Software development for the OpenMoko Linux phone

A free and open platform for mobile communications

Skill Level: Intermediate

Peter Seebach (developerworks@seebs.plethora.net)
Freelance writer
Freelance

13 Nov 2007

The OpenMoko environment provides a completely free development environment for running application and system code on supported phone hardware, eliminating all dependency on proprietary code. This tutorial introduces the OpenEmbedded build environment used to create filesystem images for OpenMoko phones, such as the Neo 1973.

Section 1. Before you start

About this tutorial

This tutorial introduces the OpenMoko development platform and shows the details of using it to target the Neo 1973, a unique cell phone developed and sold with the intent of working anywhere, on any network, and running whatever software you want to develop for it.

Tutorial objectives

After a brief overview of the configuration and flashing process, I show you the files you need and what to do with them. Following that, I get into the details of software development. Rather than duplicate existing documentation, this tutorial shows you the big picture and brings everything together.

Prerequisites

This tutorial is for Linux developers with basic software development experience, and either experience with or interest in telephony and embedded systems. Some basic familiarity with the command line is assumed.

You don't need a Neo 1973 phone (or any other) to understand the tutorial, but it will be most rewarding if you have at least a working OpenMoko target, even if it's emulated. I wrote the tutorial using Ubuntu as a host environment, but virtually any Linux® system should work. It's also possible to use Mac OS X or even Windows® as host environments.

Section 2. Platform overview

Is it a phone or is it software?

First International Computer produces the Neo 1973, which is often referred to as "an OpenMoko." In fact, OpenMoko is a Linux platform and associated applications, not a phone. It is, however, a Linux application for phones, and as of this writing, the Neo 1973 is the only fully viable phone to run it on. You can also build an emulated environment for software development, which I cover in this tutorial, but it won't connect to a phone network very well.

OpenEmbedded

OpenMoko is built using OpenEmbedded for cross-compilation (see Resources for a link to more information on OpenEmbedded). OpenEmbedded provides tools and support for build and release functions, whether you're trying to build a native program with debugging or an optimized target binary. As a result, developing for OpenMoko requires getting a feel for how to use OpenEmbedded.

MokoMakefile

One of the difficulties of a new environment is getting everything set up, files in the right places, and so on. Enter the MokoMakefile (see Resources for a link), a huge makefile that does all of this for you for the OpenMoko environment. The MokoMakefile is being developed and maintained by Rod Whitby; it's not yet part of the official OpenMoko environment.

Set up an emulator

As I mentioned, if you don't have a phone handy, you can also set up an emulator. The QEMU emulator provides a fairly full-featured emulation of the OpenMoko phone environment. One of the options is a branch of QEMU that provides nearly complete emulation of the phone's hardware, lacking only the GPS unit. For more information about setting up a QEMU-based emulator, have a look at the OpenMoko Wiki page on "OpenMoko under QEMU" (see Resources for a link).

While it's obviously most convenient to use an actual phone, there's something to be said for running an emulator. Most noticeably, the emulator can be downloaded right away, for free. The OpenMoko wiki (see Resources for a link) has additional information on configuring an emulator. This tutorial was developed using both the emulator and an actual phone, but not every function can be tested on the emulator.

Set up the phone

If you do have a Neo 1973, you should flash it to a current version of the OpenMoko software. Rather than duplicate here the excellent instructions from the OpenMoko wiki, look for the link to "Flashing openmoko" in Resources. Once your phone is set up, you can turn it on now or go ahead and set up an emulator as well.

Booting for the first time

If you're using the emulator, you won't be able to see this screen for a while, but here is what you're looking for:

Figure 1. OpenMoko splash screen

Section 3. Getting your environment together

Collect your tools

You need to have a number of tools handy to run OpenMoko; different Linux distributions have some of them installed already, but rarely all of them, so be prepared to fetch a few things. Perhaps most importantly, you must have gcc 3.4—not gcc 4.anything—to build and run QEMU. (This is almost certainly a bug, somewhere, but it hasn't been fixed in some time.) You also need subversion, various documentation-related tools such as texinfo, development headers for ncurses, a compression library (zlib or libz), OpenSSL, and GTK++. You may need additional things based on your system. The MokoMakefile wiki page has more information on a few systems.

