Maemo Diablo Testing the installation

Training Material

February 9, 2009

Contents

Chapter 1

Testing the installation

1.1 Testing Scratchbox

The following shows how to create a small non-graphical Hello World program, to verify that the Scratchbox environment works:

```
/**
 * helloworld.c
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 *
 * Simple standard I/O (printf)-based Hello World that we can use to
 * test our toolchains.
 */

#include <stdio.h> /* printf */

/* main implementation */
int main(int argc, char** argv) {

  printf("Hello world\n");

  /* In Linux, each process upon termination must set its exit code.
     Exit code 0 means success to whoever executed this program. It
     is routinely used inside scripts to test whether running some
     program succeeded or not. Other exit codes mean failure. Each
     program is free to use different non-zero codes to signify
     different kinds of failures. These are normally listed in the
     manual page for the program (since there is no standard). If you
     forget to set your exit code, it will be undefined. */
  return 0;
}
```

Listing 1.1: Program listing for our Hello World (helloworld.c)

First, it has to be verified that the proper directory is chosen. This can be done by using pwd (print working directory). At this point, the directory should be the home directory:

```
[sbox-DIABLO_X86: ~] > pwd
/home/user
```

Then, start an editor and write the small hello world program (you may use the above listing as a template if you wish):

```
[sbox-DIABLO_X86: ~] > nano helloworld.c
```

nano is a GNU version of "pico" editor, which is a simple text file editor. Use Control+character to execute the commands listed on the bottom of the screen. WriteOut means "save". You may also use vi or an external editor to the SDK environment (see below for hints on using vi and emacs).

```
[sbox-DIABLO_X86: ~] > gcc -Wall -g helloworld.c -o helloworld
[sbox-DIABLO_X86: ~] > ls -F hello*
helloworld* helloworld.c
```

The -g option to gcc tells the compiler to add debugging symbols to the generated output file. -Wall will tell the compiler to enable most of the syntax and other warnings that the source code could trigger. -o helloworld then tells the output filename to which gcc will write the result binary.

The -F option to ls is mainly useful when working with a non-color terminal (e.g. paper) to indicate the type of different files. The asterisk after helloworld signifies that the file is an executable.

```
[sbox-DIABLO_X86: ~] > ./helloworld
Hello world
```

Running the binary should not produce any surprises.

```
[sbox-DIABLO_X86: ~] > file helloworld
helloworld: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.0, dynamically linked (uses shared libs), not stripped
```

The file tool is a generic utility that will load some bytes from the start of the given file and then use its internal database to decode what the file might "mean". In this case, it will correctly decode the file as a X86 format binary file.

```
[sbox-DIABLO_X86: ~] > ldd helloworld
    linux-gate.so.1 =>  (0xffffe000)
    libc.so.6 => /lib/libc.so.6 (0xb7e9f000)
    /lib/ld-linux.so.2 (0xb7fd2000)
[sbox-DIABLO_X86: ~] > ls -l /lib/libc.so.6
lrwxrwxrwx  1 user user 11 Nov 12 15:52 /lib/libc.so.6 -> libc-2.5.so
[sbox-DIABLO_X86: ~] > ls -l /lib/libc-2.5.so
-rwxr-xr-x  1 user user 1213256 Sep  7 13:28 /lib/libc-2.5.so
```

The names of dynamic libraries that the executable uses will be shown on the left-hand column, and the files where the libraries live on the system if executing the program will be shown on the right-hand column. After that, use ls to check out the exact version of the C library that is used in the SDK by using the "long listing format" -l option (running these commands using the ARMEL target would yield more or less the same results). N.B. The linux-gate.so.1 is a so-called hack to support a certain way of doing system calls on the X86 architecture, and is not always present on newer systems.

When comparing the version of libc used on the real system with ls -l, it will probably show a difference (in version numbers). This means that the executables that were built inside sbox use libraries that are also inside sbox. This also means a stable development platform, especially when working in team where each member has their own Linux, which they have customised. This might not seem very important at this stage, but when encountering all the different tools that are used in free software development, this feature of sbox will come in handy.

