F3

Faculty of Electrical Engineering
Department of Control Engineering

Master's Thesis

CAN FD Support for Space Grade Real-Time RTEMS Executive

Michal Lenc
michallenc@seznam.cz

May 2024
Supervisor: Ing. Pavel Píša, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Lenc Michal** Personal ID number: **492387**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

CAN FD Support for Space Grade Real-Time RTEMS Executive

Master’s thesis title in Czech:

Podpora sběrnice CAN FD pro operační systém RTEMS

Guidelines:

RTEMS is a mature system used in space and other critical fields. The CAN and CAN FD communication gains momentum in these areas as well. There exist multiple implementations of standard CAN systems for RTEMS already, but none of them is on pair with implementations used on GNU/Linux and other RTOSes. The goal of the project is to provide a sound base for actual and future projects using CAN FD.
1) Familiarize with RTEMS executive and already existing included CAN solutions and alternative approaches from other RTOSes, SocketCAN, LinCAN, NuttX
2) Design the solution for RTEMS with help or reuse fitting code sources
3) Implement and integrate RTEMS driver for CTU CAN FD
4) Document infrastructure and demonstrate it on pysimCoder generated application

Bibliography / sources:

[1] RTEMS User Manual and related documentation, On-Line Applications Research Corporation (OAR) and others, 1988-2020, (available online athttps://docs.rtems.org/)
[2] Píša, P.: Linux/RT-Linux CAN Driver (LinCAN), Ocera, June 2005-2013, available online https://cmp.felk.cvut.cz/~pisa/can/doc/lincandoc-0.3.5.pdf
[3] pysimCoder - Open Source Rapid Control Prototyping Development Tool, available online https://github.com/robertobucher/pysimCoder

Name and workplace of master’s thesis supervisor:

Ing. Pavel Píša, Ph.D. Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **05.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Pavel Píša, Ph.D. Supervisor’s signature	prof. Ing. Michael Šebek, DrSc. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt

Student’s signature

Acknowledgement / Declaration

I would like to thank my supervisor Pavel Píša for his mentorship of many projects and for valuable advices and experience sharing during my bachelor's and master's studies.

My acknowledgment also goes to my colleagues at Elektroline for giving me the opportunity to combine my study and work responsibilities and for creating a great work environment.

Last but not least, I would like to thank my family and friends for their support in my activities.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague 24. 5. 2024

.....

Abstrakt / Abstract

Kritické systémy reálného času, mezi které patří řídicí prvky v automobilech a hromadné dopravě, systémy pro družice, zdravotní zařízení nebo síťové prvky, často vyžadují deterministický přenos kritických zpráv mezi zařízeními. K tomuto přenosu se mimo jiné využívá sběrnicový standard Controller Area Network. Operační systémy obvykle ke zjednodušení přístupu na CAN sběrnici poskytují obecné rozhraní mezi aplikacemi a ovladači řadičů.

Hlavním cílem této práce byla implementace takového rozhraní pro otevřenou exekutivu reálné času RTEMS využívanou také v kosmických aplikacích. Práce dokumentuje design a vývoj CAN/CAN FD subsystému pro RTEMS s podporou prioritních front, blokujícího a neblokujícího přístupu, reportu chyb, rozhraní pro konfiguraci kontroléru a dalších funkcí. Práce měla za cíl poskytnout potřebné rozhraní k využívání CAN sběrnice a k implementaci budoucích řadičů. Jako první byl k testům a demonstraci vybrán CTU CAN FD řadič.

Práce také naráží na známý problém inverze priorit během arbitrážní fáze na CAN sběrnici a navrhuje jeho řešení. To počítá s dynamickou redistribucí hardware bufferů řadiče na prioritní fronty. Toto řešení umožňuje využití všech bufferů řadiče při zachování správného odchozího pořadí CAN zpráv. Navržený algorithmus je opět otestován na CTU CAN FD řadiči.

Klíčová slova: RTEMS, operační systémy reálného času, softwarový CAN stack, CAN sběrnice, inverze priorit, prioritní fronty, CAN řadič, CTU CAN FD, Xilinx Zynq

Critical real-time control fields such as automotive, public transport, space systems, medical devices, or networking systems often require deterministic transmission of safety-critical messages between devices. Controller Area Network bus standard is widely used for these purposes. To provide complete and unified access to CAN bus devices, operating systems implement common CAN stacks used as an interface between applications and device drivers.

Implementation of such a stack for open-source space grade RTEMS executive is the key part of this thesis. It documents the design and development of a full-featured CAN/CAN FD stack for RTEMS with support for multiple priority classes, blocking and nonblocking access, error reporting, controller configuration interface, and more. The work aims to provide a sound base for CAN bus usage and future controller implementation. Support for CTU CAN FD IP core, chosen as the first target for testing and demonstration, is implemented as well.

The thesis also deals with the common problem of CAN bus arbitration phase priority inversion. It proposes the solution of dynamic redistribution of CAN transmission buffers to priority classes that allow mapping of all controller's transmission buffers to priority classes while preserving the correct transmission order. The proposed algorithm is once again demonstrated on CTU CAN FD target.

Keywords: RTEMS, real-time operating system, CAN software stack, CAN bus, priority inversion, priority classes, CAN controller, CTU CAN FD, Xilinx Zynq

Contents /

/ Figures

Chapter 1
Introduction

Real Time Executive for Multiprocessor Systems[1], abbreviated as RTEMS, is an open source real time operating system (RTOS) designed mainly for embedded devices. It supports POSIX application programming interface, making the designed application compatible with for example GNU/Linux or NuttX operating systems as well. RTEMS is widely used in critical real time control fields, such as space systems, medical devices, or networking systems. These are the fields where transfers of critical messages between used devices are usually required and Controller Area Network (CAN) bus standard is an ideal choice to ensure the highest priority message gets precedence over lower priority ones in case of collision on the bus.

However, the executive does not have a general purpose CAN/CAN FD stack implementation that would provide a common application interface. Instead of that, target dependent solutions in applications/BSPs are required for current CAN drivers. This might be sufficient for some applications but a common stack with blocking and non-blocking access, priority classes, or message filtering is a necessity for more complex usage.

The main goal of the thesis was to design and implement a common CAN/CAN FD stack that would implement features needed for full utilization of CAN bus, provide the missing unification of the application interface for CAN controllers and simplify both the porting of new drivers and CAN bus usage from the application perspective. The presented implementation of RTEMS common CAN stack is based on the Linux kernel loadable module LinCAN developed at Czech Technical University in Prague by Pavel Píša in the early 2000s and subsequently used in real time applications for decades. It is based on message FIFO queues, organized into oriented edges between chip drivers and CAN users, that are responsible for message transfers from the application to the controller and vice versa.

A common setback of general purpose CAN drivers utilizing software FIFO queues is the bus arbitration priority inversion problem. It occurs when the bus is saturated by middle priority messages from one controller and a mix of low and high priority messages to transmit is pending on the other controller. In that case, high priority message is blocked by medium priority one and thus priority inversion scenario occurs.

The usual solution is to introduce priority classes assigned from CAN message identifier ranges and route messages to different queues based on those classes. 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.

However, the transmission order based on CAN identifiers can not be used if the preservation of the message order determined by the application is required. On the other hand, using more priority FIFO classes requires sending messages in the order determined by those classes. As a result, the correct transmission order should send

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

messages based on their priority class and keep the order within one priority class. This usually forces the CAN driver to limit transmission to one buffer per class or even to one buffer at all and thus not utilize the full potential of the controller.

The solution proposed in scope of this thesis solves 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. This way, the controller may use all its hardware transmission buffers, and the correct transmission order is preserved. Open source CTU CAN FD IP core developed at Czech Technical University was chosen for the demonstration of presented solution. It supports the abort of currently queued TX buffer and the buffer priority updates, which changes the requested buffer transmission order, on the fly.

The thesis starts with chapter 2 introducing the reader to RTEMS executive, its basic principles, and compilation steps for thesis results reproduction. The core part of the thesis, the implementation of new full-featured CAN/CAN FD stack to RTEMS is described in chapter 3. The implemented infrastructure is described from both theoretical and implementation based perspective. The next chapter faces the priority inversion problem on CAN bus and proposes a solution that can be used with the implemented infrastructure. This chapter is more focused on theoretical part. Practical impacts are discussed in chapter 5 together with the implementation of CTU CAN FD controller. The thesis is completed with chapter 6 presenting the achieved results.

The text is partially based on a science article published at the 18th international CAN Conference in 2024 [1].

Chapter 2
Real Time Executive for Multiprocessor Systems

This chapter introduces Real Time Executive for Multiprocessor Systems, the main system used in this thesis, to the reader and discusses its usage and advantages. Compilation steps for the target used for result demonstration are presented at the end of the chapter.

The goal of this chapter is not to supplement comprehensive documentation provided by RTEMS project itself, but to introduce its concepts used for CAN/CAN FD stack implementation to readers not familiar with the executive. RTEMS documentation should be studied to obtain information about its concepts surpassing the scope of this thesis.

2.1 Introduction

The development of Real Time Executive for Multiprocessor Systems, commonly abbreviated as RTEMS, began in the late 1980s at the request of US Army Missile Command for executive based on open standards and free of royalties. Thanks to its open source license, RTEMS quickly became a popular solution for embedded devices even in the commercial sector [2].

The executive has come a long way since that and is a mature open source real time operating system used in many critical fields such as military, space, medical, or industry control systems. RTEMS provides support for commonly used instruction set architectures including ARM designs, x86 architecture, PowerPC, RISC-V or 32-bit i386 by Intel [3].

It is compatible with POSIX standard but also provides its own interface used mostly for kernel functions. Networking, if configured and enabled, is supported by full-featured IPv4/IPv6 TCP/IP networking stack from FreeBSD project as well as the DNS server. Most architectures have a subset of board support packages (BSP) that supports symmetric multiprocessing (SMP) used for targets with multiple processor cores.

Schedulers optimized for multiple core scheduling are used if SMP support is enabled. Preemtive round-robin scheduling and timeslices for aperiodic tasks are used. RTEMS has implemented support for several schedulers, both fixed priority and dynamic task, fixed job priority for periodic tasks [3].

RTEMS has a shared memory architecture meaning all processes and threads may access the same data stored in memory. It is a single address space system, therefore application and kernel reside in the same memory [3].

The kernel is written in C language, but the system also provides support for the applications written in other programming languages as C++, Ada, Java, or Lua for example. Modified build automation tool `Waf` is used for the kernel and BSP build and provides the benefit of having more builds of different architectures/BSPs available all at once.

2.2 Resources

The following sections briefly describe RTEMS handlers and synchronization routines used in the implementation of CAN/CAN FD stack and CAN controllers. RTEMS implements most of the presented resources in both POSIX and its own API. While POSIX API is a better choice for applications, RTEMS API (also called classic) is the preferable choice for the kernel itself.

