Scheduling of CAN Message Transmission when Multiple FIFOs with Assigned Priorities are Used in RTOS Drivers

Michal Lenc, Pavel Píša
Czech Technical University in Prague, Faculty of Electrical Engineering
Department of Control Engineering

Typical RTOS general-purpose CAN bus subsystems offer interfaces that queue CAN messages for transmission in FIFO order. This leads to bus arbitration priority inversion situations when a critical message is sent after a low-priority one and a bus is fully loaded by another device. This is usually solved by adding FIFOs for different traffic classes. The presented solution allows the dynamic redistribution of a fixed set of CAN controller transmission buffers to these classes. The CTU CAN FD open-source IP core has been chosen as the first supported device because it allows transmission order reassignment of these buffers on the fly.

The current work targets an open-source RTEMS executive because it needs a new full-featured CAN/CAN FD stack. The executive is used in satellites and critical applications and CAN bus popularity is rising in these areas. The project builds on the infrastructure designed for LinCAN driver used in GNU/Linux real-time applications for decades, even before SocketCAN.

When tested on CTU CAN FD and RTEMS, other RTEMS CAN drivers can be ported to the framework. It can even be used for SocketCAN driver updates when Linux kernel network interface multiple queues are used, as well as for NuttX drivers.

Real-Time Executive for Multiprocessor Systems, commonly abbreviated as RTEMS, is an open-source real-time executive designed for embedded systems and offering a standard POSIX interface. It is widely used for space systems and other critical applications; in the areas where CAN bus usage is gaining momentum. However, the executive does not yet have a general-purpose CAN/CAN FD stack that would provide a common application interface but the developers have to implement target-dependent solutions.

Our goal towards RTEMS was to introduce a common CAN/CAN FD stack that would provide a unified interface for chip drivers and thus simplify both the porting of new drivers and CAN bus usage from an application perspective.

Bus Priority Inversion Problem Introduction

A common problem of general-purpose CAN drivers utilizing software FIFO queues is the bus arbitration priority inversion problem. This problem occurs when the bus is saturated by middle-priority messages from one controller and a mix of low and high-priority messages is pending on the other controller. This problem is usually solved by introducing priority classes and messages being routed to different queues based on their priority class assigned from CAN message identifier ranges.

Mapping priority classes to a limited count of the controller‘s hardware transmission buffers however tends to be challenging. This is caused by controllers usually providing transmission of messages based on their CAN identifiers or in the fixed order determined by the TX buffer index.

Applications and sometimes even protocol requirements expect the preservation of message order written by the application, even when different identifiers are used. This requirement however disqualifies the messages transmission order based on CAN identifier. On the other hand, more priority FIFO classes lead to the need to send messages in the order determined by those classes. Usually, the driver limits the transmission to one buffer per class or even to one TX buffer at all and thus not using the full potential of the controller.

Dynamic allocation of TX buffers to multiple priority groups

The proposed design deals with the bus arbitration priority inversion problem by extending the common solution of FIFO queues for each priority class. This solution of multiple traffic classes is extended by adding the dynamic redistribution of CAN transmission buffers to these classes.

The dynamic redistribution ensures the hardware TX buffers are assigned the correct priority based on the priority class of the inserted message and that the order determined by the user application for given stream/FIFO is preserved. After the message is released by the application to the queue structure, it is inserted into the proper priority class based on the identifier match filter. The controller's driver is notified of the new TX message to be processed and checks whether it has a space in hardware buffers. This is a standard behavior for most of the CAN controllers.

If no space available in hardware buffers, the controller checks the priority of the highest pending priority class. If a message of lower priority class occupies the buffers, it is replaced by the pending message and scheduled for later processing. The buffer with a newly inserted message has to be inserted in the transmit sequence after all messages of the same or higher priority class but before all messages of a lower priority class. This way the controller ensures the messages of the higher priority class are sent first and also the order of messages within the same priority class defined by one or more applications is preserved.

The sequence of buffers to be transmitted is stored in the TX order array holding the numbers of hardware buffers as they should be processed. This can be either a standard array or for a controller with up to 8 TX hardware buffers 32-bit large unsigned integer can be used as 0x01234567 value, representing transmit order with buffer 0 (having the priority 7) being sent first, fits into 32 bits. Using operations on unsigned integers means the resulting code for message promotion or demotion just consists of bitwise operations and shifts, therefore entirely omitting if statements and loops and resulting in speeding up the code execution.

