EuroForth 2024

A Forth binding for GTK4

Abstract

For some years, a Forth binding to major version 3 of the widely used graphical user interface toolkit GTK has been available. The major version 4 of GTK introduces many incompatibilities, so that a completely different approach to the binding is needed. It will be shown how the unique features of Forth can be leveraged to overcome the difficulties introduced by GTK4.

N.J. Nelson B.Sc. C.Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

GTK is a popular toolkit for user interfaces. It was introduced in 1998 and remains under active development.

The developers have a policy of not maintaining backward compatibility across major versions, in favour of innovation.

Bindings and wrappers for major versions 2 and 3 have been available for some time, for VFX Forth.

The current version 4 of GTK has some very useful new features, which encourage application developers to switch to this version.

Unfortunately, in creating version 4, the developers of GTK made no provision for a relatively unusual language such as Forth. This resulted in the need for major changes in the bindings and wrapper code. Some particular difficulties have been solved in a way that would be possible only in Forth.

2. Overview of existing technique - GTK3

Application GUI elements such as windows and dialog boxes are designed using an interface builder program. Although there have been several of these available over the years, in practice only the program "Glade" was developed to full functionality. this produces XML type files with the .glade extension.

At an early stage in the compilation process, after compiling the GTK bindings and wrappers, the glade files are read in using functions in the GtkBuilder class. This creates all the graphical elements.

A Forth wrapper word then scans the list of all these elements and creates a Forth VALUE word for every named element. These words can then be used later in the code to manipulate the GUI, and to define all the callbacks using the CallProc: function.

Just as with Windows, GTK interacts entirely using callbacks. When all the GTK code has been compiled, the signals (e.g. button pressed) that were specified in the Glade design, can be automatically connected to the corresponding Forth CallProc: function, using another GtkBuilder function.

The main application window is then displayed. The function gtk_main word is then called, to run an infinite loop, passing signals and events as required, until one of them called gtk_main_quit. This was in accordance with the example code originally given by GTK3.

As GTK can operate only in a single thread, in order to preserve Forth interactivity while GTK is running, an extra thread was created to accept Forth input from the terminal.

3. Changes to the technique required for GTK4

3.1 Changes to the Interface Builder Program

The Glade interface builder program cannot produce XML files that are compatible with GTK4. Instead, a new builder program "Cambalache" (by the same developers as Glade, but looking rather different) can be used. This uses an intermediate file format - the format with a .ui extension compatible with GTK4 is produced using export. An interesting feature is that the builder components are more modular within Cambalache itself - the user interfaces for even a complex application could be accommodated into a single XML file. On the other hand, external modularity would be lost.

3.2 Changes to GtkBuilder

There is a major and most unfortunate change to the GtkBuilder class. It attempts to connect signals at the same time as creating the elements. This is non optional. By default, it tries to connect using the C symbol table. To accommodate other languages, there is a new class "GtkBuilderScope", but this is only briefly documented, with no examples. Other languages have already implemented GTK4 support, so I chose the open source language Rust to investigate how they implemented signal bindings. Unfortunately, the code for GtkBuilderRustScope is very complex. It did not look like the kind of solution that a Forth programmer would devise.

3.3 Changes to the GTK main loop

As GTK3 developed, the example code changed to recommend the use of a new GtkApplication class, which hid the main loop functions. The stated aim was the make it easier for application developers, but of course for a language like Forth this technique could not be used because Forth itself is always the application. It was most disconcerting to discover that in GTK4, the main loop functions gtk_main and gtk_main_quit had actually disappeared.

3.4 Changes to widget naming

In GTK3, widgets could optionally have an ID and / or a "name". The ID was used to refer to the widget in code. The second was used to uniquely identify the widget in CSS. Confusingly, the function gtk_buildable_get_name actually returned the ID.This function has disappeared in GTK4. This was very concerning because if we were unable to get at the name, we could not automatically create Forth VALUEs.

4. New solutions

4.1 Forth style solution to the signal connection problem

The issue about GtkBuilder performing a mandatory attempt at signal connection at the same time as creating the GTK widgets seemed to be insoluble. After a great deal of thought, we analysed our existing workflow, as follows:

a) Define the signal in Glade, usually naming it using the convention on_<widget-id>_<signal name>.
e.g. on_mybutton_clicked

b) Define the signal action, for the example above, typically:
2 0 CallProc: on_mybutton_clicked { pbutton puser -- }
...
;

We realised that we were typing the on_mybutton_clicked twice, and that the parameter list is duplicated in the 2 0 and the { pbutton puser -- }. Duplication of effort is not a Forth thing to do.

