PBS Modelling 1: Developer's Guide

Alex Couture-Beil, Jon T. Schnute, and Rowan Haigh

February 22, 2007

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

This document is intended for future developers of PBS Modelling and describes the internal data-structures and algorithms used to implement PBS Modelling's GUI functionality. Users of PBS Modelling should consult "PBS Modelling 1: User's Guide".

Résumé

Ce document est prévu pour des développeurs futures de PBS Modelling. Ce document décrit les algorithmes et structure de données pour la fonctionnalité de la création des interface graphique (ou GUI «Graphical User Interface») de PBS Modelling. Les utilisateurs de PBS Modelling devraient consulture «PBS Modelling 1: User's Guide».

Preface

Prior to working on the development of PBS Modelling, I had no experience with the R environment. After reading through "An Introduction to R" and trying out various code samples from Jon Schnute, I began to feel confident with the R language. PBS Modelling initially evolved from samples of tcl/tk obtained from various sources including "A Primer on the R-Tcl/Tk Package" by Peter Dalgaard. I quickly learned that searching google for "R tcltk" provided a wealth of information including a list of R tcl/tk examples compiled by James Wettenhall which I initially used as a starting point for experimenting with different widgets.

I hope to offer some insights of PBS Modelling's GUI creation algorithms and data structures. It is not crucial to understand ever detail in this document, as it is really here to aid in the navigation and understanding of the source code which ultimately determines the (correct or incorrect) behaviour of PBS Modelling.

Alex Couture-Beil
February 2007

1. Overview of creating GUI windows

Graphical User Interface windows are defined in a text file using a special format as described in the Tech Report. In brief, the text file has multiple lines, with each line defining a widget. A widget definition can be extended to multiple lines by using a backslash '\'. A widget may have pre defined parameters that either requires an argument or has a default value for missing argument values. The ordering and default values of these parameters are defined in the `widgetDefs.r` file.

This ASCII text file must be parsed and converted into a tree structured list before the createWin function can call the required tk function to build the window. A tree-structured list of widgets is formed. The window widget defines the root, or starting point of the tree. This tree will have a branch whenever a grid or menu is encountered. These two special types of widgets will contain one or more children widgets.

Table 1. Window descriptions file containing a grid and labels.

```
#file.txt
Window
Grid 2 2
  Label A
  Label B
  Label C
  Label D
Label end
```

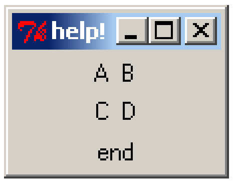

Figure 1. GUI generated by the description file `file.txt` in Table 1**.**

Table 1 describes a window that has a 2x2 grid which will contain labels A,B,C and D, and a fifth label "end" which is not associated with the grid. The window description file is parsed into a tree-structured list that resembles Figure 2.

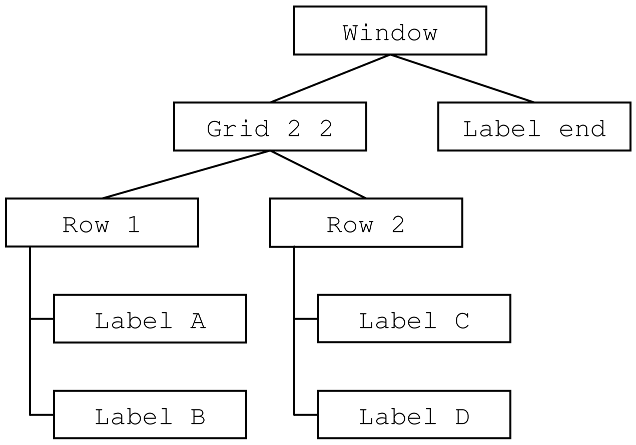

Figure 2. Tree representation of file.txt from figure 1.

However before converting the window description into a tree, it must first be broken up into tokens representing each widget by using the following process.

1. Extract lines describing widgets into a string, and if required, collapse widget definitions that span multiple lines into a single string. This occurs in the main code of the `parseWinFile` function.
2. Convert each widget definition string into a non tree structured list by using the `.convertPararmStrToList` function.