2.2.1 Task

RTEMS documentation defines a task as the smallest thread of execution that can compete on its own for system resources [4]. Each task has assigned a priority number from 1 to 255 with 1 being the highest priority. The task is created by the directive `rtems_task_create()` and started by `rtems_task_start()`.

There are various settings available during task creation, for example stack size available to the task, preemtion settings, interrupt level, and so on. The infrastructure presented in this thesis uses tasks for the implementation of controller workers responsible for frame transmission and reception.

The following code shows task creation and start with default options and priority determined by `priority` variable. The task is implemented in function `worker` with input argument `arg`.

```
rtems_task_create(
  rtems_build_name( 'N', 'A', 'M', 'E' ),
  priority,
  RTEMS_MINIMUM_STACK_SIZE + 0x1000,
  RTEMS_DEFAULT_MODES,
  RTEMS_DEFAULT_ATTRIBUTES,
  &task_id
);

rtems_task_start( task_id, worker, arg );
```

2.2.2 Interrupt Handler

Interrupts, requests for the processor to interrupt current execution and process a different event, are managed through the interrupt manager in RTEMS. This manager provides necessary directives to attach and manage interrupts.

Interrupt handler is installed with directive `rtems_interrupt_handler_install()`. This directive takes an interrupt vector number and installs a routine called with an optional input argument when the interrupt occurs. The following code shows a unique interrupt handler installation with interrupt number `irq` calling routine `interrupt_handler` without an argument.

```
rtems_interrupt_handler_install(
  irq,
  "name",
  RTEMS_INTERRUPT_UNIQUE,
  interrupt_handler,
  NULL
);
```

The manager also provides synchronization via `rtems_interrupt_lock_acquire()` and `rtems_interrupt_lock_release()` directives. These functions are used to protect shared data between an interrupt handler and the thread owning the handler. Maskable interrupts are disabled during interrupt lock.

These locks were initially used for CAN FIFO queues synchronization before switched for the mutex mechanism described in section 2.2.4.

2.2.3 Semaphore

Self-contained binary semaphores (mechanism with integer value 0 or 1) are used to implement critical sections in the infrastructure (such as read/write operations) or to wait for some event (until all frames are sent for example).

Both static (compile time) and dynamic (run-time) initialization of the semaphore is available. Macro `RTEMS_BINARY_SEMAPHORE_INITIALIZER` is used for the first one and function `rtems_binary_semaphore_init()` for the latter. Wait for the semaphore is implemented in `rtems_binary_semaphore_wait()` function. It returns immediately if the current semaphore value is 1, otherwise the thread is blocked and waits for semaphore to change its value to 1. Since this can wait indefinitely, the user can use `rtems_binary_semaphore_wait_timed_ticks()` to wait with a defined timeout or `rtems_binary_semaphore_try_wait()` to return with an error if the semaphore is not set.

Function `rtems_binary_semaphore_post()` wakes up the highest priority waiting thread or increments the value to 1 if no thread currently waits on the semaphore.

Examples of semaphore usage can be found in section 5.2 describing the implementation of CTU CAN FD controller.

2.2.4 Mutex

Mutual exclusion, often referred to as a mutex, is a synchronization mechanism ensuring a task does not enter a critical section if another task is already present in it. RTEMS implements mutual exclusion with self-contained objects. These can be initialized either statically with macro `RTEMS_MUTEX_INITIALIZER` or dynamically at run-time with `rtems_mutex_init()` function.

Locking is performed with `rtems_mutex_lock()` call, unlocking can be done with `rtems_mutex_unlock()`. The CAN/CAN FD stack was initially implemented with more lightway and general interrupt locks for synchronization but later was rewritten to mutual exclusion as it is more fitting for this usage. It is a more systematic solution that ensures CAN worker or CAN API calls cannot delay higher priority tasks when CAN related tasks are iterating inside CAN subsystem.

However, there are some disadvantages in this approach. Usage of mutexes does not allow to fill the CAN stack FIFO queues directly from an interrupt, but controller has to use a dedicated thread for this. Also, the choice of mutexes over interrupt locks caused small slowdown of CAN stack about approximately 6 micro seconds per frame.

2.3 Build Process

RTEMS build can be separated into three major steps: setting up the build system, BSP build and build of the networking stack (can be omitted if network subsystem is not required). The build example is done for ARM based Xilinx Zynq ZedBoard BSP used for result demonstration in this thesis.

2.3.1 Build System Setup

Source directories can be obtained either from project GIT repositories or stable releases can be downloaded from project websites. These steps present the first option as the work of the thesis is done against current development version of RTEMS.

```
mkdir rtems-git
cd rtems-git
git clone https://gitlab.rtems.org/rtems/rtos/rtems.git
git clone \
https://gitlab.rtems.org/rtems/tools/rtems-source-builder.git rsb
git clone https://gitlab.rtems.org/rtems/pkg/rtems-libbsd.git
```

Now with source directories obtained, the target directory for system build has to be selected. Directory `/opt/rtems/6` is used in this example. This is the path where RTEMS build target specific dependencies (GCC, GDB or Binutils for example) are located. These utilities can be built by following command.

```
cd rsb/rtems
../source-builder/sb-set-builder --prefix=/opt/rtems/6 6/rtems-arm
```

Other architecture (`rtems-i386` for example) can be specified instead of `rtems-arm` and more architectures can be built with common `/opt/rtems/6` prefix as source builder creates specific subdirectory for each architecture.

2.3.2 BSP Build

With architecture dependencies built and ready, board support package can be built. The initial configuration for Xilinx Zynq ZedBoard is set up by following command.

```
cd rtems-git/rtems
./waf bspdefaults --rtems-bsps=arm/xilinx_zynq_zedboard > config.ini
```

File `config.ini` can be edited and configuration changed based on user requirements (for example `RTEMS_POSIX_API` and `RTEMS_SMP` might be enabled). Following options were modified in our build for our BSP support.

```
RTEMS_POSIX_API = True
RTEMS_SMP = True
ZYNQ_UART_KERNEL_IO_BASE_ADDR = ZYNQ_UART_0_BASE_ADDR
```

Once configured, board support package can be installed to previously selected `/opt/rtems/6` with following commands.

```
./waf configure --prefix "/opt/rtems/6"
./waf
./waf install
```

2.3.3 FreeBSD Library Build

FreeBSD Library is required if networking stack (support for TCP/IP protocol for example) is used. It is once again built using Waf tool.

```
cd rtems-git/rtems-libbsd
git submodule init
git submodule update rtems_waf
./waf configure --prefix="/opt/rtems/6" \
  --rtems-bsps=arm/xilinx_zynq_zedboard \
  --buildset=buildset/default.ini
./waf
./waf install
```

2.4 Application Build with OMK

For the development and testing, OMK Make-System[1] was used to build the CAN/CAN FD stack, CTU CAN FD driver, and test applications. This section shows how to build the source code to reproduce the demonstration and results documented in chapter 6. It is currently possible to reproduce most of the tests on both Xilinx Zynq target and i386 architecture using mainline x86-64 QEMU. The source code can be obtained from CTU FEE GitLab repository.

```
git clone https://gitlab.fel.cvut.cz/otrees/rtems/rtems-canfd.git
```

2.4.1 Xilinx Zynq MzAPO Board

The build for Xilinx Zynq board is straightforward once the BSP is compiled and installed as described in previous sections.

```
export PATH=$PATH:/opt/rtems/6/bin
make
```

These commands builds the RTEMS kernel image with CAN/CAN FD stack, CTU CAN FD driver, and test applications. The image can be loaded to the board with boot from Trivial File Transfer Protocol (TFTP), the detailed description is available on OTREES wiki pages.[2]

2.4.2 i386 using QEMU

The presented CAN/CAN FD stack can also be tested using mainline QEMU `qemu-system-x86_64` by compiling the application for i386 target. It is necessary to build and install support for `i386_pc686` RTEMS targer. The steps are similar for Xilinx Zynq target described above. Once the build system is setup, the following may be used to compile and install BSP support.

```
cd targets/i386_pc686
./i386-rtems-sys.cfg
```

This script automates board support package installation. It is possible to run QEMU target with following commands.

```
export PATH=$PATH:/opt/rtems/6/bin
cd targets/i386_pc686
./setup-host-socketcan # might be required with sudo
./qemu-i386-pc686-2x-ctu-pci-build
./qemu-i386-pc686-2x-ctu-pci-run
```

[1] `http://rtime.felk.cvut.cz/omk/`
[2] `https://gitlab.fel.cvut.cz/otrees/rtems/work-and-ideas/-/wikis/home`

Chapter 3
CAN/CAN FD Stack for RTEMS

This chapter describes the implementation of a common CAN/CAN FD stack to RTEMS executive. Some RTEMS concepts introduced in section 2.2 are used here so readers not familiar with the executive is recommended to read it as well.

CAN standard and its basic principles are shortly presented at the beginning of this chapter. This is to provide a basic understanding of the requirements for CAN stack implementation. The rest of the chapter then focuses on CAN/CAN FD stack for RTEMS and its implementation.

3.1 Controller Area Network

Controller Area Network protocol (abbreviated as CAN, often referred to as CAN bus) is a vehicle bus standard developed in the early 80s at Bosh. It is currently heavily used in an automotive industry as a main communication protocol between car's computation units, but it also has its place in other fields, including real time control. The physical layer consists of two differential wires, `CAN_L` and `CAN_H`, ground wire and optional voltage supply wire.

The bus has two defined states: recessive (logical one) and dominant (logical zero) with the latter one preserved on the bus if different states are sent from multiple controllers at the same time. This characteristic is important for medium access control. CAN bus is a multi master protocol with stochastic access control (meaning there is no central node defining who can transmit) called Carrier-Sense Multiple Access with Collision Resolution (CSMA/CR). If occurs, collisions are resolved based on the data priority of messages competing for the bus.

The collisions are resolved during the arbitration phase, where each message has its unique identifier defining its priority. Since the dominant state wins over the recessive state, a message with the highest priority (the lowest identifier number) wins over messages with lower priorities and can transmit the message. The message transmission is called the data phase. Other messages compete for the bus again during the next arbitration phase.

Example: Suppose we have two controllers and 3 bit wide identifier for a message. Controller A wants to send a message with the identifier `0x1` (`001` binary) and B a message with `0x2` (`010` binary). During the arbitration phase, identifiers are compared bit by bit starting from most the significant bit. Therefore, for the first bit (`0` for both), nothing will be determined as both controllers drive the bus to the dominant state. However, for the second bit (`0` for A, `1` for B), controller A drives the bus to the dominant state and wins over the recessive state driven by B. As a result, a message with the identifier `0x2` loses the arbitration and leaves the bus to a higher priority message.