Perhaps, we could turn the problem to our advantage, by eliminating the signals completely from the Cambalache design program and instead, inventing a new word for signal definitions, using techniques that could only be done in Forth. As always, we looked at the desired end result first.

```
MYBUTTON CLICKED SIG: { pbutton puser -- } \ Defined the action for mybutton clicked
...
;

...

: MAIN ( --- ) \ Main function
...
  CONNECTUISIGS      \ Connect UI signals
...
;
```

Notice that in the example above, we save around 20 characters of typing per signal. This does not sound much, until one realises that in our main application, there are 594 signal definitions, and if we had used this technique from the start, we could have saved around 12,000 characters of code.

In order to implement this, we first have to define each type of signal, in the Gtkbindings.fth file.

```
\ *********************************************************************
\ Signal descriptions - all are ( ---z$,numins,numouts )
\ *********************************************************************

: CLICKED       Z" clicked"     2 0 ;       \ GtkButton clicked

...
```

Now we can define the signal handler word.

```
VARIABLE SIGLIST        \ Root of UI signal list

: SIG: { pwidget psigname pins pouts | pelement pentry pcb pxt -- }
\ Compiles signal response code and adds to connection list
  pwidget GETUIELEMENT -> pelement                      \ Get UI element from value
  pelement 0= ABORT" Unrecognised widget in SIG:"       \ Only valid UI elements
  HERE 7 CELLS+ -> pentry
\ 1 for link, 3 for chain data, 3 for callback prelude = 7
  SIGLIST LINK,                                         \ Add address to signal list
  pelement ,                                            \ Compile UI element
  psigname ,                                            \ Compile signal name
  pentry   ,                                            \ Compile callback entry
\ Make callback data structure, save structure address
  pins pouts CALLBACK, -> pcb
  :NONAME -> pxt                                        \ Compile the sig definition
\ Insert the call to the signal definition into the callback structure
  pxt pcb @ SET-CALLBACK
;
```

There is actually one redundant element in the chain data, in that the entry address is numerically related to the link itself. We have retained this for readability.

A slight complication is that in our application, we need to do things when we detect user inactivity (for example, automatically log users off). This requires a hook in the callback prelude in VFX, which results in a change in the magic number "7" in the above code. We will be allowing for this by making a deferred word for the entry position, so that it can be easily redefined when the hook is inserted.

Now we define the signal connection word, using the signal list we have created.

```
: CONNECTUISIGS ( --- ) \ Connect UI signals
  SIGLIST @                     \ Start of signal list
  BEGIN
  ?DUP WHILE                    \ There are still items in the list
    DUP CELL+                   \ Pointer to signal data
    DUP @ EXECUTE               \ Get object
    OVER CELL+ @                \ Get signal name
    ROT 2 CELLS+ @              \ Get callback entry point
    0                           \ User data (none)
    g_signal_connect DROP       \ Connect signal
    @                           \ Move to next item in list
  REPEAT
;
```

Note that the above solution does not allow for user data in the signal definition. All GTK signals allow for this, though it is required in only about 5% of actual code. A slightly extended SIGUSER: word could be defined, with an additional element in the chain for the user data, and SIG: could then be redefined as 0 SIGUSER:.

4.2 Maintaining interactive Forth during GTK execution

Instead of "going by the book", we looked more deeply into what was needed for both an interactive (debug) program, and an executable.

For the interactive version, we went back to the same technique that was used for the very first binding, for GTK2, which was written by MPE. Despite the known aversion to hooks in VFX, one is actually provided, deep in the interpreter, specifically for pumping user interfaces. For example:

```
: key-xterm \ sid -- key
  { | temp[ cell ] -- }
  begin
    dup key?-xterm 0=
   while
    emptyidle 2 ms
  repeat
  temp[ 1 rot read-xterm drop
  temp[ c@
;
```

Where EMPTYIDLE is a deferred word, defaulting to NOOP.

