Gtk Bindings for R

Duncan Temple Lang

June 2, 2003

In this document, we present tbe basics of the Gtk package and how to use it. The idea is to determine a programming interface and then use this to automatically generate the code.

1 Introduction

One should be aware that we need to further extend and generalize the R event loop to accomodate other the event queues from other libraries such as Tk, Gtk, Qt, etc. This is being investigated. In the meantime, most things will work properly in the "standard" R on both Windows and Unix except for timeouts, idle tasks and tooltips (which use timers). On Unix, if one configures and runs the Gnome interface (`R --gui=GNOME`), all will behave correctly. Similarly, if one runs R within another Gtk-enabled application which uses the Gtk event loop handlers, things will be fine. Two examples of this are GGobi and Gnumeric in which R has been embedded. Additionally, one can write a simple application in C that starts the R engine (within the same process), reads an R initialization script to create the GUI and register callbacks, timers, etc. and then uses the Gtk event loop. An example is given in this RGtk package. Or an even simpler version is to call *gtkMain()* at the end of one's profile script.

2 The Basic Overview

Gtk is a toolkit for creating graphical user interfaces. It provides two basic types of interaction.

Widgets A large collection of components that can be used to create the GUI. These include common primitive elements such as buttons and labels, menu items, text widgets, drawing areas, top-level windows, ... with which one can make more complex composite widgets such as dialogs, calendars, file selection interfaces, etc.

In addition to the low-level, action widgets, there are also *container* widgets whose task is to house and manage other widgets. The different types of container widgets manage the space they have for their *child* widgets in different ways to give different visual effects, specifically when a window/widget is resized. Container widgets include notebooks with tabs for each separate "page"; scrolled windows which provide horizontal and vertical scrollbars that get associated with and control the visibility of a child widget; different "packing" widgets such as a table, a box, a menu and menu bar, ...

Callbacks Also, Gtk provides a way to associate handlers or actions with particular events on these different components so that one can give the GUI its behavior. In the case of R, these callbacks are given primarily as S functions. These are called when the event occurs with arguments that identify the details of the event, including the particular widget in which the event happened.

When developing a GUI, typically one first creates the visible part, i.e. the collection of different widgets. We do this by creating instances of the different Gtk widget classes, creating the basic elements and "adding" them to the desired container widget. Having created the elements, we then display or "show" the top-level element, be it a top-level window or merely a container to be added to an existing top-level container. There are a variety of different ways to learn about what different types of Gtk widgets are available to be used in a GUI.

- One can look at the static/pre-processed Gtk documentation at `http://developer.gnome.org/doc/API/gtk/index.html`, which has pages on each of the different widgets in each of the different distributions. Your system should have these widgets, but may a) have a different version of Gtk and so be slightly different, and b) have additional widgets that extend the basic Gtk and that will not show up in the Gtk Web pages.

- One can dynamically discover the names of the Gtk classes and widgets known to R via the (S) commands

```
library(RGtk)
.GtkClasses
```

  .GtkClasses gives a list of all the Gtk classes that were available when the `RGtk` package was installed. That too might have changed or not include additional Gtk widgets.

  One can also browse the collection of Gtk classes using the function *showGtkInheritance()* in the `examples/` directory:

```
source(system.file("examples", "getHierarchy.S", package="RGtk"))
showGtkInheritance()
```

  This allows one to examine the inheritance structure of the Gtk classes and explore how the different widgets are related to each other. Also, one can use this as a definitive guide for finding out

  – the callbacks and their argument types,

  – the properties that can be read and/or modified to control a widget instance

  that are supported by a particular widget type.

hierarchy.jpg

Figure 1: Interactive Gtk class hierarchy viewer

Having created the physical display for the GUI by creating and arranging the different widgets, we next register the different callback functions with the particular widgets and specifically with the different events of interest. Again, one must learn which events are associated by which type of widget, and when and how the handler will be called. One can use the different documentation sources above for this purpose. In general, each callback function will be invoked with at least one argument: the Gtk object in which the event occurred. Callbacks for different events may provide additional arguments which provide more information about the particulars of the event. For example, when a button is released in a widget, the *button-release-event* passes the widget and also a `GdkEventButton` instance which gives information about which button was released, etc.