CAN frame has two possible formats: base frame format with 11 bit identifier and extended frame format with 29 bit identifier. The message can also be sent with RTR flag, defining it is a remote transmission request for a given identifier. Both standard and extended CAN frame formats are depicted in Figure 3.1.

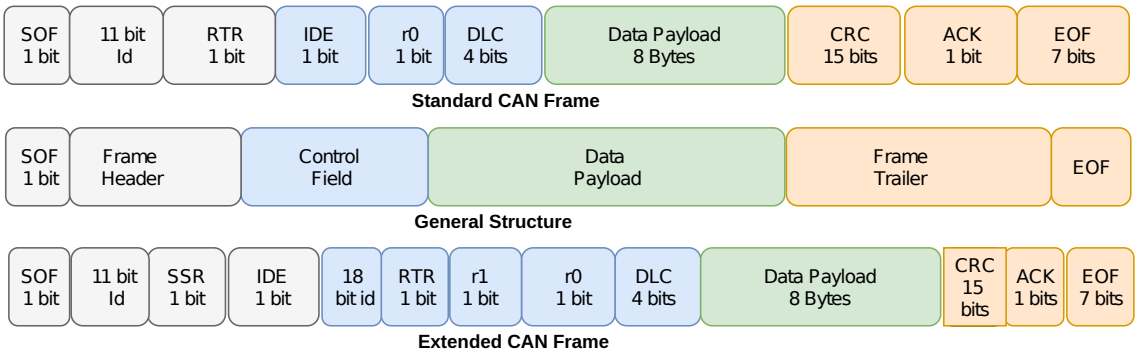

Figure 3.1. Visualization of standard and extended CAN Frame format (Source: [5]).

Original CAN protocol (nowadays referred to as CAN CC) speed is limited to 1 Mbit/s for a bus length of 50 meters. This limitation, caused by arbitration phase synchronization requirements, was overcome with Controller Area Network Flexible Data-Rate (abbreviated as CAN FD) protocol enhancing standard CAN format. It utilizes bit rate switching between arbitration and data phases: keeping the arbitration at the original speed but sending data faster. The speed of CAN FD frames can be up to 8 Mbit/s on 50 meters, although lower values are currently used in the industry. Automotive standard, for example, is 500 Kbit/s for arbitration and 2 Mbit/s for data rate.

Another advantage of CAN FD frames is the larger maximal data payload. While standard CAN frame has only up to 8 bytes of data, flexible data format can transfer up to 64 bytes. The limitation of CAN FD frames is a backward incompatibility. Therefore, CAN FD frames can be used only if every controller in the network is FD capable. CAN FD frames, both standard and extended format, are visualized in Figure 3.2

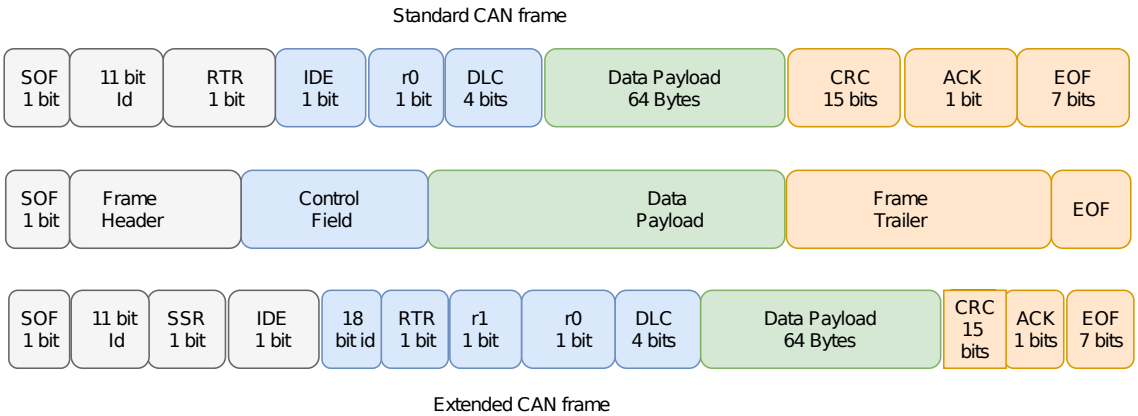

Figure 3.2. Visualization of standard and extended CAN FD Frame format (Source: [5]).

With standard CAN and CAN FD already established and heavily used in industry including automotive, the third generation of CAN is finding its way to some end applications. Protocol called Controller Area Extended Data-Field Length (CAN XL), standardized in 2018, extends data field up to 2048 bytes. The bit rate switching

between arbitration and data phase introduced in CAN FD is used as well. This protocol is not supported by the current version of RTEMS CAN/CAN FD stack, but it may be a goal of future projects.

Various CAN stack implementations can be found across operating systems. GNU/Linux has a SocketCAN networking stack that utilizes socket API and provides a similar interface to TCP/IP protocol. This approach is slowly being implemented in other operating systems such as NuttX or Zephyr, although very limited and only as an alternative to the already existing character device drivers.

While SocketCAN has the advantage of similar API to TCP/IP protocol and the socket concept also fits CAN network, there are some setbacks. It internally uses different frame formats for standard CAN, CAN FD and CAN XL with CAN FD and CAN XL flags divided between lower three bits of frame identifier and separate flag field. Also, real time operating systems like RTEMS may benefit from having a separate simpler API compared to SocketCAN API. Moreover, usage of SocketCAN in RTEMS would require enabled TCP/IP support. This is not ideal if the target does not support networking or if the system is memory limited. TCP/IP support in RTEMS comes from full-featured BSP stack, which is powerful but also demands a considerable amount of resources.

3.2 RTEMS CAN Stack Basic Principles

The common CAN/CAN FD stack implementation developed for RTEMS executive is based on LinCAN infrastructure developed as a loadable module for Linux kernel at Czech Technical University in the early 2000s. The infrastructure utilizes message FIFO queues organized into oriented edges between controller drivers and CAN users. Both controller and application then hold ends of connected edges. These edges are then responsible for message transfers from an application to a controller and vice versa.

RTEMS provides POSIX interface including a standard character device, therefore each CAN controller is registered as a device into `/dev` device file directory. Standard naming (`can0`, `can1` and so on) might be used for device registration, but it is possible to use any different scheme. The usage of POSIX character device interface allows the application to use standard read/write/ioctl system calls for CAN bus transmission and setup. Operations `read()` and `write()` are used to read CAN message from FIFO queue and write new message to the queue, respectively. Stack and controller can be configured and set up by various `ioctl()` calls that are described throughout this chapter. This includes setting of bit timing values, CAN mode operation, or creation of new queues (priority classes).

The device can be opened in both blocking and nonblocking mode and one device can be opened from multiple applications. Read and write operations wait on binary semaphore until the frame can be passed to the framework in reception or there is space to enqueue the frame for transmission. The nonblocking mode can be used in combination with polling functions implemented by ioctl calls. As opposed to standard socket polling, the current implementation of RTEMS CAN framework can not poll both RX and TX in one call. Instead, separate calls for RX available and TX ready have to be used. User defined timeout, specified as a time interval in the monotonic clock scale and passed as an input argument, is internally recalculated to an absolute time against the monotonic clock (absolute time, usually since system startup, that does not jump forward or backwards as opposed to real-time clock).

The stack supports both standard and CAN FD frames with general `can_frame` structure described in section 3.3.6. Support for CAN XL frames is not implemented,

but further extension should be possible without the redefinition of basic principles introduced in scope of this thesis.

3.2.1 FIFO Queues

Edges/FIFOs between the application and controller are divided according to their direction and priority and inserted into the correct list on its input and output ends. The interconnection of one CAN controller with two user applications is illustrated in Figure 3.3. 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).

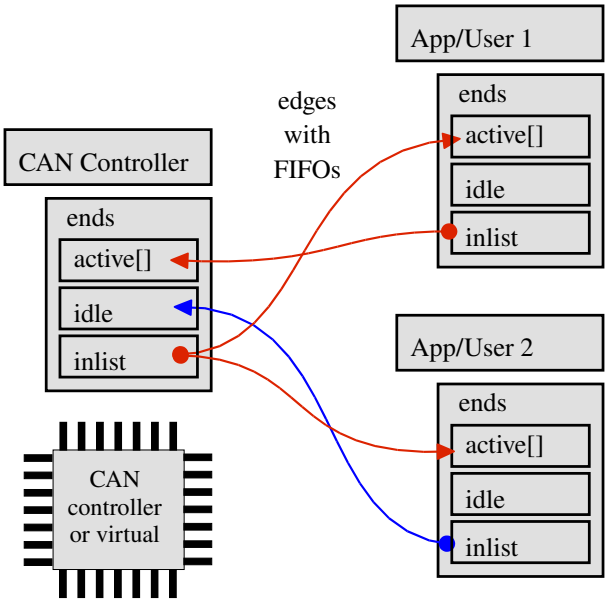

Figure 3.3. Message flow in graph edges/FIFOs (Source: [6]).

The current implementation specifies up to three FIFO queue priorities for edges routing messages from users to the controller. Each queue has an assigned priority from 0 to 2 where 2 is the highest. CAN message sent from the application to the infrastructure is inserted into the correct queue according to CAN identifier match with the queue ID mask. The controller's driver then takes frames from FIFO queues according to their priority class and transmits them to the network.

Successfully sent frames are echoed through queues back to opened file instances except the sending one (this is the default filter setting). The echo capability is fully configurable and can be turned off if required.

One TX and one RX queue (from an application point of view) is created for each of the opened instances by default. These queues have the lowest priority, default filter mask (no echo to sending instance and no error frames) and FIFO size for 64 messages configured by default. Allocated space for one CAN frame differs based on the used controller. If the chip is FD capable, queue slots are allocated for CAN frames with up to 64-data bytes, otherwise only slots for 8 data bytes are allocated. These queues are created by default to provide simple access to the network for users who do not need to create priority classes or more complicated filters.

If more complex queue setting is required, the user can create a new one with a dedicated ioctl call. This is also useful for creating more TX queues with different priorities,

representing commonly used priority classes in fact. Queues (edges) and their ends are disconnected and discarded either upon the user's call or during close operation. The stack ensures all outgoing messages already written by the user are sent before discarding the edges, therefore no message is lost. This operation is by default performed in blocking mode (regardless of what mode the instance was opened), meaning `close()` function blocks the calling thread until all messages are sent to the network. Non-blocking close behavior is configurable, the edges then still remain allocated until all messages are sent, but close operation is successfully finished without delay.

