Socket-CAN DEVICE SUPPORT FOR EPICS IOCS*

C. Burandt†, U. Bonnes, J. Enders, M. Konrad, N. Pietralla,
Institut für Kernphysik, TU Darmstadt, 64289 Darmstadt, Germany

Abstract

This contribution describes an EPICS device support for CAN bus. It makes use of the Socket-CAN framework and is thereby independent from the API of a specific vendor. The device support has been used successfully in a production environment at the superconducting Darmstadt linear electron accelerator (S-DALINAC) since almost two years.

INTRODUCTION

CAN bus (Controller Area Network) has been chosen as the preferred field bus connecting in-house developed hardware to the S-DALINAC's accelerator control system. CAN bus is a rather robust communication bus. It allows a bit rate of 1 Mbit/s for distances up to 40 m. The underlying protocol is message-based. Every frame carries up to 8 byte of user data.

CAN interface controllers for personal computers are commercially available from different manufacturers, but although they all share the same basic functionality, most of them have a vendor-specific application programming interface (API). Moreover traditional CAN drivers are usually accessed by only one process at a time, which precludes the use of sniffer programs for debugging. In contrast to that the Socket-CAN network stack [1], included in recent Linux kernels, provides access to the CAN bus via network devices (BSD sockets). Those can be accessed by multiple applications at the same time via a vendor-independent interface. A set of open source CAN drivers provides access to controllers of different vendors.

For development purposes we use USB interfaces. They are easy to transport and can therefore be utilized to hook a laptop into a CAN bus segment for on-site diagnostics. The Linux computers running the IOCs (input output controller) are equipped with PCI/PCI express slot cards.

Since the S-DALINAC's accelerator control system is currently being migrated to EPICS [2], a CAN device support module has been developed to provide access to our devices from EPICS IOCs.

SOCKET-CAN

Opposed to a *character device driver* based *device support*, our solution here described relies on the Socket-CAN framework. The latter is included in the kernel main line since Linux kernel version 2.6.25, which is exceeded even by most of the conservative Linux distributions. On the one hand it brings along a growing number of CAN device drivers, on the other hand a network layer is implemented, which allows CAN interfaces to be treated in a similar fashion to ethernet devices [3]. The connection endpoint on the PC is represented by the Linux kernel as a BSD socket. The described functionality is combined in a so-called protocol family named `PF_CAN`. It is a common analogue of the `PF_INET` protocol family, which allows ethernet interfaces to be accessed with protocols like TCP and UDP. Both protocol families are provided by the kernel simultaneously. Figure 1 shows this architecture.

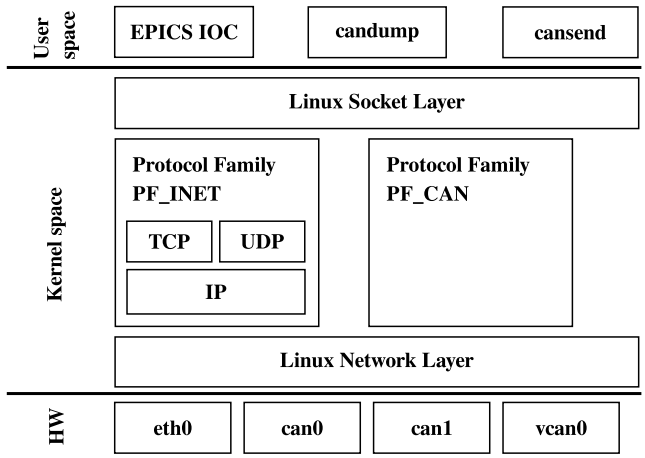

Figure 1: The Socket-CAN framework presents itself in the same fashion as the Linux kernel's network support. Different devices can be accessed by different applications at the same time. This is achieved by the network layer and appropriate protocol families.

According to this concept, a CAN adapter does not just show up as character device `/dev/canX`, but needs to be brought up by

```
ip link set can0 up type can bitrate 1000000
```

as usual with ethernet devices. A virtual CAN device can be configured easily using the `vcan` kernel module. This is useful when doing IOC development without a physical CAN interface.