In addition to the arguments provided by Gtk, one can also associate an S object with a widget and event and have this passed to the callback function as an argument. By associating different objects with different widgets, one can use the same callback function. That function can implement different behavior based on the additional argument, and using R's lexical scoping one can event modify the S object passed as an additional argument.

When we create functions or progams to create GUIs, we clearly can associate the callbacks with the widgets as soon as we create them. We do not have to separate the creation of the physical display with the actions. However, such a division can be useful in terms of being able to easily adapt and also re-use GUIs as the physical display needs to be changed.

In the next few sections, we will describe how we can implement the very basic and often-used *Hellow World* to illustrate the essential concepts in the RGtk package. This is a very simple GUI which presents a button in its own window. When the user clicks the button, we print a message on the console.

3 Creating Gtk Objects

As we saw above, one starts creating a GUI by instantiating different Gtk objects. In the case of the "Hello world" application, we need to create i) the top-level window, and ii) the button which the user clicks. We create the window using the *gtkWindow()* function

```
win <- gtkWindow(show = FALSE)
```

This creates an instance of the *GtkWindow* class. Generally, the S language constructor function for a Gtk class named *Gtk<Class>* is given by *gtk<Class>()*, i.e. replace the capital G starting the word with a lower case 'g'.

Note that the constructor functions for each class that extends *GtkWidget* have an optional *show* argument. This controls whether the widget is made ready for showing immediately or if this must be done by the programmer at a later time. The advantage of deferring this is usually a marginal gain in efficiency. Hence, the default is T. One need only prohibit the top-level container, e.g. the window in this example, from being *show*n and then none of the sub-widgets will be displayed.

We can invoke methods on the Gtk objects to query or modify their state. For example, we can set the title for the frame of the window using the underlying C-level routine `gtk_window_set_title()` provided by the Gtk libraries. We do this in S via the command

```
gtkWindowSetTitle(win, "Hello world test")
```

There are several things to note here. Firstly, we use a different naming convention than Gtk's C-level API. Specifically, we eliminate the underscores (_) and capitalize all but the first word (i.e. the next letter after the _). Secondly, we pass the Gtk object on which we are operating as the first argument. Thirdly, the type of the second argument is defined by the underlying C routine and is a string. This corresponds to a character vector of length 1 in S.

The case of *gtkWindowSetTitle()* is quite simple. We started with an object of class GtkWindow in R (created using the S constructor) and then invoked the function *gtkWindowSetTitle()* for that same class. But what about, for example, the general functions to show or hide a widget, get its parent widget, etc. These apply to all *GtkWidget* objects, and not just the *GtkWindow* objects. Accordingly, the S interface uses the names that correspond to the C-level API and are prefixed by *gtkWidget...()*, rather than *gtkWindow()*. This makes it hard to remember the precise name of the function one wants to call since it depends on the inheritance or class hierarchy of the Gtk classes.

To make things simpler, we allow one to use a more Java/*C++* style that allows users to invoke methods on an object and leave the S engine to determine the precise name to use. Specifically, we use the `$` operator on the object followed by the name of the method to identify the function. Specifically, one can invoke from S a method on an underlying Gtk object, say *g* usin the form

```
g$MethodName(arg1, arg2)
```

This eliminates the need to remember for which class the method is defined and hence the prefix. Also this form of invocations inserts the target object, *g*, as the first argument in the call to the real S function being called and so reduces typing.

