Xvisor VirtIO-CAN: Fast Virtualized CAN

Jimmy Durand Wesolowski[1][2], Aymen Boudguiga[2], Anup Patel[3],
Julien Viard de Galbert[1], Matthieu Donain[4], Witold Klaudel[5] and Guillaume Scigala[1]
[1]OpenWide, 23 Rue Daviel 75013–Paris, France, *firstName.lastName[-lastName2]@openwide.fr*
[2]IRT SystemX, 8 avenue de la Vauve 91120–Palaiseau, France, *firstName.lastName[-lastName2]@irt-systemx.fr*
[3]Individual Researcher, Bangalore, India, *anup@brainfault.org*
[4]PSA Peugeot Citroën, Route de Gisy 78140–Velizy-Villacoublay, France, *matthieu.donain@mpsa.com*
[5]Renault, 1 avenue du Golf 78288–Guyancourt, France, *witold.klaudel@renault.com*

***Abstract*—Nowadays, vehicles are embedding more and more electronics to support new functions such as driver monitoring, lane keeping and adaptive cruise control. However, adding electronics makes vehicles more expensive. Fortunately, virtualization, via a hypervisor, reduces the number of embedded chips in vehicle by running different guests, i.e. Operating Systems (OSes), offering several services on the same board.**

As the communication between embedded controllers is compulsory for vehicles to function, an optimized virtualization of the Controller Area Network (CAN) bus becomes mandatory. CAN bus virtualization is challenging as it has to tackle the CAN arbitration mechanism and to provide CAN frame broadcast in a transparent manner. In this paper, we use the VirtIO virtualization interface with a virtual CAN service and framework to manage virtualized system external and internal CAN messaging.

***Index Terms*—Embedded Systems, Controller Area Network, Virtualization, VirtIO, Xvisor**

I. Introduction

By the end of the last century, transportation systems and especially vehicles replaced some of their mechanical functions by electronically controlled applications. Such applications include automatic control of windows, fuel injection supervision and automatic activation of car headlights. By the end of 2010, cars relied on software containing millions lines of code on 70 to 100 microcontrollers, namely *Electronic Control Units* (ECUs) [1], [2]. Nowadays, the customers needs for new Advanced Driver Assistance Systems (ADAS) services are increasing and so is the total number of embedded ECUs in the vehicles [3]. In fact, proposing new services for drivers on the road implies embedding more electronic boards, more communication buses, more cables and more software, ending up by increasing vehicle complexity. Consequently, car price is rising with respect to the number of its embedded ECU while it has to stay competitive on the market in order to attract more customers.

One interesting solution to reduce the number of vehicle embedded ECUs consists in using *virtualization* via a *hypervisor*. Virtualization allows to run different *virtual* automotive Operating Systems (OSes), called *guest OSes* or *guest systems*, simultaneously over one single physical board. Virtualization defines the hypervisor as a software layer (called *host*) between the *guest* OSes and the *real* hardware. The hypervisor emulates the hardware board for each *guest system* (Figure 1). In addition, it manages resources sharing between all the guests. However, virtualization comes with a major drawback which is computation overhead. As virtualization introduces the hypervisor layer between running guests and the board, it naturally induces a delay for accessing the hardware. That is, a virtual guest takes more time to access the memory, the network interfaces and the CPU than a classical OS running directly on the hardware. In fact, every time a virtual guest needs to access a resource, it has to pass through the hypervisor which introduces some overhead. Reducing this computation overhead is really important especially for communication scenarios. In practice, vehicle ECU exchange real time data frames through the Controller Area Network (CAN) bus [4]. When virtualization is used, CAN bus virtualization overhead must stay as low as possible.

