Benchmarking the real-time Linux extension Xenomai in an embedded environment

Rik van der Kooij
6064965

August 12, 2012

Supervisor(s): drs. A. van Inge and drs. T.R. Walstra
Signed:
Abstract

Real-time extensions are available for Linux in the form of patches. When using these real-time extensions, knowledge of the performance is required to know if it will meet your requirements. It is more important to understand the behavior of the system and understand how the system handles different situations. This paper will be a report on performance measurements of the Linux real-time extension Xenomai in an embedded environment.

This is similar to research done the University of Aveiro where performance of multiple real-time operating systems was measured. Another similar study was done by Rep Invariant Systems, Inc. This study looked at the interrupt handling time of Linux and Linux with real-time extensions.

The benchmarks show that the maximum response time to external interrupts of Xenomai is at least twice as fast a standard Linux kernel. Xenomai also handles multiple incoming interrupts more predictable.
CHAPTER 1

Introduction

Embedded systems can be found in all kinds of places like automobiles, refrigerators and planes. The design of an embedded system depends on the needs of the system. For example an embedded system in an automobile or refrigerator, both measuring temperature, could use different hardware due to the difference in measurement frequency.

A subset of embedded systems are real-time embedded systems. In real-time systems tasks can have deadlines. The goal of a real-time system is to let every real-time task meet its deadline. Deadlines come in different flavours: hard, firm and soft \[7\]. Deadlines are considered hard when failing to meet one of the deadlines can have catastrophic results. An example is a pacemaker where failure means death of a patient. Missing soft deadlines will only decrease the performance of the systems. Example of this is a music player. If audio is not played at the correct time it may sound less pleasant, but the audio player is able to continue playing. Firm deadlines are a combination of hard and soft deadlines. A firm real-time system will not fail on one missed deadline, but is able to fail on a couple of missed deadlines.

Linux as general-purpose operating system is not suitable for real-time tasks. The Linux scheduler is optimized for average performance and tries to give every process a fair share of computing time \[8\]. For real-time tasks precise timing and predictable performance is more important than average performance. Another problem is the non-preemptive Linux kernel. A kernel is the center of an operating system. It is the layer between the hardware and the software. A non-preemptive kernel is able to hold the systems up for an unknown amount of time. Normally this will not be unending long, but real-time system itself could include kernel modules which will be able to shut down the system, from other more important activity, for a longer amount of time.

There are two basic approaches to modifying the standard Linux kernel into a real-time kernel \[2\]. The first uses a microkernel which runs the unchanged Linux kernel as a task. The microkernel takes control over the system for real-time processes and is responsible for scheduling real-time tasks, interrupt handling and scheduling Linux. With these changes it is possible to let a real-time task run on the CPU without the Linux kernel being able to interrupt the task. As the Linux kernel is run as a task it is preemptive at any time. With the second approach, changes are made the Linux kernel in order to be real-time. This include changes as real-time scheduling and adding preemption to the kernel.

In 1996 a real-time Linux called RTLinux was developed by a group at the New Mexico Institute of Mining and Technology in Soccorro. Early in the development it was clear that changing the kernel itself would be an enormous amount of work \[8\]. They therefore used the microkernel approach which added some code, but needed only small changes to the existing Linux kernel code. Changes had to be made to the Linux cli() routine. This routine would instead of disabling interrupts set a flag. Using this flag, the microkernel can upon catching a non-real-time interrupt decide whether to pass it to the Linux Kernel or wait for a reset of the

---

flag. The real-time tasks were non-preemptive and FIFO scheduled. Since then a lot of research has been done in preemptive real-time task scheduling with different priorities and it is in fact an NP-complete problem. In order to reduce paging delays, RTLinux will never page out real-time tasks.

Around the same time members of the Department of Aerospace Engineering, Politecnico di Milano envisioned a real-time hardware abstraction layer (HAL) onto which a real-time applications interface (RTAI) could be mounted. The Linux kernel was at that time not mature enough for this concept. It was only after the team had seen the design of RTLinux and changes came to the kernel with newer versions that RTAI was made. RTAI ended up very similar to RTLinux. The main difference is that RTLinux makes most of its changes to the kernel source files itself while RTAI limits the changes in kernel code by adding a hardware abstraction layer. The HAL was implemented with only 70 lines of code which makes it easy to maintain with new kernel versions. Since RTAI used the same HAL technique as used by RTLinux it ran into patent discussions. This resulted in abandoning HAL and the use of ADEOS. Adaptive Domains Environment for Operating Systems is in fact still a hardware abstraction layer between hardware and the running operating system. The difference is that ADEOS is able to run several kernels together.

Xenomai is another real-time extension for Linux which uses ADEOS. RTAI and Xenomai have been fused together between 2001 and 2005. This has made them very similar. ‘The major differences derive from the goals the projects aim for, and from their respective implementation. While RTAI is focused on lowest technically feasible latencies, Xenomai also considers clean extensibility (RTOS skins), portability, and maintainability as very important goals.’

In later versions of the Linux kernel changes have been made to make the kernel more real-time. The changes include an \(O(1)\) scheduler, being able to disable swapping, fixed priority for processes and most important a more preemptive kernel. The kernel has been made preemptive by protecting critical sections with spin locks instead of disabling interrupts. As of Linux kernel version 2.6 with the these changes it can be stated that the kernel is a soft real-time operating system.

Real-time systems are often used as control system. Many real-time tasks in control systems are periodic. An example is a periodic sensor reading which the system responds to. Other real-time tasks can be non periodic and start whenever an event occurs. In this paper we will be benchmarking the Linux real-time extension Xenomai within an embedded system. The embedded system will consist of an FPGA (Opal Kelly XEM6001) and a Linux minicomputer (NanosG20). The XEM6001 will be used as a digital signal processor. Sensor data which need further analysis are sent to the NanosG20. As modern FPGA include multiple processors sensor readings can be done completely parallel. The communication between the boards will on the other hand be done sequential because the NanosG20 has a single processor core. The main question of this paper will be answering is:

What do the benchmarks say about the performance of Xenomai?

