This describes the basic setup for embedding GGobi/XGobi in another application. It relates specifically to R and S, but should be of some value for other applications.

The document describes the necessary structural support for creating an embeddable facility. Next, it describes the API for communicating with this embedded application. The API allows the user to query the current state of the ggobi session, and also programmatically set much of the state.

1 Embedding Structure

The first thing we need is the ability to simply create the interface. Having GGobi allow creation of the interface without any data (either a file or actual values) allows us to create the interface and then specify the data. This allows us to treat the specification of the data as a second step in the initialization and implement that stage using a user-level (in the R/S language) function. Hence we don't have to tie the initialization to a file name or dataset.

The ggobi session is initialized via the routine `init()()`. This takes the name of a file. Alternatively, we can pass a matrix of data and variable names to the routine `()`.

Ideally we would be able to use the different elements of ggobi independently. For example, we might create an ash display and separately a scatterplot. This requires the separation of global variables.

2 Event-Loop Integration

One of the key aspects of embedding a GUI tool into an environment such as R or S is that we can merge the event models for the two. R and S are interactive environments which wait for input from the user and operate within a read-eval command loop. The X11 device and all other sources of events are merged with this via the system `select()` routine. To allow this, the event operations must support the basic operations of

- providing a file descriptor on which input can be detected to indicate the existence of events of interest for that source
- consuming all the available events without blocking waiting for more.

The X11 library provides these facilities via the #ConnectionNumber macro and `XNextEvent()`, `XtDispatchEvent()`

For GTK+ and GDK, the following snippet of code can be used as an event handler procedure with the R input source management.

```
while(gdk_events_pending()) {
  gtk_main_iteration_do(false);
  ctr++;
}
```

The routine is registered for the file descriptor computed from #ConnectionNumber called with the value of `gdk_display`. This latter variable is available by including **gdk/gdkx.h**. (Something will need to be done for windows.)

At present, the above code does not quite work to handle all events. Specifically, when one sets the dataset programmatically, the variable selection panel is not updated appropriately. The widgets are created, but now exposed.

Timer functions need to be added to the R, not Glib's, event loop. (R needs to handle these timers along with the select call.)

3 Warnings

Output that goes to standard error in GGobi should now go through `GGobi_Warning()`. This allows R/S to report it in its traditional way, namely via the *warnings()*.

`g_printerr()` and `g_print()` need to be changed.

4 Installation

The makefile contains a rule for creating a shared library, ***libGGobi.so***.

5 Calling R functions

While the embedded Ggobi in R allows R users to avail of functionality provided by Ggobi, the reverse direction is also feasible. Ggobi can use R's capabilities to implement certain functionality. Additionally, we can allow users to specify interpreted R functions to be used for performing certain computations and to respond to certain events. We outline how this works in the paragraphs below.

Consider the simple problem of taking two variables, X and Y, and computing a smoothed version of Y. The inputs are the values of X and Y and the output is a vector of the same length as Y (and hence X) with the smoothed values. A simple R function to do this is

```
function(x, y) {
 predict(loess(y ~ x, data.frame(x=x, y = y)))
}
```

GGobi can then invoke this function with a pair of variable values and extract the values for the result. The following C code illustrates how this might work.

```
USER_OBJECT_ x, y, call, tmp;

 call = allocVector(VECSXP, 3);
 PROTECT(args);
 CAR(args) = RS_smootherFunction;
 CAR(CDR(args)) =  RS_GGOBI(variableToRS)(xindex);
 CAR(CDR(CDR(args))) =  RS_GGOBI(variableToRS)(yindex);
    tmp = NEW_NUMERIC(1);
    NUMERIC_DATA(tmp)[0] = width;
    CAR(CDR(CDR(CDR(e)))) =  tmp;
 ans = eval(call, R_GlobalEnv);

 UNPROTECT(1);
```

The user can now register the function to use for smoothing, allowing them to control how it is performed. The only constraint is that it must be an argument that accepts 2 arguments and returns a numeric vector of the same length as the inputs.

On difficulty we face is how to arrange to have these routines called. One way is to them to be registered as functions to be called by GGobi. This would make the GGobi code more complicated and indirect. An alternative is to use the linking facilities to override or replace routines from GGobi with those in the R.so. When we link R.so with libGgobi.so, we arrange to have the routines we define in this package be invoked. For this to happen, we obviously must define methods with the same name. Also, they must have the same signature - parameter and return types. Then, we implement the body of the replacement routine in whatever manner we wish.