3.3 Application Programming Interface

This section describes the application programming interface (API). This includes configuration of both framework and controller and transmission and reception of CAN messages (frames). The section is written in the order the typical application will operate with CAN stack: first opening instance, configuration, and then transmission/reception.

3.3.1 Opening and Configuring Queues

As mentioned in section 3.2, every controller/device is registered as a node into `/dev` namespace. Since API is provided with standard POSIX calls, function `open()` can be used to obtain device file descriptor. It also respects standard `O_NONBLOCK` option in the call.

```
int open( const char* pathname, int flags );
```

This descriptor returned by `open()` call is then used to access the device. Opening device will create new TX and RX edge between application and controller with default settings as described in 3.2.1. New queue can be created from the application with `ioctl()` call `RTEMS_CAN_CREATE_QUEUE`.

```
ssize_t ioctl( fd, RTEMS_CAN_CREATE_QUEUE, &queue_param );
```

Input parameter `queue_param` is a `can_queue` structure containing configuration options (direction, priority, data length, buffer size, filter). Preprocessor defines `RTEMS_CAN_QUEUE_RX` and `RTEMS_CAN_QUEUE_TX` can be used to select queue direction. Structure `can_filter` holds the CAN frame identifier and flag filters, ensuring only frames matching this filter are passed to the queue's ends. Fields `id` and `flags` holds identifier bits and flags, respectively, required in CAN frame to be assigned to the corresponding FIFO queue. In other words, it specifies that only specific identifiers or flags shall be assigned to the queue. Members with `_mask` postfix are used to mask out identifiers or flags that are forbidden for a given FIFO queue.

This can be used to set if an application shall receive echo or error frames or to implement priority class mapping based on identifier range. Setting all fields of the structure to zero means all frames are passed through the queue.

```
struct can_filter {
  uint32_t id;
  uint32_t id_mask;
  uint32_t flags;
  uint32_t flags_mask;
};
```

```
struct can_queue {
  uint8_t direction;
  uint8_t priority;
  uint8_t dlen_max;
  uint8_t buffer_size;
  struct can_filter filter;
};
```

It is possible to create queues with a configurable number of slots (frames the FIFO can hold) and data length for one slot. These characteristics are set by `buffer_size` and `dlen_max` fields, respectively. It is possible to set a maximum data length of only up to 64 bytes (CAN FD data length), otherwise ioctl call results in an error. Setting data length or FIFO size to a negative value also causes an error. Default values – 8 bytes for standard CAN controller and 64 bytes for CAN FD capable controller – are set if data length is set to zero.

Note that every opened instance has its own FIFO queues/edges between the application and the driver. This means queues created for one instance will not affect the other one even if both instances opened the same device.

Queues for given instance can be removed/discarded by `RTEMS_CAN_DISCARD_QUEUES` command. It is possible to discard all TX queues at once, all RX queues at once, or all queues at once regardless of direction. The type of queues to be discarded is selected by integer parameter `type`, defines `RTEMS_CAN_QUEUE_RX` and `RTEMS_CAN_QUEUE_TX` can be used once again. These defines are basically a bit mask and bitwise or operation can be performed on them. This way, the user may discard both RX and TX queues at once.

```
ssize_t ioctl( fd, RTEMS_CAN_DISCARD_QUEUES, type );
```

Discarded TX queues ensure all messages are transmitted to the network according to the rules described in 3.2.1. The limitation of discarding only all TX or all RX queues at once is caused by the application knowing only a file descriptor for the opened instance. It does not have direct knowledge of internal arrangements and possibility to discard queues one by one would require additional multiplexing and would complicate both application and stack interface.

It is also possible to flush the queues. Once again, this operation can be imposed only on all TX or/and all RX queues at once, similarly to `RTEMS_CAN_DISCARD_QUEUES`.

```
ssize_t ioctl( fd, RTEMS_CAN_FLUSH_QUEUES, type );
```

3.3.2 Bit Time Calculation

While passing the desired bit rate and calculating bit time constants according to this parameter is commonly used in hobby projects or even in industrial applications without precise bit timing requirements, some applications may require a different approach. This is, for example, a case in an automotive industry where raw controller specific bit timing parameters have to be used.

To satisfy both approaches and the user's requirements, the infrastructure implements ioctl call allowing to pass bit rate or defined raw bit timing parameters. In case of bit rate usage, the constants are calculated by the infrastructure and selected to match the defined bit rate. The advantage of this approach is the easier configuration as the user only defines the desired bit rate. On the other hand, user can define those

parameters directly in an application and pass them through the ioctl call to the controller's registers. This is more demanding, but ensures correct timing parameters are set.

```
ssize_t ioctl( fd, RTEMS_CAN_SET_BITRATE, &set_bittiming );
```

The ioctl call `RTEMS_CAN_SET_BITRATE` unifies both approaches by passing a pointer to structure `struct can_set_bittiming`. The definition of the structure follows.

```
struct can_bittiming {
  uint32_t bitrate;
  uint32_t sample_point;
  uint32_t tq;
  uint32_t prop_seg;
  uint32_t phase_seg1;
  uint32_t phase_seg2;
  uint32_t sjw;
  uint32_t brp;
};

struct can_set_bittiming {
  uint16_t type;
  uint16_t from;
  struct can_bittiming bittiming;
};
```

Field `type` selects whether nominal bit time or data bit time (for CAN FD capable chips only) should be set. Preprocessor defines `CAN_BITTIME_TYPE_NOMINAL` and `CAN_BITTIME_TYPE_DATA` are recommended to be used. This approach also allows future extension of CAN XL bit time settings. Subsequent field `from` selects whether the bit time is calculated from a given bit rate or whether the values are already precalculated and passed through `can_bittiming` structure. Defines `CAN_BITTIME_FROM_BITRATE` and `CAN_BITTIME_FROM_PRECOMPUTED` are prepared for this.

Example: Suppose the user has CAN FD capable controller and wants to configure nominal bit rate to 500000 and data bit rate to 2000000 (those are standard values used in automotive indrustry). Nominal bit timing is calculated from given bit rate, data bit timing is given as precomputed values. The configuration would require following structures and ioctl calls.

```
static struct can_set_bittiming nominal = {
  .type = CAN_BITTIME_TYPE_NOMINAL,
  .from = CAN_BITTIME_FROM_BITRATE,
  .bittiming = {
    .bitrate = 500000,
  }
};

static struct can_set_bittiming data = {
  .type = CAN_BITTIME_TYPE_DATA,
  .from = CAN_BITTIME_FROM_PRECOMPUTED,
```

```
  .bittiming = {
    .tq = 10,
    .prop_seg = 18,
    .phase_seg1 = 18,
    .phase_seg2 = 13,
    .sjw = 1,
    .brp = 1,
  }
};

ioctl( fd, RTEMS_CAN_SET_BITRATE, &nominal);
ioctl( fd, RTEMS_CAN_SET_BITRATE, &data);
```

Bit time setup via ioctl call, although being recommended, is not mandatory. These values can be instead passed to the controller's initialization function during board initialization (if supported by the controller) or filled directly to its private structure.

Currently set bit timing values can be obtained from the controller together with maximum and minimum bit timing constants (these should be defined by the controller itself) with `RTEMS_CAN_GET_BITTIMING` call.

```
ssize_t ioctl( fd, RTEMS_CAN_GET_BITTIMING, &get_bittiming );
```

Argument `get_bittiming` is a pointer to `can_get_bittiming` structure. This structure, apart from current bit timing values and controller's constants, has field `type` defining type of bit timing values (nominal or data). Again, defines `CAN_BITTIME_TYPE_NOMINAL` and `CAN_BITTIME_TYPE_DATA` can be used as the type field is both input and output from ioctl call. Other fields are filled by ioctl according to actual controller settings.

3.3.3 Mode Setting

The CAN controller can be used in various modes (FD capable, loop back or listen only for example). Definitions of supported modes are taken from Linux kernel to provide a unified approach in settings. Controller specific ioctl call should be used if some controller supports unique mode not provided in the following table. Mode setup is done with `RTEMS_CAN_CHIP_SET_MODE` ioctl command, where selected modes are passed as 32-bit large unsigned integer.

```
#define CAN_CTRLMODE_LOOPBACK       ( 1 << 0 )
#define CAN_CTRLMODE_LISTENONLY     ( 1 << 1 )
#define CAN_CTRLMODE_3_SAMPLES      ( 1 << 2 )
#define CAN_CTRLMODE_ONE_SHOT       ( 1 << 3 )
#define CAN_CTRLMODE_BERR_REPORTING ( 1 << 4 )
#define CAN_CTRLMODE_FD             ( 1 << 5 )
#define CAN_CTRLMODE_PRESUME_ACK    ( 1 << 6 )
#define CAN_CTRLMODE_FD_NON_ISO     ( 1 << 7 )
#define CAN_CTRLMODE_CC_LEN8_DLC    ( 1 << 8 )
#define CAN_CTRLMODE_TDC_AUTO       ( 1 << 9 )
#define CAN_CTRLMODE_TDC_MANUAL     ( 1 << 10 )

ssize_t ioctl( fd, RTEMS_CAN_CHIP_SET_MODE, modes );
```

This command sets modes defined in `modes` argument and disable all the others. Thus, the same ioctl can be used for both mode enable and disable operations. The call results in an error if the application tries to set a mode not supported by the controller. Setting mode is also enabled only if the controller is stopped (see section 3.3.4), otherwise the ioctl call returns an error.

The controller's driver can have some default modes that are set during controller initialization. For example, CAN FD capable controllers may have `CAN_CTRLMODE_FD` set by default.

3.3.4 Chip Start and Stop

Opening the driver instance does not mean the chip is started. Quite the opposite, it is useful not to start the chip during open operation as the configuration of bit timing or mode might follow. To start the chip, the application should call `RTEMS_CAN_CHIP_START` ioctl.

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_START );
```

This operation can be done repeatedly as repeated calls after the chip is started do not have any effect. There is a possibility to start the chip directly from the board support layer with function `rtems_can_chip_start()`.

```
int rtems_can_chip_start( struct can_chip *chip );
```

Starting the chip right after initialization at the board support level brings the advantage of having the controller ready and simplifies the application a bit. However, correct bit timing or modes should be set before the chip starts, otherwise it may cause errors on the bus. Similarly, the chip can be stopped with `RTEMS_CAN_CHIP_STOP` ioctl. This is required if the user wants to reconfigure the bit timing for example.

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_STOP );
```

It is important to check the number of users (applications) using the chip before turning it off as there can be more than one user per chip. The infrastructure allows turning off the controller even if there are other users on it. Read and write calls from other applications return an error in that case.

3.3.5 Controller Related Information

To obtain information about the controller, ioctl call `RTEMS_CAN_CHIP_GET_INFO` may be used. It takes integer argument `info_type` that defines what kind of information user needs. The information is provided as a return value of an ioctl call.

