Some Notes on R Event Handling

Luke Tierney
Statistics and Actuatial Science
University of Iowa

December 9, 2003

1 Some Things that Do Not Work

Here is a non-exhaustive list of a few issues I know about. If we can come up with solutions that address these then we may be on the right track; if not, . . .

1.1 Gnome

- There is an interrupt button but it isn't much use: it cannot interrupt a running computation. It can interrupt recursive reads.
- The default device is `x11`, but X events are not handled. Nor are other things registered as input handlers. This is probably one reason why `tcktk` special-cases gnome. `locator` on the X11 device cannot be interrupted, even by a `kill SIGINT`.
- `rgl` does not work under gnome because it depends on the registered input handlers being run.

1.2 Tk GUI on X11

- There is no interrupt widget.
- `kill SIGINT` will sometimes work, but other times results in a `Segmentation fault`. These seem to happen when `longjmp`-ing out of `Tcl_DoOneEvent`—we need a more sophisticated approach that allows us to go through the Tcl error return mechanism.
- Closing the window with the window manager close button hangs the process, probably in the Tcl event loop; `SIGINT` causes a segfault.

1.3 Aqua

. . .

2 Outline of Issues

A simplified model is that we have

- a host program that runs a main event loop.
- the main event loop occasionally calls `Rf_eval`
- eval may need to enter a recursive event handling process
  - to check whether the host program has a pending interrupt signal
  - to keep the host program appear responsive

The host program can be R itself, it can be a GUI front end like gnome or Tk GUI, or it can be something like a browser in which R is embedded.

Recursive event handling can occur in three different contexts:

- Non-blocking: periodic checks for interrupt during long computations (interpreted or compiled).
- Blocking: `sleep`, `scan`, `system` calls, socket operations, etc.
- Modal: `locator`, `file.choose`, `data.entry`, `pager` on some systems, etc.

Modal is in some ways in between the other two. Modal should do somewhat less event processing than blocking and perhaps somewhat more than non-blocking. All should allow user interrupts, though most modal interactions seem to have their own cancel mechanisms and handle standard user interrupts, where they work, somewhat inconsistently. None of these should be allowed to run R code since we don't adequately support this sort of concurrency.

For the host application/main loop level there are similar considerations. Once R has returned to the host the host application may

- be running other things and need to give R a little time to handle events on things it knows about
- be idle and need to block until the next R or non-R input arrives

In some cases, like Tk GUI and Duncan TL's examples [2] the host program might be a call from R into a C main event loop routine. This would then shadow any outer host and also need to do some magic about establishing an appropriate top level context.

3 Outline of Strategy

3.1 Recursive Event Handling

All recursive setting need a callback into the host to allow the host program do do whatever processing is appropriate. The host program needs to have available a routine it can call to indicate that a user interrupt has been signaled. The interface might look like

```
typedef void (*R_HostRecursiveEventHandler_t)(void);
void R_RegisterHostRecursiveEventHandler(R_HostRecursiveEventHandler_t);
void R_RecordUserInterrupt(void);
```

The host registers a handler, which R calls in non-blocking situations and after a possible sleep in blocking ones. The handler can record an interrupt if one has been indicated.

For blocking settings, R needs to be able to ask the host how long it can block and whether there are file descriptors R should wait on. This could be handled with an interface like

```
typedef void (*R_HostPrepareToSleep_t)(R_fd_set_t *rwefd, double *time);
void R_RegisterHostPrepareToSleep(R_HostPrepareToSleep_t);
```

The host registers a handler, which R calls before blocking. The `rwedf` argument is a pointer to a structure that the host can fill in with file descriptor information. It may need to look slightly different on Windows and Unix. If the host uses file descriptors, then it can supply all those it cares about to R with some indication that R can wait without a timeout (say a negative time). If the host does not use file descriptors then it will not enter any and supply some polling timeout. R will then do something like a `select` and then call the recursive event handler.

3.2 Top Level Event Handling

The host program needs something analogous for top level event handling but in reverse. The host needs a routine to call to ask R to handle any pending events and return immediately, say

```
void R_HandlePendingEvents(void);
```

If the host wants to block efficiently it will need to ask R how long it can block and if there are any file descriptors it should wait on.

```
void R_PrepareToSleep(R_fd_set_t *rwefd, double *time);
```

If the host is not able to make use of file descriptors then it can pass `NULL` as the `rwefd` argument. R will compute the appropriate timeout value accordingly.

