Learning to Program with Haiku

Lesson 15

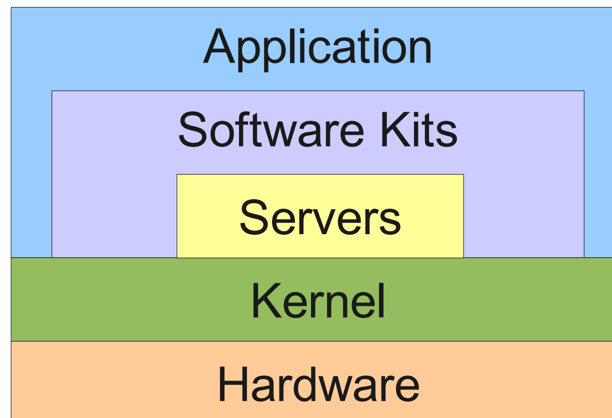
Written by DarkWyrm

All material © 2010 DarkWyrm

Now that we've written our first working – albeit simple – GUI application for Haiku, we will start learning the fundamentals of what is considered the Haiku API.

Overview of the Haiku API

All of the operating system libraries that are available to us are organized into categorized groups called kits. Some of these kits have a server associated with them. Some do not. The majority of Haiku programs work with the kits and little else, although device drivers work directly with the hardware and mostly call kernel functions. A layout of Haiku as an operating system and its "layers" looks a little like this:



As of this writing the official kits are the following:

- Application
- Device
- Game
- Interface
- Kernel
- Mail
- Media
- MIDI
- Network
- OpenGL
- Storage
- Support
- Translation

In addition to these official kits, there are also two other which are under development and considered experimental: the Layout Kit and the Locale Kit. If this seems like a lot, it's because it is, but depending on the kinds of programs that you write you may never have to deal with certain kits. We'll look at some of the kits in detail later on, but for now, we will focus primarily on the Application, Interface, and Support kits.

Application Kit

The Application Kit is small, but vital. The focus centers around BMessage, the means for communication within a program and between programs, and BApplication, which must be subclassed to write a program which uses BMessages[1].

1 This is actually a lie, but let's pretend it's not for now. It's easier that way.

Device Kit

The Device Kit provides classes for certain hardware. Because there are only two classes in this kit, it is not generally used, but it may expand in future versions of Haiku.

Game Kit

The Game Kit goes with the assumption that game programmers pretty much want to be given direct access to the section of memory used to display the screen – the video buffer – and then left alone to work their dark arts. Also here are some classes to make playing game sounds simple for game writers.

Interface Kit

To an applications programmer, the Interface Kit is as important as it is big. Windows, buttons, checkboxes, and more appear under its umbrella. Printing is also handled by this kit.

Kernel Kit

The Kernel Kit is the only kit which is not a collection of C++ classes. Instead, it consists of low-level C function calls.

Mail Kit

The Mail Kit is for constructing and sending e-mails. There's not very much else to say about it.

Media Kit

The Media Kit is all about audio and video processing. Haiku is especially good at playing and recording audio and video with a minimum of **latency**, or lag.

MIDI Kit

MIDI stands for **M**usical **I**nstrument **D**igital **I**nterface, a standard established in the ages of yore which defines how to get musical instruments to communicate with computers. The MIDI kit handles MIDI data just as the Media Kit is all about video and audio processing.

Network Kit

If you come from a UNIX-based programming background, you are probably accustomed to using C function calls for networking. The Network Kit's classes approach communications coding from a slightly friendlier perspective.

OpenGL Kit

If you're into 3D graphics, this is the kit for you. There is only one class, BGLView, but it opens the door to incorporating OpenGL graphics into your applications.

Storage Kit

The Storage Kit provides friendly ways of working with the filesystem. In addition to reading and writing files, there are also classes for reading directories, running queries, and working with attributes.

Support Kit

This kit is designed to support other kits with helper classes. BString, BList, and BLocker are especially helpful and are very commonly used.