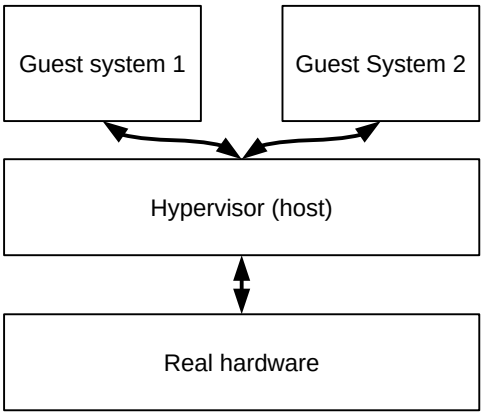

Fig. 1. Bare metal hypervisor

In this paper, we introduce an optimized virtualized CAN device. We use VirtIO API [5][6] to implement it for our hypervisor namely Xvisor [7].

The remainder of this paper is organized as follows. Section II reviews the CAN specification and introduces VirtIO and Xvisor. Section III depicts our solution. Section IV concludes the paper.

II. State of The Art

In this section, we give a brief description of the CAN protocol. Then, we introduce VirtIO and Xvisor.

A. Controller Area Network

The Controller Area Network (CAN) connects Electronic Control Units (ECUs) via a broadcast bus [4]. Each ECU is in charge of controlling multiple actuators or sensors. In mid-range cars, ECUs are used for Adaptive Cruise Control (ACC), fuel injection supervision, anti-lock braking system and comfort services management. ECUs communicate by exchanging CAN frames. These frames have one of the following types:

- *Data* frames exchange data between ECUs.
- *Remote* frames request the transmission of a specific *Data* frame.
- *Error* frames indicate the presence of an error over the CAN bus.
- *Overload* frames add an extra delay before the next frame transmission.

CAN frames do not contain information about their sender (i.e. source) or receiver (i.e. destination). They do not rely on interfaces addressing. In practice, each ECU manages a limited set of unique and distinct frame *identifiers*. A frame identifier defines an action that must be taken by the frame receivers.

CAN bits are encoded using a Non-Return to Zero (NRZ) code where 0-bits and 1-bits are encoded with different non-null voltage. The value of each transmitted bit is sampled at the end of a *nominal bit time* i.e. a bit time slot. In practice, CAN 0-bits are generally referenced as *dominant* bits and 1-bits as *recessive* bits.

CAN relies on Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) with a *bitwise arbitration* as bus access method. The bitwise arbitration concerns the value of frame identifiers. When two ECUs start the delivery of two different frames at the same time, the ECU sending the frame with the greatest identifier value stops its transmission at the reception of a dominant bit while it is transmitting a recessive bit.

B. Virtual Input Output

Virtual Input Output (VirtIO), is a virtualization abstraction API. It was created by Paul "Rusty" Russel to create a common layer for most virtual devices. As such, it avoids the proliferation of virtualization techniques in drivers, whose code and execution is similar to each others [8].

We choose to use VirtIO for CAN interface virtualization for three reasons. On the one side, VirtIO defines a clear and flexible API with well defined and optimized transport abstractions, that fit our needs. On the other side, VirtIO API is becoming the virtualization standard for host-guest communication interfaces [9]. Finally, it is recommended and supported by the Linux community.

C. eXtensible Versatile hypervISOR

The eXtensible Versatile hypervISOR (Xvisor) is a *type-1 monolithic* embedded open-source hypervisor. A *type-1* hypervisor, also known as *bare metal hypervisor*, is a virtualization layer that runs directly on the hardware. Meanwhile *type-2* hypervisor is executed on top of an operating system. Xvisor is said to be *Monolithic*, as it has a common software for host hardware access, CPU virtualization, and guest IO emulation. Meanwhile, micro-kernelized software kernels contain basic hardware access and CPU virtualization, but they relies on a *management guest* to handle the other services (e.g. drivers, file systems, ...) [7].

By design, it has a fast interruption management. Benchmarks demonstrated a low overhead on both guest instruction execution and guest memory operations [10]. For example, a Xvisor guest yields an average of 2.550.336 Dhrystone Millions of Instruction Per Second (*DMIPS*), compared to 2.558.851 DMIPS for a native system. In addition, the Read-Modify-Write throughput on a guest is about 556,13 MB/s compared to 564,58 MB/s on a native system.

Xvisor allows having an emulated interface if needed, a paravirtualized one, or even a direct access to hardware when the resource is not shared.

