# Top-level Task Callbacks in R

Duncan Temple Lang

December 2, 2001

*This is an early version of a description of the new callback mechanism that is invoked at the successful completion of each top-level task.*

The basic idea here is that we want to allow R functions and C routines to be automatically invoked at the end of *each* top-level expression. The end of the expression evaluation is an "event" and we can use it to do numerous things. For example,

**saving state** A question on R-news recently asked how one might arrange to have a call to *save.image()* periodically invoked to reduce the potential for data loss if a crash were to occur subsequently. Rather than being based on time and the current semantics of the event loop, it makes more sense to call *save.image()* at the end of a top-level task. This avoids intermediate computations, calling from within other contexts, etc. and generally has more obvious and meaningful semantics. We can even examine the expression and determine if any assignments were made.

**validation** The methods package introduces the need to validate the contents and structure of objects relative to the definition of the their declared class(es). This can be done when a top-level assignment occurs and can be performed at the end of that task.

**auditing** The S-style auditing mechanism could be implemented optionally by an extension which catches each top-level task and stores the appropriate information.

**transactions** One can treat successful top-level tasks as "transactions" in the relational database sense and propagate changes caused by the expression when it has completed. For example, one can update spreadsheets, commit assignments to tables in remote databases, etc.

**display updates** One can output the user input and result to different rendering engines such as ESS, XML/HTML browsers, a different console for collaborative sessions, an object browser, search path GUI, etc. Again this is an optional extension that is customizable by the user rather than being one of a fixed set of options implemented internally.

This idea of invoking callbacks at the end of each top-level task has been used in the Omegahat interpreter since its inception and promotes a more modular and extensible mechanism for controlling the user's environment. It provides a reasonably easy way to customize the Read-Eval-Print loop (REPL).

We provide an extensible mechanism for calling a collection of C routines and R functions after each top-level task is completed. We describe this interface below.

## 1 The Interface

The basic interface allows one to register a C routine to be invoked at the end of each top-level task. Those programming at the C-level can use this directly. Most people however will use the R-level interface and register R functions instead of C routines. A third level allows one to register a single R function with the C-level code and have it act as the manager or dispatcher for the real R functions to be invoked. We will first describe the basic R interface which registers R functions directly with the C code.

## 1.1 The R Interface

There are two basic functions with which one can add and remove R functions which are to be used as task callbacks that are to be invoked at the end of each top-level expression evaluation. These are *addTaskCallback()* for registering a handler, and *removeTaskCallback()* for removing one.

*addTaskCallback()* takes a function that is to be called after each top-level task has been completed in the R session. By default, the function is called with 4 arguments:

**expression** the S-language expression for the top-level task.

**value** the result of evaluating the expression for top-level task.

**status** a logical value indicating whether the evaluation was successful. In the future, this may be used to indicate whether the handler has been called in the case of an error or not.

**visible** whether the output from the task was printed or not.

Therefore, each function that is used as a task callback must accept at least four arguments.

Closures provide a convenient way to create a function that has access to any additional data that it needs. For example, suppose we want to keep a count of the number of times the callback has been invoked, we might define a function that returns the actual callback function.

```
counter <-
function() {
 ctr <- 0

 function(expr, value, ok, visible) {
   ctr <<- ctr + 1
   cat("Count", ctr, "\n")
   TRUE
 }
}
```

Now, each time we call the function *counter()*, we get back a function that has its own independent count. And we can pass that function to *addTaskCallback()*, e.g.

```
addTaskCallback(counter())
```

In other cases, it is more convenient to not use closures but to have R call the function with an additional argument that gives additional context. In these cases, we can supply a value for the *data* argument of *addTaskCallback()* function. This object is stored by the callback mechanism and is given as a fifth argument to the callback when it is invoked. The *periodic()* function example below is an example of this approach.

Either of the closure or an additional-argument approaches works well. Closures are needed when we want to change the value during the call and see those changes in subsequent calls. User-specified data is convenient when we want a single, perhaps existing, function and just want to change it behaviour based on some additional data – the fifth argument.