Scratchbox does not contain any logic to emulate the kernel (or to use a different kernel for running programs inside sbox). The only easy possibility for this is using the sbrsh CPU-transparency option.

vi

It is also possible to use vi (Visual Interactive) editor inside sbox. It is possible to install own favourite editors inside sbox (with the debian-devkit), but the following examples will use nano, since it is the easiest to start with. To learn vi, it is best to ask an Internet search engine for a "vi tutorial". There are lots of them to be found. To understand why vi can be considered to be "strange", it is useful to know its history first. Using vi is optional of course.

The version of vi that is commonly installed on Linux systems is really vim (VI iMproved), which is a more user friendly vi, including syntax high-lighting and all kinds of improvements. sbox has a program called vimtutor installed to help in learning the use of vi interactively.

It is also fairly simple to use existing editors. /scratchbox/users/x/home/x/ is the home directory of user x when accessing it from the real Linux desktop. Ubuntu comes with gedit, which is a fairly good graphical editor that also supports syntax high-lighting and multiple tabs for editing multiple files at the same time.

And as a final note, also emacs can be used.

Here is how to do it:

- Start emacs outside of sbox
- In emacs, use M-x server-start
- Inside sbox use emacsclient filename to open the file for editing in your emacs

1.2 Writing a GUI Hello World

The following example shows how to write the first GUI program. N.B. Only GTK+ library is used here, meaning that the platform provided widgets or coding style are not utilised. That will be discussed soon.

```
/**
 * gtk_helloworld-1.c
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 *
 * A simple GTK+ Hello World. You need to use Ctrl+C to terminate
 * this program since it doesn’t implement GTK+ signals (yet).
 */

#include <stdlib.h> /* EXIT_* */
/* Introduce types and prototypes of GTK+ for the compiler. */
#include <gtk/gtk.h>

int main(int argc, char** argv) {
  /* We'll have two references to two GTK+ widgets. */
  GtkWindow* window;
  GtkLabel* label;

  /* Initialize the GTK+ library. */
  gtk_init(&argc, &argv);

  /* Create a window with window border width of 12 pixels and a
     title text. */
  window = g_object_new(GTK_TYPE_WINDOW,
    "border-width", 12,
    "title", "Hello GTK+",
    NULL);

  /* Create the label widget. */
  label = g_object_new(GTK_TYPE_LABEL,
    "label", "Hello World!",
    NULL);

  /* Pack the label into the window layout. */
  gtk_container_add(GTK_CONTAINER(window), GTK_WIDGET(label));

  /* Show all widgets that are contained by the window. */
  gtk_widget_show_all(GTK_WIDGET(window));

  /* Start the main event loop. */
  g_print("main: calling gtk_main\n");
  gtk_main();

  /* Display a message to the standard output and exit. */
  g_print("main: returned from gtk_main and exiting with success\n");

  /* The C standard defines this condition as EXIT_SUCCESS, and this
     symbolic macro is defined in stdlib.h (which GTK+ will pull in
     in-directly). There is also a counter-part for failures:
     EXIT_FAILURE. */
  return EXIT_SUCCESS;
}
```

Listing 1.2: A simple GTK+ Hello World (gtk_helloworld-1.c)