To conclude, Xvisor flexibility suits well our need to use the already existing VirtIO interfaces, and to adapt it to create a new one for the CAN bus `VirtIO-CAN`.

III. The CAN VirtIO

In the following section, we first list the industrial contraints that must be fulfilled by our proposed solution, `VirtIO-CAN`. Then, we describe it in details.

A. VirtIO-CAN Prerequisites

The following list depicts the requirements regarding CAN virtual software:

1) The guest high level (userland) certified softwares must remain the same. Thus, the interface for the CAN must not change, i.e. it has to present a `SocketCAN` compatible driver for Linux guests.
2) The different guest systems can communicate not only together, but also indistinguishably with the external physical bus.
3) The access must be controlled. Some systems must not have access to or read from the virtual and physical buses.
4) The message priority must be supported.
5) Even guest-to-guest messages must go through the physical bus, as those frames can be used for monitoring or debugging, for example, during a diagnosis with an On-Board Board Diagnostics plug.
6) Xvisor interface for the CAN management must be simple.
7) The overall performance must be comparable, if not indistinguishable, from a non-virtualized (or native) systems.

B. VirtIO-CAN

In order to reach good performances, the CAN communication cannot be fully emulated, i.e. a *fully virtualized*. Full virtualization implies trapping and handling every operation on the device from the virtualized system, in a complex manner, to simulate the hardware expected behavior.

As CAN resources are shared, direct hardware access, namely *passthrough*, cannot be used. This would give all guests the complete access to the same hardware and registers in particular. A guest could then overwrite the controller configuration or the data set by another guest. In addition, it may perform conflicting operations that cause errors. In the worst case, it may render the whole controller or system unusable. Thus, direct hardware access can only be used for exclusive hardware access from one system, either the host or one guest.

Our considered solution, referred to by *paravirtualization*, uses a compromise between full virtualization and passthrough approaches. The guest driver provides only an interface on the host, with no knowledge of the underlying hardware. This results in a better overall performance and a lower complexity than emulation. Meanwhile, paravirtualization keeps the latter advantages of controlling the resource access.

We choose to achieve this communication through the VirtIO mechanism (presented in Section II-B). VirtIO defines a communication standard for the virtualization of multiple devices (console, network, disk, and soon graphical devices) over a variety of tranport medium (memory-mapped or PCI bus). It allows an identification of the device type, with the available features on the host and the guest side, and the configuration desired by the guest. High throughput data can be passed through VirtIO queues called *virtqueues*, and smaller specific data through the VirtIO *configuration read/write functions*.

VirtIO queues are designed for large data transfers. Their data footprint is at least 4 KB of memory with a queuing mechanism. CAN data frame has 29-bit long identifier and 64-bit long payload. Consequently, using VirtIO queues is inadequate because it introduces a large footprint and a cumbersome overhead. So, we decide to use the alternative data transmission mechanism: read/write configuration functions. That is, the guest access are done at specific offset on the host interface through VirtIO.

IV. VirtIO-CAN Implementation

In this section, we describe the implementation and mechanisms of the virtual CAN. First, we describe the initialization of the different layers. Then, we not only depict a frame transmission from a virtual system to the physical bus, but we also detail the reception process in the opposite direction. Finally, we present an internal frame transmission between guests on the same board.

A. Initialization Phase

1) Initializing the hypervisor driver: In order to avoid mixing the virtualization mechanism and the hardware management, the *hardware controller driver* is initialized and operates the same way without virtualization. Depending on the underlying material, CAN *mailbox* number varies. Note that a mailbox is a buffer designed to filter CAN frames with an identification mask. Each mailbox contains only one CAN frame. It is used as reception and emission buffer with a priority management. Mailboxes can be grouped to be used as First In First Out (FIFO) queues. Due to the design of the CAN bus, mailboxes assignment and setup is known and set in advance. We can define the priority of mailboxes and FIFOs. That is a FIFO contains a group of mailboxes with a defined set of priority.

