Writing a native component for Gambas

Tobias Boege ⟨tobias@gambas-buch.de⟩

Abstract. In this article, we will develop a small component in C for the Gambas interpreter. It brings you all the way from the source code organisation imposed by the build system over writing code in the Gambas interpreter environment, and actually building it, to using your component in a Gambas project.

The example component developed, `gb.termios`, will mirror some of the often-used terminal control functions from the C library into Gambas. The reader is expected to know C and to have some understanding of terminals (and be able to read the rest off the manpage `termios(3)`). A good knowledge of Gambas is necessary to begin with native component development.

1 The objective

Say you have a shared library (.so) on your system that you want to use in your Gambas projects. An easy solution *might* be to use the `Extern` keyword. This solution fails very quickly, though, if your library is big or complicated, uses own structures, etc. and you won't get nice object- and event-oriented code out of it. `Extern` is simply not made to translate whole complex libraries into Gambas.

In this scenario, one should consider the preferred alternative: native components. These are shared libraries which act as mediator between Gambas and your shared library. They take the library's structures and wrap them into classes as the Gambas interpreter knows them. The component `gb.qt4`, for example, interfaces Gambas with the C++ library QT4 and makes some of its objects available to Gambas programmers (like `Button`, `Window`, etc.).

In this article, we will write a small example component which brings low-level terminal control functionality from the C library to Gambas. It will be called `gb.termios`. As a motivation, imagine the situation that you are writing a console application. Often, these two needs occur:

1. to be able to switch the terminal echoing off (e.g. to let the user enter a password without it being shown on the screen),
2. to be able to receive keystrokes as soon as they are made, not whole lines after the Return key was pressed.

There are functions for these tasks and more already in the (GNU) C library.[1]

[1] Note that there is also the `stty` utility in the GNU `coreutils` package but we will ignore this option and write a component.

2 Preparation

Native components need to be built against the Gambas source tree, so we need to download that first, as is explained in the wiki. Indeed we will be using the latest Gambas revision from the SVN repository. When we want to test our component, we need to install the Gambas version we developed it against. So, this article is a superset of "How to install the latest Gambas?". Also be sure to uninstall your local Gambas before we begin our journey. It happened previously[1] that installing a self-compiled Gambas *over* an existing installation (i.e. without uninstalling the old one first) would damage the package's integrity (random crashes being among the consequences). In some cases (depending on when you read this article), the latest revision may not compile, because it is a development snapshot. File a bug report in this case.

```
$ svn co svn://svn.code.sf.net/p/gambas/code/gambas/trunk gambas3-dev
[...]
A gambas3-dev/gb.net/missing
A gambas3-dev/gb.net/reconf
A gambas3-dev/gb.net/NEWS
Checked out revision 6903.
$ cd gambas3-dev
```

We now proceed to create the build infrastructure for our component. There is a script which can do that for you if you fill in some information into the right files. Change into the `TEMPLATE/conf` directory and create a new file `gb.termios.conf`. Looking at the other configuration files in there, we fill it with the appropriate values[2]:

```
$ cd TEMPLATE
[TEMPLATE] $ cp conf/gb.{cairo,termios}.conf
[TEMPLATE] $ vim conf/gb.termios.conf
[TEMPLATE] $ cat conf/gb.termios.conf
/* Copyrights */
#define __COPYRIGHT (C) 2015
#define __AUTHOR Tobias Boege
#define __EMAIL <tobias@gambas-buch.de>

/* Name of the component */
#define __COMPONENT gb.termios
/* Name of the component with points replaced by underscore */
#define __COMPONENT_UNDERSCORE gb_termios
/* Short name of the component */
#define __NAME termios
/* Short name of the component in uppercase */
#define __UNAME TERMIOS
/* Description of the component */
#define __DESCRIPTION Termios from the C library
[...]
       /* Includes to search for */
       #define __SEARCH_INCLUDE termios.h
       /* Includes directories search path */
       #define __SEARCH_INCLUDE_PATH /usr/local/lib /usr/local /usr/lib /usr
[...]
/* Source file list */
#define __SOURCES main.c main.h
/* Main C/C++ source basename in uppercase */
#define __MAIN_UNAME MAIN
```

[2]The complete file will not be shown but is hopefully available for download wherever you find this article.