In chapter 2 we will look at ways to build real-time Linux kernels for the NanosG20 and useful benchmarks for real-time systems. Chapter 3 will be used to look at the hardware characteristics of the NanosG20 for real-time systems and we create an user space C program to configure the XEM6001. In chapter 4 we will be performing benchmarks and discuss the results. In chapter 5 we discuss the results and possible future work.

---

4 Difference between Xenomai and RTAI by the Xenomai wiki page [http://www.xenomai.org/index.php/FAQs](http://www.xenomai.org/index.php/FAQs)
2.1 Creating real-time Linux kernels

Creating a real-time Linux kernel is similar to creating a standard Linux kernel. Creating a standard Linux kernel starts with downloading kernel source from for instance the Linux Kernel Archives\(^1\) and unpacking kernel sources with the Linux/Unix `\texttt{tar -xvj} \texttt{r /path/to/linuxsources}` command. After that the kernel can be configured with the Makefile. The command `\texttt{make menuconfig}` starts a user-interface where kernel options can be selected. Configuration includes selecting the hardware type, device drivers, file system support and more. Compiling the kernel and kernel modules with `\texttt{make}` is what comes next. We end up with a kernel image and loadable kernel modules.

A soft real-time Linux kernel is created by simply setting the `CONFIG_PREEMPT` configuration flag. The process of creating a hard real-time Linux kernel is for each real-time extension different, but it usually consists of patching the kernel and configuring real-time kernel options. Creating a hard real-time kernel with Xenomai is more than just patching. Xenomai uses a shell script to change the kernel code and includes user-space components. The shell script is run with the command `\texttt{scripts/prepare-kernel.sh --linux=/path/to/linux/sources \[--adeos=/path/to/adeos/patch\] \[--arch=<target-arch>\]}`. It should be used right after unpacking the source code. Xenomai includes user space components these components should also be compiled using `\texttt{make}`.

For the NanosG20 custom Linux source code is available from the Ledato, the company which created the board, website\(^2\). These kernel sources include default configuration for the NanosG20. We can use these settings by running the `\texttt{make ARCH=arm nanosg20_defconfig}` command before menuconfig.

A kernel created as described above is compiled for the hardware of the machine compiling the kernel. The kernel compilation for embedded boards is usually done on another machine, a host. A cross compiler will be needed in order to compile the kernel for the hardware of the NanosG20. The Linux and Xenomai support the use of cross compilers. This is done by adding extra arguments when using make. For the ARM NanosG20 board this changes the make and shells script commands explained above into: `\texttt{make ARCH=arm menuconfig}`, `\texttt{make ARCH=arm CROSS_COMPILE=arm-angstrom-linux-gnueabi-}` and `\texttt{scripts/prepare-kernel.sh --linux=/path/to/linux/sources \[--adeos=/path/to/adeos/patch\] \[--arch=ARM\]}`. The Xenomai user space components have a shell script for cross compilation configuration. For the NanosG20 the configuration command is `\texttt{.configure CC=/usr/local/angstrom/arm/bin/arm-angstrom-linux-gnueabi-gcc --host=arm-linux-gnueabi \--enable-arm-eabi \--enable-arm-mach=at91sam9 \--disable-arm-tsc}`. The default installation directory for the Xenomai user components is `/usr/xenomai`. The directory can be changed by adding DESDIR to the make command: `\texttt{make DESTDIR=/new/destination/dir}`.

\(^1\)The Linux Kernel Archives, \url{http://www.kernel.org}
\(^2\)Custom Linux kernel (version 2.6.35.7) source code can be downloaded from: \url{http://www.ledato.de/}
2.2 Building embedded Linux systems with Buildroot

We have seen how we can create a kernel for the NanosG20 board, but there are still open questions. Where do we get a cross compiler? How do we create the Linux file system structure? Does every user space application and library need custom compilation? We use Buildroot to get these things done. Buildroot is a set of Makefiles and patches that makes it easy to generate a complete embedded Linux system. Buildroot can generate any or all of a cross-compilation toolchain, a root file system, a kernel image and a bootloader image.\(^3\)

Buildroot can be configured by using the command ‘make menuconfig’. Configuration of Buildroot covers hardware, build process and software-related options\(^4\). Buildroot will download all necessary files, compiles a kernel with the cross compiler it creates, creates a file system structure and compile selected user space packages just by running ‘make’.

![Figure 2.1: Configuration interface for Buildroot after using ‘make menuconfig’](image)

We use Buildroot to create Linux systems for the NanosG20. Using Buildroot we create a non real-time Linux system. We will go through the Buildroot configuration for the NanosG20. As can be seen in figure 2.1, we use target architecture arm, variant arm926t and target ABI (EABI). Linux 2.6.35.x kernel headers are used for the toolchain. At the system configuration we change the ‘to run a getty (login prompt) on’ option from ttyS0 to ttyS1. At package selection we select the network application OpenSSH and text editor Vim. We change the kernel options to use our custom Linux source code, for Linux kernel configuration we select the nanosg20 defconfig file and choose the zImage as kernel binary format. We leave build options, host utilities, file system images and bootloaders untouched.

The root file system Buildroot created are stored on a SD card for the NanosG20. We add a boot directory to the Linux file system and copy the zImage to this directory. We also copy two boot files from the default Debian file system\(^5\) to this directory, /boot/boot.sh and /boot/cmdline.

We could repeat all these steps to create a real-time Xenomai kernel. To save time we use the file system we just created. We change the kernel image to a Xenomai kernel and store the user

---

3Buildroot website, http://buildroot.uclibc.org/
4Roll Your Own Embedded Linux System with Buildroot, Linux Journals http://www.linuxjournal.com/article/10795
5The NanosG20 comes default with a Debian version. This can be downloaded from http://www.ledato.deaswell
components on the new Xenomai system. The Xenomai kernel image was created as described at section ‘Creating real-time Linux kernels’. We now have one real-time Linux system and one non real-time Linux system. In future references to Xenomai stands for the real-time system and normal stands for the non real-time system.