Build your program:

```
[sbox-DIABLO_X86: ~] > gcc -Wall -g gtk_helloworld-1.c -o gtk_helloworld-1
gtk_helloworld-1.c:15:21: gtk/gtk.h: No such file or directory
gtk_helloworld-1.c: In function ‘main’:
gtk_helloworld-1.c:20: error: ‘GtkWindow’ undeclared (first use in this function)
gtk_helloworld-1.c:20: error: (Each undeclared identifier is reported only once
gtk_helloworld-1.c:20: error: for each function it appears in.)
gtk_helloworld-1.c:20: error: ‘window’ undeclared (first use in this function)
gtk_helloworld-1.c:21: error: ‘GtkLabel’ undeclared (first use in this function)
gtk_helloworld-1.c:21: error: ‘label’ undeclared (first use in this function)
gtk_helloworld-1.c:24: warning: implicit declaration of function ‘gtk_init’
gtk_helloworld-1.c:28: warning: implicit declaration of function ‘g_object_new’
gtk_helloworld-1.c:28: error: ‘GTK_TYPE_WINDOW’ undeclared (first use in this function)
gtk_helloworld-1.c:34: error: ‘GTK_TYPE_LABEL’ undeclared (first use in this function)
gtk_helloworld-1.c:39: warning: implicit declaration of function ‘gtk_container_add’
gtk_helloworld-1.c:39: warning: implicit declaration of function ‘GTK_CONTAINER’
gtk_helloworld-1.c:39: warning: implicit declaration of function ‘GTK_WIDGET’
gtk_helloworld-1.c:42: warning: implicit declaration of function ‘gtk_widget_show_all’
gtk_helloworld-1.c:45: warning: implicit declaration of function ‘g_print’
gtk_helloworld-1.c:46: warning: implicit declaration of function ‘gtk_main’
```

Compiling leads to a lot of errors

As can be seen, this does not look at all promising. At the start of the source code, there was #include. The compiler needs to be told where it should look for that critical GTK+ header file. It is also quite likely that some special flags need to be passed to the compiler in order for it to use the proper compilation settings when building GTK+ software. How to decide which flags to use?

This is where a tool called pkg-config comes to the rescue. It is a simple program that provides a unified interface to output compiler, linker flags and library version numbers. Its utility will be discussed later, when starting the automating of the building process. For now, the pkg-config will be used manually.

```
[sbox-DIABLO_X86: ~] > pkg-config --list-all | sort
.. listing cut to include only relevant libraries ..
dbus-glib-1     dbus-glib - GLib integration for the free desktop message bus
gconf-2.0       gconf - GNOME Config System.
gdk-2.0         GDK - GIMP Drawing Kit (x11 target)
gdk-pixbuf-2.0  GdkPixbuf - Image loading and scaling
glib-2.0        GLib - C Utility Library
gnome-vfs-2.0   gnome-vfs - The GNOME virtual file-system libraries
gtk+-2.0        GTK+ - GIMP Tool Kit (x11 target)
hildon-1        hildon - Hildon widgets library
hildon-fm-2     hildon-fm - Hildon file management widgets
pango           Pango - Internationalised text handling
x11             X11 - X Library
```

Listing the installed `pkg-config` packages.

`pkg-config` also has some other commands that will prove useful:

```
[sbox-DIABLO_X86: ~] > pkg-config --modversion gtk+-2.0
2.10.12
```

Listing the version of an installed `pkg-config` package, GTK+ in this case.

```
[sbox-DIABLO_X86: ~] > pkg-config --cflags gtk+-2.0
-I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include -I/usr/include/atk-1.0
-I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/glib-2.0
-I/usr/lib/glib-2.0/include -I/usr/include/freetype2
-I/usr/include/libpng12
```

Getting the necessary options and flags for `gcc` that allow using proper header files with GTK+

.

As can be seen, there are many. With this version of GTK+, all of them are -I options. They are used to tell the compiler which additional directories to check for system header files in addition to the default ones.

```
[sbox-DIABLO_X86: ~] > pkg-config --libs gtk+-2.0
-lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm
-lpangocairo-1.0 -lpango-1.0 -lcairo -lgobject-2.0 -lgmodule-2
-ldl -lglib-2.0
```

Getting the necessary options and flags for `gcc` that allow linking against the GTK+ library.

When linking the application, the linker has to be told which libraries to link against. In fact, the whole program linking phase will fail (as shown shortly) without this information.

