

Linux on board: Developing for the Nokia N810

Put the alarm API to work

Skill Level: Intermediate

[Peter Seebach \(developerworks@seeb.net\)](mailto:developerworks@seeb.net)

Senior Engineer

Wind River Systems

20 Aug 2008

The Nokia N810 alarm interface allows developers to efficiently and easily set alarms programmatically. Peter Seebach illustrates how a small command-line program can hook into this API and make good use of it.

The most serious limitation of the Nokia 770 as a potential PDA replacement was the lack of any way to set alarms that could wake the device. A much-improved alarm interface was introduced on the N800 and is still present on the newer N810. In this article, I take a look at the C language API to the alarm interface, and suggest an interface to make this available to shell scripts or programs in other languages.

First, a brief overview of the N810 is in order. The N810 is an embedded handheld system with an 800x480 display. It has Bluetooth, wireless, and USB connections. The underlying kernel is a 2.6.21 Linux® kernel, adapted for the hardware.

The N810 is fairly similar to the previous N800. New features include GPS and a built-in keyboard. The only way in which the N810 might seem less flexible than the N800 is that it provides only a single available MMC/SD card slot. The N800 provides two slots for user-provided MMC/SD cards, both full size. The N810 comes with a hard-wired 2GB card in the "internal" slot and a miniSD slot for removable media. (This can be quite frustrating if, like me, you've accumulated stacks of SD media over a few years of playing with embedded systems.)

More of Peter's articles on developerWorks

- [Linux on board: Linux powers Nokia 770](#)

- [Linux on board: Developing for the Nokia N800](#)
- [Linux on board: Accessing the Nokia N800 camera](#)
- [Linux on board: Auto-uploading Nokia N800 photos](#)
- [All of Peter's *Linux on board* articles](#)
- [All of Peter's articles on developerWorks](#)

The development environment for the N810 is essentially the same as that described for the N800 (see my [other articles on the 770 and N800](#)). There have been updates to Scratchbox and the maemo environment, but the essential process remains the same, and the Scratchbox and SDK install remain quick and easy. There are two major changes that are likely to affect a regular developer. The first is that xterm is installed out of the box; this is a big improvement. The second is that, when you install the openssh-server package, you are prompted to set a new root password. This is a big step up from the previous behavior: a default root password of "rootme". Obviously, you should pick some other password.

The alarm API

The alarm API was introduced in the maemo 3.0 release last year. It presents a set of calls to interact with the alarm daemon, which provides alarm services. You should prefer using this interface to trying to write your own, and you should definitely stay away from writing your own timer code in an environment like the N810. Power management on embedded hardware is an advanced topic, and it is easy to get it wrong; instead, just hand the job off to the specialized code provided.

Even if your alarm code is written elegantly and well, there is a compelling advantage to a centralized service. Imagine that you have written a perfect alarm interface, which has to wake the system only rarely, perhaps once every five minutes or so. This will have almost no impact on battery life. Now, imagine that other people, just as skilled as yourself, write similar interfaces. With several of them installed, the system must wake up several times every five minutes. Worse, users tend to pick different intervals. So, if you have one thing that wakes every three minutes, and one every five, and one every seven, you end up with the worst of all worlds: even though no individual alarm goes off more than once every three minutes, you actually have an alarm about once every two—and worse, pairs of alarms going off may involve separate wake and sleep cycles. So use the standard API.

The alarm API can do a number of things. In essence, it can configure the behavior of a scheduled notification with a great deal of flexibility. An alarm event may or may not involve a display to the user. The three primary things alarm tasks can do are display messages, run programs, or send messages over D-Bus to other

applications. You might not think of all of these as alarms, but they share substantial functional overlap with the simple reminder messages that a calendar application could provide. In short, if you are implementing enough alarm functionality to support calendar applications and the like, it is better to generalize a little further and handle everything.

