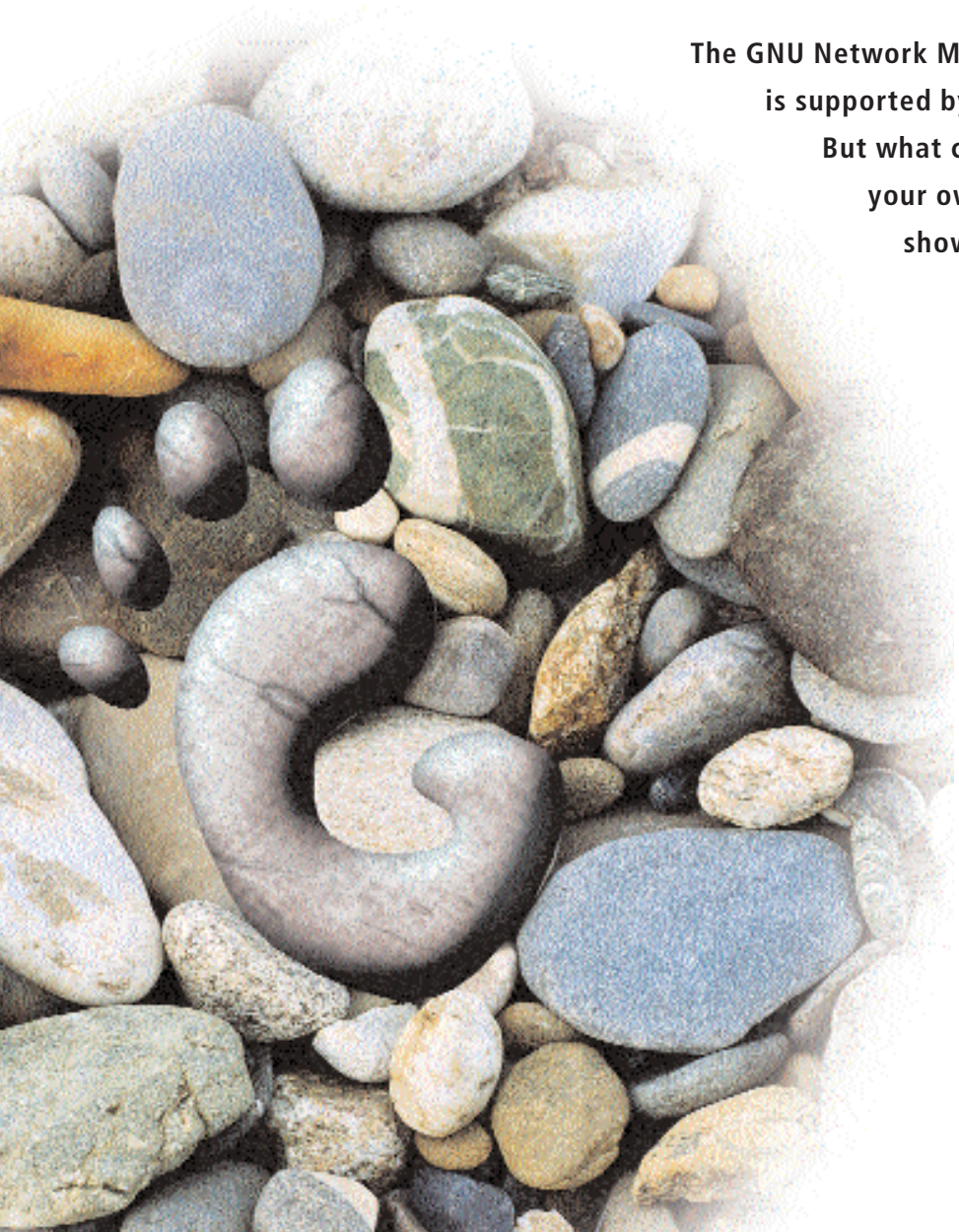
GNOME programming

TAMING THE GNOME

THORSTEN FISCHER



The GNU Network Model Environment (GNOME) is supported by a range of programs. But what could be better than writing your own? In this article we'll show you how.

GNOME is a graphical user environment for Unix systems. Its GNU General Public License status means that it is absolutely free. Perhaps its best attribute is its standard look and feel which creates a consistent appearance and behaviour (in the case of errors for example.) Furthermore, GNOME programs are intended to interact with one another easily. To edit an HTML file for instance, all you need do is drag and drop the file from the file manager to the editor. This is a familiar feature of Microsoft Windows.

The uniform appearance of applications may seem boring but it is one of the main reasons an operating system like Windows has become so widely accepted. GNOME's attempt to bring the advantages of standardisation to a free operating system should therefore be supported. The best way to do this is to create your own applications for GNOME. Here we show you the basic procedures.

Where to begin?

