Graphical User Interfaces

Introduction

In the world of Linux, there are several graphical toolkits available. Some famous examples are GTK, Qt, Motif, and FLTK. But there are many others which allow the creation of advanced user interfaces. Each of these toolkits follow their own logic and provide an API which is specific for their implementation.

The BaCon project tries to be as generic as possible and it strives to be compatible with all toolkits. But it is almost impossible to support such a diverse landscape. However, if BaCon would stick to a specific graphical toolkit then this would be a limitation.

To overcome this problem, BaCon follows a strategy which implements a simple, small and flexible API to embed the essential functions of a graphical toolkit. First of all, BaCon will always assume an object/property model. This means that each individual widget in a toolkit, like a window or a button or a text field, is considered to be an object with properties. Secondly, the particular naming of widgets and their respective properties will not be part of the BaCon syntax itself. Instead, a very small set of native high-level functions shall be able to define the GUI.

This approach works fine for X-based toolkits like Xaw, Xaw3d, Motif and GTK. But a lot of graphical toolkits use different design principles. Other toolkits already use an object/property model, like the TK toolkit. Because if this, TK has a slightly different embedding into BaCon.

Of course, BaCon can *impose* its object/property philosophy upon a toolkit like TK. However, the question is where that leads us. The code base required to forcefully embed TK into BaCon would be a lot bigger. Also, designing a graphical interface would become lot more tedious for the programmer. It would be very clumsy to define a GUI requiring multiple lines of code while the same thing can be done within TK in just a few statements. The TK toolkit is known for its flexibility, but this advantage would then disappear soon. Furthermore, we’re not really interested to force the universality of an object-property model upon all graphical user interfaces. At the end of the day, the goal is to see if BaCon can integrate graphical toolkits using a small set of functions.

In total, six functions are available to integrate with an external graphical toolkit. These are the following:

- `GUIDEFINE`
- `GUIEVENT$`
- `GUIFN`
- `GUISET`
- `GUIGET`
- `GUIWIDGET`

At the time of writing, BaCon supports the following toolkits: Xaw, Xaw3D, Motif with optional support for MRM and UIL, GTK2, GTK3, GTK4, TK and CDK.

The next paragraphs contain an overview of the purpose and meaning of these functions.

GUIDEFINE

The function `GUIDEFINE` accepts one string argument in which the user interface is defined. The string argument describes the objects with their properties, each object surrounded by curly brackets (except for the TK toolkit). A property consist of a name-value pair. A few of these names are BaCon specific:

- **type**
  Defines the object type (obligatory).
- **name**
  Defines the name for the object which the program may refer to in a later stage (obligatory).
- **parent**
  Defines the widget to which the current object will be attached (optional).
- **callback**
  Defines the name of the signal to which the object will respond. Multiple callbacks can be defined. By default, the signal will emit the name of the widget. Instead, and alternative string can be provided (optional, will be discussed later).
- **map**
  When the user interface is provided by a User Interface Language (UIL) definition then this function maps the signals to the keywords defined in the UIL. Mostly used by the Motif toolkit.
- **uid**
  When the user interface is provided by a User Interface Language (UIL) definition then this function sets module name defined in the UIL. Mostly used by the Motif toolkit.
- **resources**
  Defines external properties related to X, like coloring and font. Mostly used by Xaw, Xaw3d and Motif.
- **args**
  Defines the properties directly as arguments to the current object. Used by the Curses Development Kit (CDK).

The `GUIDEFINE` function returns an identifier which has to be used in the other GUI functions as a reference.

GUIEVENT$

Executes the main loop of the GUI and returns the name of the widget or the defined string from the callback definition. This function accepts the reference ID returned by `GUIDEFINE` . The optional second argument allows returning a pointer to data which was passed to the internal callback function (discussed later).

GUIFN

Calls a GUI helper function based on a previously defined function pointer. It allows to omit specific type casting and does not perform argument type checks providing more code flexibility. For the TK toolkit, `GUIFN` enables adding TCL code in the current namespace.

GUIGET

Fetches a value from a property of a widget. It sets the result into a pointer variable.

GUISET

Sets a property of a widget to a value.

GUIWIDGET

Returns the memory address of the widget in a GUI.

Hello World

To explain the aforementioned functions, we will build a simple "Hello World" application for all supported toolkits.

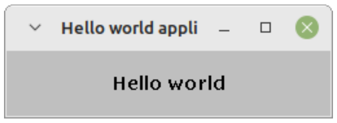
Xaw

The following code will setup a plain window in X Athena Widgets (Xaw):

```
OPTION GUI TRUE

id = GUIDEFINE(" \
{ type=window name=window XtNtitle=\"Hello world program\" XtNwidth=250 XtNheight=50 } \
{ type=labelWidgetClass name=label parent=window XtNlabel=\"Hello world\" }")

CALL GUIEVENT$(id)
```



BaCon can compile this code without any additional command line parameters.