An example will make things clear. Consider again setting the title of the window. Rather than using *gtkWindowSetTitle()*, we can use the command

```
win$SetTitle("Hello world test")
```

This looks for the appropriate function given the class and parent classes of *win* and then invokes the "nearest" function. This corresponds to the command

```
gtkWindowSetTitle(win, "Hello world test")
```

above, but is easier for the user and is also more robust to changes in the class hierarchy and C-leve API. There is a marginal penalty in computational performance, but this may disappear in the future and is also not likely to be a serious issue a) when creating the GUI, b) given the overhead in setting up callbacks to S functions.

We can now continue with our "Hello world" example. We have created the window and set its title and hence seen how to create Gtk objects and invoke methods. And so creating the button becomes quite simple. We choose which the appropriate Gtk class – *GtkButton* – and find the appropriate constructor. There are two constructors in the C-level API for this class: one that takes no arguments and another that takes a string to display as the text in the button. In S, these two constructors map to a single constructor function, whose name is the name of the class suitably (de-)capitalized, *gtkButton()*. If one calls it with no arguments, the first C-level constructor is called. Alternatively, if one gives a character vector of length 1 as the first argument, the second version is called. More generally, the R interface to Gtk attempts to map the constructor routines into a single S function that can determine which C routine to call based on the number and/or type of the arguments. For the most part, this is quite simple and works effectively.

In our example, we specify text for the button's display and so call

```
btn <- gtkButton("Say 'Hello World'")
```

Next we put the button into the top-level window. The latter is a *GtkContainer* object and has a default mechanism for placing children widgets. Since this is the only widget we will display in the window, we don't have to worry about how to aportion space between different widgets, etc. All we need do is invoke the *Add()* method on the window, giving it the child widget which is the button.

```
win$Add(b)
```

When we create the button, we did not provide a value for the *show* argument and so the button is potentially visible. To actually see it, however, we need to show the top-level widget, i.e. the window. We do this by explicitly calling its *Show()* method.

```
win$Show()
```

We might chose to specify the size of window before showing (or even afterwords). We could do this using the *SetUsize()* method, as in

```
win$SetUsize(300, 300)
```

4 Callbacks

At this point, we have created a Gtk GUI that one can see on the screen and can even interact with by clicking on the button. The next step is to make it do something when we click on the button, and this is where we look at callbacks.

The usual types of events are user interactions such as clicking on a button, dragging the thumb of a slider, moving the mouse over a drawing area, etc. Other types of events might be less visible and more abstract such as text being pushed or popped onto a status bar, a new data set being created, and so on. Basically, each type of event is associated with a Gtk object in which it "occurs". A Gtk object can support different types of events, and events in different objects are treated independently. One creates and customizes an application by connecting different pieces of code that are executed when particular Gtk objects raise/emit particular events.

In our example, we want to execute a simple piece of S code that is executed when the user clicks on the button. The code simply writes the string `"Saying hello from the button"` to the console via the *cat()* function. To arrange this, we can look at the different signals that the button supports. (Of course, we chose the *GtkButton* class because it provided the appropriate signal, so this seems like we are going round in circles. In general, knowing the widget to use and appropriate signal is the trick to using any toolkit.) Using the help pages for Gtk or the hierarchy viewer above, we can find out that the button supports 5 diffferent types of signals itself, and inherits many others from its ancestor classes (*GtkBin*, *GtkContainer*, *GtkWidget* and *GtkObject*). These signal names are *pressed*, *released*, *clicked*, *enter* and *leave*. The one we are interested in is *clicked*.

We specify our callback for the particular button using the method *AddCallback()*. We specify the name of the signal (i.e. *clicked*) and an invokable S object which will be called when the signal occurs:

```
btn$AddCallback("clicked",  quote(cat("Saying hello from the button\n")))
```

Now, when you click on the button, the string will be printed on the console.

The code that is to be called when the event occurs can be an S expression or call, or a function. If it is an expression or call, then it is evaluated when the event occurs. One typically creates such callable objects using *quote()*, *expression()* or *substitute()*. Each of these types of callbacks is evaluated as a top-level expression and one is presumably interested in its global side effects, such as changing the value of a session-wide variable, writing to a file or the console, or updating one or more graphics devices.