Download the MokoMakefile

Make a new directory named whatever you want, and put it wherever you want on your system. I named mine "om" and put it in my home directory. Change to that directory and download the MokoMakefile.

Prepare for the build

The MokoMakefile can build everything you need, from scratch. This will take some time, so be prepared to wander off and do something else for a while. If you have a multicore system, you can save time by creating a file named build/conf/local.conf (create the build/conf subdirectory of your working directory first) containing two lines specifying a parallel build:

```
PARALLEL_MAKE = "-j 4"
BB_NUMBER_THREADS = "4"
```

The number 4, in this case, was calculated for a dual-core system; some people recommend running one more thread than your system has cores, and some recommend two threads per core. Four threads will work reasonably on a dual-core system.

Run the build

The first thing to do is run `make setup` for initial setup and configuration. Then run `make openmoko-devel-image`, and watch as a complete OpenMoko environment is built—but you may not want to watch the whole thing, as it can easily take five hours or longer. If you plan to use the QEMU emulator, build it now as well, using `make build-qemu`.

Section 4. The directory tree, as built

What's all this stuff?

When you typed `make setup`, you were probably in a directory containing only a makefile, or possibly a makefile and a single configuration file. Now, you should have directories with names like "bitbake," "images," and "sources." These are the working structure of an OpenEmbedded build environment.

The bitbake directory

Bitbake (the tool used to actually build target binaries and so on) has its own directory containing tools, documentation, and more. This directory holds the bitbake distribution; don't mess with it unless you know exactly what you're doing.

The build directory

The build directory holds configuration files (build/conf), QEMU stuff (build/qemu), and a temporary directory used for intermediate states. The temporary directory (build/tmp) holds its own selection of subdirectories; cache, cross, deploy, rootfs, staging, stamps, and work. When you're building, whether for the host or target environment, files end up in here. Of particular interest are the subdirectories of fic-gta01-angstrom-linux-gnueabi, which correspond to the Neo 1973 phone. In general, you will poke around in here only when debugging the build; you may want to pay special attention to the stamps directory, which indicates when each task was last performed.

The openembedded directory

This directory contains the OpenEmbedded distribution, including all of the packages available. The openembedded/packages directory is where new packages will be added—you'll need that a little later on. Mostly, though, this is internal architecture for OpenEmbedded, and you don't need to do much to it.

The openmoko directory

The openmoko directory contains additional files and documentation specific to OpenMoko, not just related to OpenEmbedded. Once again, you shouldn't have to interact with this too much directly.

The patches directory

As of this writing, this directory contains a tree structure providing for patches against openmoko, bitbake, or openembedded, but none are in it to begin with.

The sources directory

The sources directory is where source files are downloaded to for use in building the target filesystem. It holds actual distribution tarballs, as well as lock files and checksums. The sources/svn subdirectory holds Subversion checkouts for things that were downloaded via Subversion rather than as tarballs.

The stamps directory

Another collection of timestamps, this is used by MokoMakefile to check which components have been updated recently.

Section 5. Building a simple app

Welcome to OpenEmbedded

Since OpenMoko is built using OpenEmbedded, the right way to build a test application is to use OpenEmbedded. To keep things nicely focused, start with a command-line application to verify that you have the tools working correctly.

Make a package directory

In the previous section, you learned about the openembedded/packages subdirectory, which contains existing packages. Now it's time to add your own. It's common enough to name the sample program hello, so start with that; make a directory called "hello" in openembedded/packages, and make a subdirectory of that named "files." You really need just two files—a BitBake recipe and a source file for your program.

Hello, sample program!

