extended raster operations are pervasive, meaning that they affect all subsequent drawing operations, including bit-blit operations and images. The old style raster operations still exist and behave the same way they did in earlier versions of the Photon microGUI.

The extended raster operations are defined as **`Pg_DrawMode`***characters*, in reverse notation, where the *characters* are chosen from the following:

Character	Meaning
`P`	Pattern
`S`	Source
`D`	Destination
`o`	OR
`a`	AND
`n`	NOT
`x`	XOR

For example:

Pg_DrawModeS	Copy all source data.
Pg_DrawModePSo	Logically OR the source data with the pattern data.

For a complete list of all raster operations available, see **`<photon/Pg.h>`**.

Here's some sample code:

```
PdOffscreenContext_t *context1;
PhRect_t rsrc,rdst;

/* Initialize the offscreen area and render the data
   we want in it. */
   ⋮

/* Copy an image stored in an offscreen context to
   the display, ORing the source and pattern data
   together. */
rsrc.ul.x = rdst.ul.x = rsrc.ul.y = rdst.ul.y = 0;
rsrc.lr.x = rdst.lr.x = rsrc.lr.y = rdst.lr.y = 100;
```

```
PgSetDrawMode(Pg_DrawModePSo);
PgSetFillDither(Pg_BLUE,Pg_BLACK,Pg_PAT_CHECKB8);
PgContextBlit(context1, &rsrc, NULL, &rdst);

/* OR a blue and black checkerboard pattern with
   source data and copy it to the display area. */
PgFlush();
```

Video modes

A *video mode* describes what the display (what you see on your monitor) looks like. The description includes:

Width	The width of the display, in pixels.
Height	The height of the display, in pixels.
Pixel depth	The number of bits used to represent a pixel. This affects how many unique colors you can see on the screen at one time.
Refresh rate	How many times per second the phosphor on the CRT of your monitor is updated (represented in Hz).

The Photon microGUI's method of video mode enumeration is similar to the VESA spec, where there are "mode numbers", numerical representations of the width, height, and pixel depth of a video mode. The refresh rate is independent of the mode numbers (it's a different member of **`PgDisplaySettings_t`**).

The driver determines the mode numbers, so for one video card 640x480x8 might be mode 2, while on another card it might be mode 3022. Use *PgGetVideoModeInfo()* to determine the properties of any given mode number. Use *PgGetVideoModeList()* to get a list of the mode numbers supported by a particular graphics driver.

The functions for working with video modes are:

PdSetTargetDevice()	Set the target device
PgGetGraphicsHWCaps()	Determine the hardware capabilities
PgGetVideoMode()	Get the current video mode
PgGetVideoModeInfo()	Get information about a video mode
PgGetVideoModeList()	Query a graphics driver for a list of its supported video modes
PgSetVideoMode()	Set the current video mode

Here's some sample code:

```
PgVideoModes_t ModeList;
PgVideoModeInfo_t ModeInfo;
PgDisplaySettings_t ModeSetting;
int i=0, done=0;

if (PgGetVideoModeList(&ModeList))
{
    /* Error -- driver doesn't support this. */
}

/* Use the default refresh rate for this mode. */
ModeSetting.refresh = 0;

while (!done)
{
    if (PgGetVideoModeInfo(ModeList.modes[i], &ModeInfo))
    {
        /* Error code */
    }

    if ((ModeInfo.width == 640) && (ModeInfo.height == 480) &&
(ModeInfo.bits_per_pixel == 16))
    {
        /* We found the mode we were looking for. */
        done = 1;
        ModeSetting.mode = ModeList.modes[i];
    }

    i++;
    if (i >= ModeList.num_modes)
    {
        /* Error -- Mode wasn't found. */
        done=1;
    }
}

PgSetVideoMode (&ModeSetting);
```

Gradients

A gradient is a gradual blend of two colors. The Photon library supports:

- Driver-level gradients — quick, but not sophisticated. Accuracy is sacrificed for speed.
- Application-level gradients — slower, but more accurate.

Driver-level gradients

Although the Photon library supports a large variety of gradients (see **`PhImage_t`**), there are times when you would just want a simple gradient to be rendered without having to store it in a **`PhImage_t`**. As a result, some basic gradient rendering operations have been added to the graphics driver:

PgDrawGradient()

Ask the graphics driver to render a gradient

Application-level gradients

These functions let you create your own gradients:

PgBevelBox()	Draw a beveled box with gradients
PgCalcColorContrast()	Compute light and dark colors to use for a gradient
PgContrastBevelBox()	Draw a beveled box with gradients and a given level of contrast
PgDrawGradientBevelBox()	Draw a beveled box with gradients and two flat colors

Video overlay

A video overlay scaler is a hardware feature that allows a rectangular area of the visible screen to be replaced by a scaled version of a different image. The prescaled video frames are typically stored in offscreen memory, and are fetched from memory and overlaid on top of the desktop display image in real time, by the overlay scaler.

Chroma keying is used to control what parts of the video frame are visible. Typically, the application picks a color to be the chroma-key color and draws a rectangle of this color where video content is to appear. When another application's window is placed on top of the video playback application, the chroma-colored rectangle is obscured. Since the video hardware is programmed to display video content only where the chroma-key color is drawn, video doesn't show through where the chroma-colored rectangle is obscured.

The following functions and data types deal with video overlay:

PgConfigScalerChannel()	Configure a video overlay scaler channel
PgCreateVideoChannel()	Create a channel for video streaming
PgDestroyVideoChannel()	Destroy resources associated with a video channel
PgGetOverlayChromaColor()	Return the color used for video overlay chroma-key operations
PgGetScalerCapabilities()	Get the capabilities of a video overlay scaler
PgNextVideoFrame()	Get the index of the next video buffer to fill

`PgScalerCaps_t`

Data structure that describes video overlay scaler capabilities

`PgScalerProps_t`

Data structure that describes video overlay scaler properties

`PgVideoChannel_t`

Data structure that describes a video overlay channel

Example

```
#include <stdio.h>

#include <Ph.h>

#define SRC_WIDTH   100
#define SRC_HEIGHT  100

#define DATA_FORMAT Pg_VIDEO_FORMAT_YV12

unsigned char   *ybuf0, *ybuf1;
unsigned char   *ubuf0, *ubuf1;
unsigned char   *vbuf0, *vbuf1;

void
grab_ptrs(PgVideoChannel_t *channel)
{
    /* Buffers have moved; get the pointers again. */
    ybuf0 = PdGetOffscreenContextPtr(channel->yplane1);
    ybuf1 = PdGetOffscreenContextPtr(channel->yplane2);
    ubuf0 = PdGetOffscreenContextPtr(channel->uplane1);
    ubuf1 = PdGetOffscreenContextPtr(channel->uplane2);
    vbuf0 = PdGetOffscreenContextPtr(channel->vplane1);
    vbuf1 = PdGetOffscreenContextPtr(channel->vplane2);

    if (channel->yplane1)
        fprintf(stderr, "ybuf0: %x, stride %d\n", ybuf0,
            channel->yplane1->pitch);
    if (channel->uplane1)
        fprintf(stderr, "ubuf0: %x, stride %d\n", ubuf0,
            channel->uplane1->pitch);
    if (channel->vplane1)
        fprintf(stderr, "vbuf0: %x, stride %d\n", vbuf0,
            channel->vplane1->pitch);

    if (channel->yplane2)
        fprintf(stderr, "ybuf1: %x, stride %d\n", ybuf1,
            channel->yplane2->pitch);
    if (channel->uplane2)
        fprintf(stderr, "ubuf1: %x, stride %d\n", ubuf1,
            channel->uplane2->pitch);
    if (channel->vplane2)
        fprintf(stderr, "vbuf1: %x, stride %d\n", vbuf1,
            channel->vplane2->pitch);
}

void
overlay_example()
{
    PgVideoChannel_t *channel;
```

```
PgScalerCaps_t  vcaps;
PgScalerProps_t props;
unsigned char   *ptr;
unsigned short  *ptr16;
int     i = 0, j, k, index;
int     color;
PhDrawContext_t *old;
int     rc;

if ((channel = PgCreateVideoChannel(
                 Pg_VIDEO_CHANNEL_SCALER, 0)) == NULL) {
    perror("PgCreateVideoChannel");
    exit(1);
}

/*
 * Cycle through the available formats looking for the one
 * we’re interested in.
 */
vcaps.size = sizeof (vcaps);
while (PgGetScalerCapabilities(channel, i++, &vcaps) == 0) {
    if (vcaps.format == DATA_FORMAT)
        break;
    vcaps.size = sizeof (vcaps);
}
if (vcaps.format != DATA_FORMAT) {
    fprintf(stderr, "Format not supported?\n");
    exit(1);
}

props.size = sizeof (props);
props.format = DATA_FORMAT;
props.viewport.ul.x = 20;
props.viewport.ul.y = 20;
props.viewport.lr.x = 600;
props.viewport.lr.y = 440;
props.src_dim.w = SRC_WIDTH;
props.src_dim.h = SRC_HEIGHT;
props.flags =
    Pg_SCALER_PROP_SCALER_ENABLE |
    Pg_SCALER_PROP_DOUBLE_BUFFER |
    Pg_SCALER_PROP_DISABLE_FILTERING;
if (PgConfigScalerChannel(channel, &props) == -1) {
    fprintf(stderr, "Configure channel failed\n");
    exit(1);
}

grab_ptrs(channel);

for (i = 0; i < 100; i++) {
    index = PgNextVideoFrame(channel);
    delay(50);
    ptr = (void *)(index ? ybuf1 : ybuf0);
    color = rand() & 0xff;
    for (k = 0; k < props.src_dim.h; k++) {
        memset(ptr, color, channel->yplane1->pitch);
        ptr += channel->yplane1->pitch;
    }
}

props.flags &= ~Pg_SCALER_PROP_DISABLE_FILTERING;
switch (PgConfigScalerChannel(channel, &props)) {
case -1:
```

```
            fprintf(stderr, "Configure channel failed\n");
            exit(1);
            break;
        case 1:
            grab_ptrs(channel);
            break;
        case 0:
        default:
            break;
        }

        fprintf(stderr, "\"TV snow\" effect\n");
        for (i = 0; i < 1000; i++) {
            index = PgNextVideoFrame(channel);
            ptr = (void *)(index ? ybuf1 : ybuf0);
            for (k = 0; k < props.src_dim.h; k++) {
                for (j = 0; j < channel->yplane1->pitch; j++)
                    *(ptr + j) = rand() & 0xff;
                ptr = (void *)((char *)ptr + channel->yplane1->pitch);
            }

            /* Set the chromanance to neutral for monochrome */
            ptr = ubuf0;
            for (i = 0; i < props.src_dim.h; i++) {
                memset(ptr, 128, props.src_dim.w / 2);
                ptr += channel->uplane1->pitch;
            }
            ptr = vbuf0;
            for (i = 0; i < props.src_dim.h; i++) {
                memset(ptr, 128, props.src_dim.w / 2);
                ptr += channel->vplane1->pitch;
            }

            if (rand() % 200 == 23) {
                props.viewport.ul.x = rand() % 400;
                props.viewport.ul.y = rand() % 300;
                props.viewport.lr.x =
                    props.viewport.ul.x + SRC_WIDTH + rand() % 200;
                props.viewport.lr.y =
                    props.viewport.ul.y + SRC_HEIGHT + rand() % 200;
                if (PgConfigScalerChannel(channel, &props) == 1)
                    grab_ptrs(channel);
            }
        }

        /*
         * This isn't really necessary, since the video resources
         * should automatically be released when the app exits
         */
        PgDestroyVideoChannel(channel);
    }

    int
    main(int argc, char *argv[])
    {
        PhAttach(NULL, NULL);

        overlay_example();

        fprintf(stderr, "Exiting normally\n");
    }
```

Layers

Some display controllers allow you to transparently overlay multiple "screens" on a single display. Each overlay is called a *layer*.

Layers can be used to combine independent display elements. Because overlaying is performed by the graphics hardware, it can be more efficient than rendering all of the display elements onto a single layer. For example, a fast navigational display can be implemented with a scrolling navigational map on a background layer, and pop-up GUI elements, such as menus or a web browser, on a foreground layer.

Layer capabilities vary depending on the display controller and the driver. Some display controllers don't support layers. Different layers on the same display may have different capabilities. You should use *PgGetLayerCaps()* to determine whether a layer exists and which features are supported by the layer.

Layers are indexed per-display, starting from 0, from back to front in the default overlay order.

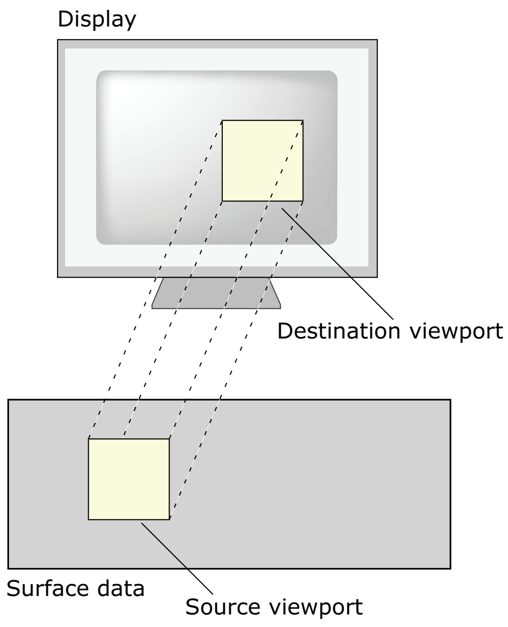
A layer is either *active* (shown) or *inactive* (hidden). It may not be possible to activate a layer if its configuration is incomplete (if, for example, the layer format is unspecified, or there aren't enough surfaces assigned to it). A layer's configuration persists when it's inactive. After a video mode switch, all layers revert to their default configuration.

The images on all the active layers of a display are combined, using alpha blending, chroma keying, or both, to produce the final image on the display.

Surfaces

The image on a layer is fetched from one or more offscreen contexts, also called *surfaces*. The number of surfaces needed by a layer is determined by the layer format. For example, a layer whose format is Pg_LAYER_FORMAT_ARGB888 requires one surface, while a layer whose format is Pg_LAYER_FORMAT_YUV420 requires three surfaces for a complete image. The format of a layer is set using *PgSetLayerArg()*.

Viewports



Source and destination viewports.

The *source viewport* defines a rectangular window into the surface data. This window is used to extract a portion of the surface data for display by the layer.

The *destination viewport* defines a rectangular window on the display. This window defines where the layer will display its image.

Scrolling and scaling, if supported by the layer, can be implemented by adjusting the source and destination viewports. To scroll or pan an image, move the position of the source viewport. To scale an image, increase or decrease the size of the destination viewport.

You must target these functions at a device by calling *PdSetTargetDevice()*.

Layer API

The layer API includes:

PgGetLayerCaps()	Query the capabilities of a layer
PgCreateLayerSurface()	Create an offscreen context displayable by a layer
PgSetLayerSurface()	Display an offscreen context on a layer

PgSetLayerArg()	Configure a layer parameter
PgLockLayer()	Lock a layer for exclusive use by an application
PgUnlockLayer()	Release a locked layer
`PgLayerCaps_t`	Data structure that describes the capabilities for a layer

WARNING: The layer API is incompatible with the existing video overlay API (*PgCreateVideoChannel()*, *PgConfigScalerChannel()*, *PgNextVideoFrame()*, and so on). Don't run two applications that use different APIs simultaneously.

Note the following:

- Photon cannot render in offscreen contexts that are in a different format from the current video mode. Thus, it may not be possible to use Photon draw functions in an offscreen context allocated by *PgCreateLayerSurface()*. Instead, the application should use *PdGetOffscreenContextPtr()* to get a pointer to the video memory and write data directly into the video memory.
- If an application changes the main display's surface by calling *PgSetLayerSurface()*, Photon will continue to render on the old surface. The application should keep a pointer to the old main display surface and restore it when it releases the layer. See the code below for an example.

Using layers

To use layers, you typically do the following:

1. Call *PgGetLayerCaps()* with successively incremented indexes to enumerate your hardware capabilities (unless you already know them). If *PgGetLayerCaps()* fails for all values, the driver doesn't support layers.
2. If you want to prevent other applications from accessing a layer, call *PgLockLayer()*.
3. Allocate surfaces for the layer, and offscreen contexts for the surfaces, by calling *PgCreateLayerSurface()*.
4. Call *PgSetLayerArg()* with an *arg* argument of Pg_LAYER_ARG_LIST_BEGIN.
5. Call *PgSetLayerArg()* to set other arguments as required.
6. Call *PgSetLayerSurface()* to display a surface's offscreen context on a layer.
7. Call *PgSetLayerArg()* to set other arguments as required. You can specify Pg_LAYER_ARG_ACTIVE to display the layer.
8. Call *PgSetLayerArg()* with an *arg* argument of Pg_LAYER_ARG_LIST_END.

9 If the layer format is one of the RGB or PAL8 formats, set the current draw context to render into a surface's associated draw context(s), and then use the *Pg** functions to draw into the offscreen context.

10 If the layer format is YUV, and so on, you typically dump data directly to the buffer (like the video channel buffers).

11 If you locked a layer, you must use *PgUnlockLayer()* to unlock it before your application exits.

See the code below for an example of using the layers API.

Example

```
#include <errno.h>
#include <stdio.h>
#include <Ph.h>

int
FindFormatIndex(int layer, unsigned int format)
{
    PgLayerCaps_t caps;
    int format_idx = 0;

    while (PgGetLayerCaps(layer, format_idx, &caps) != -1) {
        if (caps.format == format)
            return format_idx;

        format_idx++;
    }
    return -1;
}

int
main(int argc, char **argv)
{
/*
 * For best results, these values should match your video mode.
 */
#define LAYER_FORMAT    Pg_LAYER_FORMAT_ARGB8888
#define SURFACE_WIDTH   1024
#define SURFACE_HEIGHT  768

    struct _Ph_ctrl         *ph;
    PgLayerCaps_t           caps;
    PdOffscreenContext_t    *surf;
    PdOffscreenContext_t    *scr = NULL;
    PhDrawContext_t         *olddc;
    PhRid_t                 driver_rid = -1;
    int     layer_idx = -1;
    int     format_idx = -1;
    int     active = 1;
    int     i;

    PhArea_t sarea, darea;

    /*
     * Arguments:
     * -d <driver region>
     * -l <layer index>
     */
```

```
while ((i = getopt(argc, argv, "d:l:")) != -1) {
    switch(i) {
    case 'd': /* driver region */
        driver_rid = atol(optarg);
        break;
    case 'l': /* layer index */
        layer_idx = atoi(optarg);
        break;
    default:
        break;
    }
}

if (layer_idx == -1) {
    printf("Specify layer index.\n");
    exit(-1);
}

if (driver_rid == -1) {
    printf("Specify graphics driver region.\n");
    exit(-1);
}

ph = PhAttach(NULL, NULL);
if (ph == NULL) {
    perror("PhAttach");
    exit(-1);
}

if (-1 == PdSetTargetDevice(PhDCGetCurrent(), driver_rid)) {
    perror("PdSetTargetDevice");
    exit(-1);
}

/* Check if the layer supports the required format */
format_idx = FindFormatIndex(layer_idx, LAYER_FORMAT);
if (format_idx == -1) {
    printf("Layer doesn't support format\n");
    exit(-1);
}

/* Get the layer capabilities */
PgGetLayerCaps(layer_idx, format_idx, &caps);

if (caps.caps & Pg_LAYER_CAP_MAIN_DISPLAY) {
    /* Save a reference to the current display surface */
    scr = PdCreateOffscreenContext(0, 0, 0,
                                   Pg_OSC_MAIN_DISPLAY);
}

/* Allocate a surface for the layer */
surf = PgCreateLayerSurface(layer_idx, 0, format_idx,
        SURFACE_WIDTH, SURFACE_HEIGHT,
        Pg_OSC_MEM_PAGE_ALIGN);
if (surf == NULL)
    exit(-1);

/* Draw some stuff on the surface */
olddc = PhDCSetCurrent(surf);
PgSetFillColor(Pg_BLACK);
PgDrawIRect(0, 0, SURFACE_WIDTH-1, SURFACE_HEIGHT-1,
            Pg_DRAW_FILL);
```

```
PgSetFillColor(Pg_YELLOW);
PgDrawIRect(0, 0, 100, 100, Pg_DRAW_FILL);
PgSetFillColor(PgRGB(255,180, 0));
PgDrawIRect(70, 80, 600, 500, Pg_DRAW_FILL);
PhDCSetCurrent(olddc);

/* Lock the layer */
if (-1 == PgLockLayer(layer_idx))
    exit(-1);

/* Start configuring arguments */
PgSetLayerArg(layer_idx, Pg_LAYER_ARG_LIST_BEGIN, 0, 0);


/* Select the layer format */
PgSetLayerArg(layer_idx, Pg_LAYER_ARG_FORMAT_INDEX,
        &format_idx, sizeof(int));

/* This changes the current display surface */
PgSetLayerSurface(layer_idx, 0, surf);

PgSetLayerArg(layer_idx, Pg_LAYER_ARG_ACTIVE,
        &active, sizeof(int));

/* Configure other arguments ... */

if (!(caps.caps & Pg_LAYER_CAP_MAIN_DISPLAY)) {
    sarea.pos.x = 0; sarea.pos.y = 0;
    sarea.size.w = SURFACE_WIDTH;
    sarea.size.h = SURFACE_HEIGHT;
    PgSetLayerArg(layer_idx, Pg_LAYER_ARG_SRC_VIEWPORT, &sarea,
                  sizeof(sarea));

    darea.pos.x =0; darea.pos.y =0;
    darea.size.w =SURFACE_WIDTH/2 ;
    darea.size.h =SURFACE_HEIGHT/2 ;
    PgSetLayerArg(layer_idx, Pg_LAYER_ARG_DST_VIEWPORT, &darea,
                  sizeof(darea));
 }

/* End configuration */
PgSetLayerArg(layer_idx, Pg_LAYER_ARG_LIST_END, 0, 0);

/* Application continues ... */
sleep(3);

/* Finished using layer; Restore the current display
   surface */

   active = 0;

    PgSetLayerArg(layer_idx, Pg_LAYER_ARG_LIST_BEGIN, 0, 0);
    PgSetLayerArg(layer_idx, Pg_LAYER_ARG_ACTIVE, &active,
                  sizeof(int));
    PgSetLayerSurface(layer_idx, 0, scr);
    PgSetLayerArg(layer_idx, Pg_LAYER_ARG_LIST_END, 0, 0);


PgUnlockLayer(layer_idx);

if (scr) PhDCRelease(scr);
PhDCRelease(surf);
```

```
    PhDetach(ph);
    exit(0);
}
```

Chapter 20

Understanding Encodings, Fonts, Languages and Code Tables

This chapter describes the differences between encodings, fonts, languages, and code tables with respect to QNX Neutrino and Photon.

This chapter describes the following concepts:

- Terminology Definitions
- Unicode encoding in Photon
- Advanced Graphics
- General Notes
- Language Notes

Terminology Definitions

Each concept is described below:

Language

With respect to computers, *language* has a vague meaning. When someone asks, “Does your system support language *X*?”, you should say, “What do you mean? Input? Display? Unicode only? Translation of antiquated encodings?”. Some languages are complex, including, but not limited to, Thai, Hebrew, Arabic. Complex language can either require compositing of multiple glyphs to form a single character, or bi-directional (right-to-left) processing.

Code Table

In basic terms, a *code table* is a two column list that maps a numerical value to a glyph. The most widely used code table is Unicode (see http://www.unicode.org).

Encoding

Encoding values are “stored” from a code table. There are many different encoding types to choose from depending on your application. Here are some encodings for storing Unicode:

- UTF-8
- UTF-16 (UCS-2)
- UTF-32 (UCS-4)

Font

A *font* file is a binary file that contains glyphs, or “pictures”, of symbols representing the building blocks of a displayable character set. Depending on the language, multiple glyphs can comprise a single character. In Unicode encoded fonts, some languages, such as Chinese, Japanese, and Korean, have combining characters that are comprised of multiple glyphs.

Characters

A *character* can be one of the following:

- a single Unicode point glyph (for example, `0x00CB`)
- a combined character (pre-composed glyphs, already assigned a single Unicode point, for example `0x6E90`, `?`)
- a composited character (composed of multiple glyphs, from multiple Unicode points)

Unicode encoding in Photon

Photon supports the Unicode 3.x (16-bits) code table through its font rendering subsystem. Range: 0x0000 to 0xFFFF.

Photon completely supports UTF-8 (preferred) and UTF-16 encodings. There is limited support for UTF-32 encoding.

Photon does not currently support complex language processing. For example, Photon does support Thai TIS-620 encoding, so that TIS-620 encoded data can be converted to and from Unicode, however it does not support the processing of this complex language for display and input.

The following lists specify UTF encoding support on a per-function basis. This information is also provided in the documentation for each function.

UTF-8, UTF-16 (Pg_TEXT_WIDECHAR)

- *PgDrawText()*, *PgDrawTextCx()*
- *PgDrawTextv()*
- *PgDrawTextChars()*
- *PgDrawTextvCx()*, *PgDrawTextCharsCx()*
- *PgDrawTextArea()*, *PgDrawTextAreaCx()*
- *PfExtentFractTextCharPositions()*
- *PfExtentTextCharPositions()*, *PfExtentTextCharPositionsCx()*

UTF-8

- *PgExtentMultiText()*
- *PgExtentText()*
- *PfExtentText()*
- *PfExtentTextToRect()*
- *PfFractionalExtentText()*

- *PfExtentComponents()*, *PfExtentComponentsCx()*
- *PfTextWidthBytes()*
- *PfTextWidthChars()*

UTF-8, UTF-16 (PF_WIDECHAR), UTF-32 (PF_WIDECHAR32)

- *PfRender()*, *PfRenderCx()*
- *PfExtent()*, *PfExtentCx()*
- *PfExtent16dot16()*

UTF-16

- *PfExtentWideText()*
- *PfWideTextWidthBytes()*
- *PfWideTextWidthChars()*

String representation

In this history of Unix, the size of **`wchar_t`** was changed from 16-bits to 32-bits. Therefore, when working in a Photon environment, you must use **`uint16_t`** for wide character strings. In general, there is no need to use **`wchar_t`**, except possibly when porting while creating a Photon application. Even in this situation, there is no true benefit of using a 32-bit **`wchar_t`**, because Photon currently only supports Unicode 3.x (16-bits). To avoid these sizing issues altogether, you should use multi-byte UTF-8 encoded strings.

Translation

Photon provides the *PxTranslate*()* family of functions in order to translate non-Unicode character set multi-byte strings to and from UTF-8, and Unicode-encoded UTF-16/UTF-32 to and from UTF-8. For more information, see Appendix: Unicode Multilingual Support.

Services

Photon uses a central font server called **`phfont`**. This server uses plugins to support the rendering of different types of font file formats. You can find more details on each plugin by reading the documentation, or by using the **`use`** utility on the plugins located in **`/lib/dll/font`**. A central server allows for the consolidation of resources, and the decoupling of vendor-specific API calls from client applications.

The font server **phfont** is not coupled to the Photon environment directly. Non-photon applications can use *Pf*Cx()* calls by linking against the **libfont.*** library.

Advanced Graphics

The Advanced Graphics API has no font-specific API calls. There are currently two options for Advanced Graphics applications:

1. Use *phfont()* and *libfont.*()* utilizing the *Pf*Cx()* API calls.
2. Use the Bitstream Font Fusion 2.4 library directly. This library provides a very low-level font rendering solution, which is used by several of the **phfont** rendering plugins. Header files for this library are installed at **/usr/include/FontFusion/T2K**.

General Notes

In general, Photon and Advanced Graphics can both utilize the same font files. The only exception is Photon's PHF file format, which is generated from widely available BDF files. If you must use a bitmap font, the utility **bdftophf2** can be used to generate them.

The following font file formats are currently supported:

- Bitmap (**.phf**, converted from Unicode, ascending sorted, BDF files, legacy)
- TrueType (**.ttf**)
- Adobe Type 1 (**.pfa**)
- Adobe Type 2 (**.ccf**)
- Bitstream Stroke (**.ffs**)
- Bitstream T2K (**.t2k**)
- Bitstream Speedo (**.spd**, public encryption only)
- TrueType Collection (**.ttc**)
- Bitstream PFR (**.pfr** — legacy, not recommended for use in new products)

You should always use scalable font technology, unless you use a highly contained system that only requires specific point sizes. Once you start adding additional bitmap fonts for more point sizes, any benefits are quickly lost due to large file sizes. Due to the flexibility and speed of current scalable font technology, bitmap fonts are no longer recommended.

In Photon, fonts are installed under **/usr/photon/font_repository**. See the documentation for **mkfontdir**, **phfont**, and the chapter Appendix: Photon in Embedded Systems.

Documentation for the current shipping version of FontFusion is available at: **`$QNX_TARGET/usr/help/FontFusion/FF_Ref_Guide.pdf`**

Language Notes

Languages that use characters which are combined of glyphs that are already in the Unicode range (for example Chinese, Japanese, Korean) are currently supported for display and input. Input is supported by the input methods *cpim()*, *vpim()*, and *kpim()*.

Languages that are complex, such as Thai, Arabic, and Hebrew (though single glyphs can be displayed) are not currently supported for display or input.

Chapter 21

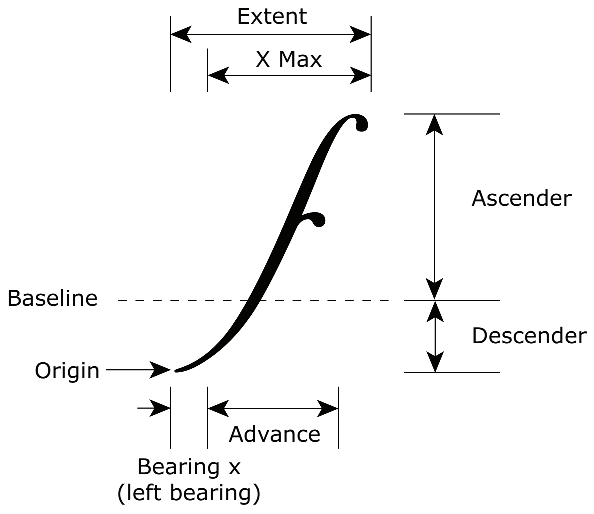
Fonts

In this chapter. . .

Although the Photon and font libraries provide many functions that deal with fonts (see the Pf—Font Server chapter of the Photon *Library Reference*), most of them are low-level routines that you probably don't need to use. This chapter describes the basics of using fonts.

Symbol metrics

Let's start with some definitions:



Symbol metrics.

Advance	The amount by which the pen x position advances after drawing the symbol. This might not be the full width of the character (especially in an italic font) to implement kerning.
Ascender	The height from the baseline to the top of the character.
Bearing x or *left bearing*	The amount of the character to the left of where the character is deemed to start.
Descender	The height from the bottom of the character to the baseline.
Extent	The width of the symbol. Depending on the font, this might include some white space.
Origin	The lower left corner of the character
X Max	The width of the symbol, not including the bearing x.

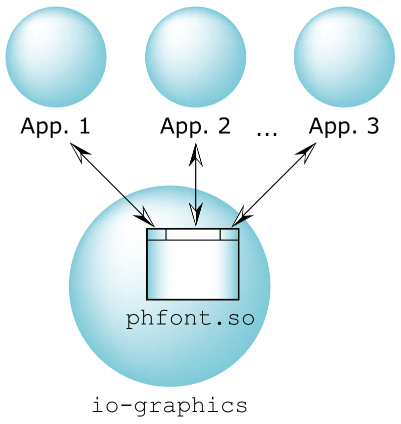
To save time and memory, kerning isn't supported.

Font function libraries

There are two libraries of font-related functions shipped with Neutrino:

- the functions that come with the Photon library **ph**
- the separate **libfont** library. All **libfont** functions have a **Cx** or **Dll** suffix.

The Photon library font functions reference **libfont** functions using global contexts.



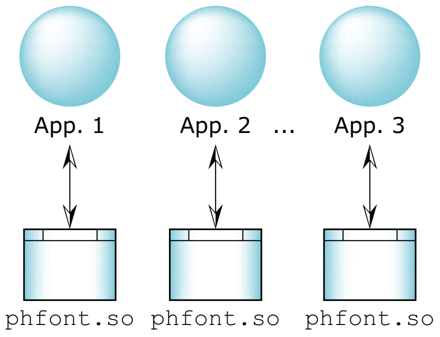
Font architecture using **io-graphics** *with a resource manager font instance*

The font library, **libfont**, offers three different methods for obtaining font services:

- The first method uses message passing to communicate to an external font server process.
- The second method allows an application to create a private font server instance, eliminating message passing for font processing.
- The third method creates a resource manager font server instance using a separate thread within the application. The resource manager thread will attach a device (e.g. **/dev/phfont**), which other applications may connect to via *PfAttachCx()* , if the applications are aware of the device name. Note that to run a resource manager font server private instance, the application must be root-privileged. The application which invokes the font server instance does not require message passing for font processing. This method is used by **io-graphics** to run global font server by default.

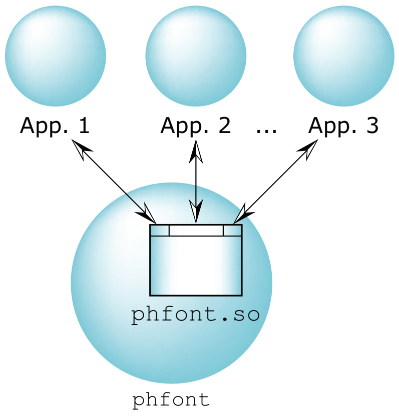
These methods are made possible through the font server plugin, **phfont.so**, which contains all common font server code. This allows the memory footprint of the font server to be potentially much smaller than it was before. The font library also allows you to fine-tune your system's memory requirements for your particular embedded environment.

For example, if your system had maximum resources, you could run a private font server instance for every application. Each application would use *PfAttachLocalDLL()*, passing either your own schema or NULL.



Every application with its own private font server.

Now say you had only minimal resources. Your applications would use the external font server **phfont**, or **io-graphics** that uses **phfont.so**, with each application performing a message pass to process fonts, which would require minimal memory but higher CPU usage. In the case of **io-graphics**, font rendering is done locally with no memory passing.



Applications sharing a common font server.

The **libfont** library DLL functions introduce the concept of a schema, which is a configuration file you can use to override the default settings for private font server instances.

Font names

A font is identified by its name, which can be in one of these forms:

foundry name	The name assigned by the font foundry to identify the font family, such as Helvetica, Comic Sans MS, and PrimaSans BT. Note the use of capitals. The foundry name doesn't include information about the style (e.g. bold, italic) or the size. This name is universal across operating environments (e.g. X, Photon).
stem name	A unique identifier that includes an abbreviation of the foundry name, as well as the style (e.g. **b**, **i**) and the size. For example, **helv12** is the stem name for 12-point Helvetica, and **helv12b** is the stem name for 12-point bold Helvetica.

To specify a font in the Photon API, you always use a stem name. You should consider stem names to be constant identifiers, not modifiable strings.

You can hard-code all references to fonts in a Photon application. But your application can be more flexible if it uses the foundry name to choose the best match from whatever fonts are available. That way, there isn't a problem if a particular font is eventually renamed, removed, or replaced.

For example, the following call to *PtAlert()* uses the hard-coded stem name **helv14** to specify 14-point Helvetica:

```
answer = PtAlert(
            base_wgt, NULL, "File Not Saved", NULL,
            "File has not been saved.\nSave it?",
            "helv14", 3, btns, NULL, 1, 3, Pt_MODAL );
```

You can get the available stem names from the names of the files in **${**PHOTON_PATH**}/font_repository** — just remove any file extension (e.g. **.phf**).

Alternately, if you have a **$HOME/.ph** directory, check in **$HOME/.ph/font/**.

Querying available fonts

The above example takes a shortcut by using a hard-coded stem name (**helv14**). And, like any shortcut, this approach has trade-offs. First, stem names are subject to change. More importantly, all versions of the Photon microGUI up to and including 1.13 have only 16 characters available for the stem name. This isn't always enough to give each font a unique stem. The current version of the Photon microGUI allows 80 characters.

We've defined the **FontName** data type for you to use for the buffer you pass to *PfGenerateFontName()*. It's an array of size MAX_FONT_TAG. For successful font programming, don't use a font identifier storage buffer that's smaller than **FontName**.

To get around hard-coded stem name issues, you can use *PfQueryFonts()* to determine which fonts are available and provide the information needed to build a stem name. This function queries the font server, and protects you from future changes.

FontDetails structure

Once you've got the list of fonts, you need to examine each **FontDetails** structure in it to find the font you need and determine the string to use as the stem name.