Now it is time to try and compile the software again, this time using the compilation flags that pkg-config provides:

```
[sbox-DIABLO_X86: ~] > gcc -Wall -g gtk_helloworld-1.c  \
 ‘pkg-config --cflags gtk+-2.0‘ -o gtk_helloworld-1
/var/tmp/ccQ14x4c.o: In function ‘main’:/home/user/gtk_helloworld-1.c:24:
 undefined reference to ‘gtk_init’
:/home/user/gtk_helloworld-1.c:28: undefined reference to ‘gtk_window_get_type’
:/home/user/gtk_helloworld-1.c:28: undefined reference to ‘g_object_new’
:/home/user/gtk_helloworld-1.c:34: undefined reference to ‘gtk_label_get_type’
:/home/user/gtk_helloworld-1.c:34: undefined reference to ‘g_object_new’
:/home/user/gtk_helloworld-1.c:39: undefined reference to ‘gtk_widget_get_type’
:/home/user/gtk_helloworld-1.c:39: undefined reference to ‘g_type_check_instance_cast’
:/home/user/gtk_helloworld-1.c:39: undefined reference to ‘gtk_container_get_type’
:/home/user/gtk_helloworld-1.c:39: undefined reference to ‘g_type_check_instance_cast’
:/home/user/gtk_helloworld-1.c:39: undefined reference to ‘gtk_container_add’
:/home/user/gtk_helloworld-1.c:42: undefined reference to ‘gtk_widget_get_type’
:/home/user/gtk_helloworld-1.c:42: undefined reference to ‘g_type_check_instance_cast’
:/home/user/gtk_helloworld-1.c:42: undefined reference to ‘gtk_widget_show_all’
:/home/user/gtk_helloworld-1.c:45: undefined reference to ‘g_print’
:/home/user/gtk_helloworld-1.c:46: undefined reference to ‘gtk_main’
:/home/user/gtk_helloworld-1.c:49: undefined reference to ‘g_print’
collect2: ld returned 1 exit status
```

Trying to build again, with proper C flags.

The command above might seem somewhat strange to someone not having used UNIX command shells. What is happening here is the back-tick expansion. It is an operation where the shell will start another shell to execute just the text inside the back-ticks. In this case, another shell is started to run `pkg-config --cflags gtk+-2.0`. Normal output from the commands is then read into the main shell, and this output is replaced into the location where the back-ticks were. N.B. It is very important to use the ‘ character. Not ’, nor the other quote character that might be used in a Swedish keyboard layout (also used in Finland). In some keyboard layouts, it will be necessary to press space after the back-tick since it is also used for character composition (try back-tick and letter ’a’).

Something like $(pkg-config ..) might also be encountered. This is the same operation as back-tick. However, back-tick is more portable across antique UNIX shells. Nowadays, it is a matter of taste which way to use it.

The errors printed by gcc are quite different this time. These errors come from ld, which is the binary code linker in Linux systems and it is complaining about missing symbols (the undefined references). Obviously something is still missing.

The linker needs to be told where to find the missing symbols. Since it is the linker this is all about, and not the compiler, the missing symbols are found in the library files. To fix the problem (again with the back-ticks), `pkg-config --libs` can be used:

```
[sbox-DIABLO_X86: ~] > gcc -Wall -g gtk_helloworld-1.c  \
 ‘pkg-config --cflags gtk+-2.0‘ -o gtk_helloworld-1  \
 ‘pkg-config --libs gtk+-2.0‘
[sbox-DIABLO_X86: ~] >
```

Successful compile and linking.

The order and placement of the pkg-configs above is important: the `--cflags` need to be placed as early as feasible, but the `--libs` must come last (this does matter in some problematic linking scenarios).