Example for smoothing,identify, etc. Transforming data.

The function that is registered can be a closure with its own "local" state.

Consider identifying points on a plot. We can arrange to have an R function be invoked when a new point is encountered or a point selected by explicitly click on the canvas. We can make this behave like the locator() function in R and S in that it gathers up the collection of (x, y) pairs for all the clicks and returns this.

Since this is event driven, rather than a blocking call as with locator(), we can arrange to to this with an "object" with its own state.

```
function () {
  vals <- list(x=numeric(0), y= numeric(0))

  click <- function(x, y) {

    vals$x <<- c(vals$x, x)
    vals$y <<- c(vals$y, y)
  }

  return(click)
}
```

Closures for mutable state and local variables.

6 Multiple GGobi

Occassionally, it is useful to interact with two or more datasets simultaneously. Constrasting the same views with different data, comparing characteristics and working on different projects simultaneously are obvious contexts. The R-GGobi connection allows multiple ggobi "devices" to be created. These are indexed by name and/or number and are similar to the regular graphics devices.

The following command identifies the symbols that are static variables

```
nm -A *.o | grep ' [a-tA-Z] ' | grep -v ' [UTtb] ' | grep -v '.*\.[0-9]*$'
```

There are still a lot of statics to be dealt with.

We need to remove the macros from ggobi.h as we go to make things simpler. Just need to find the time. (This is just getting it down there on "paper"....)

One of the important steps in introducing the potential for having multiple instances of the ggobid structure in existence in an application is the removal of global variables. This involves passing a reference to the particular ggobid object to each routine that needs it. (This is very similar to threading.) This is typically a tedious task, but not very challenging. (Remove the global variable and compile to identify which routines need to have it explicitly passed as an argument. Then change their signatures and recompile. Iterate until messages are gone.)

In an event driven framework, there is an added challenge. The callbacks invoked by the underlying widget toolkit have predefined signatures as defined in the implementation of the toolkit. Hence we cannot add an extra argument. Instead, we must be able to have the ggobid object passed to it or computed from within that routine.

In many cases, the user-level data that can be registered with a widget to be passed to the routine is not used. In such cases, we specify that the reference to the ggobid object should be supplied as that argument.

In other cases, the user-level data is already being used. There are two obvious possibilities for handling these situations. Firstly, we can store the reference to the ggobid object in such a way that we can uniquely identify it and that this works for different ggobid instances. A simple strategy is to assign it to user-level data maintained by a widget. When the callback is invoked, the source of the event (a widget) is supplied as an argument. From that, we can retrieve the particular ggobid reference associated with that interface. Rather than setting the storing the value in each widget, it is convenient to store it only for the top-level window that houses each widget. In this way, it is in a single place for each widget hierarchy.

An alternative approach is to change the type of the user-level data. We would allocate a structure that contained the current data and the reference to ggobid and specify this as the user-level data argument. For example, suppose we current pass an integer as the user-leve data to an callback. This is the case for many menu items. Then we would have a structure

```
typedef struct {
  int value;
  ggobid *gg_reference;
} IntegerCallbackData;
```

When registering the routine, we would allocate one of these and fill in its fields, and then specify it as the user-level argument.

This has the potential of being faster than retrieving the value from a widget's root window. If speed becomes an issue (as in the tour where there may be a large number of events/callbacks) we can implement this approach.

7 Resetting the data

Change vardata˙init to set the group identifier to 0.

Need to change dataset˙init to make certain that there is only one display in the list when this is finished..

8 Accessing displays and plots

Rather than relying on user gestures to select windows/displays and plots within these and make these active, we provide a programmatic version of this selection mechanism. The functions *setActivePlot()* and *getActivePlot()* can be used to set and query which display and plot within that are active.

One can query what plots are currently in existence using the command *ggobi.getDisplays()*. This returns a list with an element for each of the displays. Each description identifies the type of the plots the display contains and a list of all the sub-plots contained in the display. Each of these sub-descriptions identifies the variables involved.

Need to ensure that the display is updated correctly when we set the active plot.