2.3 Benchmarking real-time Linux

There are three categories of tests for a real-time kernel [5]: functionality, throughput and latency. When a real-time Linux kernel is expected to have the same behavior for all the APIs as a standard kernel, this can be tested. There are multiple standard functionality tests available for Linux systems. Throughput benchmarks can also be used for real-time systems. The throughput of a real-time kernel can be lower due to the changes in the kernel where predictability had a higher priority over performance. For instance real-time Linux using a microkernel will introduce extra latency for non-real-time interrupts because they will now have to go through the microkernel as well. The latency benchmarks is the most important for real-time systems since most real-time tasks are responding to an event.

We want to know the rate at which sensor data can be exchanged between the NanosG20 and XEM6001. The protocol we will create in our last benchmark uses external interrupts to let the NanosG20 know new data can be read. Before we create this protocol we can ask the question: What is the response time to external interrupts for the NanosG20 kernels? The first benchmark will be answering this question by measuring the response time. Our second benchmark will test the behavior of the kernels when multiple interrupts arrive at the same time. In our last benchmark we will try to communicate between the two boards and measure the rate of data transfer.

More detailed information about the benchmarks will be given in chapter 4.
CHAPTER 3

Characteristics of the platforms

3.1 Characteristics of the NanosG20

3.1.1 Hardware specific

We will look briefly at the characteristics of the NanosG20 that are important for a real-time system.

The board has 32KB of data cache, 32KB of instruction cache and 128MB of SDRAM. The Linux kernel version used for this board (2.6.35.7) has a size of 2.0MB. This means that it cannot fit inside the cache, but it fits inside the SDRAM. The same size is observed with custom compiled (real-time) kernels. The Debian file system, which comes standard with the board, has a size of almost 150MB. It can thereby not fit inside the SDRAM, but a custom compiled Linux has a smaller file system and uses only 85MB. The result is a system which will experience a high amount of cache misses even when running in kernel space and will only have to go to the flash memory once.

As stated before interrupts are an important part of an embedded real-time system. The board has an Advanced Interrupt Controller (AIC) which is able to handle 32 interrupts. The interrupts can either be internal or external. All the boards PIO pins can also be used for interrupts. It is important to note that the GPIO pins on the NanosG20 can only be configured to trigger on both the rising and falling edges.

3.1.2 Kernel modules for NanosG20

The benchmarks in the next chapter will consist of kernel modules for the NanosG20. ‘Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.’

As kernel modules are run in kernel space they do not use the main() function and user space libraries are not available. This makes module programming different from user space programs. The Makefile used to create kernel modules for the benchmarks is included in appendix A. The Makefile specifies the compiler for the system to be the arm-linux- compilers Buildroot created for us. Compiling a kernel module gives us a KO file. These can be inserted in the kernel using insmod module_name.ko and removed with rmmod module_name.

3.2 XEM6001 Configuration

Opal Kelly, the company which created the XEM6001, provides a program called FrontPanel for interaction with the XEM6001. FrontPanel is able to run on Windows and Apple platforms. There is no support for Linux platforms. In our environment the FPGA will have to be configured.

from a Linux environment, the NanosG20 board. We will have to reverse engineer the protocol used by FrontPanel in order to create a configuration program that can be used on the NanosG20.

We are not the first with this problem. On the online source code repository SourceForge Jennifer Holt placed a C program named OpenOK to configure an OpalKelly FPGA in a Linux environment\(^2\). The code was made for the OpalKelly FPGAs XEM3005 and XEM3010. OpenOK does not work for our XEM6001. We will reuse the OpenOK code to create a configuration program for the XEM6001.

3.2.1 FrontPanel configuration protocol

Using a program to log USB messages, we can see what type of packages and data is sent to the FPGA during configuration. We will use this to reproduce the protocol.

Analyzing the output we can see that FrontPanel starts with getting information from the device by sending standard USB GET DESCRIPTOR and GET STATUS messages. FrontPanel continues by reading information every 110 milliseconds with control transfer messages. The FPGA responds with the byte 0x02. We do not know what it means, but since it is always sending this package before the configuration, a good guess would be that the FPGA is telling FrontPanel that it is ready to be configured. As we start configuration, FrontPanel reads information from the device by using a control transfer packet. FrontPanel then sends a control transfer package. The information it reads and answer it sends remain the same when using different configuration files. FrontPanel continues by sending the configuration file using bulk or interrupt transfer. FrontPanel uses a last control transfer to check whether or not the configuration worked.

3.2.2 Implemented configuration protocol

We did not have to follow the full protocol we have seen FrontPanel use for our configuration program. The final code, which can be seen in appendix B, uses almost all the OpenOK code. The code does not use the setup messages before configuration that OpalKelly used. When we try to create those setup messages we run into a problem with the answer message. USB control transfer messages use a setup packet. The second byte of the setup packet is used to let the device know what the request is. The USB log did not show the values of this byte. Trying every number, 0 to 255, show us that all the numbers from 150 to 250 seem to arrive correctly. Since we do not know which request to use and the configuration works without these messages we left it out of our protocol.

The configuration is split into three functions. The function `get_OK_devices` creates a list of all connected OpalKelly devices. This is done by looping through all the connected devices and checking the Vendor ID of the device. We select the OpalKelly device we want to use from the list and ‘open’ the device using the OK_open_dev function. This is done using two Libusb functions and sending a control transfer message. The last function `OK_configure_FPGA` configures the FPGA by sending the configuration file using bulk transfer.