If the callback is a function, then it is invoked slightly differently. There is more information available to the callback, specifically, the arguments that are made available at the C level by the Gtk API. These are passed onto the S function. This collection of arguments always includes the Gtk object for which the signal is being emitted. Many signals also provide additional values that parameterize the event and allow the callback to be written generally but parameterized by the widget or other event-specific values. These values are converted to S objects using the basic conversion mechanism described in 6. In addition to the event-specific values passed from Gtk, one can also specify S objects that Gtk remembers and passes to the function when it is called. Again, this allows one to parameterize general S functions to act on the specifics of the event. We'll look at how this can be used in ??.

Note that we added the callback after the button was created and visible. This is not necessary and we can add it before the top-level window is shown. However, it does illustrate that we can dynamically add callbacks at any time. Indeed, we can add multiple callbacks to the same Gtk object, and even for the same signal. For example, let's add a second that prints `And again`.

```
id <- btn$AddCallback("clicked", quote(cat("And again\n")))
```

Go ahead and click on the button now and see that two lines of output are produced.

And, of course, if we can dynamically add callbacks, we must also be able to remove them at any time. To do this, we use the *DisconnectCallback()* method for the Gtk object. We give it the identifier for the registered callback that we returned in the call to *AddCallback()*. So to un-register the second callback, we issue the S command

```
btn$DisconnectCallback(id)
```

Again, click on the button and you should get only one line of output, specifically `Saying hello from the button`.

4.1 Why Use Functions as Callback Actions

To be good citizens, we will register callbacks that catch the destroy event on the top-level window so that we can detect when a user kills the window using the window manager rather than programmatically.

```
win$AddCallback("destroy-event", function(w, ev) cat("Being destroyed\n"))
```

5 User-Data in Callbacks

Callbacks:UserData As we mentioned above, when a function callback is invoked it is passed values that provide information about the specific event that triggered the callback. For example, when a button is clicked, the callback

6

6.1 Enumerations and Flags

Enumerated types and flags are symbolic constants that are used to identify different states or combinations of states. In R, we represent these as named integers. The intent is that the user will provide the name (or names for flags) and not a simple integer value. So, for example, when specifying the type of window in

a call to *gtkWindowNew()* we can use any of the names from the *GtkWindowType* vector representing the enumeration:

⟨ ⟩≡
```
> GtkWindowType
toplevel   dialog    popup
       0        1        2
```

Since this is an enumeration, we specify just one of these values, as in

6a ⟨ ⟩+≡
```
> gtkWindowNew("toplevel")
```

When a flag value is expected, we can combine different values together. Since we can OR (|) names together, we need an alternative syntax. For this, we use a simple character vector containing the names of the flag elements. As an example, consider the display options for controlling the appearance of the calendar widget. The *GtkCalendarDisplayOptions* is a named integer vector giving the different names for the flag values. If we weant to have weeks start on a monday and also show week numbers, we can do this as

6b ⟨ ⟩+≡
```
> gtkCalendarDisplayOptions(cal, c("week-start-monday","show-week-numbers"))
```

To activate all options, we can use

6c ⟨ ⟩+≡
```
gtkCalendarDisplayOptions(cal, names(GtkCalendarDisplayOptions))
```

The calendar can be create and display using the following code

6d ⟨ ⟩+≡
```
cal <- gtkCalendar()
gtkCalendarDisplayOptions(cal, c("week-start-monday","show-week-numbers"))
w <- gtkWindow()
w$Add(cal)
```

Using names guarantees the validity of the value as it is resolved and checked at run time. However, to guard against erroneous values, we have C-level code that checks an integer value is within the appropriate set of C-level values and returns an object representing that symbolic value.