The first line in the above program enables access to GUI programming. By default, BaCon will assume the Xaw toolkit. The reason for this is the generic availability of Xaw. It automatically is available when the package for the X environment is installed.

In the second line, the objects for the "Hello World" application are defined. The definition is contained in a single string which wraps to the next line. Each object is enclosed in curly brackets.

A user interface always is kept within a window. So the first object is defined as a window using the following property definition: “type=window”. The value “window” is a placeholder for Xaw. It is the only exception for this toolkit. The other widgets can be defined by their respective classes.

The window then gets a name assigned. We will use this name in the definition for the label.

The window also has a text in the title bar. In the above definition, the text will show “Hello world application”. It should be noted that this is defined by a toolkit specific property called “XtNtitle”. It may be unclear where this property comes from. As mentioned in the introduction, the particular naming of toolkit dependent properties is not part of BaCon itself. It will therefore remain undocumented in any BaCon reference manual. Instead, the programmer should lookup the toolkit dependent properties in the manual of that toolkit.

Similarly, the width and height of the window are defined by assigning values to toolkit dependent properties as well. The “XtNwidth” defines the width of the window in pixels, and “XtNheight” defines the height.

The next object is a label. The type is set to “labelWidgetClass”. This type can be looked up in the Xaw documentation as well.

Also the label gets a name assigned. The name can be used later in the program to refer to the label. This can be handy when the contents of the label needs to be changed, for example.

Then, the label should be put somewhere. The “parent” property defines the name of the widget or window on top of which the label will appear. In this example, it is the main window.

Lastly, the label should display some text. This is defined by the toolkit dependent property “XtNlabel”.

After defining the GUI, it should be displayed on the screen. The events caused by any user interaction should be caught and brought to the attention of the BaCon program. This is the purpose of the function `GUIEVENT$`. It contains the reference number of the previous GUI definition as an argument. The GUI is displayed on the screen and events are monitored and returned to the program.

In this simple example, any event simply will end the program and it will close the user interface.

Motif

A simple program to implement “Hello world” in Motif:

```
OPTION GUI TRUE
PRAGMA GUI motif

id = GUIDEFINE(" \
{ type=window name=window XmNtitle=\"Hello world program\" XmNwidth=250 XmNheight=50 } \
{ type=xmLabelWidgetClass name=label parent=window }")

CALL GUISET(id, "label", XmNlabelString, XmStringCreateLocalized("Hello world"))

CALL GUIEVENT$(id)
```

The structure of this program is similar to the Xaw version shown in the previous paragraph. However, after enabling GUI programming, there is a second line to define the type of GUI. In this case, the program uses the Motif toolkit. As mentioned, BaCon always assumes the Xaw toolkit by default, but the `PRAGMA` statement can override this assumption.

The `GUIDEFINE` function again contains one string with two objects: a window and a label. Note how the property names used by Motif are slightly different compared to Xaw.

The `GUISET` function is new. As Motif expects a localized string with text, BaCon needs to set it explicitly after the GUI definition. The arguments are the reference ID of the user interface, the actual name of the label, the property to set and a Motif function to create a localized string. Details for the Motif properties can be looked up in the related documentation.

GTK

BaCon supports GTK version 2, 3 and 4.

```
OPTION GUI TRUE
PRAGMA GUI gtk4

id = GUIDEFINE(" \
{ type=WINDOW name=window callback=destroy title=\"Hello world program\" \
    width-request=250 height-request=50 } \
{ type=LABEL name=label parent=window label=\"Hello world\" }")

CALL GUIEVENT$(id)
```

This program will work for all versions of GTK. The `PRAGMA` statement defines the GTK version. In the above example, it is defined as GTK4. Instead, it also is possible to use “gtk2” or “gtk3” as well.

Again the `GUIDEFINE` function contains a string with two objects: a window and a label. Note that the “type” is set to the GTK definition of its known object types. In the GTK documentation, the definition for a window is set to “GTK_WINDOW”, and a label to “GTK_LABEL”. The BaCon “type” property allows omitting the “GTK_” prefix. However, the property should still use capitals.

The “callback” property is new. It defines the signal to which the window will listen. In the above program, the window will listen to the “destroy” signal. The name of the signal should be looked up in the GTK documentation. The “destroy” signal occurs when the user presses on the “close” symbol top-right of the window. The definition will return the name of the main window to the BaCon program. In this case, the program ends and the returned name of the window is ignored.