3. Scan through this list and verify the widget type is valid, all required arguments are given and valid, and assign default values to any missing arguments. This is done in the `.getParamFromStr` function.
4. Scan through the list looking for `menu` or `grid` widgets. These widgets will cause a branch in the tree. If a grid is found, associate the next `ncol*nrow` widgets as children of the grid widget by using the helper function `.parsegrid`. This helper function is recursive and will handle nested grids. Similarly `menuitem` and `menu` widgets are associated together and the `.parsemenu` function.

This process is initiated by calling `parseWinFile`. However, the order of the call stack does not appear in the same order of the listed steps above. Instead the following call stack is produced.

```
createWin
    parseWinFile
         .getParamFromStr
              .convertParamrmStrToList
                   calls strToList C code
```

Table 2. Function call stack produced while parsing description files.

1.1. Widget list representation

Widgets are represented by using a list. The list must have the named element `"type"`, which is used to identify the type of widget that the list represents. Once the `"type"` value is known, PBS modelling can insert or extract other elements as defined in `.widgetDefs` from `widgetDefs.r`. During the parsing process any missing values will be inserted with defaults as defined in `.widgetDefs`.

During the parsing process, additional internal fields may be added to the widget. These should typically start with a dot `(.)` to differentiate an internal field from a user specified one.

Currently every widget is internally assigned a `.debug` element which is a list containing information used for displaying error messages to the user.

▪ `$.debug$sourceCode`	*source code entered by user for the widget*
▪ `$.debug$fname`	*filename of WD file*
▪ `$.debug$line.start`	*beginning line of widget description*
▪ `$.debug$line.end`	*end line of widget description*

To familiarize yourself with the list representation of a vector, examine the output of
`str(parseWinFile("vector myVec 3", astext=T)[[1]])`

`parseWinFile` returns a list of unnamed lists which represent individual windows. The `[[1]]` suffix is used to target the first window. However, in this case only a single window is defined, so including the index simply reduces one level of indentation.

1.1.1. The grid widget

The grid widget includes the internal field `".widgets"` which is used to store all children widgets of the grid. This is a two dimensional list, with the first index representing the row, and the second index representing the column. Recall figure 1 represented a 2x2 grid containing four labels A,B,C,D.

```
> str(parseWinFile("file.txt")[[1]]$.widgets[[1]], 4)
List of 15
 $ type       : chr "grid"
 $ nrow       : int 2
 $ ncol       : int 2
…omitted arguments…
 $ .widgets   :List of 2
  ..$ :List of 2
  .. ..$ :List of 9
  .. .. ..$ type  : chr "label"
  .. .. ..$ text  : chr "A"
            …omitted arguments…
  .. ..$ :List of 9
  .. .. ..$ type  : chr "label"
  .. .. ..$ text  : chr "B"
            …omitted arguments…
  ..$ :List of 2
  .. ..$ :List of 9
  .. .. ..$ type  : chr "label"
  .. .. ..$ text  : chr "C"
            …omitted arguments…
  .. ..$ :List of 9
  .. .. ..$ type  : chr "label"
  .. .. ..$ text  : chr "D"
            …omitted arguments…
```

Figure 3: Children widgets of a grid – try reproducing the same output

Note that the grid has a `.widgets` element which is a list of 2. This is used to store the rows. And each row has two widgets, A, B for row 1, and C, D for row 2. The suffix `[[1]]$.widgets[[1]]` of parseWinFile, targets the first widget of the first window.

1.2. Internal data structure (.PBSmod)

TCL/TK relies heavily on the use of pointers. These pointers are required for controlling windows and extracting data. PBS Modelling makes use of a global list to store TCL/TK pointers as well as other information that is associated with each widget, or window.

PBS Modelling specifies that each window has a name, either defined explicitly by the user in a window description file, or by using the default name of `window`. All information to do with a specific window is stored in a list under `.PBSmod$windowName` where `windowName` is the actual name of the window.