A central feature of the Socket-CAN framework is to reflect its bus property on the PC. While in hardware several participants can be connected to a CAN bus segment, the same should be possible on the PC. The Socket-CAN network stack makes this possible. Therefore, multiple applications, *e. g.* several IOCs and diagnostic tools, can be run

* Supported by DFG through CRC 634.
† burandt@ikp.tu-darmstadt.de

ISBN 978-3-95450-124-3

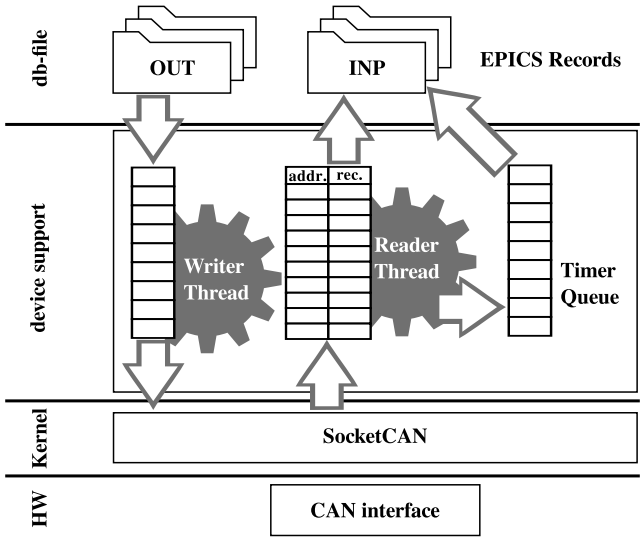

Figure 2: The device support keeps two queues. One is used by the writer thread as a buffer. The other one tracks the time which has passed since the last arrival of an incoming frame to allow a second thread to identify timeouts. The reader thread maps incoming frames to the different records. It uses an allocation table for this task.

on one PC and can use a single interface to connect to a CAN segment.

DESIGN OVERVIEW

Figure 2 shows the structure of the device support module. The detailed design of the device support depends on the capabilities of the different devices which will be connected to the CAN bus. Asynchronous communication with the devices is possible. Nevertheless, we decided to use polling, that is requesting all data periodically. The repetitive transmission makes sure the information is never totally outdated or changed unnoticed by the control system. Polling allows broken devices, which do not respond anymore and that cannot even signal errors, to be identified quickly. A timeout can be triggered when a certain device does not send frames anymore. A feature shared by many of the in-house developed devices is the ability to transmit certain values periodically without the need for repeated requests. This makes the *polling with timeouts* approach even more convenient.

Since CAN frames may carry more than one piece of information, the device support needs to be able to distribute them to multiple records. A record table holding the mapping between the CAN address IDs and the corresponding input records is created during the IOC startup process.

All records sensitive to timeouts are stored in a separate table. Writing to the bus is handled by a queue. It holds all the frames intended to be sent on the bus. Multithreading allows each table/queue to be processed with equal priority.

DEVICE SUPPORT MODULE

Software Engineering Features

The Socket-CAN device support has been written in plain C. About 1300 lines of code are necessary to interface the Linux kernel and to deliver the data to different types of EPICS records.

Two build dependencies exist apart from some standard C library header files and from EPICS header files. The headers, describing the Linux kernel's Socket-CAN network stack, are needed. This is nowadays trivial, since it is part of the kernel main line. The second dependency concerns the firmware embedded in all devices that are equipped with a CAN interface. This header file defines the addressing scheme which is used for sending messages to specific devices.

There is no need for any include file specific to the CAN interface manufacturer. Therefore every CAN interface which is supported by the Linux kernel's socket-CAN framework can be used with this device support.

The IOC has to supply the name of the interface which is supposed to be used. During IOC initialization the `INP` fields of all records using Socket-CAN device support will be parsed. Every interface listed in one of these fields is being connected to and listened for incoming frames.

On arrival of a matching frame the requested bytes are extracted from the data frame and interpreted within the meaning of C data types. Subsequently they are handed to the record support for further conversion.

User Features

The Socket-CAN device support currently supports the basic record types:

- analog in/out
- binary in/out
- multi binary in/out
- multi binary direct in/out
- long in/out

There was no need for the more complex record types yet, but in principle these could be implemented also.

