QEMU CAN Controller Emulation with Connection to a Host System CAN Bus

Pavel Pisa
Czech Technical University in Prague
Technická 2, 121 35 Praha 6, Czech Republic
pisa@cmp.felk.cvut.cz

Jin Yang
University of Chinese Academy of Sciences
ShenYang Province, China
jinyang.sia@gmail.com

Michal Sojka
Czech Technical University in Prague
Technická 2, 121 35 Praha 6, Czech Republic
sojkam1@fel.cvut.cz

Abstract

Fast, easy and frequent/continuous testing is critical for embedded operating systems and applications development. The QEMU system emulator provides a solution to easily test developed operating systems kernels and drivers on multiple emulated architectures and board models. But emulation of automotive and industrial control buses is not available in QEMU yet. Presented project implements CAN controller emulation for QEMU. The emulated controllers can be connected to another ones inside the QEMU instance or can be connected to the host system virtual or physical CAN bus network if GNU/Linux is used as QEMU host system. The PCI based CAN bus interface card was selected as the first target for our work because it can be easily attached to all QEMU architectures that support PCI/PCIe.

1 Introduction

System level virtualization is technique used massively for server and cloud services but same solutions can help significantly in applications, operating systems and drivers development as well. QEMU (together with KVM) is used on servers but the same project provides ground for emulation of client system architecture different from a host system one. QEMU is base of Android SDK where it is used to emulate hardware of thousands of different Android system based devices and mobile phones to ease applications cross-development.

The QEMU provides quite broad and precise emulation of many architectures and computer boards used in embedded control systems but support for development of industrial and automotive communications and input/output peripherals and cards is quiet limited till now. But testing or even continuous integration and automated testing is important for development of such systems.

The presented project fills a gap and add CAN bus and CAN controller emulation to QEMU. CAN (Control Area Network) is bus which has been developed by Bosch company and is probably present in all modern cars and in many industrial applications now. Preparation of environment for CAN drivers development for RTEMS operating has been initial impulse for this project creation. Intention has been to test generic CAN bus code for more CPU architectures and for that reason generic pluggable hardware has been selected instead of specific SoC controller.

The SJA1000 controller has been selected to be emulated because it is classic well knows and quite often used chip. It has been integrated in QEMU as simple PCI memory mapped device to allow it easy combination with different computer systems (QEMU machines). Then emulation of Kvaser PCI CAN card with I/O mapped SJA1000 has been added to allow run unmodified mainline Linux kernel with appropriate drivers as guest and compare emulation with real hardware which we use in many of our another projects.

Support to connect QEMU virtual CAN buses to hos system CAN infrastructure has been implemented because remote side/devices are required to establish and test CAN communication and complete applications. This solution works with SocketCAN (standard Linux CAN infrastructure) on host side for now. The emulated CAN bus can be connected to real CAN interface or software only VCAN network device. The mix of real physical devices and emulated ones running on host system can be communicated with from guest system then.

Options to emulate CANopen industrial control protocol communicating devices by our OrtCAN project software and combined emulation of COMEDI drivers supported data acquisition cards to mimic complete industrial environment for guest system is described as well.

The last section discusses possible future project development directions – option to emulate SoC integrated controllers and extend developed infrastructure to support CAN FD (flexible datarate) controllers emulation. Many CAN based projects needs to prepare for CAN FD technology upgrade but the hardware is now rare, expensive and its combination with different CPU systems can be impossible in real word due to connectors and CPU address and data buses cannot be adapted as easily as QEMU software ones.

2 Project Background

Project originates in interest of RTEMS[1] operating system community to develop generic CAN infrastructure for this system. The intention has been formulated as one of Google Summer of Code (GSoC) 2013 topic. When student Jin Yang applied for this project, discussion about target architecture and platform started. RTEMS supports many target systems but many of them are extremely expensive and or hard to obtain (e.g. Aeroflex GR712RC SPARC). There exist target specific emulators with CAN controller emulation included (for example for the mentioned GR712RC it is TSIM) but RTEMS community runs periodic check of system builds for most architectures under more widely available QEMU and Skyeye emulators. Because these emulators did not include CAN bus support and RTEMS project maintainers have preferred to have option to test results without need of testing on specific hardware, it has been decided that an RTEMS GSoC slot would be assigned to work on extending QEMU to provide test bench for future RTEMS CAN related projects development and the actual development of CAN bus generic infrastructure has been postponed.