3.3 Boards setup

The setup of the boards can be seen in figure 3.2. We use the serial port (2) and the program minicom to log in to the Linux system. The USB connection between the FPGA (4) and NanosG20 (5) is used to configure the FPGA with a configuration file. Our Linux kernels are stored on different SD cards and is used by the boards micro SD card slot (7). We use the internet connection (8) for easy file exchange between the NanosG20 and a desktop computer using OpenSSH. The benchmarks will use GPIO pins (9) for external interrupts on the NanosG20. The wired connection between the two boards (3) will be used to exchange sensor data.

We created a communication protocol which the boards will use. The protocol will use four of the five wires since one of them is used as electrical ground. The four wires are used as a bit of data. Both boards can read or set the values of these ‘bits’. Each board uses two lines for output to the other board: one for data and one to let the other board know data is ready. For

\(^2\)Link to the OpenOK repository: [http://sourceforge.net/projects/openok/](http://sourceforge.net/projects/openok/)

12
the NanosG20 the ‘is ready line’ is handled as interrupt. The XEM6001 will poll this line from the NanosG20 every cycle to check whether data from the NanosG20 is ready.

Using the protocol the boards should send bits of data back and forth. A timing diagram of the protocol can be seen in figure 3.1. Time 0 is the start situation with all the lines at zero. At time is 1 the NanosG20 wants to send a bit of data. It sets the data bit and the ‘is ready’ high for a short time. At time 3 the XEM6001 reads the data and starts sending data back: data
lines goes to 1, interrupt line is set high and low right after. At time 5 the NanosG20 handles the incoming interrupt and begins with sending a bit to the XEM6001. This should be able to go on until the desired amount of bits have been send. We will see in our benchmark that the boards will not be able to handle the protocol with our implementation.

Figure 3.2: Timing diagram for communication protocol between NanosG20 and XEM6001. W: write edge line from NanosG20. Dn: data line from NanosG20. I: interrupt line from XEM6001. Dx: data line from XEM6001.
4.1 Measurement of the response time

This benchmark is a measurement of the response time on periodic external interrupts of the NanosG20. The setup for the benchmark includes a pulse generator and an oscilloscope. The setup can be seen in figure 4.1. The pulse generator will be used to generate periodic external interrupts on a GPIO pin of the NanosG20. The NanosG20 interrupt handler will toggle an output GPIO pin likewise. The output of the pulse generator and the output GPIO pin of the NanosG20 are given to an oscilloscope. From the display of the oscilloscope we can read the response time.

Kernel modules will be used to configure the GPIO pins as interrupts and output. We created a kernel module for the normal and Debian kernel and a kernel module for the Xenomai kernel. The kernel modules can be seen in appendix C. The difference is the use of the real-time driver model for the Xenomai kernel.

We measure the minimal response time and the total jitter. The total jitter is the time difference between the minimal response time and the maximal response time.

We will benchmark the upper limit of the response time by benchmarking multiple times with different amount of workload tasks running on the system. Each benchmarks will run for
Figure 4.2: Average response time of response time benchmark. 
Time in microsecond ($10^{-6}$).
Workload is ‘dd if=/dev/zero of=/dev/null’

Figure 4.3: Jitter of response time benchmark. 
Time in microsecond ($10^{-6}$).
Workload is ‘dd if=/dev/zero of=/dev/null’

10 minutes. Figures 4.2 and 4.3 show the results.

The two graphs clearly show that the Xenomai kernel has the best performance. Not only is the response time and the jitter significantly lower, the jitter itself is more consistent. This means a more predictable response time, which is what one would like to see in a real-time system.
While the average response time of the Debian and the normal kernel are almost identical, the total jitter of the Debian makes it the kernel with the worst performance.

4.2 Multiple Interrupts

Some real-time systems must be capable of handling multiple events. In our last benchmark we only had one interrupt at a time. The next benchmark will consist of testing the behaviour of the kernels on multiple interrupts.

The setup of the benchmark is shown in figure 4.4. The setup is similar to the last benchmark, but now we use the pulse generator to trigger two interrupts on GPIO pins. We use the same kernel modules from the last benchmark, but now use two of them to handle both interrupts. We create a copy of the kernel modules and change the GPIO pins for our second module. We insert the two kernel modules and are able to use the two output pins to look at the handling of the interrupts.

The benchmarks were performed using a different oscilloscope. The new oscilloscope had more options and was able to calculate the mean, minimum, maximum and mean of the response times. The benchmark used 15000 samples to create reliable values to one decimal.

![Figure 4.4: Setup for multiple interrupts benchmark](image)

Table 4.1: Normal kernel multiple interrupt benchmark. Workload 0 and 200. Time in microseconds ($10^{-6}$s)

<table>
<thead>
<tr>
<th>Workload</th>
<th>pin A</th>
<th>pin B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mean</td>
<td>minimum</td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>6.7</td>
<td>4.1</td>
</tr>
<tr>
<td>200</td>
<td>mean</td>
<td>minimum</td>
</tr>
<tr>
<td>pin A</td>
<td>3.5</td>
<td>.002</td>
</tr>
<tr>
<td>pin B</td>
<td>4.9</td>
<td>.002</td>
</tr>
</tbody>
</table>
Table 4.2: Xenomai kernel multiple interrupt benchmark.
Workload 0 and 200. Time in microseconds ($10^{-6}$s)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Pin A</th>
<th>Pin B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>minimum</td>
</tr>
<tr>
<td>0</td>
<td>3.3</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>7.7</td>
<td>7.4</td>
</tr>
<tr>
<td>200</td>
<td>3.3</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>9.0</td>
<td>7.6</td>
</tr>
</tbody>
</table>

Looking at the results in the tables 4.1 and 4.2 we can see that the kernels handle interrupts differently. The Xenomai kernel handles one interrupt at a time. This can be seen at workload 0 where the standard deviation of 0.292s is almost the double of 0.156s. The normal kernel favors interrupt pin A over B, but seems to handle interrupts in another way since the maximum value and standard deviation are less apart. It could be that some of the interrupts on pin B preempt running interrupts of pin A. This is possible since we had the configuration for a preemptive kernel. The mean and minimum response time of both pins seem to decrease with a workload of 200 processes for the normal kernel. This is not consistent with previous benchmark where the response time climbed above 13s. It is not clear why the normal kernel has this behavior.

