

# The Basics of Replacing the R Event Loop

Duncan Temple Lang  
Bell Labs  
duncan@research.bell-labs.com

October 9, 2002

The idea behind this package is to allow us to use an alternative event loop to control R and manage how events and input are handled. This makes it easier to predict and understand what happens when we use external toolkits within R. It is also a necessary step to allow us to use R within other applications and get the correct behavior in both stand-alone and embedded R.

Let's take a look at a simple example of what this package achieves for us. Let's use tool-tips from the RGtk package<sup>1</sup>. To do this, we create a simple little function that creates a button inside a window and associates a tool-tip string (`This is a tooltip for the Button`) with that button.

```
1a < 1a>≡
    button <-
    function()
    {
      require(RGtk)
      win <- gtkWindow(show = FALSE)
      win$SetTitle("Tooltips")
      tips <- gtkTooltips()

      b <- gtkToggleButton("Button")
      win$Add(b)
      tips$SetTip(b, "This is a tooltip for the Button")

      tips$Enable()
      win$Show()
    }
```

Now, we can call this function and the button will appear. Move the mouse over the button and leave it there for a second or two.

```
1b < 1a>+≡
    button()
```

---

<sup>1</sup>The Gtk event loop can be used within this package without using RGtk. We are only using RGtk as an easy way to create Gtk widgets.

We would expect to see the tool-tip appear over the button in a little window of its own. However, if you are running R from the regular console (i.e. without the `-gui=gnome` flag), no tool-tip window will appear. There is a very good (but technical) reason for this. Essentially, R doesn't know anything about our tool-tip and the Gtk event loop that would display it for us is not active. Instead, the way we get any of the Gtk events is by grafting them onto R's event loop. That also explains the slight sluggishness in the response from the Gtk windows and why we sometimes have to move the mouse simply to get the windows to display their contents when they first appear or are updated.

But we can fix this situation using the `REventLoop` package. Let's load that package and use its `runEventLoop()`.

```
2a < 1a)+≡
    library(REventLoop)
    runEventLoop()
```

Now try leaving the mouse over the button for a few seconds. Assuming all is installed correctly, you will see the tool-tip window. So now we seem to have got the Gtk event loop working. What about the R prompt – can we still type commands at the R prompt and have them evaluated in the same way? Yes, the command line is still active. When the `runEventLoop()` returned, a new prompt was displayed and R is ready for new input. Try it and see:

```
2b < 1a)+≡
    mean(rnorm(100))
```

So, what we have done is somehow connected the R and Gtk event loops in a richer way than the `RGtk` does. What is really happening is that we are running the Gtk event loop and have persuaded it to listen for input on the console (or standard input). Gtk events are handled in the usual way, including the timer events and idle/background tasks. At the same time as dispatching these events, the Gtk event loop is looking for user input on the standard input connection for R and when it finds it, it passes the characters to R to process in its own way.

If you have `Rggobi` installed, you can use its cycling and tour facilities by using the `runEventLoop()` function. To test this, create a `GGobi` instance

```
2c < 1a)+≡
    library(Rggobi)
    ggobi(system.file("data", "flea.xml", package="Rggobi"))
```

Now, activate the plot-cycling by clicking on the checkbox named `Cycle` in the `GGobi` control panel. If you did this in the standard R event loop, not much would happen. As you moved the mouse, you might see the plot change. Using the Gtk event loop via `runEventLoop()`, the cycling should behave as it does in the stand-alone `GGobi`, i.e. cycling through the different pairs of variables.

So all this sounds good. What have we given up? Well, just create a new graphics device by calling `plot()`.

```
2d < 1a)+≡
    plot(1:10)
```

Now resize the graphics window and you'll notice that things don't work so well. However, if you have the `gtkDevice`, we can use the `gtk()` device and that will keep its display up-to-date as we resize it, etc.

```
2e < 1a)+≡
    library(gtkDevice)
    gtk()
    plot(1:10)
```

Why does this work? Because the `resize`, `expose`, etc. events are being dispatched by the Gtk event loop. When we created an X11 graphics device, nobody knew to handle its events. The R event loop is not running since it has been replaced by the Gtk event loop. And the Gtk event loop hasn't been told to pay attention to that source of events, i.e. the X11 events.

We should note that if we reversed the order in which we ran the new event loop and create the X11 device, things would work correctly. In other words, creating the X11 device and then calling `runEventLoop()` would result in a responsive X11 device that would update when exposed and resized.