Alarms are stored as XML in `/var/lib/alarmd/alarm_queue.xml`. This file is moderately human-readable, but does make liberal use of magic numbers to encode flags. You can get a lot of insight into the components of alarm event structures by creating events using other applications, then reading the XML stored in this file.

Why use getopt?

For some reason, people love to write their own argument parsers. (I can hardly claim immunity, having written a very full-featured replacement for `getopt()` once.) However, in a UNIX®-like environment, nine times out of ten, you should just use `getopt()`. It offers you well-understood and familiar semantics, and will do what users expect. For most programs, it has plenty of capacity to express a reasonable variety of options. Furthermore, it simplifies code and eliminates bugs. Time and time again, I've found that programs with their own argument parsers had buggy ones. I wrote mine in 1997, and I think it still has bugs.

Whether you're programming in shell, C, or some other language, find the `getopt`-like functionality and use it. (In shell, the POSIX answer is the shell builtin `getopts`, which is better integrated with the shell than the `getopt` command.)

A simple reminder

For brevity, I'm skipping over the D-Bus part of the alarm interface, and looking at the message and code execution features, simply because they don't require the development of the D-Bus code to handle the incoming notifications. It would be nice to be able to create simple event reminders from the command line. With that in mind, it would be nice to be able to create arbitrary alarm events from a command-line program, which would execute programs or display messages.

The alarm system is extremely flexible. An alarm has an initial time, a setting for how often it should recur (only counts in minutes are possible), and a number of times to recur. This can't handle quite everything; for instance, it can't handle a weekly meeting that is at the same time every week (because Daylight Savings Time will throw it off). You can specify a sound to play for the alarm, a picture to draw, and a message.

There are likewise a number of flags provided to control the behavior of alarms. These flags are simple boolean flags; if you do not specify a flag, you have implicitly specified its opposite. A collection of individual flags, plus a small collection of values to set, is a wonderful match for `getopt()`, but first we need to think about what the

flags would be, and where we'd store them. Because the flags correspond precisely to the alarm structure, I started there.

To begin with, we need an `alarm_event_t` structure, zeroed out. One of the maemo sample programs zeroes it out using `memset()`, but this is pedantically incorrect; there is no guarantee that using `memset` to fill something with zeroes will produce null pointers (although it happens to on this system). The standard taketh away, but the standard also giveth: you can use a single zero initializer to initialize the first field, and every following field is initialized as though by an explicit 0, which does guarantee null pointers:

Listing 1. Zeroing out our event

```
alarm_event_t event = { 0 };
```

Now, all we have to do is initialize the members accordingly. The most complicated, by far, will be the time. The type of people who would be comfortable just typing in the time they want a reminder in seconds since the epoch are probably not going to need a reminder program. So, a bit of thought is needed. The following hunk of code interprets three likely time formats:

Listing 2. And when would you like your wakeup call?

```
time_t
parse_time(char *s) {
    int Y, M, D, h, m;
    time_t secs = time(NULL);
    struct tm t = *(localtime(&secs));

    if (sscanf(s, "%d-%d-%d %d:%d", &Y, &M, &D, &h, &m) == 5) {
        if (Y < 100) {
            t.tm_year = (t.tm_year - (t.tm_year % 100)) + Y;
        } else {
            t.tm_year = Y - 1900;
        }
        t.tm_mon = M - 1;
        t.tm_mday = D;
        t.tm_hour = h;
        t.tm_min = m;
    } else if (sscanf(s, "%d:%d", &h, &m) == 2) {
        t.tm_hour = h;
        t.tm_min = m;
    } else {
        m = strtol(s, &s, 10);
        if (*s || !m) {
            usage("enter '[YYYY-MM-DD] hh:mm' or delay in
minutes.");
        }
        t.tm_min += m;
    }
    return mktime(&t);
}
```

This takes the most common standard date formats: ISO dates (including year), just

a time of day, or an offset in minutes. There's no support for 12-hour clocks, or for just entering month and day. The latter is due to the difficulty of guessing in what order the user will expect to enter them; the former is because I am lazy.

