

A Dynamic, run-time Specifiable Event Loop for R

Duncan Temple Lang

October 9, 2002

Abstract

This is a note on the R event loop and a suggestion for a more general framework which allows R to be driven by other event loops such as Gtk, and Tcl/Tk, Qt, Carbon and CORBA, while still providing its existing functionality. Essentially, this allows us to replace the standard R event loop with another, more standard one and add the R event sources to that event loop. This aids greatly in embedding R in other applications and also makes integration of these other event-driven environments into R both easier and more responsive. We can also use these event loops to provide the basic mechanism for asynchronous events to R, including timeouts and readers on file descriptors without having to reimplement them ourselves.

In this package, we provide implementations of a few different ideas.

1. allow either the Tcl/Tk or Gtk event loop to be run “on top” of the R event loop;
2. make the readline mechanism in R a regular event driven callback rather than building any knowledge of it into the R event loop.
3. process Tcl/Tk events as part of the Gtk event loop and vice-versa. (This has temporarily ceased working. This may be related to changes in the tcltk package and some internal changes to the R event loop or it may be a coincidence in the timing.)
4. an abstract data structure for a generic event loop with implementations for both Gtk and Tcl/Tk.

We outline a mechanism by which this general event loop structure can be defined and integrated directly into R.

A specific example of what this mechanism provides is the integration of GGobi, its background tour motion, tooltips from Gtk applications, running Tcl GUIs within this environment as well as dynamic resizing graphics devices all while R is “blocked” in either a sleep or a call to, e.g. *scan()*.

This mechanism currently applies only to Unix-like environments. Soon we will explore integrating this with Windows and OS X.

1 Gtk+

We start by looking at how we can use the Gtk+ event loop to run R. Many of the basic steps and concepts map directly to Tcl/Tk and other event loops. The least intrusive way to do this (in terms of modified code in the R tree) involves stacking or running the Gtk+ event loop on top of R’s existing event loop. This essentially involves starting R in the usual way and invoking a call to a C routine that itself calls the event loop `gtk_main()`. This will block indefinitely, processing all event sources known to the Gtk event loop. For this to work transparently, we must setup the same user input reader that is running in R and make it known to the Gtk+ event loop. Before we do this, we can just supply a readline-based input mechanism. To do this, we extract the fundamental readline mechanism used in R and put it in `gtkReader.c`. Then, we register the standard input file descriptor with the Gtk+ event loop (using `gtk_input_add_full()`) and place an appropriate callback to get readline to handle the input:

```
gtk_input_add_full(fileno(stdin), GDK_INPUT_READ, gtkReadlineHandler,  
                  NULL, NULL, NULL);
```

In addition to the standard input, we must also ensure that other event sources currently being monitored by R are also registered with the Gtk+ event loop. To do this, we loop over the `InputHandler` linked list and register them using

```
gtk_input_add_full(handler->fileDescriptor, GDK_INPUT_READ, handler->handler,  
                  NULL, handler->userData, NULL);
```

Now, we are almost ready to start handling events. However, we must ensure that any errors in top-level evaluations return us to this new event loop. For this, we create a top-level context handler using R's internal mechanism.

`setReader()` and `setMonitor()` now need to go to this event loop. `Rf_AddInputHandler()` needs to add to the existing event loop, not just R's. For this, we need pointers to routines that parameterize the event loop.

1.1 Other Interactive Input

Browser and scan are two examples of functions that can query the user for input. Sleep also needs to be handled carefully so as to allow other events but block execution of the evaluator and the R engine at the current spot. In each of these cases, a recursive/nested event loop can be used. We arrange to use the currently registered event loop mechanism but with a callback registered which terminates that loop when the specific event of interest occurs.

`parse()` and other functions call `Rstd_ReadConsole()` which is extended to handle this type of event loop. (See the routine in `gtkReader.c` of the same name.)

1.2 Terminating the event loop & the prompt

When `gtk_main_quit()` is called within this event loop (and not a nested one), we terminate this event loop and fall back to the top-level R event loop.

Unfortunately, we get an extra emission of the prompt after calling `gtk_main_quit()`. For the moment, we insist that the interactive user call the C routine `R_gtk_main_quit()` which sets a flag to indicate the prompt should not be emitted. In the future, we would like to be able to trap this more elegantly. But since it doesn't matter if `gtk_main_quit()` is called non-interactively, i.e. as part of an event handler, etc. the point is of marginal relevance.

2 A General Event Loop

In order to be able to handle different event loops, we need to identify the different components of the different event loops we may use. (In the future we may need to extend this, and again a good object system even in C would help greatly.)