Translation Kit

The Translation Kit is one of Haiku's innovations. It provides a single interface for reading and writing pictures and text without having to understand the underlying file formats. For us programmers, it makes life much, much easier.

Event-Based Programming

Writing a program for the console is pretty simple in that you, the developer, control the flow of execution. The same cannot be said for the GUI, with the reason that such programs are an interaction between the user and your program. The user does something and your program responds. Something strange happens in the system and your program alerts the user. Your code becomes a set of responses to different events. Much of this amounts to sending messages and handling any that come in.

An example of event-based programming is responding to the mouse. If the user clicks on a window's close button, the system will notify your program that the user is requesting the window to close. It's your job to do something about the request. When the user clicks a button, it sends a message. Who gets the message and what is done in response is up to you.

Haiku Messaging

A great deal of the communication that takes place within Haiku as an operating system centers around sending and handling messages. Most of the classes in the Application Kit center around messaging. Even though we won't use all of these right away, let's just take a quick peek at each of the classes in the kit:

- BApplication – The application class. It is also the main channel for communications between your program and the rest of the system.
- BClipboard – BClipboard handles storing information on a clipboard. The clipboard itself uses the BMessage class for storing and exchanging data with programs.
- BCursor – Not related to messaging, but BCursor takes care of changing what the mouse pointer looks like.
- BHandler – A class which is used to take appropriate actions for messages.
- BInvoker – A message-sending class used for controls such as buttons and checkboxes. Give it a message to send and a target to send messages to and it will send a copy of the message given to it each time its `Invoke()` method is called.

- BLooper – BLooper receives messages and passes them through a series of BHandlers before handling a message. It might sound confusing now, but it won't later on.
- BMessage – The type of object sent around the system for communications. It has an identifier property, `what`, and methods for attaching and retrieving data and for sending replies.
- BMessageFilter – A class used for filtering out desired – or undesired – messages.
- BMessageQueue – BMessageQueue stores messages in a first-in, first-out fashion. It is primarily used by BLooper instances to temporarily hold messages while it is handling others.
- BMessageRunner – This class sends messages at a specified interval.
- BMessenger – BMessenger is a message-sending class. It can send messages to BHandlers and BLoopers regardless of whether they are in your program or in another one.
- BPropertyInfo – Scripting is the purpose behind BPropertyInfo. If you're not enabling scripting your program from outside, you won't need this one often, if ever.
- BRoster – The BRoster class communicates with the system's application roster daemon. It is used for sending messages to all programs running in the system, launching programs, or for checking if a particular program is running.

Of all of these classes, the ones that are used in the course of regular applications programming are BLooper, BInvoker, BMessage, BHandler, and BApplication, so what seems like a lot to remember isn't really very much, especially when you consider that only a few methods of each of these classes are used frequently.

Reacting to most events in Haiku programming boils down to one function: `MessageReceived()`. It is a **hook function**, that is, a virtual function intended to be implemented by child classes to react to an event. In this case, `MessageReceived()` is implemented by child classes to handle messages that are not already handled by the parent class. Any child class of BHandler, including BLooper, BApplication, BWindow, and BView, have this hook function. Most of the time, it will look like this:

```
void
MyWindow::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		case M_SOME_MESSAGE:
		{
			DoSomething();
			break;
		}
		default:
		{
			// This calls the version of MessageReceived implemented by
			// MyWindow's parent class, BWindow.
			BWindow::MessageReceived(msg);
			break;
		}
	}
}
```

`MessageReceived()` can end up handling many different message codes, so a `switch` statement is called for here, and the `switch` differentiates between messages using the `what` identifier. Calling the

BWindow version of `MessageReceived()` is important because it handles all the messages that are ignored by the version that we have written.

Understanding how messaging works in Haiku is best learned in code, so we'll look at a second example, very much similar to the one from the last lesson, but which expands on what we know. We will create a window with a button. Clicking the button will change the title of the window to show the number of times the button has been clicked since the program was started. First, let's look at our window class. All of the code here can be found in the file 15ClickMe.zip, but it would still be best to manually type out all of this code to increase your familiarity with it.

MainWindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>

class MainWindow : public BWindow
{
public:
				MainWindow(void);

	// We are implementing the virtual BWindow method MessageReceived so that we
	// can handle the message that the button will send to the window
	void		MessageReceived(BMessage *msg);

private:
	// This property will hold the number of times the button has been clicked.
	int32 fCount;
};

#endif
```

MainWindow.cpp

```
#include "MainWindow.h"

// Button.h adds the class definition for the BButton control
#include <Button.h>

// The BView class is the generic class for creating controls and drawing things
// inside a window
#include <View.h>

// The BString class is a phenomenally useful class which eliminates almost all
// hassle associated with manipulating strings.
#include <String.h>

// This defines the identifier for the message that our button will send. The
// letters inside the single quotes are translated into an integer. The value for
// M_BUTTON_CLICKED is arbitrary, so as long as it's unique, it's not too
// important what it is. Note that we could use a #define for the message
// constant, but using an enum is the better way to go.
enum
{
	M_BUTTON_CLICKED = 'btcl'
```

```
};

MainWindow::MainWindow(void)
	:	BWindow(BRect(100,100,300,200),"ClickMe",B_TITLED_WINDOW,
			B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE),
		fCount(0)
{
	// Create a button in pretty much the same way that we did the label in
	// the last lesson. The BRect() call inside the BButton constructor is a
	// quick shortcut that eliminates having to create a variable.
	BButton *button = new BButton(BRect(10,10,11,11),"button","Click Me!",
						new BMessage(M_BUTTON_CLICKED));

	// Like with last lesson's label, make the button choose how big it should
	// be.
	button->ResizeToPreferred();

	// Add our button to the window
	AddChild(button);
}


void
MainWindow::MessageReceived(BMessage *msg)
{
	// The way that BMessages are identified is by the public property 'what'.
	switch (msg->what)
	{
		// If the message was the one sent to the window by the button
		case M_BUTTON_CLICKED:
		{
			fCount++;

			BString labelString("Clicks: ");

			// This converts fCount to a string and appends it to the end of
			// labelString. More on this next lesson.
			labelString << fCount;

			// Set the window's title to the new string we've made
			SetTitle(labelString.String());
			break;
		}
		default:
		{
			// If the message doesn't match one of the ones we explicitly
			// define, it must be some sort of system message, so we will
			// call the BWindow version of MessageReceived() so that it can
			// handle them. THIS IS REQUIRED if you want your window to act
			// the way that you expect it to.
			BWindow::MessageReceived(msg);
			break;
		}
	}
}
```

The main part of this program centers around the `M_BUTTON_CLICKED` case. When our button is added to the window, it sets the window as the target for its messages so that every time the button is clicked the window will receive a `M_BUTTON_CLICKED` message. When the window receives the button's message, it increments the member variable `fCount` and uses it to generate a new title.

Creating the title isn't difficult, especially if we use the `BString` class. The C way of doing it would be by allocating a string big enough to hold the title and then using `sprintf()`, but `BString` was designed to make working with strings in C++ a lot easier. Memory allocation is handled for us, and there are methods which combine strings, return the length, and much more. The `labelString << fCount` converts `fCount` into a string and tacks it onto the end of the string held by `labelString`.

The rest of the code kept in App.h and App.cpp is almost exactly the same as it was in the last lesson. The main difference is that App.cpp includes MainWindow.h. By including it, we have the definition for the MainWindow class and we can allocate and show one.

Going Further

Here are some possible changes you might like to explore to make this program do more. I would encourage you to try some or all of these changes. Experimentation leads to many "Aha!" moments and better programming.

- Change the numbers in the `BRect()` used to create the button and disable the `ResizeToPreferred()` call to make the button really, really big – almost as big as the window itself.
- Move the button to one of the window's corners
- Add a second button which sends a `B_QUIT_REQUESTED` message to the window to make it close.
- Create several buttons which move the window. (Hint: use a different message ID for each, and call `BWindow's MoveBy()` method in `MessageReceived()`)