A sample program doesn't need much; just some kind of output so you can verify that it works. Here's mine, saved as hello/files/hello.c:

Listing 1. A sample program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   puts("Help!  I'm trapped in a sample program factory.");
   return 0;
}
```

A recipe for baking

So now to the guts of it: a bitbake recipe. Luckily, a great deal of work has gone into making these as easy as possible. Under the hood, bitbake recipes resemble very specialized shell scripts (or perhaps more accurately, python scripts containing shell commands); here's a sample one, corresponding to the program above; I named it hello_0.0.bb:

Listing 2. A sample bitbake recipe

```
DESCRIPTION = "Tutorial example"
PR = "r0"
SRC_URI = "file://hello.c"
do_compile() {
        ${CC} ${CFLAGS} ${LDFLAGS} ${WORKDIR}/hello.c -o hello
}
do_install() {
        install -m 0755 -d ${D}${bindir}
        install -m 0755 ${S}/hello ${D}${bindir}
}
```

Variable assignments

The recipe starts with setting three variables. The description is straightforward enough. PR sets the revision—I'll explain that in a moment. SRC_URI should be a URI pointing to a file. You'll note that there is no third slash in the file:// URI used here—that's because bitbake is clever enough to look in the files directory automatically. If you provided a third slash, you'd end up asking for a file named hello.c in the root directory of your local drive.

Revisions

The trickiest part is the shortest part: the PR variable, which is used to indicate a revision of a project. If you make other changes to your file and don't bump the PR variable, bitbake may not realize you wanted it updated. Start with r0 and increment freely. As an example, even if a build fails (say, due to a typo in a source file), updating the file won't cause a new version to be copied in for the build unless you've changed PR.

do_this, do_that

The `do_compile()` function, if present, gives instructions for compiling your program. These instructions can be complicated or simple; in this case, they are quite simple. Note the large variety of predefined variables you can use to get correct build settings automatically. Similarly, `do_install()` is run, after your program is built, to copy files into appropriate positions in the target root filesystem. Be careful with modes; the sample bitbake file I started with had 0644 (owner read/write, group, and other read) as the mode for the target binary, but without execute permission, it's not so useful.

Adding your package

To add your package, go back to the local.conf file and add the following line:

```
DISTRO_EXTRA_RDEPENDS += "hello"
```

Note that the variable's name ends with RDEPENDS, not DEPENDS. This line causes bitbake to add the hello package (described by the bitbake file above) to the distribution.

Rebuilding

To rebuild with the new package added, run `make update` followed by `make openmoko-devel-image`. When you're done, you can use your new image, whether on an actual phone or in QEMU. (For QEMU, use `make flash-qemu-local` to "flash" the virtual phone and `make run-qemu-snapshot` to run it.) Running `hello` in a terminal should print an informative message.

So much for doing a trivial sample program. Now what?

Section 6. A minimal GTK+ application

Why GTK+

A lot of users, especially those who have done other small embedded systems, want to know why they can't use Qt. Well, you can, but you shouldn't. GTK+ and Qt are both generally built as shared libraries; this means that every GTK+ application on your system is using the same copy in memory of the compiled GTK+ code, and every Qt application uses the same copy of Qt. All of the standard applications provided with the system use GTK+, so your application using GTK+ is taking advantage of that code, which has already been loaded. A Qt application would have much higher overhead in this environment. You might compare this to the choice a country makes between driving on the left or the right side of the road; although there are arguments for either, they are all completely swamped by the importance of everyone on a given road making the *same* decision.

A minimal GTK+ program

I'll start with a minimal program, one that opens a window and does nothing. You don't even need a "quit" option, as phones all have a nice convention for closing applications—hit the power button. (In fact, this will be a bit inconvenient on the emulator, because the USB keyboard driver steals the space events.)

Here's the minimal program:

Listing 3. A minimal GTK program

```
#include <gtk/gtk.h>

int
main(int argc, char *argv[]) {
        GtkWidget *window;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_widget_show(window);

        gtk_main();

        return 0;
}
```

Updating the bitbake recipe

Building a bitbake recipe gets a little more complicated now. The first thing to do is specify that the application depends on GTK+. That's done with a line reading `DEPENDS = "gtk+"`. You also need to update the revision, or bitbake won't notice that you changed anything—merely changing the source doesn't count! The only other change needed is to the compiler line, which gets to be sort of long. You can't just link against libgtk—you have to get all the include directories right, and each of the packages involved has a separate include directory. The final `do_compile()` rule looks like this:

Listing 4. The complete compiler command line