Data that is related to PBS Modelling as a whole, and not a particular window, such as user defined options and the current active window, is stored under `.PBSmod` with a name that begins with a dot. This avoids conflicts with windows since a window name may not begin with a dot.

```
.PBSmod <- list(
    myWindow <- list(tcl pointer stuff...),
    mySecondWin <- list(more tcl stuff...),
    .options <- list(openfile=..., option2=...),
    .activeWin <- "myWindow"
)
```
Figure 4: Example of the top level of .PBSmod list

Each window uses a list with the following named components:

- `widgetPtrs`
  a named list containing widget pointers. Each element of the list is named after the variable name of the widget. Not all widgets will appear in this list, only widgets which have a corresponding tk widget.

- `widgets`
  a named list containing important “widget lists” as extracted from the window description file. This list will include every widget that has a name or names argument. Unnamed widget will never be referenced again once the window is created, and therefore do not need to be stored for later usage.

- `tkwindow`
  a pointer to the window created by `tktoplevel()`

- `functions`
  a vector of all function names that are referenced by the GUI.

- `actions`
  a vector of containing the last N actions triggered by the window, where N is defined in defs.R under the `.maxActionSize`

2. Creating widgets from a tree-structured list

Once createWin has parsed the window description file into a tree structured list, it can start creating the actual widgets by calling the appropriate tk commands. Due to the nature of tk, it is easiest to wrap all widgets in a grid. createWin creates a 1xN grid and adds all user supplied widgets to this grid. This guarantees that we only have a single widget on the top level to create, and therefore createWin does not require any loop for creating the widgets, instead it uses a recursive function, `.createWidget`.

`.createWidget` determines the type of widget that needs to be created by looking at the `type` element. It then calls `.createWidget.xxx` where `xxx` is the widget type. For example a label results in a call to `.createWidget.label`. In the case of grids, `.createWidget.grid` creates a tkframe, and then recursively calls createWidget to create every child widget. In some cases these will be nested grids, however, due to the wonderful properties of recursion, this is no different than any other widget.

During the `.createWidget` process, functions that require a `tclvar`, namely widgets with a `name`, will have to store the tcl pointer in the `widgetPtrs` section of `.PBSmod`.

It is important to understand how PBS Modelling creates high level widgets like `vector` in order to understand why some widget information is only stored in the `widgets` list, while not in the `widgetPtrs` list. Some widgets might not have a corresponding tcl/tk widget. For example the `vector` widget is implemented by inserting many `label` and `entry` widgets into a grid. For this reason the `vector` widget will never have a single tcl/tk pointer, but rather a collection of pointers for each `entry` widget.

A vector defined by
```
vector name=foo length=3
```
will create three entry widgets named `foo[1]`, `foo[2]`, and `foo[3]`, which will be inserted into `widgetPtrs` with the three corresponding pointers; however, the name "`foo`" will never appear in the list since there is no pointer to associate with it. It is still necessary to save some reference of the higher level widget (in this case "foo") since it might be reference in `setWinVal(c(foo=1:3))`. Otherwise setWinVal would return an error saying it could not locate the widget named `foo`; however, it would not complain if it received the name `foo[1]`.

All elements of widgetPtrs are lists with exactly a single named element: `tclvar`, or `tclwidget`.

- `tclvar` is the standard pointer for most widgets which is used to get or set values with the standard `tclvalue()` interface function.
- `tclwidget` is only used for `text` widget, since tcl/tk uses a different interface via `tkconfigure()`

2.1.1. Accessing and modifying data stored in widgetPtrs

While it is possible to access the data directly, it is advised to make use of the internal functions: `.map.init`, `.map.add`, `.map.set`, `.map.get`, and `.map.getAll`. These functions use error checking to avoid overwriting a currently saved value.

`.map.init`	initialize a blank list to store data
`.map.add`	only save the value if nothing previously was saved
`.map.set`	save a value even if it requires overwriting previous data
`.map.get`	retrieve a value
`.map.getAll`	retrieve all values

These functions range from a single line to 25 lines of code.

In computer science the term map is used to describe a data structure that maps a key (in string form) to a value. Another common name for a map is a hash table.