The next step is to repeat the basic commands that were used before with the non-GUI hello world:

```
[sbox-DIABLO_X86: ~] > ls -l gtk_helloworld-1
-rwxrwxr-x  1 user user 16278 Nov 20 00:22 gtk_helloworld-1
[sbox-DIABLO_X86: ~] > file gtk_helloworld-1
gtk_helloworld-1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.0, dynamically linked (uses shared libs), not stripped
[sbox-DIABLO_X86: ~] > ldd gtk_helloworld-1
  linux-gate.so.1 =>  (0xffffe000)
  libgtk-x11-2.0.so.0 => /usr/lib/libgtk-x11-2.0.so.0 (0xb7c4c000)
  libgdk-x11-2.0.so.0 => /usr/lib/libgdk-x11-2.0.so.0 (0xb7bc8000)
  libatk-1.0.so.0 => /usr/lib/libatk-1.0.so.0 (0xb7bad000)
  libgdk_pixbuf-2.0.so.0 => /usr/lib/libgdk_pixbuf-2.0.so.0 (0xb7b97000)
  libm.so.6 => /lib/libm.so.6 (0xb7b71000)
  libpangocairo-1.0.so.0 => /usr/lib/libpangocairo-1.0.so.0 (0xb7b68000)
  libpango-1.0.so.0 => /usr/lib/libpango-1.0.so.0 (0xb7b2b000)
  libcairo.so.2 => /usr/lib/libcairo.so.2 (0xb7ab5000)
  libgobject-2.0.so.0 => /usr/lib/libgobject-2.0.so.0 (0xb7a7a000)
  libgmodule-2.0.so.0 => /usr/lib/libgmodule-2.0.so.0 (0xb7a76000)
  libdl.so.2 => /lib/libdl.so.2 (0xb7a71000)
  libglib-2.0.so.0 => /usr/lib/libglib-2.0.so.0 (0xb79dd000)
  libc.so.6 => /lib/libc.so.6 (0xb78b2000)
  libX11.so.6 => /usr/lib/libX11.so.6 (0xb77bd000)
  libXfixes.so.3 => /usr/lib/libXfixes.so.3 (0xb77b8000)
  libXtst.so.6 => /usr/lib/libXtst.so.6 (0xb77b3000)
  libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0xb7788000)
  libXext.so.6 => /usr/lib/libXext.so.6 (0xb777a000)
  libXrender.so.1 => /usr/lib/libXrender.so.1 (0xb7771000)
  libXi.so.6 => /usr/lib/libXi.so.6 (0xb7769000)
  libXrandr.so.2 => /usr/lib/libXrandr.so.2 (0xb7762000)
  libXcursor.so.1 => /usr/lib/libXcursor.so.1 (0xb7759000)
  /lib/ld-linux.so.2 (0xb7fc3000)
  libpangoft2-1.0.so.0 => /usr/lib/libpangoft2-1.0.so.0 (0xb772b000)
  libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0xb76c6000)
  libz.so.1 => /usr/lib/libz.so.1 (0xb76b7000)
  libpng12.so.0 => /usr/lib/libpng12.so.0 (0xb7692000)
  libXau.so.6 => /usr/lib/libXau.so.6 (0xb768f000)
  libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0xb7689000)
  libexpat.so.1 => /usr/lib/libexpat.so.1 (0xb7669000)
```

Victory at last!

As can be seen from the last ldd listing, this simple Hello World manages to require quite a number of other libraries to run. The program directly only requires GTK+, but GTK+ needs GDK (and all the other libraries that were covered in the introduction). Those libraries in turn need other libraries and so on.

So, what is seen here is almost the full list of all required libraries to run. Almost, because modern UNIX systems (and Linux) can also load libraries on demand (called runtime dynamic module loading).

This might make one wonder, whether writing simple a Hello World really is so painful. It is actually much simpler in real life. The reason why this chapter introduces the various errors is that they will be encountered in actual situations quite early on. This chapter serves as a reference to some possible errors, and (hopefully) also show a solution.

All of these tools will be needed later on, when starting the packaging of the software, and they will not be covered at this level of detail there.

1.3 Running the GUI Hello World

Let's try to execute our nice Hello World (inside sbox):

```
[sbox-DIABLO_X86: ~] > ./gtk_helloworld-1
gtk_helloworld-1[4759]: GLIB WARNING ** Gtk - cannot open display:
[sbox-DIABLO_X86: ~] > echo $DISPLAY
```

Don't despair, the end is near.

Seems that GTK+ is having problems opening the connection to the X server. This can be verified by displaying the contents of the DISPLAY environmental variable, and indeed, it comes out empty. If the DISPLAY variable contains :0.0, it means that the value has been copied from the real graphical session into sbox, and clients will try to connect to the real X server (and probably fail in authentication).

Xephyr was set up in the previous chapter, so now it has to be started so that it can be used as the server for all clients running inside the Scratchbox session.

1.4 Starting virtual X server (Xephyr)

Open another terminal emulator (don't close your sbox session).