```
do_compile() {
   ${CC} ${CFLAGS} ${LDFLAGS} -Wl,-O1 -g -o hello ${WORKDIR}/hello.c \
   -I ${STAGING_INCDIR}/pango-1.0 -I ${STAGING_INCDIR}/gtk-2.0 \
   -I ${STAGING_INCDIR}/cairo -I ${STAGING_LIBDIR}/gtk-2.0/include \
   -I ${STAGING_INCDIR}/glib-2.0 -I ${STAGING_INCDIR}/glib-2.0/glib \
   -I ${STAGING_INCDIR}/atk-1.0 -lgtk-x11-2.0
}
```

In fact, all this does is include a whole lot of include directories, and a single library. You don't have to specify a library path; the GTK+ libraries are in the default library location (/usr/lib) on the target, so the defaults work fine.

Why so many include directories?

For a program that only includes <`gtk/gtk.h`>, needing to specify seven additional include directories might seem excessive. There are two issues here. The first is that the GTK+ main header includes a number of other headers, some of which in turn include still more headers. The second is that, like some other systems, the OpenMoko build environment puts each package's include tree in its own subdirectory to reduce the possibility of clashes. The reference `<gtk/gtk.h>` refers, in this case, to /usr/include/gtk-2.0/gtk/gtk.h, but if you had more than one version of GTK+ installed, changing the include path would change which header you got, without requiring you to update large numbers of source files.

Running and testing again

Once again, build the application using `make openmoko-devel-image`. You should see that the hello application gets rebuilt. Now, put it on the phone using `make flash-qemu-local` if you're using the emulator. If you're using the phone, copy the file over however you please. Run your application and you'll get a grey screen that doesn't do anything. That's good! The goal here was not to get the application doing anything, but to verify that the compile and link options are right for a minimal GTK+ app. Now, let's add some functionality.

Section 7. System information

A system information applet

Browsing the OpenMoko site, I found a design document for a stylus-based system information applet. Looks like fun! It's fairly easy to write a program that fetches and displays a little bit of information about the system, and such a program sounds useful.

Table layout

I'm used to table layouts, so I'll start with one here. A GTK+ table layout lets you divide a window into a grid and then put objects into cells of the grid. The interesting API chunk is `gtk_table_attach()`, which has the following prototype:

Listing 5. The full prototype for gtk_table_attach

```
GtkWidget* gtk_table_attach(GtkTable* table, GtkWidget* child,
   guint left_side, guint right_side,
   guint top_side, guint bottom_side,
   GtkAttachOptions xoptions, GtkAttachOptions yoptions,
   guint xpadding, guint ypadding);
```

In the sample program, I use the same line breaks used in this prototype each time the function is called to make it a bit easier to see which arguments are what. For now, you're going to have a very simple table; the top cell will give the kernel version, and the bottom cell will have a Quit button (added in case you're running on the emulator).

Building and filling in the table

The first version of the program is pretty simple. All it has to do is create two labels and a button, put them in the table, and display them. Easy, right? First, you have to create a table, and insert it into the window:

Listing 6. Creating a table

```
table = gtk_table_new(8, 3, TRUE);
gtk_container_add(GTK_CONTAINER(window), table);
gtk_container_set_border_width(GTK_CONTAINER(table), 10);
```

The table is 8 rows, and 3 columns, with each cell the same size as every other cell (that's the TRUE argument to `gtk_table_new()`; if it had been FALSE, they could differ).

Adding a Quit button

Here's the code for the Quit button I added to make it easier to close apps:

Listing 7. Populating a table