Let the script generate our component source directory:

```
[TEMPLATE] $ ./make-component gb.termios
Creating component directory gb.termios...
Applying template...
Creating source files...
Creating symbolic links...
mv: cannot stat 'gb.xxx.component': No such file or directory
[TEMPLATE] $ cd ..
[gambas3-dev] $ ls -ld gb.termios
drwxr-xr-x 3 tab users 360 Feb 21 20:33 gb.termios
```

And indeed, there is the directory. As a next step, we need to tell the build system that there is a new component. The relevant files are `configure.ac` and `Makefile.am` in the source tree root. We add one line to each file, in the fashion suggested by the other lines, to introduce our component:

```
[gambas3-dev] $ vim configure.ac
[gambas3-dev] $ cat configure.ac
[...]
GB_CONFIG_SUBDIRS(openssl, gb.openssl)
GB_CONFIG_SUBDIRS(openal, gb.openal)
GB_CONFIG_SUBDIRS(termios, gb.termios)

AC_CONFIG_SUBDIRS(comp)
AC_CONFIG_SUBDIRS(app)
[...]
[gambas3-dev] $ vim Makefile.am
[gambas3-dev] $ cat Makefile.am
[...]
 @openssl_dir@ \
 @openal_dir@ \
 @termios_dir@ \
 comp \
 app \
[...]
```

Now it's time to configure the build

```
[gambas3-dev] $ ./reconf-all && ./configure
[...]
||
|| THESE COMPONENTS ARE DISABLED:
|| - gb.db.sqlite2
|| - gb.openal
|| - gb.sdl2
|| - gb.sdl2.audio
||
```

As long as `gb.termios` is not among the latter ones, all is good. We can verify that `gb.termios` will actually be built by examining its directory:

```
[gambas3-dev] $ ls gb.termios/Makefile
gb.termios/Makefile
[gambas3-dev] $ ls gb.termios/DISABLED
ls: cannot access gb.termios/DISABLED: No such file or directory
```

If it contains a `Makefile` and *no* `DISABLED` file, it will be compiled.

Now we want to write something to be compiled – but not quite yet. It is customary to create a new file for each class (although associated classes may be in one file). The naming convention for the source file of a class `MyClass` is `c_myclass.c`. Note that each class .c file should be accompanied by an .h file. We will see the reason later. Our main class will be called `Termios`:

```
$ cd gb.termios/src
[gb.termios/src] $ touch c_termios.c c_termios.h
```

The first we are going to do is to register these files, in turn, to the build system:

```
[gb.termios/src] $ vim Makefile.am
[gb.termios/src] $ cat Makefile.am
[...]
gb_termios_la_CPPFLAGS = @TERMIOS_INC@

gb_termios_la_SOURCES = \
main.c main.h \
c_termios.c c_termios.h
```

We also want to create a component description file as apparently the `make-component` script above failed to do so:

```
[gb.termios/src] $ vim gb.termios.component
[gb.termios/src] $ cat gb.termios.component
[Component]
Key=gb.termios
Author=Tobias Boege
State=Experimental
```

It is always a good idea (and most often your only hope) to look at how the existing components do things.[3]

`c_termios.{c,h}` are the only source file we are going to write for the component in this article. After we reconfigure the `gb.termios` component, the build system will be ready. Whenever you add new files to your component, this procedure will become necessary again. Note, that we are calling the configuration scripts from inside the `gb.termios` component. Each component can be configured separately once it is registered as "to be built" with the top-level `Makefile`, so you don't have to reconfigure the whole Gambas (as you may have noticed, that takes a while) when you add files to an isolated component only.

```
[gb.termios/src] $ cd ..
[gb.termios] $ ./reconf && ./configure
[...]
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating config.h
config.status: config.h is unchanged
config.status: executing depfiles commands
config.status: executing libtool commands
[gb.termios] $ cd src
```

While I write the code which we are going to analyze in the next section, you should read the manpage `termios(3)`. This is the interface we want to transfer (very partially!) into Gambas. The class will contain just enough functionality to enable us to input passwords sanely into a terminal application. We meet in the next section when you are done.

[3] As for the `State` values, have a look at the IDE source code, `app/src/gambas3/.src/Component/CComponent.class:InitComponent()`. As a component developer, doing self-study of the Gambas code base is most vital.

3 Writing a class

3.1 The periphery

The code is listed (almost) completely but also to be found as a tarball for download. We start our tour at `main.c` which glues the parts of our component together and implements component-global hooks.

main.c

```
1  /***************************************************************************
2
3    main.c
4
5    gb.termios component
6
7    (C) 2015 Tobias Boege <tobias@gambas-buch.de>
8
9    This program is free software; you can redistribute it and/or modify
10   it under the terms of the GNU General Public License as published by
11   the Free Software Foundation; either version 2, or (at your option)
12   any later version.
13
14   This program is distributed in the hope that it will be useful,
15   but WITHOUT ANY WARRANTY; without even the implied warranty of
16   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
17   GNU General Public License for more details.
18
19   You should have received a copy of the GNU General Public License
20   along with this program; if not, write to the Free Software
21   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22   MA 02110-1301, USA.
23
24 ***************************************************************************/
```