FIFOs are set if the number of virtual-mailboxes is greater than the one available in the hardware controller. If the hardware does not allow setting FIFOs, the service will not be initialized to prevent misuse.

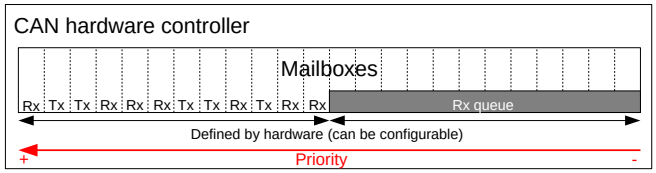

Fig. 2. CAN hardware controller system

2) Initializing the hypervisor service: We denote by `VirtCAN` the hypervisor service for the CAN framework. As such we avoid the ambiguity with the VCAN [11]. `VirtCAN` creates virtual-mailboxes with FIFO support to separate the hardware support from virtual system needs. It also provides a communication layer through the VirtIO framework to the guest system.

The host configuration sets the number of mailboxes and FIFOs to be created (Figure 3). The virtual-mailboxes are buffer slots of four 32-bit words each. The three first slots correspond to the CAN frame itself, and the last one serves for control and status storage (Figure 4).

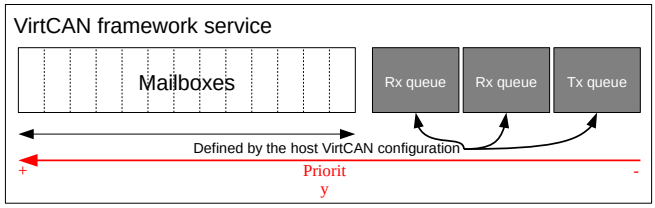

Fig. 3. `VirtCAN` system

Sending a frame on a particular mailbox returns an error to the guest if the latter already stores one that was not sent. A mailbox read twice without new matching frame also returns an error. However, by design a CAN messaging system avoids that.

The virtual FIFOs use the same buffer format (Figure 4), but frames are queued. A guest system can read or write a defined number of frames before an error is reported. In general, FIFOs are used for lower priority frame reception.

However, the VirtCAN also permits the creation of transmission queues.

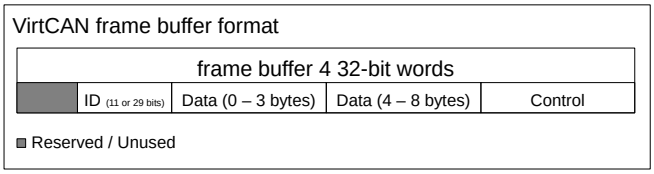

Fig. 4. VirtCAN framework buffer format

The hypervisor also has a configuration file for each guest to describe which and how a virtual-mailbox is used by this virtual system. Two guests cannot share the same virtual-mailbox to transmit or to receive a frame. However, they can have access to the same FIFOs. Each guest mailbox is a memory pointer to a virtual-mailbox.

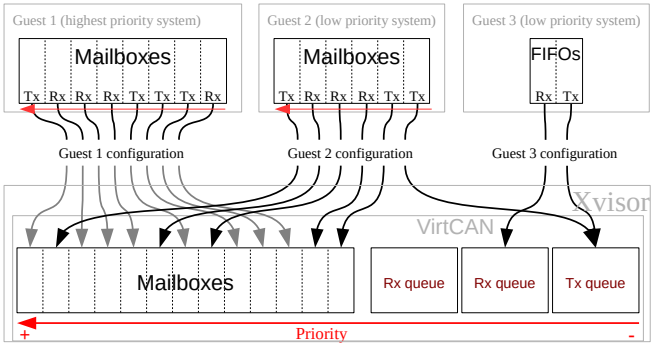

Fig. 5. Guest to VirtCAN mailbox association

3) Initializing the guest driver: The guest driver uses the VirtIO API to detect the VirtIO device and configure the required mailboxes and FIFOs. The hypervisor checks every guest setup to ensure its consistency.