You won't have to download any major files from the Internet to begin programming. Development can begin with the GNOME version shipped with

your current distribution. They all come with a complete GNOME including the developer files. However, if you really can't resist, you'll find newer packages in various formats on the GNOME website (listed below.)

Anyone with a distribution based on the package manager rpm can easily install the downloaded packages (don't forget to uninstall the old packages first!) Install them using the command:

```
rpm -U <packagename.rpm>
```

Normally, that should be it. Anyone who intends to compile larger projects should prepare themselves for a time-consuming orgy of downloading and compiling that will not be hassle-free. You'll get the software from the sites below, or from mirror servers.

If you want an installation that just contains the basics you'll be perfectly happy with glib, gtk+, imlib, ORBit, gnome-libs, libgtop and the gnome-core packages. Installation is described in the README files of the source package. It generally consists of the well-known commands:

```
../configure
make
make install
```

Making a start

The unimaginative but customary "Hello World" program is a standard first program. Linux programmers have already covered this ground, so we'll write a small program that produces just one window. We'll then take this solitary, single window as a cry from GNOME, a call into the big wide world. This approach saves us creating an output function.

Listing 1 shows the simplest (and, as we have seen, the most philosophical) program for GNOME (mini.c):

Listing 1: mini.c

```
#include <gnome.h>

int main (int argc, gchar *argv[]) {

  GtkWidget *my_application;

  gnome_init ("gnomovision", "0.0.1", argc,➥
argv);
  my_application = gnome_app_new ("gnomovis➥
ion", " gnomovision ");

  gtk_widget_show (my_application);

  gtk_main ();

  return (0);

} /* end of mini.c */
```

As you can see, it is no longer necessary to call up *gtk_init* or add gtk.h, as with gtk+. GNOME encapsulates almost all the necessary calls for gtk. However, *gtk_main* still has to be called up to start the "main loop" of the program.

The small program is compiled using the following compiler call:

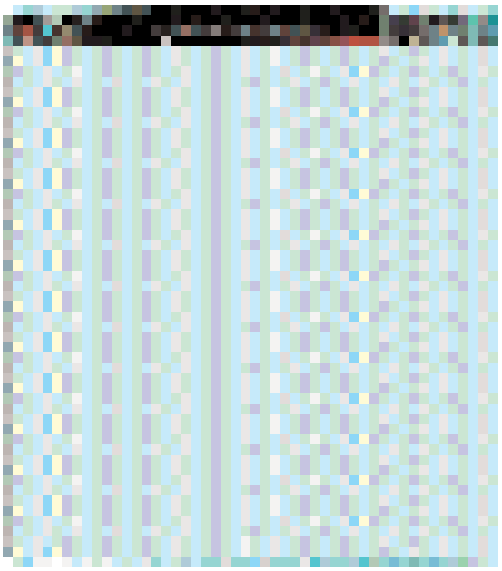
```
gcc mini.c `gnome-config --libs --cflags gno➥
meui` -o mini
```

The finished program is, in the best gtk+ style, a 200x200 pixel window without any content whose only purpose is to be closed with the relevant button on the window.

What happens in these few lines? Firstly, the application itself is declared as a GtkWidget. Then *gnome_init* initialises the program and reports it to GNOME. This feature is required for session management, which ensures that after the next restart, all the programs that are open when you log off are available again in the same place on the desktop.

The two character strings represent the name and the version number of the program. *argc* and *argv* are, as usual, the arguments which are transferred from the command line to the program when the program is started. These can be edited with *popt*, the library GNOME uses as standard. The call for *gnome_app_new* creates the widget, i.e. the application itself, by carrying out various initialisations and providing memory. Again the program name and the title are transferred as they are to appear in the header line of the window. The function *gtk_widget_show ()* shows the window on the screen. The program then goes into the main loop and waits for events.

The strange way in which the compiler is called isn't new to established gtk+ programmers. The small utility *gnome-config* forwards the necessary include parameters (e.g. "-I/opt/gnome/include" or "-L/opt/gnome/lib") to the compiler. This saves the user some typing and is particularly useful if you wish to convert your program to a simple configuration and installation with GNU *autoconf*. It is now time to close the window again.