TK

BaCon supports the TK toolkit from the TCL project.

```
OPTION GUI TRUE
PRAGMA GUI tk

id = GUIDEFINE(" \
wm title . {Hello world program}; \
wm protocol . \"WM_DELETE_WINDOW\" window; \
wm geometry . 250x50; \
label .lbl -text {Hello world}; \
place .lbl -relwidth .35 -relx .35 -relheight .35 -rely .35")

CALL GUIEVENT$(id)
```

As explained in the introduction, TK has a slightly different embedding into BaCon compared to the other toolkits. The above program demonstrates that the elements of the user interface are defined in a manner suitable for the TCL/TK interpreter. It therefore does not enclose objects in curly brackets, but sends the full GUI definition to the TCL/TK interpreter instead. It already is formatted in a object/property model similar to the way BaCon uses for other toolkits.

Clearly, a thorough knowledge of the TK toolkit is required. The actual meaning of the string argument for `GUIDEFINE` can be understood when reading the documentation for TK.

Having said that, it should be clear that the global structure of the BaCon program is similar to the previous example programs. In case of an event, the `GUIEVENT$` function will return a defined text just the same. In this example, the “WM_DELETE_WINDOW” event will return the text “window”, however, when returned, the program simply will end.

Callbacks

When programming a graphical user interface, the program should be able to process user events. BaCon can return user events back to the program by using the `GUIEVENT$` function.

To allow a response from a user, the widget needs to set a “callback” property which defines the actual signal. By default, events will return the name of the widget.

In the next examples, we will expand the “Hello World” program with two buttons. One button will change the text in the label, and the other will exit the program.

Xaw

This is an enhanced version of the Hello World application, containing two buttons:

```
OPTION GUI TRUE

id = GUIDEFINE(" \
{ type=window name=window XtNtitle=\"Hello world program\" XtNwidth=250 XtNheight=150 } \
{ type=formWidgetClass name=form parent=window } \
{ type=labelWidgetClass name=label parent=form XtNlabel=\"Hello world\" XtNwidth=200 } \
{ type=commandWidgetClass name=button parent=form callback=XtNcallback XtNfromVert=label \
    XtNfromHoriz=NULL XtNwidth=100 XtNheight=40 XtNlabel=\"Click me\" \
    XtNleft=XawChainLeft XtNright=XawChainLeft XtNtop=XawChainBottom \
    XtNbottom=XawChainBottom } \
{ type=commandWidgetClass name=exit_b parent=form callback=XtNcallback XtNfromVert=label \
    XtNfromHoriz=button XtNwidth=100 XtNheight=40 XtNlabel=\"Exit\" \
    XtNleft=XawChainRight XtNright=XawChainRight XtNtop=XawChainBottom \
    XtNbottom=XawChainBottom XtNhorizDistance=100 }")

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "button"
            CALL GUISET(id, "label", XtNlabel, "Goodbye!")
        CASE "exit_b"
            BREAK
    ENDSELECT
WEND
```

The `GUIDEFINE` definition now also mentions a “form” which is attached to the window. On top of this form, the program attaches the label and two buttons. This logic is specific to the X Athena Widgets toolkit and should be looked up in their documentation. The important thing in this example is to observe how callbacks work in a BaCon program.

The `GUIDEFINE` function mentions the “callback” property for each button. It is configured as “XtNcallback” which will instruct the Xaw toolkit to respond to a user event. As mentioned before, BaCon will return the name of the widget by default.

In the `WHILE`/`WEND` loop, each event coming back from the Xaw toolkit is examined. If either the button with the name “button” is clicked or the button with the name “exit_b”, the program will perform an action.

In case the name “button” comes back, the program will set the “XtNlabel” property of the label. The BaCon function `GUISET` is used to perform such action. In the example, `GUISET` uses four arguments. The first argument refers to the ID of the GUI. The second argument refers to the name of the widget for which the property is set. The third argument contains the name of the property and the fourth argument sets the value for that property.

In case the name “exit_b” comes back, it simply breaks out the endless loop, effectively terminating the program.