The **FontDetails** structure is defined in **<photon/Pf.h>**, and contains at least these elements:

FontDescription *desc*
: The foundry name or full descriptive name of the font, such as **Helvetica** or **Charter**.

FontName *stem*
: The short form. This provides a part of the stem name used by the Photon API calls. For example, **helv** and **char** correspond to Helvetica and Charter.

Generating font names

As described earlier, the Photon API requires a stem name to identify a font, but if you want to be flexible, you should use a font foundry name.

The easiest way to get a stem name, given the font foundry name, desired point size, and style, is to call *PfGenerateFontName()*. It creates, in a buffer that you supply, a unique stem name for the font. (You can use this approach even if you don't use *PfQueryFonts()* to find all the available fonts.)

Here's the same call to *PtAlert()* as shown earlier, but this time it calls *PfGenerateFontName()*:

```
char Helvetica14[MAX_FONT_TAG];

if ( PfGenerateFontName("Helvetica", 0, 14,
                         Helvetica14) == NULL )
{
  /* Couldn't find the font! */
  ...
}

answer = PtAlert(
            base_wgt, NULL, "File Not Saved", NULL,
            "File has not been saved.\nSave it?",
            Helvetica14, 3, btns, NULL, 1, 3,
            Pt_MODAL );
```

Example

Now that we've looked at the pieces involved, it's fairly simple to follow the steps needed to build up the correct stem name for a given font.

Keep these things in mind:

- Use a **FontName** buffer to store the stem name.
- Search for a font based on the foundry name (i.e. the *desc* member of its **FontDetails** entry), not on the *stem*.

You'll probably want to do this work in the initialization function for your application, or perhaps in the base window setup function. Define the **FontName** buffer as a global variable; you can then use it as needed throughout your application.

Here's a sample application-initialization function:

```
/***************************
***   global variables   ***
***************************/

FontName GcaCharter14Bold;

int
fcnAppInit( int argc, char *argv[] )

{
   /* Local variables */
   FontDetails tsFontList [nFONTLIST_SIZE];
   short sCurrFont = 0;
   char caBuff[20];

   /* Get a description of the available fonts */

   if (PfQueryFonts (PHFONT_ALL_SYMBOLS,
          PHFONT_ALL_FONTS, tsFontList,
          nFONTLIST_SIZE) == -1)
   {
      perror ("PfQueryFonts() failed:  ");
      return (Pt_CONTINUE);
   }

   /* Search among them for the font that matches our
      specifications */

   for (sCurrFont = 0;
        sCurrFont < nFONTLIST_SIZE; sCurrFont++)
   {
      if ( !strcmp (tsFontList[sCurrFont].desc,
                    "Charter") )
      break;  /* we've found it */
   }

   /* Overrun check */
   if (sCurrFont == nFONTLIST_SIZE)
   {
      /* check for a partial match */
      for (sCurrFont = 0;
           sCurrFont < nFONTLIST_SIZE;
           sCurrFont++)
      {
         if ( !strncmp (tsFontList[sCurrFont].desc,
                        "Charter",
                        strlen ("Charter") ) )
            break;  /* found a partial match */
      }

      if (sCurrFont == nFONTLIST_SIZE)
      {
         printf ("Charter not in %d fonts checked.\n",
                 sCurrFont);
         return (Pt_CONTINUE);
      }
      else
         printf ("Using partial match -- 'Charter'.\n");
   }


   /* Does it have bold? */
   if (!(tsFontList[sCurrFont].flags & PHFONT_INFO_BOLD))
   {
      printf ("Charter not available in bold font.\n");
      return (Pt_CONTINUE);
   }


   /* Is 14-point available? */
   if ( !( (tsFontList[sCurrFont].losize ==
            tsFontList[sCurrFont].hisize == 0)
            /* proportional font -- it can be shown in
            14-point*/

        ||

        ( (tsFontList[sCurrFont].losize <= 14 )
          &&
          (tsFontList[sCurrFont].hisize >= 14 ) ) ) )
          /* 14-point fits between smallest and
             largest available size */

    {
        printf ("Charter not available in 14-point.\n");
        return (Pt_CONTINUE);
    }

    /* Generate the stem name */
    if (PfGenerateFontName( tsFontList[sCurrFont].desc,
                            PF_STYLE_BOLD, 14,
                            GcaCharter14Bol) == NULL)
    {
       perror ("PfGenerateFontName() failed:  ");
       return (Pt_CONTINUE);
    }

    /* You can now use GcaCharter14Bold as an argument to
       PtAlert(), etc. */


    /* Eliminate 'unreferenced' warnings */
          argc = argc, argv = argv;

    return( Pt_CONTINUE );
```

```
}
```

For the above code to work, you must declare the following information in the application's global header file. To do this, use PhAB's Startup Info/Modules dialog (accessed from the Application menu).

```
/*********************************
***   user-defined constants   ***
*********************************/
#define nFONTLIST_SIZE 100  /* an arbitrary choice of size */

/***************************
***   global variables   ***
***************************/

extern FontName GcaCharter14Bold;
```

You can avoid using a specific size for the list by calling *PfQueryFonts()* with *n* set to 0 and *list* set to NULL. If you do this, *PfQueryFonts()* returns the number of matching fonts but doesn't try to fill in the list. You can use this feature to determine the number of items to allocate.

Remember to define this header before you start adding callbacks and setup functions — that way, it's automatically included as a **#define**. If you forget, you'll have to go back and add the statement manually. For more information, see "Specifying a global header file" in the Working with Applications chapter.

And last of all, here's a sample callback that uses our stem name string:

```
int
fcnbase_btn_showdlg_ActivateCB( PtWidget_t *widget,
                                ApInfo_t *apinfo,
                                PtCallbackInfo_t *cbinfo )

/* This callback is used to launch a dialog box with the
   intent of exercising the global variable GcaCharter14Bold */

{
   PtNotice (ABW_base, NULL, "Font Demonstration", NULL,
             "This sentence is in 14-pt. Charter bold",
             GcaCharter14Bold, "OK", NULL, 0);

   /* Eliminate 'unreferenced' warnings */
   widget = widget, apinfo = apinfo, cbinfo = cbinfo;

   return( Pt_CONTINUE );
}
```

Writing text in a rectangular area

Writing text in a rectangle of a specific size can be tricky if the string size is unknown.

Consider a rectangle of fixed dimensions, for example a cell in a spreadsheet. How do you determine how many characters can successfully be displayed in this cell without clipping? Call *PfExtentTextToRect()*. Give it a clipping rectangle, a font identifier, a string, and the maximum number of bytes within the string, and it tells you the number and extent of the characters that fit within the clipping rectangle.

This is useful for placing an ellipsis (`...`) after a truncated string and avoiding partially clipped characters. Currently this routine supports clipping only along the horizontal axis.

Here's an example:

```
/* PfExtentTextToRect */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <Ap.h>
#include <Ph.h>
#include <Pt.h>
#include <errno.h>

PtWidget_t * pwndMain = NULL, * pbtn = NULL, * pobjRaw = NULL;
char * pcText = "pAfaBfbfffffffffffffffCfcXfxYfyZfzf";
char * pcGB = "\323\316\317\267";
char ** ppcData = NULL;

int fnDrawCanvas( PtWidget_t * ptsWidget, PhTile_t * ptsDamage );

#define FALSE 0

FontName szFont;

char * pmbGB = NULL;
struct PxTransCtrl * ptsTrans = NULL;
int iTemp1 = 0, iTemp2 = 0;

#define BUFFER_SIZE 256

int main (int argc, char *argv[])
{   PtArg_t args[4];
    PhPoint_t win_size, pntPOS, pntDIM;
    short nArgs = 0;

    if((pmbGB = calloc(BUFFER_SIZE, sizeof(char))) == NULL)
      return(EXIT_FAILURE);

    PtInit (NULL);

    if(argc > 1)
    {  if(PfGenerateFontName(argv[1], 0, 9, szFont) == NULL)
         PfGenerateFontName("TextFont", 0, 9, szFont);
    }
    else
      PfGenerateFontName("TextFont", 0, 9, szFont);

    if((ptsTrans = PxTranslateSet(NULL, "GB2312-80")) == NULL)
      return(EXIT_FAILURE);

    if(PxTranslateToUTF(ptsTrans, pcGB, 4, &iTemp1, pmbGB,
                        BUFFER_SIZE, &iTemp2) == -1)
      printf("Could not translate from GB to UTF.\n");

    if(argc > 2)
      pcText = pmbGB;

    /* Set the base pwndMain parameters. */
    win_size.x = 450;
    win_size.y = 450;

    PtSetArg(&args[0],Pt_ARG_DIM, &win_size, 0);
    PtSetArg(&args[1],Pt_ARG_WINDOW_TITLE,
             "PfExtentTextToRect", 0);

    pwndMain = PtCreateWidget (PtWindow, Pt_NO_PARENT, 2, args);

    nArgs = 0;
    pntPOS.x = 100;
    pntPOS.y = 10;
    PtSetArg(&args[nArgs], Pt_ARG_POS, &pntPOS, 0);
    nArgs++;
    PtSetArg(&args[nArgs], Pt_ARG_TEXT_STRING, pcText, NULL);
    nArgs++;
    PtSetArg(&args[nArgs], Pt_ARG_TEXT_FONT, szFont, NULL);
    nArgs++;
    pbtn = PtCreateWidget(PtButton, pwndMain, nArgs, args);
    PtRealizeWidget(pbtn);

    pntPOS.y = 100;
    pntPOS.x = 75;
    pntDIM.x = 300;
    pntDIM.y = 300;
    PtSetArg(&args[0], Pt_ARG_POS, &pntPOS, 0);
    PtSetArg(&args[1], Pt_ARG_DIM, &pntDIM, 0);
    PtSetArg(&args[2], Pt_ARG_RAW_DRAW_F, fnDrawCanvas, 0L);
    pobjRaw = PtCreateWidget(PtRaw, pwndMain, 3, args);

    PtRealizeWidget(pwndMain);

    PtMainLoop ();

    return(0);
}

#define ASCENDER tsExtent.ul.y
#define DESCENDER tsExtent.lr.y

int fnDrawCanvas( PtWidget_t * ptsWidget, PhTile_t * ptsDamage )
{   PhRect_t tsExtentClip;
    PhRect_t rect;
    PhPoint_t pnt;
    PhRect_t tsExtent;
    PgColor_t old;
    PhPoint_t pnt2;
    PhPoint_t tsPos = {0, 0};
    int iRet = 0;
    int iBytes = 0;

    /* Find our canvas. */
    PtBasicWidgetCanvas(pobjRaw, &rect);
    PtSuperClassDraw( PtBasic, ptsWidget, ptsDamage );

    old = PgSetStrokeColor(Pg_BLACK);

    PfExtentText(&tsExtent, &tsPos, szFont, pcText,
                 strlen(pcText));

    /* Draw the text. */
    pnt.x = 10 + rect.ul.x;
    pnt.y = 100 + rect.ul.y;
```

```
   PgSetFont(szFont);
   PgSetTextColor(Pg_BLACK);
   PgDrawText(pcText, strlen(pcText), &pnt, 0);

   pnt.x -= 10;
   pnt2.x = pnt.x + tsExtent.lr.x + 20;
   pnt2.y = pnt.y;

   PgSetStrokeColor(Pg_BLUE);

   PgDrawLine(&pnt, &pnt2);

   pnt.x = 10 + rect.ul.x;
   pnt.y = 100 + rect.ul.y;

   PgSetStrokeColor(Pg_RED);

   PgDrawIRect(tsExtent.ul.x + pnt.x,
               tsExtent.ul.y + pnt.y,
               (tsExtent.lr.x - min(tsExtent.ul.x, 0)
               + 1) + pnt.x, tsExtent.lr.y + pnt.y,
               Pg_DRAW_STROKE);

   if((iRet = PfExtentTextToRect(&tsExtentClip, szFont,
                 &tsExtent, pcText, strlen(pcText))) == -1)
     printf("PfExtentTextToRect failed 1.\n");
   else
   {  printf("lrx == %d, %d characters in string.\n",
             tsExtent.lr.x, utf8strlen(pcText, &iBytes));
      printf("PfExtentTextToRect lrx == %d, %d characters will\
fit in clip of %d.\n", tsExtentClip.lr.x, iRet, tsExtent.lr.x);
   }

   tsExtent.lr.x /= 2;

   if((iRet = PfExtentTextToRect(&tsExtentClip, szFont,
                 &tsExtent, pcText, strlen(pcText))) == -1)
     printf("PfExtentTextToRect failed 2.\n");
   else
   {  printf("lrx == %d, %d characters in string.\n",
             tsExtent.lr.x, utf8strlen(pcText, &iBytes));
      printf("PfExtentTextToRect lrx == %d, %d characters will\
fit in clip of %d.\n", tsExtentClip.lr.x, iRet, tsExtent.lr.x);
   }

   pnt.x = 10 + rect.ul.x;
   pnt.y = 150 + rect.ul.y;

   PgDrawText(pcText, iRet, &pnt, 0);
   PgDrawIRect(tsExtentClip.ul.x + pnt.x,
               tsExtentClip.ul.y + pnt.y,
               (tsExtentClip.lr.x - min(tsExtentClip.ul.x, 0)
               + 1) + pnt.x, tsExtentClip.lr.y + pnt.y,
               Pg_DRAW_STROKE);

   tsExtent.lr.x /= 2;

   if((iRet = PfExtentTextToRect(&tsExtentClip, szFont,
                 &tsExtent, pcText, strlen(pcText))) == -1)
     printf("PfExtentTextToRect failed 3.\n");
   else
   {  printf("lrx == %d, %d characters in string.\n",
             tsExtent.lr.x, utf8strlen(pcText, &iBytes));
      printf("PfExtentTextToRect lrx == %d, %d characters will\
 fit in clip of %d.\n", tsExtentClip.lr.x, iRet, tsExtent.lr.x);
    }

    pnt.x = 10 + rect.ul.x;
    pnt.y = 200 + rect.ul.y;

    PgDrawText(pcText, iRet, &pnt, 0);
    PgDrawIRect(tsExtentClip.ul.x + pnt.x,
                tsExtentClip.ul.y + pnt.y,
                (tsExtentClip.lr.x - min(tsExtentClip.ul.x, 0)
                + 1) + pnt.x, tsExtentClip.lr.y + pnt.y,
                Pg_DRAW_STROKE);

    PgSetStrokeColor(old);

    return( Pt_CONTINUE );
}
```

Repairing damage to proportional text

When dealing with proportional fonts, sometimes the vectors of one glyph run into the vectors of another. This is especially evident when using a font such as Nuptial BT. You need to take special care when repairing damage to such fonts.

PfExtentTextCharPositions() addresses this issue. You can use this routine to obtain the position after each character, incorporating the bearing x of the following character. This position is where you should draw the next character.

If you use the PF_CHAR_DRAW_POSITIONS flag, the bearing x of the following character isn't applied to the position, which is useful when you're placing cursors.

For example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <Ap.h>
#include <Ph.h>
#include <Pt.h>
#include <errno.h>

PtWidget_t * pwndMain = NULL,
           * pbtn = NULL,
           * pobjRaw = NULL,
           * pobjLabel = NULL;
char ** ppcData = NULL;

int fnDrawCanvas( PtWidget_t * ptsWidget,
                  PhTile_t * ptsDamage );

#define FALSE 0

#define __WIN_SIZE_X_ 1000

FontName szFont;

int main (int argc, char *argv[])
{   PtArg_t args[8];
```

```
PhPoint_t win_size, pntPOS, pntDIM;
short nArgs = 0;
char caTitle[50];

if(argc < 2)
{  printf("Usage:  pen text_string\n");
   exit(EXIT_FAILURE);
}

PtInit (NULL);

ppcData = argv;

PfGenerateFontName("TextFont", 0, 9, szFont);

/* Set the base pwndMain parms. */
win_size.x = 800;
win_size.y = 600;

sprintf(caTitle, "Get the pen position");
PtSetArg(&args[0],Pt_ARG_DIM, &win_size, 0);
PtSetArg(&args[1],Pt_ARG_WINDOW_TITLE, caTitle, 0);

pwndMain = PtCreateWidget (PtWindow, Pt_NO_PARENT, 2, args);

nArgs = 0;
pntDIM.x = 80;
pntDIM.y = 20;
PtSetArg(&args[nArgs], Pt_ARG_DIM, &pntDIM, 0);
nArgs++;
pntPOS.x = 100;
pntPOS.y = 10;
PtSetArg(&args[nArgs], Pt_ARG_POS, &pntPOS, 0);
nArgs++;
PtSetArg(&args[nArgs], Pt_ARG_TEXT_STRING, argv[1], NULL);
nArgs++;
pbtn = PtCreateWidget(PtButton, pwndMain, nArgs, args);
PtRealizeWidget(pbtn);

nArgs = 0;
pntDIM.x = 80;
pntDIM.y = 20;
PtSetArg(&args[nArgs], Pt_ARG_DIM, &pntDIM, 0);
nArgs++;
pntPOS.x = 100;
pntPOS.y = 600;
PtSetArg(&args[nArgs], Pt_ARG_POS, &pntPOS, 0);
nArgs++;
PtSetArg(&args[nArgs], Pt_ARG_TEXT_STRING, argv[1], NULL);
nArgs++;
PtSetArg(&args[nArgs], Pt_ARG_RESIZE_FLAGS,
         Pt_RESIZE_XY_ALWAYS, Pt_RESIZE_XY_ALWAYS);
nArgs++;
PtSetArg(&args[nArgs], Pt_ARG_BORDER_WIDTH, 0L, 0L);
nArgs++;
PtSetArg(&args[nArgs], Pt_ARG_MARGIN_LEFT, 0L, 0L);
nArgs++;
PtSetArg(&args[nArgs], Pt_ARG_MARGIN_RIGHT, 0L, 0L);
nArgs++;
pobjLabel = PtCreateWidget(PtLabel, pwndMain, nArgs, args);
PtRealizeWidget(pobjLabel);

pntPOS.y = 100;
```

```
    pntPOS.x = 75;
    pntDIM.x = __WIN_SIZE_X_ - 75 - 10;
    pntDIM.y = 300;
    PtSetArg(&args[0], Pt_ARG_POS, &pntPOS, 0);
    PtSetArg(&args[1], Pt_ARG_DIM, &pntDIM, 0);
    PtSetArg(&args[2], Pt_ARG_RAW_DRAW_F, fnDrawCanvas, 0L);
    pobjRaw = PtCreateWidget(PtRaw, pwndMain, 3, args);

    (void) PtRealizeWidget(pwndMain);

    PtMainLoop ();

    return(0);
}

int fnDrawCanvas( PtWidget_t * ptsWidget, PhTile_t * ptsDamage )
{   unsigned char const * pucFont = NULL;
    int * piIndx = NULL;
    int * piPos = NULL;
    char ** argv = (char **)ppcData;
    PhRect_t rect;
    PhPoint_t pnt;
    PhPoint_t tsPos = {0, 0};
    PhRect_t tsExtent;
    short n = 0;
    char * pc = NULL;
    PgColor_t old;

    pucFont = szFont;
    pc = argv[1];
    piIndx = (int *)calloc(50, sizeof(int));
    piPos = (int *)calloc(50, sizeof(int));

    if(strlen(pc) < 4)
    {  printf("Pick a longer string, must be at least\
 4 characters.\n");
       exit(EXIT_SUCCESS);
    }

    for(n = 0; n < strlen(pc); n++)
      piIndx[n] = n + 1;

    /* Find our canvas. */
    PtBasicWidgetCanvas(pobjRaw, &rect);

    old = PgSetStrokeColor(Pg_BLACK);

    PfExtentText(&tsExtent, &tsPos, pucFont, pc, strlen(pc));

    PgSetFont(pucFont);
    PgSetTextColor(Pg_BLACK);

    for(n = 0; n < strlen(pc); n++)
      piIndx[n] = n + 1;

    /* Draw the string, one character at a time. */
    PfExtentTextCharPositions(&tsExtent, &tsPos, pc,
                              pucFont, piIndx, piPos,
                              strlen(pc), 0L, 0, 0, NULL);
    pnt.x = 10 + rect.ul.x;
    pnt.y = 200 + rect.ul.y;

    PgDrawIRect(tsExtent.ul.x + pnt.x,
                tsExtent.ul.y + pnt.y,
                (tsExtent.lr.x - min(tsExtent.ul.x, 0) + 1) +
                pnt.x, tsExtent.lr.y + pnt.y, Pg_DRAW_STROKE);

    for(n = 0; n < strlen(pc); n++)
    {  PgDrawText(pc + n, 1, &pnt, 0);
       pnt.x = 10 + rect.ul.x + piPos[n];
       printf("Single[%d]:  %d\n", n, piPos[n]);
    }
    /* End draw one character at a time. */

    /* Draw the string, then overlay individual characters on
       top from right to left. */
    printf("Overlay test.\n");

    PfExtentText(&tsExtent, &tsPos, pucFont, pc, strlen(pc));
    pnt.x = 10 + rect.ul.x;
    pnt.y = 400 + rect.ul.y;

    PgDrawIRect(tsExtent.ul.x + pnt.x,
                tsExtent.ul.y + pnt.y,
                (tsExtent.lr.x - min(tsExtent.ul.x, 0) + 1) +
                 pnt.x, tsExtent.lr.y + pnt.y, Pg_DRAW_STROKE);

    PgSetFont(pucFont);
    PgSetTextColor(Pg_BLACK);
    PgDrawText(pc, strlen(pc), &pnt, 0);

    for(n = strlen(pc) - 1; n >= 0; n--)
    {  switch(n)
       {  case 0:  pnt.x = 10 + rect.ul.x;
                   PgDrawText(pc + 0, strlen(pc), &pnt, 0);
                   break;

          default: piIndx[0] = n;
                   PfExtentTextCharPositions(&tsExtent,
                      &tsPos, pc, pucFont, piIndx, piPos,
                      1, 0L, 0, 0, NULL);
                   printf("Position:  %d\n", piPos[0]);
                   pnt.x = 10 + rect.ul.x + piPos[0];
                   PgDrawText(pc + n, strlen(pc) - n, &pnt, 0);
                   PgFlush();
                   sleep(1);
                   break;
       }
    }
    /* End draw string, then overlay individual characters
       on top from right to left. */

   PgSetStrokeColor(old);
   free(piPos);
   free(piIndx);

   return( Pt_CONTINUE );
}
```

Chapter 22

Printing

In this chapter. . .

Printing and drawing are the same in Photon—the difference depends on the *draw context*, a data structure that defines where the draw stream (i.e. the draw events) flows:

- by default, to the graphics driver for drawing on the screen

  Or:

- to a *memory context* (or *MC*) for storing images in memory for later use

  Or:

- to a *print context* (or *PC*) for printing. See "Print Contexts," below.

To print in Photon:

1. Create a print context by calling *PpCreatePC()*.
2. Set up the print context automatically via the **`PtPrintSel`** widget, or programmatically via *PpSetPC()*.
3. Initialize the print job by calling *PpStartJob()*.
4. Any time after *PpStartJob()* is called, make the print context "active" by calling *PpContinueJob()*. When a print context is active, anything that's drawn via *PpPrintWidget()* or Pg* calls, including widgets, is directed to the file opened by the print context during the *PpStartJob()* call.
5. Insert page breaks, as required, by calling *PpPrintNewPage()*.
6. The print context can be made inactive without terminating the current print job by calling *PpSuspendJob()*, or by calling *PpContinueJob()* with a different print context. To resume the print job from where you left off, call *PpContinueJob()*.
7. Complete the print job by calling *PpEndJob()*.
8. When your application doesn't need to print anything else, call *PpReleasePC()* to free the print context.

Print contexts

A print context is a **`PpPrintContext_t`** structure whose members control how printing is to be done. For information about what's in a print context, see the Photon *Library Reference*.

Never directly access the members of a **`PpPrintContext_t`** structure; use *PpGetPC()* to extract members, and *PpSetPC()* to change them.

Creating a print context

The first step to printing in Photon is to create a print context by calling *PpCreatePC()*:

```
PpPrintContext_t *pc;

pc = PpCreatePC();
```

Modifying a print context

Once the print context is created, you must set it up properly for your printer and the options (orientation, paper size, etc.) you want to use. You can do this by calling:

- *PpLoadDefaultPrinter()*
- *PpLoadPrinter()*
- *PpSetPC()*
- *PtPrintPropSelect()*
- *PtPrintSelect()*
- *PtPrintSelection()*

These functions are described in the Photon *Library Reference*.

You can also use **PtPrintSel** (see the Photon *Widget Reference*).

You can get a list of available printers by calling *PpLoadPrinterList()*. When you're finished with the list, call *PpFreePrinterList()*.

Starting a print job

If you're using an application that needs to know anything about the print context, you can use *PpGetPC()* to get the appropriate information. For example, you might need to know the selected orientation (in order to orient your widgets properly). If you need to know the size of the margins, you can call *PpGetCanvas()*.

Before printing, you must set the source size or resolution. For example:

- If you want a widget to fill the page, set the source size to equal the widget's dimensions. You can call *PpSetCanvas()* to do this.
- by default, the source resolution is 100 pixels per inch so that fonts are printed at an appealing size. You can get the size of the interior canvas by calling *PpGetCanvas()*, which gives dimensions that take into account the marginal, nonprintable area.

When setting the source size, take the nonprintable area of the printer into account. All printers have a margin around the page that they won't print on, even if the page margins are set to 0. Therefore, the size set above is actually a bit larger than the size of a page, and the font will be scaled down to fit on the printable part of the page.

In the following example, the page size and nonprintable area are taken into account to give the proper source size and text height. Try this, and measure the output to prove the font is 1″ high from ascender to descender:

```
#include <stdio.h>
#include <stdlib.h>
#include <Pt.h>

PtWidget_t *label, *window;
PpPrintContext_t *pc;

int quit_cb (PtWidget_t *widget, void *data,
             PtCallbackInfo_t *cbinfo )
{
    exit (EXIT_SUCCESS);
    return (Pt_CONTINUE);
}

int print_cb (PtWidget_t *widget, void *data,
              PtCallbackInfo_t *cbinfo )
{
    int action;
    PhDim_t size;
    PhRect_t const *rect;
    PhDim_t const *dim;

    action = PtPrintSelection(window, NULL,
                              "Demo Print Selector",
                              pc, Pt_PRINTSEL_DFLT_LOOK);
    if (action != Pt_PRINTSEL_CANCEL)
    {
        /* Get the nonprintable area and page size. Both are in
           1/1000ths of an inch. */

        PpGetPC(pc, Pp_PC_NONPRINT_MARGINS, &rect);
        PpGetPC(pc, Pp_PC_PAPER_SIZE, &dim);
        size.w = ((dim->w  -
                  (rect->ul.x + rect->lr.x)) * 72) / 1000;
        size.h = ((dim->h  -
                  (rect->ul.y + rect->lr.y)) * 72) / 1000;

        /* Set the source size. */
        PpSetPC( pc, Pp_PC_SOURCE_SIZE, &size, 0);

        /* Start printing the label. */
        PpStartJob(pc);
        PpContinueJob(pc);

        /* Damage the widget. */
        PtDamageWidget(label);
        PtFlush();

        /* Close the PC. */
        PpSuspendJob(pc);
        PpEndJob(pc);
    }
    return (Pt_CONTINUE);
}

int main(int argc, char *argv[])
{
    PtArg_t args[10];
    PtWidget_t *print, *quit;
    PhDim_t   win_dim = { 400, 200 };
    PhPoint_t lbl_pos = {0, 0};
    PhArea_t print_area = { {130, 170}, {60, 20} };
    PhArea_t quit_area = { {210, 170}, {60, 20} };
    PtCallback_t callbacks[2] = { {print_cb, NULL},
                                  {quit_cb, NULL} };

    if (PtInit(NULL) == -1)
        PtExit(EXIT_FAILURE);

    /* Create the main window. */
    PtSetArg (&args[0], Pt_ARG_DIM, &win_dim, 0);
    PtSetArg (&args[1], Pt_ARG_WINDOW_TITLE,
              "Print Example", 0);

    if ((window = PtCreateWidget(PtWindow, Pt_NO_PARENT,
                                 1, args)) == NULL)
        PtExit (EXIT_FAILURE);

    /* Create a print context. */
    pc = PpCreatePC();

    /* Create a label to be printed. */
    PtSetArg (&args[0], Pt_ARG_POS, &lbl_pos, 0);
    PtSetArg (&args[1], Pt_ARG_TEXT_STRING,
              "I am 1 inch high", 0);
    PtSetArg (&args[2], Pt_ARG_TEXT_FONT, "swiss72", 0);
    PtSetArg (&args[3], Pt_ARG_MARGIN_HEIGHT, 0, 0);
    PtSetArg (&args[4], Pt_ARG_MARGIN_WIDTH, 0, 0);
    PtSetArg (&args[5], Pt_ARG_BEVEL_WIDTH, 0, 0);
    label = PtCreateWidget (PtLabel, window, 6, args);

    /* Create the print button. */
    PtSetArg(&args[0], Pt_ARG_AREA, &print_area, 0);
    PtSetArg(&args[1], Pt_ARG_TEXT_STRING, "Print", 0);
    PtSetArg(&args[2], Pt_CB_ACTIVATE, &callbacks[0], 0);
    print = PtCreateWidget (PtButton, window, 3, args);

    /* Create the quit button. */
    PtSetArg(&args[0], Pt_ARG_AREA, &quit_area, 0);
    PtSetArg(&args[1], Pt_ARG_TEXT_STRING, "Quit", 0);
    PtSetArg(&args[2], Pt_CB_ACTIVATE, &callbacks[1], 0);
    quit = PtCreateWidget (PtButton, window, 3, args);

    PtRealizeWidget(window);
    PtMainLoop();
    return (EXIT_SUCCESS);
}
```

You should also set the source offset, the upper left corner of what's to be printed. For example, if you have a button drawn at (20, 20) from the top left of a pane and you want it to be drawn at (0, 0) on the page, set the source offset to (20, 20). Any other widgets are drawn relative to their position from this widget's origin. A widget at (40, 40) will be drawn at (20, 20) on the page. The code is as follows:

```
PhPoint_t offset = {20, 20};
...
PpSetPC( pc, Pp_PC_SOURCE_OFFSET, &offset, 0 );
```

Once the source size and offset have been set, you can start printing:

```
PpStartJob(pc);
PpContinueJob(pc);
```

PpStartJob() sets up the print context for printing and *PpContinueJob()* makes the print context active, causing all Photon draw commands to be redirected to the destination specified in the print context.

Printing the desired widgets

After you've made the print context active, you can start printing widgets and so on. This can be done by calling any combination of the following:

- *Pg** functions
- *PpPrintWidget()* — you can even print a widget that hasn't been unrealized.

If you want to print all the contents of a widget that scrolls, you'll need to do some special preparations. See "Printing scrolling widgets" below.

Printing a new page

You can force a page break at any point by calling *PpPrintNewPage()*:

```
PpPrintNewPage(pc);
```

Note that once you call *PpStartJob()*, any changes to the print context take effect after the next call to *PpPrintNewPage()*.

Photon assumes that the page numbers increase by one. If this isn't the case, manually set the Pp_PC_PAGE_NUM member of the print context to the correct page number. Don't make the page number decrease because the print drivers might not work properly.

Printing widgets that scroll

If you want to print all the contents of a widget that scrolls, you need some special processing:

PtList

The only way to make a **PtList** print (or draw) all the items is by resizing it to be the total height of all the items. The easiest way is probably by using the resize policy:

This will work only if the total height is smaller than 65K pixels.

1. Open and start the print context.
2. Get the current resize flags (*Pt_ARG_RESIZE_FLAGS*) for the **PtList** widget.
3. Change the resize flags to Pt_RESIZE_XY_ALWAYS, to make the list resize to fit all of its text.

4. Call *PpPrintWidget()* for the widget or parent widget.
5. Reset the resize flags for the **`PtList`** widget.
6. Stop and close the print context.

`PtMultiText`

To print a **`PtMultiText`** widget's entire text, breaking the output into pages:

1. Create another multitext widget — let's call it the print widget — that isn't visible to the user (i.e. hide it behind the user's multitext widget).
2. Get the printer settings for printing: the orientation, page size, and the margins.
3. Adjust the printer settings for what you want and then use *PpSetPC()* to set them.
4. Set the print multitext widget's margins to match those of the printer that you just set.
5. Use *PpStartJob()* to start your print job.
6. Get the user's multitext widget resources (i.e. text, fonts, number of lines) and set the print multitext widget's resources to match them.
7. Go through the user's multitext and get the attributes for each line (color, font, tabs, etc) and set the print multitext widget's attributes accordingly.
8. Once you've set all of the attributes to match, specify the top line of the print multitext widget. This positions the widget to start printing.
9. Get the number of lines that are completely visible in the print multitext widget, as well as the total number of lines.
10. Use *PpContinueJob()*, *PpPrintWidget()*, and *PpSuspendJob()* to print the current page.
11. Delete the lines that you just printed from the print multitext widget. Doing this causes the next group of lines that you want to print to become the visible lines of the widget.
12. Call *PpPrintNewPage()* to insert a page break.
13. Continue printing pages and deleting the visible lines until you've reached the end of the text in the print multitext widget.

`PtScrollArea`

For a **`PtScrollArea`**, you need to print its *virtual canvas*, which is where all widgets created within or moved to a scroll area are placed:

1. Get a pointer to the virtual canvas by calling:

```
PtValidParent( ABW_Scroll_area, PtWidget );
```

2. Get the area (*Pt_ARG_AREA*) of the virtual canvas, and use its *size* member as the source size in the print context.
3. Set the print context's source offset to:

   ```
   PtWidgetOffset( PtValidParent( ABW_Scroll_area,
                                  PtWidget ));
   ```

4. Print the scroll area's virtual canvas by calling:

   ```
   PpPrintWidget( pc, PtValidParent( ABW_Scroll_area,
                                     PtWidget ),
                  NULL, NULL, 0);
   ```

Suspending and resuming a print job

To suspend a print job and direct all draw events back to the graphics driver at any point after calling *PpStartJob()*, call:

```
PpSuspendJob( pc );
```

To resume a print job, reactivating the print context, causing draw events to be directed towards the destination specified in the print context, call:

```
PpContinueJob( pc );
```

Ending a print job

When you're finished printing your widgets, the print context must be deactivated and closed. This is done by calling:

```
PpSuspendJob(pc);
PpEndJob(pc);
```

All draw events will be directed to the graphics driver.

You can reuse the print context for new print jobs, eliminating the need to create and initialize it again.

Freeing the print context

When printing is complete and you no longer need the print context, you can free it, which in turn frees any resources used by it.

If you want to remember any information from the print context for future use, save it by calling *PpGetPC()* before freeing the print context. For example:

```
short const *orientation;
...
PpGetPC( pc, Pp_PC_ORIENTATION, &orientation );
```

To free a print context, call:

```
PpReleasePC( pc );
```

Example

This example creates an application with a main window, and a pane with a few widgets on it. When you press the Print button, a Print Selection Dialog appears. When you select this dialog's Print or Preview button, the pane is "drawn" on the printer.