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_GET_INFO, info_type );
```

Following information can be retrieved from the controller. This ioctl also offers another way to obtain the currently set bit rate values. As opposite to the approach presented in section 3.3.2, this way is much easier since there is no need to fill larger structure. However, only bit rate can be obtained this way.

```
RTEMS_CAN_CHIP_BITRATE,
RTEMS_CAN_CHIP_DBITRATE,
RTEMS_CAN_CHIP_NUSERS,
RTEMS_CAN_CHIP_FLAGS,
RTEMS_CAN_CHIP_MODE, and
RTEMS_CAN_CHIP_MODE_SUPPORTED.
```

The defines listed above may be used to obtain information from the controller. It is possible to obtain only one information for one ioctl call. `RTEMS_CAN_CHIP_MODE` and `RTEMS_CAN_CHIP_MODE_SUPPORTED` are used to obtain currently set controller modes and all modes supported by the controller, respectively. Stop command, described in section 3.3.4, may benefit from `RTEMS_CAN_CHIP_NUSERS` providing number of users currently using the controller. Controller's flags obtained by `RTEMS_CAN_CHIP_FLAGS` provide various information including FD capability of the controller, status of the chip (configured, running), and so on.

3.3.6 CAN Frame Representation

One CAN message, called frame in RTEMS, is represented by a structure named `can_frame`. CAN frame header is statically defined by a separate `can_frame_header` structure to simplify the implementation of write operation. It also provides a better starting point for future extension to CAN XL format.

The frame header has an 8 bytes long timestamp, 4 bytes long CAN identifier, 2 bytes long flag field and 2 bytes long field with information about data length. The data field itself is a 64 byte long array with byte access. Only the first 11 bits of the identifier are valid (29 if an extended identifier format is used). Having any of the upper three bits set to one indicates an invalid CAN frame format. The user should check frame's flags to get information if this is not an error frame generated by the controller.

```
struct can_frame_header {
     uint64_t timestamp;
     uint32_t can_id;
     uint16_t flags;
     uint16_t dlen;
};

struct can_frame {
     struct can_frame_header header;
     uint8_t data[CAN_FRAME_MAX_DLEN];
};
```

Names of frame fields were selected to at least partially match the usage in other operating systems like GNU/Linux or NuttX. Compared to SocketCAN implementation, all frame flags are defined only in `flags` field, and the top three bits of the identifier are not used except for additional tagging of error frames returned to user. RTEMS frame format is also unified for both standard CAN and CAN FD (and intended for CAN XL in the future as well), therefore differences between frame formats are defined in `flags` field. The following flags are supported.

```
CAN_FRAME_IDE,
CAN_FRAME_RTR,
CAN_FRAME_ECHO,
CAN_FRAME_LOCAL,
CAN_FRAME_TXERR,
CAN_FRAME_ERR,
CAN_FRAME_FIFO_OVERFLOW,
CAN_FRAME_FDF,
CAN_FRAME_BRS, and
CAN_FRAME_ESI.
```

Flag `CAN_FRAME_IDE` (identifier extended format) is set automatically if CAN frame identifier exceeds 11 bits. Flags `CAN_FRAME_FDF` and `CAN_FRAME_BRS` (if bit rate switch between arbitration and data phase is intended) should be set for CAN FD frame transmission.

Some of these flags are automatically masked for the first queues created during an instance open operation. These include `CAN_FRAME_ECHO` and both error flags `CAN_FRAME_TXERR` and `CAN_FRAME_ERR`. Flag `CAN_FRAME_FIFO_OVERFLOW` is set automatically by the stack for RX frames and can not be filtered out. It indicates FIFO overflow occurred, and some frames on the reception side have been discarded. More specifically, it informs the user there are discarded frames between the frame with `CAN_FRAME_FIFO_OVERFLOW` flag and the previous correctly received frame.

3.3.7 Frame Transmission

The application can pass the message to the framework by calling the standard POSIX `write()` function. The frame is filtered to the correct FIFO queue based on CAN identifier match with queue ID mask (this provides the filtering to priority classes if more queues are used).

It is possible to send only one CAN frame during one `write()` call. Write size of the buffer filled with the frame header and data is specified by `count` parameter. This size can be easily calculated with `can_framsize()` function. The function takes the header size and adds the data length defined in header's `dlen` field. Passing incorrect frame length (less than header size or larger than maximum CAN frame size) results in an error. Using a frame size larger than the actual part holding header `dlen` bytes is acceptable.

```
ssize_t write( int fd, struct can_frame *frame, size_t count );
```

This operation is either blocking or nonblocking based on mode in which the file instance was opened. If opened in blocking mode and target FIFO queue being full, the thread waits until there is a free space in a FIFO (this occurs when some message was released from the FIFO upon successful/error transmission). Nonblocking mode can be used to return from `write` immediately with error value in that case. Value `errno` is set to `EAGAIN`.

In nonblocking mode, the application may utilize TX polling function implemented as `RTEMS_CAN_POLL_TX_READY` ioctl call. The call has `timeout` argument defined in `timespec` structure.

```
ssize_t ioctl( fd, RTEMS_CAN_POLL_TX_READY, &timeout );
```

The thread waits on ioctl call either until there is a free space in any of the FIFO queues or until timeout. The application may also require a confirmation that critical messages were indeed passed from FIFO queue to the network. This functionality is implemented in ioctl call `RTEMS_CAN_WAIT_TX_DONE`.

```
ssize_t ioctl( fd, RTEMS_CAN_WAIT_TX_DONE, &timeout );
```

Once again, it has defined `timeout` as a pointer to `timespec` structure. This call waits until all frames from all FIFO queues are transferred to the network or until timeout.

3.3.8 Frame Reception

Applications can access received messages with standard POSIX call `read()`. The call returns an error if the read size specified by count is less than the length of the frame header. It is possible to read only one frame with one `read` call.

```
ssize_t read( int fd, struct can_frame *frame, size_t count );
```

Once again, this operation is either blocking or nonblocking based on file descriptor mode. Blocking mode waits indefinitely until there is a RX message available in the queue, nonblocking mode returns with error if no message is to be read at the moment.

The parameter `count` gives the read size in bytes. The entire size of CAN frame (use operator `sizeof( struct can_frame )`) might be used to ensure all frames are received, but the user may also specify a smaller read size. If this size is less than the length of the frame, only the number of bytes specified in `count` is copied to `frame` and `EMSGSIZE` errno is set. The user can then obtain required length from frame header. It is not possible to request a read of fewer bytes than the size of the header.

The application can implement polling function on reception side using ioctl call `RTEMS_CAN_POLL_RX_AVAIL`. This call has a defined `timeout` argument, with `timespec` structure as previous calls, and waits until there is an available message or until timeout.

```
ssize_t ioctl( fd, RTEMS_CAN_POLL_RX_AVAIL, &timeout );
```

3.3.9 Hardware Timestamping

Some controllers may provide support for precise hardware timestamping. The infrastructure provides an ioctl call via which the application may obtain this value. The argument is a pointer to `uint64_t` timestamp value.

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_GET_TIMESTAMP, &timestamp );
```

3.3.10 Controller's Statistics

The controller device driver can keep track of CAN frame statistics. This includes a number of successfully sent/received frames, bytes or number of errors. The number of RX overflows, i.e. frames not received because the hardware RX buffer is already full, can also be tracked.

The statistics can be obtained by ioctl call `RTEMS_CAN_CHIP_STATISTICS` with argument `statistics` providing a pointer to `can_stats` structure. Support of statistics tracking is controller dependent, although well written drivers should provide support for it.

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_STATISTICS, &statistics );

struct can_stats {
  unsigned long tx_done;
  unsigned long rx_done;
  unsigned long tx_bytes;
  unsigned long rx_bytes;
  unsigned long tx_error;
  unsigned long rx_error;
  unsigned long rx_overflows;
  uint8_t chip_state;
};
```

3.4 Controller Initialization

CAN controller has to be initialized (typically allocation of resources and initial setup of controller's registers) and register as a character device before being used from an application via API described in the section 3.3. Although the operation is target dependent, it is usually done in two steps.

The first step is the initial configuration and initialization of a specific CAN controller (i.e. CTU CAN FD, SJA1000 or MCAN). The presented stack requires the user to call controller specific function to handle this step. This function, often called `chipname_initialize` (where `chipname` is the name of the controller), allocates and returns `can_chip` structure if successful and provides initial setup of the controller. The function should also fill the `can_chip` structure fields with the correct default values. This includes controller's frequency, default flags, modes, supported modes, bit timing constants and so on. This step is important to provide correct support of various ioctl calls presented in previous sections.

System resources like semaphores, interrupt handlers or task threads are also allocated here. However, since the function is controller specific, the implementation and usage may slightly differ for each chip.

The structure also has a pointer to controller specific structure, also allocated during initialization. This is controller specific and differs from implementation to implementation. It is usually used to define controller unique features that do not belong to the generic structure. It also should be used only in the controller's driver itself as it is not a good practice to put target dependent parts into a portable application.

The initialized controller then has to be registered as a character device into `/dev` namespace. Function `can_bus_register()` takes care of this operation. Standard naming `canX` is used in the examples, but it is possible to select a different name scheme per application requirements. The entire process of initialization and registration is demonstrated on virtual CAN controller in the code below. Note that the code also has to include `can-bus.h` header for `can_bus` structure definition and `can_bus_register()` function declaration. Controller dependent header is also required for the initialization function.

```
/* Allocate can_bus structure */
struct can_bus bus = malloc( sizeof( struct can_bus ) );

/* Initialize virtual CAN controller */
bus->chip = virtual_initialize();

/* Register controller as dev/can0, returns 0 on success */
int ret = can_bus_register( bus, "dev/can0" );
```

Note that the user has to specify the path to which the controller is registered, and this path has to be unique. Chip specific function `xxx_initialize()` may also have different input parameters for different chips or can even have a different name according to chip specific implementation.

The process of initialization and registration can be either implemented in the board support package and done during board startup or in an application itself. However, the latter complicates the application's portability to different controllers.

3.5 Driver Interface

Every controller driver has to interact with the infrastructure to receive messages from FIFO queues (transmission from an application point of view) and to send messages back (reception from an application point of view). To achieve this, the controller holds ends (a container or a graph node) to which edges are connected. These ends should be initialized during controller initialization described in section 3.4. Function `canqueue_ends_init_chip()` is provided to initialize the controller's side ends.

