NXT Sense - a Sensor Driver Framework for the GumstixNXT Platform

Emil Holmegaard, Mads Bahrt, Michael Bejer-Andersen
{emhol07, mabah01, mibej07}@student.sdu.dk

November 18, 2011

1 Introduction

This is the report evaluating the second mini project in the course RM-EMB4 taught in the fall of 2011 on the University of Southern Denmark. The objective of the second mini project is to explore writing Linux device drivers for embedded systems. The produced code can be found on:
`git://github.com/bejer/SDU-emb4-group1.git`

2 Design Choices

The assigned goal of the device driver mini project, was to develop a motor driver and a sensor driver. After considering the complexity of multiple drivers sharing the same resources we focused our effort at working with sensor drivers at the expense of not making a motor driver. We aimed to develop a framework, called NXT Sense, to take care of the interaction with the shared resources, such as the ADC, GPIO pins and the four sensor ports.

An overview of the NXT Sense, is depicted in figure 1. The rhomboids in the figure indicates sysfs entries, and the squares indicates driver modules. For the figure 1, there are three general hardware drivers specific for the Gumstix NXT board - Level shifter, ADC and voltage sensor.
The ADC driver module makes use of the SPI bus on the Gumstix board which is connected to the ADC on the Gumstix NXT board. The ADC driver has been developed, such that a device file for each used ADC channel is created. Currently five devices are created, one for each NXT port and one for the voltage sensor. The reason for creating a device for each channel is to provide easy debugging of the devices connected to the ADC.

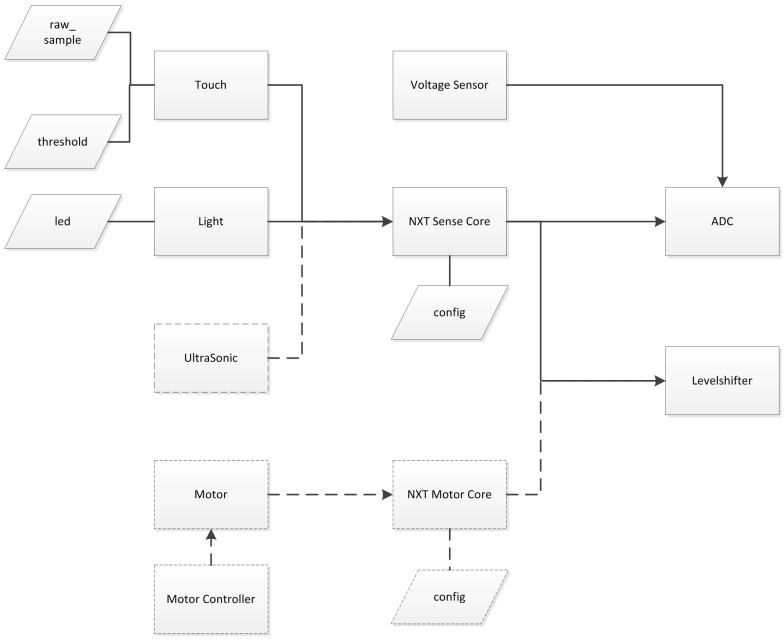

Figure 1: The overview of the NXT Sense framework. The boxes and connections with dotted lines are not implemented.

2.1 Dynamic Reconfiguration Through Sysfs Attributes

The module "NXT Sense core" controls which NXT ports there should be used, and which driver submodules to load for the specific ports. The NXT Sense core makes use of a sysfs file called "config", where the user specifies which kind of sensors are attached on the NXT ports. The config file consists of four numbers representing the four ports. At the moment there are three possible values for each port, 0 for port without a sensor, 1 for a touch sensor and 2 for a light sensor. When the config file has been updated, driver submodules are loaded and unloaded. This means that device files are created to reflect the config file. For example writing "1 2 0 0" to the config file will result in creation of a touch0 and a light1 device file, where 0 and 1 corresponds to the port which the sensor is attached. This approach has been chosen, such that code specific to the hardware has been placed in separate modules, so the sensor code could be reused for another board.
For the touch sensor, two sysfs files has been generated. One which provide the raw sample data of the sensor with read-only permissions. The other sysfs file, which is used for setting the threshold value, has read permissions for everybody and write permissions the owner. Normally the file would be owned by root.
For the light sensor, one sysfs file has been generated. This file is used to specify whether or not the led on the light sensor should be on or off, indicated by a 0 or 1 respectively.