4.3 Communication benchmark

We use a kernel module for the NanosG20 to implement the communication protocol described in chapter 3. The XEM6001 uses a configuration file created with the hardware description language Verilog. The code can be seen in appendix D. Our first implementation failed and figure 4.5 is a timing diagram showing what goes wrong.

![Timing diagram for communication protocol between NanosG20 and XEM6001.](image)

Figure 4.5: Timing diagram for communication protocol between NanosG20 and XEM6001. W: write edge line from NanosG20. Dn: data line from NanosG20. I: interrupt line from XEM6001. Dx: data line from XEM6001.

The problem arises when the XEM6001 starts sending a bit to the NanosG20. After the two lines are set the board checks the ‘is ready’ line again from the NanosG20. The NanosG20 has not set this line low yet and the XEM6001 sends another bit at time 3. This results in an overload of multiple interrupts send to the NanosG20.

In our second implementation we let the XEM6001 only send new data when it has seen the write edge go down. This way the NanosG20 can take as much time as needed to set the data
line low without the XEM6001 sending more interrupts. We still had a problem in the XEM6001 code. The interrupt line is set high and low the cycle afterwards. Since the GPIO pins on the NanosG20 can only be triggered on both the rising and falling edges this sends two interrupts to the NanosG20.

We tested the second implementation on the normal and Xenomai kernel. Due to the high number of interrupts the kernels had to process they would not be able to do anything else. We had a counter keep track of the number of handled interrupts. Using a stopwatch we checked how many interrupts were handled after 30 seconds. For the normal kernel this was somewhere in between 10 and 50. The Xenomai kernel handled on the other hand 10514194 interrupts. The Xenomai kernel was able to handle this interrupts because it only preempts real-time tasks with real-time task having higher priority. It was handling the interrupts in FIFO order. It was doing this at the amazing speed of only 3\(\mu\)s per interrupt. Interrupts on the normal kernel interrupts were able to preempt each other since they were implemented as kernel threads.
In this paper, we have seen Xenomai performing better to our benchmarks. Xenomai was able to respond twice as fast to external interrupts than the standard Linux kernel. Xenomai handled multiple interrupts with the same priority in a first come first serve order. This lowered the maximum response time of the interrupt handled first and gave a more predictable response time. Even when too many interrupts were given to handle, Xenomai was able to handle a lot of them.

The way Xenomai handles interrupts is more predictable. This is the exact result we would like to see for a real-time system.

5.1 Future work

Future research can be done by creating a working communication protocol between the two boards. A new protocol could make use of the measured response times of the NanosG20. It is then possible to see how the system reacts to sensor data being send sequential. Questions as how long sensor data will need buffering inside the FPGA can be answered.

It is also possible to go another way and benchmark other Linux real-time extensions. Real-time extensions as RTAI and real-time linux patches are examples.


NanosG20 kernel module makefile

1 ARCH=arm
2 CROSS_COMPILE=/path/to/buildroot/output/host/usr/bin/arm-linux-
3 KDIR := /path/to/linux−2.6.35.7−nanosg20
4 PWD := $(shell pwd)
5
6 obj-m += module_name.o
7
8 all:
9 $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules ARCH=$(ARCH)/
10 CROSS_COMPILE=$(CROSS_COMPILE)
11
12 clean:
13 $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
OpenOK communication code for XEM6001

B.1 OpenOK.h

```c
/*
 * openok.h
 *
 * Created on: Oct 15, 2009
 * Author: Jennifer Holt
 * Open Source Opal Kelly interface library
 *
 Copyright (c) 2009 Jennifer Holt

 Permission is hereby granted, free of charge, to any person obtaining a copy
 of this software and associated documentation files (the "Software"), to deal
 in the Software without restriction, including without limitation the rights
 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 copies of the Software, and to permit persons to whom the Software is
 furnished to do so, subject to the following conditions:

 The above copyright notice and this permission notice shall be included in
 all copies or substantial portions of the Software.

 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 THE SOFTWARE.

 */

#ifndef OPENOK_H
#define OPENOK_H

#endif /* OPENOK_H */
```
#ifndef USB_H
#include <usb.h>
#define USB_H
#endif /* usb.h */

/*usb.h*/

/**************************** defines ****************************/
define USB_DIR HOST 0x80

/**************************** structures ****************************/

struct OKdevice{
  struct usb_device *OK_dev; // usb device identifier, use with usb_open(OK_dev)
  char OK_idVendor[255]; //usb vendor id
  char OK_idProduct[255]; //usb product id
  char OK_serial[255]; //device serial number
  char OK_man[255]; //manufacturer string
  char OK_prod[255]; //product string
};

/**************************** functions ****************************/

int get_OK_devices(struct OKdevice device_list[]);
usb_dev_handle* OK_open_serial(char serial[]);
usb_dev_handle* OK_open_dev(struct usb_device*OK_dev);
int OK_close(usb_dev_handle* device);
int OK_configure_FPGA(usb_dev_handle* OK_dev, const char* Filename);
int OK_update_wirein(usb_dev_handle* OK_dev, char* data);
int OK_get_wireout(usb_dev_handle* OK_dev, char* data);
int OK_set_trigger_in(usb_dev_handle* OK_dev, int ep, char* data);
int OK_getTrigger_out(usb_dev_handle* OK_dev, char* data);

B.2 OpenOK.c

/*
 * openok.c
 * * Created on: Oct 15, 2009
 * * Author: Jennifer Holt
 * * Open Opal Kelly interface library
 */

Copyright (c) 2009 Jennifer Holt

* version 0.3: august 1, 2012
* Author: Rik van der Kooij, Taco Walstra
* Small changes to support XEM6000 series

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
#include <stdio.h>
#include <string.h>
#include "openok.h"
#include "usb.h"