This is the preamble. The GPL wants us to put it at the beginning of every source file. You also want to put a joke in there somewhere at the beginning to make the people a little happy before they have to wade through your code.

main.c (cont.)

```
26 #define __MAIN_C
27
28 #include <stdio.h>
29 #include <unistd.h>
30 #include <termios.h>
31
32 #include "main.h"
33 #include "c_termios.h"
34
35 GB_INTERFACE GB EXPORT;
36
37 int tty_fd;
38 int restore = 1;
39 static struct termios orig_tty;
```

In each `<file>.c` you want to define a macro `__<FILE>_C` to tell that we are currently in that file. We will see in the corresponding .h file how this is useful. The `GB_INTERFACE GB EXPORT` line makes the external Gambas API structure `GB` available. The `EXPORT` macro is defined in the header `../gambas.h` (included via `main.h`) so that the declared symbol is externally visible while linking. We also define some global variables.

main.c (cont.)

```
41  GB_DESC *GB_CLASSES[] EXPORT = {
42          CTermios,
43          CTermiosLocal,
44          CTermiosSpecial,
45          NULL
46  };
```

The `GB_CLASSES` array contains all classes provided by the component. When the interpreter loads a component, it looks for this symbol to enumerate all classes and load them into its symbol table. The elements are pointers to `GB_DESC` arrays. The above-mentioned classes are defined in `c_termios.c`. The list has to be terminated with a `NULL` pointer.

main.c (cont.)

```
48  int EXPORT GB_INIT()
49  {
50          /* Find a tty among stdin, stdout and stderr */
51          if (isatty(fileno(stdin))) {
52                  tty_fd = fileno(stdin);
53          } else if (isatty(fileno(stdout))) {
54                  tty_fd = fileno(stdout);
55          } else if (isatty(fileno(stderr))) {
56                  tty_fd = fileno(stderr);
57          } else {
58                  GB.Error("No terminal found");
59                  return 1;
60          }
61
62          tcgetattr(tty_fd, &orig_tty);
63          return 0;
64  }
65
66  void EXPORT GB_EXIT()
67  {
68          /* Restore the original terminal settings, cf. c_termios.c */
69          if (restore)
70                  tcsetattr(tty_fd, TCSANOW, &orig_tty);
71  }
```

Besides `GB_CLASSES` there are other symbols recognised by the interpreter: the functions `GB_INIT()` and `GB_EXIT()`. They are called when the component is loaded and unloaded respectively. In `gb.termios`, we use the init function to see if we can find a TTY among the standard file descriptors. If we don't, the component will raise an error and shut the program down. Else, the TTY fd will be saved for later and we obtain the initial TTY attributes.

The exit function may restore the initial TTY configuration, depending on the value of `restore`. As you can see, this is `true` by default but can be changed, as we will see later, by the Gambas programmer through the interface of the `Termios` class.

The .h files are not particularly interesting usually. They are mostly used to declare `extern` variables, as we see in `main.h`:

main.h

```
/***************************************************************************

  main.h

  (C) 2015 Tobias Boege <tobias@gambas-buch.de>

  This program is free software;
  [...]

***************************************************************************/

#ifndef __MAIN_H
#define __MAIN_H

#include "gambas.h"

#ifndef __MAIN_C
extern GB_INTERFACE GB;
/* The tty fd or termios calls */
extern int tty_fd;
/* Whether to restore the original terminal on component exit */
extern int restore;
#endif

#endif /* __MAIN_H */
```

Here, the `__MAIN_C` macro is used to conditionally provide `extern` declarations of those variables declared in `main.c` which we will need in other compilation units, too. Now, let's get to the actual classes in `c_termios.c`.

3.2 Interface description

c_termios.c

```
/***************************************************************************

  c_termios.c - Termios class and friends

  (C) 2015 Tobias Boege <tobias@gambas-buch.de>

  This program is free software;
  [...]

***************************************************************************/

#define __C_TERMIOS_C

#include <termios.h>

#include "../gambas.h"
#include "main.h"
#include "c_termios.h"

typedef struct {
        GB_BASE ob;
        struct termios current;
        struct termios begin; /* Settings at beginning of transaction */
        int in_transaction;
} CTERMIOS;
```

Here, we see the declaration of our main structure, `CTERMIOS`. It will be the backing data for `Termios` Gambas objects. The naming scheme employed is customary in native Gambas code and you should adhere to it. The first member of each structure which is going to represent Gambas objects has to be a `GB_BASE`. The interpreter uses this part of the structure to store a pointer to the class description and a reference counter. The rest is data members for the `Termios` functionality.