The essential methods that we need are as follows (and listed in the `R_EventLoop` definition in `Reventloop.h` in this package).

init A way to initialize the system such as `gtk_init()` or a routine that calls both `Tcl_init()` and `Tk_Init()`.

main This is a routine that is a blocking event loop. It processes all events until it is explicitly terminated by a call to the `quit()` method. In Gtk, this is `gtk_main()`. In Tcl/Tk, it is simply

```
while(Tcl_DoOneEvent(TCL_ALL_EVENTS)) {
    if(tclEventLoopQuit) {
        tclEventLoopQuit = FALSE;
        break;
    }
}
```

quit This is the routine that terminates the *current* blocking event loop.

non-blocking iteration We also provide a routine for processing a single event from the event queue iff there is one available. This routine should return immediately (i.e. be non-blocking) if there is no event in the queue.

add/remove timeout These allow one to register and unregister an event (or specifically a call to a routine) scheduled at a particular time interval from now. Since Tcl/Tk returns a pointer to a struct, while Gtk returns an integer identifier to allow one to unregister this, we need to use a common base type. We use a `SEXP` and consider its contents as opaque, known only to the event loop system itself.

add/remove idle

add/remove input source

3 Event-Driven Programming

3.1 Sys.sleep

Let's take a look at *Sys.sleep()*. Rather than simply calling the system level `sleep()` (or using a call to `select()` as a time-out), we can use either of Gtk or Tcl/Tk's event loop facilities to specify a timeout for the desired sleep period. If we simply did this and returned, we wouldn't have attained the desired effect. Instead, we want to block the R evaluator until the timeout callback has been invoked. To get this blocking, while still processing all the other event sources in the event loop, we simply create a new sub-event loop. In the case of Gtk, we call `gtk_main()` and this serves exactly that purpose.

In order to exit this event loop at the right time, we arrange to call `gtk_main_quit()` after the specified period. This is easily done by the timeout handler function.

```
gint
Rf_gtkMainQuit(gpointer unused)
{
    gtk_main_quit();
    return(0); /* Don't reset this timeout. */
}

SEXP
R_sleep(SEXP interval)
{
    gtk_timeout_add(asInteger(interval), Rf_gtkMainQuit, NULL);
    gtk_main();
    return(R_NilValue);
}
```

3.2 locator

The function *locator()* is used to interactively identify positions in a graphics device. Currently, this is left to each graphics device to implement and each is responsible for running a sub-event loop and filtering out the mouse click events itself. While most process all events in their known event queue, their definition of the event queue is quite narrow. For instance, the X11 device processes both X and R event sources. Similarly, the Gtk device driver handles Gtk and R events. (What happens with tkrplot? It is not a device of this type, it seems from a very, very quick glance.) Under this model, an X11 driver will not process Gtk idle or timeout events. So tooltips and other things won't work while we are in a call to *locator()* on a different device type from the GUI.

How can we simplify this. Clearly, if we had a global event loop object which had a method for handling all events, we could call it. This is clearly desirable and would solve the issue of handling the diverse event sources from different "systems" (e.g. Tcl/Tk, X and Gtk). But the way the *locator()* event loops are structured requires that they can access the events directly in the queue and look at the relevant ones in detail. Specifically, they need to look at each "Button Press" event and examine the x and y values it contains.

Instead, we can create a callback specifically for button events. When *locator()* is called, we register this callback for these events and enter a new level of the basic event loop. The user-level data for this callback contains information about the number of entries to read (possibly just a single value) and the arrays into which to put the (x, y) pairs. When this callback determines that it has read enough entries (as specified by the n argument to *locator()*), it breaks out of this event loop (via a call to, e.g. `gtk_main_quit()`) and so *locator()* returns.

This is already the way the Gtk device works and many modal dialogs, etc. So it is a typical and well tested approach to blocking in one call while still handling other asynchronous events. However, the Gtk device again only knows about its event loop, namely Gtk+. As a result, no other event sources will be processed during a call to *locator()*. For example, any Tcl/Tk GUI will become inactive and not even update its display.

4 Integratin with R

Currently, this functionality is provided as a package. This means that when we start the new event loop, we do not actually return from the `.C()` call that performs the initiation. Instead, this emits an identical prompt as the R event loop would and then it takes over. This means that, within the low-level native code, we don't return from the `do_dotCode()`.

In the longer term, we would like to make this more seamless. It should be relatively easy to move the event loop structure into R itself and have the call to initialize the new event loop merely replace the event loop structure with a new collection of routines. The actual loop will continue in the next iteration but using these new routines.

5 Notes

One can tire gdb out when testing this. But I have seen this happen in other circumstances also, so it may not be an inherent property of this setup.

Issues about running two event loops. How do we avoid re-registering the same input/event sources. Do we chain so that we process event loop 1, then 2, etc. until all are finished and then continue. Or do we just run the top-most one in the stack.

Error handling and unregistering events.

Signals.