```
int canqueue_ends_init_chip (
  struct canque_ends_dev_t *qends_dev,
  struct can_chip *chip,
  rtems_binary_semaphore worker_sem
);
```

This function also assigns `worker_sem`, a self-contained binary semaphore (refer to section 2.2.3). It is used by the framework to inform the controller's side about new message being available in FIFO queues. A controller can also use this semaphore to trigger its worker thread in response to an interrupt coming from the hardware.

Driver should also register several functions used by ioctl calls (if supported). These are assigned through `can_chip_ops` structure. Not used functions should be assigned a `NULL` value.

It is also possible to have driver specific ioctl calls. The common layer passes all calls that are not recognized to controller's specific function (if assigned in `can_chip_ops` structure) and let the controller deal with them. This way options unique to just one controller (not defined in control modes for example) can be set up during its implementation and registration or even based on application requirements.

All functions and structures required for the driver implementation are included with `can-devcommon.h` header file. The driver should therefore include this header to utilize all functions mentioned in this section.

3.5.1 Frame Transmission

The controller has to retrieve a message from one of the FIFO queues to transmit it to the network. The information about new message being available in FIFO queues is announced to the controller through the already mentioned binary semaphore. The process of waiting for the semaphore varies based on the controller's implementation. One of the possible designs is the implementation of worker thread as an infinite loop and waiting for a semaphore there. The thread can manage frame transmission and reception once a semaphore is posted.

The message can be retrieved from the framework with `canque_test_outslot()` function. This function retrieves the oldest ready slot from the highest priority edge (FIFO queue) and, therefore tries to send high priority class first. It is implemented as a for loop iterating through all queues starting with the highest priority one. This is fitting for the current maximum number of 3 priorities per opened instance, but it would not be optimal if more classes were used. The addition of bit map, holding active classes in it, and iterating only through active classes would be a fitting extension if more than 3 classes were used.

```
int canque_test_outslot(
  struct canque_ends_t *qends,
  struct canque_edge_t **qedgep,
```

```
  struct canque_slot_t **slotp
);
```

Once retrieved from the framework, the CAN frame (represented by slot structure `canque_slot_t`) can be inserted into the controller's hardware buffer and sent to the network. The `canque_test_outslot()` does not free the slot's space in the FIFO, but holds it until the transmission is finished. This is an important feature of the stack that allows hardware buffer abort and slot transmission rescheduling. The controller should inform the framework by calling `canque_free_outslot()` function upon successful transmission of the message to the network or in case of transmission error. The function frees the allocated slot in the FIFO queue.

```
int canque_free_outslot(
  struct canque_ends_t *qends,
  struct canque_edge_t *qedge,
  struct canque_slot_t *slot
);
```

The framework also provides a unique feature to push slots back to the correct FIFO and schedule the slot for later processing. This is useful in case of transmission abort. The abort might be used 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 for example. Scheduling for later processing is implemented in `canque_again_outslot()` function.

```
int canque_again_outslot(
  struct canque_ends_t *qends,
  struct canque_edge_t *qedge,
  struct canque_slot_t *slot
);
```

The previously described function `canque_test_outslot()` retrieves the slot from the FIFO once called. This is not convenient in case there is no free space in the controller's hardware TX buffers. The preferable option would be to just check whether there is some pending message from a higher priority class compared to priority classes inserted in buffers. This can be done with `canque_pending_outslot_prio()`.

```
int canque_pending_outslot_prio(
  struct canque_ends_t *qends,
  int prio_min
);
```

The minimal priority of the FIFO queue to be considered is defined by `prio_min` argument. The controller should deal accordingly with pending frame from a higher priority class to avoid priority inversion problem. This is in detail described in chapter 4 and section 5.2.2.

3.5.2 Frame Reception

Frame reception is usually triggered in a controller with RX interrupt. The driver should read the message, assign correct flags to structure `can_frame` and then pass the frame to FIFO queues. Function `canque_filter_frame2edges()` is prepared for this.

```
int canque_filter_frame2edges(
  struct canque_ends_t *qends,
```

```
    struct canque_edge_t *src_edge,
    struct can_frame *frame,
    unsigned int flags2add
);
```

Argument `src_edge` defines optional source edge for echo detection, argument `flags2add` optional additional flags. These are omitted for standard CAN frame reception, but can be used for sending echo frame upon successful transmission or for the case of error frames reporting for example.

The usage of mutexes – refer to section 2.2.4 for more details – in the infrastructure does not allow to fill the queues, i.e. call `canque_filter_frame2edges()` function, directly from an interrupt handler. Instead, the controller has to use a dedicated thread for this operation. The example of such a worker thread used in CTU CAN FD driver is described in section 5.2.1.

The frame structure presented in section 3.3.6 holds the length of the data in bytes. This is much more comfortable for application usage, but the controller has to convert this value to 4 bit data length code before sending it to the network. Function `rtems_canfd_len2dlc()` provides a unified way to do this and supports conversion for both standard and FD frames.

```
static inline uint8_t rtems_canfd_len2dlc( uint8_t len );
```

The return value of the function is the calculated data length code. The conversion is done with the lookup table for all possible values.

3.6 Error Reporting

CAN frame has two flags that can be used for error reporting: `CAN_FRAME_TXERR` and `CAN_FRAME_ERR`. The first one is used to report frame transmission error. This error might be caused by various errors in the controller's hardware transmission buffer and is controller dependent. In this case, the controller should send the frame that caused the error back to its opened instance – source edge argument of `canque_filter_frame2edges` function described in section 3.5.2 might be used – with added `CAN_FRAME_TXERR` flag. Therefore, the original message, its identifier, and flags shall remain the same and only additional flag is inserted.

The infrastructure sends the transmission error frame only to the first available edge (matching identifier range and flags) of a given opened instance. This is to avoid error frames spamming multiple reception queues. Error flag `CAN_FRAME_TXERR` is filtered out by default queue settings.

The rest of the errors are sent to the application with `CAN_FRAME_ERR`. This includes error counter (active, passive, and bus off status), RX buffer overflows, bus errors, arbitration lost errors, and so on. These errors, their detection, and handling are controller specific. However, the stack defines the format in which CAN error frame should be sent to the application. This format is partially taken from SocketCAN implementation to provide easier understanding, but there are some differences.

Error frame format has the flag set to `CAN_FRAME_ERR` value. It is not forbidden to have other flags present as well, but these might be masked out for some queues, so caution is in place. Since error frame only contains information about the error and not about the frame that caused it (it even might not know what frame caused it), other fields can be freely used to pass error information.

CAN identifier is used to inform about error type. This field can have more error types present (for example both controller error and arbitration lost error) if needed. The stack also sets the 31st bit of CAN identifier to a logical one, notifying the application the identifier is invalid concerning the standard CAN frame. The following errors types are supported:

```
CAN_ERR_TXTIMEOUT /* TX timeout */,
CAN_ERR_LOSTARB   /* lost arbitration */,
CAN_ERR_CRTL      /* controller problems */,
CAN_ERR_PROT      /* protocol violations */,
CAN_ERR_TRX       /* transceiver status  */,
CAN_ERR_ACK       /* received no ACK on transmission */,
CAN_ERR_BUSOFF    /* bus off */,
CAN_ERR_BUSERROR  /* bus error */,
CAN_ERR_RESTARTED /* controller restarted */, and
CAN_ERR_CNT       /* error counter values */.
```

Having error types located in CAN frame identifier brings the possibility to create new RX queues with identifier mask set in such a way that only some of these errors are propagated to the application. Additional information providing a deeper description of raised errors are also available in data fields for some error types. Only a standard frame with 8 byte long data field is used.

The first byte (8 bits) of data – data[0] – field keeps detailed information regarding lost arbitration error (`CAN_ERR_LOSTARB`). This just informs in what bit the arbitration was lost. Another field stores controller related problems (`CAN_ERR_CRTL`). This includes RX or TX overflows and the controller changing its error state (error active, warning, passive).

Protocol related violations (`CAN_ERR_PROT`) are stored in data[2] and data[3] fields. The first mentioned informs what kind of violation is present. This may be incorrect bit stuffing, controller incapability to generate dominant, or recessive bit or bus overload for example. The following field provides the location of this violation.

Transceiver status (`CAN_ERR_TRX`) is located in data[4] field. This is used to report hardware layer issues such as missing wire or wire being short-circuited to ground or supply voltage. The sixth data field – data[5] – is reserved and not used. The infrastructure also reports a number of the current values of TX and RX error counters (`CAN_ERR_CNT`). These data are passed through data[6] and data[7] fields for transmission and reception, respectively.

3.7 Source Code Organization

The implementation of RTEMS CAN/CAN FD stack is separated into several source code files and header files. The latter will be located in `cpukit/include/dev/can` directory once the stack is merged to the RTEMS mainline, and source codes will be located in `cpukit/dev/can/` folder. The directory selection is consistent with other device drivers as SPI or I2C.

Accessible FIFO queue functions (most of them presented in section 3.5) are implemented in `can-queue.c` source file with a header named `can-queue.h`. The header file defines both queues and ends structures and also static inline functions used for FIFO queue implementation.

Initialization of kernel (character device) side of queues is located in `can-quekern.c` file, driver (controller) side in `can-devcommon.c`. The latter includes header `can-devcommon.h`[1] providing structure definitions and function declarations used for controller implementation. This includes the already presented structure `can_chip` for example. Apart from its C file, the header is also supposed to be included from source files implementing the controller. The general bus functions as open/read/write/ioctl operations are implemented in `can-bus.c` file to match the organization of SPI or I2C stacks. Header `can-bus.h`[2] declares CAN bus registration function described in section 3.4 and `can_bus` structure. It should be included from the BSP layer for the registration of the controller.

Functions used for bit timing calculation from a given bit rate are located in `can-bittiming.c` file. The infrastructure also provides a simple virtual CAN controller that can be used for stack testing and also serves as an example for future controllers. This virtual driver is located in `can-virtual.c` file.

Headers `can-queue.h` and `can-devcommon.h` were already introduced. From an application perspective, the most important include is `can.h`[3]. This file provides all defines, ioctl calls, and structures required to operate with the infrastructure from application layer. It also includes other headers providing CAN frame definition, filter structure, statistics or bit timing. This header file is the only required one to operate with CAN bus character device through standard POSIX calls.

CAN frame structure is written in `can-frame.h` file, filter structure is located in `can-filter.h` header, statistics are implemented in `can-stats.h` and bit timing structures in `can-bittiming.h`. Detail description of CTU CAN FD related files is available in section 5.2.

[1] `https://otrees.pages.fel.cvut.cz/rtems/rtems-canfd/doc/doxygen/html/can-devcommon_8h.html`

[2] `https://otrees.pages.fel.cvut.cz/rtems/rtems-canfd/doc/doxygen/html/can-bus_8h.html`

[3] `https://otrees.pages.fel.cvut.cz/rtems/rtems-canfd/doc/doxygen/html/can_8h.html`

Chapter 4
CAN Bus Priority Inversion Problem

This chapter introduces CAN bus arbitration priority inversion problem to the reader and proposes a solution by introducing the TX buffers dynamic allocation mechanism. This chapter deals with the theoretical part of the problem and its solution. The implementation of presented algorithm to CTU CAN FD target controller is described in the following chapter, results can be found in section 6.2.

4.1 Introduction

CAN bus arbitration phase ensures the message with the highest priority (i.e. the lowest identifier) preempts the messages with lower priority (i.e. higher identifier). A problem occurs when the bus is fully saturated by middle priority messages from one or more controllers and a mix of low and high priority messages is pending on the other controller. If all controller's hardware TX buffers are filled with these low priority messages, middle priority messages will always take precedence over low priority ones. This means that high priority messages will not be sent until other controllers stop or pause their messages stream (or indefinitely).

The problem, when low priority message, correctly preempted by middle priority message, is holding a resource (hardware TX buffers) required by high priority message and thus indirectly delaying the execution of it, is called priority inversion in computer science. The problem mostly occurs in task scheduling, however it can affect CAN bus as well.

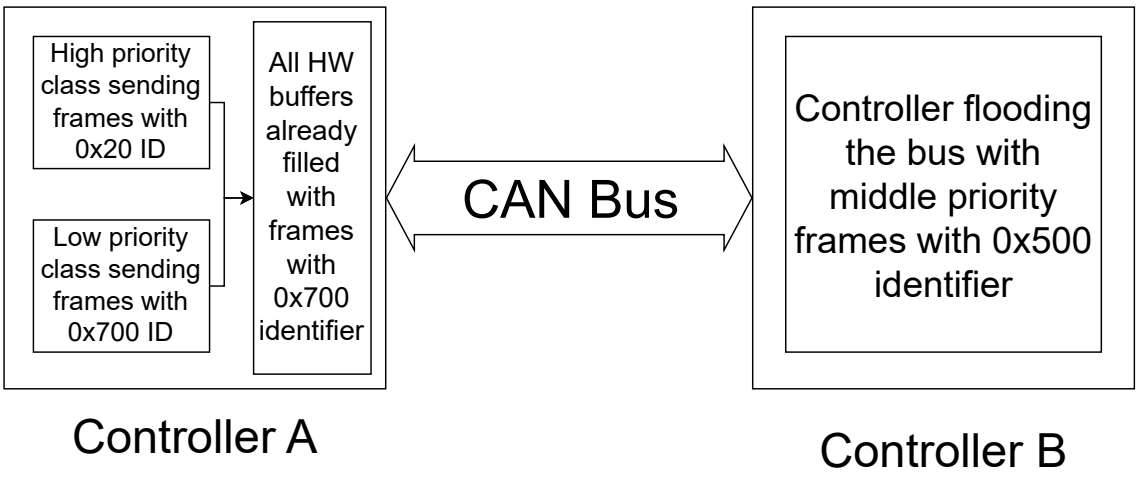

Figure 4.1. Visualization of CAN Bus Arbitration Priority Inversion.

The priority inversion problem is visualized in Figure 4.1. In this visualization, controller B floods the bus with `0x500` identifier while controller A tries to send a mix of `0x20` and `0x700` identifiers. However, if all hardware buffers are already filled with low priority `0x700` identifier, high priority `0x20` will never get to the arbitration phase and is incorrectly held by the middle priority message on the bus as a result.

4.2 Common Solutions

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.

However, application and sometimes even protocol implementations usually require the preservation of message order, even if different CAN identifiers are used. This common requirement disqualifies the message 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.

4.3 Dynamic Allocation of TX Buffers to Multiple Priority Groups

The priority inversion problem described above can be solved by adding the dynamic redistribution of CAN transmission buffers to priority classes. This approach therefore keeps the common solution of assigning CAN messages to priority classes based on identifier range and having a FIFO queue for each class.

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.

If there is a free space available in hardware buffers, the message is inserted into the free one. The insertion of the new message into one of the buffers is followed by transmit sequence reorganization to ensure the correct transmission order based on both priority class and application defined order. In general, the buffer with newly inserted message has to be placed 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 from 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.

If no space is available, the controller checks whether a message of lower priority class occupies the buffers. If yes, the last message of the lowest priority class occupying TX buffer is replaced by the pending message and scheduled for later processing. Transmit sequence reorganization follows if the new message is inserted. The message replacement poses some requirements on both CAN stack and controller. The stack has to support pushback messages and must schedule them for later processing, so they are not lost, and the controller must be able to abort TX buffer processing if needed. The algorithm requirements on both the software stack and CAN controller are discussed in detail in section 4.3.3.

4.3.1 Priority Classes Mapping

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. The priority classes are

mapped as continuous ranges in this array using TX order tail array where each entry represents the tail index in the transmission sequence array for a given priority class.

The tail points one position beyond the last allocated slot for the corresponding class. Only the tail is needed as the head for the highest priority class is fixed and heads for lower priority classes are at the exactly same position as the previous priority class tail. 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.

4.3.2 Example

This example is provided for CAN stack with three priority classes with priorities 0 (low), 1 (middle), and 2 (high) and a controller with 4 HW buffers. The application(s) passed four messages to the controller in this order: low priority message, high priority message, low priority message and high priority message.

The relationship between hardware buffers, SW queues, and the usage of TX order array mapping is visualized in Figure 4.2. The messages are inserted into the following hardware buffers: first low priority message to buffer 0, first high priority message to buffer 3, second low priority message to buffer 2, and second high priority message to buffer 1. The buffer order does not match the insertion order because other messages might have been inserted into those buffers previously, and therefore they were replaced in different order. Moreover, the buffer numbering is not important here as the transmission sequence is defined TX order array.

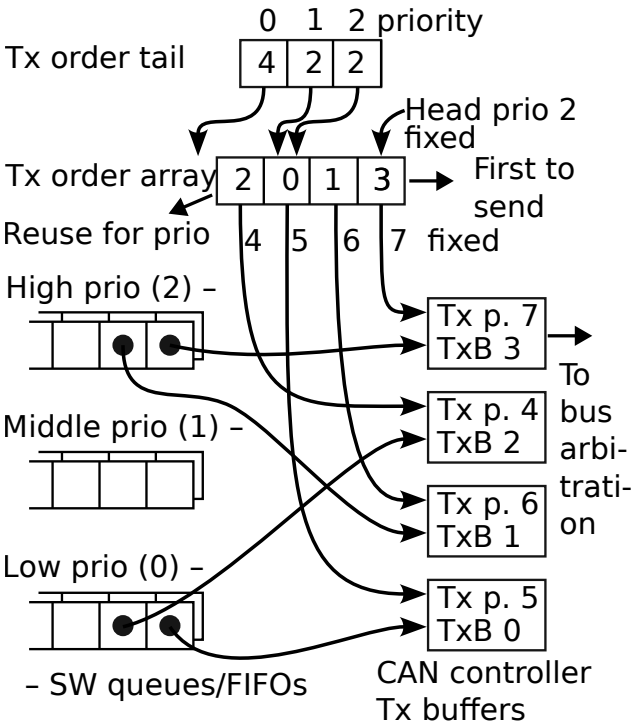

Figure 4.2. Visualization of dynamic allocation of TX buffers.

Without rotation of buffer priorities, this insertion order 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 2 would then be reused for the new message. The situation, where low priority message in buffer 2 was replaced by middle priority message and transmission order was correctly reorganized, is depicted in Figure 4.3

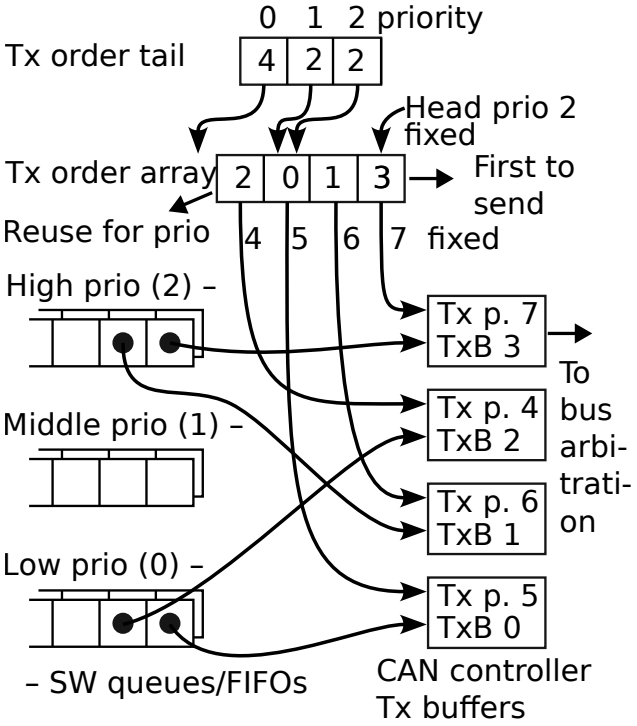

Figure 4.3. Visualization of dynamic allocation of TX buffers after middle priority frame insertion.

The results of presented mechanism and demonstration on real hardware are available in section 6.2. The mechanism is presented on RTEMS CAN infrastructure and CTU CAN FD core on educational kit MicroZed APO.

4.3.3 Algorithm Requirements

While the algorithm is generic and not tied to the presented infrastructure or CTU CAN FD controller, there are some requirements both stack and controller have to fulfil to use this solution. This section presents them and provides some possible workarounds if available.

An obvious requirement for CAN stack is the support of multiple priority queues. The number of queues is up to stack implementations, but at least three possible queues should be considered to satisfy application requirements. The stack also has to provide

a feature to return frames passed to the controller back to the FIFO in case of buffer abort. The returned message should be scheduled for later processing. In general, this means the stack has to keep the slot allocated until notified of successful transmission or error. This, in fact, is also done in SocketCAN, therefore possible extension to Linux kernel should be possible.

From the controller side, the requirements might be a bit more demanding. There must be a way to determine the order in which the transmission hardware buffers should be sent to the network. This might be a problem for some controllers that do not support local priority for message buffers but only consider the identifier of the inserted frame or even just fixed FIFO order. The possibility to change the priorities of hardware buffers on the fly – without deactivating them – provides a huge advantage for the presented algorithm. Without this feature, the controller would have to disable all buffers for which the priority shall be changed, modify the value, and enable them again. This might cause a small delay between frame transmissions. Hardware buffer abort feature is also required, but this is supported by most of the current controllers.

Chapter 5
CTU CAN FD Driver

This chapter describes open source CTU CAN FD core developed by Ondrej Ille initially at the Department of Measurement of FEE at CTU under the lead of Jiří Novák and later self-funded. The IP core was later rearchitected to functional general purpose CAN/CAN FD controller and equipped with TX priority register used for cyclic FIFO in the frame of the CAN FD Open Cores Support Linux Kernel Based Systems project funded by Digiteq Automotive led by Pavel Píša as the solution architect at the Department of Control Engineering of FEE at CTU. The support for the IP core is also included in Linux kernel mainline [7].

Short introduction of the core specification is followed by the description of its implementation to RTEMS executive.

5.1 Core Specification

The developed core is a softcore written in VHDL without vendor-specific libraries. It can be configured with up to 8 independent TX buffers for messages with each buffer having 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 a different priority and those priorities are rotated after the transmission is completed [8].

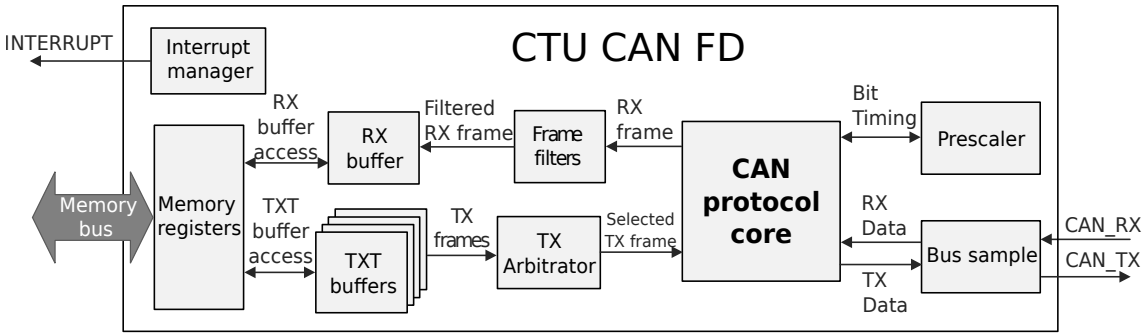

Figure 5.1. CTU CAN FD IP core structure (Source: [8]).

The core is compliant with ISO 11898-1:2015 standard and supports CAN FD communication protocol. It is equipped with a 64-bit clock counter common to all CAN FD interfaces and provides timestamp with 10 nanoseconds resolution [9] for the case of Zynq integration.

5.2 Implementation to RTEMS

This core was chosen for first RTEMS implementation 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. This capability is useful for dynamic allocation

of TX buffers algorithm presented in the previous chapter. The possibility to measure precise timestamp is also useful for latency testing.

The driver is implemented in `ctucanfd` subdirectory as the implementation is split into more files. This approach is also used by Linux kernel for example. The core of the controller is implemented in `ctucanfd.c`, rest of the code is placed in header files. This includes `ctucanfd_priv.h` defining CTU CAN FD controller private structure, `ctucanfd_kframe.h` mapping frame format to CTU CAN FD registers, `ctucanfd_kregs.h` defining registers' offsets and fields and `ctucanfd_txb.h` with `static inline` functions for hardware buffer rotations. The first two files are autogenerated and taken from Linux kernel with the license being changed to more permissive one on approval from all original authors.

The entry point for CTU CAN FD controller initialization is the function `ctucanfd_initialize()` defined in installed header `ctucanfd.h`.