Because x86, PowerPC, ARM and SPARC are RTEMS most important supported architectures and CAN controllers integrated on SoC vary or are not present (e.g. in case of x86), it has been decided to start with controller which can be connected to at least one platform/machine for each architecture. There exists machines with PCI bus support for each of these architectures and that was why implementation of virtual PCI card with widely used NXP SJA1000 controller has been chosen.

[1] https://www.rtems.org/

3 Implemenation

Figure 1 depicts the CAN bus related virtual components and their connection to the host system hardware and applications running on host and guest systems. Real hardware and physical/real CAN bus can be seen at the top. Real hardware is available to host system user-space programs through SocketCAN infrastructure as network socket of CAN protocol/address family (AF_CAN) when QEMU is run on Linux kernel based host system. QEMU runs as user-space program on the host system and emulates virtual CAN controllers hardware (QEMU devices) which are seen as PCI devices by the guest operating system kernel. The individual emulated controllers can be interconnected together in groups representing virtual can buses. The group membership is specified by `canbus` device parameter. The virtual CAN bus can be connected to the host system one when host system network device is specified for one of configured QEMU CAN devices (`host` parameter).

QEMU represents emulated hardware components by QEMU Object Model (QOM) which is self based on GLib Objects (GTK+/GNOME origin). The emulated devices are represented by Device objects (QDev structure DeviceState) which are connected to buses (structure BusState). Object

PCIDevice inherits from QDev to represent peripherals connected to the emulated PCI bus.

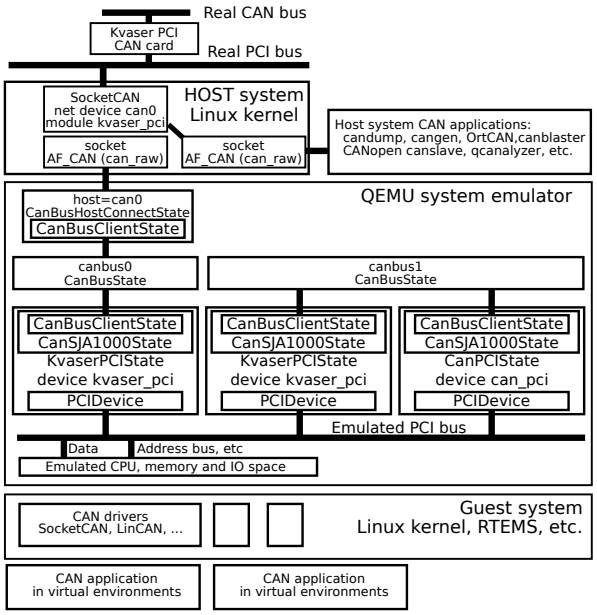

FIGURE 1: *CAN Buses and Hardware Architecture in QEMU*

Actual CAN controller is usually accessible as set of registers which can be mapped into computer systems memory address space, represented as I/O ports or even hidden behind indirect access through externally visible index and data registers. An operating system device driver configures controller, prepares message identifier, parameters and data by writing to the registers and then marks message as ready to be sent by write to some control register. The reception of a message is usually signaled to CPU as interrupt activation and then driver uses read operations to obtain message content and write to confirm that next message can replace buffer holding read message data. There are alternatives to use DMA or bus-mastering for more complex controllers but register read/write and interrupts signaling is all what is used by SJA1000 controller which has been selected as the first target of the project.

The NXP SJA1000 CAN controller is standalone chip which is used in many systems. There exists ISA, PCI, PCIe cards based on this chip and it has been often connected to local bus of SoCs without CAN controller support as well. Because mapping of the chip registers to buses can differ from system to system the chip registers and functionality emulation has been implemented separate from the actual integration to the system and mapping on the system bus. The SJA1000 chip state is represented by CanSJA1000State structure and provides only minimal set of functions – can_sja_mem_read(), can_sja_mem_write() to manipulate chip state from a emulated CPU side. The corresponding source code is located in the qemu/hw/can/can_sja1000.c file. The mechanism to signal interrupt is generic as well, irq_raise and irq_lower parameters of can_sja_init() are used to supply actual implementation for interrupt handling for concrete chip connection to the system.