```
button = gtk_button_new_with_label("Quit");
gtk_table_attach(GTK_TABLE(table), button,
   1, 2,
   7, 8,
   GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
   0, 0);
```

This shows a Quit button, but what good does that do? Nothing will happen when it's clicked.

Callback functions

A couple of callback functions are needed to support closing the window cleanly. The first, `delete_event()`, is used to indicate that someone would like to terminate the application; a FALSE return value indicates that the application should be destroyed, and a TRUE return value indicates that nothing should be done. The second, `destroy()`, simply exits. Note that these names are not mandated at all; however, they were used in the GTK+ tutorial code, making them easier for future readers to recognize and understand.

Listing 8. Quitting a GTK app

```
static gboolean
delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
        return FALSE;
}

static gboolean
destroy(void)
{
        gtk_main_quit();
}
```

Hooking up the wires

Now that you have both a button and a function for closing the application, it makes sense to hook them up by registering the signals to the callback functions. Here's the code to set up both the clicking of the button and the handling of regular window close events:

Listing 9. Connecting the button to its callbacks

```


g_signal_connect(G_OBJECT(button), "clicked",
                 G_CALLBACK(destroy), NULL);
g_signal_connect(G_OBJECT(window), "delete_event",
                 G_CALLBACK(delete_event), NULL);
g_signal_connect(G_OBJECT(window), "destroy",
                 G_CALLBACK(destroy), NULL);
```

And now, some data

Start by just printing the kernel version. You may be wondering how one obtains the kernel version. Why, `uname()` of course! This code is simple to write:

Listing 10. Reading system version information

```


struct utsname u;
[...]
uname(&u);
label = gtk_label_new(u.release);
```

The `struct utsname` is filled in with C strings denoting the characteristics of the current environment. The "release" string usually consists mainly of the kernel version number. That obtains the data, but how to display it?

Adding labels to the table

Listing 11 shows the code used to insert labels (which are really just text fields for our purposes) into the table. The first label, in the leftmost column, is simply a caption; the second, occupying the center and rightmost columns, holds the release value obtained via `uname()`.

Listing 11. Displaying a label

```
label = gtk_label_new("Kernel version:");
gtk_table_attach(GTK_TABLE(table), label,
   0, 1,
   0, 1,
   GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
   0, 0);

uname(&u);
label = gtk_label_new(u.release);
gtk_table_attach(GTK_TABLE(table), label,
   1, 3,
   0, 1,
   GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
   0, 0);
```

It may seem odd to overwrite the "label" variable with the second call to `gtk_label_new()`, but the previous allocated object has already been put in the table, so it won't get lost.

Bringing it all together

The one other change is to replace `gtk_widget_show()` with `gtk_widget_show_all()`, as the name suggests, `gtk_widget_show_all()` shows not only the window, but everything in it (in this case, a table, two labels, and a button). The resulting code, all together, looks like this:

Listing 12. A complete, if minimal, program

```
#include <gtk/gtk.h>
#include <sys/utsname.h>

static gboolean
delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
   return FALSE;
}

static gboolean
destroy(void)
{
   gtk_main_quit();
}

int
main(int argc, char *argv[]) {
   GtkWidget *window;
   GtkWidget *table;
   GtkWidget *button;
   GtkWidget *label;
   struct utsname u;

   gtk_init(&argc, &argv);

   window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
   g_signal_connect(G_OBJECT(window), "delete_event",
                    G_CALLBACK(delete_event), NULL);
   g_signal_connect(G_OBJECT(window), "destroy",
                    G_CALLBACK(destroy), NULL);
   gtk_window_set_title(GTK_WINDOW(window), "Hello, System!");
   table = gtk_table_new(6, 3, TRUE);
   gtk_container_add(GTK_CONTAINER(window), table);
   gtk_container_set_border_width(GTK_CONTAINER(table), 10);
   button = gtk_button_new_with_label("Quit");
   gtk_table_attach(GTK_TABLE(table), button,
      1, 2,
      7, 8,
      GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
      0, 0);
   g_signal_connect(G_OBJECT(button), "clicked",
                        G_CALLBACK(destroy), NULL);