2.2. The parseWinFile function

The parseWinFile function is responsible for converting a window description file into an equivalent window description list. A text file can be represented as a vector of strings, with each element of the vector representing a new line of the file. If `astext` is true, parseWinFile does exactly this; otherwise, it will read in the filename into a vector of strings.

parseWinFile then iterates over every element of the vector (one line at a time). It is important that comments are stripped out at the appropriate time, otherwise the line count used in error messages can be wrong.

During the iteration process, if a single backslash is found, then it will continue to the next line without parsing any data. It will continue joining all extended lines together until no more backslashes are found, thus transforming a spanning description into a single line.

The function `.getParamFromStr` is used to convert and validate the complete widget line into the beginning of a widget list. Once the entire file has been converted into these lists, the function then rescans these lists looking for any of the following special widgets: `window`, `menu`, `menuitem`, and `grid`. `Window` widget data is extracted and stored in top level variables; menus are stored in a special list, with each menu containing a recursive `menuData` list that holds `menuitems` and sub-menus.

Whenever a `grid` widget is found, a `.widgets` element is created to hold a list of lists. This two dimensional list will hold the next `nrow*ncol` widgets. The functions that do this are recursive and are designed to handle grids nested in grids (nested in grids and so on…). Menus use similar recursive functions, except without the need for a two dimensional list.

2.3. Getting your feet wet

Here are a few examples you can try to get familiar with the functions used for parsing.

PBSmodelling:::.convertPararmStrToList("entry name=foo")
This simply breaks up the string into an unvalidated list. Try giving it data that is not a valid widget.

PBSmodelling:::.getParamFromStr("entry name=foo")
This includes a call to the above function, and then validates the returned results to the accepted arguments as defined in widgetDefs.r

2.4. An exercise

Try to add a widget called "mywidget" to PBS Modelling.

1. Create a list in the widgetDefs.r file named `.widgetDefs$mywidget`
2. Create a function called `.createWidget.grid` that have the parameters `(tk, widget, winName)`
   - tk – a tcl pointer to the parent tk object
   - widget – the list describing the widget
   - winName – the name of the window being created

   The return value must be a valid tcl/tk pointer to a widget.
3. Start with a fairly simple function that only displays a tklabel, such as:
```
.createWidget.XXXX <- function(tk, widget, winName)
{
     return(tklabel(parent=tk,
                    text="A limited widget"))
}
```
   You may want to include a call to str(widget) to display what sort of information is passed to this function.

2.5. Diving deeper into PBS Modelling

By this point you should be familiar with the main data types of PBS Modelling: widget lists, recursive widget lists (like grid and menu), and the global .PBSmod list.

The uses of these data types will become clearer as you explore the source code of PBS Modelling.

References

Daalgard, P. 2001. A primer on the R Tcl/Tk package. *R News* 1 (3): 27–31, September 2001. URL: http://CRAN.R-project.org/doc/Rnews/

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2006. PBS Modelling 1: User's Guide. Can. Tech. Rep. Fish. Aquat. Sci. 2674: viii + 112 p.

Wettenhall, J. 2004. R TclTk Examples
URL: http://bioinf.wehi.edu.au/~wettenhall/RTclTkExamples/