The final argument to *addTaskCallback()* is optional and allows one to specify a name by which the callback can be identified. This is stored with the callback and used when we query the list of registered callbacks. The function *getTaskCallbackNames()* returns these names in the order the active callbacks were registered. Also, when we remove a callback using *removeTaskCallback()*, we can give the name of the callback.

If no name is specified in the call to *addTaskCallback()* call, we give it one. This default is the string whose contents are the position in the list of this new callback.

## 1.2 The Return Value of a Callback

In the case of these callback functions, the return value has special significance. It must be a single logical value. If the function returns *TRUE()*, the callback is maintained in the list of callbacks and it will be invoked after the next task is completed. If it is *FALSE()*, then the callback is removed from the list and won't be called again by this mechanism. This provides a convenient way to "retire" a callback, or more specifically, have it retire itself.

## 1.3 Registering Multiple Handlers

There is a slight issue if one wants to add multiple handlers and have them all registered before the first is called. This shouldn't be important and it is typically not good design to have handlers depend on each other. However, one could use successive calls to *addHandler()* and arrange for the handlers to check for some state to allow them to operate. This is very clumsy. A simple way to do it is

```
for(i in list(periodic(), periodic())) {
  addTaskCallback(i)
}
```

or

```
handlers <- list( list(function = periodic(), data = quote(print("ok"))),
                  list(function = periodic(), data = qote(print("now"))))
for(i in handlers) {
  addTaskCallback(i[["function"]], i[["data"]])
}
```

In each case, the for-loop is considered as a single top-level task and so the callbacks will not be invoked until the loop is completed and all of the handlers have been registered. It is pretty trivial to write a function to do this taking an arbitrary number of function arguments and a list of user-level data.

However, there is a much simpler way to do this that uses R a single R function to manage the R-level callbacks. That is the topic of the next (sub) section.

## 1.4 An R Handler Handler

One of the good tests of a model in an interpreted environment is whether a feature developed internally can then be implemented in the interpreted language itself. In this particular case, it is quite simple and useful to implement task callbacks in R itself. The reason it is useful is that we have more control over the list of handlers. We can index the elements by name, deal with garbage collection more readily, etc. Of course we can implement such facilities in C, but the details of linked lists, creating names, supporting optional names, adding features, etc. quickly make the software more complex than is desirable to maintain. In S, however, such list creation and manipulation, indexing and so on is trivial and trivially portable!

The function *taskCallbackManager()* is a function that can be used to create a manager which handles other callbacks. It is a meta-callback. *taskCallbackManager()* returns a list of functions which share state via some variables in their environment. This is a closure. The functions allow one to *add()* and *remove()* functions to and from a list of callbacks. When a top-level task is completed, the managers central callback (*evaluate()*) is called by the C-level mechanism and this, in turn, evaluates each of the callbacks it manages.

The *add()* method provided by the manager is very similar to *addTaskCallback()* but stores the callback in the R manager. It takes a function and optionally a *data* argument which will be passed to the function when it is called after a top-level task. One can also specify a name for the element. This allows us to update and replace existing functions in the list of callbacks and also remove them by name rather than index.

We can show how the *taskCallbackManager()* works with a simple example. Suppose we want to save the session at the end of each top-level task to avoid any chance of losing data. We can define a simple task callback to do this with the following code:

```
saveImage <-
function(expr, value, ok, visible)
{
 cat("Saving the session\n")
 save.image()
 TRUE
}
```

We can then arrange to have this called at each top-level task using *taskCallbackManager()*.

```
> h <- taskCallbackManager()
> h$add(saveImage, name = "autoSave")
[1] "saveImage"
Saving the session
```

We first create the callback manager and it contains no callbacks. Then we add the *saveImage()* callback. This has the side effect of arranging for the callback handler to become active. We can check this by looking at the names of the C-level callback handlers using *getTaskCallbackNames()*.

```
> getTaskCallbackNames()
[1] "R-taskCallbackManager"
Saving the session
```

Notice that calling *getTaskCallbackNames()* caused the session to be saved, i.e. the callback to invoked. And other top-level expressions will do the same.

```
> x <- 100
Saving the session
>
```

We can of course enhance *saveImage()* to check whether the expression was an assignment and only save the session if there are changes to the global environment.