   /* uname */
   label = gtk_label_new("Kernel version:");
   gtk_table_attach(GTK_TABLE(table), label,
      0, 1,
      0, 1,
      GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
      0, 0);

   uname(&u);
   label = gtk_label_new(u.release);
   gtk_table_attach(GTK_TABLE(table), label,
      1, 3,
      0, 1,
      GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
      0, 0);

   gtk_widget_show_all(window);

   gtk_main();

   return 0;
}
```

Section 8. Adding more features

Where to next?

Having developed a basic interface allowing a quick readout of data, the next question is, what other data might be useful? One thing is available disk space. There are a number of ways to query this.

Querying disk space

You can get available disk space using the `statfs()` system call. This function yields a string containing the current usage of the disk:

Listing 13. Finding free disk space

```
static char *
diskfree(void)
{
   static char capacity[16];
   struct statfs buf;
   if (statfs("/", &buf) < 0) {
      strcpy(capacity, "unavailable");
   } else {
      double used = 1.0 - ((double) buf.f_bavail / buf.f_blocks);
      sprintf(capacity, "%.1f%%", (used * 100));
   }
   return capacity;
}
```

The `statfs()` system call, much like `stat()`, yields information about a target path; however, it shows filesystem statistics rather than file statistics. As a quirk, although people are used to seeing capacity measured in percentage used, `statfs()` tells you only how much is currently available. This code calculates the amount used (as a range from 0 to 1), then converts it to a percentage. Note that the error-checking requires the addition of the `<string.h>` header, used for `strcpy()`.

Plugging in the information

By now, the procedure is fairly familiar:

Listing 14. More labels

```

/* disk space */
label = gtk_label_new("Disk usage:");
gtk_table_attach(GTK_TABLE(table), label,
    0, 1,
    1, 2,
    GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
    0, 0);

label = gtk_label_new(diskfree());
gtk_table_attach(GTK_TABLE(table), label,
    1, 3,
    1, 2,
    GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
    0, 0);
```

In fact, this kind of thing starts looking like a pattern that could be exploited. The only real change here is the row number and the two strings.

Refactoring time

To generalize a little, you might want to make a function to simply put one of these items in the display table. The essence of effective programming is creativity; I creatively named the function `label_in_table()`. What arguments does it need? It needs the table, the row, the descriptive label (which goes on the left), and the text to put in it (which goes on the right). So:

Listing 15. Extracted commonalities

```

static void
label_in_table(GtkWidget *table, int row, char *name, char *value)
{
   GtkWidget *label;
```

```
    label = gtk_label_new(name);
    gtk_table_attach(GTK_TABLE(table), label,
       0, 1,
       row, row + 1,
       GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
       0, 0);

    label = gtk_label_new(value);
    gtk_table_attach(GTK_TABLE(table), label,
       1, 3,
       row, row + 1,
       GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
       0, 0);
}
```

Calling label_in_table()

Next, let's simplify the actual main function. This is a lot easier to follow now than it was above:

Listing 16. Revised and simpler code

```
/* uname */
uname(&u);
label_in_table(table, 0, "Kernel version:", u.release);

/* disk space */
label_in_table(table, 1, "Disk usage:", diskfree());
```

This example points out a couple of things. One is that `table` might be a reasonable candidate for a global variable; it is, after all, the only table like that. Another is that you might think it'd make sense to use a static or global to keep track of row numbers—and it would, except that a likely future update might include replacing the contents of an existing row, so being able to specify a target row is probably a good feature.

Memory information

The next stop is a bit of memory information. The `sysinfo()` call is the starting point for this. Since the `sysinfo()` call yields more than one useful piece of information, the code to use its output will be slightly more elaborate—but only slightly. Much like `uname()` or `statfs()`, `sysinfo()` extracts data into a structure (`struct sysinfo`).

Memory consumption is easy enough to extract from this:

Listing 17. Finding memory usage