By using sysfs entries it becomes easier and more transparent to set and get attributes related to the device. This could also have been achieved using ioctl, but is less convenient. Similarly it could also have been achieved by designing a small command language for the device file interface, but in the sysfs based solution the driver does not need to have a complicated parser, because each attribute has its own file.

2.2 Plans for Future Functionality

Figure 1 shows a driver for the Ultra Sonic sensor, but this driver has not been implemented, but the future implementation was considered during our process. I^2C uses two bidirectional lines, Serial Clock Line(SCL) and Serial Data Line(SDA). This means that the NXT Sense can reuse the functions for SCL on each port, and a function for SDA can be implemented in the same way. The Ultra Sonic sensor driver, would then make use of the function pointer for SCL and the new function pointer for the SDA.

3 Implementation Details

3.1 ADC Driver

The used SPI bus is connected to the ADC through two level shifters, where both are shared with other drivers and devices. Since the shared GPIOs can only be requested once through gpio_request, a wrapper around the GPIOs is needed. This is achieved by using a level shifter module, which keeps a reference count on the use of the GPIOs and only releases them when no driver is claiming their use of them.

3.2 NXT Sense Core Module

It has been chosen to provide one function per NXT port, for each of the ADC sampling and SCL pin manipulation functionalities. This is done to avoid having multiple submodules call the same function and having to deal with concurrency due to this. It is the ADC's responsibility of handling concurrency when using its sampling interface. Another goal of this choice is to minimise the need of having the submodules identify themselves when using the NXT Sense services. To avoid code duplication the sampling and SCL functions are being generated by a macro, an example of this can be seen in code snippet 1.

```
1  #define SAMPLE_FUNCTION(_port, _adc_channel)                          \
2    static int get_sample_##_port (int *data) {                         \
3      int status = 0;                                                   \
4      status = adc_sample_channel(_adc_channel, data);                  \
5                                                                        \
6      if (status != 0) {                                                \
7        printk(KERN_ERR DEVICE_NAME ": Some error happened while
             communicating with the ADC: %d\n", status);                 \
8      }                                                                 \
9                                                                        \
10     return status;                                                    \
11   }
12
13 /* applying the macro above - sample_function(port, corresponding
       adc_channel) */
14 SAMPLE_FUNCTION(0, 0)
15 SAMPLE_FUNCTION(1, 1)
16 SAMPLE_FUNCTION(2, 2)
17 SAMPLE_FUNCTION(3, 3)
```

Code snippet 1: Macro for generating sampling functions.

4 Design Alternatives

In this section we will discuss some of the alternatives we considered and their pros and cons.

4.1 ADC Polling Loop

As designed the ADC will only sample ports whenever there is an actual request for a value. Each request to the ADC is triggered by a read to the corresponding logical port in the ADC module.
An alternative to this might be to repeatedly poll all of the ADC ports, and store the latest value for each port in an internal data structure. Asynchronously from this a separate thread might emulate the existing interface by returning the latest value from this data structure, while other threads might issue software interrupts when certain conditions occur. In favor of our existing solution is that no power is wasted on polling when there is no use for it. When there is a low amount of polling requests each result will be more recent.
The alternative implementation based on a polling loop would be more predictable in terms of putting maximum upper and lower bounds on the time delay on each result.
A completely different approach would be to change fully to a publish/subscriber model where the application code is only triggered by changes in the hardware ports. Ideally such a decision should not be imposed on the user application by the driver, but instead the driver should provide both execution paradigms.