Handling most of the command-line arguments is simple using the standard `getopt()` function (see the [sidebar](#) for more detail). The `getopt()` routine processes the arguments (you must pass it `argc` and `argv`) according to a string of accepted option characters. Only single-character options are accepted; each character in the string represents an option that is known to the program. If an option's character is followed by a colon, it accepts an additional argument. For instance, the option string `ab:` indicates that the calling program knows two options: `-a` and `-b`, and that the next word after `-b` is a supplemental argument. Here's how the code looks:

Listing 3. A partial list of options

```
while ((o = getopt(argc, argv, "ABDIZc:C:i:nr:R:s:t:z:")) != -1) {
    switch (o) {
        case 'A':
            event.flags |= ALARM_EVENT_ACTDEAD;
            break;
        [...]
        case 'c':
            event.exec_name = strdup(optarg);
            break;
        [...]
        case '?':
        default:
            usage("unknown argument -%c.", optopt);
            break;
    }
}
```

If you're not familiar with `getopt()`, this may require some explanation. The return from `getopt()` is `-1` when there are no more options. Otherwise, it is the character of the next matched flag. If the flag takes an argument, the argument is stored in the global variable `optarg`. A question mark indicates an invalid option, in which case the character triggering it is stored in the variable `optopt`. In fact, the error message above is incorrect; it can also be generated if an option that requires an argument does not have one. However, that cannot happen for a valid call to this program, which requires additional arguments.

After `getopt()` has returned `-1`, the global variable `optind` holds the index of the first argument that was not part of the command-line options. There are two remaining things to parse: the time specifier, which is the first argument after the options, and the message, which is any remaining arguments. The message is optional; if the `-c` flag has been specified, no message is needed. Since this program can only display messages or run commands, it refuses to create an alarm that has neither a message nor a command.

Once the event structure is filled in, handing it to the alarm API is simple:

Listing 4. Wake me when it's over

```
cookie = alarm_event_add(&event);
if (cookie == 0) {
    die("got an error: %d", alarmd_get_error());
    exit(1);
}
```

That gets the basics in order; we can drop an alarm off and expect it to run. Unfortunately, there are a few quirks to deal with.

Quirks

I found these quirks in a little light testing. There may be more, but these four suggest some things you should watch out for while using the C API to the alarm daemon, or the alarm daemon itself.

Quirk #1: %s Alarm

The default title, if none is specified, seems to be "%s alarm"—probably a bug. This happens if you leave the `event.title` field null, or set it to an empty string.

Here's my solution: Set `event.title` to "Alarm!" before parsing user arguments. If the user does not specify a title, a safe title is provided. This prevents inconvenient accidents.

Quirk #2: Should the dialog be displayed?

There is a flag to not display a dialog for the event. Since the dialog makes sense only when there's a message for it, the program sets the "no dialog" flag when the message is empty. However, an empty message is considered an error unless there's a command to run, because an alarm that neither runs a command nor displays a message seems redundant. In fact, this restricts functionality some; a sneaky user might want to use the boot-on-alarm functionality to wake the system without doing anything else.

Listing 5. User interface logic

```
if (!event.exec_name && (argc - optind) < 2) {
    usage("if you do not specify a command, you must specify a
message.");
}
event.alarm_time = parse_time(argv[optind]);
event.message = parse_message(argv, optind + 1);
if (event.message[0] == '\0') {
    event.flags |= ALARM_EVENT_NO_DIALOG;
```

```
}

```

Quirk #3: Sound files

The sound argument is supposed to be the name of a sound file, but how many users have a lot of sound files handy? Well, all of them. There's a large collection of useful files in `/usr/share/sounds`. But specifying full paths annoys the user.

