SocketCAN in automation: A case study

By Luotao Fu (Pengutronix)

The SocketCAN [1] framework has been established for a while in the field of embedded Linux driven CAN controllers. Now with the migration of the SocketCAN layer into the mainline kernel, it is gaining more attention for its usage in industrial automation.

In this article I want to introduce our experience in working with the SocketCAN framework and the effort to improve it. I will describe a fictional project, which starts with the following scenario: There is a customer, who has a PowerPC-based board with a SOC built-in mscan controller. The board currently runs Linux with a 2.4.x kernel A separate processor unit is used to control measurement units. Several CAN-based higher-layer protocols are used for the communication between the controller board and the DSP.

The customer has his own application. Third-party stack libraries provide implementation of the fieldbus protocols. The CAN related routines were used to provide by a set of vendor libraries and own drivers. The call-back functions are integrated into the application and the stack libraries.

The goal of our project is a kernel update to 2.6.31, using SocketCAN. In short, drivers for the hardware and the application itself have to be updated. Due to the clear subproject goals we chose an iterative development model. The first iteration would contain kernel development, while the second iteration would take care of user-space development.

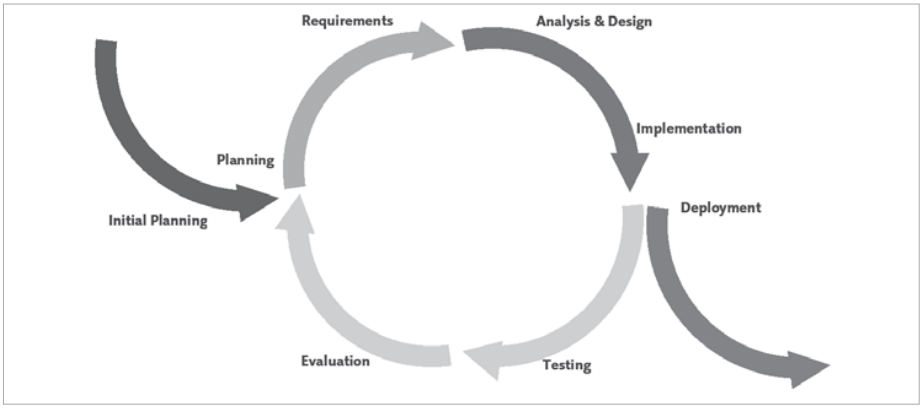

Fig. 1: The iteration cycle from initial planning to evaluation

First iteration: Kernel space

The first iteration of the project is the kernel space part. The main goal of this iteration is to update the 2.4.x Kernel to the recent 2.6.x series, which includes SocketCAN. The requirement is straightforward: A running 2.6.x kernel providing all functionalities, which the old 2.4.x kernel used to have. The related work here is porting the platform support and drivers for customer specific hardware. In this article, we will concentrate on the porting activities of CAN functionalities. The exact kernel version we want to port our board support to is 2.6.31. Since kernel version 2.6.25, the SocketCAN framework is merged into the mainline kernel. This means our target kernel would not need any additional patches for basic SocketCAN software infrastructure. After having looked more deeply into the existing SocketCAN support in 2.6.31 we, however, discovered that the SocketCAN driver coverage was quite low. In 2.6.31 only several SJA 1000 chip based CAN devices were supported. To get support for our MSCAN based device, we had to take the driver from the main tree of SocketCAN [1]. Since the main development of SocketCAN is partly unsynchronized with the Linux kernel development, we would have to port the MSCAN driver to the 2.6.31 kernel.

As mentioned above, the development of SocketCAN is hosted separately from the kernel. However, it was simple to strip the latest mscan driver code out of the SocketCAN archive. Only minor changes to the code were needed to fit the network infrastructure in kernel 2.6.31. The mscan driver was ported quickly, so that we could push the iteration forward to the test phase. We will give a more detailed view about the tools and strategies we used in the test phase in the following paragraph. However, after the initial tests we discovered several unconventional behaviours of the controller. So we had to repeat our iteration. After analyzing the driver code, we found some bugs, which were apparently unknown to the community until then. So we proceeded to work on fixes for those bugs. During this process we were able to benefit from the experiences in working with the SocketCAN framework and the open source development model: Unclear points were discussed in public. We were able to locate bugs quickly in such open discussions.

Code changes were reviewed by the community and tested on platforms of other developers.

Fixes and enhancements were collected and applied to the main development tree, so the joint efforts did not get lost.

After approval in the SocketCAN community of our rework on the mscan driver we sent the driver directly to the Linux Kernel mainline. The mscan core driver is accepted for the ▶

release cycle of version 2.6.33. Some more SOC bindings to this core will be following in the coming kernel versions.

The basic testing criteria for the CAN network is as simple as "can communicate on CAN bus". We divided this into three sections:

- Sending/receiving CAN messages between two boards connected to the same bus
- Bus error handling
- Configuration of various parameters like bit-rate

For sending and receiving we used the package can-utils. It contains simple utilities, which are capable to send customizable CAN messages or to wait for incoming CAN messages and dump the content of received message to the display. Further we used the `iproute2` [2] package to change configuration parameters of our device. More details about the configuration interface of SocketCAN will be described in the next chapter. With these tools we can initiate communication on our bus and verify the content. Turning off single bus entries or change run modes during runtime can manually produce bus errors. We made some simple scripts to automate the test procedure in different variations for long durations.

Second iteration: User space

The main goal of the second iteration of our development cycle is the user-space part, which mainly consists of porting our existing customer application and its own software stacks. Most parts of basic system access functions can be left untouched. The CAN related call-backs, however, have to be adapted to the SocketCAN interface.

Aside from simply receiving and sending, a CAN device driver must be capable of doing certain tasks ▶

like setting up bit-timing/bitrate, (re)initializing the error counter, setting up the receiving mode etc. To access these functions from user space, the SocketCAN device layer provides certain interfaces. These interfaces have been changed during the development of SocketCAN. In very early versions of SocketCAN, the system call `ioctl` [3] was used for this purpose. Later a generic `sysfs` [4] interface replaced the `ioctl()` calls. In the process of merging SocketCAN into the mainline kernel, the netlink [5] mechanism took over the role of `sysfs` and became the final solution for configuring CAN devices. Netlink is a socket based IPC and is used for transferring information between kernel and user-space processes. As it has proven to be very flexible in other networking subsystems, it was also adapted to SocketCAN. The requirements for this iteration will be thus briefly "Porting `ioctl()` to netlink based communication".

As usual for the development of an open source project, we started our requirement analysis by looking for existing sample code for CAN specific netlink communication. A good start would be e.g. source code of existing tools, which are capable of doing CAN related configuration tasks. It turned out that the only existing CAN compatible tool was `iproute2`. The CAN related code is however hooked deeply into the given infrastructure of `iproute2`, so that we have to at least handle the sending/receiving and creating/parsing of the netlink messages ourselves. To get an idea about the effort we need to put into the implementation, a closer look into the netlink infrastructure was needed before we could proceed to the next step in our iteration.

Basically, a netlink socket acts as a bridge between kernel space and user space. To use netlink, the user opens a socket with a special domain, type and protocol. Both kernel space and user space can then read from/write to the socket bi-directionally, this way status information and commands can be exchanged. The communication through netlink is so far straightforward. Parsing or creating a SocketCAN netlink message was, however, a non-trivial task. As an example, let's have a look into what we have to do to get the link information of a single interface: After opening a netlink socket, we send a netlink message with the type of `RTM_GETLINK` to instruct the kernel to send us back the link information. The required information is packed into a reply message, which is sent back to the socket. The payload in the reply message contains link information of all interfaces. The message consists of a message header and payload data. The payload data, again, has multiple layers. One will eventually get the data he wants after stripping down five layers into the payload data. Parsing and generating such messages can be a quite complex task. So we decided to create a library to deal with the CAN relevant netlink communications.

The basic ideas of our library are:

- Keep things simple to reduce the effort for porting the application
- Create a generic API so that we can also use the library for other projects
- Keep the memory footprint small

Based on the directives listed above we implemented three groups of functions in the API of the library:

- Do action: Certain actions like start/stop/restart the CAN device are implemented in this group.
- Set values: Functions in this group can be used to set configuration values like bit-timing, auto restart interval, control mode of the CAN device.
- Get values: Acquire configuration values.

All together, we have 14 functions in our API. A set of static functions is used internally for the basic communication with the netlink layer. The external functions share the internal callbacks as far as possible to avoid code redundancy. After finishing the library, the main effort for porting the customer application is replacing the legacy callbacks for CAN controlling functions with our library API.

We defined the customer application and our library as two separate test targets. For the application, the customer has his own qualification tool sets. To define a testing scenario for the library we picked the canconfig tool, which is contained in the canutils package [6]. canconfig provides functionalities to do configuration tasks on a CAN interface. Till then it was only capable with the obsolete `sysfs` interface of earlier SocketCAN infrastructure. We ported this tool to make usage of our new library. This way we have a handy tool to test the functionality and reliability of our library.

Conclusion

Using CAN in embedded Linux systems used to be a complicated topic because of missing standards. If one did not want to be bound to vendor dependent libraries, the remaining option was to write an own driver to implement the CAN related functionalities. Both solutions lead to high incompatibility and low portability of the project. With the SocketCAN framework, a first step was taken for platform independent CAN support in embedded Linux projects. Since the framework is meanwhile merged into the mainline kernel, it will be available on long term and receive high-quality maintenance. These are essential criteria for software frameworks in industry. As we could see in the example project, there is still work to do regarding drivers and access-functions. Nevertheless, as the situation constantly improves, we expect to see a lot more projects based on SocketCAN in professional fields.

info@pengutronix.de

Servo drive with CANopen

The book-sized Lexium 32 servo drive series by Schneider Electric is designed for space limited applications. It reduces machine footprint and decreases costs. With optional Memory Cards the user can parameterize servo drives without a PC within short time. Most of the models support CANopen and CANmotion communication protocols: The Lexium 32 Advance drives for motion centric applications have CANopen/CANmotion on board. The Lexium 32 Modular drives may be integrated into PLC or motion controller architectures regardless of supplier. (hz)

www.schneider-electric.com

References

[1] http://lxr.linux.no/#linux+v2.6.32/Documentation/networking/can.txt

[2] http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2

[3] http://www.opengroup.org/onlinepubs/009695399/functions/ioctl.html

[4] http://lxr.linux.no/#linux+v2.6.31/Documentation/sysfs-rules.txt

[5] http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto

[6] http://www.pengutronix.de/software/socket-can/download/canutils/