It is necessary to deliver message to other controllers in the group (connected to the same CAN bus) when it is ready to be sent. The connection of the CAN controller to virtual CAN bus is represented by CanBusClientState structure which can be embedded to the object representing controller (CanSJA1000State for SJA1000). The list of these structures is attached to the object representing virtual CAN bus (structure CanBusState). The client notifies bus than it sends message by calling can_bus_client_send() function. CAN bus generic code iterates over CanBusClientState structures. An client method can_receive() is called to check if client is active and reception is enabled. In the case of positive reply, receive() method is called. These methods are specified in CanBusClientInfo for each CanBusClientState. When bus client/controller specific receive() is called, CAN message content represented by structure qemu_can_frame is copied to internal controller buffers and irq_raise() is called if interrupts are enabled in appropriate controller register. The respective code can be found in files qemu/hw/can/can_core.c and qemu/include/can/can_emu.h.

The bare SJA1000 chip emulation offers only functions to access registers and actual mapping to the computer bus is left on the wrapper objects which includes CanSJA1000State and derives from QOM object specific for the bus integration. Simple but complete PCI card/device (in QEMU meaning) which maps SJA1000 to single memory region (specified by base address register BAR0) has been implemented first. The region operations (MemoryRegionOps can_pci_bar0_read and can_pci_bar0_write) maps directly to the SJA1000 chip read/write operations and only byte size access is implemented (wider access is emulated by QEMU infrastructure). Interrupt connection is directly represented by qemu_irq object embedded in the CanPCIState CAN card structure and SJA1000 interrupts emulation is routed to core QEMU qemu_irq_raise() and qemu_irq_lower() functions. The card state contains and initialization fills PCI device header. Arbitrarily chosen PCI Vendor/ID combination is used for now (1af4:beef).

The functionality of simple PCI SJA1000 card can controller emulation has been tested by LinCAN driver. LinCAN driver is simple but low-latency character device based driver for Linux kernel and its extension to support selected PCI device identification and SJA1000 chip mapped directly to PCI device memory range has been straightforward. (Note: we are not aware of real PCI card/device with this simple SJA1000 mapping so we cannot use some standard identifications nor unmodified driver).

The single, dual and quad Kvaser PCI CAN controllers cards are used by our CTU Industrial Informatics group in more projects where we need to connect CAN bus to PC based systems. Because we know that hardware well and even implemented LinCAN driver support for it and Linux kernel mainline SocketCAN driver `kvaser_pci` is based on our code, the emulation of this board has been next target for our QEMU CAN bus emulation project.

There are one, two or four SJA1000 CAN controller chips present on the Kvaser PCI CAN card. The interfacing of the chips to PCI bus is realized by AMCC S5920 PCI bridge which collects and gates interrupts. The chips are mapped to an I/O space region controlled by BAR1. An region 0 is occupied by the bride control registers and region 2 contains registers to identify card flavor, number of populated channels and simulate interrupts. The card vendor/device PCI identification is 10e8:8406. Mapping of the SJA1000 chip (single channel version) to the I/O space is straightforward – functions kvaser_pci_sja_io_read() and kvaser_pci_sja_io_write(). The IRQ processing takes in account local gating/enable state in S5920 register and stores state for readback. Implemented emulation works with unmodified mainline Linux driver as well as with LinCAN driver.

The implementation is maintained to be compatible with actual stable QEMU versions. Branch *can-pci* of CTU IIG QEMU repository [2] points to the actual supported CAN emulation version. Branches named as *merged-X.Y* point to the merge of several locally developed branches, including *can-pci*. Actual version for QEMU-2.4 can be found on a branch *merged-2.4*.