```
#include <stdio.h>
#include <stdlib.h>
#include <Pt.h>

PtWidget_t *pane, *window;
PpPrintContext_t *pc;

int quit_cb ( PtWidget_t *widget, void *data,
              PtCallbackInfo_t *cbinfo)
{
    PpReleasePC (pc);
    exit (EXIT_SUCCESS);
    return (Pt_CONTINUE);
}

int print_cb ( PtWidget_t *widget, void *data,
               PtCallbackInfo_t *cbinfo)
{
    int action;

    /* You could make these calls to PpSetPC() right
       after creating the print context. Having it here
       lets you reuse the print context. */
    PhDim_t size = { 850, 1100 };
    PhDim_t size2 = { 200, 150 };

    /* Set the source resolution to be proportional to
       the size of a page. */
    PpSetPC(pc, Pp_PC_SOURCE_SIZE, &size, 0);

    /* Uncomment this to set the source size to be the size
       of the widget. The widget will be scaled when printed. */
    /* PpSetPC(pc, Pp_PC_SOURCE_SIZE, &size2, 0); */

    action = PtPrintSelection(window, NULL,
                 "Demo Print Selector", pc,
                 Pt_PRINTSEL_DFLT_LOOK);
    if (action != Pt_PRINTSEL_CANCEL)
    {
        /* Start printing the pane. Note that we're using
           the same print context for all print jobs. */
        PpStartJob(pc);
        PpContinueJob(pc);
        /* Print the widget. */
        PpPrintWidget(pc, pane, NULL, NULL, 0);

        /* Close the print context. */
        PpSuspendJob(pc);
        PpEndJob(pc);
    }

    return (Pt_CONTINUE);
}

int main(int argc, char *argv[])
{
    PtArg_t args[4];
    PtWidget_t *print, *quit;
    PhDim_t  win_dim = { 200, 200 };
    PhArea_t pane_area = { {0, 0}, {200, 150} };
    PhArea_t print_area = { {30, 170}, {60, 20} };
    PhArea_t quit_area = { {110, 170}, {60, 20} };
    PhArea_t cir_area = { {35, 20}, {130, 110} };
    PhArea_t cir2_area = { {67, 40}, {20, 20} };
    PhArea_t cir3_area = { {110, 40}, {20, 20} };
    PhArea_t cir4_area = { {85, 80}, {30, 30} };
    PtCallback_t callbacks[2] = { {print_cb, NULL},
                                  {quit_cb, NULL} };

    if (PtInit(NULL) == -1)
        PtExit(EXIT_FAILURE);

    /* Create the main window. */
    PtSetArg (&args[0], Pt_ARG_DIM, &win_dim, 0);
    PtSetArg (&args[1], Pt_ARG_WINDOW_TITLE,
              "Print Example", 0);
    if ((window = PtCreateWidget(PtWindow, Pt_NO_PARENT,
                                 2, args)) == NULL)
        PtExit(EXIT_FAILURE);

    /* Create a print context. */
    pc = PpCreatePC();

    /* Create the pane to be printed. */
    PtSetArg (&args[0], Pt_ARG_AREA, &pane_area, 0);
    pane = PtCreateWidget (PtPane, window, 1, args);

    /* put some stuff in the pane to be printed. */
    PtSetArg (&args[0], Pt_ARG_AREA, &cir_area, 0);
    PtCreateWidget (PtEllipse, pane, 1, args);

    PtSetArg (&args[0], Pt_ARG_AREA, &cir2_area, 0);
    PtSetArg (&args[1], Pt_ARG_FILL_COLOR, Pg_BLACK, 0);
    PtCreateWidget (PtEllipse, pane, 2, args);

    PtSetArg (&args[0], Pt_ARG_AREA, &cir3_area, 0);
    PtSetArg (&args[1], Pt_ARG_FILL_COLOR, Pg_BLACK, 0);
    PtCreateWidget (PtEllipse, pane, 2, args);

    PtSetArg (&args[0], Pt_ARG_AREA, &cir4_area, 0);
    PtCreateWidget (PtEllipse, pane, 1, args);

    /* Create the print button. */
    PtSetArg(&args[0], Pt_ARG_AREA, &print_area, 0);
    PtSetArg(&args[1], Pt_ARG_TEXT_STRING, "Print", 0);
    PtSetArg(&args[2], Pt_CB_ACTIVATE, &callbacks[0], 0);
    print = PtCreateWidget (PtButton, window, 3, args);

    /* Create the quit button. */
    PtSetArg(&args[0], Pt_ARG_AREA, &quit_area, 0);
    PtSetArg(&args[1], Pt_ARG_TEXT_STRING, "Quit", 0);
    PtSetArg(&args[2], Pt_CB_ACTIVATE, &callbacks[1], 0);
    quit = PtCreateWidget (PtButton, window, 3, args);

    PtRealizeWidget(window);
    PtMainLoop();
    return (EXIT_SUCCESS);
}
```

Chapter 23

Drag and Drop

In this chapter. . .

Drag and drop lets you drag arbitrary data within an application or between applications.

If you simply need to drag graphical objects around, see "Dragging" in the Events chapter.

Transport mechanism

Photon's transport mechanism lets you transfer arbitrary data from one application to another, even if the applications are on different platforms with different endian-ness. This mechanism is used as the basis for drag and drop, but could be used for other purposes such as configuration files.

There are two ways to transport data:

Inline	The data is packed into a stream and sent to the destination.
By request	Descriptions of the data are packed into a stream and sent. The destination decides which type(s) of data it wants and sends the request back to the source, which then packs only the data required.

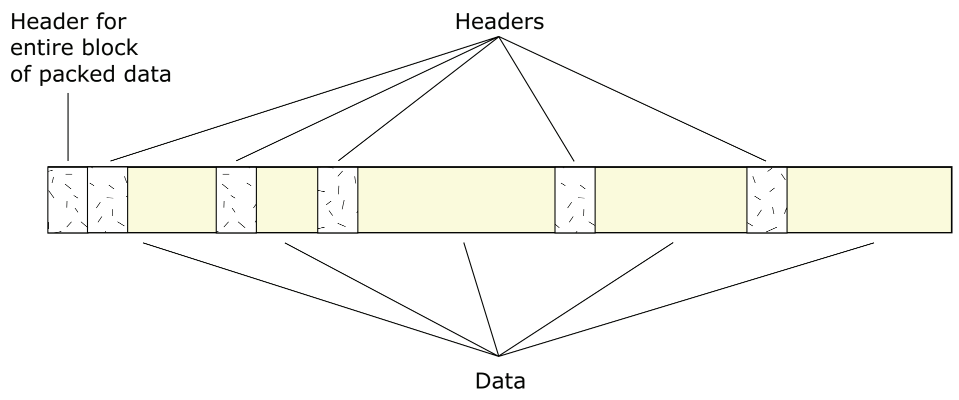
In order to transport data, the transport mechanism must pack data at the source application or widget and unpack it at the destination. It has to have a means of recognizing the type of data to determine what packing and unpacking must be done. This is accomplished via the *transport registry*.

There are a number of system-registered types that exist after *PtInit()* or *PtAppInit()* initializes the Photon library — this is done automatically for PhAB applications. The system-registered types are:

- **`string`**
- **`raw`**
- **`PhDim`**
- **`PhArea`**
- **`PhPoint`**
- **`PhImage`**

You can add other data types to the registry, as described in "Registering new transport types," later in this chapter.

The transport mechanism works by building a linked list of data to be transported, packing up the data into a stream, with each block preceded by a header that describes the data.



Packed data and headers.

When the data arrives at the destination, the headers are extracted to get the unpacking instructions for the data. The transport mechanism automatically unpacks the data; the application gets the data in its original form.

Using drag and drop

You can use drag and drop to move data from one widget to another, using Photon's transport mechanism. You can transport several types of data at once, giving the destination the choice of which ones to receive. All of the communication between the source and destination is nonblocking.

The basic steps (described in more detail in the sections that follow) are:

1. The user holds down the pointer button on the widget that's to be the source of the drag-and-drop operation.
2. In its *Pt_CB_OUTBOUND* callback, the source widget packs up the data to be dragged, and starts a drag-and-drop operation.
3. The user drags the data and decides to drop it on a widget.
4. In its *Pt_CB_DND* callback, the destination widget decides which pieces of dragged data (if any) it will accept. Any data that's accepted is unpacked automatically. The data is stored in allocated memory; the destination should free the memory when it doesn't need the data any more.

The source widget can also cancel the operation if it wishes.

Starting drag and drop

To start a drag-and-drop operation, the source widget must pack up the data to be dragged, and then initiate drag-and-drop. This is typically done in one of the widget's *Pt_CB_OUTBOUND* callbacks.

Pt_CB_OUTBOUND callbacks are invoked only if the widget has Pt_SELECTABLE set in its *Pt_ARG_FLAGS*.

The steps to follow are:

1. If the data isn't of one of the system-defined types of transport data, create a transport registry entry for it. For more information, see "Registering new transport types," below.
2. Create a transport control structure (of type **`PtTransportCtrl_t`**) for use with drag and drop, by calling *PtCreateTransportCtrl()*. The transport control structure is freed automatically when the drag-and-drop operation is done.
3. The data to be dragged can be packed *inline* (i.e. included directly in the structure passed to the destination) or it can be *requestable* (i.e. the data isn't packed up until the destination asks for it).
   - For each piece of data to be packed inline, call *PtTransportType()*.
   - For each piece of requestable data, call *PtTransportRequestable()*.

     The **`PtTransportCtrl_t`** structure has a list of *response data*, which is automatically sent if the destination requests it. The source widget can add data to this queue by calling *PtAddResponseType()*.

     If the source widget doesn't want to pack the requestable data at this point, it must provide a callback when calling *PtTransportRequestable()*.
4. When all the data is packed, call *PtInitDnd()* to initiate the drag-and-drop operation.

Example

Here's an example of a callback that initiates a drag-and-drop operation for a **`PtLabel`** widget. You can use this callback for the label widget's *Pt_CB_OUTBOUND* callback.

Be sure to set Pt_SELECTABLE in the label's *Pt_ARG_FLAGS*.

This callback sets up a drag-and-drop operation involving these pieces of data:

- The label's text, if any, packed as inline data.
- The label's image, if any, packed as requestable data. The image isn't transported until the destination asks for it, but the callback prepares the data in advance so we don't need a request callback.
- Alternate text to use if there's no image. This string is also requestable data, but we provide a request callback to pack it up if the destination requests it.

The callback assigns the same grouping number to the image and the alternate text, to indicate that they're different forms of the same data.

```
/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "abimport.h"
#include "proto.h"

#define TEXT_GROUP   0
#define IMAGE_GROUP  1

PtTransportReqDataCB_t request_callback;

int start_dnd( PtWidget_t *widget,
               ApInfo_t *apinfo,
               PtCallbackInfo_t *cbinfo )
{

    char *widget_text = NULL;
    char *label_type;
    PhImage_t * image = NULL;
    PtRequestables_t *req;
    PtTransportCtrl_t *tctrl =
        PtCreateTransportCtrl();
    int ret;

    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo;
    cbinfo = cbinfo;

    /* Get the type of label so we can determine
       what data to pack. */
    PtGetResource( widget, Pt_ARG_LABEL_TYPE,
                   &label_type, 0);

    if ((*label_type == Pt_Z_STRING) ||
        (*label_type == Pt_TEXT_IMAGE))
    {
      /* Get the widget's text and pack it inline. */
      PtGetResource( widget, Pt_ARG_TEXT_STRING,
                     &widget_text, 0);
      PtTransportType( tctrl, "text", "plain",
         TEXT_GROUP, Ph_TRANSPORT_INLINE, "string",
         widget_text, 0, 0);
    }

    /* If there's an image, add it as requestable
       data. Prepare the response data (to allow an
       automatic response). */

    if ((*label_type == Pt_IMAGE) ||
        (*label_type == Pt_TEXT_IMAGE))
    {
        PtGetResource( widget, Pt_ARG_LABEL_IMAGE,
                       &image, 0);
        if (image)
```

```
            {
               req = PtTransportRequestable ( tctrl,
                       "image", "an image", IMAGE_GROUP,
                       Ph_TRANSPORT_INLINE, "PhImage",
                       NULL, NULL );
               PtAddResponseType( tctrl, req, "image",
                  "an image", Ph_TRANSPORT_INLINE,
                  "PhImage", image, 0, 0);
            }
        }

        /* Add a requestable string that will be
           provided by a callback. */

        PtTransportRequestable( tctrl, "text",
           "image description", IMAGE_GROUP,
           Ph_TRANSPORT_INLINE, "string",
           (PtTransportReqDataCB_t *) &request_callback,
           "This was requested");

        /* Initiate the drag and drop. */

        ret = PtInitDnd( tctrl, widget, cbinfo->event,
                         NULL, 0);
        return( Pt_CONTINUE );
    }

    int unsigned request_callback( int unsigned type,
                     PtReqResponseHdr_t *req_hdr,
                     PtRequestables_t *req)
    {
       if (type == Pt_DND_REQUEST_DATA)
       {
          /* Respond to the request with the string in
             req->rq_callback_data, the last argument
             to PtTransportRequestable(). */

          PtAddResponseType( req->ctrl, req, "text",
             "request", Ph_TRANSPORT_INLINE, "string",
             req->rq_callback_data, 0, 0);
          return Pt_CONTINUE;
       }

       /* Deny the request. */
       return Pt_END;
    }
```

Receiving drag-and-drop events

To make a widget able to receive drag-and-drop events, attach a *Pt_CB_DND* callback to the widget (see **PtWidget** in the Photon *Widget Reference*).

A widget doesn't have to have Pt_SELECTABLE set in its *Pt_ARG_FLAGS* for its *Pt_CB_DND* callbacks to be invoked.

Whenever the widget is involved in a drag-and-drop event in some fashion, its *Pt_CB_DND* callback is invoked. In the callback, the *cbinfo*->*reason_subtype* indicates the type of drag-and-drop action that's occurring.

The sections below describe the drag-and-drop events that are of interest to the source and destination widgets. Of course, if a widget can be the source and destination of (separate) drag-and-drop operations, its *Pt_CB_DND* callbacks need to have both sets of events.

For more information about events, see **`PhEvent_t`** in the Photon *Library Reference*.

Source widget

The source widget of a drag-and-drop operation can receive events that describe the status of the operation. If you don't want these events, set Pt_DND_SILENT in the *flags* argument to *PtInitDnd()*.

Don't set Pt_DND_SILENT if you've included requestable data in the control structure.

The subtypes of a drag-and-drop event that are of interest to the source of the operation are:

Ph_EV_DND_INIT

> The operation has started successfully.

Ph_EV_DND_CANCEL

> The operation was canceled (for example, if the drop occurred when not over a drop zone, or the destination terminated the operation before receiving the drop or before it finished fetching requestable data).
>
> If the operation is canceled in this way, the library cleans up the data structures automatically.

Ph_EV_DND_COMPLETE

> The drag-and-drop event is enqueued at the destination (the destination hasn't seen it yet).

Ph_EV_DND_DELIVERED

> The destination has dequeued the drag-and-drop event.

Destination widget

The subtypes of a drag-and-drop event that are of interest to the destination of the operation are:

Ph_EV_DND_ENTER

> Someone has dragged some data into the widget's region but hasn't yet released it. This is the *reason_subtype* the first time that the drag-and-drop callback is called.
>
> At this time, your application decides if it will accept the data to be dropped. It must build an array of **`PtDndFetch_t`** structures and pass it to *PtDndSelect()*.

This array describes the acceptable types, descriptions, and transport methods for drag-and-drop data to be accepted by a widget. *PtDndSelect()* returned the number of selected items from the array. If the event contains data, or references to data, in an acceptable format, those pieces of the event are selected.

If none of the data is acceptable, this widget isn't notified of any other events for the current drag-and-drop operation.

Ph_EV_DND_MOTION

The pointer is moving inside the widget's region. This type of event is emitted only if the Pt_DND_SELECT_MOTION bit is set in the *select_flags* member of the **`PtDndFetch_t`** structure for a piece of selected data.

Ph_EV_DND_DROP

The user has dropped the data.

For this *reason_subtype*, the callback should retrieve the selected data from the event. This might involve some automatic, nonblocking communication with the source of the data — to prevent any communication with the source, specify Ph_TRANSPORT_INLINE as the only acceptable transport protocol.

If the drop is successful, the memory used by the transport mechanism is automatically freed.

Ph_EV_DND_LEAVE

The pointer has moved out of the widget's region, but the user didn't drop the data.

Here's an example that works with the callback given above for a **`PtLabel`** widget. This callback accepts the following from the drag-and-drop data:

- text
- an image
- an alternate string if there's no image in the data (Pt_DND_SELECT_DUP_DATA is set in the **`PtDndFetch_t`** entry).

The source widget packed the image and the alternate text as requestable data, but the destination doesn't have to do anything to request it; the transport mechanism does it automatically.

```
/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>
```

```
/* Local headers */
#include "abimport.h"
#include "proto.h"

static PtDndFetch_t stuff_i_want[] =
{
   {"text", "plain", Ph_TRANSPORT_INLINE, },
   {"image", NULL, Ph_TRANSPORT_INLINE, },
   {"text", "image description",
    Ph_TRANSPORT_INLINE,
    Pt_DND_SELECT_DUP_DATA, },
};

enum {
     PLAIN_TEXT = 0,
     IMAGE,
     IMAGE_TEXT,
};


int dnd_callback( PtWidget_t *widget,
                  ApInfo_t *apinfo,
                  PtCallbackInfo_t *cbinfo )

{
  PtDndCallbackInfo_t *dndcb = cbinfo->cbdata;
  int deep_free = 1, num_matches;

  /* eliminate 'unreferenced' warnings */
  widget = widget, apinfo = apinfo;
  cbinfo = cbinfo;

  switch (cbinfo->reason_subtype)
  {
    case Ph_EV_DND_ENTER:
      num_matches = PtDndSelect (widget,
         stuff_i_want,
         sizeof( stuff_i_want ) / sizeof( stuff_i_want[0] ),
         NULL, NULL, cbinfo );
      break;

    case Ph_EV_DND_DROP:
      switch (dndcb->fetch_index)
      {
        case PLAIN_TEXT:
          PtSetResource (widget, Pt_ARG_TEXT_STRING,
            dndcb->data, strlen(dndcb->data));
          break;

        case IMAGE:
          PtSetResource (widget, Pt_ARG_LABEL_IMAGE,
            dndcb->data, 0);
            free (dndcb->data);
          deep_free = 0;
          break;

        case IMAGE_TEXT:
          printf (
             "No image; the alternate text is: %s\n",
             (char *)dndcb->data);
          break;
      }
```

```
        if (deep_free) {
           PhFreeTransportType (dndcb->data,
              dndcb->trans_hdr->packing_type);
        }
        break;

  }

  return( Pt_CONTINUE );
}
```

Canceling drag and drop

The source widget can cancel the drag-and-drop operation by calling *PtCancelDnd()*.

The widget must then clean up the transport-control structures and the packed data by calling *PtReleaseTransportCtrl()*. (If the drop is successfully done, the control structures are cleaned up automatically. The destination decides when to free the dropped data.)

Registering new transport types

To transport data other than the types automatically defined by Photon, you must define the type and register it with the *transport registry*, a collection of type descriptions, each of which includes:

- a type name in the form of a string, e.g. **PhImage**
- the packing method to be used (one of Ph_PACK_RAW, Ph_PACK_STRING, or Ph_PACK_STRUCT)
- a list of members within the type that reference data outside the base size of the type in question (reference or pointer-type members)
- a list of members that are endian-sensitive (these members are endian corrected if ever unpacked on a machine whose endian-ness differs from that on the machine where the data was packed)
- a list of members that should be cleared when the type is unpacked (for example, a pointer to data — such as a password — that you don't want to be transported).

The source and destination applications must both define the data type in their transport registries before the data can be successfully transported.

A simple data structure

Let's consider a simple data structure:

```
typedef struct simp1 {
    int num;
    int nums_10[10];
    char name[10];
    short vals_5[5];
} Simp1_t;
```

This structure could easily be packed using the **raw** type because it doesn't make any external references (i.e. it has no pointer members). But that doesn't protect the transported data from endian differences between the source and destination. So even for this simple structure, a type description detailing its endian sensitivity is beneficial.

The type definition starts with an array of **int unsigned** entries that described the endian-sensitivity for each member:

```
static const int unsigned Simp1Endians[] = {
    Tr_ENDIAN( Simp1_t, num ),
    Tr_ENDIAN_ARRAY( Simp1_t, nums_10 ),
    Tr_ENDIAN_ARRAY( Simp1_t, vals_5 ),
    0 /* End of the endian list */
};
```

Note that this list must end with an entry of 0. The *name* member isn't endian-sensitive, so it isn't included in the list.

All types or references to types correct the endian-ness of their members based on the endian array defined for the type. The classifications of endian-sensitive members are:

Tr_ENDIAN(typedef_name, member)

: **int**, **long**, **short**, (**signed** or **unsigned**). For example, **unsigned int** *my_scalar*.

Tr_ENDIAN_ARRAY(typedef_name, member)

: Array of **short** or **int** entries. For example, **short** *short_nums*[10].

Tr_ENDIAN_REF(typedef_name, member, num)

: A reference to endian scalars. For example, **int** **nums*.

Having defined the endian list for our simple data type, let's create the definition to go into the transport registry:

```
static const PhTransportRegEntry_t Simp1TransDef = {
    "simp1",
    Ph_PACK_STRUCT,
    sizeof( Simp1_t ),
    0,
    NULL,
    &Simp1Endians,
    NULL
};
```

The **PhTransportRegEntry_t** structure includes the following members:

**char ***_type_ The name of the type being registered.

int unsigned *packing*

: The packing method to be used (one of Ph_PACK_RAW, Ph_PACK_STRING, or Ph_PACK_STRUCT).

int unsigned *size*

The size, in bytes, of the data type.

int unsigned *num_fixups*

The number of entries in the *fixups* arrays.

PhTransportFixupRec_t const ******fixups*

A list of instructions for dealing with references to data outside the type being defined. We'll discuss this later.

int unsigned const ******endians*

The zero-terminated array of endian information described above.

int unsigned const ******clear_refs*

The zero-terminated array of members that should be cleared (i.e. set to NULL) when this type is unpacked.

To register this newly defined type, call *PhRegisterTransportType()*:

```
PhRegisterTransportType( &Simp1TransDef );
```

This new type, **simp1**, can now be used with any of the transport functions to pack or unpack data.

The destination application doesn't need to concern itself with the endian orientation of the source. When the destination unpacks this type, the transport mechanism automatically corrects the endian-ness using the endian definition in the registered transport type. This is particularly beneficial in a multiplatform, networked environment. If the transport mechanism is used to write binary configuration files, the same files can be used by applications regardless of the endian orientation of the machine they are running on.

A more complicated structure

You'll frequently need to transport more complex data types that have references to data external to themselves (pointer members). These members need special handling when performing packing and unpacking operations. In order for these members to get the treatment they deserve, they must be described in the *fixup* member of the entry in the transport registry.

Here's a more complicated structure:

```
typedef struct simp2 {

    /* Scalar and reference to a scalar array */
    int num_ref_vals;
    int *ref_vals;

    /* Scalar array */
    int nums_10[10];

    /* Scalar array (not endian sensitive) */
    char first_name[10];

    /* Reference to a string */
    char *last_name2;

    /* Scalar array */
    short vals_5[5];

    /* Registered type member */
    Simp1_t simp1_instance;

    /* Registered type member array */
    Simp1_t simp1_array[4];

    /* Reference to a registered type */
    Simp1_t *simp1_reference;

    /* Scalar and reference to a registered
       type array */
    int     num_simps;
    Simp1_t *ref_simp1_array;

    /* Scalar and reference to a registered
       type reference array */
    int     num_simp_refs;
    Simp1_t **ref_simp1_ref_array;

    /* Two scalars and a reference to arbitrarily
       sized data */
    short bm_height;
    int bm_bpl;
    char *bitmap;

    /* Something we don't want packed, but want
       cleared when unpacking */
    char *dont_pack_this;

} Simp2_t;
```

Clear-references list

Here's the *clear_refs* list for this structure:

```
static const int unsigned Simp2ClearRefs[] = {
    offsetof( Simp2_t, dont_pack_this ),
    0 /* End of the clear-refs list */
    };
```

Endian list

Here's the endian list for this structure:

```
static const int unsigned Simp2Endians[] = {
    Tr_ENDIAN( Simp2_t, num_ref_vals ),
    Tr_ENDIAN_REF( Simp2_t, ref_vals ),
    Tr_ENDIAN_ARRAY( Simp2_t, nums_10 ),
    Tr_ENDIAN_ARRAY( Simp2_t, vals_5 ),
    0 /* End of the endian list */
    };
```

Here's the full list of endian manifests for each type of member:

Scalar (**`char`**)	None
Scalar (**`short`**)	**`Tr_ENDIAN(`** *type*, *member* **`)`**
Scalar (**`int`**)	**`Tr_ENDIAN(`** *type*, *member* **`)`**
Scalar Array (**`char`**)	None
Scalar Array (**`short`**)	**`Tr_ENDIAN_ARRAY(`** *type*, *member* **`)`**
Scalar Array (**`int`**)	**`Tr_ENDIAN_ARRAY(`** *type*, *member* **`)`**
Reference (**`char`**)	None
Reference (**`short`**)	**`Tr_ENDIAN_REF(`** *type*, *member* **`)`**
Reference (**`int`**)	**`Tr_ENDIAN_REF(`** *type*, *member* **`)`**
Reference (**`char`** array)	None
Reference (**`short`** array)	**`Tr_ENDIAN_REF(`** *type*, *member* **`)`**
Reference (**`int`** array)	**`Tr_ENDIAN_REF(`** *type*, *member* **`)`**
Simple structure	List each endian-sensitive member of the member structure
Registered type	None
Reference to registered type	None

For example, for a **`Sample_t`** structure:

`int` *i*;	**`Tr_ENDIAN( Sample_t, i )`**
`int` *array*[7];	**`Tr_ENDIAN_ARRAY( Sample_t, array )`**
`short *`*short_nums*;	**`Tr_ENDIAN_REF( Sample_t, short_nums )`**

int **long_nums*;	`Tr_ENDIAN_REF( Sample_t, long_nums )`

struct my_simp *ms*;	`Tr_ENDIAN( Sample_t, ms.width ),`
	`Tr_ENDIAN( Sample_t, ms.height )`

Fixup list

The **Simp2_t** structure given earlier includes some entries that reference data outside the structure. These elements need **PhTransportFixupRec_t** entries in the *fixup* list to tell the transport mechanism how to get the data:

```
static const PhTransportFixupRec_t
Simp2Fixups[] = {
    Tr_REF_ARRAY( Simp2_t, ref_vals,
                  Tr_FETCH( Simp2_t, num_ref_vals ) ),
    Tr_STRING( Simp2_t, name2 ),
    Tr_TYPE( Simp2_t, simp1_instance ),
    Tr_TYPE_ARRAY( Simp2_t, simp1_array ),
    Tr_REF_TYPE( Simp2_t, simp1_reference ),
    Tr_REF_TYPE_ARRAY(
        Simp2_t, ref_simp1_array,
        Tr_FETCH( Simp2_t, num_simps ) ),
    Tr_REF_TYPE_REF_ARRAY(
        Simp2_t, ref_simp1_ref_array,
        Tr_FETCH( Simp2_t, num_simp_refs ) ),
    Tr_ALLOC( Simp2_t, bitmap,
            Tr_FETCH( Simp2_t, bm_bpl ), '*',
            Tr_FETCH( Simp2_t, bm_height ) )
    };
```

When defining a fixup entry, you might need to use information within the structure that the entry is defining. In these circumstances, use this manifest:

Tr_FETCH(*type*, *member*)

This manifest causes the value of the specified member to be used at runtime when data is being packed or unpacked. Also, any members that are defined via other registered types are automatically endian corrected using the endian definition from that type's transport registry entry.

Here's the full list of fixup manifests:

Scalar	None.
Scalar Array	None.
Reference (string)	**Tr_STRING(** *type*, *member* **)**
Reference (scalar array)	**Tr_REF_ARRAY(** *type*, *member*, *number_of_elements* **)**
Registered type	**Tr_TYPE(** *type*, *member*, *type_name* **)**

Registered type array

```
Tr_TYPE_ARRAY( type, member, type_name )
```

Reference (registered type)

```
Tr_REF_TYPE( type, member, type_name )
```

Reference (registered type array)

```
Tr_REF_TYPE_ARRAY( type, member, num_elements, \
                   type_name )
```

Reference(registered type reference array)

```
Tr_REF_TYPE_REF_ARRAY( type, member, \
                       num_elements, type_name )
```

Here are some sample members and their fixup manifests:

char **name*;

```
Tr_STRING( Sample_t, name )
```

int *num_nums*;
int **int_array*;

```
Tr_REF_ARRAY( Sample_t, int_array, \
              Tr_FETCH( Sample_t, num_nums) )
```

or if the number is known:

```
Tr_REF_ARRAY( Sample_t, int_array, 7 )
```

Simp1_t *simple_instance*

```
Tr_TYPE( Sample_t, simple_instance, "simp1" )
```

or, as a single instance if it's just an array of one:

```
Tr_TYPE_ARRAY( Sample_t, simple_instance, \
               "simp1" )
```

Simp1_t *simple_array*[5]

```
Tr_TYPE_ARRAY( Sample_t, simple_array, "simp1" )
```

**Simp1_t ***_simp1_ref_

```
Tr_REF_TYPE( Sample_t, simp1_ref, "simp1" )
```

short *num_simp1s*;
**Simp1_t ***_simp1_ref_

```
Tr_REF_TYPE_ARRAY( Sample_t, simp1_ref, \
    Tr_FETCH( Sample_t, num_simp1s ), "simp1" )
```

short *num_simp1s*;
**Simp1_t *****simp1_ref*;

```
                    Tr_REF_TYPE_REF_ARRAY( Sample_t, simp1_ref, \
                        Tr_FETCH( Sample_t, num_simp1s ), "simp1" )
```

Registry entry

Finally, here's the registry entry for **Simp2_t**:

```
static const PhTransportRegEntry_t
Simp2TransDef = {
    "simp2",
    Ph_PACK_STRUCT,
    sizeof( Simp2_t ),
    sizeof(Simp2Fixups)/sizeof(Simp2Fixups[0]),
    &Simp2Fixups,
    &Simp2Endians,
    &Simp2ClearRefs
    };
```

Transport functions

This section describes the low-level functions and data types that deal with the transport mechanism. Some functions are called by the application that's the source of the data, some are called by the destination, and some are called by both.

Both applications

Both applications use these:

PhTransportRegEntry_t
: Data structure that describes data to be transported

PhRegisterTransportType()
: Add a new transport type to the transport registry

PhFindTransportType()
: Find a transport type in the transport registry

Source application

The source application uses these, in roughly this order:

PhTransportCtrl_t
: Control structure for the Photon transport mechanism

PhCreateTransportCtrl()
: Allocate a *PhCreateTransportCtrl()* structure

PhTransportType()	Pack data into a **`PhTransportCtrl_t`** structure
PhTransportFindLink()	Search a linked list of transport data for some specific data
`PhTransportLink_t`	Entry in a linked list of transport data
PhLinkTransportData()	Add transport data to a linked list
PhGetNextInlineData()	Get the data for the next entry in a linked list of transport data
PhGetTransportVectors()	Build an I/O vector of data to be transported
PhFreeTransportType()	Free data associated with a transport registry entry
PhReleaseTransportCtrl()	Free a **`PhTransportCtrl_t`** structure

These are low-level functions that you'll probably never need to call directly:

PhAllocPackType()	Allocate a buffer and pack transport data into it
PhPackEntry()	Pack transport data, given a transport registry entry
PhPackType()	Pack transport data, given the type of data

Destination application

The destination application uses these, in roughly this order:

PhGetAllTransportHdrs()	Extract all the headers from a buffer of packed transport data
PhGetTransportHdr()	Extract the header from a buffer of packed transport data
PhGetNextTransportHdr()	Get the next header from a buffer of packed transport data
PhLocateTransHdr()	Look for specific data in a linked list of transport headers

PhMallocUnpack()	Unpack transport data, using a custom memory-allocation function
PhUnpack()	Unpack transport data
PhUnlinkTransportHdr()	Remove an entry from a linked list of transport headers
PhReleaseTransportHdrs()	Free a linked list of headers for packed transport data

Chapter 24

Regions

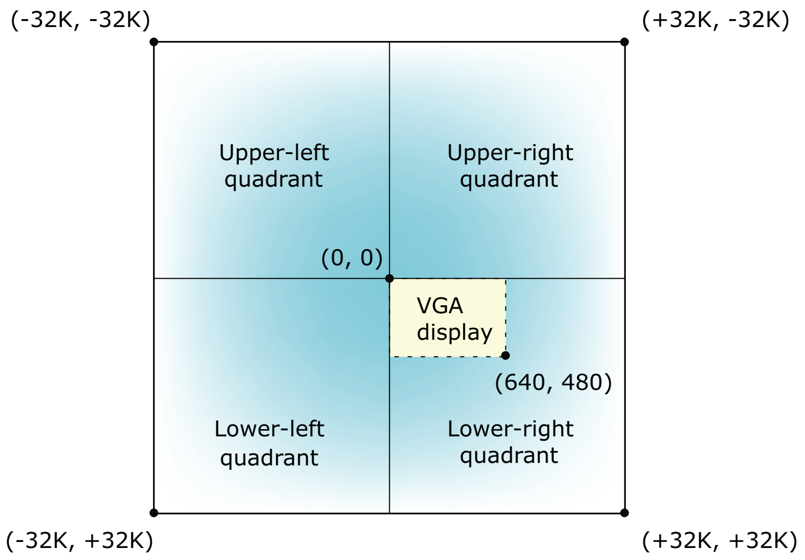
In this chapter. . .

In Photon, all applications consist of one or more rectangles called *regions*, which reside in an abstract, three-dimensional *event space*. Regions are assigned an identification number of type **`PhRid_t`**.

You can use **`phview`** to see what regions exist on your machine. For more information, see the *Utilities Reference*.

Photon coordinate space

The Photon coordinate space looks like this:



Photon coordinate space.

Unlike the typical Cartesian layout, the lower-right quadrant is the (+,+) quadrant.

The root region has the same dimensions as the entire coordinate space. As a rule, graphics drivers map the display screen to the location shown in the above diagram and place the Photon origin at the upper-left corner of the display screen. (Graphics drivers equate a single Photon coordinate to a single pixel value on your display screen).

Region coordinates

Region origins

When an application specifies coordinates within a given region, these are relative to the region's *origin*. The application specifies this origin when it opens the region.

Initial dimensions and location

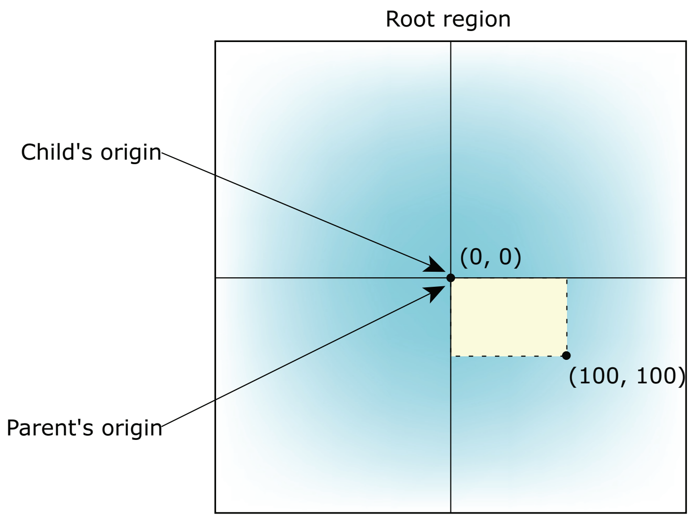
The initial dimensions of a region (i.e. *rect* argument in *PhRegionOpen()*) are relative to its origin. These dimensions control the range of the coordinates that the application can use within the region.

Let's look at some examples to get an idea of the relationship between a region's origin and its initial rectangle coordinates. These examples illustrate how opened regions are placed in relation to the root region, which has its origin in the center of the Photon coordinate space.

Origin at (0,0) and initial rectangle at (0,0)

By default, applications use the following approach for regions. (These kinds of regions are described in "Photon window manager" in the Photon Architecture appendix.)

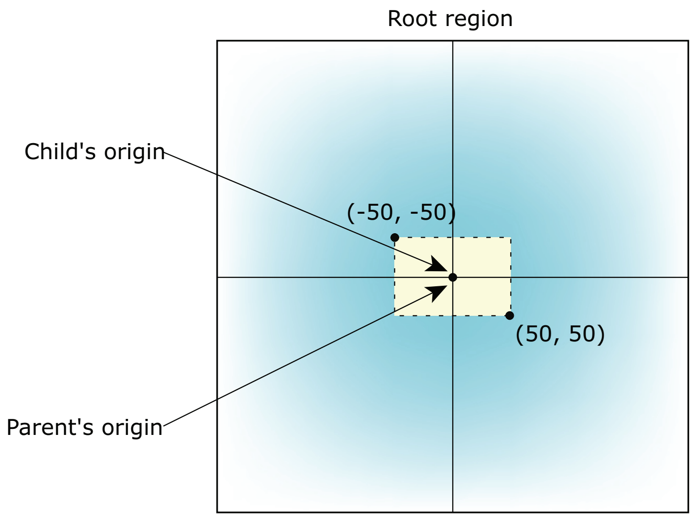
Coordinates:	Origin = (0,0)
	Upper left of initial rectangle = (0,0)
	Lower right of initial rectangle = (100,100)



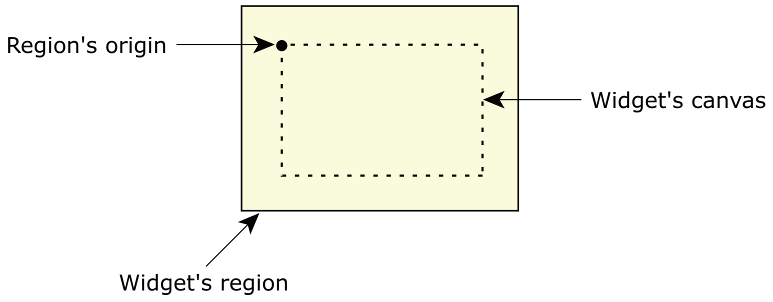
Origin at (0,0) and initial rectangle *not* at (0,0)

The following example shows an approach typically used for regions that fill the entire coordinate space. For example, for the device region and the workspace region, the upper left is (-32768,-32768) and the lower right is (32767,32767).

Coordinates:	Origin = (0,0)
	Upper left of initial rectangle = (-50,-50)
	Lower right of initial rectangle = (50,50)



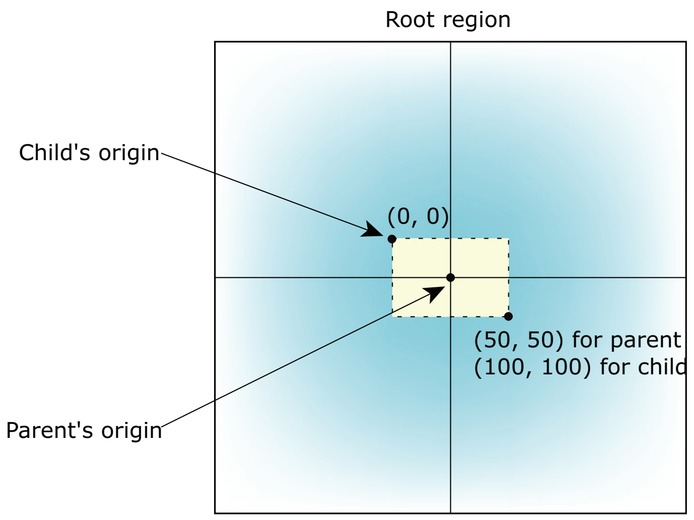
Many widgets create regions that have their upper-left corners at negative coordinates, so that the origin of the widgets' canvas is at (0, 0):



Origin *not* at (0,0) and initial rectangle *not* at (0,0)

The following example shows how a child's origin can differ from its parent's origin.

Coordinates:	Origin = (-50,-50)
	Upper left of initial rectangle = (0,0)
	Lower right of initial rectangle = (100,100)



About child regions

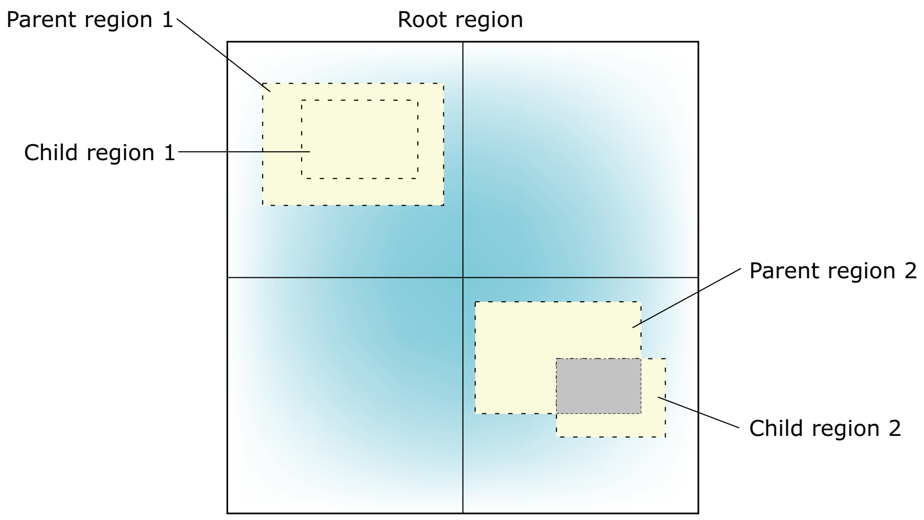
A child region's origin is specified relative to the parent's origin. So, when a region is moved, all its children automatically move with it. Likewise, when a region is destroyed, its children are destroyed.

If you want to make a region larger than any other of your application's regions, make it a child of the root region by calling *PhRegionOpen()* or *PhRegionChange()*, specifying Ph_ROOT_RID as the parent.

Regions and event clipping

A region can emit or collect events only where it overlaps with its parent. For example, in the following diagram:

- Child 1 can emit or collect events anywhere in its region
- Child 2 can emit or collect events only in the smaller gray area that overlaps with its parent region



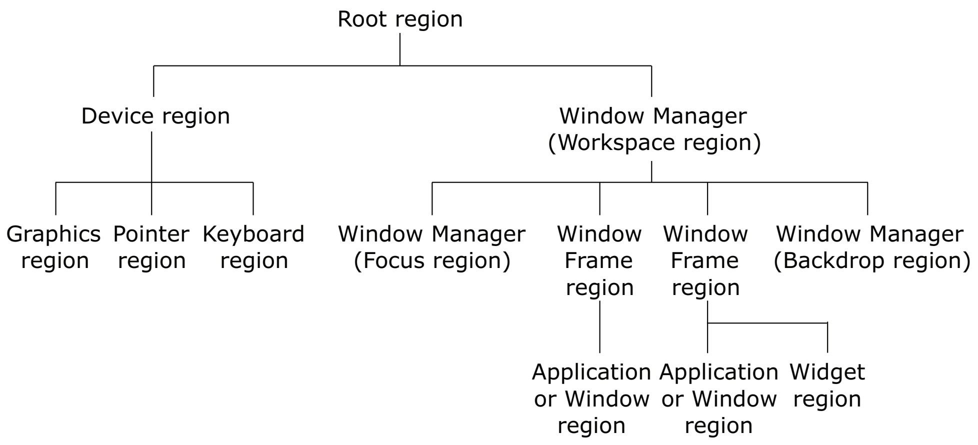
Regions and event clipping.

Because of this characteristic of regions, any portion of a region that doesn't overlap its parent is effectively invisible.

Placement and hierarchy

Region hierarchy

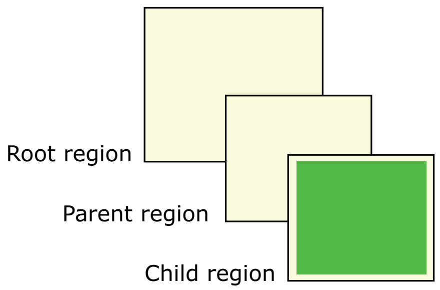
In Photon, every region has a parent region. This parent-child relationship results in a region hierarchy with the root region at the top. The following diagram shows the hierarchy of a typical Photon system:


Root region
Device region
Window Manager (Workspace region)
Graphics region
Pointer region
Keyboard region
Window Manager (Focus region)
Window Frame region
Window Frame region
Window Manager (Backdrop region)
Application or Window region
Application or Window region
Widget region

Hierarchy of regions for a typical Photon system.

Parent region

The Photon Manager always places child regions in front (i.e. on the user side) of their parents:

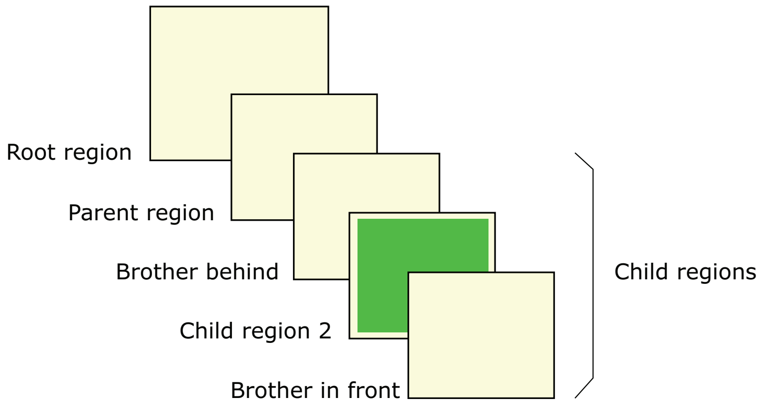


When opening a region, an application specifies the region's parent. If an application opens a region without specifying its parent, the region's parent is set to a default — basic regions become children of the root region and windows become children of the window manager's backdrop region.

Brother regions

Besides having a parent, a region may have "brothers," i.e. other regions who have the same parent. A region knows about only two of its brothers — the one immediately in front and the one immediately behind.

The following diagram shows a parent with three children, and the relationship that child region 2 has with its brothers:



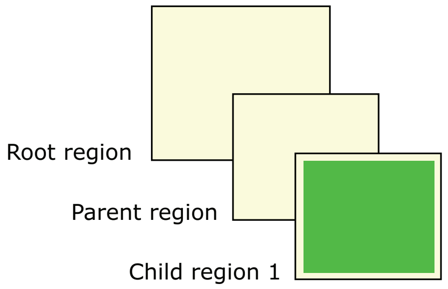
When the application opens a region (e.g. child region 2 in the above diagram), it can specify neither, one, or both immediate brothers. Depending on how the application specifies these brothers, the new region may be placed according to default rules (see below) or at a specific location.

If an application opens a region, specifying both brothers, and this action results in an ambiguous placement request, the request fails.

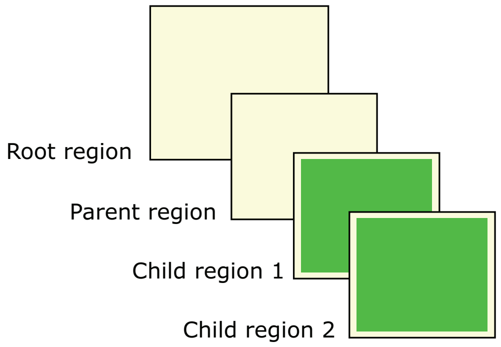
Default placement

If an application opens a region without specifying brothers, the Photon Manager places that region using default placement rules. In most cases, these rules cause a newly opened region to be placed in front of its frontmost brother, which then becomes "brother behind" of the new region. (To use different placement rules, you can specify the Ph_FORCE_FRONT flag.)

For example, in the following diagram, child region 1 is the frontmost region:



When the application opens child region 2 with default placement (next diagram), region 2 is placed in front of region 1. Region 1 becomes region 2's brother "behind." Region 2 becomes region 1's brother "in front."

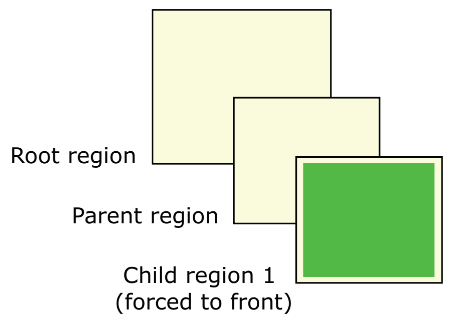


Ph_FORCE_FRONT flag

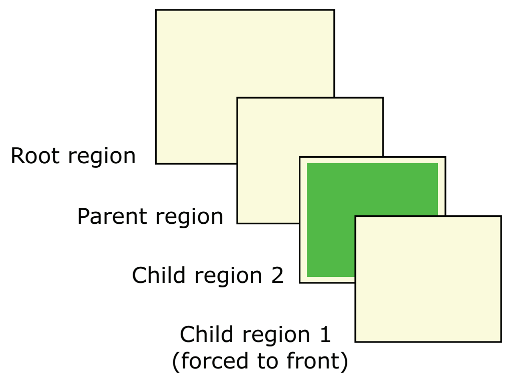
An application uses the Ph_FORCE_FRONT flag when it wants a region to remain in front of any subsequent brothers that rely on the Photon Manager's default placement.

As mentioned earlier, when a region is opened with default placement, it's placed ahead of its frontmost brother. But if any brother has the Ph_FORCE_FRONT flag set, then the new region is placed *behind* the farthest brother that has the Ph_FORCE_FRONT flag set.

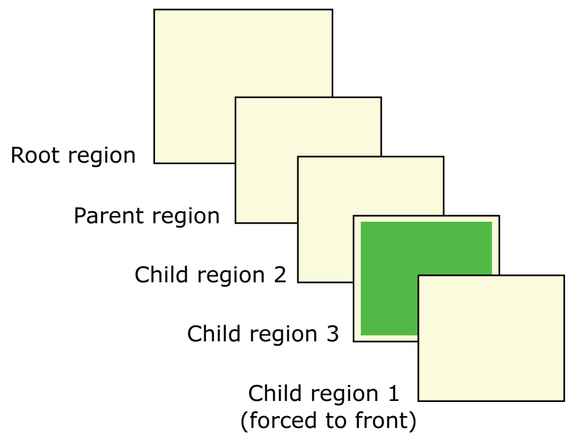
For example, let's see what would happen in the following example if child region 1 had the Ph_FORCE_FRONT flag set:



When child region 2 is opened with default placement (next diagram), it's placed behind region 1, and region 1 becomes its "brother in front." Because region 2 was placed using default rules, it doesn't inherit the Ph_FORCE_FRONT setting of region 1:



Then, if child region 3 is opened with default placement, it's placed as follows:



The application can set the Ph_FORCE_FRONT flag when it opens a region (or later) by changing the region's flags. The state of this flag doesn't affect how the region itself is placed, but rather how subsequent brothers are placed if those brothers are opened using default placement rules. That is, the Ph_FORCE_FRONT state of existing brothers doesn't affect the placement of a new region if it's opened with specified brother relations. See the next section, Specific placement.

Remember that the Ph_FORCE_FRONT flag affects placement only among brother regions — a child region always goes in front of its parent.

Specific placement

In contrast to default placement, if any brother is specified when a region is opened, then that specification controls the placement of the new region. We refer to this as *specific placement*.

If a "behind" brother is specified, then the newly opened region automatically is placed in front of that brother.

If an "in front" brother is specified, then the newly opened region is automatically placed behind that brother.

The Ph_FORCE_FRONT setting of the specified brother is inherited by the new region. If an application opens a region, specifying both brothers, and this results in an ambiguous placement request, then the open fails.

Using regions

Opening a region

To open a region, create a **`PtRegion`** widget. The **`PtRegion`** widget isn't included in PhAB's widget palette; to instantiate it:

- Call *PtCreateWidget()* in your application.

  Or:

- Create a window module, select it, and use the Change Class item in PhAB's Edit menu to turn the window into a **`PtRegion`**. For more information, see "Changing a widget's class" in the chapter on Creating Widgets in PhAB.

For more information on the **`PtRegion`** widget, see the *Widget Reference*.

Placing regions

While a region is always in front of its parent, the region's placement relative to its brothers is flexible. See "Placement and hierarchy" for more information about "default" and "specific" placement.

The **`PhRegion_t`** structure (see the *Library Reference*) contains the following members. These indicate the relationship of a region with its siblings:

- *bro_in_front* — indicates the sibling immediately in front
- *bro_behind* — indicates the sibling immediately behind

To retrieve this information, you can use *PhRegionQuery()*.

Changing region placement

An application can specify a region's placement when it opens the region, or it can change the placement later on. To change a region's placement, the application must change the relationship between the region and the region's family.

The application does this by doing any or all of the following:

- setting the *parent*, *bro_front*, and *bro_behind* members of the **`PhRegion_t`** structure
- setting the corresponding *fields* bits to indicate which members are valid (only those fields marked as valid will be acted on)
- calling the *PhRegionChange()* function

Since an application can be sure of the position of only the regions it owns, it shouldn't change the position of any other regions. Otherwise, by the time the application makes a request to change the position of a region it doesn't own, the information retrieved by *PhRegionQuery()* may not reflect that region's current position. That is, a request to change a region's placement may not have the results the application intended.

Changing the parent

You can change a region's parent in these ways:

- If the region has a parent widget, call *PtReparentWidget()* to make the region the child of another widget. Don't reparent the region directly.
- Specify the parent in the *parent* member of the child's **`PhRegion_t`** structure. The child region becomes the frontmost of the parent region's children.
- Specify a child of another parent as the region's brother. This makes the region a child of that parent, but lets you specify where the child region fits in the parent region's hierarchy.

The following constants are defined in **`<photon/PhT.h>`**:

- Ph_DEV_RID — the ID of the device region.
- Ph_ROOT_RID — the ID of the root region.

Specifying brothers

If you set:	**Then:**
bro_behind	The region indicated in the *rid* member of **`PhRegion_t`** moves in front of the *bro_behind* region

continued...

If you set:	**Then:**
bro_in_front	The region indicated in the *rid* member of **`PhRegion_t`** moves behind the *bro_in_front* region

As discussed in "Changing the parent," a region inherits the parent of any specified brothers that are children of another parent.

System information

You can get the following information about your system:

- the version of your Photon server
- an estimate of the bandwidth of communication between your window and the Photon server
- information about regions that intersect your window:
  - graphics regions
  - keyboard regions
  - pointer regions
  - input group regions

You don't get information about each region. Instead, you get the minimum value of each type of information.

For example, if several graphics-driver regions overlapping your window have different bandwidths, the bandwidth given is the minimum of them.

There are two functions that you can use to get system information:

PhQuerySystemInfo()
: Get the information for a given region.

PtQuerySystemInfo()
: Get the information for a widget (usually a window).

PhQuerySystemInfo() sends a message to the server each time that you call it.

PtQuerySystemInfo() calls *PhQuerySystemInfo()*, but buffers the information. When a region that intersects your widget changes (for example, it's moved), the buffer is marked as invalid. The next time you call *PtQuerySystemInfo()*, it calls *PhQuerySystemInfo()* again. By using the buffer whenever possible, *PtQuerySystemInfo()* keeps the number of messages to a minimum.

Both *PtQuerySystemInfo()* and *PhQuerySystemInfo()* fill in a structure of type **`PhSysInfo_t`** that your application has allocated. For more information, see the Photon *Library Reference*.

One field that's of particular interest is the graphics bandwidth, in *gfx.bandwidth*. This value can be used to modify the behavior of an interface based on the connection speed. For example, a simple state change could be substituted for an elaborate animation if the bandwidth is Ph_BAUD_SLOW or less. It's also a good idea to see if shared memory can be used for drawing; the Ph_GCAP_SHMEM flag is set in *gfx.capabilities* if all the graphics drivers support the *...mx()* functions and they're all running on your node.

Chapter 25

Events

In this chapter. . .

The interactions between applications, users and the Photon server are represented by data structures called *events*.

Event information is stored in structures of type **PhEvent_t**; see the Photon *Library Reference*.

Pointer events

Most of the time, you can use a widget's callbacks to handle what the user does while pointing to it. If you're working with event handlers, you'll need to know what events Photon emits.

Pressing a button

When you press the pointer button, Photon emits a Ph_EV_BUT_PRESS event to the widget that currently has focus.

Releasing a button

When you release the button, Photon emits *two* Ph_EV_BUT_RELEASE events:

- One with subtype Ph_EV_RELEASE_REAL
- One with subtype Ph_EV_RELEASE_PHANTOM.

The real release hits whatever the mouse points to when you release the button. The phantom release *always* goes to the same region (and position) that received the press.

In other words, if your widget saw the press, it also sees the phantom release. And depending on where the mouse was pointing to, you may or may not get the real release. If your widget gets both the real and phantom releases, the real one always comes first.

Multiple clicks

Whenever you press or release the mouse button, the event includes the click count. How can your application determine that you clicked, instead of double clicked?

There's a click counter in the event data that's associated with Ph_EV_BUT_PRESS and Ph_EV_BUT_RELEASE events; to get this data, call *PhGetData()*. The data for these events is a structure of type **PhPointerEvent_t** (see the Photon *Library Reference* for details); its *click_count* member gives the number of times that you clicked the mouse button.

If you keep clicking quickly enough without moving the mouse, the counter keeps incrementing. If you move the mouse or stop clicking for a while, the counter resets and Photon emits a Ph_EV_BUT_RELEASE event with a subtype of Ph_EV_RELEASE_ENDCLICK.

In other words, the first click generates a Ph_EV_BUT_PRESS event and a pair of Ph_EV_BUT_RELEASE events (one REAL and one PHANTOM) with *click_count* set to 1. Then, depending on whether the user clicks again soon enough or not, you get either:

- A Ph_EV_BUT_PRESS event and a pair of Ph_EV_BUT_RELEASE events with *click_count* set to 2

  Or:

- A Ph_EV_BUT_RELEASE event with a subtype of Ph_EV_RELEASE_ENDCLICK.

After the second click, you either get a third one or an ENDCLICK, and so on. But eventually you get an ENDCLICK — and the next time the person clicks, the click count is 1 again.

Modifier keys

If you need to determine what keys were pressed in a pointer event, call *PhGetData()* to get the event data that's included for Ph_EV_BUT_PRESS and Ph_EV_BUT_RELEASE events. The data for these events is a structure of type **PhPointerEvent_t** (described in the Photon *Library Reference*); check its *key_mods* member to determine the modifier keys that were pressed.

For example, this *Pt_CB_ACTIVATE* callback lists the modifier keys that were pressed when the pointer button was released:

```
/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "abimport.h"
#include "proto.h"


int
check_keys( PtWidget_t *widget, ApInfo_t *apinfo,
            PtCallbackInfo_t *cbinfo )

{

   PhPointerEvent_t *event_data;

   /* eliminate 'unreferenced' warnings */
   widget = widget, apinfo = apinfo, cbinfo = cbinfo;

   if (cbinfo->event->type != Ph_EV_BUT_RELEASE) {
     printf ("Not a Ph_EV_BUT_RELEASE event\n");
   } else {
     printf ("It's a Ph_EV_BUT_RELEASE event\n");

     event_data = (PhPointerEvent_t *)
                   PhGetData (cbinfo->event);

     if (event_data->key_mods & Pk_KM_Shift )
        printf ("   Shift\n");
     if (event_data->key_mods & Pk_KM_Ctrl )
        printf ("   Ctrl\n");

     if (event_data->key_mods & Pk_KM_Alt )
        printf ("   Alt\n");

   }
   return( Pt_CONTINUE );
}
```

Emitting events

The most general way for your application to emit an event is to call *PhEmit()*:

```
int PhEmit( PhEvent_t *event,
            PhRect_t *rects,
            void *data );
```

The arguments are:

event A pointer to a **PhEvent_t** structure. The application emitting the event needs to set the following members:

- *type* — the type of event.
- *subtype* — the event subtype (if necessary).
- *flags* — event modifiers (e.g. direction).
- *emitter* — a **PhEventRegion_t** structure; you need to set at least the ID of the region that's emitting the event.
- *translation* — typically set to (0,0) when emitting an event.
- *num_rects* — the number of rectangles in the function's *rects* argument. If you set *num_rects* to 0, you must also pass *rects* as NULL.
- *event*->*collector.rid* — if you set the collector ID to zero, the event is enqueued to every appropriately sensitive region that intersects with the event.

  If you set *collector.rid* to a region ID, only that region notices the event.

The Photon Manager sets the following members of the event structure after it has enqueued a copy of the event to an application:

- *timestamp* — the time when this event was emitted (in milliseconds).
- *collector* — a **PhEventRegion_t** structure that includes the ID of the collecting region.

rects An array of **PhRect_t** structures (see the Photon *Library Reference*) indicating the event's initial rectangle set. If this argument is NULL, the set consists of a single rectangle corresponding to the emitting region.

data Valid data for the type of event being emitted. Each type of event has its own type of data, as described for the **PhEvent_t** structure in the Photon *Library Reference*.

If the event-specific data isn't in contiguous memory, you may find *PhEmitmx()* more useful than *PhEmit()*:

```
int PhEmitmx( PhEvent_t *event,
              PhRect_t *rects,
              int mxparts,
              struct _mxfer_entry *mx );
```

The return codes for *PhEmit()* and *PhEmitmx()* are:

A nonnegative value
Successful completion.

-1 An error occurred; check the value of *errno*.

Targeting specific regions

Sometimes an application needs to target an event directly at a specific region, without making the event travel through the event space before arriving at that region. You can use an inclusive event or a direct event to ensure that the targeted region sees the event.

Inclusive event

For an inclusive event, do the following:

- Set the emitter's region ID (i.e. *event*->*emitter*.*rid*) to the ID of the target region — this causes the event to be emitted automatically *from* that region.
- Set Ph_EVENT_INCLUSIVE on in the *flag* member of the event — this causes the Photon Manager to emit the event to the emitting region before emitting it into the event space.

If you don't want an inclusive targeted event to continue through the event space, you must make the emitting region *opaque* to that type of event, or use a direct event instead.

Direct event

For a direct event, do the following:

- Set the emitter's region ID (i.e. *event*->*emitter*.*rid*) to the ID of your application's region.
- Set the collector's region ID (i.e. *event*->*collector*.*rid*) to the ID of the target region.
- Set Ph_EVENT_DIRECT on in the *flag* member of the event — this causes the Photon Manager to emit the event directly from *emitter* to *collector*.

Targeting specific widgets

If you want to send an event to a specific widget, you could call *PhEmit()* as described above, but you'll need to look after a lot of details, including making sure:

- The event is delivered to the right window.
- The widget still has focus — there might be other events enqueued before yours.

It's easier to call *PtSendEventToWidget()*. This function gives the event to the specified widget directly and without delay. It's much more deterministic and efficient than *PhEmit()*.

The prototype is:

```
int PtSendEventToWidget( PtWidget_t *widget,
                         PhEvent_t *event );
```

Emitting key events

Sometimes you might need to simulate a key press in your application. Depending on what exactly you want to achieve, you can choose from several ways of generating key events:

- Emit a Ph_EV_KEY event from the device region:

  ```
  event->emitter.rid = Ph_DEV_RID;
  ```

  The rectangle set should consist of a single pixel — if you're not using the window manager, or if PWM is set to use cursor focus, the position of that pixel determines which window the event will hit.

- If you know which region you want to send the event to, emit a Ph_EV_KEY event directly to that region:

  ```
  event->collector.rid = rid;
  event->flags |= Ph_EVENT_DIRECT;
  ```

In both of these cases, use a **PhKeyEvent_t** structure as the event data. For more information, see the Photon *Library Reference*.

Here's an example:

```
static void send_key( long key )
{
    struct{
    PhEvent_t event;
    PhRect_t  rect;
    PhKeyEvent_t pevent;
    } new_event;

    PhEvent_t        event;
    PhKeyEvent_t  key_event;

    PhRect_t rect;
```

```
    rect.ul.x = rect.ul.y = 0;
    rect.lr.x = rect.lr.y = 0;

    memset( &event    , 0, sizeof(event)     );
    memset( &key_event, 0, sizeof(key_event) );

    event.type          = Ph_EV_KEY;
    event.emitter.rid   = Ph_DEV_RID;
    event.num_rects = 1;
    event.data_len      = sizeof(key_event);
    event.input_group   = 1;

    key_event.key_cap   = key;
    key_event.key_sym   = key;

    if ( isascii( key ) && isupper( key ) )
    {
        key_event.key_mods = Pk_KM_Shift;
    }

    /* Emit the key press. */

    key_event.key_flags = Pk_KF_Sym_Valid | Pk_KF_Cap_Valid |
                          Pk_KF_Key_Down;
    PhEmit( &event, &rect, &key_event );

    /* Emit the key release. */

    key_event.key_flags &= ~(Pk_KF_Key_Down | Pk_KF_Sym_Valid) ;
    PhEmit( &event ,&rect, &key_event );

    return;
}
```

Event coordinates

When an event is emitted, the coordinates of its rectangle set are relative to the emitting region's origin. But when the event is collected, its coordinates become relative to the collecting region's origin.

The Photon Manager ensures this happens by translating coordinates accordingly. The *translation* member of the **`PhEvent_t`** specifies the translation between the emitting region's origin and the collecting region's origin.

Event handlers — raw and filter callbacks

The **`PtWidget`** widget class provides these callbacks for processing events:

Pt_CB_FILTER	Invoked *before* the event is processed by the widget. They let you perform actions based on the event before the widget sees it. They also give you the opportunity to decide if the event should be ignored, discarded, or allowed to be processed by the widget.
Pt_CB_RAW	These callbacks are invoked *after* the widget has processed the event, even if the widget's class methods consume it.

These callbacks are called every time a Photon event that matches an event mask (provided by the application) is received. Since all the widget classes in the Photon widget library are descended from the **`PtWidget`**, these callbacks can be used with *any* widget from the Photon widget library.

When you attach a raw or filter callback to a widget, the widget library creates a region, if necessary, that will pick up specific events for the widget. This increases the number of regions the Photon Manager must manage and, as a result, may reduce performance.

For this reason, use event handlers only when you have to do something that can't be done using standard widget callbacks. If you do use event handlers, consider using them only on window widgets, which already have regions.

Whenever a Photon event is received, it's passed down the widget family hierarchy until a widget consumes it. (When a widget has processed an event and prevents another widget from interacting with the event, the first widget is said to have *consumed* the event.)

In general, the *Pt_CB_FILTER* callbacks are invoked on the way down the hierarchy, and the *Pt_CB_RAW* callbacks are invoked on the way back up. Each widget processes the event like this:

1 The widget's *Pt_CB_FILTER* callbacks are invoked if the event type matches the callback's mask. The callback's return code indicates what's to be done with the event:

Pt_CONSUME	The event is consumed, without being processed by the widget's class methods.
Pt_PROCESS	The widget's class methods are allowed to process the event.
Pt_IGNORE	The event bypasses the widget and all its descendants, as if they didn't exist.

2 If the widget is sensitive to the event, and the *Pt_CB_FILTER* callback permits, the widget's class method processes the event. The class method might consume the event.

3 If the widget consumes the event, the widget's *Pt_CB_RAW* callbacks are invoked if the event type matches the callback's mask. The raw callbacks of the widget's parents aren't called.

4 If the widget doesn't consume the event, the event is passed to the widget's children, if any.

5 If no widget consumes the event, the event is passed back up the family hierarchy, and each widget's *Pt_CB_RAW* callbacks are invoked if the event type matches the callback's mask.

The value returned by a widget's *Pt_CB_RAW* callback indicates what's to be done with the event:

Pt_CONSUME	The event is consumed and no other raw callbacks are invoked as the event is passed up to the widget's parent.
Pt_CONTINUE	The event is passed up to the widget's parent.

If a widget is disabled (i.e. Pt_BLOCKED is set in its *Pt_ARG_FLAGS*), the raw and filter callbacks aren't invoked. Instead, the widget's *Pt_CB_BLOCKED* callbacks (if any) are invoked.

Let's look at a simple widget family to see how this works. Let's suppose you have a window that contains a pane that contains a button. Here's what normally happens when you click on the button:

1. The window's *Pt_CB_FILTER* callbacks are invoked, but don't consume the event. The window's class methods don't consume the event either.
2. The event is passed to the pane. Neither its *Pt_CB_FILTER* callbacks nor its class methods consume the event.
3. The event is passed to the button. Its *Pt_CB_FILTER* callbacks don't consume the event, but the class methods do; the appropriate callback (e.g. *Pt_CB_ACTIVATE*) is invoked.
4. The button's *Pt_CB_RAW* callbacks are invoked for the event.
5. The pane's and window's *Pt_CB_RAW* callbacks aren't invoked because the button consumed the event.

If the pane's *Pt_CB_FILTER* callback says to ignore the event:

1. The window processes the event as before.
2. The pane's *Pt_CB_FILTER* callbacks say to ignore the event, so the pane and all its descendants are skipped.
3. There are no more widgets in the family, so the window's *Pt_CB_RAW* callbacks are invoked.

For information on adding event handlers, see:

- "Event handlers — raw and filter callbacks" in the Editing Resources and Callbacks in PhAB chapter
- "Event handlers" in the Managing Widgets in Application Code chapter.

Collecting events

Most applications collect events by calling *PtMainLoop()*. This routine processes Photon events and supports work procedures and input handling.

If your application doesn't use widgets, you can collect events:

- *asynchronously* by calling *PhEventRead()*. You must call *PhEventArm()* before you call *PhEventRead()* for the first time.
- *synchronously* by calling *PhEventNext()*. You can check for events without blocking by calling *PhEventPeek()*.

However, writing your own mainloop function isn't a trivial matter; it's easier to create at least a disjoint widget (such as **`PtRegion`** or **`PtWindow`**) and then use *PtMainLoop()*.

PhGetRects() extracts the rectangle set, and *PhGetData()* extracts the event's data portion.

A region can collect a given event only if portions of the region intersect the event, and the region is sensitive to that type of event.

Event compression

The Photon Manager compresses drag, boundary, and pointer events. That is, if an event of that type is pending when another event arrives, the new event will be merged with the unprocessed events. As a result, an application sees only the latest values for these events and is saved from collecting too many unnecessary events.

Dragging

If you need to capture mouse coordinates, for example to drag graphical objects in your application, you'll need to work with events.

If you want to transfer arbitrary data within your application or between applications, see the Drag and Drop chapter.

There are two types of dragging:

outline dragging	The user sees an outline while dragging. When the dragging is complete, the application repositions the widget.
opaque dragging	The application moves the widget as the dragging progresses.

Dragging is done in two steps:

1 Initiating the dragging, usually when the user clicks on something.

2 Handling drag (Ph_EV_DRAG) events.

These steps are discussed in the sections that follow.

Initiating dragging

Where you initiate the dragging depends on how the user is meant to drag widgets. For example, if the user holds down the left mouse button on a widget to drag it, initiate dragging in the widget's Arm (*Pt_CB_ARM*) or Outbound (*Pt_CB_OUTBOUND*) callback. Make sure that Pt_SELECTABLE is set in the widget's *Pt_ARG_FLAGS* resource.

Dragging is started by calling the *PhInitDrag()* function:

```
int PhInitDrag( PhRid_t rid,
                unsigned flags,
                PhRect_t *rect,
                PhRect_t *boundary,
                unsigned int input_group,
                PhDim_t *min,
                PhDim_t *max,
                const PhDim_t *step,
                const PhPoint_t *ptrpos,
                const PhCursorDescription_t *cursor );
```

The arguments are used as follows:

rid	The ID of the region that *rect* and *boundary* are relative to. You can get this by calling *PtWidgetRid()*.
flags	Indicate whether outline or opaque dragging is to be used, and which edge(s) of the dragging rectangle track the pointer, as described below.
rect	A **PhRect_t** structure (see the Photon *Library Reference*) that defines the area to drag.
boundary	Rectangular area that limits the dragging
input_group	Get this from the event in the callback's *cbinfo* parameter by calling *PhInputGroup()*.
min, *max*	Pointers to **PhDim_t** structures (see the Photon *Library Reference*) that define the minimum and maximum sizes of the drag rectangle.
step	Dragging granularity.
ptrpos	If not NULL, it's a pointer to a **PhPoint_t** structure (see the Photon *Library Reference*) that defines the initial cursor position for the drag. Applications should take it from the event that makes them decide to start a drag. If the cursor moves from that position by the time your *PhInitDrag()* reaches Photon, your drag is updated accordingly. In other words, Photon makes the drag behave as if it started from where you thought the cursor was rather than from where it actually was a few moments later.

cursor	If not NULL, defines how the cursor should look while dragging.

If Ph_DRAG_TRACK is included in *flags*, then opaque dragging is used; if Ph_DRAG_TRACK isn't included, outline dragging is used.

The following flags indicate which edge(s) of the dragging rectangle track the pointer:

- Ph_TRACK_LEFT
- Ph_TRACK_RIGHT
- Ph_TRACK_TOP
- Ph_TRACK_BOTTOM
- Ph_TRACK_DRAG—all the above

Outline dragging

The following example shows an Arm (*Pt_CB_ARM*) callback that initiates outline dragging:

```
/* Start dragging a widget                               */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "globals.h"
#include "abimport.h"
#include "proto.h"

int
start_dragging( PtWidget_t *widget,
                ApInfo_t *apinfo,
                PtCallbackInfo_t *cbinfo )

{
    PhDim_t *dimension;
    PhRect_t rect;
    PhRect_t boundary;

    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo, cbinfo = cbinfo;

    /* Set the dragging rectangle to the position and size of
       the widget being dragged. */

    PtWidgetExtent (widget, &rect);

    /* Set the boundary for dragging to the boundary of the
```

```
       window. */

    PtGetResource (ABW_base, Pt_ARG_DIM, &dimension, 0);
    boundary.ul.x = 0;
    boundary.ul.y = 0;
    boundary.lr.x = dimension->w - 1;
    boundary.lr.y = dimension->h - 1;

    /* Initiate outline dragging (Ph_DRAG_TRACK isn't
       specified). */

    PhInitDrag (PtWidgetRid (ABW_base),
                Ph_TRACK_DRAG,
                &rect, &boundary,
                PhInputGroup( cbinfo->event ),
                NULL, NULL, NULL, NULL, NULL );

    /* Save a pointer to the widget being dragged. */

    dragged_widget = widget;

    return( Pt_CONTINUE );
}
```

The above callback is added to the Arm (*Pt_CB_ARM*) callback of the widget to be dragged. It can be used for dragging any widget, so a pointer to the widget is saved in the global variable *dragged_widget*.

Opaque dragging

If you want to use opaque dragging, add the Ph_DRAG_TRACK flag to the call to *PhInitDrag()*:

```
PhInitDrag( PtWidgetRid (ABW_base),
            Ph_TRACK_DRAG | Ph_DRAG_TRACK,
            &rect, &boundary,
            PhInputGroup( cbinfo->event ),
            NULL, NULL, NULL, NULL, NULL );
```

Handling drag events

To handle drag (Ph_EV_DRAG) events, you need to define a Raw (*Pt_CB_RAW*) or Filter (*Pt_CB_FILTER*) callback.

The raw or filter callback must be defined for the widget whose region was passed to *PhInitDrag()*, not for the widget being dragged. For the example given, the Raw callback is defined for the base window.

As described in "Event handlers — raw and filter callbacks" in the Editing Resources and Callbacks in PhAB chapter, you use an event mask to indicate which events your callback is to be called for. For dragging, the event is Ph_EV_DRAG. The most commonly used subtypes for this event are:

Ph_EV_DRAG_START

The user has started to drag.

Ph_EV_DRAG_MOVE

The dragging is in progress (opaque dragging only).

Ph_EV_DRAG_COMPLETE

The user has released the mouse button.

Outline dragging

If you're doing outline dragging, the event subtype you're interested in is Ph_EV_DRAG_COMPLETE. When this event occurs, your callback should:

1. Get the data associated with the event. This is a **`PhDragEvent_t`** structure that includes the location of the dragging rectangle, in absolute coordinates. For more information, see the Photon *Library Reference*.
2. Calculate the new position of the widget, relative to the dragging region. This is the position of the upper left corner of the dragging rectangle, translated by the amount given in the event's *translation* field.
3. Set the widget's *Pt_ARG_POS* resource to the new position.

Remember, the callback's *widget* parameter is a pointer to the container (the base window in the example), not to the widget being dragged. Make sure you pass the correct widget to *PtSetResources()* or *PtSetResource()* when setting the *Pt_ARG_POS* resource.

For example, here's the Raw callback for the outline dragging initiated above:

```
/* Raw callback to handle drag events; define this
   for the base window. */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Toolkit headers */
#include <Ph.h>
#include <Pt.h>
#include <Ap.h>

/* Local headers */
#include "globals.h"
#include "abimport.h"
#include "proto.h"

int
end_dragging( PtWidget_t *widget,
              ApInfo_t *apinfo,
              PtCallbackInfo_t *cbinfo )

{
    PhDragEvent_t *dragData;
    PhPoint_t new_pos;
    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo, cbinfo = cbinfo;

    /* Ignore all events until dragging is done. */

    if (cbinfo->event->subtype != Ph_EV_DRAG_COMPLETE)
    {
      return (Pt_CONTINUE);
    }

    /* Get the data associated with the event. */

    dragData = PhGetData (cbinfo->event);

    /* The rectangle in this data is the absolute
       coordinates of the dragging rectangle. We want to
       calculate the new position of the widget, relative to
       the dragging region. */

    new_pos.x = dragData->rect.ul.x
                + cbinfo->event->translation.x;
    new_pos.y = dragData->rect.ul.y
                + cbinfo->event->translation.y;
    printf ("New position: (%d, %d)\n", new_pos.x, new_pos.y);

    /* Move the widget. */

    PtSetResource (dragged_widget, Pt_ARG_POS, &new_pos, 0);

    return( Pt_CONTINUE );

}
```

Opaque dragging

The callback for opaque dragging is similar to that for outline dragging—the only difference is the subtype of event handled:

```
if (cbinfo->event->subtype != Ph_EV_DRAG_MOVE)
{
  return (Pt_CONTINUE);
}
```

Chapter 26

Window Management

In this chapter. . .

Sometimes you'll need to interact with the Photon Window Manager to make your windows and dialogs behave the way you'd like.

Remember that PhAB's window and dialog modules are implemented as **`PtWindow`** widgets. **`PtWindow`** has many resources that are used to interact with the Window Manager.

For information about the Window Manager's regions, see the appendix on Photon architecture. For a list of related functions, see "Window Manager" in the Summary of Functions chapter of the Photon *Library Reference*.

Window-management flags

The **`PtWindow`** widget defines various types of flags:

Pt_ARG_WINDOW_RENDER_FLAGS
: Which window decorations appear in the window frame.

Pt_ARG_WINDOW_MANAGED_FLAGS
: How the Window Manager operates on the window.

Pt_ARG_WINDOW_NOTIFY_FLAGS
: Which Window Manager events your application would like to be notified of.

Pt_ARG_WINDOW_STATE
: The current state of the window.

If you change the state of the window after it's realized, you'll need to let the Window Manager know. See "Getting and setting the window state" later in this chapter.

Window-rendering flags

The *Pt_ARG_WINDOW_RENDER_FLAGS* resource specifies what appears in the window's frame.

In PhAB, if you turn these flags off, PhAB provides a border, title, and buttons that allow you to re-size, move, minimize, and close the window in design mode. The flags still affect how the window appears when the application is running.

To display:	Set this bit:	Default:
Border	Ph_WM_RENDER_BORDER	Yes
Resize handles	Ph_WM_RENDER_RESIZE	Yes
Title bar	Ph_WM_RENDER_TITLE	Yes
Menu button	Ph_WM_RENDER_MENU	Yes
Close button	Ph_WM_RENDER_CLOSE	
Help button (question mark)	Ph_WM_RENDER_HELP	
Minimize button	Ph_WM_RENDER_MIN	Yes
Maximize button	Ph_WM_RENDER_MAX	Yes
Collapse button	Ph_WM_RENDER_COLLAPSE	Yes
An extra line inside the standard borders	Ph_WM_RENDER_INLINE	

Using these flags to display a decoration doesn't cause the Window Manager to do anything with it. You may need to set the window managed flags and/or notify flags.

Window-managed flags

The *Pt_ARG_WINDOW_MANAGED_FLAGS* resource specifies what operations you want the window manager to handle:

To let the window manager:	Set this bit:	Default:
Close the window	Ph_WM_CLOSE	Yes
Give focus	Ph_WM_FOCUS	Yes
Build and control the window menu	Ph_WM_MENU	Yes
Move the window to the front	Ph_WM_TOFRONT	Yes
Move the window to the back	Ph_WM_TOBACK	Yes
Move the window to a new console as the user switches consoles	Ph_WM_CONSWITCH	
Resize the window	Ph_WM_RESIZE	Yes
Move the window	Ph_WM_MOVE	Yes
Hide (i.e. minimize) the window	Ph_WM_HIDE	Yes
Maximize the window	Ph_WM_MAX	Yes

continued. . .

To let the window manager:	Set this bit:	Default:
Display the window as a backdrop	Ph_WM_BACKDROP	
Restore the window	Ph_WM_RESTORE	Yes
Provide context-sensitive help	Ph_WM_HELP	
Make the window force-front	Ph_WM_FFRONT	
Collapse the window to just the title bar	Ph_WM_COLLAPSE	
Prevent you from cycling focus to the window by pressing Alt-Esc, Alt-Shift-Esc, or Alt-Tab	Ph_WM_NO_FOCUS_LIST	

By default, a selection of these flags are set, as defined by Ph_WM_APP_DEF_MANAGED in **`<PhWm.h>`**. You'd turn the management flags off if:

- You don't want the operation to happen.
- You want the application to handle the operation. In this case, you'll need to set the appropriate notify flag as well.

Window-notify flags

The *Pt_ARG_WINDOW_NOTIFY_FLAGS* resource specifies which window-manager operations your application should be notified of. This resource uses the same bits as *Pt_ARG_WINDOW_MANAGED_FLAGS*:

To be notified when:	Set this bit:	Default:
The window is to be closed (see below)	Ph_WM_CLOSE	Yes
The window is to gain/lose focus	Ph_WM_FOCUS	
The window menu is requested or dismissed	Ph_WM_MENU	
The window is to be moved to the front	Ph_WM_TOFRONT	
The window is to be moved to the back	Ph_WM_TOBACK	
The window is to switch consoles	Ph_WM_CONSWITCH	
The window is to be resized	Ph_WM_RESIZE	Yes
The window is to be moved	Ph_WM_MOVE	
The window is to be hidden or unhidden	Ph_WM_HIDE	
The window is to be maximized	Ph_WM_MAX	
The window is to be made into a backdrop	Ph_WM_BACKDROP	

continued...

To be notified when:	Set this bit:	Default:
The window is to be restored	Ph_WM_RESTORE	
The help button is pressed	Ph_WM_HELP	Yes
The window is to be made force-front or not force-front	Ph_WM_FFRONT	

The default setting is Ph_WM_RESIZE | Ph_WM_CLOSE | Ph_WM_HELP.

When the requested operations occur, the window's *Pt_CB_WINDOW* callback is invoked. See "Notification callback" below.

If you set the Ph_WM_CLOSE notify flag, your application's *Pt_CB_WINDOW* callback is invoked when someone wants the window to close. Your application doesn't have to close the window — it could decide to leave it open.

In contrast, the *Pt_CB_WINDOW_CLOSING* callback is called when a window is being unrealized, but before its region is removed. At this point, the application can't stop the window from being closed.

If you've set the Ph_WM_CLOSE managed flag, the window manager is told to handle the window's closing. In this case, the *Pt_CB_WINDOW_CLOSING* callback is invoked, but the *Pt_CB_WINDOW* callback isn't.

Notification callback

When a window manager operation occurs that's listed in the window's notify flags (*Pt_ARG_WINDOW_NOTIFY_FLAGS*), the window's *Pt_CB_WINDOW* callback is invoked.

Each callback function listed in this resource is passed a **`PtCallbackInfo_t`** structure (see the Photon *Widget Reference*) that contains at least the following members:

reason	Pt_CB_WINDOW
reason_subtype	0 (not used).
event	A pointer to the event that caused the callback to be invoked.
cbdata	A pointer to a **`PhWindowEvent_t`** (described in the Photon *Library Reference*).

These callback functions should return Pt_CONTINUE.

Example: verifying window closure

Suppose you want to verify that the user really wants to exit the application when the window is closed. Here's what you need to do:

- Unset Ph_WM_CLOSE in the *Pt_ARG_WINDOW_MANAGED_FLAGS*. This tells the window manager not to close the window.
- Set Ph_WM_CLOSE in the *Pt_ARG_WINDOW_NOTIFY_FLAGS*. The window manager will notify you when the user tries to close the window.
- Add a *Pt_CB_WINDOW* like the following:

```
int
window_callback( PtWidget_t *widget, ApInfo_t *apinfo,
                 PtCallbackInfo_t *cbinfo )

    {
    PhWindowEvent_t *we = cbinfo->cbdata;
    char *btns[] = { "&Yes", "&No" };
    char Helvetica14[MAX_FONT_TAG];

    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo, cbinfo = cbinfo;

    if ( we->event_f == Ph_WM_CLOSE ) {

      /* Ask the user if we should really exit. Use
         14-point Helvetica for the font if it's
         available. */

      switch( PtAlert( ABW_base, NULL, NULL, NULL,
                       "Do you really want to exit?",
                       PfGenerateFontName("Helvetica", 0, 14,
                                          Helvetica14),
                       2, btns, NULL, 1, 2, Pt_MODAL ) ) {

          case 1:  /* yes */
          PtExit (EXIT_SUCCESS);
          break;

          case 2:  /* no */
          return (Pt_CONTINUE);
      }
    } else {
      /* Check for other events. */
    }

    return( Pt_CONTINUE );

    }
```

There's a significant difference between the Ph_WM_CLOSE event and the Window Closing (*Pt_CB_WINDOW_CLOSING*) callback.

A *Pt_CB_WINDOW* callback with a Ph_WM_CLOSE event is just a notification from PWM that the user has clicked on the Close button or chosen Close from the PWM menu. If the Ph_WM_CLOSE bit is unset in the *Pt_ARG_WINDOW_MANAGED_FLAGS*, the library takes no further action.

Window Closing is invoked when the window is about to unrealize for any reason. This includes transporting to another Photon and explicit calls to *PtDestroyWidget()* or *PtUnrealizeWidget()*. If you want to make sure in a Window Closing callback that the window is actually being destroyed, check the Pt_DESTROYED flag in *Pt_ARG_FLAGS*. You can also use the *Pt_CB_DESTROYED* callback to know when a window is marked for destruction, or *Pt_CB_IS_DESTROYED* to know when it is being destroyed.

Also note that calling *exit()* explicitly bypasses all those callbacks.

Getting and setting the window state

The *Pt_ARG_WINDOW_STATE* resource controls the window's state:

To do this:	Set this bit:
Maximize the window	Ph_WM_STATE_ISMAX
Make the window a backdrop	Ph_WM_STATE_ISBACKDROP
Minimize the window	Ph_WM_STATE_ISHIDDEN
Place the base window in front of the windows of all other applications	Ph_WM_STATE_ISFRONT
Give keyboard focus to the window if cursor focus is disabled	Ph_WM_STATE_ISFOCUS
Pass Alt key combinations to the application	Ph_WM_STATE_ISALTKEY
Block the window	Ph_WM_STATE_ISBLOCKED (read-only)

The default value is Ph_WM_STATE_ISFOCUS.

You can get and set the state of the window at any time by using the *Pt_ARG_WINDOW_STATE* resource, but you might get unexpected results if the user is changing the window state at the same time.

The safest time to use this resource to set the window state is before the window is realized. For example, you could set it when creating the **PtWindow** widget or in the window's *Pt_CB_WINDOW_OPENING* callback. The setting will be in effect when the window is realized.

You can set *Pt_ARG_WINDOW_STATE* after the window has been realized, basing your changes on what you think the current window state is, but it's safer to tell the window manager how you want to change the state, by calling:

PtForwardWindowEvent()

Change the state for the window associated with a given region ID

PtForwardWindowTaskEvent()

Change the state for a window associated with a given Photon connection ID

For example, to minimize a window that belongs to your application and is already open:

```
PhWindowEvent_t event;

memset( &event, 0, sizeof (event) );
event.event_f = Ph_WM_HIDE;
event.event_state = Ph_WM_EVSTATE_HIDE;
event.rid = PtWidgetRid( window );
PtForwardWindowEvent( &event );
```

In order to change the state of a window that belongs to another application, you need a connection ID (of type **PhConnectId_t**) for the application. If you have the region ID of a region that belongs to the application, you can call *PhRegionQuery()* and extract the connection ID from the *owner* member of the **PhRegion_t** structure.

If you don't have a region ID, but you know the application's process ID, you can call *PhGetConnectInfo()* like this to get the connection ID:

```
PhConnectId_t get_connect_id( pid_t pid )
{
   PhConnectInfo_t buf;
   PhConnectId_t id = 1;

   while ((id = PhGetConnectInfo(id, &buf)) != -1
          && (buf.pid != pid ||
              ND_NODE_CMP(buf.nid, ND_LOCAL_NODE)))
      ++id;

   return id;
}
```

Once you have the connection ID, you can minimize an open window that belongs to the other application with this code:

```
PhWindowEvent_t event;

memset( &event, 0, sizeof (event) );
event.event_f = Ph_WM_HIDE;
event.event_state = Ph_WM_EVSTATE_HIDE;

PtForwardWindowTaskEvent( connection_id, &event );
```

When you call these functions, you're asking the window manager to do the specified action. If the action isn't set in the managed flags (*Pt_ARG_WINDOW_MANAGED_FLAGS*) for the given window, the window manager doesn't do it.

Managing multiple windows

If your application has more than one window, you'll need to take the relationships between them into account.

By definition, a child window is *always* in front of its parent. The child windows can move above and below siblings. For windows to be able to go behind other windows, they must be siblings. So for a window to be able to move behind the base window, that window would have to have no parent.

Window-manager functions

The following low-level functions are associated with the window manager, but you shouldn't use them in an application that uses widgets:

PhWindowChange()	Modify the attributes of a window's region
PhWindowClose()	Close a window
PhWindowOpen()	Create a window region

These functions can be called in an application that uses widgets:

PhWindowQueryVisible()	Query a visible extent
PtConsoleSwitch()	Switch to another virtual console
PtForwardWindowEvent()	Forward a window event
PtForwardWindowTaskEvent()	Forward a window event to a task
PtWindowConsoleSwitch()	Switch to the console a given window's displayed on
PtWindowGetFrameSize()	Determine the size of a window's frame

Running a standalone application

If your application is intended to run by itself, you might want to:

- Make your application fill the screen. Set Ph_WM_STATE_ISMAX in the base window's *Pt_ARG_WINDOW_STATE* resource.
- Turn off all the flags in the base window's *Pt_ARG_WINDOW_RENDER_FLAGS* so that the window won't get a title bar, borders, menu buttons, and so on — there's no use having them if there aren't any other applications running.

Modal dialogs

Sometimes, you want your program to prompt the user for information before continuing. You usually do this by popping up a dialog; if you don't want the user to be able to select any other operations before providing the information, you should use a *modal dialog*.

A modal dialog doesn't allow user input to go to any of the other widgets in the application. To use a modal dialog to prompt for information, you have to make sure that events are processed within the callback function.

To create a modal dialog, you have to create a new **`PtWindow`** widget, normally as a child of the main application window.

To activate the modal dialog, you have to realize the dialog widget and block all the other window widgets in the application. To block the window or windows, call one of:

PtBlockAllWindows()	Block all windows except the one with a given widget
PtBlockWindow()	Block a given window

Both of these routines return a list of blocked widgets, which you'll need when you unblock them. Instead of blocking the windows, you can make the dialog modal by calling *PtMakeModal()*.

After the modal dialog has been activated, call *PtModalBlock()* to start a modal loop to process Photon events until a termination condition is met.

When the operation associated with the modal dialog is completed or aborted, you have to dismiss the dialog. To do so:

1. Call *PtModalUnblock()* to stop the modal loop. You can specify the value to be returned by *PtModalBlock()*.
2. Destroy or unrealize the dialog itself.
3. Call *PtUnblockWindows()* to unblock any window widgets that you blocked when you created the dialog. You don't need to do this if you called *PtMakeModal()* instead of *PtBlockAllWindows()* or *PtBlockWindow()*.

We can easily change our previous example of work procedures so that its progress dialog behaves as a modal dialog. We'll add a **PtModalCtrl_t** structure to the callback closure, for *PtModalBlock()* and *PtModalUnblock()* to use.

The *done()* callback is altered to call *PtModalUnblock()* rather than free the closure:

```
int done(PtWidget_t *w, void *client,
          PtCallbackInfo_t *call)
{
   CountdownClosure *closure =
      (CountdownClosure *)client;

   call = call;

   if (!closure->done) {
      PtAppRemoveWorkProc(NULL, closure->work_id);
   }
   PtDestroyWidget(closure->dialog->widget);
   free(closure->dialog);

   /* New: end the modal loop, return the counter's
      value as the response. */
   PtModalUnblock(&(closure->modal_control),
                  (void *) &(closure->value));

   return (Pt_CONTINUE);
}
```

All that remains at this point is to change the *push_button_cb()* callback function so that it blocks the window after realizing the progress dialog, starts the modal loop, and unblocks the windows and frees the closure after the dialog is dismissed.

Here's the new version of the *push_button_cb()* callback function:

```
int push_button_cb(PtWidget_t *w, void *client,
                   PtCallbackInfo_t *call)
{
   PtWidget_t   *parent = (PtWidget_t *)client;
   WorkDialog *dialog;
   PtBlockedList_t * blocked_list;
   void * response;

   w = w; call = call;

   dialog = create_working_dialog(parent);

   if (dialog)
    {
      CountdownClosure *closure =
         (CountdownClosure *)
          malloc(sizeof(CountdownClosure));

      if (closure)
        {
         PtWorkProcId_t *id;

         closure->dialog = dialog;
         closure->value = 0;
         closure->maxvalue = 200000;
         closure->done = 0;
         closure->work_id = id =
            PtAppAddWorkProc(NULL, count_cb, closure);

         PtAddCallback(dialog->ok_button, Pt_CB_ACTIVATE,
                       done, closure);
         PtRealizeWidget(dialog->widget);

         /* New: Block all the windows except the dialog,
            process events until the dialog is closed,
            and then unblock all the windows. */

         blocked_list = PtBlockAllWindows (dialog->widget,
                           Ph_CURSOR_NOINPUT, Pg_TRANSPARENT);

         response = PtModalBlock( &(closure->modal_control), 0 );
         printf ("Value reached was %d\n", *(int *)response );
         free (closure);

         PtUnblockWindows (blocked_list);

        }
    }
    return (Pt_CONTINUE);
}
```

Here's the new version of the whole program:

```
#include <stdlib.h>
#include <Pt.h>

typedef struct workDialog {
   PtWidget_t *widget;
   PtWidget_t *label;
   PtWidget_t *ok_button;
} WorkDialog;

typedef struct countdownClosure {
   WorkDialog *dialog;
   int value;
   int maxvalue;
   int done;
   PtWorkProcId_t *work_id;

   /* New member: */
   PtModalCtrl_t modal_control;
} CountdownClosure;


WorkDialog *create_working_dialog(PtWidget_t *parent)
{
   PhDim_t      dim;
   PtArg_t      args[3];
   int          nargs;
   PtWidget_t   *window, *group;
   WorkDialog *dialog =
      (WorkDialog *)malloc(sizeof(WorkDialog));

   if (dialog)
    {
      nargs = 0;
      PtSetArg(&args[nargs], Pt_ARG_WIN_PARENT, parent, 0);
               nargs++;
      PtSetParentWidget(NULL);
      dialog->widget = window =
         PtCreateWidget(PtWindow, parent, nargs, args);

      nargs = 0;
      PtSetArg(&args[nargs], Pt_ARG_GROUP_ORIENTATION,
               Pt_GROUP_VERTICAL, 0); nargs++;
      PtSetArg(&args[nargs], Pt_ARG_GROUP_VERT_ALIGN,
               Pt_GROUP_VERT_CENTER, 0); nargs++;
      group = PtCreateWidget(PtGroup, window, nargs, args);

      nargs = 0;
      dim.w = 200;
      dim.h = 100;
      PtSetArg(&args[nargs], Pt_ARG_DIM, &dim, 0); nargs++;
      PtSetArg(&args[nargs], Pt_ARG_TEXT_STRING,
               "Counter:        ", 0); nargs++;
      dialog->label = PtCreateWidget(PtLabel, group,
                                        nargs, args);

      PtCreateWidget(PtSeparator, group, 0, NULL);

      nargs = 0;
      PtSetArg(&args[nargs], Pt_ARG_TEXT_STRING, "Stop", 0);
               nargs++;
      dialog->ok_button = PtCreateWidget(PtButton, group,
                                            1, args);
    }
   return dialog;
}

int done(PtWidget_t *w, void *client,
          PtCallbackInfo_t *call)
{
   CountdownClosure *closure =
      (CountdownClosure *)client;

   call = call;

   if (!closure->done) {
      PtAppRemoveWorkProc(NULL, closure->work_id);
   }
   PtDestroyWidget(closure->dialog->widget);
   free(closure->dialog);

   /* New: end the modal loop, return the counter’s
      value as the response. */
   PtModalUnblock(&(closure->modal_control),
                  (void *) &(closure->value));

   return (Pt_CONTINUE);
}

int
count_cb(void *data)
{
   CountdownClosure *closure =
      (CountdownClosure *)data;
   char    buf[64];
   int     finished = 0;

   if ( closure->value++ == 0 || closure->value %
        1000 == 0 )
    {
      sprintf(buf, "Counter: %d", closure->value);
      PtSetResource( closure->dialog->label,
                     Pt_ARG_TEXT_STRING, buf, 0);
     }

   if ( closure->value == closure->maxvalue )
     {
      closure->done = finished = 1;
      PtSetResource( closure->dialog->ok_button,
                     Pt_ARG_TEXT_STRING, "Done", 0);
     }

    return finished ? Pt_END : Pt_CONTINUE;
}

int push_button_cb(PtWidget_t *w, void *client,
                   PtCallbackInfo_t *call)
{
   PtWidget_t   *parent = (PtWidget_t *)client;
   WorkDialog *dialog;
   PtBlockedList_t * blocked_list;
   void * response;

   w = w; call = call;

   dialog = create_working_dialog(parent);

   if (dialog)
    {
      CountdownClosure *closure =
         (CountdownClosure *)
          malloc(sizeof(CountdownClosure));

      if (closure)
        {
         PtWorkProcId_t *id;

         closure->dialog = dialog;
         closure->value = 0;
         closure->maxvalue = 200000;
         closure->done = 0;
         closure->work_id = id =
            PtAppAddWorkProc(NULL, count_cb, closure);

         PtAddCallback(dialog->ok_button, Pt_CB_ACTIVATE,
                       done, closure);
         PtRealizeWidget(dialog->widget);

         /* New: Block all the windows except the dialog,
            process events until the dialog is closed,
            and then unblock all the windows. */

         blocked_list = PtBlockAllWindows (dialog->widget,
                             Ph_CURSOR_NOINPUT, Pg_TRANSPARENT);

         response = PtModalBlock( &(closure->modal_control), 0 );
         printf ("Value reached was %d\n", *(int *)response );
         free (closure);

         PtUnblockWindows (blocked_list);

        }
    }
    return (Pt_CONTINUE);
```

```
}

int main(int argc, char *argv[])
{
   PhDim_t      dim;
   PtArg_t      args[3];
   int          n;
   PtWidget_t   *window;
   PtCallback_t callbacks[] = {{push_button_cb, NULL
   }
   };
   char Helvetica14b[MAX_FONT_TAG];

   if (PtInit(NULL) == -1)
      exit(EXIT_FAILURE);

   dim.w = 200;
   dim.h = 100;
   PtSetArg(&args[0], Pt_ARG_DIM, &dim, 0);
   if ((window = PtCreateWidget(PtWindow, Pt_NO_PARENT,
                                1, args)) == NULL)
      PtExit(EXIT_FAILURE);

   callbacks[0].data = window;
   n = 0;
   PtSetArg(&args[n++], Pt_ARG_TEXT_STRING, "Count Down...", 0);

   /* Use 14-point, bold Helvetica if it’s available. */

   if(PfGenerateFontName("Helvetica", PF_STYLE_BOLD, 14,
                         Helvetica14b) == NULL) {
       perror("Unable to generate font name");
   } else {
       PtSetArg(&args[n++], Pt_ARG_TEXT_FONT, Helvetica14b, 0);
   }
   PtSetArg(&args[n++], Pt_CB_ACTIVATE, callbacks,
            sizeof(callbacks)/sizeof(PtCallback_t));
   PtCreateWidget(PtButton, window, n, args);

   PtRealizeWidget(window);

   PtMainLoop();
   return (EXIT_SUCCESS);
}
```

If your modal dialog is self-contained and you just need to wait for it, you might find this function useful:

ApModalWait()	Process Photon events until a given widget is destroyed

Chapter 27

Programming Photon without PhAB

In this chapter. . .

We strongly recommend that you use PhAB to develop Photon applications—this chapter is for those who insist on not using PhAB.

Basic steps

All applications using the Photon widget library follow the same basic sequence:

1. Include **`<Pt.h>`**, the standard header file for the widget library.
2. Initialize the Photon widget toolkit by calling *PtInit()* (or *PtAppInit()*, which also creates the main window).
3. Create the widgets that make up the UI by calling *PtCreateWidget()* . This function can make the new widgets into children of a given widget or the current container, or with no parent.
4. Register any callback functions in the application with the appropriate widgets using *PtAddCallback()* or *PtAddCallbacks()*.
5. Realize the widgets by calling *PtRealizeWidget()*. This function needs to be called only once by the application.

   The *realize* step actually creates any Photon regions that are required and maps them to the screen. Until this step is performed, no regions exist, and nothing is displayed on the screen.
6. Begin processing photon events by calling *PtMainLoop()*.

   At this point, the Photon widget toolkit takes control over the application and manages the widgets. If any widgets are to call functions in your application, they must have been registered as callbacks before this.

Compiling and linking a non-PhAB application

To compile and run an application that uses the Photon widget library, you must link against the main Photon library, **`ph`**. There are both static and shared versions of this library.

CAUTION:

The **`libphoton.so.1`** library is for applications created with version 1.14 of the Photon microGUI only. Don't combine this library with the current libraries or header files, or your application won't run properly.

We recommend that you always link against the *shared* library. This lets you keep your applications smaller and allows them to inherit new features that are added to the widget library when new releases of the shared library are installed.

The Photon library includes most of the function and widget definitions. If your application uses **`Al`** (translation) or **`Px`** (extended) functions, you'll also need to link with the **phexlib** library. If your application uses **Ap** (PhAB) functions, you'll also need to link with the **Ap** library.

The names of the shared and static libraries are the same. By default, **qcc** links against the shared library; to link against the static library, specify the **-Bstatic** option for **qcc**.

For example, if we have an application called **hello.c**, the command to compile and link against the shared libraries is:

```
qcc -o hello hello.c -lph
```

To link against the static libraries, the command is:

```
qcc -o hello hello.c -Bstatic -lph -lfont
```

Sample application

The following example illustrates a very simple application using the widget library. The program creates a window that contains a single pushbutton.

```
/*
 * File: hello.c
 */
#include <Pt.h>

int main( int argc, char *argv[] )
{
   PtWidget_t *window;
   PtArg_t    args[1];

   if (PtInit(NULL) == -1)
      PtExit(EXIT_FAILURE);

   window = PtCreateWidget(PtWindow, Pt_NO_PARENT, 0, NULL);

   PtSetArg(&args[0], Pt_ARG_TEXT_STRING,
            "Press to exit", 0);
   PtCreateWidget(PtButton, window, 1, args);
   PtRealizeWidget(window);

   PtMainLoop();
   return (EXIT_SUCCESS);
}
```

What's going on

Although this is a simple application, a lot of work is being done by each of these calls.

PtInit()

PtInit() calls *PhAttach()* to attach a channel to the Photon server, and then initializes the widget libraries.

PtCreateWidget() — first call

The first call to *PtCreateWidget()* creates a window widget that interacts with the window manager and serves as the parent for other widgets created in the application. The arguments are:

- the class of widget to create (**`PtWindow`** in this case)
- the window's parent (Pt_NO_PARENT because the window doesn't have one)
- the number of elements in the argument list
- an argument list of initial values for the widget's resources.

PtCreateWidget() returns a pointer to the widget created.

For more information, see "Creating widgets" in the Managing Widgets in Application Code chapter. For more information about widgets and their resources, see the Photon *Widget Reference*.

PtSetArg()

The *PtSetArg()* macro sets up an argument list that's used to initialize the button's resources when it's created. For more information, see the Manipulating Resources in Application Code chapter.

PtCreateWidget() — second call

All the widgets in the application — except the top-level window — have a container widget as a parent. Container widgets may have other containers within them. Creating the widgets in the application produces a hierarchy called the *widget family*.

The second call to *PtCreateWidget()* creates a pushbutton widget as a child of the window widget, using the argument list to initialize the button's resources. You can pass Pt_DEFAULT_PARENT as the parent to make the widget the child of the most recently created container widget; in this case, the results are the same.

PtRealizeWidget()

PtRealizeWidget() displays the widget and all its descendants in the widget family. Our sample application calls *PtRealizeWidget()* for the top-level window, so all the widgets in the application are displayed.

When a widget is realized, it uses the values of its resources to determine how big it must be to display its contents. Before realizing a widget, you should set any of the resources that may affect its size. You may change some of the resources after the widget has been realized, but it's up to the widget to determine if it can or will resize to accommodate the change in the resource's value.

You can set *resize flags* that the widget uses to determine whether or not to adjust its size in response to such changes, but note that if the widget exceeds the dimensions allocated to it by its parent, it's clipped to the parent's size. There's no mechanism for the widget to negotiate with its parent to obtain more space. See the Geometry Management chapter for more information.

If a Photon *region* is required to display the widget correctly, it's created each time the widget is realized. A region is required under *any* of the following conditions:

- The widget sets a cursor.
- The widget needs to get events that aren't redirected to it by its parent container (e.g. boundary, pointer-motion events)
- The Pt_REGION flag is set in the widget's *Pt_ARG_FLAGS* resource (see **`PtWidget`** in the *Widget Reference*).

You can unrealize a widget by calling *PtUnrealizeWidget()*. This affects the visibility of the widget and its descendants, but not the rest of the widget family hierarchy. You can redisplay the widget later by calling *PtRealizeWidget()*.

You can prevent a widget and its descendants from being realized when the widget's ancestor is realized. To do this, set Pt_DELAY_REALIZE in the widget's *Pt_ARG_FLAGS* resource. If you set this flag, it's your responsibility to call *PtRealizeWidget()* on the widget when you want it to appear.

PtMainLoop()

Calling *PtMainLoop()* transfers control of the application to the Photon widget library.

The widget library waits for Photon events and passes them on to the widgets to handle them. Application code is executed only when callback functions that the application has registered with a widget are invoked as a result of some event.

Connecting application code to widgets

If you compile, link, and run the sample application, you'll see that a window appears with a button in it. If you push the button, nothing happens because no application code has been associated with it.

The Photon widget library is designed so that the UI code can be kept distinctly separate from the application code. The UI is composed of the code to create and manipulate the widget family hierarchy, and must call the application code in response to particular events or user actions. The connection between the application code and the UI that allows it to use the application code is the single point where these two parts have intimate knowledge of each other.

Connections are made between the UI and the application code using *callbacks* and *event handlers*.

A callback is a special type of widget resource that allows the application to take advantage of existing widget features. Using a callback, the application can register a function to be called by the widget library later in response to a particular occurrence within the widget.

Event handlers (raw and filter callbacks) are normally used to add capabilities to a widget. For example, you could add behavior to a button press inside a widget that has no callbacks associated with button-press events.

Callbacks

A callback resource is used to notify the application that a specific action has occurred for a widget (e.g. you've selected a button). Every callback resource represents some user action that we thought your application might be interested in.

As with all resources, a widget has its callback resources defined by its widget class, and it inherits the callback resources defined by all the ancestors of its class. This means that a widget may have several user actions that it can notify the application about.

The value of a callback resource is a callback list. Each element of the list is an application function to be called in response to the behavior and *client data* associated with the callback. Client data is a pointer to any arbitrary data that your application may need to provide to the callback function for it to work correctly.

For information about callbacks, see "Callbacks" in the Managing Widgets in Application Code chapter.

Event handling

When we create a widget class, we can't possibly anticipate all of your application's needs. Your application may want to be notified of some occurrence on a widget that doesn't have an associated callback resource. In such cases, your application can define event-handling functions.

For information about event handlers, see "Event handlers" in the Managing Widgets in Application Code chapter.

Complete sample application

We can now use our newly acquired knowledge of resources and callbacks to create a more functional version of the sample application given earlier.

Using resources, we can give the pushbutton widget the same dimensions as the window, and specify which font to use for the label's text. We can also define the callback to be executed when the pushbutton is pressed. We'll make the callback function display a simple message and exit.

Here's the complete source code for our sample program with these changes:

```
#include <stdio.h>
#include <stdlib.h>
#include <Pt.h>

int main( int argc, char *argv[] )
{
    PtArg_t args[3];
    int n;
    PtWidget_t *window;
    int push_button_cb( PtWidget_t *, void *,
                        PtCallbackInfo_t *);
    PtCallback_t callbacks[] = {{push_button_cb, NULL}};
    char Helvetica14[MAX_FONT_TAG];
    if (PtInit(NULL) == -1)
       PtExit(EXIT_FAILURE);

    window = PtCreateWidget(PtWindow, Pt_NO_PARENT, 0, NULL);

    n = 0;
    PtSetArg(&args[n++], Pt_ARG_TEXT_STRING,
             "Press to exit", 0);

    /* Use 14-point, bold Helvetica if it’s available. */

    if(PfGenerateFontName("Helvetica", 0, 14,
                          Helvetica14) == NULL) {
        perror("Unable to generate font name");
    } else {
        PtSetArg(&args[n++], Pt_ARG_TEXT_FONT, Helvetica14, 0);
    }
    PtSetArg(&args[n++], Pt_CB_ACTIVATE, callbacks,
        sizeof(callbacks)/sizeof(callbacks[0]));
    PtCreateWidget(PtButton, window, n, args);

    PtRealizeWidget(window);
    PtMainLoop();
    return (EXIT_SUCCESS);
}

int
push_button_cb(PtWidget_t *w, void *data,
               PtCallbackInfo_t *cbinfo)
{
    printf(  "I was pushed\n " );
    PtExit( EXIT_SUCCESS );