```
sprintf(buffer, "%.1f%%",
    (100 * (1.0 - ((double) si.freeram / si.totalram))));
label_in_table(table, 2, "Memory usage:", buffer);
```

System load

System load ought to be easy; the one-, five-, and fifteen-minute load averages are stored in the `struct sysinfo`, as unsigned longs. However, load average is traditionally exported to the user as the average length of the run queue, generally a floating point number. A load average of 1.00 indicates a system on the low end of a full load. So how do you convert the unsigned long values to the more familiar values? Well, it's not clearly mentioned in the `sysinfo()` man page, but the secret is a macro called `SI_LOAD_SHIFT`, which holds the scaling factor; in fact, the unsigned longs are being treated as fixed-point fractional values. Most often, `SI_LOAD_SHIFT` is 16, meaning that a load average of 1.0 is represented as 65,536.

This is the kind of calculation that calls for a helper function:

Listing 18. Getting the load average

```
static double
scaled_load(long int unscaled) {
        return (double) unscaled / (1 << SI_LOAD_SHIFT);
}
```

The resulting values can be printed the same way other values have been displayed to the user:

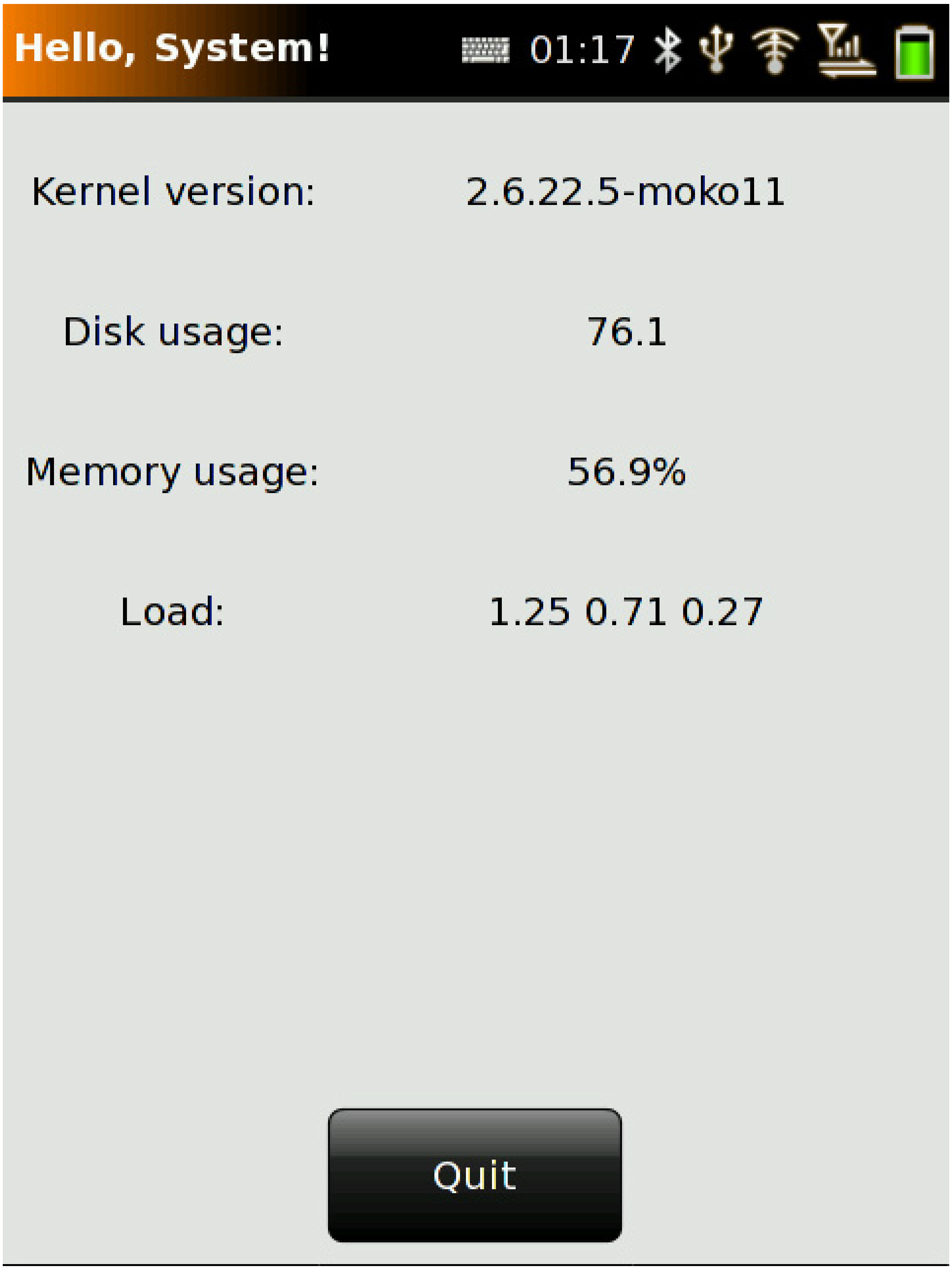
Listing 19. Displaying the load average