Why does the order matter, you may ask? It is because when we first create the new event loop, we copy the existing event sources registered with R to the new event loop. This means that we tell Gtk to watch for events on the X11 connection and use the C routine that R uses to handle these events. So we can see how it is possible to transfer event handling from one loop to another. This is a useful mechanism that we can exploit to dynamically switch between different event loops as it suits us.

We can also intercept the registration of event handlers and add them to the current event loop. It would be best to do this directly in R. We can do this by integrating this abstract event loop mechanism directly into R, and not having it in a separate package. More information about this is given in [EventLoop.pdf](#). Since we have access to the Gtk event loop, we can use it also to implement idle functions and timed actions. Let's use it to sleep for 800 milli-seconds. While we do this, all the GUI windows will remain active and tool-tips, etc. will still appear

```
< 1a)+≡  
  Sys.msleep(800)
```

In addition to the continued responsiveness of the GUI, we also have finer resolution of how long we sleep for. The duration is in units of milli-seconds, not seconds.

Let's also look at how we can use nested event loops. Suppose we use the Gtk file dialog to ask the user for the name of a file. If we use this directly, when we create and show it, our program will continue. However, we would like to spin our wheels here and wait until the user has selected a file and clicked the Ok button. Obviously, if we use the regular sleep, we can stop R from executing other commands, but we will also stop it updating the GUI and handling the very click on the Ok button we are looking for. Instead, we want to allow *all* the events to be handled as usual, but to not continue executing the R commands until the Ok button is clicked. So how do we do this? We run a second or nested event loop, and we arrange for the callback on the Ok button to exit this event loop. What this will do is block R by calling the event loop. This call doesn't return until the event loop finishes. However, the events and the R callbacks will still be processed.

We'll define a function – *selectFileDialog()* – which implements this functionality. It creates the `Gtk-FileSelection` window and starts the secondary or local event loop. Before it enters that event loop, it registers callbacks with the Ok and Cancel buttons. Both callbacks call *quitEventLoop()* to exit the nested event loop. The callback for the Ok button also retrieves the selected file and stores it in a variable accessible to the original *selectFileDialog()* function (using a closure).

```
4a < 1a)+≡
    selectFileDialog <-
    function(title = "Select a file")
    {
        require(RGtk)
        require(REventLoop)
        d <- gtkFileSelection(title)

        # The value selected by the user.
        fileName <- NULL
        ok <- function(b, ...) {
            quitEventLoop()
            fileName <- d$GetFilename()
        }
        cancel <- function(b, ...) {
            quitEventLoop()
        }

        d[["OkButton"]]$AddCallback("clicked", ok)
        d[["CancelButton"]]$AddCallback("clicked", cancel)

        runEventLoop(FALSE) # don't emit a prompt

        d$Destroy()
        return(fileName)
    }
```

We can call this function now and see how it behaves. Before clicking on either the Ok or Cancel buttons, check to see that the tooltip for the *button()* function still works while the dialog is active. (Rerun the *button()* function before if you don't still have the button window available.)

```
4b < 1a)+≡
    x <- selectFileDialog()
    print(x)
```

Notice that unlike other dialogs, we don't have to destroy the dialog window in the callback functions. Since we know we will return to the `selectFileDialog()` and continue to evaluate the code after the call to `runEventLoop()`, but only after one of the callbacks has been invoked, we can put the call to `gtkObjectDestroy()` after that.

An important fact to note is that by using the Gtk event loop, other applications that we embed in R (e.g. GGobi <http://www.ggobi.org>) which use the Gtk event loop features (e.g. GGobi's touring mechanism which uses the timers and idle tasks) will work.

There are ways to get this behavior using only the regular R event loop. Indeed the `tcltk` package does this. However, it is inefficient as it requires *polling* the Tk event sources. Polling involves consuming cycles by asking whether there is anything to do. Event loops normally sleep until there is something to do. So this hybrid polling approach wastes resources by repeatedly checking with Tk whether it has anything to do. If the checking is done less frequently (to avoid wasting cycles if there is nothing to do), the responsiveness is jeopardized and will generally feel sluggish.

Using the native event loop with R input as sub-events is generally more efficient. This is because we will generally only use one toolkit – either Gtk or Tcl/Tk. If we use its event loop and allow it detect R events, then it will sleep unless there is nothing to do.

