

---

# Introducing the Glib Mainloop

Ikke

Version 0.1

Copyright © 2005 Ikke

Published under the "Creative Commons Attribution-ShareAlike License Belgium"

## Table of Contents

Getting all tutorial samples under one project directory .....	1
The Glib Mainloop .....	2
Creating and starting the mainloop .....	3
Using timeouts .....	3
Conclusion .....	4

## Getting all tutorial samples under one project directory

After finishing my last article, I thought it'd be a good idea to keep everything we write together in one project dir. So let's do this first.

First we create a base directory for this project:

```
cd ~
mkdir -p code/gnome-tutorial
cd code/gnome-tutorial
mkdir -p src/gob-signals src/glib-mainloop
```

Now move the necessary files we wrote in the autotools part of the Makefiles tutorial to their new destination: `configure.ac`, the root `Makefile.am` and `autogen.sh` go into `~/code/gnome-tutorial`, whilst `src/Makefile.am`, `src/test-signal.gob` and `src/test-signal-test.c` go into `src/gob-signals`

This is what we got now:

```
./Makefile.am
./autogen.sh
./configure.in
./src
./src/gob-signals
./src/gob-signals/test-signal-test.c
./src/gob-signals/test-signal.gob
./src/gob-signals/Makefile.am
./src/glib-mainloop
```

Also touch the `NEWS`, `README`, `AUTHORS` and `ChangeLog` again in the root dir.

You should notice we're missing at least one file to be able to build everything again: we have no `Makefile.am` in `src/`, and the list of files to output in `configure.in` is wrong now too.

I leave this as an exercise to the reader:

1. Create `src/Makefile.am` so `gob-signals` will be treated as a subdir.
2. Edit `configure.in`, so the `Makefile.in` files in `src/` and `src/gob-signals/` will be processed
3. Update the "AC\_INIT" directive in `configure.in` so it points to a valid sourcefile

Now recreate all files (run `autogen.sh`), execute the configure script, run **make** and try to execute `src/gob-signals/test-signal` to see whether everything is ok. If all goes well, you can continue to the next part :-)

## The Glib Mainloop

Before we start writing code and getting a step closer to hackers heaven, we need to figure out what a "mainloop" is. Let's take a look at some dumb "Hello World" C program:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

What does happen here? We get into our `main()` function, call `printf()` with some string, and return.

Now think of a GUI application. If we'd write a GUI application like this:

```
#include <someframework.h>

int main()
{
    Window *w;

    w = new Window();
    window_display(w);

    return 0;
}
```

Most likely, the window will flash up, disappear immediately, and the program will exit, which is not what we want: it should stay visible, and our program shouldn't quit, but should listen to user events.

Let's return to CLI level, we're no GUI guru's (yet ;-)). If we want the same behavior as a GUI application should have in a CLI app, we could do it like this:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    while(1) {}
    return 0;
}
```

The while loop is only one possible implementation of course, I guess you can figure out some other yourself too :-)

Basically, that's what a mainloop does: it "hangs" your program.

Now you could be asking "Hey, I don't need to read all this just to know how to hang some program, I could have figured this out myself too". Please read on, because as you'll see in a minute, the Glib

mainloop offers much more functionality than just "hanging your program".

So, what \*does\* it offer? Remember the article on GObject signals? One of the things a Glib mainloop offers you is the ability to connect to a whole lot of signals: stuff you defined yourself, or things offered by Glib itself, or some other library.

## Creating and starting the mainloop

Creating a Glib mainloop [<http://developer.gnome.org/doc/API/2.0/glib/glib-The-Main-Event-Loop.html>] is very simple and straightforward. First go into the `src/glib-mainloop` directory, this is the place where we'll play around with some sample code.

```
#include <glib.h>

int main()
{
    GMainLoop *loop;

    loop = g_main_loop_new(NULL, FALSE);
    g_print("Starting loop...\n");
    g_main_loop_run(loop);

    return 0;
}
```

I don't think this code needs a lot of explanation, next to the `g_main_loop_new()` [<http://developer.gnome.org/doc/API/2.0/glib/glib-The-Main-Event-Loop.html#g-main-loop-new>] call maybe.

This is the `g_main_loop_new` definition given in the Glib API [<http://developer.gnome.org/doc/API/2.0/glib/>]:

```
GMainLoop* g_main_loop_new (GMainContext *context, gboolean is_running)
```

I won't explain here what a `GMainContext` is, read up the API docs [<http://developer.gnome.org/doc/API/2.0/glib/glib-The-Main-Event-Loop.html#GMainContext>] to figure out what it is. If we give "NULL" as context, the default context will be used. Our loop is not running yet, so we can use `FALSE` as second argument.

Now we got a program that does +- the same thing as our first C code, using the while loop.

We want to compile this program. We could do this manually using `gcc` and some options on the command line, or by using `Automake`, which is the best way to do this, of course.

Try to write a `Makefile.am` file by yourself, and make sure `example1` is built inside the `src/glib-mainloop/` directory.

This is what the `Makefile.am` file could look like:

```
INCLUDES = $(GLIB_CFLAGS)

bin_PROGRAMS = example1

example1_SOURCES = example1.c
example1_LDADD = $(GLIB_LIBS)
```

Let's take a look at one of the most simple things we can do with our mainloop: implementing a timer.

## Using timeouts

These are the steps to perform when implementing a timer in an application:

1. Create a Glib mainloop
2. Write a callback function that will be called on every "tick"
3. Add a timeout to the mainloop
4. Start the mainloop

Adding a timeout is done using the "g\_timeout\_add [[http://developer.gnome.org/doc/API/2.0/glib/glib-The-Main-Event-Loop.html#g\\_timeout\\_add](http://developer.gnome.org/doc/API/2.0/glib/glib-The-Main-Event-Loop.html#g_timeout_add)]" function, which takes a timeout value in microseconds, a function to call on every tick, and a pointer to some object to give to the callback function.

Here's the code:

```
#include <glib.h>

static void callback(gpointer data)
{
    g_print("Callback called\n");
}

int main()
{
    GMainLoop *loop;

    loop = g_main_loop_new(NULL, FALSE);
    g_print("Starting loop...\n");

    g_timeout_add(1000, (GSourceFunc)callback, NULL);
    g_main_loop_run(loop);

    return 0;
}
```

As you can see, we use NULL a user\_data here, because we don't need this functionality.

Adjust Makefile.am so example2 gets built too, build and execute it. Simple, isn't it?

## Conclusion

I can imagine you're not very impressed after reading this. Everything we did can also be done using normal C code using an infinite loop and/or sleep() [<http://www.rt.com/man/sleep.3.html>] calls. This is certainly true, but keep in mind the Glib Mainloop object offers much more functionality than what I showed here: socket polling in 3 lines of code,... Notice all Gnome [<http://www.gnome.org>]/GTK+ [<http://www.gtk.org/>] based applications use a GTK+ mainloop, which is basically a Glib mainloop.

A little exercise for the reader could be to get the user\_data functionality of a timeout working, using some struct containing an integer counting, so you can display "Xth run" instead of "Callback called". Of course you can do this using a static integer inside the callback function, but that's no fun ;-)