Template Development

The following is a showcase for a definition of a record which uses the Socket-CAN device support:

```
record(ai, "rf:rdSupplyVoltADC") {
  field(LINR, "SLOPE")
  field(ESLO, "9.3e-9")
  field(EOFF, "0")
  field(EGU, "V")
  field(SCAN, "I/O Intr")
  field(DTYP, "tudSocketCan")
  field(INP, "@can0 07 01 06 519 02 1 sl 50")
}
```


ISBN 978-3-95450-124-3

The field DTYP makes the record use our Socket-CAN device support. To define a valid INP field, the template designer needs to collect certain information about the device the record is meant to communicate with. The line starts with the interface which is going to be used (can0). The further properties are as follows:

- 07: marks the CAN frame as being sent from a device (as opposed to frames being sent to a device)
- 01 06: address of the device
- 519: defines the specific request, in this case some ADC's value is being transmitted
- 02: specifies the ADC (there are more than one)
- 1: omit the first data byte, when converting the value, since this is the one used to identify the ADC
- sl: interpret as a signed long number
- 50: timeout in seconds

Usually the address is defined as a macro configured in a substitutions file. The number which specifies the request (519 in this example) is taken from a header file belonging to the firmware running on the devices. The user can use mnemonics to specify the request, and a perl script will substitute those automatically with their numerical code.

Socket-CAN Tools

A significant advantage of the Socket-CAN approach is testability. The Socket-CAN project provides a set of basic command-line tools. The following tools have proven as very convenient during template development:

- cansend: send a single CAN frame to the specified interface. Address and data content are given in hex notation and can be arbitrarily chosen.
- candump: dump the traffic of one or many CAN interfaces to stdout. Precise timing information can also be obtained. Filtering can be achieved by specific command-line options or piping to a grep command.

Considering that the CAN bus uses a message-based protocol, one can easily monitor the traffic between the IOC and the hardware. Utilizing caput and camonitor additionally, the whole chain from the CAN bus to the operator interface can be evaluated on a minimal setup.

FUTURE WORK

The current implementation has been used successfully at the S-DALINAC for almost two years. Still some minor issues remain unresolved.

The architecture of an EPICS IOC with its scanning implementation can lead to bursts of CAN frames sent to the microcontrollers. These can cause overruns of the receive buffers of the devices. Since dealing with such an issue needs changes deep inside the device support, a complete rewrite in the C++ programming language is being considered. The object-orientated design of this language plus a large set of functionality provided by the Standard Template Library (STL) would allow for a much cleaner design, from a software engineer's point of view. The table which holds the connecting information between the EPICS records and the corresponding CAN messages is currently implemented as a static array. During a rewrite this would be replaced by a dynamic data structure which consumes only as much memory as needed. Additional functionality could also be implemented, for example the logging of bus error frames to the IOC log or to a certain record.

The extension of the supported record types is possible. Some of our hardware components can send or receive strings. Thus string in, respectively string out, device support is needed. As long strings can require more than one CAN frame to be transmitted, this proves to be different from the already implemented records. The same applies to waveform records which could be used to transfer larger binary data sets.

CONCLUSION

The presented device support has been in use at the S-DALINAC for nearly two years. In particular the low-level rf control system relied on this module [4]. Since then it proved to be quite suitable for the given task. However the effort of a rewrite may be reasonable.

The characteristics of the device firmware are met well and allow an easy definition of large numbers of records, that share a pattern in nomenclature and the same firmware request.

The concept is simple enough to allow studends to write template files for yet unsupported firmware features after a short time of familiarization.

ACKNOWLEDGMENT

Important advice for a successful implementation of the Socket-CAN device support from members of the control-system group at BESSY is gratefully acknowledged. This work has been supported by DFG through CRC 634.

REFERENCES

[1] The Socket-CAN project at berliOS.de http://developer.berlios.de/projects/socketcan/

[2] C. Burandt et al., "Status of the Migration of the S-DALINAC Accelerator Control System to EPICS," PCaPAC'12, these proceedings.

[3] Readme file for the Controller Area Network Protocol Family http://www.kernel.org/doc/Documentation/networking/can.txt

[4] M. Konrad et al., "A digital base-band RF control system," ICALEPCS'11, Grenoble, October 2011, MOMMU012, p. 82, http://www.JACoW.org

ISBN 978-3-95450-124-3