We are going to skip the class implementation which follows these lines and take a look at the class *interface* first, which is an array of `GB_DESC` structures – also referred to as the class *description*.

c_termios.c (`Termios` interface)

```
154 GB_DESC CTermios[] = {
155         GB_DECLARE("Termios", sizeof(CTERMIOS)),
156         GB_AUTO_CREATABLE(), /* Default object inherits current terminal settings */
```

`GB_DECLARE` begins a class description. It takes two arguments: a string which contains the name of the class to declare, and its structure size.

`GB_AUTO_CREATABLE()` is the native analogue of `Create Static` in Gambas. It makes the class auto-creatable, i.e. you can use the class name like an object. The interpreter will create a backing object (the so-called *automatic instance*) behind the scenes.

We see another naming convention here. The variable referencing the class description array has contained the same letters in its name as the structure but not all upper-case this time.

c_termios.c (`Termios` interface cont.)

```
158         /* Constants for Commit() */
159         GB_CONSTANT("Now", "i", TCSANOW),
160         GB_CONSTANT("Drain", "i", TCSADRAIN),
161         GB_CONSTANT("Flush", "i", TCSAFLUSH),
162 
163         GB_STATIC_PROPERTY("Restore", "b", Termios_Restore),
164         GB_STATIC_PROPERTY_READ("Handle", "i", Termios_Handle),
165 
166         GB_METHOD("_new", NULL, Termios_new, "[(Parent)Termios]"),
167 
168         GB_METHOD("Begin", NULL, Termios_Begin, NULL),
169         GB_METHOD("Commit", NULL, Termios_Commit, "[(Action)i]"),
170         GB_METHOD("Abort", NULL, Termios_Abort, NULL),
```

Now, `CTermios` is still an array of `GB_DESC` structures. So why are there so many different macros in its contents? — `GB_DESC` is a very generic structure. It is capable of describing every type of class symbol in Gambas. The macros used above create `GB_DESC`s corresponding to their type, out of their parameters. The names make it obvious what each macro generates. Only the parameters are not so clear at a first glance. All of these get the name of the symbol as their first argument, though.

`GB_CONSTANT` then wants the data type of the constant symbol to declare. In the lines above, this is always `"i"` which denotes `Integer`. It follows the actual value of the constant. In this case it's some `TCSA*` macros from the `termios.h` header.

The `GB_PROPERTY` or `GB_STATIC_PROPERTY` macros are similar but don't take a constant value as their last parameter but instead the name of a property implementation which is a C function. Note that your C compiler needs functions

to be declared before their names are used. This is the reason why you will find the implementation functions always before the class description in the source files[4]. The implementation functions show yet another naming convention: `<Class>_<Symbol>`. Keep their names in mind until we get to look at how symbol implementations are defined.

`GB_METHOD` and `GB_STATIC_METHOD` want four parameters. The second is now the return value data type or `NULL` if the method has no return value. The third is the method implementation function and the fourth is a string describing the method parameters. As you may be able to tell, the format is `(<ParameterName1>)<Type1>(<ParameterName2>)<Type2>...`. For the native data types, we have to use abbreviations, like `i` for `Integer`, `b` for `Boolean` or `s` for `String`. Multiple parameters are denoted by juxtaposition:

```
GB_METHOD("Test", NULL, MyClass_Test, "(MyInt)i(SomeString)s"),
```

If you want to take arguments of non-native types (classes) you have to write down the class name. If there are still parameters after a non-native parameter, the class name has to be followed by a semicolon:

```
GB_METHOD("Test", NULL, MyClass_Test, "(MyInt)i(ObjA)ClassA;(ObjB)ClassB"),
```

Another syntax strikes the eye: square brackets. They enclose optional arguments. Look, for instance, at the `_new()` method. In Gambas, we would declare it as

```
Public Sub _new(Optional Parent As Termios)
```

You should take a look at the definitions of those `GB_*` macros (and of the other possible `GB_DESC`-defining macros!) in the `gambas.h` header.

c_termios.c (`Termios` interface cont.)

```
172         /* GB_PROPERTY_SELF("Input", ".Termios.Input"), */
173         /* GB_PROPERTY_SELF("Output", ".Termios.Output"), */
174         /* GB_PROPERTY_SELF("Control", ".Termios.Control"), */
175         GB_PROPERTY_SELF("Local", ".Termios.Local"),
176         GB_PROPERTY_SELF("Special", ".Termios.Special"),
177
178         GB_END_DECLARE
179 };
```

One type of symbol is particularly interesting for native programmers for two reasons: (1) it is not available for Gambas programmers and (2) it can be used to design an elegant interface with very efficient code. It's the `GB_PROPERTY_SELF`s a.k.a. *virtual classes*. As you can see, `GB_PROPERTY_SELF` takes only two arguments: the name of the symbol to declare and then another name. The latter is a class name of a virtual class. Virtual class names are customarily prefixed by a dot and are named after the class they are defined in followed by a dot and the symbol name they are the type of.[5]

Why this weird naming? — The answer will be clear after (a while after) we learned what virtual classes are and what they are used for. A virtual class is a class which has only interface. Doesn't sound useful and, indeed, alone they're not worth much. But the Gambas interpreter can use them together with ordinary classes. Virtual classes can be used to paper a new interface over

[4] You can also use forward declarations but that's just ugly.