Listing 2: first.c

```
/*
*  Example program for the gtk+ library from the
*  article for Linux Magazine
*
*  (c) 2000 Thorsten Fischer
*
*  first.c, compile with
*  gcc first.c `gnome-config --libs --cflags gnomeui` -o first
*
*/

#include <gnome.h>

gint create_about_box (void) {
  GtkWidget *aboutdialog;
  gchar *authors [] = {
    "Thorsten Fischer <frosch@cs.tu-berlin.de>",
    "Your name <your name@your provider >",
    NULL };
  gchar *abouttext =
    "gnomovision: The definitive program for GNOME!\n
    This program is subject to the GPL. It may be used and
    passed on
    without any restrictions as long as this copyright notice
    remains in place.
    You can find further information in the file COPYING.";
  aboutdialog = gnome_about_new ("gnomovision", "0.0.2",
    "(c) 1999 Free Software Foundation", (gpointer) authors,
abouttext,
    "./image.png");
  gtk_widget_show (aboutdialog);
  return;
}

gint end_program (GtkWidget *widget, gpointer data) {
  gtk_main_quit ();
  return;
}

static GnomeUIInfo menu_file [] = {
  GNOMEUIINFO_ITEM_STOCK ("Exit", "exit gnomovision",
  end_program, GNOME_STOCK_MENU_EXIT),
  GNOMEUIINFO_SEPARATOR,
  GNOMEUIINFO_END
};

static GnomeUIInfo menu_help [] = {
  GNOMEUIINFO_ITEM_STOCK ("About gnomovision", "About gnomovi
sion",
  create_about_box, GNOME_STOCK_MENU_ABOUT),
  GNOMEUIINFO_END
};

static GnomeUIInfo menu_main [] = {
  GNOMEUIINFO_SUBTREE (N_("File"), menu_file),
  GNOMEUIINFO_SUBTREE (N_("Help"), menu_help),
  GNOMEUIINFO_END
};

static GnomeUIInfo menu_toolbar [] = {
  GNOMEUIINFO_ITEM_STOCK ("Exit", "exit gnomovision",
  end_program, GNOME_STOCK_PIXMAP_EXIT),
  GNOMEUIINFO_SEPARATOR,
  GNOMEUIINFO_ITEM_STOCK ("About", "About gnomovision",
  create_about_box, GNOME_STOCK_PIXMAP_ABOUT),
  GNOMEUIINFO_END
};

gint show_popup (GtkWidget *widget, GdkEvent *event) {
  GtkWidget *popup;
  popup = gnome_popup_menu_new (menu_help);
  gnome_popup_menu_do_popup_modal (popup, NULL, NULL, NULL, e
vent);
  gtk_widget_destroy (popup);
}

int main (int argc, gchar *argv[]) {
  GtkWidget *my_application;
  GtkWidget *abox;
  GtkWidget *abutton;
  gchar buf [40] = "Left click for popup menu!";

  gnome_init ("gnomovision", "0.0.2", argc, argv);

  my_application = gnome_app_new ("gnomovision", " gnomovisio
n ");
  gtk_widget_set_usize (GTK_WIDGET (my_application), 200, 100
);
  gtk_signal_connect (GTK_OBJECT (my_application), "delete_ev
ent",
  GTK_SIGNAL_FUNC (end_program), NULL);

  abox = gtk_hbox_new (FALSE, 0);
  gnome_app_set_contents (GNOME_APP (my_application), abox);

  abutton = gtk_button_new_with_label (buf);
  gtk_box_pack_start (GTK_BOX (abox), abutton, TRUE, TRUE, 0);
  gtk_signal_connect (GTK_OBJECT (abutton), "clicked", GTK_SI
gNAL_FUNC
  (show_popup), NULL);

  gnome_app_create_menus (GNOME_APP (my_application), menu_ma
in);
  gnome_app_create_toolbar (GNOME_APP (my_application), menu_
toolbar);

  gtk_widget_show (abutton);
  gtk_widget_show (my_application);

  gtk_main ();

  return;

} /* end of first.c */
```

What's going on here?

Anyone who, after closing mini, views the list of current processes with ps will see that the small program still seems to be hanging around in memory. This is unsurprising, as we only closed the window with the help of the window manager. To exit the program we need to call up the exit program function specifically.

In order to bring about events, GNOME (once again, and not for the last time!) uses gtk+. This toolkit uses what are known as callbacks to call up functions at the events defined by the programmer. The individual widgets which make up a window are linked with the functions to be called based on certain conditions via the function:

```
gtk_signal_connect ();
```

This function ensures that the loop *gtk_main* responds to the events. If our window has a widget such as an exit button by the name of *exitbutton*, the program could contain the following line based on its definition:

```
gtk_signal_connect (GTK_OBJECT (exitbutton)➥
, "clicked",
GTK_SIGNAL_FUNC
(gtk_main_quit), NULL);
```

Now each time the user clicks on the exit button the program's main loop is interrupted, allowing the user to exit the program. It makes sense when these events occur to call up a function you have written yourself showing a dialog box that offers to save changed and still open files. The last parameter, NULL, can be filled with any pointer which can transfer data to the function.

Each widget can respond to standard events (is clicked on, is closed etc.) Most widgets also have their own specific events, which the user can take advantage of. The widget GtkListItem, which describes the individual entries in a list, responds to the selection of individual entries in the list – a property which would serve no purpose for a simple button.