/* USE DEFINES DURING MAKE TO COMPILE FOR THE CORRECT BOARD TYPE!! */

int get_OK_devices(struct OKdevice device_list []) {
    /* call usb_init() before calling this function*/
    struct usb_bus *bus;
    struct usb_device *dev;
    int i;
    i=0; // used to keep track of devices
    usb_find_busses();
    usb_find_devices();

    for (bus = usb_busses; bus; bus = bus->next) {
        for (dev = bus->devices; dev; dev = dev->next) {

            usb_dev_handle *udev;
            int ret;
            char string[256];

            if (dev->descriptor.idVendor==0x151F){ //check if device is made by
                // opal kelly
                device_list[i].OK_dev = dev; //save device
                sprintf(device_list[i].OK_idVendor,"%04X",
                        dev->descriptor.idVendor); //save vendor id

                sprintf(device_list[i].OK_idProduct,"%04X",
                        dev->descriptor.idProduct); //save product id

                udev = usb_open(dev); //open device
                if (udev) {
                    if (dev->descriptor.iManufacturer) {
                        ret = usb_get_string_simple(udev,
                            dev->descriptor.iManufacturer,
                            string, sizeof(string));
                        if (ret > 0)
                            sprintf(device_list[i].OK_man,"%s", string);
                        else
                            sprintf(device_list[i].OK_man,
                                "Unable_to_fetch_manufacturer_string");
                    }
                }
            }
        }
    }
}
if (dev->descriptor.iProduct) {
    ret = usb_get_string_simple(udev, 
        dev->descriptor.iProduct, 
        string, sizeof(string));
    if (ret > 0)
        sprintf(device_list[i].OK_prod,"%s", string);
    else
        sprintf(device_list[i].OK_prod, 
            "Unable to fetch product string");
}

if (dev->descriptor.iSerialNumber) {
    ret = usb_get_string_simple(udev, 
        dev->descriptor.iSerialNumber, 
        string, sizeof(string));
    if (ret > 0)
        sprintf(device_list[i].OK_serial,"%s", string);
    else 
        sprintf(device_list[i].OK_serial, 
            "Unable to fetch serial number string");
}

usb_close (udev);

i++;

return i;

}

usb_dev_handle* OK_open_serial(char serial []) {
    usb_dev_handle *udev;
    return udev;
}

usb_dev_handle* OK_open_dev(struct usb_device *OK_dev) {
    usb_dev_handle* udev;
    char c [1];
    int i;
    udev=usb_open(OK_dev); //open device
    if (udev) {
        //need to send setup packet on endpoint 0x00
        //packet is 0x00 09 01 00 00 00 00 00 00
        //libusb provides a convenient function usb_set_configuration for this
        i = usb_set_configuration(udev, 1);
        if (i == 0) {
            i = usb_claim_interface(udev,0);
            if(i == 0){ //claim interface 0
                //need to send USB_CONTROL packet , 0xc0:b9 00 00 00 00 00 01 00
                //this is copied from a capture of traffic when using the
                //official interface library
                //don’t know what this request is, but it returns a byte of 0x80
                //when called by real library
            }
i = usb_control_msg(udev, 0xc0, 0xb9, 0, 0, c, 1, 500);
if(i == 1) {
  // should have read 1 byte
  if (c[0] != (char)0x80) {
    printf("Device did not return 0x00\n");
    usb_close(udev);  // close device
    udev = NULL;  // return NULL
  }
}
else {  // usb_control_message failed
  printf("Control message failed\n");
  usb_close(udev);  // close device
  udev = NULL;  // return NULL
}
else {
  printf("claming interface fail %d\n", i);
  usb_close(udev);  // claim didn’t work
  udev = NULL;  // return NULL
}
else {  // problem with setting configuration, close device and return NULL
  usb_close(udev);
  udev = NULL;
}
return udev;

int OK_close(usb_dev_handle* OK_dev) {
  return usb_close(OK_dev);
}

#ifdef XEM3001
*/Use this function for all the Opal Kelly older boards (non 6000 series)*/
int OK_configure_FPGA(usb_dev_handle* OK_dev, const char* Filename) {
  // FPGA configuration is sent as a bulk transfer, with a control transfer
  // preceding to tell the device to expect configuration data. the control
  // transfer has no data bytes ,
  // and has setup packet 0x40 b2 00 00 00 00 00 00
  // the bulk transfer is the bitstream from a Xilinx .bit file, starting with
  // the sync word FF FF FF FF AA 99 55 66 however the bitstream is sent with
  // each byte padded to 16bits on the right, so 0xFF becomes 0xFF00 in the
  // data actually sent over USB, in the real library, a configure call is
  // ended with another control transfer, this appears to be a check to see if
  // configuration was successful the transfer has 1 data byte and the setup
  // packet is 0xc0 b2 00 00 00 00 01 00, a byte of 0x01 is returned on
  // success, and I have seen a byte of 0x02 returned when it didn’t work, but
  // I don’t know what the 0x02 means, or if there are other return values
  // indicating other errors

  char c[9];
  char* buffer;
  int i, size;
  long pos, position;
FILE * pFile;

// configures the fpga on an xem with the bitstream in Filename

// first get and parse the bitstream from the file

pFile = fopen(Filename,"r");
if (pFile) {
    // get length of file
    pos = ftell(pFile);
    fseek(pFile, 0, SEEK_END);
    size = ftell(pFile);
    fseek(pFile, pos, SEEK_SET);