Now we can just hook up a message pump.

```
: GTKEMPTYIDLE ( --- )
\ While messages and GTK events are available, process them - used in key etc.
  BEGIN NULL FALSE g_main_context_iteration 0= UNTIL
;

: INSTALLGTKHOOK ( --- )
\ Install the GTK4 version of the message pump in the Forth interpreter
  ASSIGN GTKEMPTYIDLE TO-DO EMPTYIDLE
;
```

For the executable version, we can simply recreate a much simplified version of gtk_main.

```
: GTKMAIN ( --- ) \ Run the GTK main loop outside the Forth interpreter
  BEGIN
    gtk_window_get_toplevels g_list_model_get_n_items 0>
  WHILE                                         \ There are any top level windows
    NULL TRUE g_main_context_iteration DROP     \ Process messages and events
  REPEAT
;
```

Now select which one to use, in our MAIN word.

```
: MAIN ( --- ) \ Main function
...
  DEBUGGING IF              \ In debug mode
    INSTALLGTKHOOK          \ Install the message pump in the Forth interpreter
  ELSE
    GTKMAIN                 \ Run the GTK main loop outside the Forth interpreter
  THEN
;
```

4.3 The widget naming problem

Fortunately, there is now a function gtk_buildable_get_id, which now does what it says it does. Therefore, the automatic creation of Forth VALUEs is almost unchanged from GTK3. This was described in a previous paper but it's worth repeating because it is a very useful technique.

```
: MAKEUINAMES { | pslist pobject pname -- } \ Create values for every builder object
  PBUILDER gtk_builder_get_objects -> pslist          \ Make list of objects
  pslist g_slist_length 0 ?DO                          \ For all objects
    pslist I g_slist_nth_data -> pobject               \ Get data
    pobject gtk_buildable_get_buildable_id -> pname    \ Get name
    pname Z" ___" 3 S= 0= IF                           \ Filter out unnamed objects
      pname pobject ZVALUE                             \ Create value for each name
      pobject pname gtk_widget_set_name                \ Set widget name to be same
    THEN
  LOOP
  pslist g_slist_free                                  \ Free list
;
```

One other change from GTK3 is that all widgets now have IDs - if no ID is defined in Cambalache, the builder creates one automatically, and returns this in the object list. The automatically generated names all start with three underscores, and since we have no use for these in Forth, we can discard them.

5. Other changes required

We have described above the issues that are particular to Forth. Upgrading an application from GTK3 to GTK4 is still a major undertaking. The documentation lists 20 things that can be done in GTK3 to prepare for the change, and 93 things that must be done at the point of change. Of course, for any particular application not all of these will be relevant.

In the VFX GTK4 bindings file, we have commented out using
\ ** extern: ...
all the functions that have been deleted from GTK4 and where we will have to rewrite code to use new functions. There are 53 of these functions in our code. Some of them are in our code only once or twice, but some are called on a very large number of occasions.

We have allocated three man-months to the conversion.

6. Special Forth techniques that still work in GTK4

6.1 Internationalisation

In past papers I have described the techniques we use for internationalisation, with on-the-fly translation of the user interface. The base phrase (for example, of a label) is defined in Cambalache.

When the application starts, we can go through the following process:

a) Get the object list provided by GtkBuilder
b) Identify the type of each object (e.g. GtkLabel) as potentially requiring translation
c) Ask the object in its type appropriate way for its base phrase
d) Check that it is marked as a phrase that needs to be translated
e) Check whether that phrase is already in our database
f) If not, inserrt it into the database
g) Add the phrase reference number to the object
h) Set the phrase in the currently selected language into the object in the type appropriate way.

In a similar way, when the user requests a change of language, we just read back the phrase reference number from the object, and set the phrase for the new language.

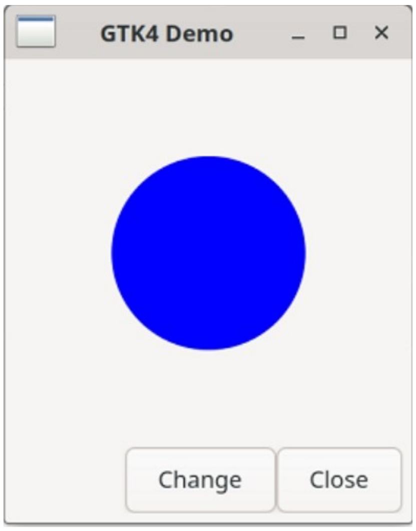
The good news is, all of these techniques appear to still work correctly in GTK4, even though one of the necessary functions is not directly documented - it is in the header file as a macro.