Navigation

Naturally, a graphical user interface program cannot manage without a menu. In most cases, a toolbar with icons which provide shortcuts to the most frequently used functions is also desirable. GNOME has a simple system for creating menus. This system restricts the programmer's actions to defining the names of menu entries and icons and their functions.

The structure in which the menu entries are defined is called *GnomeUIInfo*. The following call creates a main menu containing a file and help menu:

```
GnomeUIInfo menu_main [] = {
  GNOMEUIINFO_SUBTREE (N_("File"), menu_file),
  GNOMEUIINFO_SUBTREE (N_("Help"), menu_help),
  GNOMEUIINFO_END
};
```

Based on the same model, the two structures menu_file and menu_help are defined and displayed using the call for the function *gnome_app_create_menus*. You proceed in exactly the same way with a toolbar, which you can define as follows:

Info

GNOME home page
http://www.gnome.org/

GNOME Developers web site
http://developer.gnome.org/

GTK Information
http://www.gtk.org

```
GnomeUIInfo toolbar [] = {
GNOMEUIINFO_ITEM_STOCK ("Exit", "Exit gnom
esite",
end_application, GNOME_STOCK_PIXMAP_EXIT),
GNOMEUIINFO_SEPARATOR,
GNOMEUIINFO_ITEM_STOCK ("Help", "Help! Hel
p!",
end_application, GNOME_STOCK_PIXMAP_HELP),
GNOMEUIINFO_END
};
```

The call for GNOMEUIINFO_END, which defines the closure of the menu or toolbar, is always important. GNOME generates a wealth of pixmaps – small images and icons – which should be used in a consistent way to obtain the standard look and feel. Table 1 lists some of the pixmaps used for toolbars. A fully comprehensive list would be too exhaustive.

Pop-up menus are produced in a similar way. They appear at the click of a mouse and only in certain parts of the window – i.e. in particular widgets. The procedure uses the aforementioned method of signals and callback functions. We simply link the widget in which we wish to be able to call up the pop-up menu with the function which creates the menu. Menu entries can be easily added to the menu, removed and switched on or off using the relevant commands. We can also make "intelligent" or context-sensitive menus which display certain functions only in certain situations. A "Save" menu, for instance, only makes sense if some data has changed since the last save.

About us

GNOME applications also display dialog boxes to display information or allow users to set preferences. One of the better known examples is the About box, which offers information about the program and its authors. This standard box is created using a call of the function:

```
gnome_about_new ();
```

Table 1: Pixmaps for toolbars

GNOME_STOCK_PIXMAP_NEW	New (e.g. for a new file)
GNOME_STOCK_PIXMAP_OPEN	Open (e.g. an existing file)
GNOME_STOCK_PIXMAP_SAVE	Save (e.g. a changed file)
GNOME_STOCK_PIXMAP_SAVE_AS	Save as (a new file name)
GNOME_STOCK_PIXMAP_CUT	Cut (e.g. a piece of text)
GNOME_STOCK_PIXMAP_COPY	Copy (e.g. a piece of text)

The function lists the authors, the program name, version number, copyright notice, an explanatory test and may display a pixmap. Everything is laid out in an attractive dialog box. The exact procedure is shown in Listing 2 (first.c), which also repeats everything covered so far.

All finished

That was a lot to grasp! All the basics discussed here are packed into one example program. It is compiled in exactly the same way as the first example, except of course that you should not enter "mini.c" in this case. As you can see, I have given the *gnomovision* program the version number 0.0.2. This leap to the next version seems justified in view of the increased functionality.

We shouldn't leave this listing as it is without some final comments. The small buffer *buf* contains the text to be put on the button. Also, I define a box with *abox*. The reason for this is that gtk+ uses what are known as "Container" widgets in order to pack and organise other widgets. Not all widgets can function as containers, whilst others are designed exclusively as containers. At least one box is needed to pack and display other widgets. Packing (shown here at the button *abutton*) is executed quite easily with the call for the function *gtk_box_pack_start*.

gnome_app_set_contents informs GNOME that the interesting part of the program is executed within the box. The only new thing is the function *create_about_box* (described above). In the same directory as first.c there must be an image named *image.png*, which improves the visuals of the dialog box. Other graphics formats may also be used for the image.

It is good practice to define or create the widget first in gtk+ or GNOME. Then define its properties and link it with the relevant signals. Finally, show it. When you show it you should do the reverse of what you did when you created it. The least important buttons and so on come first and the window widget last. This prevents ugly cross-fading effects while waiting for the window to be displayed.

More documentation

Although developer documentation on the GNOME project is available, it is still extremely disorganised. However, the project organisers have now set up a special website for developers. This will hopefully help to concentrate all the documentation in one place. In this respect, gtk+ is also undergoing improvements. ■