The priority classes are mapped to the TX order array using the TX order tail array with information about the tail for a given priority class. The tail points one position beyond the previous valid class slot. Only the tail is needed as the head for the highest priority is fixed and heads for lower priorities are at the exactly same position as previous priority tals. This tail is used for the insertion of new frames if the array is not full or for reorganization when inserting new frames into the array.

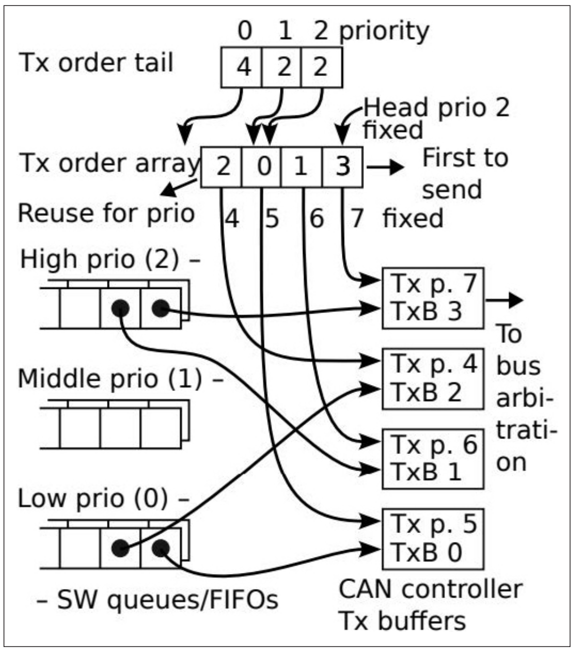

Figure 1: Visualization of dynamic allocation of TX buffers

The relationship between hardware buffers and SW queues and the usage of TX order array mapping is visualized in Figure 1. Only four hardware buffers and three priority classes are used for the simplification. The sequence of messages from the application is the following; low priority message inserted to buffer 0, high priority message inserted to buffer 3, next low priority message inserted to buffer 2, and next high priority message inserted to buffer 1. Without rotation of buffer priorities, this would mean the controller would try to send a low-priority message first and in a better case delay a high-priority message or in a worse case cause priority inversion problem.

However the presented approach reorganizes the hardware buffer priorities and as a result buffers 3 and 1 are sent in prior to buffers 0 and 2, ensuring the correct priority and application-based order. TX order tail array moved priority classes tails to position 2 for high and middle priority classes, therefore new messages from those classes would be inserted there and lower priority classes would move left. Note that the tail has to be moved also for all lower priority classes, therefore middle priority class is moved as well. Adding a new high-priority or a middle-priority message would result in low-priority from TX buffer 2 transmission deactivation and being pushed to software FIFO if not sent yet. Buffer two would then be reused for a high-priority message.

RTEMS CAN/CAN FD stack

Our implementation of a common CAN/CAN FD stack to RTEMS is based on an infrastructure called LinCAN designed at Czech Technical University by Pavel Píša as a loadable module for Linux kernel and its real-time variants [2]. It has been used even in industrial applications for decades.
Each CAN controller is registered as a character device into the /dev by name (standard naming can0, can1, and so on is used). The usage of the POSIX character device interface allows the application to use standard read/write/ioctl system calls for CAN bus transmission and setup. Multiple open of one device in blocking or non-blocking mode from multiple applications or threads is supported.

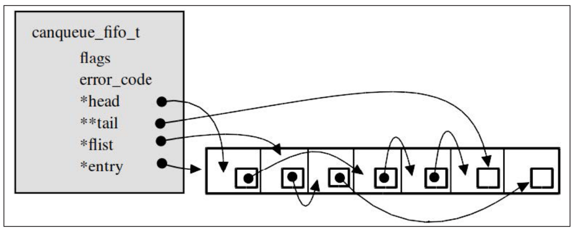

Figure 2: CAN FIFO implementation [2]

The stack infrastructure is based on message FIFO queues, illustrated in Figure 2, with a configurable number of slots for messages. These queues are organized into oriented edges between chip drivers and CAN users (applications that access FIFO through an opened character device) and are responsible for message transfers from application to chip driver and vice versa.