4.2 SPI Bus Message Request-Reply Interleaving

When communicating with the ADC, the reply is delayed one period from the request is sent. This is illustrated by the timing diagram in figure 2. This means, that if only requesting one port to be sampled, then one period of time (half of the total) will be wasted waiting for a reply in the next period. If, on the other hand, several ports were to be sampled, their requests and responses could be interleaved. To do this, the addresses of the ports to be sampled would need to be readily available. This could either be achieved by the "event loop ADC polling" scheme described above, or by having a queue of polling requests to be serviced. In the current implementation this interleaving has not been implemented, effectively wasting half the bandwidth of the SPI bus. The reason for this was the simplicity of implementing it in this manner, but this is a potential future improvement.

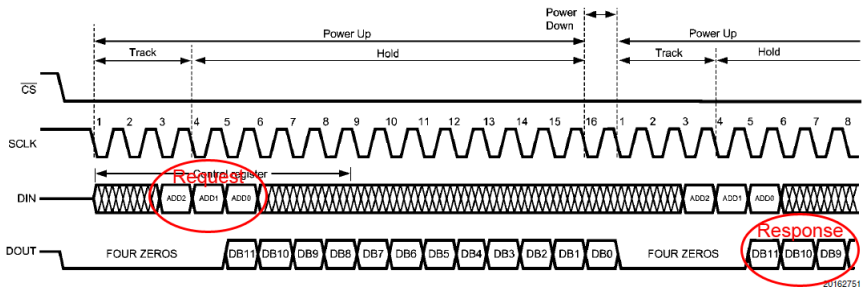

Figure 2: Timing diagram for A/D Converter used on Gumstix NXT board, Semiconductor [2011]

4.3 NXT Sense Configuration Interface

As described above the current syntax for indicating a sensor is to send a string in the form of "1 2 1 0" to the /sys/class/nxt_sense/nxt_sense/config file. This is not a very user friendly interface, but it is easy to parse in the driver. To alleviate this we suggest writing a user land application to generate this output based on a more user friendly interface. We considered including this functionality in the module, but decided that there was no good reason to keep this functionality inside the kernel.

4.4 Handling Concurrency

A certain amount of effort has already been spent on making the developed modules threadsafe. We think we have addressed the concurrency issues in "normal reading operations", but might still have some weaknesses if functionalities are accessed before the module has been fully loaded and initialised. Also, our level of locking sensitive spots in the code might be changed to a finer granularity, which might improve performance.

4.5 Dependencies Between Nxt Sense and the Sensor Submodules

Right now the NXT Sense framework compiles to one big kernel module, but it would be better to have the driver submodules be independent kernel modules, while still providing some reusable services in the NXT Sense core. It has been prioritized to minimize the static dependencies between NXT Sense core and driver submodules, so the task of separating them will be easier and is probably recommended to do. The static dependencies that exist now are the hooks to add and remove the sensor drivers to a given NXT port. These hooks could be dynamically given to NXT Sense once the sensor driver is loaded, by passing on function pointers for its add and removal functions, to NXT Sense.
Within the registration process the sensor driver modules could also provide a label, e.g. "touch" or "light". These labels could then be exported to a sysfs file maybe together with the corresponding number for the config file, allowing the user to see which sensor drivers are supported (currently loaded and registered with NXT Sense).

5 Experiments

We have performed a few latency tests on the device drivers on the Gumstix NXT board, running a Linux 2.6.39 kernel. The experiments were done without any load on the system, so the measured latency will be in the lower bound of the latency to be expected when using the drivers.
Table 1 shows the time it takes to obtain a sample through the char device interface of a touch sensor, starting from when the device file is opened and until it is released. The 10 measurements were done in a sequential order using the following command:

```
for i in ‘seq 1 10‘; do cat /dev/touch1; done
```

Open [μs]	**Release [μs]**	**Difference [μs]**
1692531103531616	1692531103533416	1800
1692531103538208	1692531103539886	1678
1692531103546081	1692531103547760	1679
1692531103553711	1692531103555420	1709
1692531103561370	1692531103563049	1679
1692531103568969	1692531103570648	1679
1692531103576233	1692531103577911	1678
1692531103583496	1692531103585174	1678
1692531103591064	1692531103593231	2167
1692531103599151	1692531103600830	1679
	Mean	1743
	Stdev	154

Table 1: Measurements of the time from opening the /dev/touch1 device file and until it is released again.