[5] You may want to re-read this when you are through this whole section.

one of your objects and this can be used in a way that makes a class look much more object-oriented. We have declared above a symbol called `Local` in the `Termios` class to be a `GB_PROPERTY_SELF`. When you write

```
myTermios.Local
```

in Gambas, the interpreter looks up the symbol and sees that it is `SELF` and links to the class `.Termios.Local`. What the interpreter now does is to keep the object `myTermios` as is (that is the `SELF` part) and treats it as if it was an object of class `.Termios.Local`.

To get the gist of it, consider a famous representative of virtual class usage: the `Editor` from `gb.qt4.ext`. As you will readily look up, it has a property called `Flags` which is of type `.Editor.Flags`. This very much looks like a virtual property. If you have ever used the `Editor`, you know that settings flags goes like this

```
myEditor.Flags[Editor.NoFolding] = True
```

Access to `myEditor.Flags` triggers the virtual property treatment of the interpreter: the current object is at that point `myEditor` but with its class description replaced by that of the virtual class `.Editor.Flags`. Then the array accessors are used on that object. What now happens is magic: the interpreter picks the array accessor implementation of the `.Editor.Flags` class and makes it act on the `myEditor` object. The `.Editor.Flags` code can be used to modify internal data of the `Editor`. I want to stress again that this is accomplished by switching a class description in an object.

Even though it looks as if `myEditor.Flags` is an independent object with its own implementation of array accessors (and potentially more methods and properties), it is internally just another view of the `myEditor` object. One result is purely cosmetic: it looks more object-oriented than the alternatives, like

```
myEditor.SetFlag(Editor.NoFolding, True)
```

and comes at virtually no cost. Note that no new object needs to be created to provide that new interface. The interpreter just has to replace a pointer. Virtual classes are intimately bound to the class and symbol they are used with. This explains their naming scheme.

We will be using virtual objects in much the same way. You see that the `struct termios` has bit masks and an array member. Instead of having the Gambas programmer use predefined constants and bit operations, we use virtual classes to make it object-oriented. The different flags will come as boolean-valued properties of the virtual class. `Termios.Local` mirrors the `c_lflag` member and `Termios.Special` the `c_cc` member of `struct termios`.

Do also not forget the `GB_END_DECLARE` at the end of the class description. There was a case where its absence has not caused a crash on GNU/Linux systems but did on Cygwin [2], and being a missing “end of array” marker it should actually cause crashes.

3.3 Implementation

We can no longer procrastinate to look at the implementation:

c_termios.c (`Termios` implementation)

```
39  /**G
40   * Whether to restore the terminal settings, when the program exits, to what
41   * they were when it started.
42   *
43   * This is True by default.
44   */
45  BEGIN_PROPERTY(Termios_Restore)
46
47          if (READ_PROPERTY) {
48                  GB.ReturnBoolean(restore);
49                  return;
50          }
51          restore = VPROP(GB_BOOLEAN);
52
53  END_PROPERTY
54
55  /**G
56   * Return the file descriptor which is taken as the terminal.
57   **/
58  BEGIN_PROPERTY(Termios_Handle)
59
60          GB.ReturnInteger(tty_fd);
61
62  END_PROPERTY
```

This is the implementation of the `Restore` property. It already contains lots of macros that need explanation. As you might have guessed by now, the actual challenge in learning to program for the Gambas interpreter is to memorise its macro language, along with memory and object management and, of course, its API.

Symbol implementation functions are not declared like usual functions in C because the Gambas interpreter has special ways to call them from a Gambas program (it needs to manage the stack of the Gambas program, etc.). Therefore we have macros to begin the definition of implementation functions. To define a property implementation, use `BEGIN_PROPERTY` and give it the desired name of the function. This name is used in the class description, as we have seen, to link a symbol to its implementation function. The property implementation is ended by `END_PROPERTY` which, at the time of this writing, just translates to the closing `}` after a function body in C. Nevertheless it's nice and future-proof to use the macros.

Properties in Gambas can be read and written. If you write a class in Gambas, you have for each read-write property two methods: one which implements the read and another one for the write. Native components have a single function but can use a macro `READ_PROPERTY` to determine if the property is read is written to. The above structure of a read-write property implementation is idiomatic. `Termios.Handle` on the other hand was declared as a `GB_PROPERTY_READ` in the class description, i.e. a read-only property. The interpreter will forbid any writes to this property so we can save the `if` block and only implement the read.

Reading the `Restore` property causes a call to `GB.ReturnInteger()`, an interpreter API to return an integer from an implementation function. Note

the difference to programming in C: symbol implementation functions for the Gambas interpreter always return `void` in C. They use special interpreter APIs to transport the return value back to the Gambas program. The interpreter API does also not imply a `return` from the implementation function. If you want to leave the function body, a C `return` must follow.

`Restore` can also be written to. The last non-empty line in `Termios_Restore` takes care of that. The macro `VPROP` gives the value assigned to the property – in a property assignment context. Its parameter specifies the data type of the property and it should match the one in the class description.

You also want to document your class interface by the way. This is done by writing a comment of the special format

```
1  /**G
2   * Documentation goes here.
3   **/
```

just before the symbol implementation starts. During the development cycle before Gambas 3.7, programs were added to the repository which can produce help files importable into the online wiki out of such source code comments, a fine way to keep the documentation in sync with the code.[6]

c_termios.c (`Termios` implementation cont.)

```
64  #define THIS ((CTERMIOS *) _object)
```

You will often find a line similar to this in other components. What the `BEGIN_*` macros, which begin a symbol implementation function, hide from you is that all implementation functions get a `void *_object` as an argument. This is simply the Gambas object the property or method is applied to. In the case of `Termios` symbol implementations, this is always a `Termios` object which is, in turn, our `CTERMIOS` structure.

With this in mind, the majority of the code should not be difficult to read:

c_termios.c (`Termios` implementation cont.)

```
66  /**G
67   * Create a new Termios object. If ~Old~ is given, its settings are copied,
68   * if not the current terminal settings are copied.
69   *
70   * Normally, you won't need multiple Termios objects. Only one configuration
71   * can be active at a time. Use multiple objects if you have recurring needs
72   * in your program to apply a specific configuration (like regular password
73   * confirmation).
74   **/
75  BEGIN_METHOD(Termios_new, GB_OBJECT old)
76
77          if (MISSING(old)) {
78                  tcgetattr(tty_fd, &THIS->current);
79          } else {
80                  CTERMIOS *old = (CTERMIOS *) VARG(old);
81
82                  THIS->current = old->current;
83          }
84          THIS->in_transaction = 0;
85
86  END_METHOD
```

[6] ***author pats hisself on the back****

Here we have an example of a method which takes a parameter. As with C functions, this is written down in the function signature, with a comma after the function name. However, if a function receives multiple parameters, everything after the first parameter needs to be separated by semicolons as in this fictive method:

```
BEGIN_METHOD(MyClass_Test, GB_INTEGER myint; GB_STRING somestring)
```

If you are curious, look into the definition of `BEGIN_METHOD` to see a nifty hack:

gambas.h (excerpt)

```
431 #define BEGIN_METHOD(_name, par) \
432 typedef \
433         struct { \
434                 par; \
435                 } \
436                 _##_name; \
437 \
438 void _name(void *_object, void *_param) \
439 { \
440 _##_name *_p = (_##_name *)_param;
```

That definition reveals to us an important point about method parameters. They are always wrapped inside a structure, and if you have a look at the types we use to declare parameters, like `GB_INTEGER`, they are slightly more complicated to access than we would want them to be. The reason is that these parameters are not meant to be accessed directly. We have again macros for this purpose.

The first one used in the function above is `MISSING()`. It can be applied to optional arguments and tells if they were given in that call. Recall that `Termios` is `GB_AUTO_CREATABLE()`. You must know that the automatic instance of a class is created by the interpreter on demand and without providing any arguments. The `_new()` method of an auto-creatable class must therefore have only optional arguments – or else an error is raised whenever the interpreter decides to create the automatic instance. In case the parameter `Old As Termios` was not specified – a `Termios` object to inherit the TTY configuration from –, we get the current TTY's configuration.

If, on the other hand, the `old` parameter is given, we use another argument accessor macro, `VARG()`, which stands for "value of argument". It returns the actual value behind the given argument as one would use it in a C program, that is, given the name of a `GB_INTEGER` parameter name, it returns the numerical value of that integer, or given a `GB_OBJECT` parameter name, it returns a pointer to that object – whichis what we use above.