Both controller and user applications hold edge ends. These edges are divided according to their direction and priority and inserted to the correct list on its input and output ends. The interconnection of one CAN controller with two user applications is illustrated in the Figure 3. The input ends of edges/FIFOs are held in an inlist, the inactive/empty out ends in an idle list and active out ends are held in an active list corresponding to the edge priority. The controller and application then examine the active list and determine if there are any messages to process (either TX or RX based on edge direction).

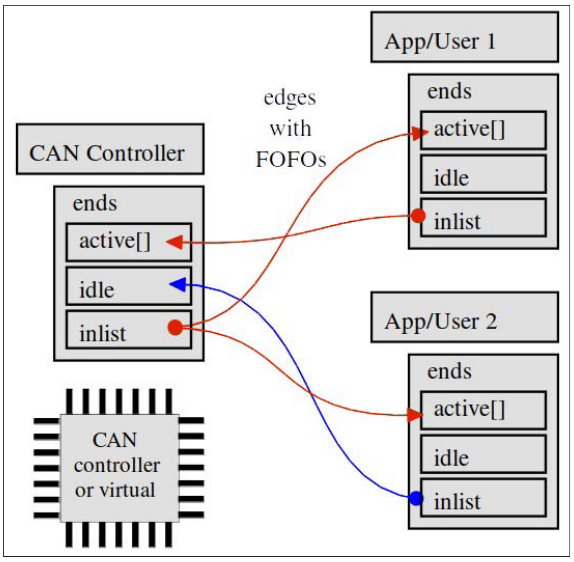

Figure 3: Message flow in graph edges [2].

Three FIFO queues for each application that can route messages to the controller are used in current implementation. Each queue is assigned with its priority from 0 to 2 where 2 is the highest. The controller's driver takes frames from FIFOs according to their priority class and transmits them to the network. When the frame is successfully sent, it is echoed to all queues back to open file instances except the sending one (this option is configurable). Received frames are filtered to all queues to applications ends of the queues, which filter matches the CAN identifier and frame type.

The FIFO queues are designed to support the need to provide services optimal to CAN frame delivery. On the controller side, multiple slots, each with one CAN frame, can be taken from FIFO and kept until the transmission is finished. Then, the frame from the slot is distributed to inform clients that the frame was sent. The framework has a unique feature to allow pushback slots (frames) when some later scheduled low-priority frame occupies the hardware TX buffer, which is urgently demanded for a higher priority pending message from other FIFO. The frame is pushed back to the originating FIFO head and scheduled for later TX processing. This functionality is necessary for the dynamical allocation mechanism described in the previous chapter.

is a possibility to handle chip start from the board support package level as well.

The can_frame structure represents one CAN message (called a frame in the RTEMS driver). This structure has a statically defined header represented by a can_frame_ header structure. The header has 8 bytes timestamp, 4 bytes CAN ID, 2 bytes flags, and 2 bytes data length. Data are statically allocated to a 64-byte long array with byte access.

```
struct can_frame_header {
  uint64_t timestamp;
  uint32_t can_id;
  uint16_t flags;
  uint16_t len;
};
struct can_frame {
  struct can_frame_header header;
  uint8_t data[CAN_FRAME_MAX_DLEN];
};
```