Also, what we have said about using the Gtk event loop applies equally well to the Tk event loop. And we expect that there will be no significant problem in applying the same framework to Qt (the Trolltech toolkit underlying KDE), and Carbon (the Mac OS X GUI toolkit).

To see how things work for Tk, let's start by making it the registered event loop. For simplicity, it is probably best to restart R. We haven't spent much time making this package robust since the details will likely change if/when we incorporate this event loop abstraction directly into the R source.

```
5a < 1a)+≡
    library(REventLoop)
```

We register the C variable `R_TclEventLoop()` as the default event loop using the `eventLoop()` function. There is a variety of different forms of this generic function but they all basically find the associated C symbol and register it with the `REventLoop` code.

```
5b < 1a)+≡
    eventLoop("R_TclEventLoop")
```

Now, we can use it as we did the Gtk event loop. We merely call `runEventLoop()` and that starts it running.

```
5c < 1a)+≡
    runEventLoop()
```

Now the Tk event loop is in charge. To verify this, we have provided a way to execute Tcl code by specifying the name of a file. Sample code modified from the `tcltk` package's demos is available in the file `plot.tcl` in the `examples/` directory of the `REventLoop` package. We run this via the command

```
5d < 1a)+≡
    .C("R_testTk", system.file("examples", "plot.tcl", package="REventLoop"))
```

Now you can move the mouse over a point and it will turn red. You can also drag the points around. (Note that there is no regression line drawn in this version.)

One might ask "So what? We can do this with the `tcltk` package already!" And that is perfectly correct. But if you take a look at the output of `search()`, you will notice that the `tcltk` package is not loaded. Nor did we need the `RGtk` package when using the `Gtk` event loop. This is a general mechanism to use any event loop without having to interface entirely to that system.

It is possible to mix the `Tk` and `Gtk` event loops. For example, we can run the `Gtk` event loop using `runEventLoop()` and then instruct it to take care of dispatching `Tk` events also.

```
6a < 1a)+≡
    runEventLoop()
    .Call("R_addTclTkToGtk")

    .C("R_testTk", system.file("examples", "plot.tcl", package="REventLoop"))

    button()
```

This now handles `R`, `Gtk` and `Tk` events. If we were running the `Tk` event loop, we could call `R_addGtkToTclTk()` and we would get the equivalent effect. In this example, we will run a `Tcl/Tk` demo modified from the `tcltk` to run as stand-alone which displays a GUI for selecting and controlling a density estimator and a graphics device displaying the smoother density for a randomly generated dataset. Next, we specify and run the `Tk` event loop and then add the `Gtk` event loop to it. And finally, we run some `Gtk` code: the `button()` example from above and create a `GGobi` instance. We should check that the slider and checkboxes work in the `Tk` GUI and that the tooltips and `GGobi` cycle functionality work in the `Gtk` displays.

```
6b < 1a)+≡
    library(REventLoop)
    library(tcltk)
    source(system.file("examples", "tk.S", package = "REventLoop"))

    eventLoop("R_TclEventLoop")
    runEventLoop()

    .Call("R_addGtkToTclTk")

    button()

    library(Rggobi)
    ggobi(system.file("data", "flea.xml", package = "Rggobi"))
```

A simpler example uses the same code as before.

```
6c < 1a)+≡
    library(REventLoop)

    eventLoop("R_TclEventLoop")
    runEventLoop()

    .Call("R_addGtkToTclTk")
    button()

    .C("R_testTk", system.file("examples", "plot.tcl", package="REventLoop"))
```

Both of these routines are implemented in terms of the generic event loop interface, using `R_localAddIdle()` and specify an action routine that calls the non-blocking single event processing routine.

While we can interleave these two event loops in this way, it is not without its problems. Firstly, if we have no idle time in Gtk, say, the Tk events won't get handled. Similarly, if we pass control to the idle handler that checks the Tk events and it doesn't return for a significant time because there are a lot of Tk events, Gtk will be starved. So it is possible to starve either system. Neither of these situations is very common, so it will probably be okay for most situations, but will be difficult to diagnose and cure if it does happen. We'd like a system that was simpler and more robust. Also, this hybrid configuration is inefficient. We are back to polling the secondary system whenever we execute the idle action.