For more sophisticated features (e.g. some provided by Tcl's event loop such as prepare-to-block initialization of event sources, etc.), one should use the Tcl event loop. In other words, we don't have to provide this in full generality since users can swap in a Tcl event loop and use that. Obviously this isn't as good as handling it in general since there may be a conflict of interest in a session that prefers to use two different loop implementations. For the foreseeable future, this is not a high priority.

Using the R event loop structure, one should be able to create a hybrid loop. Need classes of events ideally, or specifically to target input handlers to a particular loop type.

Use the event loop associated with the toolkit which implements the primary UI element. For example, if we are running the Gnome interface, use Gtk's event loop. If Tcl/Tk is the preferred toolkit we use for creating GUIs, use that.

6 Examples and Tests

6.1 Tcl/Tk as a Gtk event source

Perhaps we can simply add the other event loop's "do one event" routine as an idle handler to the main event loop.

```
library(tcltk)
dyn.load("gtkReader.so")
source("tk.S")
invisible(.C("R_mainLoop"))
.Call("R_addTclTkToGtk")
library(Rggobi)
ggobi("/home/duncan/Projects/ggobi/ggobi-cvs/data/flea.xml")
```

Both systems should be interactive and work correctly.

6.2 GGobi tour and the stand-alone Gtk device's locator

This example shows GGobi running with the

```
dyn.load("gtkReader.so")
invisible(.C("R_mainLoop"))
library(gtkDevice)
```

```

library(Rggobi)
ggobi("/home/duncan/Projects/ggobi/ggobi-cvs/data/flea.xml")

gtk()
plot(rnorm(100))
locator(n = 3)

```

Turning on the tour (manually at present) seems to swamp the events and the end button click events don't get through (?)

7 Readers and Monitors

It is useful to be able to watch for input on different connections and to have an S function be called when input becomes available. This is the concept of a reader in S4 and similar to facilities in numerous other environments. For example, in Gtk these are “input” sources.

Similarly, it is useful to be able to schedule calls to S functions to happen after a certain period of time, or every t time units. In S4, such timed actions are called monitors. In Gtk, these are timeouts.

These two styles of asynchronous events. Rather than adding a facility to R to register them and implement them in the absence of an external event loop (e.g. Tcl/Tk or Gtk), it is simplest to use the interfaces to those external systems to add such handlers. Specifically, one can use *gtkTimeoutAdd()* from the RGtk package. We already handle input sources in the R event loop on Unix, so we can provide an interface to this. Since we have to get the file descriptor associated with an R connection object, this is also reasonable to do in R.

8 Additional Advantages

We can split out the readline functionality from core and allow people to add it on demand. This means that they can recompile just a single package.

The framework makes it easier to add a new event loop engine (e.g. wxWindows, etc.)

9 To Do

X11 device locator. This is tricky because of the lack of callbacks. We can use the centralized event loop global variable rather than *handleEvent()* and that will use the regular event loop. We can also have a variable in the X11 event loop for holding the current XEvent and mimicing the callback mechanism ourselves. Or, we can suspend (or insert earlier in the callback queue) the standard

Tcl/Tk event loop integration. Seems basically okay with the methods. Need to get the signatures accurately defined.

Mixing tcl and GTK. Use idle or timeout handlers to put one on the other is pretty much working. It would be good to add a mechanism that smooths out the differences between the two systems regarding the way timers and idles are automatically re-registered if they return non-zero.

Continuations in the prompt and not keeping the initial text around. (e.g.

```

plot(1:10
)

) This is now fixed.

```

Merge with existing R code. Use *eLoop->addInput* for implementation of *AddInputHandler()*.

In the long run, have an event loop pointer in R's main and use the routines for that.

Generic event loop structure An initial version is in `Reventloop.h` and can be put into `eventloop.h` in `include/R_ext/`.

Orbacus and other CORBA event loops.

error in the `browser()` and `readline` when debug a callback in `Gtk`.

This is a general problem with asynchronous calls to `browser` and anything that uses the standard console reading facilities asynchronously. The basic problem is as follows. We are in the usual input loop awaiting characters typed by the user. Then asynchronously, we enter the `browser` due to a callback that is invoked from the background event loop that is active while waiting for the user input. At this point, we essentially are starting a new `readline` session and it is important that we restore the old one when we complete the `browser`-related one. But unfortunately, we are using global variables and restoring it is not currently being done.

Cleaning up after errors is still an issue that needs investigation.

Error in `scan` means we get two prompts: the one from returning from `scan` and then another from falling through the contexts up to the new event loop's version.

The `GGobi` tour dominates `Tcl/Tk` as an idle task. Perhaps we can put the `Tcl/Tk` event loop as a timeout on the `Gtk` event loop. Same problem with the `gtk device` and `locator` when the tour is running.