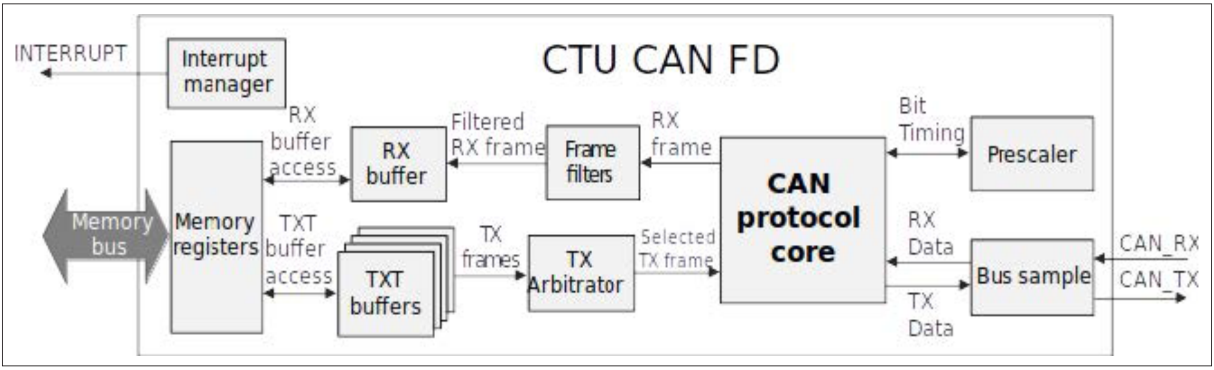

Figure 4: CTU CAN FD IP core structure [5]

The ends of the edges are disconnected during close operation. For outgoing edges, the driver waits for the messages to be sent and therefore ensures that no message already written by the user is lost. Close function by default waits for this operation in blocking mode. It is possible to use the ioctl call to set non-blocking return from close as edges are still kept allocated until the messages are sent. User can use other ioctl call to check whether there are still some pending messages.

Opening or closing a file descriptor does not necessarily have to start or stop the chip itself. Instead, this operation is handled by the ioctl call from the application. This way the user can control the chip usage. There

CTU CAN FD

The development of open source CAN core originated at the Department of Measurement of FEE at CTU under the lead of Jiří Novák. The core developed by Ondrej Ille and called CTU CAN FD is a softcore written in VHDL [4] The design does not require any vendor-specific libraries and is compliant with ISO 11898-1:2015 standard.

CTU CAN FD core can have up to 8 independent buffers for TX messages and each buffer has its own state and three bit priority number. These priorities can be used for FIFO behavior simulation in case only one FIFO queue is supported for TX messages as in SocketCAN for example.

Each buffer is then assigned with different priority and those priorities are rotated after the transmission is completed [4].

This core was chosen for the first demonstration as it supports the abort of currently queued TX buffer and the buffer priority updates on the fly which changes the requested buffer transmission order.

Dynamic allocation demonstration with RTEMS and CTU CAN FD

The ability of dynamic allocation of TX buffers to priority groups was demonstrated on educational kit MicroZed APO based on MicroZed evaluation kit with Xilinx Zynq-7000 system on chip. The programmable logic part was configured with four independent CTU CAN FD IP cores/CAN FD controllers, each one with four TX buffers. The application ran on RTEMS executive and used the newly introduced common CAN/CAN FD stack.

One controller was fully loading the CAN bus with 8-byte long messages with 0x500 identifier (used as middle priority in this test), while the other one was accessed from two applications. One application was attempting to send messages with a 0x700 identifier (low priority) and the other one was sending 8-byte long messages with a 0x20 identifier (high priority). This way the test simulated a fully loaded bus by one controller and another controller trying to access this bus.

The high-priority messages were sent in the burst of size 4 and the application subsequently waited long enough to send those messages to the bus. The latency profile in Figure 5 shows the write-to-receive latency in microseconds for 10,000 sent high-priority messages. RX side timestamps are captured at Start of Frame bit. High-priority messages would wait indefinitely for low-priority ones to leave the FIFO if only traditional driver with FIFOs would be used. The black vertical lines represents the length of one 8-byte long message transmission (about 120 microseconds).