Motif

This is the “Hello World” program with two more buttons:

```
OPTION GUI TRUE
PRAGMA GUI motif

id = GUIDEFINE(" \
{ type=window name=window XmNtitle=\"Hello world program\" XmNwidth=250 XmNheight=150 } \
{ type=xmFormWidgetClass name=form parent=window } \
{ type=xmLabelWidgetClass name=label parent=form XmNalignment=XmALIGNMENT_CENTER } \
{ type=xmPushButtonWidgetClass name=button parent=form callback=XmNactivateCallback \
    XmNbottomAttachment=XmATTACH_FORM XmNleftAttachment=XmATTACH_FORM XmNheight=30 \
    XmNwidth=90 } \
{ type=xmPushButtonWidgetClass name=Exit parent=form callback=XmNactivateCallback \
    XmNbottomAttachment=XmATTACH_FORM XmNrightAttachment=XmATTACH_FORM XmNheight=30 \
    XmNwidth=90 }")

CALL GUISET(id, "label", XmNlabelString, XmStringCreateLocalized("Hello world"))
CALL GUISET(id, "button", XmNlabelString, XmStringCreateLocalized("Click Me"))

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "button"
            CALL GUISET(id, "label", XmNlabelString, XmStringCreateLocalized("Goodbye!"))
        CASE "Exit"
            BREAK
    ENDSELECT
WEND
```

The above program achieves the same thing but uses Motif. Note that the name of the button by default appears as text on the button, but also can be defined explicitly using `GUISET`.

The callback signal for the buttons is defined as “XmNactivateCallback”, which is a toolkit dependent setting.

GTK

The expanded GTK version:

```
OPTION GUI TRUE
PRAGMA GUI gtk3

id = GUIDEFINE(" \
{ type=WINDOW name=window callback=delete-event title=\"Hello world program\" \
    width-request=250 height-request=50 } \
{ type=BOX name=vbox parent=window orientation=GTK_ORIENTATION_VERTICAL } \
{ type=LABEL name=mylabel parent=vbox label=\"Hello world\" } \
{ type=BUTTON_BOX name=bbox parent=vbox layout-style=GTK_BUTTONBOX_EDGE } \
{ type=BUTTON name=button parent=bbox callback=clicked margin=5 label=\"Click Me\" } \
{ type=BUTTON name=exit_b parent=bbox callback=clicked margin=5 label=\"Exit\" }")

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "button"
            CALL GUISET(id, "mylabel", "label", "Goodbye!")
        CASE "exit_b", "window"
            BREAK
    ENDSELECT
WEND
```

The above program only works for GTK3. The GTK example from the previous chapter worked with all versions of GTK. However, there are many subtle changes and differences between various GTK versions. It is almost impossible to create a GUI which works for all GTK versions without modification of code.

The program shows a callback for the main window which is defined as “delete-event”. As always, this is a toolkit dependent setting. The main loop at the end of the program will be notified when this event occurs. Both the name of the window and the name of the “Exit” button will break out the endless loop.

TK

The following example shows callbacks in a TK program:

```
OPTION GUI TRUE
PRAGMA GUI tk

id = GUIDEFINE(" \
    wm title . {Hello world program}; \
    wm protocol . \"WM_DELETE_WINDOW\" window; \
    label .lbl -justify center -text {Hello world}; \
    button .btn -text \"Click me\" -command btn; \
    button .exit_b -text \"Exit\" -command exit_b; \
    grid .lbl -row 0 -column 0 -columnspan 2 -padx 5 -pady 5; \
    grid .btn -row 1 -column 0 -padx 5 -pady 5 -sticky w; \
    grid .exit_b -row 1 -column 1 -padx 5 -pady 5 -sticky e;")

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "btn"
            CALL GUIFN(id, ".lbl configure -text {Goodbye!}")
        CASE "exit_b", "window"
            BREAK
    ENDSELECT
WEND
```

Again, it should be noted that the TK toolkit has a slightly different embedding into BaCon. The GUI definition consists of one string which is sent to the TCL/TK interpreter. BaCon examines this string to determine the signals for a callback. When a widget is configured with a “-command” parameter then BaCon will take the argument for this parameter as a string to be passed back to the program, unless that argument is enclosed in curly brackets. In that case, BaCon will skip the definition and leave the handling of the “-command” parameter to the TCL/TK interpreter.

Next to the “-command” parameter, BaCon also will verify any “bind” TK command. The occurrence of “bind” will be considered a callback as well, and BaCon will return the 4th parameter to “bind” as a string back to the program, unless this parameter is enclosed in curly brackets.

The main loop verifies the strings coming back when a user event has occurred. Note that due to the interpreted nature of TK and its tight connection to TCL, setting a property of a widget works in a different way. BaCon can execute TCL/TK code directly to achieve a similar result. In the example, the `GUIFN` function is used for this purpose. This function can contain any TCL or TK code, as long as it refers to the same ID returned by the `GUIDEFINE` function.

For all toolkits is should be clear that additional knowledge of the particular design of a toolkit is required. Most documentation can be found online and is not in scope of this BaCon manual.