Here's my solution:

Listing 6. Find me a sound

```
case 's':
    if (*optarg == '/') {
        event.sound = strdup(optarg);
    } else {
        s = malloc(strlen(optarg) + 23);
        sprintf(s, "/usr/share/sounds/%s", optarg);
        if (access(s, R_OK) == 0) {
            event.sound = s;
            break;
        }
        sprintf(s, "/usr/share/sounds/%s.mp3", optarg);
        if (access(s, R_OK) == 0) {
            event.sound = s;
            break;
        }
        sprintf(s, "/usr/share/sounds/%s.wav", optarg);
        if (access(s, R_OK) == 0) {
            event.sound = s;
            break;
        }
        usage("can't find '%s' in /usr/share/sounds, try
absolute path.",
            optarg);
    }
    break;

```

This searches for sound files in the system default location, and checks the common (and supported) MP3 and WAV format suffixes. It makes no effort to find files in the current directory, because that would be making it too easy; users need to sweat a little. On a side note, the sound is played only if a dialog is displayed.

If you've used the Clock application to set alarms, you may be wondering how to control the rising volume of alarms. So am I. As far as I can tell, it is simply automatic; the alarm is played quietly at first, then gradually louder.

Quirk #4: Command execution

Several quirks are associated with command execution. The first is that if you have a dialog, execution happens only when the user closes the dialog, not when snoozing it. If there's no dialog, the execution happens immediately. This suggests that, if you really want something to happen for sure, you should separate it from any alarm

dialog you want to show the user; just make a second event.

The second quirk is that you cannot pass arbitrary shell commands in. The `exec_name` parameter in the C API is passed to the `g_shell_parse_argv()` function, not a full shell. In particular, this means that there is no parameter expansion or globbing.

If you want parameter expansion or globbing, you must invoke a shell explicitly. To that end, I added an option to do this for you:

Listing 7. Escape to the shell

```
case 'C':
    s = malloc(strlen(optarg) + 9);
    sprintf(s, "sh -c '%s'", optarg);
    optarg = s;
case 'c':
    event.exec_name = strdup(optarg);
    break;
```

The careful reader will realize that this may fail badly if the passed argument includes single quotes. Fixing this is an exercise in shell programming, though, not really part of the alarm API.

Conclusions

The alarm API is extremely flexible, and fairly well maintained. The C API part of the alarm daemon is simple to use and provides a good range of features. While this simplified interface can't do everything you could do in a larger application, it shows a great deal of flexibility; you could probably plug a day's errands and appointments into it quite easily. It wouldn't hurt to offer some way to manage existing alarm events.

Some applications for the N810 (and N800) do their own alarm management; this seems like a poor choice. If you're doing an application that needs to provide alarms, take the few minutes to learn the alarm API and use it. It will save you time, and integrate better with other programs that need to set alarms.

As with previous articles, thanks to the chatters in the #maemo IRC channel, who offered useful advice while I was researching this.

Resources

Learn

- The [online API reference for the alarm daemon](#) is reasonably comprehensive, though not always as detailed as you might like.
- The maemo page has a simple [howto on using the alarm interface](#), which talks about features this article doesn't cover, such as getting or deleting existing alarms.
- Peter's article "[Linux on board: Linux powers Nokia 770](#)" (developerWorks, November 2006) discusses the original Nokia 770. (Note that the model number was just 770, not N770.)
- See Peter's three-part series on N800 development:
 - "[Linux on board: Developing for the Nokia N800](#)" (developerWorks, November 2007)
 - "[Linux on board: Accessing the Nokia N800 camera](#)" (developerWorks, November 2007)
 - "[Linux on board: Auto-uploading Nokia N800 photos](#)" (developerWorks, December 2007)
- 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

- [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 spaces.

About the author

Peter Seebach

Peter Seebach does not like to leave the house without at least three Linux devices. This has gotten a lot less strenuous since the early 90s.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries.