SynDEx Communications Under Linux

Pierre POMIERS <pierre@robosoft.fr>
Tony NOEL <Tony.Noel@inria.fr>

Abstract

This document aims to present how to program SynDEx CAN (or others) communications sequence under Linux (RTLinux or RTAI). It will collect precise information about system calls under Linuses, obtained from various programming guides. By this way, this will help us to design a good communication sequence functioning implementation according to the SynDEx concept.

1 SynDEx communication sequence [1]

1.1 Principles of operation

Each communication medium, i.e. here each CAN bus, is a sequential resource where interprocessor data transfers must be ordered. Rather than a dynamic ordering, done at run time by the CAN arbitration on transfers contents, and which requires identification information to be transferred and decoded, a static ordering is chosen at compile time, where each transfer is uniquely identified simply by its position in the communication sequence. As the CAN bus has no centralized communication sequencer, the communication sequence must be distributed and synchronized on all the processors sharing the bus: at each communication step, all processors know the transfer type and size, the transmitting processor knows its source buffer address, each receiving processor knows its destination buffer address, and each remaining processor is synchronizing, i.e. waits for data as if it were receiving, but does not store the data. Moreover, to avoid dynamic buffer allocation and overruns, the transmitting processor does not start transmitting data before it has received, from each receiving processor, an empty synchronizing frame meaning that the destination buffer is available for reception.

1.2 Present implementation

Each communication step (see `CAN_send_`, `CAN_recv_` and `CAN_sync_` macros) takes as arguments the data buffer name, the transmitting processor type and name, and the receiving processors names. Each communication step is processed in several stages under the control of an automaton. Initially, and after each communication step completes, a synch-frame counter is reset, and then incremented by the interrupt triggered by each received synch-frame (automaton state: `CAN_init_state`). When a communication step is initiated (see `CAN_send_shared`, `CAN_recv_shared`, and `CAN_sync_shared`), the base address and the byte size of the data buffer (computed from the data buffer name), as well as the address of the next instruction in the communication sequence, are stored in a "transfer context" for later use, and the synch-frame counter is decremented by the number of synch-frames to be received. Moreover, in the case of a receive step, a synch-frame is transmitted, regardless of the number of synch-frames already received, but with a CAN message identifier different for each processor; as only the number of synch-frames matters, it is easier to let the CAN bus arbitrate any contention between synch-frames, than to statically order synch-frames (which would require each receiving processor to transmit its synch-frame only after it has received the number of synch-frames which were to be transmitted before it). The automaton then waits for synch-frames in the `CAN_send/recv_waitingSynch_state`, counting receive interrupts. When the synch-frame counter reaches zero, all the receiving processors have their destination buffer ready for reception, then data transfers may begin using and updating the "transfer context" which holds the number of data bytes remaining to transfer and the address of the next byte to transfer. In the case of the transmitting processor (`CAN_sending_state`), CAN frame buffers are loaded with data and submitted for transmission; for a receiving processor (`CAN_recving_state`), frame buffers are (stored and re-)submitted for reception; for a synchronizing processor (`CAN_sync_state`), frame buffers are only submitted for reception (their contents is not stored to memory). After each frame buffer(s) submission, the automaton waits for the end-of-transfer interrupt, letting the processor execute the computation sequence. When the number of bytes remaining to transfer reaches zero, the communication sequence is resumed (`CAN_resume_state`) at the address stored when the communication step was initiated.

1.3 What happens from now?

The communication sequence described before is the one implemented both for 555 and 386/DJGPP machines. Its functioning reliability has been proofed through numbers of tests. Porting SynDEx communication kernel under Linux will consist in re-programming the set of CAN macros using Linux system programming possibilities. This way is a bit different from what has been done before. Indeed, when we were programming whether under 386/DJGPP or 555, we had a total control of the machine resources (IRQs, interrupt vector table, system calls...). Using Linux, we do not have direct access to low level programming anymore. As a secured Operating System, Linux kernel forbid direct hardware access by handling all the machine resources.

2 How do we access BIOS functions from Linux? [2]

No way. This is protected mode, use OS services instead. Again, you can't use int 0x10, int 0x13, etc. Fortunately almost everything can be implemented through system calls or library functions. In the worst case you may go through direct port access, or make a kernel patch to implement needed functionality.