B. Guest CAN Frame Transmission

Guests start transmitting at the end of the initialization phase. The VirtIO driver receives the frame from the higher layers (application, library, runnable, task,...). Then, it sends it to the hypervisor VirtCAN framework. The guest side VirtIO writes the frame at the offset formed by the addition of the mailbox or FIFO offset and an initial base offset for mailboxes. The latter is set by the VirtIO-CAN API.

The CAN frame identifier (ID) is written on a 32-bit word while the frame data are written on 8 bytes. Finally, the 32-bit control word is set. Note the frame payload can be less than the reserved 8 bytes. If a guest try to write a CAN frame with an unauthorized identifier, it is discarded and a warning is returned by the framework.

For example, to write a frame in its mailbox n , a guest has to write the 4 words of 32-bit in the VirtIO device at the offset: mailbox offset + VirtCAN buffer size × n

When the control word is written, the hypervisor VirtCAN service writes the frame to the CAN hardware controller. If virtual-mailboxes number exceeds the physical-mailboxes one, the excedent content is written in the highest priority transmit FIFO. The system integrator knows the number of excedent mailboxes as he knows the total number of the mailboxes of the guest, the VirtCAN and the hardware.

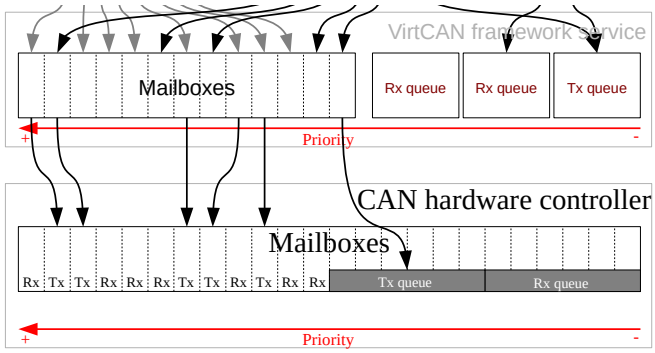

Fig. 6. VirtCAN mailbox to hardware frame transmission

C. Guest CAN Frame Reception

The frame reception process adapts the same idea of frame transmission of Section IV-B. When a frame is received by the hardware controller, it is copied in the virtual-mailbox matching the frame identifier. The number of physical-mailboxes can be lower than the total number of virtual-mailboxes. Then, the host must be able to receive all the frames. If no mailbox matches a frame identifier, it is queued in the highest priority FIFO. Note that the physical-mailboxes must be principally used for the higher priority frames. If all the reception FIFOs are full, or if there is no FIFO at all, the frame is discarded. In addition, an error message is returned.

Once a message is received in the VirtCAN layer, the service checks which guest is associated to this mailbox, write a *flag* corresponding to the guest mailbox number in the VirtIO configuration memory at a specified offset, and triggers a CAN interrupt to the guest.

When the guest driver receives the interrupt, it reads the VirtIO device for the mailbox flags. Then, the driver can submit the frame content to the upper layers. The reception flags synchronization between VirtCAN and the guest is not necessary. In fact, all VirtIO write operations done by the guest are trapped by the hypervisor. Then they are used by VirtCAN which updates itself the real flag status.

D. Guest-to-Guest CAN Frame Transmission

The Guest-to-Guest CAN frame transmission matches the aforementioned CAN frame transmission. The different layers operates as described in Section IV-B. In addition, we create a special association between a guest reception mailbox and a transmitting virtual-mailbox. When a frame is sent by a guest, these association sets are checked to notify the destination guest(s). Then, VirtCAN sends the frame to the hardware controller.

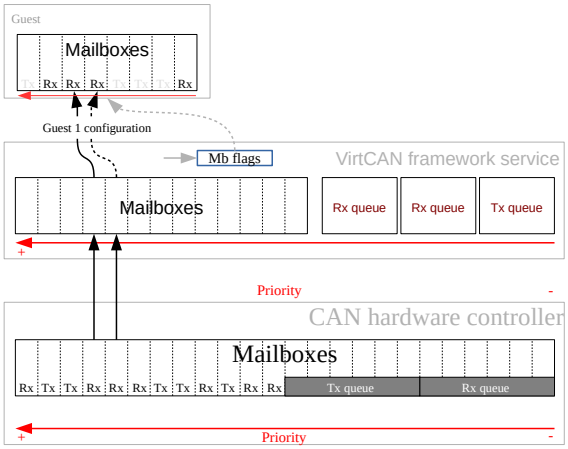

Fig. 7. Hardware frame reception to guest layers

If the frame is also destinated to another guest on the same board, `VirtCAN` set the guest flags, accordingly to the predefined association.