    // allocate memory buffer for bitstream (size of file*2 bytes, since the
    // bitstream is sent padded to 16 bits)
    buffer = (char*)malloc(size*2);
    position = 0; // put buffer position at 0
    if (buffer) {
        // need to throw out everything until we reach
        // the FF FF FF FF AA 99 55 66 sync word
        // first read in 7 bytes
        for (i=1; i<8; i++)
            c[i] = fgetc(pFile);
        c[8] = 0x00; // terminate string
        // shift all chars in buffer left, read in a char, compare buffer with
        // FFFFFFFFA995566, if match, exit loop
        do {
            for (i=0; i<7; i++)
                c[i] = c[i+1]; // shift chars in buffer
            c[7] = fgetc(pFile); // read in new char
            if (strcmp(c,"\xFF\xFF\xFF\xFF\xFF\xFF\xAA\x99\x55\x66") == 0)
                break;
        } while (!feof(pFile));
    }
    if (!feof(pFile)) { // if we reached the end of the file, the sync
        // word wasn’t found
        // put sync word at start of buffer, appending 0x00 to each byte
        for (i=0; i<8; i++) {
            buffer[position++] = c[i];
            buffer[position++] = 0x00;
        }
        // read in bitstream, appending 0x00 to the end of each byte
        // (to make 16 bit words)
        while (!feof(pFile)) {
            buffer[position++] = fgetc(pFile);
            buffer[position++] = 0x00;
        }
    }

    // first there is a control transfer to set up the operation
    // 0x40:0xb2:0x0000000000000000
    i = usb_control_msg(OK_dev, 0x40, 0xb2, 0, 0, c, 0, 700);
    if (i>0){ // check for error
        // do a bulk transfer of the file to endpoint 0x02, starting
        // with the header of the xilinx file FF FF FF FF AA 99 55 66
        // however every byte is sent as an int16 with the actual
250 // byte in the upper bits, i.e. 0xff -> 0xff00. this was taken
251 // care of in the loading and parsing of the file
252 i=usb_bulk_write(OK_dev, 0x02, buffer, position-2, 2000);
253 if (i==position-2) {
254 // do another control transfer to check status
255 i=usb_control_msg(OK_dev, 0xc0, 0xb2, 0, 0, c, 1, 700);
256 if (i==position-2) {
257 i=0;
258 if (c[0]!=0x01) { // check for successful status
259 i=0;
260 } else {
261 printf("Status not correct (returned: 0xRegards") ,
262 c[0]);
263 }
264 } // end control check
265 } // end bulk write success
266 } // end control setup
267 } // end sync word not found
268 free(buffer); // deallocate memory for bitstream buffer
269 } // end allocate buffer for bitstream
270 } // end file open
271 return 0;
272 }
273 } #endif
274
275 #ifdef XEM6001
276 int OK_configure_FPGA (usb_dev_handle* OK_dev, const char* Filename) {
277 char c[17];
278 char* buffer;
279 int i, size;
280 long pos, position;
281 FILE * pFile;
282 // configures the fpga on an xem with the bitstream in Filename
283
284 // first get and parse the bitstream from the file
285 pFile = fopen (Filename,"r");
286 if (pFile)
287 {
288 // get length of file
289 pos = ftell (pFile);
290 fseek (pFile, 0, SEEK_END);
291 size = ftell (pFile);
292 fseek (pFile, pos, SEEK_SET);
293
294 // allocate memory buffer for bitstream. The bitstream is as single bytes
295 buffer = (char*)malloc(size);
296 position = 0; // put buffer position at 0
297
298 if (buffer)
299 {
300 // first read in 16 bytes
301 for(i = 1; i < 16; i++)
302 c[i] = fgetc(pFile);
303 c[16] = 0x00; // terminate string
304 do {
305 for(i = 0; i < 15; i++)
c[i] = c[i+1];  //shift chars in buffer
if(!strcmp(c, "\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xF
int OK_get_wireout(usb_dev_handle* OK_dev, char* data) {
    //wireout's are sent as a control transfer. setup packet is
    //0xc0 b5 20 00 00 00 40 00. the 0x0020 value in wValue was somewhat
    //unexpected, since there is no corresponding one in the wirein call. there
    //are 64 bytes representing the values on each of the 32 possible
    //wireout's. wireout endpoint 0x20 is sent as the first pair of bytes
    //LSB:MSB, and the rest follow in order until the last possible wireout
    //endpoint of 0x3F, which takes the last two bytes in the data
    //
    //the real wireout function allows you to specify a particular endpoint you
    //want data for. my function fills the buffer pointed to by data with all 64
    //bytes and you have to extract the one you want yourself. endpoint 0x20 is
    //the first two bytes LSB:MSB and the rest follow in order until endpoint
    //0x3F which is the last two bytes.
    int i = usb_control_msg(OK_dev, 0xc0, 0xb5, 0x0020, 0, data, 64, 700);
    return i;
}

int OK_set_trigger_in(usb_dev_handle* OK_dev, int ep, char* data) {
    //trigger ins are sent as a control transfer. the format appears is
    //0x40 0xb5 0x[endpoint] 0x00 0x01 0x00 0x02 0x00 which corresponds to
    //usb_control_msg(OK_dev, 0x40, 0xb5, endpoint, 1, data, 2, 700)
    //the 2 data bytes represent the 16 possible triggers associated with the
    //endpoint. The original library only allows for setting 1 bit at a time, so
    //for each transaction, the data bytes would have only 1 bit set. I don't
    //know what the result of setting more than on bit would be. A reasonable
    //guess would be that multiple triggers would be set. so for compatible
    //operation, only have one set bit in the 2 bytes pointed to by data
    int i = usb_control_msg(OK_dev, 0x40, 0xb5, 0x0001, 1, data, 2, 700);
    return i;
}

int OK_getTrigger_out(usb_dev_handle* OK_dev, char* data) {
    //trigger outs are sent exactly the same as wire outs, but with a different
    //setup byte: 0xc0 0xb5 0x00 0x00 0x10 0x00 0x40 0x00
    int i = usb_control_msg(OK_dev, 0xc0, 0xb5, 0x00060, 1, data, 64, 700);
    return i;
}
APPENDIX C