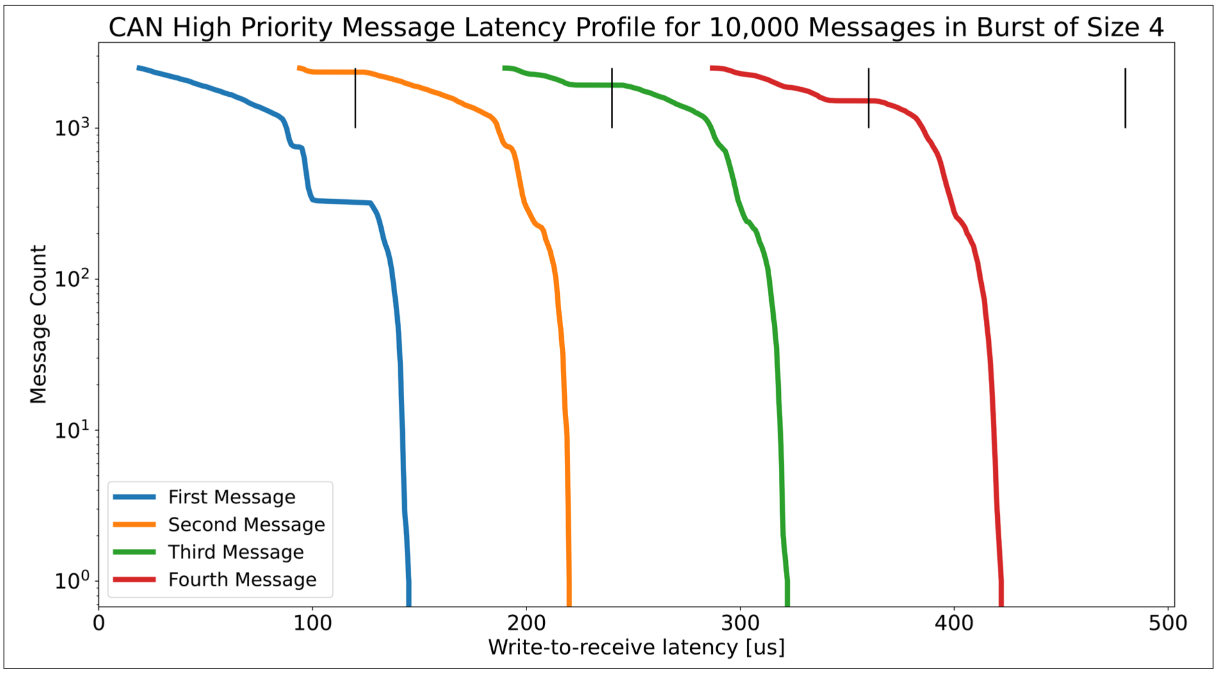

Figure 5: High-priority message latency profile

Conclusion

Based on the tests on real hardware, the presented solution can be used to ensure the correct transmit order of various priority messages while using all the controller's hardware resources. This way the bus arbitration priority inversion problem is solved. The solution is also not only specific for CTU CAN FD softcore, but can be ported to other drivers and systems if the upper layer stack supports more priority classes and their rescheduling.

From the RTEMS point of view, the presented common stack can be a step towards the unification of CAN controller usage as it provides a common application interface standardized with other operating systems such as NuttX or GNU/Linux.

In the future, the dynamic rescheduling can be implemented in other systems focused on real-time performance, namely NuttX OS.

Development version is available at [6] with future plans on implementation to RTEMS mainline. More comprehensive list of CTU CAN related projects is available at [1].

References

[1] CAN bus CTU FEE Projects, available online at https://canbus.pages.fel.cvut.cz/

[2] Píša, P., Vacek, F.: Open Source Components for the CAN Bus, 5-th RTLWS, 2003

[3] Píša, P.; Linux/RT-Linux CAN Driver (LinCAN), 2005, available online at https://cmp.felk.cvut.cz/~pisa/can/doc/lincandoc-0.3.pdf

[4] Jeřábek, M.; CTU CAN FD Driver, The Linux Kernel documentation, available online at https://docs.kernel.org/networking/device_drivers/can/ctu/ctucanfd-driver.html

[5] Ille, O.; Novák, J.; Píša, P.; Vasilevski, M.: CAN FD open-source IP core, In: CAN Newsletter 3/2022, CAN in Automation, 2022

[6] CTU FEE GitLab, available at https://gitlab.fel.cvut.cz/otrees/rtems/rtems-canfd

Michal Lenc
Czech Technical University in Prague Faculty of Electrical Engineering Department of Control Engineering – K13135
Karlovo náměstí 13
CZ-120 00 Prague 2
lencmich@fel.cvut.cz

Pavel Píša
Czech Technical University in Prague Faculty of Electrical Engineering Department of Control Engineering – K13135
Karlovo náměstí 13
CZ-120 00, Prague 2
pisa@fel.cvut.cz

This work was supported in part by the Technology Agency of the Czech Republic, program National Competence Centres under Grant TN02000054 BOVENAC.