Chapter 8. FXRuby's Message-Target System

Background

One of the biggest flaws with earlier releases of FXRuby was its strict reproduction of FOX's process for mapping GUI events (messages) to instance methods (handlers). That process involved four distinct steps:

1. Initializing a *message identifier*, an integer that helps to disambiguate the sender of the message and/or its purpose.
2. Mapping a specific message type and identifier to an instance method for the message target object.
3. Implementing the actual handler method in the message target.
4. Registering the message target and message identifier with the widget that's going to send the messages.

So, for example, let's say you wanted to create a button widget that, when pressed, prints the string "Ouch!" to the terminal. In the old scheme of things, you'd need to identify some object to act as the target for any messages generated by this button. To keep things simple, let's say that the application's main window (*mainWindow*) is designated as the target for the button. We'll need to generate a unique identifier associated with the button:

```
class MyMainWindow < FXMainWindow

  include Responder

  ID_BUTTON = FXMainWindow::ID_LAST

  ... other stuff ...
end
```

Next, you'd want to specify the mapping for a specific message type to the target's instance method that handles that message:

```
FXMAPFUNC(SEL_COMMAND, MyMainWindow::ID_BUTTON, 'onCmdButton')
```

Finally, you'd need to implement the instance method (`onCmdButton`) named in the call to `FXMAPFUNC`:

```
def onCmdButton(sender, sel, ptr)
  puts "Ouch!"
end
```

The last step is to tell the button who it's message target is, and which message identifier to use when sending messages to that target:

```
aButton = FXButton.new(parent, "Push Me", nil, mainWindow, ID_BUTTON)
```

This was an extremely tedious process, especially for programmers who are used to Ruby/Tk's or Ruby/GTK's approach for connecting signals (events) to blocks that handle the signal. After some discussions at RubyConf 2001 and subsequent discussions on the Ruby newsgroup, a new model was proposed and hashed out on the RubyGarden Wiki. This new model was introduced with the FXRuby-0.99.179 release.

Event Model

FXRuby implements a new, simplified approach to this built on top of the old model. It more or less mimics the syntax used in Ruby/GTK; you can attach a message handler block to a widget using a new `connect` instance method, e.g.

```
aButton = FXButton.new(parent, "Push Me")
aButton.connect(SEL_COMMAND) { |sender, sel, ptr|
  puts "Ouch!"
}
```

Alternate forms of the `FXObject#connect` method can take either a `Method` or `Proc` instance as a second argument (i.e. instead of attaching a block), e.g.

```
def push(sender, sel, ptr)
  puts "Ouch!"
end

aButton = FXButton.new(parent, "Push Me")
aButton.connect(SEL_COMMAND, method(:push))
```

It works by creating a special target object (behind the scenes) that stands-in as the message target for your widget and passes off incoming messages to the appropriate block. The single argument to `connect` is the FOX message type you're handling (e.g. `SEL_COMMAND`, `SEL_CHANGED`, etc.) The three arguments to the block are the same as those for regular FOX message handler methods, namely, the sender object, the message type and identifier and the message data. And of course, for simple handlers like this one, you can just leave the arguments off altogether:

```
aButton = FXButton.new(parent, "Push Me")
aButton.connect(SEL_COMMAND) { puts "Ouch!" }
```

Timers

Timers are scheduled by calling `FXApp#addTimeout`. There are three different forms of `addTimeout`, but the first argument to each is the timeout interval in milliseconds. The most primitive version of this method takes two additional arguments to specify the target object and message identifier for the object that will handle the timeout event:

```
aTimer = getApp().addTimeout(1000, timeoutHandlerObj, ID_TIMER)
```

The second form takes either a `Method` or `Proc` instance as its second argument, e.g.

```
aTimer = getApp().addTimeout(1000, method(:timeoutHandlerMethod))
```

The last form uses a code block as the handler for the timeout event:

```
aTimer = getApp().addTimeout(1000) { |sender, sel, ptr|
  # handle this timeout event
}
```

Chores

Chores are scheduled by calling `FXApp#addChore`. There are three different forms of `addChore`; the most primitive version requires two arguments to specify the target object and message identifier for the object that will handle the chore event:

```
aChore = getApp().addChore(choreHandlerObj, ID_CHORE)
```

The second form takes either a `Method` or `Proc` instance as its single argument, e.g.

```
aChore = getApp().addChore(method(:choreHandlerMethod))
```

The last form uses a code block as the handler for the chore:

```
aChore = getApp().addChore { |sender, sel, ptr|
  # handle this chore
}
```

Signals

Operating system signal handlers are designated by calling `FXApp#addSignal`. There are three different forms of `addSignal`, but the first argument to each is the signal name (e.g. "SIGINT") or number. Each version also has two optional arguments (which should come at the end of the list) to specify *`immediate`* and *`flags`*. The most primitive version of this method takes two additional arguments to specify the target object and message identifier for the object that will handle this operating system signal:

```
aSignal = getApp().addSignal("SIGINT", signalHandlerObj, ID_SIGINT)
```

The second form takes either a `Method` or `Proc` instance as its second argument, e.g.

```
aSignal = getApp().addSignal("SIGINT", method(:signalHandlerMethod))
```

The last form uses a code block as the handler for the signal:

```
aSignal = getApp().addSignal("SIGINT") { |sender, sel, ptr|
  # handle this signal
}
```

Input Events

Input event handlers are designated by calling `FXApp#addInput`. There are three different forms of `addInput`, but the first two arguments to each are the file object (including sockets) and the mode flag (some combination of `INPUT_READ`, `INPUT_WRITE` and `INPUT_EXCEPT`). The most primitive version of this method takes two additional arguments to specify the target object and message identifier for the object that will handle this input event:

```
getApp().addInput(aFile, INPUT_READ, inputHandlerObj, ID_INPUT)
```

The second form takes either a `Method` or `Proc` instance as its third argument, e.g.

```
getApp().addInput(aSocket, INPUT_READ|INPUT_EXCEPT, method(:inputHandlerMethod))
```

The last form uses a code block as the handler for the input event:

```
getApp().addInput(aFile, INPUT_WRITE|INPUT_EXCEPT) { |sender, sel, ptr|
  # handle this input
}
```

This API is a little different from the other cases. For example, timeout events always send the same message type (`SEL_TIMEOUT`) to their message target, so you just have a single handler method (or block) to handle the timeout. In contrast, input sources (e.g. pipes or sockets) can generate three different FOX messages, `SEL_IO_READ`, `SEL_IO_WRITE` and `SEL_IO_EXCEPTION`, depending on what happens, so your handler method (block) needs to check the message type, e.g.

```
getApp().addInput(socket, INPUT_READ|INPUT_WRITE) { |sender, sel, ptr|
  case SELTYPE(sel)
    when SEL_IO_READ
      # handle read event
    when SEL_IO_WRITE
      # handle write event
  end
}
```