4 Running QEMU with CAN Bus

QEMU `-device` parameter inserts non default machine hardware devices emulation into QEMU system level emulator for specific architecture and machine – i.e. `qemu-system-x86_64` for PC compatible 64-bit computer. For simple directly mapped CAN PCI card parameter `pci_can` device needs to be added. Parameter `canbus=` can be specified for given device to select into which virtual bus is device connected. If a bus is not specified, default bus name `canbus0` is assumed. Parameter `host=` allows to specify connection of the virtual CAN bus to CAN network device on the host system. SocketCAN on Linux is only supported by actual version. Usual host system CAN network device name is in form `can0` for real controllers and usually `vcan0` for software only host side bus. Optional parameter `model=` is included to allow select can controller chip connected to PCI card. The `SJA1000` is only supported one for now. The Debian amd64 distribution installed as host operating system as well as guest operating system on disk image is assumed for next examples.

4.1 Virtual x86 Native Targets

Next command sequence configures the host side virtual CAN bus network

```
modprobe can-raw
modprobe vcan
ip link add dev vcan0 type vcan
ip link set vcan0 up
```

The command line that starts QEMU with single internal bus connected to host-side vcan0 interface is:

```
qemu-system-x86_64 -enable-kvm \
-kernel vmlinuz-3.16.0-4-amd64 \
-append "root=/dev/sda1"
-hda disk-image \
-device can_pci,model=SJA1000,\
canbus=canbus0,host=vcan0
```

The LinCAN driver then can be compiled on guest system and loaded to access virtual CAN bus controller:

```
insmod lincan.ko hw=pcisja1000mm io=0
```

Next example demonstrates how to connect two Kvaser CAN PCI single channel cards to the target system and interconnect them with Kvaser card on the host system. SocketCAN on the host system is configured like this:

```
modprobe can-raw
modprobe kvaser-pci
ip link set can0 type can bitrate 1000000
ip link set can0 up
```

Virtual machine is started by command:

```
qemu-system-x86_64 -enable-kvm \
-kernel vmlinuz-3.16.0-4-amd64 \
-append "root=/dev/sda1"
-hda disk-image \
-device kvaser_pci,canbus=canbus0,host=can0 \
-device kvaser_pci,canbus=canbus0
```

Notice, that the specification of connection to the host system CAN bus is not repeated for the second device on the same bus. Multiple connections of virtual bus to the host level would lead to infinite message looping between QEMU and the kernel. The same mainline driver is used for guest Linux system:

```
modprobe can-raw
modprobe kvaser-pci
ip link set can0 type can bitrate 1000000
ip link set can0 up
```

This way, applications running in the guest system can access real CAN devices connected to host computer.

Functionality of the setup can be verified with SocketCAN utilities. CAN messages can be generated on the guest side by:

```
cangen can0 -e -I 123 -g 1000 \
  -D 11223344DEADBEEF -L 8
```

and delivery checked on host side by:

```
candump vcan0
```

To check the communication in the opposite direction, cangen and candump can be swapped but bus names `vcan0` and `can0` should be preserved.

The QEMU can be run without graphics support and guest kernel output can be passed to host system through serial port which appears on QEMU process standard output on the host side.

```
-nographic \
-append "root=/dev/sda1 console=ttyS0"
```

This setup eases automatic testing quite well.

4.2 ARM Targets

Next configuration has been used to test drivers on virtual ARM platform with Raspbian distribution.

```
qemu-system-arm -cpu arm1176 \
  -m 256 -M versatilepb
```

Cortex based systems (`realview-pbx-a9` or `vexpress-a15`) can be used with Debian armhf distribution. `xilinx-zynq-a9` is very interesting machine as well but it does not support PCI in actual QEMU versions. Another promising but not tested option is `virt` machine which hardware structure is configured by device tree.

4.3 CANopen and Industrial I/O Devices

The setup with multiple CANopen devices connected to the bus CAN bus can be setup easily. OrtCAN projects provides program `canslave` which can mimic real CANopen device when appropriate Electronics Data Sheet (EDS) describing device is available. The program reads EDS file and builds device object dictionary during startup. It even allows to connect selected objects from CANopen object dictionary (OD) to the data/variable objects provided by dynamically loadable libraries. This way it is possible to connect emulated devices to controlled plant process model environment. Connection of CANopen device emulator to the host system software only/virtual CAN device.

```
./canslave -d socketcan:vcan0 -n 5 \
  -g 6 -e device-ds-401.eds
```

Parameters selects CANopen node ID 5, higher level of debugging output and loading of specified EDS file. More such devices for different CANopen nodes with same or different CANopen profiles can be started this way.