We can now give a more detailed explanation of the properties of *taskCallbackManager()*. The *add()* and *remove()* functions merely manipulate contents of the list of callbacks. The *evaluate()* is the one that does the work. This function is a regular task callback and can be registered using *addTaskCallback()*. This is done automatically when the first function is added to this R-level callback list via the *add()* function. At the completion of each top-level task, this *evaluate()* function is invoked. It then iterates over the elements in the list of callbacks and evaluates them in the same way as the C-level dispatching does, i.e. passing them the same 4 arguments it was called with. Any user-level *data* argument given when registering the function is supplied as the fifth argument in the call to that handler.

Any handler that returns **F** is dropped from the list.

One can use the *callbacks()* function to look at the list of currently registered handlers. For example, we can register two callbacks and then examine the list of callbacks.

```
h <- taskCallbackManager()
h$add(times(), register=FALSE)
h$add(times(6))
names(h$callbacks())
```

The *times()* function is shown below. It is a callback that prints an identifier. When it has been called $n$ times, it removes itself from the list of callbacks. The result of activating this callback is as follows.

```
> addTaskCallback(h$evaluate)
[1] 1
[Task a] 1
[Task a] 1
> 2
[1] 2
[Task a] 2
[Task a] 2
> 3
[1] 3
[Task a] 3
[Task a] 3
Removing 1
> 4
[1] 4
```

```
[Task a] 4
> 5
[1] 5
[Task a] 5
> 6
[1] 6
[Task a] 6
Removing 2
>
```

Note that the *evaluate()* callback is still active. It is called at the end of each top-level task. However, it has no handlers to call so simply returns **T**. It could remove itself by returning **F**. Then when one adds a new function using the *add()* method, it could add itself via a call to *addTaskCallback()*. The difference lies in efficiency versus convenience and simplicity.

Since the code is written in R, it is easy to extend and/or replace the basic functionality. For example, we can add a *suspend()* function which temporarily prohibits the callbacks from being invoked. This simply sets a flag to **T** or **F**. If this flag is set to **T**, then the *evaluate()* function returns immediately and does not invoke the different callbacks.

# 2 Errors and Warnings

Any error that occurs when evaluating an R function registered as a callback will be caught and displayed. However, the callback will be removed from the list of handlers and will not be called again. Also, warnings will be displayed at the end of each callback and will be preceded by a message identifying the callback by name.

## 2.1 The C Interface

The C interface is similar to the R function interface. One registers a callback routine using the `Rf_addTaskCallback()` C routine. This expects five arguments.

1. The address (usually the name) of the routine to call after each task has been completed.
2. A reference to user-level data which is passed to the routine when it is called. This is used to parameterize the C routine, providing additional context with which it can perform its job. For example, this might contain a handle to a spreadsheet or a GUI object.
3. A second C routine that cleans up after the handler is removed from the list of task callbacks. This is called with the user-level data as its only argument.
4. An optional string giving the name to use for the element in the callback list. If this is NULL, the registration routine will create one which is the index of the position into which the callback is being added.
5. An optional integer pointer which, if specified, will contain the index into which the element has been added.

The handler routine (1) is called with 5 arguments that are the C versions of the arguments passed to an R function callback. The expression and value are given as `SEXP` objects and are the actual expression and value used by R (so don't change them!). The success and visible values *Rboolean* variables. Finally, the user-level data is specified as a `void *` and is the value given when registering the callback routine.

The routine can do whatever it wants. It should catch any errors and guarantee that it returns to the caller (rather than doing any long jumps, etc.)

At present, one cannot get at the warnings from the the top-level task as they have already been emitted by R. We may want to change the current setup so that the handling of printing the result and warnings is done via one of these handlers. That would allow one to easily override this and redirect output to a different rendering engine such as a browser, e.g. Netscape/Mozilla when using R in Netscape via the SNetscape package, or a SOAP connection. (Perhaps in 1.5.0!)