Be aware that implementation function parameters do *not* belong to the inner scope of the C function body. They are wrapped inside a hidden structure. So you can still create local variables of the same name as the parameters. Argument accessor macros (`ARG()`, `VARG()`, `MISSING()` and `VARGOPT()` to name them all), on the other side, operate on that hidden structure and interpret their given arguments not as variables but as names of members in that structure.

c_termios.c (`Termios` implementation cont.)

```
88 /**G
89  * Begin a transaction. All changes to the terminal settings are buffered
90  * and can be committed using [Commit()](../commit) or forgotten by using
```

```
 * [Abort()](../abort).
 *
 * Transactions can **not** be imbricated.
 **/
BEGIN_METHOD_VOID(Termios_Begin)

	/* Back current settings up */
	THIS->begin = THIS->current;
	THIS->in_transaction = 1;

END_METHOD

static int apply(CTERMIOS *term, int action)
{
	int ret = tcsetattr(tty_fd, action, &term->current);

	/* Not all changes need to succeed. We must get the settings now
	   to be sure to have the effective configuration. */
	tcgetattr(tty_fd, &term->current);
	return ret;
}

/**G
 * Commit all the changes made since [Begin()](../begin). The ~Action~
 * argument specifies when the changes shall be performed:
 *
 * - [Now](../now): immediately,
 * - [Drain](../drain): when all data written to the terminal was actually
 * transmitted,
 * - [Flush](../flush): when all data written to the terminal was actually
 * transmitted but all data which was received but not read yet is
 * discarded.
 *
 * By default, the changes are applied [Now](../now).
 **/
BEGIN_METHOD(Termios_Commit, GB_INTEGER action)

	int action = VARGOPT(action, TCSANOW);

	if (!THIS->in_transaction) {
		GB.Error("Not in a transaction");
		return;
	}
	apply(THIS, action); /* TODO: Error management */
	THIS->in_transaction = 0;

END_METHOD

/**G
 * Aborts the current transaction by forgetting all changes since the
 * [Begin()](../begin) call.
 **/
BEGIN_METHOD_VOID(Termios_Abort)

	if (!THIS->in_transaction) {
		GB.Error("Not in a transaction");
		return;
	}
	THIS->current = THIS->begin;
	THIS->in_transaction = 0;

END_METHOD
```

We see two new things here. The more prominent one is another interpreter API: `GB.Error()`. This function registers a runtime error. It does nothing else. Like the `GB.Return*()` functions, it does not imply a `return` from the C function, so you need to do that yourself. The error is not raised immediately but along the path when control is transferred back from the native component to the Gambas program. The Gambas programmer is free to `Try` and `Catch` your error.

Also in the `Commit()` function, another argument accessor macro is used: `VARGOPT()` which stands for "value of argument optional". It is a combination of `VARG()` and `MISSING()`, namely if the named optional argument was given, its value is returned. If it is missing, the second argument to `VARGOPT()` is returned as the default value.

c_termios.c (`.Termios.Local`)

```
181 #define IMPLEMENT_LFLAG(_name, _flag) \
182 /**G \
183  * Set the _flag setting. Refer to the 'termios(3)' manpage. \
184  **/ \
185 BEGIN_PROPERTY(TermiosLocal_##_name) \
186                                                                 \
187         if (READ_PROPERTY) { \
188                 GB.ReturnBoolean(THIS->current.c_lflag & _flag); \
189                 return; \
190         } \
191         if (VPROP(GB_BOOLEAN)) \
192                 THIS->current.c_lflag |= _flag; \
193         else \
194                 THIS->current.c_lflag &= ~_flag; \
195         if (!THIS->in_transaction) \
196                 apply(THIS, TCSANOW); \
197                                                                 \
198 END_PROPERTY
199
200 IMPLEMENT_LFLAG(Signal, ISIG)
201 IMPLEMENT_LFLAG(Canonical, ICANON)
202 IMPLEMENT_LFLAG(Echo, ECHO)
203
204 GB_DESC CTermiosLocal[] = {
205         GB_DECLARE_VIRTUAL(".Termios.Local"),
206
207         GB_PROPERTY("Signal", "b", TermiosLocal_Signal),
208         GB_PROPERTY("Canonical", "b", TermiosLocal_Canonical),
209         GB_PROPERTY("Echo", "b", TermiosLocal_Echo),
210
211         GB_END_DECLARE
212 };
```

You will find the pattern shown at the top of these lines at several other places in native Gambas code. It automates the creation of implementation functions where some of them are very similar. In the case above we are already in the implementation of the virtual `.Termios.Local` class where, according to the description of virtual classes given in the previous subsection, `Boolean` properties are used to provide a more object-oriented interface to the bitmasks in `struct termios`. Modulo the flag constant and the property name, all these implementations are the same, so their creation can be automated. Just below its definition, it is used thrice to implement the three properties of `.Termios.Local`. The code for `.Termios.Special` is very similar.