Callback renaming

From the previous examples, it became clear that the callback mechanism will return the name of a widget when a signal occurs. However, when we want to define multiple signals for the same widget, then for all events the same widget name will be passed back to the main loop. The BaCon program then cannot see which event actually has happened.

This is where the callback renaming comes in. When defining a callback with `GUIDEFINE`, the callback property defines signal for the widget. This property supports additional information, which can be added after a comma. The BaCon event handler for the toolkit will then return this information instead of the name of the widget. The following Motif program demonstrates this principle:

```
OPTION GUI TRUE
PRAGMA GUI motif

prompt = GUIDEFINE( " \
{ type=window name=window } \
{ type=transientShellWidgetClass name=form parent=window XmNtitle=\"Info\" } \
{ type=xmMessageBoxWidgetClass name=dialog parent=form callback=XmNcancelCallback,cancel \
     callback=XmNokCallback,ok callback=XmNhelpCallback,help \
     XmNdialogType=XmDIALOG_INFORMATION }")

CALL GUISET(prompt, "dialog", XmNmessageString, XmStringCreateLocalized("Hello world"))
CALL XtPopup(GUIWIDGET(prompt, "form"), XtGrabNone)

WHILE TRUE
    SELECT GUIEVENT$(prompt)
        CASE "ok"
            PRINT "OK clicked"
        CASE "help"
            PRINT "HELP clicked"
        CASE "cancel"
            PRINT "Cancel clicked"
            BREAK
    ENDSELECT
WEND
```

This example creates a dialog box with multiple buttons. The dialog is one object for which multiple callback signals must be defined, one for each button. The callback property sets the name of the signal, a comma, and a string which will be passed back to the main loop. Note that in the definition, the signal name, comma and additional information should all be attached together, without spaces.

The following is an example using GTK, distinguishing between button pressed and released events:

```
OPTION GUI TRUE
PRAGMA GUI gtk3
```

```
id = GUIDEFINE(" \
{ type=WINDOW name=window callback=delete-event title=\"Hello world program\" \
    width-request=250 height-request=50 } \
{ type=BOX name=vbox parent=window orientation=GTK_ORIENTATION_VERTICAL } \
{ type=LABEL name=mylabel parent=vbox label=\"Hello world\" } \
{ type=BUTTON_BOX name=bbox parent=vbox layout-style=GTK_BUTTONBOX_EDGE } \
{ type=BUTTON name=button parent=bbox callback=pressed,press callback=released,unpress \
    margin=5 label=\"Click Me\" } \
{ type=BUTTON name=exit_b parent=bbox callback=clicked margin=5 label=\"Exit\" }")

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "press"
            PRINT "Button pressed"
        CASE "unpress"
            PRINT "Button released"
        CASE "exit_b", "window"
            BREAK
    ENDSELECT
WEND
```

In this example, the signals for “pressed” and “released” are renamed, so different strings will be passed back to the main loop for each event on the same button widget.

Callback codes (TK)

In case of TK, BaCon will setup a callback mechanism in case it either discovers a “-command” widget definition or detects a TK “bind” command. The latter command however has a peculiarity which should be taken into account. The way TK works is that some widgets already contain a set of default bindings. For example, the text widget already has a lot of key combinations defined. The famous <CTRL>+<c> will copy text, <CTRL>+<v> will paste the text, <CTRL>+<a> will select all text, and so on. If a “bind” command adds a callback to a key combination, then TK will execute this user definition first, but it also will continue to perform the default bindings. To prevent this from happening, the 4th argument to the “bind” command allows additional information. It should be added after a “+” sign and without any spaces in between:

```
bind .textwidget <Control-KeyPress-m> callback+break;
```

This line of code connects the <CTRL>+<m> key combination to the string “callback”. The additional “break” will make sure that the event is not propagated further to the text widget. This prevents a <return> symbol being inserted somewhere in the text.

So after the event occurs in TK, the callback now will return a break to the toolkit. Various return codes are possible, causing different results. The programmer can choose between “error”, “return”, “break”, or “continue”. These correspond to the TCL_ERROR, TCL_RETURN, TCL_BREAK and

TCL_CONTINUE return codes. If no additional information is specified then Bacon assumes “ok” (TCL_OK).

Advanced events

In some cases it may be necessary to obtain a special value which is passed to the callback by the widget library. In such situation, the `GUIEVENT$` function accepts an optional second boolean parameter. This will add a pointer to the internal callback value which is attached as a string to the returned result. The format of the result is a delimited string which can be processed using default BaCon functions.