We would like to avoid this polling. A desirable approach is to provide an abstraction for the registration of input sources, idle tasks and timer actions. Rather than using `gtkAddIdle()` and `gtkAddTimeout()` in Gtk, or S functions that call `after/vwait` and `after idle` in Tcl/Tk, it would be better to catch these on the S side of the interfaces and to provide implementations that use the abstract event loop's methods to register the S functions or C routines. This makes the interface uniform across toolkits and allows developers to use these concepts without having to check which event loop is being used. This makes it easier to introduce different event loops such as Qt, Carbon, etc. without having to change code. Perhaps importantly, it also means that we don't have to use `tcltk`, `RGtk` or any of the large packages that provide an interface to all of the GUI system just to get access to the event loop.

The functions `addTimer()` and `addIdle()` in this package provide this uniform, event-loop-independent manner for specifying timer and idle actions respectively. Both functions accept either a function, an expression or call (i.e. a language object) as the action. One can also parameterize a function action by specifying a value for the `data` argument. This value is stored with the callback and passed to the function as its first and only argument. This allows us to use a single function for several actions and have each behave differently based on the value it is given.

`addTimer()` takes the number of milli-seconds it should wait before calling the action. These examples illustrate the different types of actions one can use and how to specify an argument to a function action.

< 1a)+≡

```

addTimer(500, expression(cat("Timer running once\n")))

addTimer(500, substitute(cat(msg), list(msg = "Timer running once\n")))

addTimer(500, quote(cat("Timer running once\n")))

addTimer(500, quote({cat("Timer running incessantly\n"); TRUE}))

addTimer(500, function() {
    cat("Timer running incessantly from a function\n")
    TRUE # FALSE here for just once!
})

# Use a function but give it an argument
# from the callback. The string is passed to
# cat() as its first (and only) argument.
addTimer(500, cat, "Timer running once\n")

# Use a closure to call
counter <-
function(ctr = 0) {
    function() {
        ctr ─ ctr + 1
        cat("Count:", ctr, "\n")
    }
}

```

```

        TRUE
    }
}
addTimer(500, counter())

```

Each of these examples can be used for idle actions. Merely call *addIdle()* and discard the interval argument. Again, the result of calling the action should be a logical value. If it is T, the idle action is rescheduled. Otherwise, it is discarded.

Of course, we also need to be able to unregister these types of actions. The API for the event loop supports this. We will implement the S interface as demand increases! We will also make it possible to register C routines from S in the same way S functions are given as actions. Also, the API supports registering new event sources from connections (i.e. file descriptors). We will provide an S interface for this that follows the *setReader()* mechanism in S4, but this will wait until we extend the connection mechanism

Providing a centralized S-level interface to these concepts means that S code can work independently of the choice of active event loop. However, C code that uses event loop facilities such as GGobi will still require the particular event loop to be active (either as the primary event handler or registered with the primary handler using a routine such as *R\_addGtkToTclTk()*). We can overcome this by providing a general interface for C code to add event sources to the generic event loop. The Tcl/Tk event loop is setup to do this. We can implement our own version of the Tcl *monitor* object. For Gtk, we can provide our own implementations of *gtk\_input\_add()*, *gtk\_idle\_add()* and *gtk\_timeout\_add()*. Depending on the linking options, our versions rather than the standard ones in the Gtk libraries will be used and ours will communicate with the general event loop.

Currently, we a slightly non-standard event loop in R. For good historical reasons, it is heavily oriented toward getting input from the console. The loop consists of repeated calls to *Rf\_ReplIteration()*. Other event sources such as the X11 connection, etc. are handled as a sub-part of *R\_ReadConsole()*. It is more symmetric and flexible to have an event loop to which we can add different input sources. It is clear where and how to replace the event loop and it makes the semantics of copying the event loop contents or charges to another loop. It also removes the need to have an input source within the C code. Instead, it is useful to move this to S code and use the concept of a reader introduced in S4 to handle input, be it from the console, file or socket. (We will introduce readers by extending the connection class. I have some code that does this but will wait to introduce the extensible *SEXP* classes and then move the connections to those.) As we experiment with different GUIs, etc. we will probably find that we want to be able to by-pass the console entirely and use different event sources.

So there is good reason to introduce these ideas directly into the R source. What would this involve? The basic approach is to replace *R\_ReplConsole()* with the routine *R\_mainLoop()* in the *REventLoop* package.

Before we add this to R, it would be good to get feedback about this package. Please report whether it works successfully or let me know of any problems. Also, I am looking for comments about how this can be extended to support the native event lops on both Windows and Mac OS X. Please mail [duncan@research.bell-labs.com](mailto:duncan@research.bell-labs.com) with any thoughts or remarks.

< 1a>