6.2 Drawing using Cairo

The appears to have been no change in the ability to use Cairo Graphics within GTK4, even though the widgets themselves appear now to be rendered using OpenGL, with Vulkan in progress.

7. The demonstration program

As with GTK3, we prepared a very small demonstration program to illustrate the basic principles.



The build and application files:

```
\ 64 bit floating point is required for Cairo

' REC-NDPFLOAT FORTH-RECOGNIZER \ Remove default floating point recogniser
-STACK
REMOVE-FP-PACK                  \ Remove default floating point package
include FPSSE64S.fth            \ Install SSE64 floating point
$26 -> ignSSEmask               \ Mask out floating point division by zero flag

TRUE CONSTANT DEBUGGING         \ Change when compiling an executable

include gtk4.fth                \ GTK4 tools
include DemoUI.fth              \ List of UI files for this application
include Cairo.fth               \ Cairo graphics file
gtk_init                        \ Initialise GTK
LOADUIS                         \ Load as part of compilation process
MAKEUINAMES                     \ Make values for each object
include gtk4demo.fth            \ Main program

.BadExterns cr                  \ Report any library failures
checkdict                       \ Report any dictionary corruption

GTK4DEMO                        \ Start demo program
```

```
1 1 CallProc: DEMOTIMER { pval  -- f } \ 2s timer event handling
  CURRCOL CASE                          \ Next colour
    0 OF 1 ENDOF
    1 OF 0 ENDOF
    2 OF 3 ENDOF
         2 SWAP
  ENDCASE -> CURRCOL
  demodrawingarea gtk_widget_queue_draw   \ Redraw drawing window
  TRUE                                    \ Return processed flag
;

5 0 CallProc: DEMODRAW { pwidget pcr pwidth pheight puser -- } \ Draw function
  pwidget -> CWIDGET          \ Set current widget
  pcr -> CCR                  \ Set current cairo context
  CURRCOL CASE                \ Select fill colour
    0 OF RED    ENDOF
    1 OF BLUE   ENDOF
    2 OF GREEN  ENDOF
         ORANGE SWAP
  ENDCASE BRUSH
  50 50 150 150 ELLIPSE       \ Draw a circle in the middle
;

DEMOCLOSEBUTTON CLICKED SIG: { pbutton puser -- } \ Close button clicked
  demowindow gtk_window_destroy
;

DEMOCHANGEBUTTON CLICKED SIG: { pbutton puser -- } \ Change button clicked
  CURRCOL 0 1 WITHIN IF 2 ELSE 0 THEN -> CURRCOL \ Switch colour pair
;

: GTK4DEMO-WINDOW ( --- ) \ Initialise and show demo window
  demodrawingarea DEMODRAW NULL NULL gtk_drawing_area_set_draw_func \ Set function
  demowindow gtk_window_present                                     \ Show window
  2000 DEMOTIMER 0 g_timeout_add DROP                               \ Start timer
;

: GTK4DEMO ( --- ) \ Main function
  #BADLIBS IF                     \ Any libraries not loaded
    Z" Libraries not loaded" FATAL
  THEN
  PBUILDER 0= IF                  \ UI not loaded ( in executable mode )
    gtk_init                      \ Initialise GTK
    LOADUIS                       \ Load UI files
    SETUIVALS                     \ Set values for UI objects
  THEN
  CONNECTUISIGS                   \ Connect UI signals
  gtk_init                        \ Initialise GTK
  GTK4DEMO-WINDOW                 \ Initialise and show demo window
  DEBUGGING IF                    \ In debug mode
    INSTALLGTKHOOK                \ Install the pump in the Forth interpreter
  ELSE
    GTKMAIN                       \ Run the GTK main loop outside interpreter
  THEN
;
```

8. Conclusion

In Forth, there is always a way to overcome the problems that library developers inadvertently create.

Creating a new Forth application in GTK4 is very straightforward, however, updating an existing application from GTK3 to GTK4 is a major undertaking, whichever language you use.