For example, to obtain the selected item in a XawList widget, the Xaw library returns an internal pointer to a C-struct containing information about the XawList. This can be fetched as follows:

```
OPTION GUI TRUE

DECLARE info TYPE XawListReturnStruct*
DECLARE data$[3]

id = GUIDEFINE(" \
{ type=window name=window XtNtitle=\"Hello world program\" XtNwidth=250 XtNheight=120 } \
{ type=formWidgetClass name=form parent=window } \
{ type=viewportWidgetClass name=view parent=form XtNwidth=250 XtNallowVert=True \
    XtNleft=XawChainLeft XtNuseRight=True XtNright=XawChainRight \
    XtNtop=XawChainTop XtNbottom=XawChainTop } \
{ type=listWidgetClass name=list parent=view callback=XtNcallback XtNverticalList=True \
    XtNforceColumns=True XtNdefaultColumns=1 XtNleft=XawChainLeft XtNright=XawChainRight \
    XtNtop=XawChainTop XtNbottom=XawChainBottom } \
{ type=commandWidgetClass name=exit_b parent=form callback=XtNcallback XtNfromVert=list \
    XtNwidth=100 XtNheight=40 XtNlabel=\"Exit\" XtNleft=XawChainRight \
    XtNright=XawChainRight XtNtop=XawChainBottom XtNbottom=XawChainBottom \
    XtNhorizDistance=100 }")

data$[0] = "Hello"
data$[1] = "World"
data$[2] = NULL

CALL GUISET(id, "list", XtNlist, data$, XtNnumberStrings, 0)

WHILE TRUE
    event$ = GUIEVENT$(id, TRUE)
    SELECT TOKEN$(event$, 1)
        CASE "exit_b"
            BREAK
        CASE "list"
            info = (XawListReturnStruct*)DEC(TOKEN$(event$, 2))
            PRINT TOKEN$(info->string, 1)
    ENDSELECT
WEND
```

A little bit of C knowledge is required to understand what is going on. When the second argument to `GUIEVENT$` is set to `TRUE` then the return string also contains a pointer which directs to the data from the list widget. After converting this pointer to the appropriate C-struct, the BaCon program can obtain the value of the item which was selected by the user.

This advanced functionality of `GUIEVENT$` also can be used in case of GTK dialogues. The next program can verify which button was pressed in a default GTK dialog:

```
OPTION GUI TRUE
PRAGMA GUI gtk3

DECLARE (*show)(GtkWidget*) = gtk_widget_show_all TYPE void
DECLARE (*hide)(GtkWidget*) = gtk_widget_hide TYPE void

DECLARE resp TYPE int

id = GUIDEFINE(" \
{ type=WINDOW name=window callback=delete-event title=\"Hello world program\" \
    width-request=250 height-request=50 } \
{ type=BOX name=vbox parent=window orientation=GTK_ORIENTATION_VERTICAL } \
{ type=LABEL name=mylabel parent=vbox label=\"Hello world\" } \
{ type=BUTTON_BOX name=bbox parent=vbox layout-style=GTK_BUTTONBOX_EDGE } \
{ type=BUTTON name=button parent=bbox callback=clicked margin=5 label=\"Click Me\" } \
{ type=BUTTON name=exit_b parent=bbox callback=clicked margin=5 label=\"Exit\" } \
{ type=MESSAGE_DIALOG name=dlg callback=delete-event callback=response,yesno \
    message-type=GTK_MESSAGE_WARNING buttons=GTK_BUTTONS_YES_NO \
    title=\"Question\" text=\"Are you sure?\" }")

WHILE TRUE
    event$ = GUIEVENT$(id, TRUE)
    SELECT TOKEN$(event$, 1)
        CASE "button"
            CALL GUIFN(id, "dlg", show)
        CASE "yesno"
            resp = *(intptr_t*)DEC(TOKEN$(event$, 2))
            IF resp = GTK_RESPONSE_YES THEN PRINT "Clicked on YES"
            IF resp = GTK_RESPONSE_NO THEN PRINT "Clicked on NO"
            CALL GUIFN(id, "dlg", hide)
        CASE "exit_b", "window"
            BREAK
    ENDSELECT
WEND
```

The additional GTK pointer attached to the result is converted to an integer value and then assigned to a variable. Then it is just a matter of finding out which button was pressed.

Clearly, these constructions require some knowledge of the respective toolkits and of the way C handles pointers. Fortunately, the programmer of graphical user interfaces will likely not run into other advanced events. The demonstration programs available at the BaCon website do not contain a usage of the `GUIEVENT$` function other than mentioned in the previous examples.

The last program also demonstrates how the `GUIFN` function works. This will be explained in more detail in the next chapter.