This multiple level of checking may seem inefficient to some. For each enumeration and flag type, one can directly compute a value and then store that value. There is a 'map' function for each enumeration or flag type that maps the specified value into a valid value of the appropriate type. For example, in the case of the *GtkWindowType*, there is a function *mapGtkWindowType()*. Similarly, for calendar display options, there is a function named *mapGtkCalendarDisplayOptions()*. One can call these with the names of the values and get the actual value.

6e ⟨ ⟩+≡
```
> mapGtkWindowType("toplevel")
GTK_WINDOW_TOPLEVEL
                  0
attr(,"class")
[1] "GtkWindowType" "enum"
> mapGtkCalendarDisplayOptions( names(GtkCalendarDisplayOptions))
[1] 31
```

One can use this in subsequent calls and this will bypass the S- and C-level verification. This is because each mapping function checks whether the argument is of the appropriate class. If it is, it assumes the value is correct. One can cheat, but this is not a good idea, especially for portability.

One can note the fact that the name `toplevel` is converted to `GTK_WINDOW_TOPLEVEL` in the value returned by the enumeraton. This is the C-level name for the enumeration. It can be used as a synonym for the value. In other words, `toplevel` and `GTK_WINDOW_TOPLEVEL` are the same. And indeed, for every enumeration or flag we have both a sets of element names available. The local version is available as described above by giving the name of type, e.g. *GtkWindowType* and *GtkCalendarDisplayOptions*. Prefixing the name with a `.` gives the alternative version with the longer, internal names. Use whichever form you desire. Those who write Gtk code in other languages may be familiar with the internal names. The local names are shorter.

7 Basic Methods

As we have seen, the visual part of a GUI is created by packing widgets into containers, and building a hierarchy of interface components. In many cases, we will have explicitly created the different elements in S and can make them available to other parts of the application which need to access them directly, e.g. to register a callback, set a property, etc. In other cases, we may not have the relevant Gtk object as an S variable. For example, if we create a composite widget such as a dialog or a color wheel, we will not have access to the internal buttons or slider within these higher-level widgets. However, it may be convenient to dynamically access them by navigating the tree of widgets. For example, given the dialog, we can ask for its work area and action area (the buttons) using properties. Given either of these, we can ask for its child widgets and recursively access the different sub-widgets within the tree. Similarly, given a widget, we can access its parent widget and traverse "up" the tree.

The functions *gtkParent()* and *gtkChildren()* can be used to navigate the widget hierarchy. *gtkParent()* is used when we want to go "up" the hierarchy to access the container widget whne we have a sub-widget. One common case is in a callback when we want to, for example, access the top-level window that contains the widget associated with the callback. To do this, we can "walk up" the hierarchy until we either find a *GtkWindow* object or find a widget whose parent widget is `NULL`.

7a ⟨*Get Window* 7a⟩≡

```
gtkGetWindow <-
function(w)
{
  while(!is.null(w) && !inherits(w, "GtkWindow") ) {
    w <- gtkParent(w)
  }

  w
}
```

In our "Hello world" example, we can find the top-level window given the button using

7b ⟨ ⟩+≡

```
gtkGetWindow()
```

Note that if we really want the top-most container, the function *gtkWidgetGetToplevel()* will do the same job, but entirely within C code.

To walk "down" the tree, we use *gtkChildren()*. For example, given the top-level window in the "Hello world" example, we can locate button using the S command

7c ⟨ ⟩+≡

```
gtkChildren(win)[[1]]
```

As one can see, *gtkChildren()* returns a list with an element for each child. The precise order in which the child widgets appear depends on the container widget, the order in which the widgets were added/packed into the container, etc. In other words, it is context-specific.

As simple syntactic sugar, one can use S's subsetting on a container widget to access the children individually by index. Specifically, we can get the i-th child from a *GtkContainer* widget *g* as

8a ⟨ ⟩+≡
```
g[[i]]
```

The example above can be given more simply as

8b ⟨ ⟩+≡
```
win[[1]]
```

Of course, this notation is also used to access properties within a *GtkObject*, but where the index is given as a string. To make code more readable, use *gtkChildren()*.