```
sprintf(buffer, "%.2f %.2f %.2f",
    scaled_load(si.loads[0]), scaled_load(si.loads[1]),
    scaled_load(si.loads[2]));
label_in_table(table, 3, "Load:", buffer);
```

And, you're done...for now

Time to look at the finished program. Here's a screen shot:

Figure 2. The system information applet



Kernel version: 2.6.22.5-moko11

Disk usage: 76.1

Memory usage: 56.9%

Load: 1.25 0.71 0.27

Section 9. Future directions

Now what?

This tutorial has introduced the OpenMoko development environment and given you an admittedly small start towards a basic system status applet. The next question is where you want to go with this. If you don't have a supported phone yet, getting one might be a good idea. I've focused on tasks that were portable to the emulator, but the phone has a number of additional features.

Networking

The early model Neo 1973 phones don't have WiFi hardware, but later phones are likely to. On the other hand, phone networks can also be used to exchange data. Configuring the network is likely to be handled by the provided software, though, and Linux networking is delightfully consistent.

GPS

The emulator doesn't provide an emulated GPS chip, but the Neo 1973 has a GPS unit built in. This offers a great deal of potential for interesting applications; for instance, your phone could send occasional packets to a server you run at home telling the server where it is. Very useful if you lose your phone a lot, and also sort of fun. You could also use one of the publicly available mapping APIs to build a "where have I been today" application.

Actual telephony

Strangely, the first dozen or so ideas I had for things to develop under OpenMoko had no connection at all to the telephone. What about an application to record phone calls? You'd want to be sure of legalities in your jurisdiction, as some areas don't allow this without consent of all parties, but it seems like it ought to be possible. This is a case where something that would be impossible on most other phones is made possible by the OpenMoko environment—normally, you could write your own applications, but they couldn't interact with the telephony applications. In the land of OpenMoko, though, the software is all provided in source.

The futility of prediction

When a new platform comes along, it can be very hard to guess what people will want to develop for it. Closed-source models, especially those with high costs of entry, have tended to see mostly well-understood business applications that meet already understood needs. Opening up development can produce real surprises, though, and that's one of the things that makes OpenMoko so interesting. The greatest potential may be, not the improvement of applications we already know are useful on a phone, but the applications no one thought of before. Now that you know enough to get started, go and write them!

Downloads

Description	Name	Size	Download method
Sample code for this tutorial	hello.tgz	1.5KB	HTTP

Information about download methods

Resources

Learn

- The OpenMoko Wiki is an ideal starting point for learning more about OpenMoko development. You'll also find more about:
    - Neo 1973
    - MokoMakefile
    - OpenMoko under QEMU
    - Flashing openmoko
- Read about OpenMoko on Wikipedia.
- Visit the OpenEmbedded site for complete information on this cross-compilation development environment.
- Read "System emulation with QEMU" (developerWorks, September 2007) for an introduction to QEMU.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Download the MokoMakefile.
- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

Peter Seebach
Peter Seebach has been using computers for years and is gradually becoming acclimated. He still doesn't know why mice need to be cleaned so often, though.

Trademarks