A callback function should return either `TRUE` or `FALSE` to indicate whether it should be kept in the callback list or discarded, respectively.

# 3 Warning

Under no circumstances should one of the handlers add or remove entries from the list of handlers. The handlers are intended to be autonomous actions that do not know about each other. They can remove themselves from the list of handlers via their return value. But relying on knowing about the existence and status of other handlers is a poor design and suggests that those handlers. Then if one needs to interact with the other, this can be done in the higher-level handler. In the future, we may allow handlers to be evaluated as background tasks. This is a very clear circumstance in which handlers that interact with each other is potentially disastrous.

# 4 Alternatives

When we move to multiple interpreter instances (i.e. different interpreters running in the same R session, either concurrently or interleaved), we may also use the notion of a task queue. One could arrange to have every user-specified top-level task be followed by a handler task and this would give us the same effect. The current approach is probably simpler and guarantees the ordering. However, we will want to be cautious about introducing numerous ways to specify task handlers and have them invoked. The potential for confusion abounds.

# 5 Examples

The following is a simple example of how things work. The example handlers are not particularly interesting. But they illustrate how one can do different types of computations.

We start by defining a *times()* function which will print a simple string to identify itself each time it is called. The key point of this function is that it removes itself from the handler list after it has been called a particular number of time. This number of calls is given in the call to *times()*. Also, we can give a label that is printed by the function when it is called to distinguish the output from the different instances of the handlers.

```
times <-
function(total = 3, name="a")
{
  ctr <- 1
  function(expr, val, ok, visible)
  {
     cat("[Task ", name, "] ", ctr,"\n", sep="")
     ctr <<- ctr + 1
     return(ctr <= total)
  }
}
```

A second handler example is *periodic()*. This is called after each top-level task, but only does something every *period* calls. This takes an additional argument - *cmd* - that is the value we give when registering the callback. In this case, we will pass an expression and *periodic()* will evaluate it.

```
periodic <-
#
# period - the number of calls between performing the action.
# ctr - can be specified to start at a different point.
#
function(period = 4, ctr = 0)
{
  function(topExpr, value, ok, visible, cmd) {
    ctr <<- (ctr + 1)%%period
    if(ctr == 0)
       eval(cmd)
```

```
    return(TRUE)
  }
}
```

Given these two functions and the basic accessors for adding and removing entries from the task-handler list, we can set some handlers and see how they perform. We first start by adding a collection of *times()* handlers. We given them different expiration numbers - 3, 4, 5, and 6. Also, we identify them as a, b, c, d. For the purpose of illustration, we ensure that none-are activated until we have registered them all. We do this by initially suspending the manager and then adding the tasks. Then we activate it again.

```
> h <- taskCallbackManager()
> h$suspend()
>
> h$add(times())
[1] "1"
> h$add(times(4,"b"))
[1] "2"
> h$add(times(5,"c"))
[1] "3"
> h$add(times(6,"d"))
[1] "4"
> h$suspend(FALSE)
[Task a] 1
[Task b] 1
[Task c] 1
[Task d] 1
>
```

The output below the second call to *suspend()* is from each of the handlers giving their counts and identifiers.

Next, we add a *periodic()* handler. We specify user data that R will pass to this function when it calls it. This is an expression that the handler function (i.e. the function returned by calling *periodic()*) will evaluate every 4-th call.

```
> addTaskCallback(periodic(), quote(print("ok")))
[1] 5
[Task a] 2
[Task b] 2
[Task c] 2
[Task d] 2
```

Again, the output below the result is from the handlers. The most recently added handler in this call (the function returned from calling *periodic()*) does not generate any output. We must wait another 3 calls for it to perform its real action.