Response time benchmark code

C.1 Normal and Debian

```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/interrupt.h>
#include <mach/gpio.h>

#define gpio_in AT91_PIN_PB17;
#define gpio_out AT91_PIN_PB1;

static irqreturn_t irq_handler(int irq, void *dev_id)
{
    int output_value = gpio_get_value(gpio_in);
    gpio_set_value(gpio_out, output_value);
    return IRQ_HANDLED;
}

int __init init_module()
{
    at91_set_gpio_input(gpio_in, 0);
    at91_set_gpio_output(gpio_out, 0);
    return request_irq(gpio_in, irq_handler, irq_handler_function
                        0, flags, "gpio_irq", name, 0);
}

return 0;
}

void cleanup_module()
{
    free_irq(gpio_in, 0);
}
```
C.2 Xenomai

```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include <rtdm/rtdm_driver.h>
#include <rtdm/rtdm.h>
#include <native/intr.h>
#include <mach/gpio.h>

rtdm_irq_t irq_handle;
int gpio_in = AT91_PIN_PB17;
int gpio_out = AT91_PIN_PB1;

int irq_handler(rtdm_irq_t *irq_handle)
{
    int output_value = gpio_get_value(gpio_in);
    gpio_set_value(gpio_out, output_value);
    return RTDM_IRQHANDLED;
}

int __init init_module()
{
    at91_set_gpio_input(gpio_in, 0);
    at91_set_gpio_output(gpio_out, 0);

    rtdm_irq_request(&irq_handle, /* irq_handler */
    gpio_in, /* irq number */
    irq_handler, /* irq handler function */
    0, /* flags */
    "gpio_irq", /* name */
    0); /* pointer for irq_handler */
    rtdm_irq_enable(&irq_handle);
    return 0;
}

void cleanup_module()
{
    rtdm_irq_disable(&irq_handle);
    rtdm_irq_free(&irq_handle);
}

MODULE_LICENSE("GPL");
```
APPENDIX D

Communication benchmark code

D.1 Normal

```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/interrupt.h>

#include <mach/gpio.h>

int D0 = AT91_PIN_PA0; /* write edge from nanosg20 processor */
int D1 = AT91_PIN_PA1; /* data line from nanosg20 processor */
int D2 = AT91_PIN_PA2; /* interrupt line to nanosg20 */
int D3 = AT91_PIN_PA3; /* data line to nanosg20 */

static irqreturn_t irq_handler(int irq, void *dev_id)
{
    int data = gpio_get_value(D3);
    gpio_set_value(D1, !data);
    gpio_set_value(D0, 1);
    gpio_set_value(D0, 0);
    return IRQ_HANDLED;
}

int __init init_module()
{
    at91_set_gpio_input(D2, 0);
    at91_set_gpio_input(D3, 0);
    at91_set_gpio_output(D0, 0);
    at91_set_gpio_output(D1, 0);

    return request_irq(D2, irq_handler, /* irq number */
            irq_handler, /* irq handler function */
            0, /* flags */
            "gpio_irq", /* name */
            0); /* pointer for irq handler */
}
```

39
38 void cleanup_module ()
39 {
40     free_irq(D2, 0);
41 }
42
43 MODULE_LICENSE("GPL");

D.2 Xenomai

1 #include <linux/init.h>
2 #include <linux/kernel.h>
3 #include <linux/module.h>
4 #include <rtdm/rtdm_driver.h>
5 #include <rtdm/rtdm.h>
6
7 #include <native/intr.h>
8 #include <mach/gpio.h>
9
10 rtdm_irq_t irq_handle;
11 int D0 = AT91_PIN_PA0; /* write edge from nanosg20 processor */
12 int D1 = AT91_PIN_PA1; /* data line from nanosg20 processor */
13 int D2 = AT91_PIN_PA2; /* interrupt line to nanosg20 */
14 int D3 = AT91_PIN_PA3; /* data line to nanosg20 */
15
16 int irq_handler(rtdm_irq_t *irq_handle)
17 {
18     int data = gpio_get_value(D3);
19     gpio_set_value(D1, !data);
20     gpio_set_value(D0, 1);
21     gpio_set_value(D0, 0);
22
23     return RTDM_IRQ_HANDLED;
24 }
25
26 int _init init_module()
27 {
28     at91_set_gpio_input(D2, 0);
29     at91_set_gpio_input(D3, 0);
30     at91_set_gpio_output(D0, 0);
31     at91_set_gpio_output(D1, 0);
32
33     return rtdm_irq_request(&irq_handle, /* irq_handler */
34                             D2, /* irq number */
35                             irq_handler, /* irq handler function */
36                             0, /* flags */
37                             "xen_irq", /* name */
38                             0); /* pointer for irq_handler */
39 }
40
41 void cleanup_module ()
42 {
43     free_irq(D2, 0);
44     rtdm_irq_disable(&irq_handle);
45     rtdm_irq_free(&irq_handle);
D.3 XEM6001

// Stripped from initiation parts of the code.
// State machine handling the PIO pins
// D0 = write edge from nanos20 processor
// D1 = dataline from nanos20 processor
// D2 = interrupt line to nanos20
// D3 = dataline to nanos20
// S0 ==> wait for write pin from nanos20. If 1 move to S1
// S1 ==> Read D1 pin of nanos20 put D3 to the same value as D1. Put D2 high
// and move to S2
// S2 ==> put D2 low and move back to S0
always @(state or reset1)
begin
  if (reset1)
    d3 <= 1'b0;
  else
    case (state)
      S0 : // default state
        begin
          d2 <= 1'b0;
        end
      S1 : // write pulse detected
        begin
          d3 <= d1; // copy d1 data
          d2 <= 1'b1; // activate the interrupt line
        end
      S2 : // release interrupt line
        default:
        begin
          d2 <= 1'b0;
        end
    endcase
  end
always @(posedge clk1 or posedge reset1)
begin
  if (reset1)
    state <= S0;
  else
    case (state)
      S0:
        begin
          if (d0)
            state <= S1;
          else
            state <= S0;
        end
      S1:
        state <= S2;
      S2:
        state <= S0;
    endcase
51   endcase
52   end
53   endmodule