8 Forcing the Event Loop

All functions that involve a call to C code have a *.flush* argument. This is expected to be a logical value and controls whether the routine `gdk_flush()` is called at the end of the C routine to ensure that events are processed as soon as possible. This is T by default. The function *.GtkCall()* is the intermediate layer that sits above the real call to the C routine via the C.*Call()*.

One can also flush the event queue manually by calling the function *gdkFlush()*.

9 Accessing Object Properties

Each `GtkObject` instance supports has values that are accessible by name. The collection of properties can be accessed via the *names()* function and this makes the object look like a list of named values. These properties also posses a hierarchical characteristic.

Each property has a specific type that can be assigned to it. Some of these values are are writeable, while most are readable. Additionally, the collection for a given instance is made up of combining the properties from the different classes from which the instance is derived. One can discover all this information using the function *gtkObjectGetArgInfo()*.

8c ⟨ ⟩+≡
```
b <- gtkButton("Some text")
names(obj)
b[["label"]]
b[["label"]] <- "A Replacement string"
```

8d ⟨ ⟩+≡
```
gtkObjectGetArgInfo(b)
```

10 Setting Callbacks

We use *gtkAddCallback()* to register an S function that is to be called when a particular Gtk event occurs on a specific object.

11 Timers & Idle Tasks

gtkTimeoutAdd() and *gtkTimeoutRemove()* provide a convenient way to register S functions to be called after a specified interval of time. If the function returns T, the task is rescheduled to run after the same interval. Alternatively, returning F discards the timer. One can programmatically remove the timer using *gtkTimeoutRemove()* and the value returned from *gtkTimeoutAdd()*.

By default, the function is called with no arguments. However, one can arrange to have it passed a value by specifying the object as the *data* argument in the call to *gtkTimeoutAdd()*. This is similar to the *data* argument for *gtkAddCallback()*.

Idle tasks are run when there are no other events to process in the Gnome event queue. These can be used to perform non-urgent background tasks. The interface is very similar to timeout functions. One registers an idle task with *gtkAddIdle()* and this returns an identifier for the tasks. One can remove the task using *gtkRemoveIdle()*.

12 Reflectance

When we create an Gtk object, we assign the appropriate classes to the resulting S object. However, in some cases, we might get acccess to an object that is not created by us and does not have the appropriate classes. In order to have full access to this object, we can compute and assign the appropriate classes to an object using *gtkObjectGetClasses()*. We can also compute just the immediate name of the object's type using *gtkObjectGetTypeName()*. We can also get a reference to the underlying `GtkType` using the function *gkObjectGetType()*.

While in most cases, we know the names of the different signals for a widget to which we might want to attach or connect a callback function, there are situations when we might want to ask for the names of the available signals. The function *gtkObjectGetSignals()* returns the available signals for an instance of a Gtk class. We can also ask for the signals for a class by giving its name or type to the function *gtkTypeGetSignals()*.

```
> sig <- gtkTypeGetSignals("GtkButton"){\tt{}"pressed"}
> gtkSignalGetInfo(sig)
$signal
pressed
     60
attr(,"class")
[1] "GtkSignalId"

$parameters
list()

$returnType
void
   1
attr(,"class")
[1] "GtkType"

$isUserSignal
[1] FALSE

$runFlags
[1] 1

$objectType
GtkButton
```

```
    40213
attr(,"class")
[1] "GtkType"
```

13 Styles and Themes

14 OOP-like methods

One can also call functions for a particular class of objects using the \$notation. We basically call a method for the instance and specify the object on which to operate as the left hand side of the \$operator.

```
w <- gtkWindowNew()
w$Show()
```

Such methods map to the function corresponding to piecing the different elements together to give `gtk`*`Class`*`MethodName`. The *Class* value is computed from the list of classes for the object and the first for which such a function exists.