`c_termios.h` just provides `extern` declarations of the class structures which we already used in `main.c`:

c_termios.h

```
1  /***************************************************************************
2
3    c_termios.h
4
5    (C) 2015 Tobias Boege <tobias@gambas-buch.de>
6
7    This program is free software;
8    [...]
9
10 ***************************************************************************/
11
12 #ifndef __C_TERMIOS_H
13 #define __C_TERMIOS_H
14
15 #include "gambas.h"
16
17 extern GB_INTERFACE GB;
18
19 #ifndef __C_TERMIOS_C
20 extern GB_DESC CTermios[], CTermiosLocal[], CTermiosSpecial[];
21 #endif
22
23 #endif /* __C_TERMIOS_H */
```

4 Using the component

Let's install and use the component. You should always install the component together with the Gambas tree you built it against. Since we haven't installed Gambas in the first section, we do it now:

```
[gb.termios/src] $ cd ../..
[gambas3-dev] $ make && sudo make install
[...]
Making all in gb.termios
make[2]: Entering directory '/home/tab/gambas3-dev/gb.termios'
make all-recursive
make[3]: Entering directory '/home/tab/gambas3-dev/gb.termios'
Making all in src
make[4]: Entering directory '/home/tab/gambas3-dev/gb.termios/src'
  CC gb_termios_la-main.lo
  CC gb_termios_la-c_termios.lo
  CCLD gb.termios.la
make[4]: Leaving directory '/home/tab/gambas3-dev/gb.termios/src'
[...]
```

Now we are home again. Time to fire the IDE up. We create a new console project, go to the project properties and look out for `gb.termios`. If the installation procedure went well, it should be there and be selectable. Tick `gb.crypt`, too. Also, we go to the options tab and "Use a terminal emulator" because the IDE console is not capable of emulating a TTY as far as we need it now. You may want to type the first few lines of the following code in yourself, to see that the IDE auto-completion popups work just like with the other components :-)

```
' Gambas module file

Private Const KeyBackspace As String = "\x08"
Private Const KeyReturn As String = "\n"

Public Sub Main()
  Dim sChar As String
  Dim sPassword As String

  Termios.Begin()
    Termios.Local.Echo = False
    Termios.Local.Canonical = False
    Termios.Special.MinRead = 0
    Termios.Special.TimeRead = -1
  Termios.Commit(Termios.Now)

  Print "Password:";;
  Do
    Read sChar, 1
    If sChar = "q" Then
      Print
      Quit
    Endif

    If sChar = KeyReturn Then
      Print
      Break
    Endif

    If sChar = KeyBackspace Then
      sPassword = Left$(sPassword, -1)
    Else
      sPassword &= sChar
    Endif
  Loop

  If Crypt.Check(sPassword, "$5$wEQEKuXszoW [...] McSKsC") Then
    Print "Wrong password!"
    Quit 1
  Endif
  Print "Correct!"
End
```

This project shows how we can now let the user input a password into our terminal-based application. The echoing of input is turned off and we are in non-canonical mode on with non-blocking reads and (practically) without timeout. This means we get each character as soon as the user types it. In the input loop, we gather the password keystroke by keystroke (we even support a little bit line editing by interpreting the backspace key). Let's see if you can guess the password (the full password hash was not printed above but can be found in the project archive, of course) :-)

5 Further steps

If you want to dwell a little more with `gb.termios`, look again at `termios(3)`. You can try to complete the support. Or read `tty_ioctl(3)` and `console_ioctl(3)` which may provoke new (static) classes. This will force you to get accustomed to the build system, because new classes should be implemented in new files. An-

other idea, although not related to native component development, is to patch the IDE to include a description for `gb.termios`.

But if you really want to dive into it, you'll need a vision and an own project to work on and learn by. The most important thing is to read other contributors' components. I find `gb.inotify` to be a small and easy-to-understand component (also it implements events which is something we missed out completely in this article). If you want to learn how to juggle with data from Gambas programs, `gb.data` might be worth a look[7] (although I would advice against looking at the `List` and `Graph*` classes). If you are interested in `Streams`, `gb.net` is a good starting point, etc. etc. pp.

Also, if you have written a component that's worth it, consider presenting it to the mailing list(s) and try to get it into the official Gambas source tree.

References

[1] When the `gb.net.smtp` component was converted from a native to a Gambas one.

[2] Reported by N. Gerrard, `http://sourceforge.net/p/gambas/mailman/message/32263708/`.

[7] Guess what `gb.inotify` and `gb.data` have in common that I mention them here...