Helper functions

A lot of times it is useful to add native GUI functions because the property we're looking for simply is not available. Some toolkits only allow to set a certain property by using a C function in their API. In BaCon, it is allowed to mix C code with regular statements. But a lot of times this looks ugly and confusing. Also, the BaCon program needs to declare the correct argument types for the API function. This can be a difficult undertaking.

But there are other situations for which the C API of a toolkit has to be used. For example, it sometimes may happen that the program needs to popup a dialog window, or hide that window again. Showing and hiding dialog windows typically is an action performed by a C function from the toolkit being used. The BaCon program then has to mix its code with the API of the toolkit.

This is where the `GUIFN` function comes in. With `GUIFN` it is possible to define a supplementary helper function. This creates a generic interface by embedding a function pointer to the API we need to invoke. Here is an example of how it works using the Xaw toolkit:

```
DECLARE (*show)() = XtPopup TYPE void
DECLARE (*hide)() = XtPopdown TYPE void
```

In the example, two function pointers are defined to point to two native C functions from the Xaw toolkit: XtPopup and XtPopdown. These functions are used to show and hide dialogs. The BaCon program then can use `GUIFN` to invoke these native functions in an unambiguous manner:

```
CALL GUIFN(id, "window", show, XtGrabNonexclusive)
```

The `GUIFN` function always requires at least 3 arguments:

- the id of the GUI, which is created by `GUIDEFINE`;
- the name of the widget to which the function will be applied;
- the name of the function pointer, previously declared in a DECLARE or LOCAL statement.

In the example, `GUIFN` has a 4th argument, "XtGrabNonexclusive", required by the native XtPopup function. It relates to the way the dialog will be shown. If the C function requires more arguments, then these can be added as well.

To demonstrate closing the window again:

```
CALL GUIFN(id, "window", hide)
```

The native XtPopdown function only requires one argument, which is the widget that needs to be hidden.

At first glance, using `GUIFN` may look a bit redundant. Of course, the programmer is always free to not use `GUIFN` and mix BaCon code with native API functions. However, the `GUIFN` function has several advantages:

- it is compliant with the overall API design of BaCon
- the BaCon program becomes smaller in size
- the BaCon program becomes more readable
- no need to worry about the argument types of the helper function

It is important to observe that the `GUIFN` function is preceded by the `CALL` statement. The reason is that native BaCon functions cannot appear standalone in the program, because then they would be a plain statement.

For the TK toolkit the situation is a little different. As TK and TCL are connected to closely, it is often more convenient to perform complicated operations in the TCL/TK interpreter itself, by using a few native TCL statements. Being an interpreter, the TK toolkit can accept this code *during runtime*, which provides a very powerful way to manipulate a user interface. Therefore, the `GUIFN` function works in such a way that it can directly enter native TCL/TK code into the interpreted TK environment.

For example:

```
CALL GUIFN(id, "tk_messageBox -title {Demo} -message {Hello world} -icon info -type ok")
```

As with the previous example in Xaw, the `GUIFN` function also needs to provide the id of the GUI as its first argument.

Then, the one line of TK code performs a few actions: it creates a dialog, defines a message to the user, it will popup that dialog as well, and shows an “ok” button, which, when pressed, lets the dialog disappear again. While the other toolkits need multiple lines of code to define a window, button and label, and code to handle the events, the TK toolkit can manage a dialog with just one line of native TK code.

The BaCon source package contains fully working code editors written both for the GTK and the TK toolkits. The implementation of these editors clearly shows how to use the `GUIFN` function, demonstrating the advantages of defining function pointers (GTK) and native interpreted code (TK).

Setting and getting

The generic GUI functions in BaCon try to follow an object/property model. It therefore makes sense to set properties of objects (widgets). Also, the BaCon program might be interested in the current value of the property and needs a way to fetch a value. These actions can be implemented by the `GUISET` and `GUIGET` functions.

For example, in GTK the user interface might contain an entry where some text can be inserted. When this entry needs to be filled with a text, this can be achieved by setting the "text" property of the widget:

```
OPTION GUI TRUE
PRAGMA GUI gtk3

id = GUIDEFINE(" \
{ type=WINDOW name=window callback=delete-event title=\"Entry Demo\" width-request=250 } \
{ type=BOX name=vbox parent=window orientation=GTK_ORIENTATION_VERTICAL } \
{ type=ENTRY name=entry parent=vbox margin=5 } \
{ type=BUTTON_BOX name=bbox parent=vbox layout-style=GTK_BUTTONBOX_EDGE } \
{ type=BUTTON name=setup parent=bbox callback=clicked margin=5 label=\"Set Text\" } \
{ type=BUTTON name=fetch parent=bbox callback=clicked margin=5 label=\"Get Text\" }")

DECLARE text$

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "setup"
            CALL GUISET(id, "entry", "text", "Fill in text")
        CASE "fetch"
            CALL GUIGET(id, "entry", "text", &text$)
            PRINT text$
        CASE "window"
            BREAK
    ENDSELECT
WEND
```