Start up the server with:

```
user@system:~$ Xephyr :2 -host-cursor -screen 800x480x16 -dpi 96 -ac  \
 -extension Composite
```

Starting Xephyr for use with the SDK.

The first parameter is the Display number (`:2`) that X server should start on (and provide to clients). `:2` is used here since it is normally unused in regular Linux desktop environments.

The `screen` parameters tells Xephyr how large the screen should be (in pixels) and how many bits to use for color-information (16). This is the resolution in pixels of Internet Tablets. `-dpi 96` tells the server to tell its clients that the logical to physical size mapping should be done with 96 dots-per-inch setting (should the client request that information). The DPI setting is mainly important when dealing with fonts and text.

`-ac` tells Xephyr that any client may connect to it. This means that you should be extremely careful about your networking environment so that rogue users will not target your Xephyr with their own clients.

The last parameter (`-extension Composite`) disables the Composite extension.

When Xephyr starts, it will connect to the X server given in the `DISPLAY` environmental variable that **it** sees. Do not modify or touch your real `DISPLAY` variable that Xephyr sees.

Figure 1.1: Only the X server running

By default the server will have one screen, and it will be filled by the default X server background pattern (a tight braid made out of white pixels on black).

Note that the terminal emulator (more specifically, the shell that the emulator started) is waiting for Xephyr to end. If you ever need to kill all the graphical applications running in the SDK, you can just close your Xephyr. This will leave the daemons running inside sbox (D-Bus and friends). Normally this is not a good idea. To ask the foreground process to terminate itself, use `Ctrl+c`. Inside sbox this same technique can be used to terminate a graphical client that you start from the command line (as will be done shortly).

1.5 Directing the client to virtual server

Now that we have an X server running, it's time to switch back to sbox.

We start by setting the environmental variable to use a local domain socket to the X server, and tell all X clients to connect to Display number 2, since that's where we just started our Xephyr on:

```
[sbox-DIABLO_X86: ~] > export DISPLAY=:2
[sbox-DIABLO_X86: ~] > ./gtk_helloworld-1
main: calling gtk_main
[[Ctrl+c]]
```

Setting the correct `DISPLAY` content and running a simple GUI program.

We'll need to terminate the program with `Ctrl+c`, since it doesn't implement any graphical methods of closing it. Note that you'll need to set your `DISPLAY` correctly on each sbox login (or target switch).

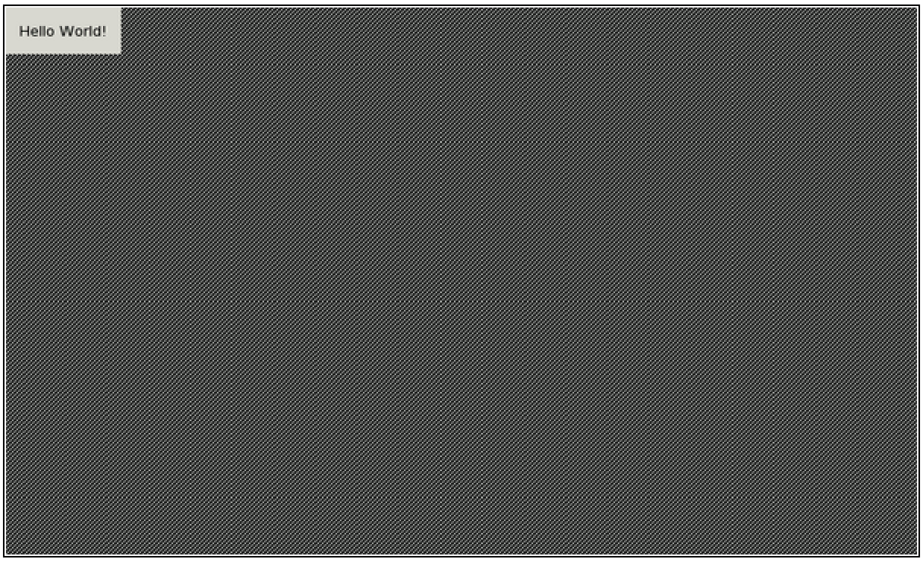


Figure 1.2: A puny little Hello World

Not really impressive, is it? If you remember what a window manager is, you will note that since it's missing (we didn't start any for the X server) you cannot control the Hello World with your mouse. Kill it with `Ctrl+c` for now (we'll start it again in a moment).