The host top level loop could look like

```
while events are available
    if timeout
        call R_HandlePendingEvents
        call R_PrepareToSleep to update fds and timeout
    else if need R to evaluate something
        call R to execure code
        call R_PrepareToSleep to update fds and timeout
```

Alternatively, the host could use a generic main loop, a timer callback, and a callback for executing R code when needed.

3.3 Package Registration Hooks

Most packages that want to register event handlers should be able to get by with

- a way to register a callback on a file descriptor (where that makes sense), maybe with a timeout
- a way to register a timer callback
- either a way to remove these or a convention that they must re-register themselves after firing.
- some way of indicating whether the callbacks should fire only at top level or also in recursive event handling.

The file descriptor callbacks should only fire if there is activity on the file descriptor. We currently have part of this but it isn't very clean.

`tcltk` on X11 should just need timer callbacks. That is basically what is being used now. In principle it would be nicer to use file descriptors as well, but this is too complex: It would require creating our own notifier and our own Tcl/Tk library with that notifier linked in. If we only use a clean timer scheme I'm not sure if Tcl/Tk really needs special handling on Windows anymore.

`rgl` should just need the file descriptor callback on Unix/X11. On Windows it needs nothing since the window callbacks will get handled by the generic Windows event loop. The same *should* be true of a pure Aqua version if we get one and if I'm reading correctly the way Aqua (or Carbon or whatever) works.

4 Some Variations

The host mechanism can be used by several external event sources (so the name is not ideal). This should work as long as

- R checks all registered event sources before it blocks
- `R_PrepareToSleep` and `R_HandlePendingEvents` check all registered event sources.

If a "host" is going to block or needs to allow other event sources to be checked then it may make sense to provide a means to exclude itself from the callbacks R checks. On the other hand, if this is needed then it is easy enough to implement in a particular "host."

It probably makes sense to modify the host registration mechanism to

- use a single registration procedure
- have the registration mechanism return a token that can be used to unregister
- have the callbacks include client data

This suggests a modified interface something like

```
typedef int R_HostTag_t;
typedef void *R_HostData_t;
typedef void (*R_HostRecursiveEventHandler_t)(R_HostData_t);
typedef void (*R_HostPrepareToSleep_t)(R_fd_set_t *, double *, R_HostData_t);
R_HostTag_t R_RegisterHost(R_HostRecursiveEventHandler_t,
                           R_HostPrepareToSleep_t,
                           R_HostData_t);
void R_UnregisterHost(R_HostTag_t);
```

5 Concurrency Issues

We have to be fairly careful about recursively calling R code from event handlers that are called from R code. In some cases we want the code to behave as if it were called from within the outer R code, for example when the event handler, possibly user-defined, signals an interrupt. In other cases, such as where the event handler runs another large piece of R code, we want it to behave like it is running concurrently but in a different dynamic context (so errors in this handler do not affect the outer R code, for example). We do not have a language to properly express these differences, but we may be able to come up with some simple approaches to get by with. Perhaps a mechanism where there is a fixed set of tokens describing in-line actions, such as "interrupt the current computation," and all other things get put on a queue and run at an appropriate time.

One case where there is currently is with tcltk. If you use the sample code in `tkbutton` to create a window with a button, start a long-running R computation (say `while(TRUE) NULL`, and click the button, then in `Rgui` on Windows the button action happens but on Unix it doesn't. We would want to be consistent here. Given our current support for concurrency (i.e. none) it would be safest to make sure the button does not fire, or at least defer the action until return to top level (defer or ignore is also a question). The user I think perceives these as separate threads of activity, at least if we don't put up some sort of "busy" indicator when the gui becomes inactive.

References

[1] Matthew Flatt. Inside PLT MzScheme. `http://download.plt-scheme.org/doc/insidemz/`, July 2003.

[2] Duncan Temple Lang. The REventLoop package. `http://www.omegahat.org/REventLoop/`, October 2002.

[3] Tcl notifier. `http://www.tcl.tk/man/tcl8.4/TclLib/Notifier.htm`.