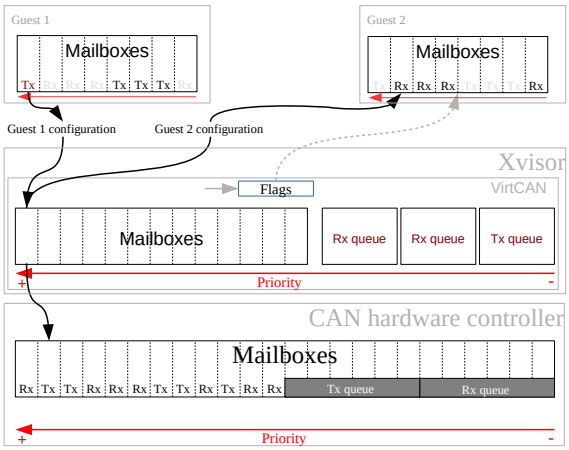

Fig. 8. Internal frame communication

V. Conclusions

In this work, we propose a framework, `VirtCAN`, to manage internal and external CAN bus virtualization with error management. The virtualization scheme is also preserved, as the hardware and the virtualization layers are clearly separated as seen in Figure 9.

As presented in the Xvisor memory benchmarks [10], the memory throughput of emulated operating systems is as good as a native one (up to 98%). There are only two memory copies for both frame direction: one from the guest to `VirtCAN`, and one from `VirtCAN` to the hardware for a frame emission or reception. Moreover, no specific scheduling is required to implement this mechanism.

Our future work consists in implementing `VirtCAN`. As the higher CAN transmission speed is up to 1 MB/s, and the memory throughput between guests and host is superior to 100 MB/s, we expect a low virtualization overhead, and a very close bus occupation and behavior between the native system and the virtualized one.

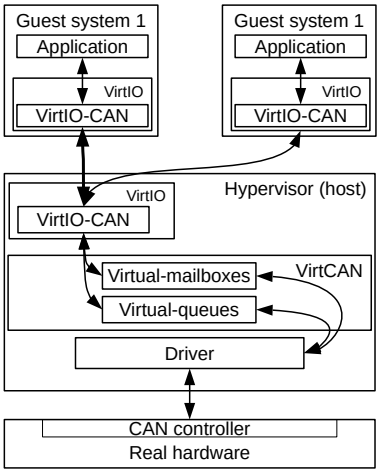

Fig. 9. System design

References

[1] Charette Robert. This car runs on code, 2009.
[2] J.Motavalli. The dozens of computers that make modern cars go (and stop). 2010. Accessed: 2015-06-16.
[3] G.Pitcher. Growing number of ecus forces new approach to cars electrical architecture. 2012. Accessed: 2015-06-16.
[4] CAN Specification Version 2.0. Standard, Bosch, Stuttgart, September 1991.
[5] Jake Edge. An api for virtual i/o: virtio. 2007. Accessed: 2015-10-23.
[6] Dor Laor. VirtIO, 2008.
[7] Anup Patel. Xvisor: eXtensible Versatile hypervISOR. Accessed: 2014-05-05.
[8] J.Edge. An api for virtual i/o: virtio. 2007. Accessed: 2015-06-17.
[9] R.Russel. virtio: Towards a de-facto standard for virtual i/o devices. pages 1–2, 2013. Accessed: 2015-04-08.
[10] A. Patel, M.Daftedar, M.Shalan, and M.W.El-Kharashi. Embedded hypervisor xvisor: A comparitive analysis. pages 4–9, 2015.
[11] Urs Thuermann. The virtual CAN driver.