To quickly detach the foreground process from your shell and continue running it in the background, use the following key-combination `Ctrl+z` and then use the shell command `bg`. Between those two steps the process will be "paused".

1.6 Starting the Application Framework

The next step is to have a nice graphical environment that will implement a nicer graphical screen. We will start a series of clients each of which have a specific role. These were introduced before.

To start the Application Framework (referred to as AF from now on), we can use a handy script that comes with the SDK:

```
[sbox-DIABLO_X86: ~] > af-sb-init.sh start
Sample files present.
Starting DBUS system bus
Starting D-BUS session bus daemon
Starting Maemo Launcher: maemo-launcher
maemo-launcher: error rising the oom shield for pid=4847 status=5632.
Starting Sapwood image server
Starting Matchbox window manager
sapwood-server[4858]: GLIB INFO default - server started
Starting clipboard-manager
Starting Keyboard
maemo-launcher: invoking '/usr/bin/hildon-input-method.launch'
Starting Hildon Desktop
maemo-launcher: invoking '/usr/bin/hildon-desktop.launch'
.. listing cut for brevity ..
[sbox-DIABLO_X86: ~] >
```

Using the `af-sb-init.sh` script to start the AF.

We use `start` and `stop` parameters to start and stop the graphical environment. If everything works, you should see a screen resembling this one:

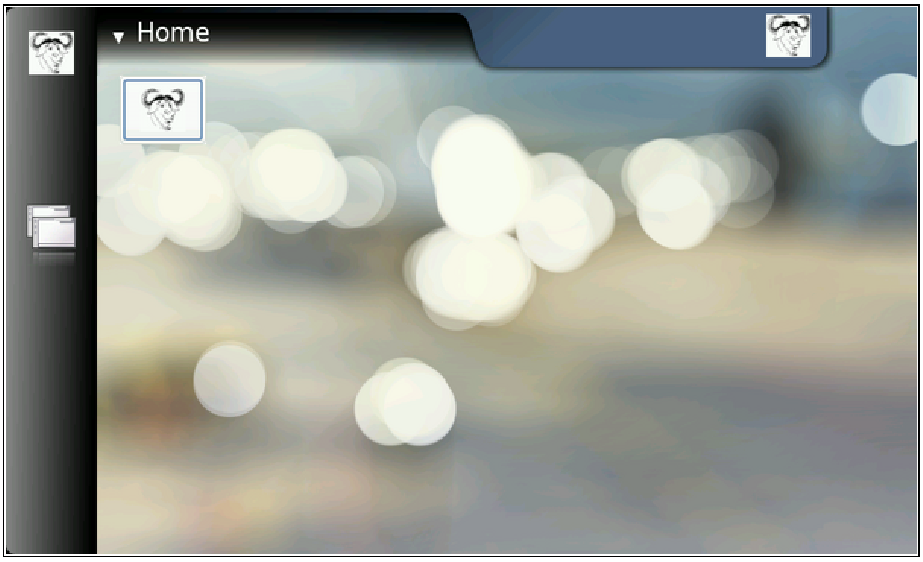
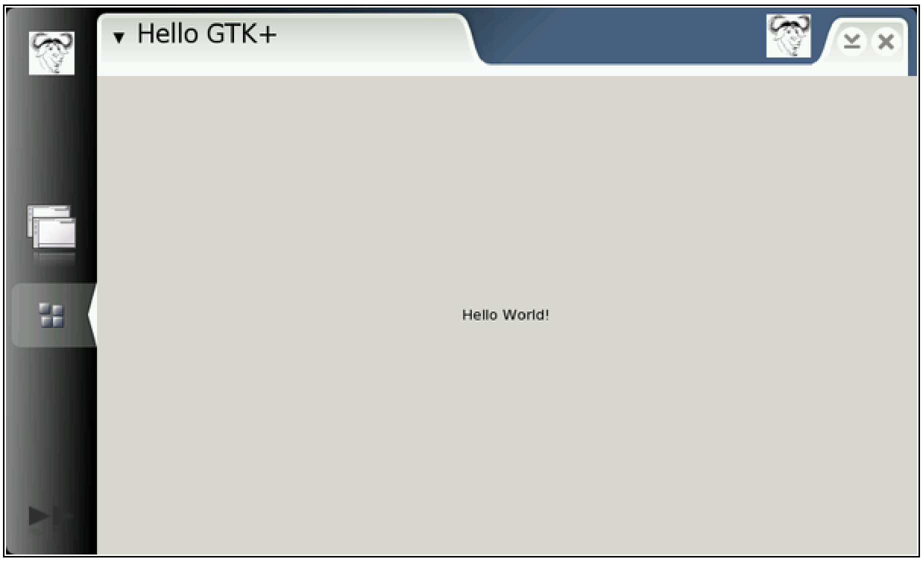