    /* This line won’t be reached, but it keeps
       the compiler happy. */

    return( Pt_CONTINUE );
}
```

Appendix A

Photon Architecture

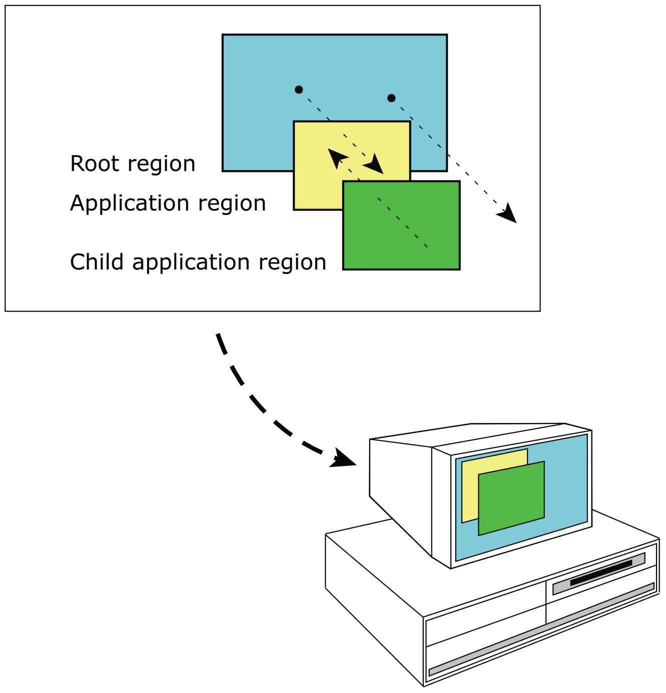
In this appendix. . .

This appendix provides a technical overview of Photon's architecture.

Event space

The essential characteristic of Photon is the way in which graphical applications are represented. All Photon applications consist of one or more rectangles called *regions*. These regions reside in an abstract, three-dimensional *event space*; the user is outside this space looking in.

Event space



Regions can emit and collect objects called *events*. These events can travel in either direction through the event space (i.e. either toward or away from the user). As events move through the event space, they interact with other regions — this is how applications interact with each other. The process maintaining this simple architecture is the *Photon Manager*.

All the services required for a windowing system — window managers, drivers, and applications — can easily be created using regions and events. And because processes whose regions are managed by the Photon Manager needn't reside on the same computer as the Photon Manager, it's also easy to implement network-distributed applications.

Regions and events

Photon programs use two basic objects: regions and events. Regions are stationary, while events move through the event space.

A region is a *single*, fixed rectangular area that a program places in the event space. A region possesses attributes that define how it interacts with events.

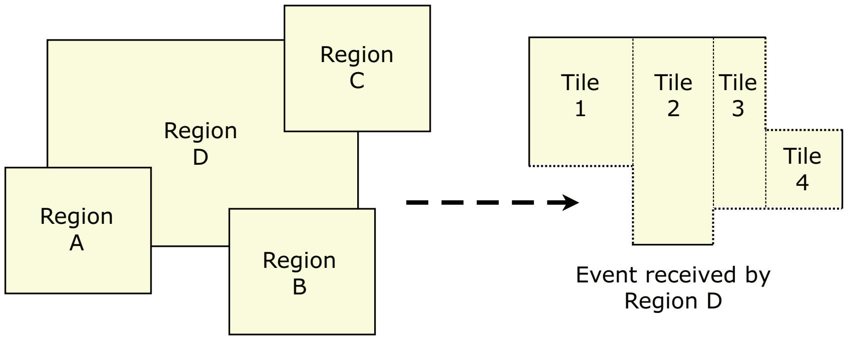
An event is a *set* of nonoverlapping rectangles that can be emitted and collected by regions in either direction in the event space. All events have an associated *type*. Some event types also possess corresponding data.

Events

As an event flows through the event space, its rectangle set intersects with regions placed in the event space by other applications. As this occurs, the Photon Manager adjusts the event's rectangle set according to the attributes of the regions with which the event intersected.

Initial rectangle set

An emitted event's default initial rectangle set contains a single rectangle whose dimensions are usually the size of the emitting region. As the event moves through the event space, its interactions with other regions may cause some portions of this rectangle to be removed. If this happens, the rectangle will be divided into a set of smaller rectangles that represent the remaining portions:



An event's rectangle set.

Certain event types (e.g. button presses) have no need for their initial rectangle set to have the dimensions of the emitting region. For such events, the rectangle set consists of a single rectangle whose size is a single point. A single-point rectangle set is called a *point source*.

Collected rectangle set

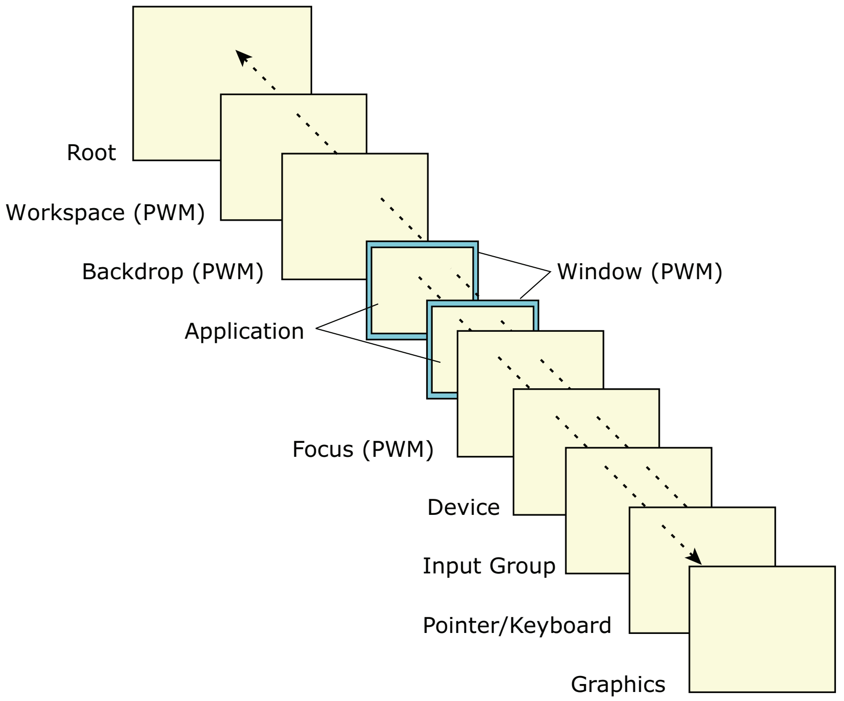
The rectangle set of a "collected" event contains the rectangles resulting from the event's interaction with prior regions in the event space. If an event is completely occluded by other regions such that it results in a set containing no rectangles, then that event ceases to exist.

For a list of the event types, see the "Event types" section.

Regions

A process may create or use any number of regions and may place them anywhere in the event space. Furthermore, by controlling the dimensions, location, and attributes of a region (relative to the other regions in the event space), a process can use, modify, add, or remove services provided by other regions.

Photon uses a series of regions, ranging from the Root region at the back of the Photon event space to the Graphics region at the front. Draw events start at an application's region and move forward to the Graphics region. Input events start at the Pointer/Keyboard region and travel towards the Root region.



Exploded view of Photon's regions.

The following constants are defined in **`<photon/PhT.h>`**:

- Ph_DEV_RID — the ID of the device region.
- Ph_ROOT_RID — the ID of the root region.

A region's owner and the Photon Manager can reside on different computers.

A region has two attributes that control how events are to be treated when they intersect with a region:

- *sensitivity*
- *opacity*

You can set these independently for each different type of event.

Sensitivity

If a region is sensitive to a particular type of event, then the region's owner collects a copy of any event of that type that intersects with the region. If other regions are sensitive to this same event type and the event intersects with them, they'll also collect a copy of the event — but with a potentially different rectangle set.

Although many regions can collect a copy of the same event, the rectangle set for the event may be adjusted and will therefore be unique for each region that collects the event. The rectangle set reflects the event's interaction with other regions in the event space before arriving at the collecting region.

If a region isn't sensitive to an event type, the region's owner never collects that type of event.

The sensitivity attribute doesn't modify the rectangle set of an event, nor does it affect the event's ability to continue flowing through the event space.

Opacity

Opaque regions block portions of an event's rectangle set from traveling further in the event space. The opacity attribute controls whether or not an event's rectangle set is adjusted as a result of intersecting with a region.

If a region is opaque to an event type, any event of that type that intersects with the region has its rectangle set adjusted in order to "clip out" the intersecting area. This changes the rectangle set such that it includes smaller rectangles. These new rectangles describe the portions of the event that remain visible to regions beyond this region in the event space.

If a region *isn't* opaque to an event type, then events of that type never have their rectangle set adjusted as a result of intersecting with that region.

Attribute summary

The following table summarizes how a region's attributes affect events that intersect with that region:

If the region is:	**Then the event is:**	**And the rectangle set is:**
Not sensitive, not opaque	Ignored	Unaffected
Not sensitive, opaque	Ignored	Adjusted
Sensitive, not opaque	Collected	Unaffected
Sensitive, opaque	Collected	Adjusted

Event logging

By placing a region across the entire event space, a process can intercept and modify any event passing through that region. If a region is sensitive to all events, but not opaque, it can transparently log all events.

Event modification

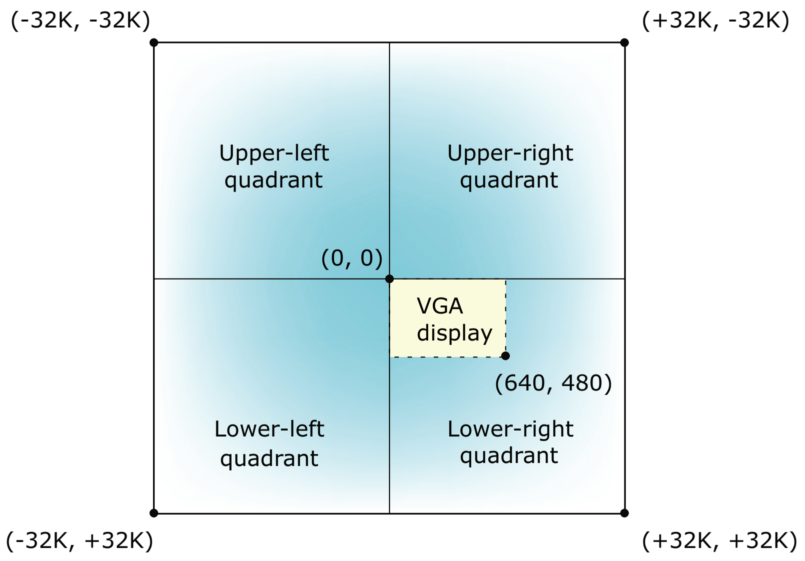
If a region is sensitive *and* opaque, it can choose to reemit a modified version of the event. For example, a region could collect pointer events, perform handwriting recognition on those events, and then generate the equivalent keyboard events.

Parent/child relationships

All regions have parent/child relationships. A child region is always placed in front of the parent region (i.e. closer to the user), and its coordinates are relative to the parent's region.

Photon coordinate space

All regions reside within the Photon coordinate space, whose dimensions are as follows:



Root region

A special region called the *root region* is always the region furthest away from the user. All other regions descend in some way from the root region. Once an event traveling away from the user reaches the root region, it ceases to exist.

The root region's dimensions are the width of the entire Photon coordinate space. As a result of the parent/child relationship of all regions, any region's location is ultimately related to the root region's dimensions.

A region can be located anywhere in the event space and still have the root region as its parent.

Event types

Events are emitted for the following reasons:

- key presses, keyboard state information
- button presses and releases
- pointer motions (with or without button pressed)
- boundary crossings
- regions exposed or covered
- drag operations
- drag-and-drop operations
- drawing functions

For more information on event types, see **`PhEvent_t`** in the Photon *Library Reference*.

How region owners are notified of events

Region owners can be notified of events by the Photon Manager in three different ways:

- polling
- synchronous notification
- asynchronous notification

Polling

To poll, the application calls a function that asks the Photon Manager to reply immediately with either an event or a status indicating no event is available.

Normally you should avoid polling, but you may find it beneficial on occasion. For example, an application rapidly animating a screen can poll for events as part of its stream of draw events. An application can also use polling to retrieve an event after asynchronous notification.

Synchronous notification

For synchronous notification, the application calls a function that asks the Photon Manager to reply immediately if an event is pending, or to wait until one becomes available before replying.

With synchronous notification, an application can't block on other sources while it's waiting for the Photon Manager to reply. You should find this behavior acceptable in most cases since it causes the application to execute only when the desired events become available. But if for some reason the possibility of blocking on the Photon Manager isn't acceptable, you may consider asynchronous notification.

Asynchronous notification

For asynchronous notification, the application calls a function that sets up a notification method (e.g. a signal or a pulse) that the Photon Manager activates when an event of the desired type is available. The application can then retrieve the event by polling.

With asynchronous notification, an application can block on multiple sources, including processes that aren't Photon applications.

Device region

The device region is owned by the Photon Manager, which divides the event space into two sections:

- *driver regions*, which reside on the user's side of the device region
- *application regions*, which reside on the other side of the device region

The Photon Manager uses the device region to focus pointer and keyboard events as well as to manage drag events.

Pointer focus

As with other windowing systems, Photon has the concept of a *pointer* (i.e. screen cursor). This pointer is graphically represented on the screen and tracks the movements of the pointing device (e.g. a mouse or touchscreen). Drivers for pointing devices emit pointer events toward the root region.

A pointer event emitted from a driver is *unfocused*, or *raw*, until it arrives at the device region, where the Photon Manager intercepts it and then assigns it a location in the Photon coordinate space.

Assigning this location — which is known as *focusing* the event — controls which regions will collect the event. The Photon Manager then reemits the event with the focused location.

Because Photon emits *focused*, or *cooked*, pointer motion events in both directions from the device region, application programs as well as driver programs can be informed of pointer actions. For example, when the graphics driver collects focused pointer events, it updates the location of the pointer's graphical image on the screen.

Keyboard focus

The keyboard driver is similar to pointing device drivers, except it emits keyboard events. As with pointer events, keyboard events are unfocused until they arrive at the device region, where the Photon Manager assigns them a location (i.e. focuses them) in the Photon coordinate space.

By default, the device region sets the same focus location for both keyboard events and pointer events. Therefore, regions directly behind the screen pointer will collect focused keyboard events.

The window manager supplements the keyboard focus methods. For more information, see the section on the Photon window manager.

Drag events

An application initiates dragging by emitting a drag event to the device region. Once this event is collected at the device region, the Photon Manager takes care of the interaction with the pointer (i.e. drag rectangle) until the drag operation is complete. Once completed, the device region emits a drag event to the application.

Drag-and-drop events

During a drag-and-drop operation, a series of events is emitted to advise the widgets involved of the operation's status. Some of these events are emitted to the source of the operation, and others to the destination. For more information, see the Drag and Drop chapter.

Photon drivers

In Photon, drivers aren't inherently different from other applications. They're simply programs that use regions and events in a particular way to provide their services. Depending on its function, a driver is either an *input driver* or an *output driver*.

For example, the mouse and keyboard drivers are input drivers because they emit, and are the source of, hardware actions. Graphics drivers, on the other hand, are output drivers because they collect events that cause them to take action with hardware devices.

Input drivers

Mouse driver

The mouse driver places a region on the user's side of the device region. It gets information from the mouse hardware and builds Photon raw pointer events that it then emits toward the root region.

When the device region collects a raw pointer event, the Photon Manager focuses it and then emits cooked events in both directions in the event space.

Keyboard driver

The keyboard driver also places a region on the user's side of the device region. The driver gets information from the keyboard hardware and builds Photon keyboard events that it then emits toward the root region.

When the device region collects keyboard events, the Photon Manager focuses those events and then reemits them toward the root region.

The window manager supplements the default focus method provided by the device region.

Since Photon makes no assumptions as to what type of keyboard is being used, the keyboard driver could acquire its event data on any hardware or even from another process.

Photon allows multiple input drivers and multiple output drivers to be associated with each other as an *input group*. This group of devices will be treated distinctly from other input groups by Photon.

To determine the current input group, call *PhInputGroup()*, passing to it the current event, if any.

Output drivers

Graphics driver

A graphics driver places a region sensitive to draw events onto the user's side of the device region. As the driver collects draw events, it renders the graphical information on the screen. Because the collected event's rectangle set contains only those areas that need to be updated, the driver can optimize its update. (This is especially efficient if the graphics hardware can handle clipping lists directly.)

The Photon drawing API accumulates draw requests into batches that are emitted as single draw events.

Multiple graphic drivers

The region a graphics driver uses can have dimensions that represent an area smaller than the entire Photon coordinate space. As a result, multiple graphics drivers can share the coordinate space by each handling different portions of it and displaying their events on different screens. And since region owners don't have to be on the same node as the Photon Manager, those graphics drivers can display their portion of the coordinate space on screens located on other computers in the network.

Drivers using separate regions

From an application's perspective, the Photon coordinate space always looks like a single, unified graphical space, yet it allows users to drag windows from one physical screen to another.

For example, let's say an operator in a factory control environment has a small handheld computer. If this computer were connected to network via a wireless link, the operator could walk up to a computer with a large-screen control application and drag a window from that screen onto the screen of the handheld unit. Taking the handheld computer, the operator could then walk out onto the plant floor and continue to interact with the control application to monitor and adjust equipment.

Drivers using overlapping regions

In another type of example, instead of having driver regions with exclusive portions of the coordinate space, you could have drivers using *overlapping* regions. This approach would let you replicate the same draw events on multiple screens, which would be ideal for support or training environments.

Encapsulation drivers

Since graphics drivers for Photon are really just applications, they can display the graphical output of Photon inside another windowing system (for example, the X Window System). A Photon driver could also take the keyboard and mouse events it collects from the X system and regenerate them within Photon, allowing the Photon window in the X system to be fully functional, both for graphical display and for keyboard/mouse input.

Photon window manager

The window manager is an optional Photon application that manages the appearance and operation of menus, buttons, scrollbars, and so on. It provides the windowing system with a certain "look and feel" (e.g. Motif).

The window manager also manages the *workspace*, supplements the methods for focusing keyboard events, and lets you display a *backdrop*. To provide all these services, the window manager places several regions in the event space:

- Window-frame regions
- Focus region
- Workspace region
- Backdrop region

Window-frame regions

Most applications rely on the windowing system to provide the user with the means to manipulate their size, position, and state (i.e. open/iconified). In order for the user to perform these actions, the window manager puts a frame around the application's region and then places controls in that frame (e.g. resize corners, title bars, buttons). We refer to these controls as *window services*.

To indicate it can provide window services, the window manager registers with the Photon Manager. When an application opens a window, the window manager sets up two regions on its behalf: a window frame region and an application region (or window region). The window frame region is slightly larger than the application region and is placed just behind it.

The window manager uses the window frame region for its controls, while the application uses its own region. But the application isn't aware of the controls. If the user uses the controls to move the application, the application notices only that its location has changed. The same goes for resizing, iconifying, and so on.

Focus region

As mentioned earlier, the device region focuses keyboard events to regions directly behind the screen pointer. But by placing a region of its own (i.e. the *focus region*) just behind the device region, the window manager intercepts these keyboard events as they're emitted from the device region and implements an alternate focus method.

The window manager can redirect keyboard events to regions not directly beneath the screen pointer. For example, it can focus events toward the last window the user "clicked" on (i.e. the *active window*). The window manager can direct keyboard events to that active region even if the region gets covered by another region.

Workspace region

From the user's perspective, the workspace is the empty space surrounding the windows on the screen. The window manager places a *workspace region* just in front of the root region to capture pointer events before they get to the root region and thus disappear. When the user presses a pointer button and no region collects the event, the window manager brings up a *workspace menu* that lets the user select a program to run.

Backdrop region

Users often like to have an ornamental backdrop image displayed behind the windows on the screen. To display such a bitmap, the window manager places a *backdrop region* in the event space.

Appendix B

Widgets at a Glance

The following table lists the Photon widget classes and the icons used in PhAB's widget palette. For more information on specific widget classes, see the Photon *Widget Reference*.

PhAB Icon	Class	Description
Arc	`PtArc`	An elliptical arc
BarGraph	`PtBarGraph`	Bar graph
Basic	`PtBasic`	A superclass of basic resources for most widgets
Bezier	`PtBezier`	Bézier curve
Background	`PtBkgd`	Background of tiled images, gradients, or bitmaps
Button	`PtButton`	A button for initiating an action
Calendar	`PtCalendar`	Calendar
N/A	`PtClient`	Superclass for client widgets — not normally instantiated
2:00 Clock	`PtClock`	Analog, digital, or LED clock
Color Panel	`PtColorPanel`	A color panel
Color Patch	`PtColorPatch`	A widget for selecting a hue and shading or tint
N/A	`PtColorSel`	Superclass for color-selector widgets—not normally instantiated
Color Sel Group	`PtColorSelGroup`	A group of color selectors
Color Well	`PtColorWell`	A rectangle that displays a color and lets you change it
Combo Box	`PtComboBox`	Text-entry field with a list of choices
N/A	`PtCompound`	Superclass for compound widgets—not normally instantiated
Container	`PtContainer`	Superclass for container widgets
N/A	`PtDisjoint`	Superclass for disjoint widgets—not normally instantiated
Divider	`PtDivider`	Widget that divides a given space among its child widgets and allows resizing

continued...

PhAB Icon	Class	Description
Ellipse	`PtEllipse`	Ellipse
File Selector	`PtFileSel`	A tree widget for selecting files and directories
Font Selector	`PtFontSel`	A widget for selecting font attributes
N/A	`PtGauge`	Superclass for gauge-like widgets—not normally instantiated
N/A	`PtGenList`	Superclass for list widgets—not normally instantiated
N/A	`PtGenTree`	Superclass for tree widgets—not normally instantiated
N/A	`PtGraphic`	Superclass for graphical widgets—not normally instantiated
Grid	`PtGrid`	Grid pattern
N/A	`PtGroup`	Group—use PhAB's Group Together button to create this
Image Area	`PtImageArea`	An area for viewing an image
Label	`PtLabel`	A text, bitmap, or image label
Line	`PtLine`	Straight line (single segment)
List	`PtList`	List of text items
N/A	`PtMenu`	Menu—use a Menu module instead
Menu Bar	`PtMenuBar`	Menubar that's placed at the top of a window
Menu Button	`PtMenuButton`	Button that pops up a menu, or an item in a menu
Meter	`PtMeter`	Meter widget
MTrend	`PtMTrend`	Medical trend widget
Multi Text	`PtMultitext`	Multiple-line stylized text field
N/A	`PtNumeric`	Numeric field superclass—not normally instantiated
Numeric Float	`PtNumericFloat`	Floating-point numeric field
Numeric Integer	`PtNumericInteger`	Integer field

continued...

PhAB Icon	Class	Description
On/Off Button	`PtOnOffButton`	Button that's either on or off
OSContainer	`PtOSContainer`	Offscreen-context container, useful for drawing flicker-free images and animations
Pane	`PtPane`	Container that manages its children
Panel Group	`PtPanelGroup`	Container that manages panels
Pixel	`PtPixel`	Set of points
Polygon	`PtPolygon`	Set of connected line segments
Print Selector	`PtPrintSel`	Compound widget for choosing printing options
Progress	`PtProgress`	Progress bar
Raw	`PtRaw`	Widget in which you can use low-level Pg drawing functions
RawList	`PtRawList`	Raw list
RawTree	`PtRawTree`	raw tree
Rect	`PtRect`	Rectangle
N/A	`PtRegion`	Photon region—must be created with *PtCreateWidget()*
N/A	`PtScrollArea`	Superclass for scrolling widgets—not normally instantiated
Scroll Bar	`PtScrollBar`	Scrollbar
Scroll Container	`PtScrollContainer`	A viewport for viewing a large virtual area
Separator	`PtSeparator`	Separator
N/A	`PtServer`	Server widget — must be created with *PtCreateWidget()*
Slider	`PtSlider`	Numerical input mechanism with a range
Tab	`PtTab`	Terminal emulator
Terminal	`PtTerminal`	Terminal emulator
Text	`PtText`	Single-line text field

continued…

PhAB Icon	Class	Description
Timer	`PtTimer`	Timer
Toggle Button	`PtToggleButton`	Toggle button
Toolbar	`PtToolbar`	Superclass for toolbar widgets
Toolbar Group	`PtToolbarGroup`	A group of toolbars
Tree	`PtTree`	Hierarchy tree
Trend	`PtTrend`	Display of connected points that shift in a specified direction at the rate in which data is fed
Tty	`PtTty`	Terminal device
Up/Down Button	`PtUpDown`	Increment/decrement button
Web Client	`PtWebClient`	Widget for displaying web pages
N/A	`PtWidget`	Widget superclass—not normally instantiated
N/A	`PtWindow`	Window—use a Window module instead

Appendix C

Unicode Multilingual Support

In this appendix. . .

Photon is designed to handle international characters. Following the Unicode standard, Photon provides developers with the ability to create applications that can easily support the world's major languages and scripts.

Unicode is modeled on the ASCII character set, but uses a 32-bit encoding to support full multilingual text. There's no need for escape sequences or control codes when specifying any character in any language. Note that Unicode encoding conveniently treats all characters — whether alphabetic, ideographs, or symbols — in exactly the same way.

In designing the keyboard driver and the character handling mechanisms, we referred to the X11 keyboard extensions and ISO standards 9995 and 10646-1.

Wide and multibyte characters

ANSI C includes the following concepts:

wide character	A character represented as a value of type **`wchar_t`**, which typically is larger than a **`char`**.
multibyte character	A sequence of one or more bytes that represents a character, stored in a **`char`** array. The number of bytes depends on the character.
wide-character string	An array of **`wchar_t`**.
multibyte string	A sequence of multibyte characters stored in a **`char`** array.

Unicode

Unicode is a 32-bit encoding scheme:

- It packs most international characters into wide-character representations (two bytes per character).
- Codes below 128 define the same characters as the ASCII standard.
- Codes between 128 and 255 define the same characters as in the ISO 8859-1 character set.
- There's a private-use area from **`0xE000`** to **`0xF7FF`**; Photon maps it as follows:

Glyphs	Range
Nondisplayable keys	`0xF000` – `0xF0FF`
Cursor font	`0xE900` – `0xE9FF`

For Unicode character values, see **`/usr/include/photon/PkKeyDef.h`**. For more information about Unicode, see the Unicode Consortium's website at **`www.unicode.org`**.

UTF-8 encoding

Formerly known as UTF-2, the UTF-8 (for "8-bit form") transformation format is designed to address the use of Unicode character data in 8-bit UNIX environments. Each Unicode value is encoded as a multibyte UTF-8 sequence.

Here are some of the main features of UTF-8:

- The UTF-8 representation of codes below 128 is the same as in the ASCII standard, so any ASCII string is also a valid UTF-8 string and represents the same characters.
- ASCII values don't otherwise occur in a UTF-8 transformation, giving complete compatibility with historical filesystems that parse for ASCII bytes.
- UTF-8 encodes the ISO 8859-1 character set as double-byte sequences.
- UTF-8 simplifies conversions to and from Unicode text.
- The first byte indicates the number of bytes to follow in a multibyte sequence, allowing for efficient forward parsing.
- Finding the start of a character from an arbitrary location in a byte stream is efficient, because you need to search at most four bytes backwards to find an easily recognizable initial byte. For example:

  ```
  isInitialByte = ((byte & 0xC0) != 0x80);
  ```

- UTF-8 is reasonably compact in terms of the number of bytes used for encoding.

The actual encoding is this:

- For multibyte encodings, the first byte sets `1` in a number of high-order bits equal to the number of bytes used in the encoding; the bit after that is set to `0`. For example, a 2-byte sequence always starts with `110` in the first byte.
- For all subsequent bytes in a multibyte encoding, the first two bits are `10`. The value of a trailing byte in a multibyte encoding is always greater than or equal to `0x80`.

  The following table shows the binary form of each byte of the encoding and the minimum and maximum values for the characters represented by 1-, 2-, 3-, and 4-byte encodings:

Length	First byte	Following bytes	Min. value	Max. value
Single byte	0*XXXXXXX*	N/A	`0x0000`	`0x007F`
Two bytes	110*XXXXX*	10*XXXXXX*	`0x0080`	`0x07FF`
Three bytes	1110*XXXX*	10*XXXXXX*	`0x0800`	`0xFFFF`
Four bytes	11110*XXX*	10*XXXXXX*	`0x10000`	`0x10FFFF`

- The actual content of the multibyte encoding (i.e. the wide-character encoding) is the catenation of the *XX* bits in the encoding. A 2-byte encoding of **`11011111 10000000`** encodes the wide character **`11111000000`**.
- Where there's more than one way to encode a value (such as 0), the shortest is the only legal value. The null character is always a single byte.

Conversion functions

In our C libraries, "wide characters" are assumed to be Unicode, and "multibyte" is UTF-8 in the default locale. The **`wchar_t`** type is defined as an unsigned 32-bit type, and *wctomb()* and *mbtowc()* implement the UTF-8 encoding in the default locale.

Multibyte characters in the C library are UTF-8 in the default locale; in different locales, multibyte characters might use a different encoding.

You can use the following functions (described in the QNX Neutrino *Library Reference*) for converting between wide-character and multibyte encodings:

mblen()	Compute the length of a multibyte string in characters
mbtowc()	Convert a multibyte character to a wide character
mbstowcs()	Convert a multibyte string to a wide-character string
wctomb()	Convert a wide character to its multibyte representation
wcstombs()	Convert a wide-character string to a multibyte string

Photon libraries use multibyte UTF-8 character strings: any function that handles strings should be able to handle a valid UTF-8 string, and functions that return a string can return a multibyte-character string. This also applies to widget resources. The graphics drivers and font server assume that all strings use UTF-8.

The main Photon library, **`ph`**, provides the following non-ANSI functions (described in the Photon *Library Reference*) for working with multibyte UTF-8 and wide characters:

utf8len()	Count the bytes in a UTF-8 character
utf8strblen()	Find the number of UTF-8 characters in part of a string

utf8strchr()	Search for a UTF-8 character in a string
utf8strichr()	Search for a UTF-8 character in a string, ignoring case
utf8strirchr()	Search backwards for a UTF-8 character in a string, ignoring case
utf8strlen()	Find the length of a UTF-8-character string
utf8strnchr()	Search for a UTF-8 character in part of a string
utf8strncmp()	Compare part of a UTF-8-character string
utf8strndup()	Create a copy of part of a UTF-8-character string
utf8strnichr()	Search for a UTF-8 character in part of a string, ignoring case
utf8strnlen()	Find the number of bytes used by a UTF-8-character string
utf8strrchr()	Search backwards for a UTF-8 character in a string
utf8towc()	Convert a UTF-8 character to a wide-character code
wctolower()	Return the lowercase equivalent of a wide character
wctoutf8()	Convert a wide-character code into a UTF-8 character

These functions are defined in **`<utf8.h>`** (notice it isn't **`<photon/utf8.h>`**), and use UTF-8 encodings no matter what the current locale is. UTF8_LEN_MAX is defined to be the maximum number of bytes in a UTF-8 character.

Other encodings

If your application needs to work with other character encodings, you'll need to convert to and from UTF-8. Character sets are defined in the file **`/usr/photon/translations/charsets`**, and include:

- Big5 (Chinese)
- Cyrillic (KOI8-R)
- Japanese (EUC)
- Japanese (Shift-JIS)
- Korean (EUC)
- Western (ISO 8859-1)

The following translation functions are provided, and are described in the Photon *Library Reference*:

PxTranslateFromUTF()	Translate characters from UTF-8

PxTranslateList()	Create a list of all supported character translations
PxTranslateSet()	Install a new character-set translation
PxTranslateStateFromUTF()	Translate characters from UTF-8, using an internal state buffer
PxTranslateStateToUTF()	Translate characters to UTF-8, using an internal state buffer
PxTranslateToUTF()	Translate characters to UTF-8
PxTranslateUnknown()	Control how unknown encodings are handled

These functions are supplied only in static form in the Photon library **`phexlib`**. The prototypes are in **`<photon/PxProto.h>`**.

In short, Photon supports any Unicode encoded TrueType font. However, Photon does not support complex languages such as Hebrew or Arabic. In order to provide support for complex languages, you must obtain a third-party font rendering engine.

Keyboard drivers

The keyboard driver is table-driven; it handles any keyboard with 127 or fewer physical keys.

A keypress is stored in a structure of type **`PhKeyEvent_t`** (described in the Photon *Library Reference*).

Example: text widgets

The text widgets use the *key_sym* field for displayable characters. These widgets also check it to detect cursor movement. For example, if the content of the field is Pk_Left, the cursor is moved left. The *key_sym* is Pk_Left for both the left cursor key and the numeric keypad left cursor key (assuming NumLock is off).

Dead keys and compose sequences

QNX Neutrino supports “dead” keys and “compose” key sequences to generate *key_syms* that aren’t on the keyboard. The *key_sym* field is valid only on a key *press* — not on a key release — to ensure that you get only one symbol, not two.

For example, if the keyboard has a dead accent key (for example, ‘) and the user presses it followed by e, the *key_sym* is an “e” with a grave accent (è). If the e key isn’t released, and then another group of keys (or more compose or dead key sequences) are pressed, the *key_syms* would have to be stacked for the final releases.

If an invalid key is pressed during a compose sequence, the keyboard drivers generate *key_sym*s for all the intermediate keys, but not an actual press or release.
For a list of compose sequences, see below.

Photon compose sequences

Photon comes equipped with standard compose sequences. If your keyboard doesn't include a character from the standard ASCII table, you can generate the character using a compose sequence. For example, ó can be generated by pressing the Alt key, followed by the ' key, followed by the o key.

These aren't keychords; press and release each key one after the other.

The following keys can be used for generating accented letters:

Key	Accent	Example sequence	Result
'	acute	Alt ' o	ó
,	cedilla	Alt , c	ç
ˆ	circumflex	Alt ˆ o	ô
>	circumflex	Alt > o	ô
"	diaeresis	Alt " o	ö
'	grave	Alt ' o	ò
/	slash	Alt / o	ø
˜	tilde	Alt ˜ n	ñ

If your keyboard doesn't have the following symbols, you can create them by pressing the Alt key, followed by the first key in the sequence, followed by the second key in the sequence.

Symbol	Description	Unicode value	Sequence
æ	small letter ae (ligature)	E6	Alt e a
Æ	capital letter ae (ligature)	C6	Alt E A
Ð	capital letter eth	D0	Alt D -
ð	small letter eth	F0	Alt d -
ß	small letter sharp s (German scharfes s)	DF	Alt s s

continued...

Symbol	Description	Unicode value	Sequence
μ	micro sign	B5	Alt / U
			Alt / u
þ	small letter thorn	FE	Alt h t
Þ	capital letter thorn	DE	Alt H T
#	number sign	23	Alt + +
@	commercial at	40	Alt A A
©	copyright sign	A9	Alt C 0
			Alt C O
			Alt C o
			Alt c 0
			Alt c O
			Alt c o
®	registered trademark sign	AE	Alt R O
[	left square bracket	5B	Alt ((
]	right square bracket	5D	Alt))
{	left curly bracket	7B	Alt (-
}	right curly bracket	7D	Alt) -
»	right-pointing double angle quotation mark	BB	Alt > >
«	left-pointing double angle quotation mark	AB	Alt < <
ˆ	circumflex accent	5E	Alt > space
'	apostrophe	27	Alt ' space
`	grave accent	60	Alt ' space
|	vertical bar	7C	Alt / ˆ
			Alt V L
			Alt v l
\	reverse solidus (backslash)	5C	Alt / /
			Alt / <
˜	tilde	7E	Alt - space

continued...

Symbol	Description	Unicode value	Sequence
	no-break space	A0	Alt space space
°	degree sign	B0	Alt 0 ˆ
¡	inverted exclamation mark	A1	Alt ! !
¿	inverted question mark	BF	Alt ? ?
¢	cent sign	A2	Alt C /
			Alt C |
			Alt c /
			Alt c |
£	pound sign	A3	Alt L -
			Alt L =
			Alt l -
			Alt l =
¤	currency sign	A4	Alt X 0
			Alt X O
			Alt X o
			Alt x 0
			Alt x O
			Alt x o
¥	yen sign	A5	Alt Y -
			Alt Y =
			Alt y -
			Alt y =
¦	broken (vertical) bar	A6	Alt ! ˆ
			Alt V B
			Alt v b
			Alt | |
§	section sign	A7	Alt S !
			Alt S 0
			Alt S O

continued…

Symbol	Description	Unicode value	Sequence
			Alt s !
			Alt s 0
			Alt s o
¨	diaeresis or umlaut	A8	Alt " "
·	middle dot	B7	Alt . .
			Alt . ˆ
¸	cedilla	B8	Alt , space
			Alt , ,
¬	not sign	AC	Alt - ,
	soft hyphen	AD	Alt - -
¯	macron	AF	Alt - ˆ
			Alt _ ˆ
			Alt _ _
±	plus-minus sign	B1	Alt + -
¹	superscript one	B9	Alt 1 ˆ
			Alt S 1
			Alt s 1
²	superscript two	B2	Alt 2 ˆ
			Alt S 2
			Alt s 2
³	superscript three	B3	Alt 3 ˆ
			Alt S 3
			Alt s 3
¶	pilcrow sign (paragraph sign)	B6	Alt P !
			Alt p !
ª	feminine ordinal indicator	AA	Alt A _
			Alt a _
º	masculine ordinal indicator	BA	Alt O _
			Alt o _

continued...

Symbol	Description	Unicode value	Sequence
$\frac{1}{4}$	vulgar fraction one quarter	BC	Alt 1 4
$\frac{1}{2}$	vulgar fraction one half	BD	Alt 1 2
$\frac{3}{4}$	vulgar fraction three quarters	BE	Alt 3 4
$\div$	division sign	F7	Alt - :
$\times$	multiplication sign	D7	Alt x x

Appendix D
Photon in Embedded Systems

In this appendix. . .

Assumptions

This appendix makes the following assumptions:

- You understand the process of building an embedded system for QNX Neutrino. For more information, see *Building Embedded Systems*.
- You have your target hardware booting into Neutrino, and can run a shell and commands such as **`pidin`**.
- You know what graphics hardware you will be using, and its parameters (such as the Vendor and Device IDs for a PCI graphics card).
- You'll be using a QNX Neutrino development system to build your embedded Photon target from the command line, or using the System Builder in the IDE.

Introduction

The Photon microGUI is an embedded Graphical User Interface (GUI). This GUI is made up of numerous processes that use Neutrino message passing in order to create a highly responsive user experience. Photon is made up of these main components:

- QNX Graphics Framework server (**`io-display`**)
- Photon server (the graphical kernel **`photon`**)
- graphics subsystem manager and hardware driver (**`io-graphics`** and associated graphics driver)
- font support (**`phfont.so`** and plugin DLLs)
- input support (a **`devi-*`** input driver)
- user applications.

QNX Graphics framework server

The **`io-display`** manager provides support for direct rendering to graphics devices using the QNX Graphics Framework and OpenGL ES.

Photon Server

The **`Photon`** server is the core server process for the GUI. This process must be the first graphical process run in the system. It is responsible for handling region creation and destruction, clipping, and managing the Photon event space.

Graphics subsystem

This process, **`io-graphics`**, handles the Photon draw stream and loads the hardware driver. This process runs before any user application processes. The graphics subsystem queries the QNX graphics framework (**`io-display`**) for the display's settings rather than its own command-line settings.

Font support

This process (**`phfont.so`**) and associated libraries are used to render and gather metrics about fonts. Photon can render the following types of fonts:

- Adobe Type 1 (**`.pfa`**)
- Adobe Type 2 (**`.cff`**)
- Bitstream Speedo — public encryption only (**`.spd`**)
- Bitstream Stroke (**`.ffs`**)
- Bitstream T2K (**`.t2k`**)
- Bitstream TrueDoc (**`.pfr`**)
- Photon bitmap (**`.phf`**)
- TrueType (**`.ttf`**)
- TrueType collections (**`.ttc`**)

Photon can render any Unicode encoded TrueType font. However, Photon does not provide support for complex languages such as Hebrew or Arabic. In order to render fonts for complex languages, you must obtain a third-party font rendering engine. For more information about supported fonts, see Appendix: Unicode Multilingual Support.

Input support

This process (**`devi-*`**) is responsible for handling user input from a mouse, keyboard, or touchscreen. This process communicates with your input hardware and then emits Photon events, which are collected and delivered to graphical processes in the system.

User applications

Once all of the other processes are running you can start user applications.

Steps to boot into Photon

Here's an outline of the steps required to start Photon yourself, in the context of an embedded, closed system:

1. Start the graphics framework server.
2. Export (at a minimum) the **PHOTON_PATH** environment variable.
3. Start the Photon server.
4. Configure your fonts.
5. Start the graphics driver.
6. Start the input driver.

7. Start the window manager, if required.
8. Start your application(s).

Each of these steps requires certain files be installed in your target system. By predetermining exactly what graphics hardware you have and what fonts your application needs, you can keep the number of files (and size of the image) to an absolute minimum. This reduction in size may in turn have a positive impact on your system's startup time.

We'll go through all the steps in detail and discuss the files needed for each step. At the end of this process, you should know exactly what Photon files you'll need to run your embedded application.

The basics

Step 1. Start the QNX graphics framework server

The QNX graphics framework server must be loaded before the graphics driver.

For more information about the QNX graphics framework server refer to **io-display**.

Step 2. Export environment variables

The **PHOTON_PATH** environment variable points to the base directory of the Photon installation. By default, this directory is **/usr/photon**. This location is expected to hold at least the following subdirectories:

font_repository	Photon font files and configuration files used by the font server (platform-independent).
palette	graphics palettes (platform-independent). These palettes are required only when you're running your graphics driver(s) with a color depth of 8 bits.
translations	Photon language translations (platform-independent) These files are required only if your application(s) handles non-UTF8 character encodings via the *PxTranslate*()* API.

You should set the **PHOTON_PATH** environment variable in the buildfile where you set other environment variables such as **PATH**:

```
PHOTON_PATH=/usr/photon
```

The **LD_LIBRARY_PATH** points the default system search path for libraries. The **procnto** process uses its setting of **LD_LIBRARY_PATH** to initialize the privileged configuration string **_CS_LIBPATH**, which limits the paths that programs running as **root** can load libraries from.

CAUTION: To avoid problems starting `phfont`, **LD_LIBRARY_PATH** must be set properly on the procnto line. This is especially true when using the IDE. **LD_LIBRARY_PATH** needs to be set on the procnto line in the project properties.

The **PATH** environment variable points to the default system search path for binaries. You should set it to include the directories in your build image that contain binaries. These settings apply to any boot image.

Step 3. Start the Photon server

If you don't need to pass any command-line arguments to the Photon server, you can start it as follows:

```
Photon
```

If you start Photon as a background process (that is, with the ampersand `&` after the command) you can tell that Photon started correctly by checking that `/dev/photon` appears in the filesystem. Use `waitfor /dev/photon` in your buildfile to check that the directory exists.

If your boot image is too large because you've included Photon or other executables, you can place them in another filesystem that you can mount at boot-time. For more information, see `mkifs` in the QNX Neutrino *Utilities Reference*.

If you do include any of the Photon executables in your boot image, you should also include `/usr/photon/bin` in **MKIFS_PATH** so `mkifs` can find them.

Files needed

`/usr/photon/bin/Photon`

Step 4. Configure fonts

If you're working on a board that has network access and can mount a network filesystem on your host machine, we recommend that you mount ${QNX_TARGET}`/usr/photon/font_repository` as `/usr/photon/font_repository` via NFS or CIFS. Although this approach uses the full Photon font system, it simplifies development significantly, and you can customize the embedded fonts later using the information in "Configuring Fonts" in the "Advanced topics" section.

For information about using NFS and CIFS to mount a network filesystem, see "CIFS filesystem" and "NFS filesystem" in the Working with Filesystems chapter of the Neutrino *User's Guide*.

Include the following libraries in your build file; `io-graphics` will load the font libraries for you when it starts:

`/lib/dll/phfont.so`
: Font manager plugin.

/lib/libfont.so

Font manager API library.

/lib/dll/font/ttfFFcore.so, **/lib/dll/font/FCcore.so**, and **/lib/dll/font/PHFcore.so**

Rendering plugins.

These plugins are used to render specific fonts on the system. Use the **use** utility to view specific support information for these plugins, and to determine which font families are rendered by this plugin. If you don't plan to use any of the fonts that this plugin supports, then you can remove this plugin to decrease the footprint of your installation.

/usr/lib/libblkcache.so

Disk block cache library, used by **ttfFFcore.so**, **FCcore.so**, and **PHFcore.so**.

/usr/lib/libFF-T2K.so

Bitstream FontFusion rendering library, used by **ttfFFcore.so** and **FCcore.so**.

/usr/lib/libFF-T2K-fm.so

Bitstream FontFusion font management library for font collections (**.pfr** and **ttc**), used by **FCcore.so**. This library has been deprecated. Use **libFF-T2K.so** instead.

/usr/lib/libFF-T2K-cache.so

Bitstream FontFusion font cache management library, used by **FCcore.so** and **ttfFFcore.so**. This library has been deprecated. Use **libFF-T2K.so** instead.

/usr/lib/libfontutils.so

Step 5. Start the graphics driver

The graphics subsystem consists of **io-graphics**, a hardware-specific driver DLL, and a collection of helper libraries. You need the following components to run the graphics subsystem on the target:

/usr/photon/bin/io-graphics

Graphics subsystem executable.

/usr/lib/libphrender.so

Photon rendering routines.

/lib/libfont.so

Font manipulation library (also required by Photon applications).

`/lib/dll/phfont.so`
: Font server plugin.

`/usr/photon/palette/`*file*
: A Photon palette file for the target display.

`/usr/lib/libgf.so`
: Advanced graphics library.

`/usr/lib/libph.so`
: Photon graphics library.

`/usr/lib/libdisputil.so`
: Display utilities library.

Additionally, you need a hardware-specific library (or graphics driver). By convention, graphics driver names begin with **`devg-`**, for example, **`devg-rage.so`**.

Most graphics drivers depend on the following shared libraries:

`/usr/lib/libffb.so.2`
: Software fallback routines for graphics drivers.

`/usr/lib/libdisputil.so.2`
: Miscellaneous utilities for graphics drivers.

Make sure that all required libraries are accessible by the dynamic loader before you start **`io-graphics`**. Use the **LD_LIBRARY_PATH** environment variable or **_CS_LIBPATH** configuration string to point to the location of the shared libraries.

Step 5. Start the input driver

Normally in a desktop environment, you use the **`inputtrap`** utility to automatically generate the correct command line and to invoke the appropriate **`devi-*`** driver. For example, it might invoke **`devi-hirun`** like this:

```
devi-hirun kbd fd -d/dev/kbd msoft fd &
```

See **`devi-hirun`** in the Neutrino *Utilities Reference* for more examples.

You typically run **`inputtrap`** because you don't know in advance what the appropriate command line should be.

In an embedded system, however, you typically specify the command line to the **`devi-*`** driver manually. This is because the input devices are often found at unusual locations, are incapable of PnP identification, or are simply not supported by an existing **`devi-*`** driver. In addition, the **`inputtrap`** utility tends to be quite large and could waste precious storage and memory in a constrained environment. If figuring out the appropriate command to run proves difficult, you can temporarily install

inputtrap in your image (or mount a networked filesystem that contains the binary) and use it to generate the correct command line. See **inputtrap** and its **query** option in the Neutrino *Utilities Reference*.

If none of the shipped input drivers are able to work with your input hardware, you can customize the input drivers by using the Input Driver Development Kit (Input DDK). For example, you can change the size of the memory footprint, or you can create a custom module to support new devices.

Files needed

The appropriate **devi-*** driver in **/usr/photon/bin**

The appropriate **.kbd** keyboard mapping file in **/usr/photon/keyboard**

Step 6. Start the window manager

The Photon window manager (**pwm**) is an optional component that provides your system with windowing functionality you may not need. If your application user interface consists of one (or more than one) program that always fills the screen, uses a “card” or “slide” paradigm (that is, a UI composed of a series of stacked cards or slides that the program flips through), or uses dialogs that your application controls itself, then you probably don’t require the window manager. On the other hand, if your UI is built using one (or more than one) program that relies on windowing behavior (such as windows or dialogs that you don’t want to manage yourself), then **pwm** is probably a good fit for your system.

Files needed

/usr/photon/bin/pwm

Step 7. Start your application

If your application is a single executable and doesn’t require the window manager, you may link statically against the Photon-related libraries (such as **libAp.a**, **libph.a**, and **libphexlib.a**). Linking statically avoids the need to include the corresponding shared components in your image, and will pull in only the symbols needed by your program, making the overall image smaller. Also, linking statically has an added benefit of slightly reducing runtime overhead. If you have multiple applications in your image (including **pwm**), you should always link against the shared libraries and include them in your image.

You can use the **pidin** utility on a host system to view the libraries that an application or OS component requires. For example, if you wanted to see the libraries required by **phcalc**, run it on a host system, and then run **pidin -p phcalc mem**.

The QNX IDE includes a tool called the Dietician that shrinks shared libraries by analyzing the executables in your system and removing the symbols that aren’t needed. This realizes most of the benefits of linking statically while still allowing the libraries to be shared. However, if your system only consists of one application (and no window manager), linking statically is probably the better way to go.

Files needed

- Your application files
- If linking shared, you require `/usr/lib/libph.so`
- If you're using executables created in PhAB, you need `/usr/lib/libAp.so`.
- You also may need `/libphexlib.so` if you load images or do language translation.

The libraries in `/usr/photon/lib (*.so.1)` are provided for runtime compatibility with Photon for QNX Neutrino 6.0 (x86 only). The libraries for QNX Neutrino 6.1 and later are located in `/usr/lib`.

Caveats

The following are observations that some customers have encountered when moving Photon to an embedded system.

Flash filesystems

The following flash filesystem properties affect how you configure Photon:

Compression and Speed

PhAB executables, by default, have their resources bound into the executable file at the end of the binary data. Since the flash filesystem is slower when it's seeking in a compressed file, you'll probably want to keep the resource records in a separate file, instead of including them at the end of the binary. To do this, change the **makefile** so that the resources are bound to a separate file. For example, change the following dependency:

```
$(ABOBJ) $(MYOBJ)
$(LD) $(LDFLAGS) $(ABOBJ) $(MYOBJ) -M -o mine
usemsg mine ../Usemsg
phabbind mine $(ABMOD)
```

to:

```
$(ABOBJ) $(MYOBJ)
$(LD) $(LDFLAGS) $(ABOBJ) $(MYOBJ) -M -o mine
usemsg mine ../Usemsg
phabbind mine.res $(ABMOD)
```

When your executable is launched, the PhAB library (`libAp.so`) automatically finds the resource file, provided the following criteria are met:

1. The resource file has the same basename as the binary, with the extension `.res`
2. The resource file is in the same directory as the binary.

If you want to group your resource files in a separate directory, you can. Place them in the directory specified by the exported **AB_RESOVRD** environment variable, and the PhAB library will look for them there. The naming of the resource files must meet the first criterion, listed above.

Graphics

Many embedded systems lack components that are typical on an x86 desktop machine, such as BIOS ROMs. Because many of the modeswitchers that Photon supports require a video BIOS to allow them to switch graphics modes, you might need a BIOS on the board. Check with us to see if a non-BIOS version is available.

Miscellaneous

Here are some other considerations:

CPU Speed
: For some embedded systems, the CPU performance will be slower than the desktop. You'll want to consider this when you design your Photon applications for the embedded environment.

Scrolling
: If the scrolling area pages down more than one page at a time when you click in the trough, try increasing the value of the mouse repeat delay in Photon. For example:

```
Photon -D1000 &
```

Input
: You can set the throttling parameters on both the Input and the Photon Server. By reducing the speed at which mouse events are emitted, you can reduce the traffic through the Photon system. On slower 486 platforms, it's common practice to lower the throttling on input to 10 or 20 ms.

Phindows and Phditto
: If your target application needs to support remote diagnostics from either Phindows or **phditto**, you'll also need to install **phrelay**, a render library, and the services configuration file.

Example

Let's look at the steps involved in embedding Photon for use in an embedded system by creating a simple buildfile that contains a few simple Photon applications.

Our goal is to build a Photon system with the following minimal capabilities that satisfies our system's requirements:

- scalable TrueType fonts — the smallest set available required for your applications
- the minimum files needed to run the graphics driver
- input support for a mouse and keyboard
- a window manager to handle multiple Photon applications.

Note that a window manager isn't strictly required for an embedded system, but we'll include one to make our example easier to use.

We'll follow these steps:

- Analyzing required binaries
- Analyzing required libraries (**.so**)
- Analyzing required fonts
- Putting it all together
- Troubleshooting

Required binaries

The first step involves figuring out all the binaries required to run Photon. You can see everything that's running on a full system. Run Photon on your PC, and look at the output of the **pidin arg** command.

From that list, you need only a few of the programs:

- **Photon** — the process that implements the windowing kernel
- **phfont** and **phfont.so** — font server
- **io-graphics** — graphics rendering subsystem
- **pwm** — provides window management
- **devi-hirun** — mouse/touchscreen and keyboard driver; see "Input drivers (**devi-***)" in the summary of the QNX Neutrino *Utilities Reference*.

Save the argument list for your system in a file. We'll need that output later.

Required libraries

On this embedded system you want only the components listed above, plus you'll run a couple of simple applications:

- **phcalc** — the calculator
- **pterm** — a terminal application

Run the applications, then look at the output of the **pidin mem** command. The resulting listing tells you every library that you need to make available to the embedded system. For a graphics driver, you'll use the generic SVGA driver (**devg-svga.so**).

So you need the following libraries (at least):

- **ldqnx.so.2** (**ldqnx.so.3** for MIPS) — an alias for **libc.so**
- **libph.so.3**

- **libphexlib.so.3**
- **libphrender.so.2**
- **libffb.so.2**
- **libdisputil.so.2**
- **libAp.so.3**
- **libm.so.2**
- **devg-svga.so**

Required fonts

Now let’s look at fonts. Sometimes an application expects a specific font, and codes directly to that font. If this is the case, you need to explicitly include every font that your application needs. If you standardize on a certain family/style of fonts or if you don’t care what exact font you have (as long as the size is okay), then you can cut down on the number of fonts and use one font to replace several other families of fonts. For example, you can use Times as a replacement for Helvetica and Courier.

In this example, because you’re using a few simple applications, and because you’re trying to create the smallest image possible, you need only two fonts: a monospace and regular TrueType version of Prima Sans.

Now’s a good time to create a play area on your system to begin testing the embedded system, and collecting required files.

Create a subdirectory called **phembed** in your home directory (or whichever directory you wish to keep your source files). Within that directory, create these subdirectories:

- **phembed/bin**
- **phembed/lib**
- **phembed/font_repository**

Now back to the fonts. In this example, you want to use the **primasansmonobts** TrueType font for everything. You’ll also want to use a mouse, so you’ll include the **phcursor.phf** file.

Here are the files you need:

- **fontdir** (automatically generated by the **mkfontdir** utility)
- **fontmap**
- **fontext**
- **fontopts**
- **phcursor.phf**

- **tt2009m_.ttf**

Copy these files (except **fontdir**) from **/usr/photon/font_repository** to **/phembed/font_repository**, then change directories to **/phembed/font_repository**.

You need to modify the **fontmap** and **fontopts** files to reflect the fonts, options and mappings you want for your embedded system. You can edit these files by hand (see **phfont** for more information on the structure of these files). In our case lets make sure that the **fontmap** file contains:

```
? = primasansmonobts
```

This ensures that all unknown fonts will be replaced with the primasansmonobts font, provided in the **tt2009m_.ttf** file.

To generate **fontdir**, use the **mkfontdir** like this:

```
mkfontdir -d /phembed/font_repository
```

Make sure that the **LD_LIBRARY_PATH** string in your build file contains the string **/lib/dll**.

Putting it all together

Now let's put all the pieces you need in place and create a buildfile for your embedded Photon system. Run **mkifs** to create an image.

- For a sample buildfile that includes more Photon components, such as the background manager **bkgmgr**, see Getting Photon on your board in the Working with a BSP chapter of the *Building Embedded Systems* guide.
- In a real buildfile, you can't use a backslash (\) to break a long line into shorter pieces, but we've done that here, just to make the buildfile easier to read.

```
[image=0x88010000]
[virtual=shle/binary +compress] .bootstrap = {
    startup-sdk7785 -Dscif..115200.1843200.16 -f600000000 -v

    [+keeplinked] PATH=/proc/boot:/bin:/sbin:/usr/bin:/opt/bin: \
    /usr/sbin:/usr/photon/bin \
     LD_LIBRARY_PATH=/proc/boot:/lib::/usr/lib:/lib/dll:/opt/lib: \
     /usr/photon/lib:/usr/photon/dll \
     PHOTON_PATH=/usr/photon procnto -v
}

[+script] .script = {
    procmgr_symlink ../../proc/boot/libc.so.3 /usr/lib/ldqnx.so.2

    display_msg Welcome to QNX Neutrino 6.4.0 on the Renesas SDK7785
    #################################################################
    ## SERIAL driver
    #################################################################
    display_msg Starting serial driver
    devc-sersci -e -F -x -b115200 -c1843200/16 scif1 &
    waitfor /dev/ser1
```

```
reopen /dev/ser1

slogger
pipe

###############################################################
## NOR FLASH driver SDK 7785
###############################################################
# Flash only recognized as a single bank 4*2 array.  Single driver
# Used to access both banks.  Care must be taken when toggling
# bank select as the base address will switch between banks.
###############################################################
# devf-edosk7780 -s0x0,128M

###############################################################
## NETWORK driver SDK 7785
###############################################################
display_msg "Starting the core network stack..."
io-pkt-v4 -dsmc9118 ioport=0x15800000,irq=6 -ptcpip
waitfor /dev/socket 15
waitfor /dev/io-net/en0

if_up -r 10 -p en0
display_msg "Setting ip address to XXX.XX.X.XX..."
ifconfig en0 XXX.XX.X.XX up netmask 0xffffff00
if_up -a en0 lo0


display_msg Starting fs-nfs2


setconf DOMAIN domain.name.com


###############################################################
## PCI server
###############################################################
display_msg "Starting pci-edosk7780..."
pci-edosk7780

display_msg "Starting devc-pty..."
devc-pty
waitfor /dev/ptyp0 4
waitfor /dev/socket 4
qconn port=8000

display_msg "Setting enviroment variables..."
SYSNAME=nto
TERM=qansi
HOSTNAME=gsdk7785
HOME=/root
PATH=:/proc/boot:/bin:/sbin:/opt/bin:/usr/sbin:/usr/bin:/usr/photon/bin
LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib:/usr/photon/ \
lib:/usr/photon/dll
PHOTON=/dev/photon
PHOTON_PATH=/usr/photon
PHOTON_PATH2=/usr/photon
PHFONT=/dev/phfont
MMEDIA_MIDI_CFG=/etc/config/media/midi.cfg

display_msg "Starting io-display..."

io-display -dvid=0x10cf,did=0x201e
waitfor /dev/io-display

display_msg "Starting Photon..."
Photon &
waitfor /dev/photon 10

display_msg "Starting io-graphics..."
io-graphics &
waitfor /dev/phfont 10

display_msg "Starting Window Manager..."
pwm &
```

```
    devc-pty &

    display_msg "Starting Terminal"
    pterm /proc/boot/ksh &
#   inetd &

    [+session] ksh &
}

[type=link] /bin/sh=/proc/boot/ksh
[type=link] /dev/console=/dev/ser1
[type=link] /tmp=/dev/shmem

libc.so
libc.so.2
libm.so

###################################################################
## uncomment for NETWORK driver
###################################################################
devn-smc9118.so
devnp-shim.so
libsocket.so



[data=c]
devc-sersci
setconf
###################################################################
## uncomment for NOR FLASH driver
###################################################################
 devf-edosk7780
 flashctl

###################################################################
## uncomment for PCI server
###################################################################
pci-edosk7780
pci
pipe

###################################################################
## uncomment for NETWORK driver
###################################################################
io-pkt-v4
ping
cat
ifconfig
netstat
nicinfo
sleep

###################################################################
## uncomment for REMOTE_DEBUG (gdb or Momentics)
###################################################################
devc-pty
qconn
/usr/bin/pdebug=pdebug

###################################################################
## general commands
###################################################################
ls
ksh
pipe
pidin
uname
slogger
sloginfo
slay
fs-nfs3
fs-nfs2
if_up
```

```
fs-cifs
mount
umount


###########################################################################
## uncomment for GF io-display
###########################################################################
/sbin/io-display=io-display

/etc/system/config/display.conf=/usr/qnx640/target/qnx6/etc/system/config/ \
display.conf
/etc/system/config/img.conf=/usr/qnx640/target/qnx6/etc/system/config/ \
img.conf


###########################################################################
## uncomment for GF libraries
###########################################################################
/lib/dll/devg-soft3d.so=devg-soft3d.so
/lib/dll/devg-coral.so=devg-coral.so
/lib/libFF-T2K.a=libFF-T2K.a
/lib/libFF-T2K.so.2=libFF-T2K.so.2
/lib/libimg.so=libimg.so

/usr/lib/libGLES_CM.so.1=libGLES_CM.so.1
/usr/lib/libffb.so.2=libffb.so.2
/usr/lib/libgf.so=libgf.so

###########################################################################
## uncomment for GF image support
###########################################################################
/lib/dll/img_codec_bmp.so=img_codec_bmp.so
/lib/dll/img_codec_gif.so=img_codec_gif.so
/lib/dll/img_codec_jpg.so=img_codec_jpg.so
/lib/dll/img_codec_png.so=img_codec_png.so
/lib/dll/img_codec_sgi.so=img_codec_sgi.so
/lib/dll/img_codec_tga.so=img_codec_tga.so


###########################################################################
## uncomment for GF binaries
###########################################################################
/bin/egl-gears=egl-gears
/bin/vsync=vsync

/lib/dll/font/ttfFFcore.so = ${QNX_TARGET}/shle/lib/dll/font/ttfFFcore.so
/lib/dll/font/PHFcore.so = ${QNX_TARGET}/shle/lib/dll/font/PHFcore.so
/lib/dll/font/FCcore.so = ${QNX_TARGET}/shle/lib/dll/font/FCcore.so
libfontutils.so
libblkcache.so
libFF-T2K.so
libfont.so
phfont.so

#######################################
## Photon LIbs
#######################################
Photon
[+raw] /usr/photon/bin/pterm = pterm
[+raw] /usr/photon/bin/phcalc = phcalc

io-graphics
pwm
libph.so
libAp.so
libphexlib.so
libdisputil.so
libffb.so
libphrender.so


##########################################################################
## font config
##########################################################################
```

```
/usr/photon/font_repository/tt2009m_.ttf = \
/usr/photon/font_repository/tt2009m_.ttf
/usr/photon/font_repository/phcursor.phf = \
/usr/photon/font_repository/phcursor.phf
/usr/photon/font_repository/pcterm12.phf = \
/usr/photon/font_repository/pcterm12.phf
/usr/photon/font_repository/fontopts = /usr/photon/font_repository/fontopts
/usr/photon/config/wm/wm.menu = /usr/photon/config/wm/wm.menu

/usr/photon/font_repository/fontdir = {
;
; fontdir config file
;
pcterm12,.phf,PC Terminal,12,,0000-00FF,Nf,6x12,13K
phcursor,.phf,Photon Cursor,0,,E900-E921,Np,32x32,3K
primasansmonobts,0@tt2009m_.ttf,PrimaSansMono BT,0,,0020-F002,f,79x170,109K
}

/usr/photon/font_repository/fontext = {
;
; fontext config file
;
+normal = primasansmonobts, phcursor
}

/usr/photon/font_repository/fontmap = {
;
; fontmap config file
;
? = primasansmonobts
}


/usr/photon/config/coral.conf=${QNX_TARGET}/usr/photon/config/coral.conf
/etc/system/config/crtc-settings=/etc/system/config/crtc-settings
/usr/photon/palette/default.pal=/usr/photon/palette/default.pal

# allow pterm to save its configuration to RAM, if the user changes it.
[type=link] /.ph/pterm = /dev/shmem
[type=link] /.ph/wm/wm.cfg = /dev/shmem
```

Note the following about the buildfile:

- You set up the environment variables **PATH**, **LD_LIBRARY_PATH**, and **PHOTON_PATH**. Passing **LD_LIBRARY_PATH** to **procnto** sets the privileged configuration string **_CS_LIBPATH**.
- You link **libc.so.3** and **ldqnx.so.2** (**ldqnx.so.3** for MIPS), because they're the same library.
- You invoke Photon, then wait for **/dev/photon** to indicate that the Photon server is running.
- You use **inputtrap** to detect input devices and configure **devi-hirun**. This makes the buildfile compatible with most input devices. For a smaller boot image, determine the correct arguments for the input driver, and start it directly.
- After the system starts **io-graphics**, you check that the font server is running correctly (**waitfor /dev/phfont**).
- You specify where libraries and binaries should be put on the target (by default they're put in **/proc/boot**).

- Some libraries are from the customized library directory in the **`phembed`** directory (**`./lib`**), while others are from the host system (such as **`/usr/lib/libdisputil.so.2`**).
- You use the **`[+raw]`** directive for the PhAB applications so that **`mkifs`** doesn't remove necessary resource information from these files.

Once you've built your image using **`mkifs`**, you can transfer it to a test machine to see how it works. See "Transferring an OS image onto your board" in the Working with a BSP chapter of *Building Embedded Systems* for more information.

Troubleshooting

1 When I start **`io-graphics`**, it seems to be running, but nothing appears on the screen.

Check the system log; **`io-graphics`** may have sent error messages to the system logger, **`slogger`**. In order to debug the problem, make sure **`slogger`** is running before starting the graphics driver. Use **`sloginfo`** to display the system log messages.

2 When I start an application, it exits with the message **`Ap: Unable to locate Photon`**.

Make sure both the Photon server and the font manager are running. You can determine if they're running by making sure that **`/dev/photon`** and **`/dev/phfont`** exist.

3 When I start an application, it exits with the message **`Ap: Unable to open resource file`**.

If you include an application that was built by PhAB in an image created by **`mkifs`**, some information will be stripped out, since **`mkifs`** does a very aggressive binary strip. You can avoid this by using the **`+raw`** attribute; see the **`mkifs`** documentation for more information. Since setting the attribute will cause the application not to be stripped, you may want to use the **`strip`** utility to manually strip the binary before building the image, to reduce the image size.

Example: Using the IDE's System Builder

Building an embedded OS image that includes Photon using the IDE is similar to writing a buildfile and using the command line. You need to perform the same analysis (described in the example above) of the required binaries, libraries and fonts. The difference is that you don't write a buildfile; instead you create a project in the IDE's System Builder.

This example assumes that you're familiar with the IDE's System Builder. See the Building OS and Flash Images chapter in the IDE *User's Guide* for more information.

Instead of building a new System Builder project from scratch, you can import the buildfile from the command-line example above. You can then modify the resulting System Builder project to suit your needs.

Here are the general steps required to make a System Builder project that's identical to the previous buildfile example:

1. Create a new System Builder project. You'll need to select the target platform. Note that the project contains **`libc.so`** and the link to **`ldqnx.so.2`** already.
2. Add these binaries:
   - **`devc-pty`**
   - **`ksh`**
   - **`slogger`**
   - **`sloginfo`**
   - **`Photon`**
   - **`io-graphics`**
   - **`io-display`**
   - **`devi-hirun`**
   - **`pwm`**
   - **`/usr/photon/bin/pterm`**
   - **`/usr/photon/bin/phcalc`**

   You need to use the equivalent of **`[+raw]`** on **`pterm`** and **`phcalc`** by selecting them, and setting the Strip File property to No.
3. Next, add the required libraries. You'll find that some of them are already in the project, as the System Builder identifies libraries required by binaries, and adds them automatically.

   Standard libraries:
   - **`libm.so`**

   Advanced Graphics libraries:
   - **`libffb.so`**
   - **`libdisputil.so`**
   - **`devg-XXXXX.so`**
   - **`libgf.so`**
   - **`libGLES_CM.so`**
   - **`libimg.so`**
   - **`img_codec_XXXXX.so`**

   Photon libraries:

- **`libph.so`**
- **`libAp.so`**
- **`libphexlib.so`**

Graphics libraries:

- **`libphrender.so`**
- **`libdisputil.so`**
- **`libffb.so`**

Font libraries:

- **`libfontutils.so`**
- **`libblkcache.so`**
- **`libFF-T2K.so`**
- **`libFF-T2K-cache.so`**
- **`libFF-T2K-fm.so`**
- **`libfont.so`**
- **`phfont.so`**

4 Now add the DLLs:

- **`devg-svga.so`**
- **`/lib/dll/font/ttfFFcore.so`**
- **`/lib/dll/font/PHFcore.so`**
- **`/lib/dll/font/Fcore.so`**

5 Add these required files.

Fonts and the font configuration files:

- **`/usr/photon/font_repository/tt2009m_.ttf`**
- **`/usr/photon/font_repository/phcursor.phf`**
- **`/usr/photon/font_repository/fontmap`**
- **`/usr/photon/font_repository/fontopts`**
- **`/usr/photon/font_repository/fontdir`**
- **`/usr/photon/font_repository/fontext`**

Other required configuration files:

- **`/etc/system/config/crtc-settings`**
- **`/usr/photon/palette/default.pal`**
- **`/usr/photon/keyboard/en_US_101.kbd`**

6 Finally set up these symbolic links:

- **`/bin/sh`** = **`/proc/boot/ksh`**
- **`/dev/console`** = **`/dev/ser1`**
- **`/tmp`** = **`/dev/shmem`**

You can now build the image, transfer it to your target, and run it.

Advanced topics

This section covers some of the more advanced topics in embedding Photon. It covers:

- Configuring fonts

Configuring fonts

Configuring fonts and installing the font server components in the correct location is the most difficult part of embedding Photon.

To configure the font system, you need to:

1. decide whether to run an internal or external font server
2. determine which fonts your system requires
3. determine the font binaries required
4. set up the font configuration files.

Internal or external?

The first decision you must make about the font service is how the server is started. It can run as a stand alone process (we refer to this as an *external* server) or as a plugin to **io-graphics** (which we call an *internal* server).

We recommend to run an external server in these conditions:

- Your system will not run io-graphics.
- Your system will be used as a server for remote photon sessions.
- Your system will be restarting io-graphics.

To run an external font server, start **phfont** before **io-graphics**. To run an internal font server, simply start **io-graphics** using the **-f local** option.

Required fonts

When building an embedded system, you also need to make careful decisions about the level of font support, including which fonts you need, and whether or not you need scalable fonts, since extra fonts make for a larger image size and potentially longer startup time.

Unlike bitmap fonts, scalable fonts are defined mathematically, and can be rendered at any font size. Scalable fonts elminate the need to store large numbers of glyphs that are required to render bitmap fonts at different font sizes.

The first step is to decide which fonts you need:

- You're very likely to need the cursor font, **phcursor.phf**, as the graphics driver requires it to render the standard Photon cursors.

- If your embedded system includes **pterm**, you need the terminal support fonts, PC Terminal (**pcterm*.phf**), PC Serif (**pcs*.phf**), and PC Sanserif (**pcss*.phf**) font families. You probably also need a **$**HOME**/.photon/pterm.rc** file, or a **$**PHOTON_PATH**/config/pterm.rc** file to configure the terminal font.
- Most widget-based applications expect these aliases to be defined by being appropriately mapped in the **fontmap** file (see below):
  - **TextFont**
  - **MenuFont**
  - **FixedFont**
  - **BalloonFont**
  - **TitleFont**
- A web browser requires these types of fonts:
  - body font (e.g. PrimaSans BT, Dutch 801 Rm BT)
  - heading font (e.g. Swis721 BT)
  - nonproportional font (e.g. Courier10 BT, PrimaSansMono BT)

  Check the browser's configuration to see which fonts are expected, and use those fonts, modifying the configuration to reflect what you've installed, or use the **fontmap** file to map them at runtime.

You can map, or substitute, font names by using the **fontmap** file. For more information on the format of **fontmap** and other font configuration files, see **phfont** in the QNX Neutrino *Utilities Reference*.

Required fonts binaries

You may be able to reduce the number of binaries required by the Photon font system, depending on the types of fonts you need to render on your target. Each font type has an associated plugin that supports that type, and each plugin in turn requires additional libraries. Each plugin requires **libblkcache.so** and **libc.so**. The following table summarizes additional, plugin-specific, requirements:

Fonts supported	Plugin	Required libs
Bitstream TrueDoc (**.pfr**) TrueType collections (**.ttc**)	**FCcore.so**	**libFF-T2K.so**
Photon bitmap (**.phf**)	**PHFcore.so**	**libfontutils.so**
TrueType (**.ttf**), Adobe Type1 (**.pfa**), Adobe Type2 (**.cff**), Bitstream Stroke (**.ffs**), Bitstream Speedo (**.spd**, public encryption only), Bitstream T2K (**.t2k**)	**ttfFFcore.so**	**libFF-T2K.so**

continued...

You can use the **`-b`** commandline option for **`phfont`** or **`io-graphics`** to generate a font usage report. The report file contains information about font names and font files used by your application while the font server was running. This allows you to put the only required fonts and DLLs on your target system. Note that the font usage report doesn't contain a record of dynamically loaded fonts (see the *PfDynamicLoad*()* set of functions).

Configure the font server

The font system is configured with various files. The minimum configuration requires:

- **`fontdir`** — a directory of known fonts. This file should list every font in your embedded system.

Recommended additional configuration files are:

- **`fontmap`** — a set of font-mapping rules
- **`fontext`** — missing/dropout characters rules
- **`fontopts`** — configuration options

For more information about the format of each of these files, see **`phfont`**.

You can configure the fonts on the embedded system itself, but it's easier to use your development system to configure the fonts to mimic the desired configuration for the embedded system, then assemble the font data and configuration files in the appropriate Embedded File System (EFS) build image directory.

If you're using a self-hosted development system to mimic your target, then the font server can aid in determining which fonts you need by logging failed requests for fonts that aren't mapped (explicitly or otherwise). See **`phfont`** for more information.

In an embedded system with only a few applications, chances are you'll need far fewer fonts than a desktop system requires. In this situation, you can provide minimal configuration files (all located in **`/usr/photon/font_repository`**):

`fontdir`
: This file needs to list only the fonts you're installing in **`/usr/photon/font_repository`**. You can edit this file in one of two ways:

  - Edit the default existing fontdir file by hand, removing all the lines that refer to fonts you're not including in your image.
    Or:
  - Generate this file using the **`mkfontdir`** utility (on all hosts).

`fontext`
: Make a copy of the default file and edit it.

`fontmap`	Make a copy of the default file and edit it. This file can consist of a single line: `?=primasansbts` (If you aren't including PrimaSans BT in your image, change this to the name of the font you want to use as a default).

Appendix E

Using PhAB under Microsoft Windows

In this appendix. . .

This appendix describes the main differences between the Windows and native QNX Neutrino versions of PhAB.

Photon in a single window

Like the native QNX Neutrino version of PhAB, the Windows version uses Photon and the Photon Window Manager (**pwm**) to manage its windows and dialogs. The main difference is that under Windows, Photon runs *within* a single window.

When you launch PhAB, it first starts a console window that it uses only for status messages. Next, the main Photon window is created. All PhAB windows and dialogs appear within this main window, and any sub-applications launched by PhAB stay within this window. Note that you can run multiple PhAB sessions within this single Photon window.

You can minimize application windows within the Photon window, but since there's no **shelf** application running, the right mouse button has been set up to list all running applications and to let you bring them to the foreground. To do this, simply click the right mouse button on a blank area of the main Photon window, then select the application you wish to bring to the foreground.

Exiting PhAB

When you exit PhAB, it attempts to shut down all Photon components as well, unless there are other applications still running within the Photon window (such as a second PhAB session or the language editor).

If all Photon components don't shut down automatically, or if you just want to force everything to exit in the event that the system is having problems, you should manually close the **Console for PhAB** window. If this does not work, type:

```
ph -kill
```

from a Windows command prompt.

Advanced options

If you wish to specify command-line options to **pwm** or to the **photon** server, you can use these environment variables:

- **PWMOPTS**
- **PHOTONOPTS**

You set these environment variables using the Environment tab (in the System program of the Windows Control Panel).

For details on the command-line options for any QNX Neutrino utility, see the *Utilities Reference*.

PHINDOWSOPTS

You can use the **PHINDOWSOPTS** environment variable to pass extra options to the special version of Phindows that's used as the display driver for PhAB for Windows. For example, if your Neutrino target is using an 8-bit 256 color mode with a nondefault palette, then to allow the PhAB display on the Windows host to show the colors as they'll appear on the target, you can do the following:

1. Set the Windows display mode to 256 colors.

   In Windows XP, unlike Windows NT/2000, you do this on a per-application basis, by using the Compatibility tab on the properties for the application's shortcut or executable.

2. Set the **PHINDOWSOPTS** environment variable to specify the same palette file as will be used on the target. For example, if the palette file is called **`grey.pal`**, you might set this variable to **`-P%QNX_TARGET%/usr/photon/palette/grey.pal`**. In this case, the direction of the slashes doesn't matter.

To set environment variables globally, you can use the Environment Variables button in the Advanced tab of the System program in the Windows Control Panel. To set a variable temporarily, use **`set`** *variable*= in a Command Prompt window, and then type **`phab`** to run the Photon Application Builder with that environment variable set.

Using the clipboard

To use the clipboard in PhAB under Windows, you must have permission to write to the directory specified by the **HOME** environment variable. (If **HOME** isn't set, PhAB uses **HOMEDRIVE** followed by **HOMEPATH**).

If none of these environment variables are set under Windows, or your user account doesn't have permission to write to the directory specified, PhAB's clipboard function (cutting and pasting widgets) won't work.

If you have **HOME** set, and it doesn't exist, PhAB won't be able to launch.

Transferring PhAB projects

When transferring PhAB projects between QNX Neutrino and Windows it is important to preserve the exact contents of all files. In other words, each file should be treated as binary, not ASCII, and the case of all filenames should be preserved. The easiest way to ensure this is to create a zip archive at the source, transferring the single binary zip file, and then extracting the contents of the archive at the destination.

Note that **`tar`** files can be used to transfer PhAB projects, but if you use **`Winzip`** to extract the contents of a tar archive on Windows it will by default treat files as ASCII and change their line endings. To prevent this you must deselect the **TAR file smart CR/LF conversion** option in **`Winzip`**.

Debugger launch line

You can launch the **GDB** debugger from within PhAB. Edit the project properties to specify the debug command that PhAB should use (for more information, see the Generating, Compiling, and Running Code chapter). The default debug command and arguments used by PhAB on Windows is:

```
gdb_phab.bat -debugger nto$TPR-gdb --symbols
```

This runs the commands in the **gdb_phab.bat** batch file.

PhAB automatically sets the **TPR** environment variable before issuing the debug command. It contains the name of the current target processor, as determined by the last build performed. Possible values are currently **x86**, **ppc**, **mips**, **sh** and **arm**. Having this variable in the debug command automatically selects the correct debugger executable to use.

Finally, since the Windows version of PhAB is never used for self-hosting, PhAB passes the **--symbols** option to GDB by default. This is like the **symbol** GDB command, and makes **gdb** load symbols from the PhAB executable without making it the program executed when you use the **run** command. This lets you run the executable on the remote target. Here are the initial commands from a typical debug session after you start GDB using the default launch line:

```
(gdb) target qnx com1
(gdb) upload myprog
(gdb) run myprog
(gdb) break main
(gdb) continue
```

For the above, we assume that we're connected to the target machine via serial port com1 and that the **pdebug** remote debug agent is already running on the target.

If you want to use a graphical debugger, use the IDE that's part of QNX Momentics. Create a Photon Appbuilder project within the IDE and launch PhAB from there.

Custom widget development and PhAB

Photon lets you create applications using widgets that you've built yourself or obtained from a third party. You can build these custom widget libraries into your application and run it on your intended target.

For documentation on writing source code for custom widgets, getting them to run within your application on the target, and getting PhAB to recognize your custom widgets, see *Building Custom Widgets*. The process for doing these things is essentially the same on all host platforms.

PhAB can dynamically load custom widgets and display them properly as you develop your application on the host machine.

To make PhAB display custom widgets correctly on the host as you develop your application, you need to take some extra steps to recompile and link the custom widget source code for the host platform and processor. This means building shared libraries that PhAB dynamically loads at runtime. If you do not do this, you can set dimensions and specify resources and callbacks, but these settings won't take effect and be displayed until you run your application on the target.

For the following procedure, we assume that you've already performed the steps that aren't specific to the host platform, namely:

- Obtain the custom widget source code.
- Build target libraries from your custom widget sources.
- Add appropriate entries in PhAB's palette definition files.
- Create icon image and resource defaults files in the templates directory.

After you've done all the above, you can begin to use the custom widget class in PhAB. To make PhAB display custom widgets correctly as you use them, follow these additional Windows-specific steps:

1. Download and install the Cygwin development environment from **www.cygwin.com**. This is an open-source UNIX like system that offers a **gcc**-based development environment under Windows. We recommend Cygwin version 1.5.5 or later.
2. After installation, start a Cygwin Bash shell to perform the following compile and link steps.
3. Compile your custom widget functions using Cygwin's **gcc**:

```
gcc -c -nostdinc \
-I /usr/lib/gcc-lib/i686-pc-cygwin/‘gcc -dumpversion‘/include \
-I$QNX_HOST/usr/include -I/usr/include \
-I$QNX_TARGET/usr/include MyWidget.c
```

4. Link your custom widget object files to create a shared library:

```
ld -shared -e _dll_entry@12 MyWidget.o \
-o MyWidget.dll -L$QNX_HOST/usr/lib -lph \
-lcygwin -lkernel32
```

5. Place your shared object (in this example **MyWidget.dll**) in a directory specified by the **PATH** environment variable. Note that **LD_LIBRARY_PATH** is not recognized by Windows.

The next time you start PhAB after completing these steps, you should be able to see custom widgets displayed correctly as you work with them. If not, consult the PhAB console window for error messages.

Using custom TrueType fonts and PhAB

You can configure PhAB for Windows to use TrueType fonts that you supply, in addition to the standard font set installed with QNX Momentics. To do this:

1. Install the font in Windows using the control panel. See the Windows help for more information on this step.
2. Copy the font's **`.ttf`** file to the Photon font repository directory, **`%QNX_TARGET%\usr\photon\font_repository`**.
3. Run **`mkfontdir`** on the directory to register the new TrueType font in the **`fontdir`** file.

   For example: **`mkfontdir -d %QNX_TARGET%\usr\photon\font_repository`**

The font is now available when you run PhAB standalone or in the IDE.

Photon Hook DLLs

Photon supports a special method to change the look and feel of widgets at runtime called "Photon hook DLLs". This method can be used to "skin" an application without recompiling it. For more information about Photon hook DLLs, see the Widget styles section of the Managing Widgets in Application Code chapter.

To make PhAB display widgets correctly on a Windows host as you develop your application, the Photon hook DLL needs to be built under Windows using the following steps:

1. Download and install the Cygwin development environment from **`www.cygwin.com`**. This is an open-source UNIX like system that offers a gcc-based development environment under Windows. We recommend Cygwin version 1.5.5 or later.
2. After installation, start a Cygwin Bash shell to perform the following compile and link steps.
3. Compile your Photon hook DLL source code using Cygwin's **`gcc`**, for example:

```
gcc -c -nostdinc \
 -I /usr/lib/gcc-lib/i686-pc-cygwin/‘gcc -dumpversion‘/include \
 -I$QNX_HOST/usr/include -I/usr/include \
 -I$QNX_TARGET/usr/include hook.c
```

4. Link the actual shared library, for example:

   ```
   ld -shared -e _dll_entry@12 hook.o -o PtHook.dll \
    -L$QNX_HOST/usr/lib -lph -lcygwin -lkernel32
   ```

The above steps create a Photon hook DLL called **`PtHook.dll`**. To use this DLL, place it in the search path (specified by **$PATH**) before running PhAB.

Alternatively, you can use a version of the PtMultiHook sample code as the **PtHook.dll** (see the Widget styles section of the Managing Widgets in Application Code chapter). This DLL can be found at $QNX_HOST**/usr/photon/bin/pt_multihook.dll**, and should be renamed to **PtHook.dll** before use. This code looks for the PHOTON_HOOK environment variable, and loads the DLL it points to. If it points to a directory, it loads all the DLLs in the directory. It then executes the *PtHook()* function in any loaded DLLs.

For example, if your hook DLL is placed in **C:\my_photon_hooks\**, you can type the following two commands from a Windows command prompt:

```
set PHOTON_HOOK=c:/my_photon_hooks
appbuilder
```

Or you can type the following from **bash**:

```
export PHOTON_HOOK=c:/my_photon_hooks
appbuilder
```

To set the environment variable permanently, use the System option on the Windows Control Panel.

Running multiple copies of PhAB

On Windows XP, multiple users might want to run PhAB. In this case, each instance of the PhAB application has to have a unique TCP/IP port for the Photon server and the PhFont server.

The default port numbers are 4871 for Photon and 4870 for PhFont. A Windows XP user who wants to run PhAB in isolation of other users needs to select unique, unused port numbers by setting the environment variables **PHOTON** and **PHFONT**. For example, the following two settings could be used:

```
set PHOTON=tcp:127.0.0.1:4873
set PHFONT=tcp:127.0.0.1:4872
```

The port numbers are at the end of the strings, and can be set to any unused TCP/IP port.

Appendix F
PhAB Keyboard Shortcuts

Quick Reference
This Appendix contains a quick reference guide to Photon Application Builder's keyboard shortcuts. The following types of shortcuts are available:

- Project management
- Editing
- Adding items
- Building
- Widget management
- Other shortcuts

Project management shortcuts

Command	**Shortcut**
New project	Ctrl-N*
Open project	Ctrl-O*
Save project	Ctrl-S*
Save project as	Ctrl-Shift-S*
Print	Ctrl-P
Exit	Ctrl-X

*These commands aren't available in the IDE-hosted version of PhAB.

Editing shortcuts

Command	**Shortcut**
Undo last action	Ctrl-Z
Redo last undo	Ctrl-Shift-Z
Cut	Ctrl-X, Shift-Delete
Copy	Ctrl-C
Paste	Ctrl-V, Shift-Insert
Move a widget into a container	Ctrl-T
Delete	Del
Select all	Ctrl-A
Select All Children	Ctrl-Shift-A

continued. . .

Command	Shortcut
Deselect current selection	Ctrl-D
Find	Ctrl-Shift-F
Add a widget class	Ctrl-W
Edit templates	Ctrl-M
Edit preferences	Ctrl-Shift-P

Adding items shortcuts

Command	Shortcut
Add window	Ctrl-Shift-W
Add dialog	Ctrl-Shift-D
Add menu	Ctrl-Shift-M
Add picture module	Ctrl-Shift-U
Internal links	F4
Project properties	F2

Building shortcuts

Command	Shortcut
Build and run	F6*
Build and debug	F5*
Rebuild all	F3*
Build	F7*
Make clean	Shift-F3*
Generate UI	Shift-F7
Run arguments	Shift-F6*
Manage targets	F11*

*These commands aren't available in the IDE-hosted version of PhAB.

Widget management shortcuts

Command	**Shortcut**
Move to front	Ctrl-F
Move to back	Ctrl-B
Group	Ctrl-G
Ungroup	Ctrl-Shift-G
Change class	Ctrl-H
Define template	Ctrl-L
Select next widget in module tree	F10
Multiple-select the current and next widget in the module tree	Shift-F10
Select previous widget in module tree	F9
Multiple-select the current and previous widget in the module tree	Shift-F9
Toggle the nudge mode through move, shrink, and expand	Ctrl-5*
Move, shrink or expand the selected widget by one pixel	Ctrl-1 to Ctrl-9* (excluding Ctrl-5) and Ctrl-arrow

*These shortcuts are on the numeric keyboard only.

View and window shortcuts

Command	**Shortcut**
Zoom in	Ctrl-+
Zoom out	Ctrl--
Close current window	Ctrl-Alt-L
Close all windows	Ctrl-Alt-A
Show/hide the resources tab	F12
Show/hide the callbacks tab	Shift-F12

Other shortcuts

Command	Shortcut
Help	F1

Appendix G
What's New

In this appendix. . .

This chapter describes what's new and changed in the Photon *Programmer's Guide*.

What's new in Photon for QNX Neutrino 6.5.0

New content

- The Photon in Embedded Systems appendix has been updated to reflect minor path changes. Some additonal information about fonts and font support has been added.
- The Working with Applications and Editing Resources and Callbacks in PhAB chapters now contain a note about editing 24-bit per pixel JPEG images using the pixmap editor.

What's new in Photon for QNX Neutrino 6.4.1

New content

- The Understanding Encodings, Fonts, Languages and Code Tables chapter has been added to help answer some common misconceptions about fonts and encodings.
- The language code table in the International Language Support chapter has been updated.

What's new in Photon for QNX Neutrino 6.4

New content

- The Photon graphics architecture on embedded systems has changed. For more information, refer to the **io-display** utility chapter in the Neutrino Utilities Reference for more information.
- The Photon in Embedded Systems appendix describes how to use the **io-display** utility to embed photon on an embedded device. When you want to run both GF/OpenGL ES and Photon applications, you must start **io-display** before starting the Photon graphics server, **io-graphics**.
- The Fonts chapter has been updated to describe how fonts are handled under the **io-display** graphics framework.
- The Widgets chapter contains a paragraph that describes how to use Shift-drag to move a widget by its resize handle. This is new functionality for 6.4.

For information on migrating from previous versions of QNX Neutrino, refer to the product release notes.

What's new in Photon for QNX Neutrino 6.3

New content

- The PhAB’s Environment — PhAB’s interface has changed, including an updated menu and simplified toolbar.
- The Geometry Management chapter now describes how to use layouts to manage widget placement.
- The directory structure for PhAB projects has changed, and is described in How application files are organized.
- The Generating, Compiling and Running Code chapter now describes how you can Manage targets.
- The Raw Drawing and Animation chapter now describes how you can use layers.
- The Fonts chapter is updated with information about the new font library.
- The Photon in Embedded Systems appendix has a new example of creating a floppy containing Photon and some applications, and is updated with new font library information pertinent to embedded systems.

What’s new in Photon for QNX Neutrino 6.2.1

New content

- Listed the supported platforms; see “Versions and platforms” in the Introduction.
- The Interprocess Communication chapter has a better description of how to use Photon connections.
- There’s a new section, “Layers,” in the Raw Drawing and Animation chapter.
- Added a description of the **PHINDOWSOPTS** environment variable to the Using PhAB under Microsoft Windows appendix.

Errata

- The libraries in `/usr/photon/lib` are provided for runtime compatibility with Photon for QNX Neutrino 6.0 (x86 only). The current libraries are in `/usr/lib`. For more information about the libraries, see “Photon libraries” in the Introduction.
- Corrected the call to *ionotify()* in “Sending the pulse message to the deliverer” in the Interprocess Communication chapter.
- The instructions for printing a `PtMultiText` widget have been corrected.
- The order of the options to the `on` command have been corrected in “Putting it all together” in the Photon in Embedded Systems appendix.
- If you want to use a graphical debugger when developing in Windows, use the IDE that’s part of QNX Momentics.

What’s new in Photon for QNX Neutrino 6.2.0

New content

- The Edit menu now includes Undo and Redo commands. For more information, see the chapter on PhAB’s Environment.
- PhAB can’t import QNX Windows picture files any more.
- You can now specify a list of library callback functions when you start PhAB. For more information, see **`appbuilder`** in the QNX Neutrino *Utilities Reference*.
- “Making a DLL out of a PhAB application” in the Generating, Compiling, and Running Code chapter
- “Widget styles” in the Managing Widgets in Application Code chapter
- “Offscreen locks” in the Raw Drawing and Animation chapter.
- Using PhAB under Microsoft Windows appendix

What’s new in Photon for QNX Neutrino 6.0

This section doesn’t try to describe all the changes to PhAB’s user interface; most you’ll discover by trying it yourself or by scanning this manual. Instead, this section lists only the major changes.

Introduction

- The geometry of a widget has changed slightly; it now includes the widget’s border. For more information, see “Widget geometry.”

PhAB’s Environment

- You no longer need to press Enter after giving an instance name to a widget.

Working with Applications

- It’s no longer possible to override the standard Photon mainloop function.

Working with Modules

- **`PtWindow`** widgets (which are used to instantiate Window modules) no longer include an icon resource. You must now use PhAB to associate an icon with the window.
- You can no longer create “other” modules (file selectors or messages) in PhAB, although they’re still supported for existing applications. Instead of the file selector, use one of:
  - **`PtFileSel`**
  - *PtFileSelection()*

  Instead of the message module, use one of:
  - *PtAlert()*

  - *PtMessageBox()*
  - *PtNotice()*
  - *PtPrompt()*

  For more information, see the Photon *Library Reference*.

Creating Widgets in PhAB

- You can now create templates, or customized widgets, to use as the basis when creating other widgets.

Geometry Management

- In the current version of the Photon microGUI, widgets are anchored immediately upon creation. In earlier versions, anchoring is done when the widgets are realized.
- If the resize policy conflicts with the anchors, the *Pt_ARG_RESIZE_FLAGS* override *Pt_ARG_ANCHOR_OFFSETS* and *Pt_ARG_ANCHOR_FLAGS*.

Working with Code

New sections:

- Timers

Manipulating Resources in Application Code

New sections:

- Setting image resources
- Setting one resource
- Getting image resources (pointer method)
- Getting one resource

Other changes:

- When setting string resources, the fourth argument to *PtSetArg()* is the number of bytes to copy; if it's 0, *strlen()* is used to determine the length of the string.
- Changes to the widget's state may invalidate the pointers returned by *PtGetResources()*; use them promptly.

Managing Widgets in Application Code

New sections:

- Ordering widgets

Context-Sensitive Help

- The *PxHelp** functions are now named *PtHelp** and are in the main Photon library, **`ph`**.

Interprocess Communication

New sections:

- Connections — the best method of IPC for Photon applications.

Other changes:

- As described in "Adding an input handler," an input handler must return one of the following:

Pt_CONTINUE	The input handler doesn't recognize the message. If there are other input handlers attached to the same process ID, they're called. If there are no input handlers attached specifically to this process ID, or if all input handlers attached specifically to this process ID return Pt_CONTINUE, the library looks for input handlers attached to pid 0. If all the input handlers return Pt_CONTINUE, the library replies to the message with an ENOSYS.
Pt_END	The message has been recognized and processed and the input handler needs to be removed from the list. No other input handlers are called for this message.
Pt_HALT	The message has been recognized and processed but the input handler needs to stay on the list. No other input handlers are called for this message.

This creates several incompatibilities with earlier versions of the Photon microGUI:

- If an input handler replies to the message and returns Pt_CONTINUE (or if the message is from a proxy/pulse), everything should be OK. The current library tries and fails to reply again, but that's harmless. Still, it's a good idea to change the code to return Pt_HALT; this prevents the library from calling other input handlers or replying.
- If an input handler returns Pt_CONTINUE without replying to the message, the old library doesn't reply either, but the current one does. You need to change the code to return Pt_HALT.
- If an input handler returns Pt_END (which is the most obvious value other than Pt_CONTINUE), the only situation that can cause a problem is when you have multiple input handlers attached to the same process ID.
- If an input handler returns a value other than Pt_CONTINUE or Pt_END, the old library removes it from the list but the new library doesn't. You need to change the code to return Pt_END.

Parallel Operations

New sections:

- Threads

Raw Drawing and Animation

New sections:

- Direct mode
- Video memory offscreen
- Alpha blending support
- Chroma key support
- Extended raster operations
- Video modes
- Gradients

Other changes:

- If you use *PxLoadImage()* to load an transparent image, set PX_TRANSPARENT in the *flags* member of the **`PxMethods_t`** structure. If you do this, the function automatically makes the image transparent; you don’t need to create a transparency mask. See “Transparency in images.”

Fonts

New chapter.

Printing

The entire API has been made simpler. Applications that call the old routines should still work, but you should reread this chapter.

Drag and Drop

New chapter.

Events

New sections:

- Pointer events
- Event handlers

Glossary

accelerator

See **hotkey**.

activate

A widget is usually *activated* when you release a mouse button while pointing at an **armed** widget.

active window

The window that currently has **focus**.

anchor offset

The distance between the edges of a widget and the parent widget it's anchored to.

anchor

A constraint mechanism used to manage what happens to a widget when its parent is expanded or contracted. For example, a pane that's anchored to the sides of a window expands or contracts as the window's size is changed.

application region

A **region** that belongs to a Photon application (as opposed to a Photon system process, such as the window manager, graphics drivers, etc.). An application region is usually placed behind the **device region**. Also called a **window region**.

argument list

An array of type `PtArg_t` used when setting and getting widget resources.

arm

A widget is usually *armed* when you press a mouse button while pointing at it.

backdrop

An image that's displayed as a background on your screen.

backdrop region

A region placed behind all windows to display a background image.

balloon

A small box that pops up to define or explain part of the user interface. A balloon is displayed when the pointer pauses over a widget.

bitmap

A color picture consisting of one or more **bitplanes**.

bitplane

An array of bits representing pixels of a single color in a **bitmap**.

blit

An operation that moves an area of a graphics context (e.g. the screen) to another area on the same or a different context.

callback

A **callback function** or a **callback resource**.

callback function

Code connecting an application's user interface to its code. For example, a callback is invoked when you press a button.

callback resource

A **resource** that specifies a list of functions and their client data to be called when a certain action occurs.

canvas

The part of a widget that's used for drawing. For **`PtWidget`**, this is the area inside the widget's borders. For **`PtBasic`** and its descendants, the canvas is the area inside the widget's border and **margins**. Other widgets, such as **`PtLabel`**, may define additional margins.

class

See **widget class**.

class hierarchy

The relationships between all of the widget classes.

client data

Any arbitrary data the application may need to provide to a callback function.

clipping list

An array of rectangles used to restrict output to a particular area.

clipping rectangle

A rectangle used to restrict output to a particular area.

CMY value

A color expressed as levels of cyan, magenta, and yellow.

CMYK value

A color expressed as levels of cyan, magenta, yellow, and black.

code-type link callback

In a PhAB application, an application function that's called when a widget's callback list is invoked.

color depth

The number of bits per pixel for a screen or pixmap.

Common User Access

See **CUA**.

compose sequence

A sequence of key presses that can be used to type a character that might not appear on the keyboard.

console

One of nine virtual screens on the **desktop**. Also called a **workspace**.

consume

When a widget has processed an event and prevents another widget from interacting with the event, the first widget is said to have **consumed** the event.

container

A widget that can have other widgets as children. For example, `PtWindow`, `PtGroup`, and `PtOSContainer`.

cooked event

A key or pointer event that has been assigned a location in the Photon event space. Also called a **focused event**.

CUA

Common User Access — a standard that defines how you can change focus by using the keyboard.

current item

The item in a list or tree widget that will be selected (or perhaps unselected) when you press Enter or Space. It's typically drawn with a blue dotted line around it when its widget has focus.

cursor

An indicator of a position on a screen, such as a **pointer** or an insertion point in a text field.

damaged

Whenever a widget needs to be redisplayed due to a change in the window (e.g. the widget is changed, moved, or **realized**), it's said to be *damaged*.

dead key

A key that, when pressed, doesn't produce a symbol, but initiates a **compose sequence**.

default placement

The placement of a region when no siblings are specified. The opposite of **specific placement**.

desktop

The virtual screen, consisting of nine **consoles** or **workspaces**.

device region

The **region** located in the middle of the **event space**, with **application regions** behind it and **driver regions** in front of it (from the user's point of view).

dialog module

A PhAB **module** similar to a **window module**, except that a dialog module can have only one instance per process.

direct-color

A color scheme in which each pixel is represented by an RGB value. Contrast **palette-based**.

disjoint parent

A disjoint widget that's the ancestor of another widget.

disjoint widget

A widget that can exist without a parent. If a disjoint widget has a parent, it can exist outside its parent's canvas. For example, `PtWindow`, `PtMenu`, and `PtRegion` are disjoint widgets, but `PtButton`, `PtBkgd`, and `PtRect` aren't.

A disjoint widget owns regions that aren't children of its parent's regions. Any clipping set by the parent of a disjoint widget isn't applied to the disjoint widget. The regions of disjoint widgets are sensitive and opaque to expose events.

dithering

A process whereby pixels of two colors are combined to create a texture or a blended color.

draw context

A structure that defines the flow of the draw stream. The default draw context emits draw events to graphics drivers. **Print contexts** and **memory contexts** are types of draw contexts.

draw stream

A series of tokens that are dispatched via draw events and can be collected by a rendering engine such as a graphics driver.

driver region

A **region** created by a driver, usually placed in front of the **device region**.

encapsulation driver

A program that displays Photon graphical output inside another windowing system such as the X Window System.

event

A data structure that represents an interaction between you and an application or between applications. Events travel through the event space either toward you or away (i.e. toward the **root region**).

event compression

The merging of events such that the application sees only their latest values. The application doesn't have to process many unnecessary events.

event handler

A callback function that lets an application respond directly to Photon events, such as dragging events.

event mask

A set of event types that are of interest to an **event handler**. When one of these events occurs, the event handler is invoked.

event space

An abstract, three-dimensional space that contains regions — from the root region at the back to the graphics region at the front. You sit outside the event space, looking in from the front. Events travel through the event space either toward the root region or toward you.

exported subordinate child

A widget created by a container widget (as opposed to an application) whose resources you can access only through the parent.

exposure

Typically occurs when a **region** is destroyed, resized, or moved. Expose events are sent to applications to inform them when the contents of their regions need to be redisplayed.

extent

A rectangle that describes the outermost edges of a widget.

File Manager

The Photon File Manager (PFM), an application used to maintain and organize files and directories.

focus

A widget that has *focus* will receive any key events collected by its window.

focus region

A region placed just behind the **device region** by the **Photon Window Manager** that lets it intercept key events and direct them to the **active window**.

focused event

A key or pointer event that has been assigned a location in the Photon event space. Also called a **cooked event**.

folder

In the Photon File Manager, a metaphor for a directory.

GC

See **graphics context**.

geometry negotiation

The process of determining the layout for a widget and its descendants, which depends on the widget's layout policy, any size set for the widget, and the dimensions and desired positions of each of the widget's children.

global header file

A header file that's included in all code generated by PhAB for an application. The global header file is specified in PhAB's Application Startup Information dialog.

graphics driver

A program that places a region that's sensitive to draw events on the user's side of the device region, collects draw events, and renders the graphical information on the screen.

graphics context (GC)

A data structure that defines the characteristics of primitives, including foreground color, background color, line width, clipping, etc.

Helpviewer

A Photon application for viewing online documentation.

hotkey

A special key or keychord that invokes an action (such as a menu item) without actually selecting a widget. Also called an **accelerator**. Contrast **keyboard shortcut**.

hotspot

The part of the pointer that corresponds to the coordinates reported for the pointer (e.g. the intersection of crosshairs, or the tip of the arrow of the basic pointer).

HSB

Hue-Saturation-Brightness color model.

HSV

Hue-Saturation-Value color model.

icon module

A PhAB module that associates icons with an application.

image

A rectangular array of color values, where each element represents a single pixel. See also **direct-color** and **palette-based**.

initialization function

In a PhAB application, a function that's called before any widgets are created.

input driver

A program that emits, and is the source of, key and/or pointer events.

input group

A set of input and output devices. There's typically one input group per user.

input handler (or input-handling function)

A function that's hooked into Photon's main event-processing loop to handle messages and **pulses** sent to the application by other processes.

instance

A concrete example of an abstract class; for example, "Lassie" is an instance of the class "dog." In Photon, an instance is usually a widget instance; for example, a pushbutton is an instance of the `PtButton` widget class. When an instance of a widget is created, the initial values of its **resources** are assigned.

instance name

In PhAB, a string that identifies a particular instance of a widget so that you can access the instance in your application's code.

instantiation

The action of creating an **instance** of a widget class in an application.

internal link

A PhAB mechanism that lets a developer access a PhAB module directly from an application's code.

Image Viewer

A Photon application (`pv`) that displays images.

key modifier

A flag in a key event that indicates the state of the corresponding **modifier key** when another key was pressed.

keyboard driver

A program that gets information from the keyboard hardware, builds Photon key events, and emits them towards the root region.

keyboard shortcut

A key that selects a menu item. The shortcut works only if the menu is displayed. Contrast **hotkey**.

language database

A file that contains the text strings used in a PhAB application; a language database makes it easier to create multilingual applications with PhAB's language editor.

link callback

A mechanism that connects different parts of a PhAB application. For example, a link callback can be invoked to display a dialog when a button is pressed.

margin

The area between a widget's border and **canvas**.

memory context

A **draw context** in which Photon draw events are directed to memory for future displaying on the screen, as opposed to a printer (**print context**) or to the screen directly (the default draw context).

menu module

A PhAB module used to create a menu.

method

A function that's internal to a widget class and invoked under specific conditions (e.g. to draw the widget). Methods are provided as pointers to functions in widget class records.

modifier key

A key (such as Shift, Alt, or Ctrl) used to change the meaning of another key.

module

An object in PhAB that holds an application's widgets. PhAB modules include windows, menus, icons, pictures, and dialogs.

module-type link callback

A link callback that displays a PhAB module.

mouse driver

A program that gets information from the pointer hardware, builds Photon raw pointer events, and emits them towards the root region.

opaque

The state of a region with regard to events. If a region is *opaque* to an event type, any event of that type that intersects with the region has its rectangle set adjusted to clip out the intersecting area. The region prevents the event from passing through.

palette

An array of colors. A **hard palette** is in hardware; a **soft palette** is in software.

palette-based

A color scheme in which each pixel is represented by an index into a palette. Contrast **direct-color**.

PDR

See **Press-drag-release**.

PFM

See **Photon File Manager**.

PhAB

Photon Application Builder. Visual design tool that generates the code required to implement a user interface.

phditto

A utility that accesses the Photon workspace on a remote node. See also **ditto**.

Phindows

Photon in Windows. An application that accesses a Photon session from a Microsoft Windows environment.

Photon File Manager (PFM)

An application used to maintain and organize files and directories.

Photon Manager or server

The program that maintains the Photon event space by managing regions and events.

Photon Terminal

An application (**`pterm`**) that emulates a character-mode terminal in a Photon window.

Photon Window Manager (PWM)

An application that manages the appearance of window frames and other objects on the screen. For example, the window manager adds the resize bars, title bar, and various buttons to an application's window. The window manager also provides a method of focusing keyboard events.

picture module

A PhAB module that contains an arrangement of widgets that can be displayed in another widget or used as a widget database.

pixmap

A **bitmap** or **image**.

plane mask

A mask used to restrict graphics operations to affect only a subset of color bits.

point source

A single-point **rectangle set** used as the source of an event.

pointer

An object on the screen that tracks the position of a pointing device (e.g. a mouse, tablet, track-ball, or joystick). Photon has several pointers indicating various states: Basic, Busy, Help, Move, Resize, I-beam, No-input.

Press-drag-release (PDR)

A method of selecting a menu item by pressing down a mouse button while pointing to a menu button, dragging until the desired item is highlighted, and releasing the mouse button.

print context

A **draw context** in which Photon draw events are directed to a file, as opposed to the screen (the default draw context) or to memory (**memory context**).

printer driver

A program that converts Photon draw stream format into a format suitable for a printer, including PostScript, Hewlett-Packard PCL, and Canon.

procreated widget

A widget created by another widget (as opposed to an application), such as the `PtList` and `PtText` created by a `PtComboBox`. Also known as a **subordinate child**.

pterm

A Photon Terminal; an application that emulates a character-mode terminal in a Photon window.

pulse

A small message that doesn't require a reply; used for asynchronous communication with a Photon application.

pv

See **Image Viewer**.

PWM

See **Photon Window Manager**.

raw event

An input event that hasn't been assigned a location in the Photon event space. Also called an **unfocused event**.

raw callback

A function that lets an application respond directly to Photon events such as dragging events. Also called an **event handler**.

realize

To display a widget and its descendants, possibly making them interactive.

rectangle set

An array of nonoverlapping rectangles associated with an event.

region

A rectangular area within the Photon event space that's used by an application for collecting and emitting events.

resize policy

A rule that governs how a widget resizes itself when its contents change.

resource

An attribute of a widget, such as fill color, dimensions, or a callback list.

root region

The region at the very back of the Photon event space.

sensitive

The state of a region with regard to events. If a region is *sensitive* to a particular type of event, the region's owner collects a copy of any such event that intersects with the region.

setup function

A function that's called after a PhAB module is created.

shelf

An application that attaches areas to the outside edge of the screen. You can add plugins to customize these areas, such as a taskbar, launcher, clock, and magnifier.

Snapshot

A Photon application for capturing images of the screen.

specific placement

The placement of a region when one or more siblings are specified. The opposite of **default placement**.

subordinate child

A widget created by another widget (as opposed to an application), such as the `PtList` and `PtText` created by a `PtComboBox`. Also known as a **procreated widget**.

table-of-contents (TOC) file

In the Photon **Helpviewer**, a file that describes a hierarchy of help topics.

taskbar

A shelf plugin that displays icons representing the applications that are currently running.

tile

A data structure used to build linked lists of rectangles, such as a list of the damaged parts of an interface.

topic path

Help information identified by a string of *titles* that are separated by slashes.

topic root

A topic path that's used as a starting point for locating help topics.

topic tree

A hierarchy of help information.

translation file

A file containing translated strings for a PhAB application. There's one translation file per language supported by the application.

unfocused event

See **raw event**.

Unicode

The ISO/IEC 10646 16-bit encoding scheme for representing the characters used in most languages.

UTF-8

The encoding for **Unicode** characters, where each character is represented by one, two, or three bytes.

widget

A component (e.g. a pushbutton) in a graphical user interface.

widget class

A template for widgets that perform similar functions and provide the same public interface. For example, `PtButton` is a widget class.

widget database

In PhAB, a module containing widgets that can be copied at any time into a window, dialog, or other container.

widget family

A hierarchy of widget *instances*. For example, a window and the widgets it contains.

widget instance

See **instance**.

window frame region

A region that PWM adds to a window. It lets you move, resize, iconify, and close the window.

Window Manager

See **Photon Window Manager**.

window module

A PhAB module that's instantiated as a `PtWindow` widget.

window region

A region that belongs to an application window.

work procedure

A function that's invoked when there are no Photon events pending for an application.

workspace

See **console**.

workspace menu

A configurable menu that's displayed when you press or click the right mouse button while pointing at the background of the desktop.

Index