In the above program, the first argument of the `GUISET` function refers to the GUI id returned by `GUIDEFINE`. The second argument mentions the name of the widget. The third argument sets the name of the property. This property name can be looked up in the documentation of the GTK toolkit. The last argument defines the actual text which should be inserted into the entry widget.

Obtaining the text from an entry widget comes down to simply fetching the value of the "text" property of the widget. For this, BaCon provides the `GUIGET` function. The arguments have the same purpose, except for the last argument, indicating where the result is stored. Note the ampersand '&' symbol in the argument. This means that the variable is referred to *by reference*, and not by value. As a result, de variable will point to an existing memory address containing the text, instead of being assigned a fresh block of memory where the text will be copied to.

Again, in TK things work different. In this toolkit, the `GUISET` and `GUIGET` functions are used to exchange the values of variables between BaCon and TK.

This is small TK program demonstrates how it works:

```
OPTION GUI TRUE
PRAGMA GUI tk

id = GUIDEFINE(" \
    wm title . {Entry Demo}; \
    wm protocol . \"WM_DELETE_WINDOW\" window; \
    entry .ent -justify left; \
    focus .ent; \
    button .btn1 -text \"Set Text\" -command setup; \
    button .btn2 -text \"Get Text\" -command fetch; \
    grid .ent -row 0 -column 0 -columnspan 2 -padx 5 -pady 5; \
    grid .btn1 -row 1 -column 0 -padx 5 -pady 5 -sticky w; \
    grid .btn2 -row 1 -column 1 -padx 5 -pady 5 -sticky e;")

DECLARE text$

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "setup"
            text$ = "Fill in text"
            CALL GUISET(id, "txt", text$)
            CALL GUIFN(id, ".ent insert 0 $txt")
        CASE "fetch"
            CALL GUIFN(id, "set txt [.ent get]")
            CALL GUIGET(id, "txt", text$)
            PRINT text$
        CASE "window"
            BREAK
    ENDSELECT
WEND
```

First, the BaCon variable “text$” is assigned some text. Then, the contents of that variable is assigned to a variable in TK. Lastly, that TK variable is used to update the “entry” widget in the user interface. Of course, in this simple case it could have been done faster by simply concatenating the text to the `GUIFN` function argument. But often it is unknown what value a variable contains. In any case, it is just an example to demonstrate the principle of using `GUISET` in TK.

The `GUIGET` function works in a similar way, except it fetches a value from the TCL/TK interpreter and passes it on to the BaCon program.

The widget ID

The last GUI function to discuss is `GUIWIDGET`. This function can obtain the memory address of the widget. It sometimes is necessary to refer to the memory address when configuring a graphical user interface. Also it can come handy when mixing BaCon code with C API functions from the toolkit.

In the following example, a button is put on a GtkFixed container after the definition in `GUIDEFINE`:

```
OPTION GUI TRUE
PRAGMA GUI gtk3

DECLARE (*put)() = gtk_fixed_put TYPE void

id = GUIDEFINE(" \
{ type=WINDOW name=window callback=delete-event title=\"Demo\" width-request=300 } \
{ type=FIXED name=fixed parent=window } \
{ type=BUTTON name=button callback=clicked margin=10 label=\"Click here\" }")

CALL GUIFN(id, "fixed", put, GUIWIDGET(id, "button"), 100, 50)

WHILE TRUE
    SELECT GUIEVENT$(id)
        CASE "window"
            BREAK
        CASE "button"
            PRINT "Button clicked"
    ENDSELECT
WEND
```

Here, as always the GUI is defined by `GUIDEFINE`. But GTK does not allow to set the position of the button using properties. Therefore, the button has no "parent" property defined. Instead, the button needs to be put onto the GtkFixed container with an explicit call to the C function "gtk_fixed_put". The above program first defines a helper function to wrap it into the more generic `GUIFN` function. However, referring to the button requires knowledge of its memory address. This address can be obtained by the use of `GUIWIDGET`.

The first argument of the `GUIWIDGET` function is the id of the created GUI. The second argument mentions the name of the button. This way the memory address of each widget can be retrieved.

For the TK toolkit, `GUIWIDGET` has no purpose. It simply returns the name of the widget.