We can continue to give regular R commands and see how the handlers work. We issue arbitrary commands and look at the output from the handlers.

```
> sum(rnorm(10))
[1] 2.791676
[Task a] 3
[Task b] 3
[Task c] 3
[Task d] 3
```

At this point, the first timer (a) has expired having reached its maximal count of 3. It has been removed from the list and so will not appear in any subsequent output.

```
> sqrt(9)
[1] 3
[Task b] 4
[Task c] 4
[Task d] 4
```

At this point, handler ‘b’ has also expired and is removed.

```
> length(objects())
[1] 6
[Task c] 5
[Task d] 5
[1] "ok"
```

Handler ‘d’ has expired. Also, since this is the 4-th call, the *periodic()* handler kicks in and evaluates its *cmd* argument. This is the expression `print("ok")` and gives rise to the last line of the output.

```
> gamma(4)
[1] 6
[Task d] 6
> gamma(4)
[1] 6
> gamma(4)
[1] 6
> gamma(4)
[1] 6
[1] "ok"
>
```

After the first of these calls, handler ‘d’ expires. The *periodic()* handler is still active. After the 4-th of these calls, it generates more output. And this will continue ad infinitum.

## 5.1 Removing Handlers

Removing handlers is quite simple. We can use either position indices or names. Names are preferred since positions change when other callbacks are removed. The name of a callback is returned in the value given by *addTaskCallback()*. It is the *name()* of the position value returned.

In the following output, we show how we add two callbacks, remove one, then add another, and finally remove a callback.

```
>  addTaskCallback(times(300), name="a")
a
1
[Task a] 1
> getTaskCallbackNames()
[1] "a"
[Task a] 2
> addTaskCallback(periodic(2), data = quote(print("periodic")), name="b")
b
2
[Task a] 3
>  getTaskCallbackNames()
[1] "a" "b"
[Task a] 4
[1] "periodic"
```

```
> removeTaskCallback("a")
[1] TRUE
>  getTaskCallbackNames()
[1] "b"
[1] "periodic"
>  addTaskCallback(times(300), name="a")
a
2
[Task a] 1
>  getTaskCallbackNames()
[1] "b" "a"
[1] "periodic"
[Task a] 2
>  removeTaskCallback("b")
[1] TRUE
[Task a] 3
>  getTaskCallbackNames()
[1] "a"
[Task a] 4
>
```

# 6 Issues

The following are some questions that arise when adding this style of a callback mechanism.

- Should these callbacks be invoked after a call to *q()*?
- Should these callbacks be used when reading input from a file (or connection)? This is easy to do (just add a call to `R_callToplevelCallbacks()`) but we chose not to do so for the moment.
- The callbacks are not invoked during the execution of examples since they are not actually top-level tasks. This might well be fixed when we have multiple evaluators and use a separate evaluator for examples. In that case, we would try treat the expressions as real input rather than emulate it.
- We don't necessarily want these callbacks run while in the browser. Do we need this to be an option? for the session or for each handler?
- Should multiple expressions on the same input line constitute a single task?

  ```
  sum(1:10); sqrt(9)
  ```

  At present, they are separate tasks and we call the handlers after each expression is evaluated, i.e. once for each of `sum(1:10)` and `sqrt(9)` twice in our example.
- An issue is how to pass the current expression (`R_CurrentExpr()`) to an R function registered as a handler so that a) it appears as the user would expect and can deal with, and b) does not' get repeatedly evaluated causing an infinite loop? In the function handler, I have done this by wrapping it inside a call to *quote()* when constructing the call to the function from within the handler. A promise may work, but didn't seem to give the appropriate value.

# A Appendix

Here is a simple example of registering a callback via C code. One needs access to **RCallbacks.h** which is currently available on in the **src/include**.

```
#include "Rdefines.h"
#include "RCallbacks.h"

Rboolean
basicTask(SEXP expr, SEXP value, Rboolean succeeded,
          Rboolean visible, void *userData)
{
    static int i = 0;
    Rboolean ans = TRUE;

    fprintf(stderr, "[basicTask] %s\n", (char *)userData);
    i++;

    if(i == 4) {
        ans = FALSE;
        i = 0;
    }
    return(ans);
}

void
addHandler()
{
    char *str = strdup("[prompt] ");
    Rf_addTaskCallback(basicTask,  str, free,
                       "MyHandler", NULL);
}
```