Figure 1.3: Much better

1.7 Running Hello World in the AF

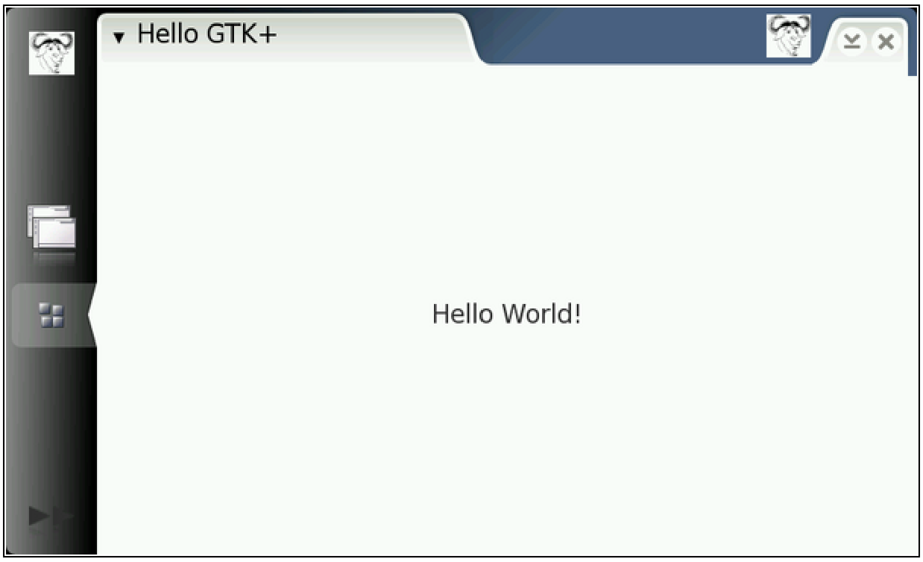
While AF is running, let's start our Hello World again (`./gtk_helloworld-1` in sbox):



Since there now is a window manager running, our client will get much larger window to draw in. GTK+ will scale the widget accordingly (only there is only one widget in our Hello World). Note that the screen still looks a bit off. If you "close the application" by pressing the X in top-right corner, you will notice that our Hello World will disappear from the screen. Since we haven't implemented signals yet and thus don't handle window destruction, our Hello World application will only be hidden. It is still running (as you can see in your sbox terminal emulator since the shell doesn't display its prompt). Stop the Hello World with `Ctrl+c`.

Next we'll use a SDK utility script called `run-standalone.sh`. Its job is to setup correct environmental variables for themes and communication for the command that is given to it as its command line parameter.

Use `run-standalone.sh ./gtk_helloworld-1` to start your X client again:



The screen is still a bit off (we don't get nice borders around our main GtkLabel widget) but looks already somewhat better. The text is scaled to be more in sync with the other text sizes and also the color is in sync with the platform color (it's not gray anymore).

In "maemo Application Development" material you'll learn how to adapt our application for maemo, so that it will "sit better" in the environment. You'll also learn how to react to HID-events, how to use widgets and how to package your software so that it can be easily distributed to users and installed on Internet Tablets.