From table 1 it can be seen that it takes roughly 2ms to obtain a sample on a system without load. Which should be more than fast enough for normal Lego NXT usage.
The time it takes to obtain an ADC sample through the SPI subsystem has also been measured and can be seen in table 2. The start time is obtained just before the call to spi_sync() and end is obtained just after it. The time it takes to communicate with the ADC to get a sample value is roughly 0.2ms.

Start [μs]	**End [μs]**	**Difference [μs]**
1692532646862579	1692532646862762	183
1692532646870117	1692532646870269	152
1692532646876465	1692532646876617	152
1692532646884247	1692532646884430	183
1692532646891784	1692532646891937	153
1692532646899231	1692532646899414	183
1692532646906707	1692532646906891	184
1692532646914123	1692532646914306	183
1692532646921722	1692532646921875	153
1692532646929138	1692532646929321	183
	Mean	171
	Stdev	16

Table 2: Measurements of the time it takes to perform the spi_sync() call to obtain a new sample.

Given that the SPI communication is running with 3 MHz, the time it takes for the ADC to actually sample and output the value is $(1/3.000.000) \times 32 \approx 0.011$ms, see figure 2. This indicates that there is a relative big latency from measuring the analog value and until it arrives in userspace, although it should be fast enough for Lego NXT usage.
We also found the resolution of the timer used, to see how it relates to the measurements made in table 1 and 2. Code snippet 2 shows the steps taken to find the resolution.

```
1  #include <linux/time.h>
2
3  struct timeval res_start;
4  struct timeval res_end;
5  int timeval_counter = 0;
6
7  do_gettimeofday(&res_start);
8  do {
9    do_gettimeofday(&res_end);
10   ++timeval_counter;
11  } while (!timeval_compare(&res_end, &res_start));
```

Code snippet 2: Code used to find the resolution of the timer.

By invoking it 300 times it shows that the timeval_counter varies between 1 and 23 and the time difference is either 31μs or 30μs. Given this information it can be concluded that the above measurements are not affected by flooring[Cohen, 1995].

6 Test and Evaluation

This section consists of a subsection about how the code has been evaluated, and a subsection about the outcome of this mini project.

6.1 Code Review

The development of NXT Sense, has been done as pair programming, Wikipedia [2011]. This means the project group has obtained different roles, doing the writing of the code. No formal code review has been done, but are planned to take place within a few weeks.
When major design issues has arrived, the group had stopped, and discussed which options was available, and which solution would be the best choice for the overall solution.

6.2 Test

Throughout the process we have repeatedly performed manual tests of the developed functionality. For the sensors this means reading the current value of the sensor, manipulating the sensor and testing to see if the expected change in value has occurred. For the NXT Sense core if was tested to see if the submodules was loaded and unloaded properly. We considered trying to load and unload modules and submodules repeatedly to look for signs of memory leaks, but have not done this yet. We have also validated the declared dependencies between modules, such as the requirement that the ADC module cannot be unloaded before the NXT Sense core module.
The concurrency has been tested, by setting a module to sleep after obtaining the mutex of the module. By having the module sleeping, we have the time to invoke the module from user space, and verifying that the module was locked by another process. We have tested this, by calling the read function of the touch sensor, which will go to sleep before it outputs the value of the sensor. On another process we tried to change the threshold value of the specific touch sensor. The process of changing the threshold value, returned an error, saying "the resource is busy!".

6.3 NXT Sense Solution

A sensor framework, NXT Sense, has been developed. The NXT Sense is a complex framework, which is responsible for loading and unloading the right submodules, corresponding to a sysfs entry. The NXT Sense core collects the

code, which is common for the individual sensor types. By having such a framework, it should be easier to develop submodules for new sensor types, with a Lego NXT interface.
The NXT Sense facilitates porting the sensor submodules, to other hardware e.g. a different ADC or another NXT board.

References

Poul R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, 1995. ISBN 978-0-262-03225-4.

National Semiconductor. Adc128s022 8-channel, 50ksps to 200ksps, 12-bit a/d converter, 2011.

Wikipedia. Pair programming, 2011. URL `http://en.wikipedia.org/wiki/Pair_programming`.