The provided merged-2.4 branch of QEMU includes emulation of data acquisition/control PCI card Humusoft MF624. The emulation includes digital and analog inputs and outputs sections. An COMEDI driver for this hardware is included in Linux kernel mainline already. The external analog and digital signals “terminals” are accessible as socket listening on specified TCP port on the host side of QEMU. When the port is connected then simple text protocol informs about signals state/levels changes which are controller by application running on the guest system. Digital signals inputs and “voltage” levels reported by emulated ADC channels to guest system application are controlled by sending text commands to the port. QEMU parameters:

```
-device mf624,port=55555
```

More detailed description can be found in [6]. CAN communication, COMEDI based I/O cards and direct UIO based Humusoft MF624 is included in ERT Linux Matlab/Simulink target as well [7].

5 Possible Future Enhancements

The SJA1000 controller implementation is minimal for now. FIFO to hold more incoming messages should be implemented in the chip emulator to be inline with a real chip.

There is not implemented mechanism to slow-down messages rate and reflect time require to deliver message on real CAN bus in virtual environment as well as there is no mechanism to generate errors and bus state transitions to passive and off mode if only one emulated CAN controller hardware is configured and active on the bus.

Some mechanism should be added to prevent lost of messages when application on emulated machine is not processing incoming messages fast enough. Some real word CAN controllers can generate overload frames to solve such situation. This mechanism can be emulated by delay in delivery of messages through the bus.

The virtual CAN bus model should be probably converted from plain "C" structures to QOM model as well to allow complete system freeze and thaw in new QEMU program instance/processes. The requirements to support state store has not been investigated for the bus. The SJA1000 and emulated cards state store is included but has not been tested.

PCI CAN cards emulation allows to use same emulated hardware to test applications in different architectures environment. But SoC integrated CAN controllers prevails in embedded systems today. Emulation of BOSCH and Texas Instruments used CAN controllers found on BeagleBone AM335x would be more alone with this trend.

CAN FD (Flexible Datarate) standardization processes finalization is now ongoing. Linux kernel/SocketCAN generic support and initial controllers support is available now. But hardware is rare and it would take years before all required application can be adapted for this new standard. Yet availability of easily accessible emulated/virtual CAN FD controller can simplify the situation.

6 Conclusions

The presented project provides simple but working environment for CAN applications development and testing in virtualized system. Code has been successful maintained for two years to be compatible with stable QEMU releases. The basic infrastructure, which allows to add emulation of more CAN controllers, is provided.

The functionality has been tested for x86 native and ARM foreign guest CPU architectures.

Project is available in a public repository on GitHub [2].

Acknowledgment

The first version of code been funded by Google Summer of Code 2013 on base of RTEMS organization slot.

References

[1] "QEMU project documentation" [Online]. Available: http://wiki.qemu.org/Manual

[2] "QEMU with CAN emulation on GitHub" [Online]. Available: https://github.com/CTU-IIG/qemu branch:can-pci

[3] Jin Yang, "GSoC 2013 CAN for QEMU" [Online]. Available: https://github.com/Jin-Yang/QEMU-1.4.2

[4] "eLinux CAN documentation" [Online]. Available: http://elinux.org/CAN_Bus

[5] "OCERA Real-Time CAN" [Online]. Available: http://ortcan.sourceforge.net/

[6] P. Pisa, R. Lisovy, "COMEDI and UIO drivers for PCI Multifunction Data Acquisition and Generic I/O Cards and Their QEMU Virtual Hardware Equivalents", in *13th Real-Time Linux Workshop*, OSADL 2011

[7] M. Sojka, P. Pisa, "Usable Simulink Embedded Coder Target for Linux", in *16th Real Time Linux Workshop*, OSADL 2014