Appendix A: List of defined functions and objects

```
widgetDefs.r - defined objects
-------------------------------------------------------------------------------
.widgetDefs                 - list defining widget paramaters and default values
.pFormatDefs                - list defining accepted paramaters (and default
                              values) for "P" format of readList and writeList
-------------------------------------------------------------------------------

supportFuns.R - defined functions
-------------------------------------------------------------------------------
.initPBSoptions             - called from zzz.R .First.lib() intialization func
.addslashes                 - escapes special characters from a string
.writeList.P                - saves list to disk using "P" format
.readList.P                 - Read a list in P format
.readList.P.convertData     - convert data into proper mode
.mapArrayToVec              - determines which index to use for a vector, when
                              given an N-dim index of an array.
.getArrayPts                - Returns all possible indices of an array
.convertVecToArray          - converts a vector to an Array
.fibCall                    - interface C code via Call()
.fibC                       - interface C code via C()
.fibR                       - iterative fibonacci in R
.fibClosedForm              - closed form equation for fibonacci numbers


guiFuns.r - defined functions
-------------------------------------------------------------------------------
.trimWhiteSpace             - remove leading and trailing whitespace
.stripComments              - remove comments from a string
.inCollection               - find a needle in a haystack
.isReallyNull               - tests if a key exists in a list
.searchCollection           - searches a haystack for a needle, or a similar
                              longer needle.
.map.init                   - initialize the datastructure that holds the map(s)
                              A map is another name for hash table (an R list)
.map.add                    - save a new value for a given key, if no current
                              value is set
.map.set                    - force a save
.map.get                    - returns a value associated with a key
.map.getAll                 - returns all values
.extractVar                 - extracts values from the tclvar ptrs of a window
.PBSdimnameHelper           - add dimnames to objects
.convertMatrixListToMatrix - converts a list into an N-dim array
.convertMatrixListToDataFrame
                            - converts a list into a dataFrame
.setMatrixElement           - helper function used by .convertMatrixListToMatrix
                              to assign values from the list into the array
.getMatrixListSize          - determine the minumum required size of the array
                              needed to create to convert the list into an array
.matrixHelp                 - helper for storing elements in an N-dim list
.validateWindowDescList     - checks for a valid PBS Modelling description List
```

```
                                and sets any missing default values
.validateWindowDescWidgets - used by .validateWindowDescList to validate each
                                widget
.parsemenu                  - associate menuitems with menus
.parsegrid                  - associate items with a grid
.stripSlashes               - removes escape backslashes from a string
.stripSlashesVec            - convert a grouping of strings representing an
                                argument into a vector of strings
.convertPararmStrToVector  - convert a string representing data into
                                a vector. (used for parsing P format data)
.catError2                  - displays parse error (P data parser)
.convertPararmStrToList    - convert a string representing a widget into
                                a vector. (used for parsing description files)
.catError                   - displays parsing errors
.stopWidget                 - display and halt on fatal post-parsing errors
.getParamFromStr            - convert a string representing a widget into a list
                                including default values as defined in
                                widgetDefs.r
.buildgrid                  - used to create a grid on a window
.createTkFont               - creates a usable TK font from a given string
.createWidget               - generic function to create most widgets, which
                                calls appropriate function:
                                    .createWidget.grid
                                    .createWidget.check
                                    .createWidget.label
                                    .createWidget.null
                                    .createWidget.matrix
                                    .createWidget.vector
                                    .createWidget.data
                                    .createWidget.object
                                    .createWidget.entry
                                    .createWidget.radio
                                    .createWidget.slide
                                    .createWidget.slideplus
                                    .createWidget.button
                                    .createWidget.text
                                    .createWidget.history

.updatePBShistory           - update widget values
.extractFuns                - get a list of called functions
.extractData                - called directly by TK on button presses (or binds,
                                onchanges, slides, ...)
.setWinValHelper            - used by setWinVal to target single widgets
.convertMode                - converts a variable into a mode without showing
                                any warnings
.autoConvertMode            - converts x into a numeric mode, if it looks like
                                a valid number
```

Appendix B: R Package Development Time Savers

Creating software is an iterative process. Compile, fix syntax errors, re-compile, install package, test package, fix bug, and start over. Luckily R provides framework that compiles packages.

In my experience, installing packages through R's menu system can take up a lot of time if you have to install a package more than 20 times in a day. I have automated the process by using the following R script.

```
#select and install most recent version of the package
pkg <- sort(grep("^PBSmapping_.*\\.zip$",
         dir("C:\\DFO\\packages\\"), value=T), decreasing=T)[1]
cat("installing"); cat(pkg); cat("\n");
install.packages(paste("C:\\DFO\\packages\\",pkg,sep=""), .libPaths()[1], repos = NULL)
```

I have a copy of the script saved as `autoInstall.r` which is invoked by a batch file with the command:

```
R CMD BATCH autoInstall.r
```

This command can easily be inserted into a modified version of the build.bat file that comes with PBS Modelling.