2.1 Kernel modules [4]

Getting access to the Linux kernel "World" is done using what we call `modules'. Modules can be seen as pieces of programs adding functionality to an actual kernel (in a way like kernel patches). Before we go further, it is worth underlying some fundamental difference between a kernel module and a classical application.

While an application performs a single task from beginning to end, a kernel module registers itself in order to serve future requests. There are two main parts in a modules. First, the module's "main" function `init_module()`, which is in fact the module entry point, aims to prepare module's functions for later invocations. The second entry point of a module, `cleanup_module`, gets invoked just before the module is unloaded and aims to terminate the module's run.

Moreover, it is important to precise that, unlike application, is not linked through libraries (or any other external references) but directly linked to the kernel. Hence, the only functions a module can call, are the one exported by the kernel. This is a very important point. It very constrain our way to program SynDEx applications.

2.2 Influence on the SynDEx code generation

Before, all code needed by a SynDEx generated application, was included in a same assembler file. This was possible because no OS was restricting the access to hardware programming. Now, under Linux, as modules things are different. In order to interface our applications with the hardware (always through kernel function calls), we will have to make use of module(s).

For the moment, documentation is not very clean on this point: it is not sure it is possible to handler several IRQs to a same module (as we did in a classical 386/DJGPP SynDEx application). Some tests has been done to test if multiple IRQs assignment was possible or not. A rapid analysis of the `/proc/interrupts` Linux system file shows us that our assumption was correct. Nevertheless, we are not sure about the OS reactions to this way of programming yet? If this become to be impossible, we have to imagine the possibility to split our SynDEx application into several parts: one CAN communication module, eventually one Ethernet communication module and one "main" application process (or task whether we will use Linux, RTLinux or RTAI). All these pieces of program running together will have to communicate in order to exchange information. This additional mechanism will surely be realized making use of shared memory or fifos or others.

2.3 Preliminary information about Interrupt Handlers [3]

There are two types of interaction between the CPU and the rest of the computer's hardware. The first type is when the CPU gives orders to the hardware, the other is when the hardware needs to tell the CPU something. The second, called interrupts, is much harder to implement because it has to be dealt with when convenient for the hardware, not the CPU. Hardware devices typically have a very small amount of ram, and if you don't read their information when available, it is lost.

Under Linux, there are two types of IRQs, short and long. A short IRQ is one which is expected to take a very short period of time, during which the rest of the machine will be blocked and no other interrupts will be handled. A long IRQ is one which can take longer, and during which other interrupts may occur (but not interrupts from the same device). If at all possible, it's better to declare an interrupt handler to be long.

When the CPU receives an interrupt, it stops whatever it's doing (unless it's processing a more important interrupt, in which case it will deal with this one only when the more important one is done), saves certain parameters on the stack and calls the interrupt handler. This means that certain things are not allowed in the interrupt handler itself, because the system is in an unknown state. The solution to this problem is for the interrupt handler to do what needs to be done immediately, usually read something from the hardware or send something to the hardware, and then schedule the handling of the new information at a later time (this is called the `bottom half') and return. The kernel is then guaranteed to call the bottom half as soon as possible -- and when it does, everything allowed in kernel modules will be allowed.

The way to implement this is to call `request_irq` to get your interrupt handler called when the relevant IRQ is received (in our case there are 16 of them as Linux will run on an Intel platforms). This function receives the IRQ number, the name of the function, flags, a name for /proc/interrupts and a parameter to pass to the interrupt handler. The flags can include `SA_SHIRQ` to indicate you're willing to share the IRQ with other interrupt handlers (usually because a number of hardware devices sit on the same IRQ) and `SA_INTERRUPT` to indicate this is a fast interrupt. This function will only succeed if there isn't already a handler on this IRQ, or if you're both willing to share.

Then, from within the interrupt handler, we communicate with the hardware and then use `queue_task_irq` with `tq_immediate` and `mark_bh(BH_IMMEDIATE)` to schedule the bottom half.

2.4 Implementation differences between Linux and (RTLinux/RTAI) [5][6]

At that step, we do not know yet what type of platform OS we will use: Linux, RTLinux or RTAI? Let us not focus on this decision for the moment, but let us just check what difference we would encounter while implementing the communication macros. Before going on, let us remember that RTLinux and RTAI are based on the same principle and are considerate to be real-time Linuses.

For writing modules under "classical" Linux, we have to use functions provided by module dedicated library such as "module.h". They allow to manipulate hardware interrupts, interrupt handling. Normally, they should also allow you to deal with scheduling. Nevertheless, Linux was originally designed as a time-sharing system, that is to say that it strive to optimize average performance. Hence, the execution of a module running under Linux kernel control, will be constrained by other running process: in other words, there is really no way to guaranty a precise execution rate or time schedule.

Writing modules under (RTLinux/RTAI) is also possible. An other set of functions is available to this aim. Here, the main advantage of using (RTLinux/RTAI) is that it will allow modules execution to respect hard real-time constraints. In other words, it will provide low latency and high predictability.

Finally, it seems to be not so different implementing a module under Linux or (RTLinux/RTAI). This only important thing we have to keep in mind is that, "classical Linux" modules can only be interfaced with other "classical Linux" process. On the opposite, "(RTLinux/RTAI)" modules can be used both by other "(RTLinux/RTAI)" processes or by "classical Linux" processes. So, for the moment, just considering future users application, it seems to be less restrictive to implement "(RTLinux/RTAI)" modules: hence, users already using (RTLinux/RTAI) functionality will continue to have the possibility to use them into their SynDEx application, where for the other users, it will be totally transparent. Nevertheless, if we decide to implement SynDEx macros using (RTLinux/RTAI) modules, future users wanting to design SynDEx application for heterogeneous architecture including Linux machine, will be obliged to install (RTLinux/RTAI) on their Linux box. Is it really a constraint? I am not sure (in fact I am sure it is not), but some persons could be reluctant to do so.

Conclusions

Considering what we described above, there are not many way to program SynDEx communication macros. We have to use modules programming to access hardware resources: no other way is possible (unless we become kernel wizards and re-program our own SynDEx oriented Linux kernel... but I think it is not the purpose).

This way of programming macros will change a bit the way of macroprocessing applications code. Unlike before, we will not have just a single file to compile and execute but several communicating modules. In fact, we will have to deal with two programming layers: one concerning the modules code proper and a second one, relying on Unix mechanisms for modules "management".

The question of Linux or (RTLinux/RTAI) modules implementation is also an important point. Implementing modules under "classical Linux" is something very possible, but the "time-sharing" schedule method used by this OS is not compatible with our needs. It would be possible to program our own "SynDEx" scheduler and our own synchronizing methods but it would surely be very time consuming. On the other side, we have (RTLinux/RTAI). One could feel there is a paradox designing SynDEx using an other real-time OS. But, from a programming point of view, (RTLinux/RTAI) have to be seen as a toolbox providing many reliable methods (shared memory, semaphores, real-time scheduler...). Using (RTLinux/RTAI) would allow us to focus on the real SynDEx software architecture, without spending time programming already existing tools.

Now, before ending this paper, let us just have a purely technical word about (RTLinux and RTAI). RTAI is an Italian variant of RTLinux that has gone off on its own path. GPL encourages such divergence. Many differences are purely cosmetic, however the fundamental difference between the two systems is attitude towards features. The main thing that could make us choose RTAI rather that RTLinux is its aptitude to handle partially-linked modules through standard or math library for example (what RTLinux does not seem to propose). Moreover, unlike RTLinux, RTAI does not obliged user to patch and recompile its working Linux kernel. RTAI features (like fifos, scheduler and so on) can be inserted into Linux kernel using `insmod` from the shell command line.

References

[1] CAN.m4x
[2] http://www.caldera.com/LDP/HOWTO/Assembly-HOWTO-7.html#ss7.2.
[3] Linux Kernel Module Programming Guide (LKMPG).
[4] Alessandro RUBINI, "Linux Device Drivers", O'REILLY, February 1998.
[5] Michael Barabanov, "A Linux-based Real-Time Operating System", June 1997.
[6] Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano
Real Time Application Interface web site http://www.aero.polimi.it/projects/rtai/