```
struct can_chip *ctucanfd_initialize(
  uint32_t addr,
  rtems_vector_number irq,
  int ntxbufs,
  rtems_option irq_option,
  unsigned long can_clk_rate
)+
```

The caller has to pass base address to which registers are mapped, interrupt number interrupt, number of transmission hardware buffers, setup and clock rate as arguments. Interrupt can be setup to either shared (`RTEMS_INTERRUPT_SHARED`) or unique (`RTEMS_INTERRUPT_UNIQUE`). The latter causes less overhead, but can not be used if more controllers are attached to the same interrupt number.

This function initializes CTU CAN FD private structure and generic CAN structure `can_chip`. Interrupts and semaphores are also initialized here with worker task being started as well. This task takes care of frames transmission and interrupt handling. The chip is not started yet as already described in 3.3.4; this has to be handled either from application with proper ioctl call or function `rtems_can_chip_start()` can be called. However, correct bit timing and modes should be set before starting the controller. CAN FD mode is already set by default by the initialization function, but other modes are disabled.

5.2.1 Worker Thread

It was already mentioned the transmission and interrupt handing is done from a worker thread. This thread is implemented in the function `ctucanfd_worker()`. It is an infinite while loop waiting on a binary semaphore. This semaphore is posted either from the infrastructure if a new frame is to be sent or from an interrupt handler if a new interrupt is received. Therefore, the interrupt handler itself just disables interrupts and releases the worker execution through the semaphore.

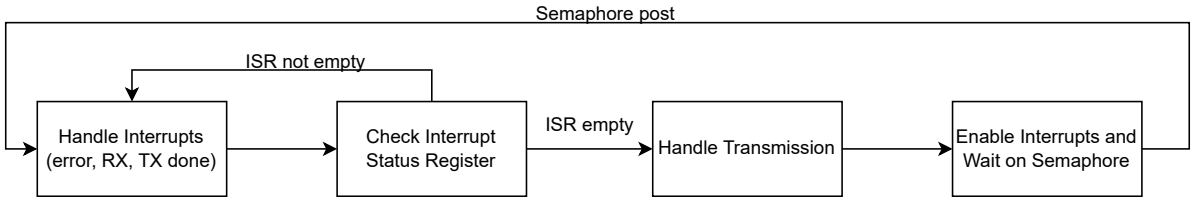

Figure 5.2. Simplified visualization of CTU CAN FD worker thread.

The simplified logic of the worker thread is depicted in Figure 5.2. It can be seen the worker first handles all interrupts (RX frame available, TX interrupt, error interrupt) and once they are all done it checks whether there is a new frame to be transmitted.

RX frame available interrupt leads to new frame reception. This process is implemented in functions `ctucanfd_receive()` and `ctucanfd_read_rx_frame()` with the latter one also passing the received frame to the infrastructure edges. Transmission interrupt reports a change in the status of TX buffers. This way the driver determines whether the message was transmitted successfully, aborted by the controller or ended with an error.

In case of successful transfer, the frame is echoed back to opened file instances if configured by the infrastructure. The frame's slot in the FIFO queue is cleared at this moment as well as the hardware TX buffer used for frame transmission. Aborted frame can not be cleared from FIFO queue, but has to be kept there and rescheduled for later transmission. In case of transmission ending with error, both FIFO outslot and HW buffer are cleared.

The transmission has to be handled differently based on whether there is a free space in some hardware transmission buffer or not. If yes, the controller just adds a new frame to one of the free TX buffers (function `ctucanfd_insert_frame()`) and reorganizes the order array (function `ctucanfd_txb_add()`) as already describes in 4. The reorganization is necessary, so the oldest frame from the highest priority class is processed in the next arbitration phase.

The implementation is a bit different in case there is no free buffer to fill the message into. In that case, the controller has to check whether any of the pending frames (frames the infrastructure wants to pass to the hardware buffer) is from a higher priority class than the frames already inserted in hardware buffers (function `canque_pending_outslot_prio()`). If yes, it has to push the correct frame from the buffer back to the FIFO with the abort command executed on the buffer. TX interrupt handles the push back as already described.

5.2.2 Mapping Priority Classes to Buffers

As already discussed in 4.3.1, priority classes have to be mapped to hardware transmission buffers in order to support the dynamic redistribution and avoid possible priority inversion. Section 4.3.1 presented the generic approach with TX order represented with an array. However, this can be enhanced if the controller has up to 8 hardware buffers as 32-bit large unsigned integer can be used to represent the order in which the buffers are sent. This is because 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. TX order tail still has to be implemented as an array thought. The implementation of promotion and demotion with 32-bit unsigned integer can be seen below.

```
#define TXT_DONE     0x4
#define TXT_BF       4
#define TXT_MASK     0xf
#define TXT_ANY_DONE ( ( TXT_DONE << ( 0 * TXT_BF ) ) | \
                       ( TXT_DONE << ( 1 * TXT_BF ) ) | \
                       ( TXT_DONE << ( 2 * TXT_BF ) ) | \
                       ( TXT_DONE << ( 3 * TXT_BF ) ) | \
```

```
                        ( TXT_DONE << ( 4 * TXT_BF ) ) | \
                        ( TXT_DONE << ( 5 * TXT_BF ) ) | \
                        ( TXT_DONE << ( 6 * TXT_BF ) ) | \
                        ( TXT_DONE << ( 7 * TXT_BF ) ) )

#define TXB_BF 4
#define TXB_MASK 0xf
#define TXB_ALL 0xffffffff
#define TXB_SH( idx ) ( ( idx ) * TXB_BF )
```

Firstly, there is a set of defines to check all TX buffer statuses in `CTUCANFD_TX_STATUS` core registers (matches TXT prefix). The `TXB_BF` specifies that four bits are used to store the state, priority and order of single TX buffer. TXB_MASK represents bit-mask of corresponding size and `TXB_SH` bit shift to given index in the array represented by nibbles in 32-bit unsigned integer.

Slot demotion is used if the frame is removed from the hardware buffer. In that case, it has to be demoted to the last position. For example, if TX buffer order is `0x76543210` and `CTUCANFD_TX_PRIORITY` is set to `0x01234567`, then the state after buffer 0 is sent will be `0x07654321` with value `0x12345670` in `CTUCANFD_TX_PRIORITY` register. This way buffer 1 will have the highest priority (7) and buffer 0 the lowest (0).

```
static inline uint32_t ctucanfd_txb_slot_demote(
  uint32_t txb_order,
  int from,
  int to
)
{
  uint32_t txb_order_new;
  uint32_t txb_move = ( txb_order >> TXB_SH( from ) ) & TXB_MASK;
  uint32_t mask_from = TXB_ALL << TXB_SH( from );
  uint32_t mask_to = TXB_ALL << TXB_SH( to );
  txb_order_new = txb_move << TXB_SH( to );
  txb_order_new |= txb_order & ( ~mask_from | ( mask_to << TXB_BF ) );
  txb_order_new |= ( ( txb_order >> TXB_BF) & ~mask_to ) & mask_from;
  return txb_order_new;
}
```

The opposite to `ctucanfd_txb_slot_demote()` function is slot promotion by `ctucanfd_txb_slot_promote()`. This function is used if a new message is added to one of the buffers. The transmission order has to be reorganized to reflect priority classes and the order determined by the application. This generally means placing the buffer to the tail of its priority class.

```
static inline uint32_t ctucanfd_txb_slot_promote(
  uint32_t txb_order,
  int from,
  int to
)
{
  uint32_t txb_order_new;
  uint32_t txb_move = ( txb_order >> TXB_SH( from ) ) & TXB_MASK;
  uint32_t mask_from = TXB_ALL << TXB_SH( from );
```

```
  uint32_t mask_to = TXB_ALL << TXB_SH( to );
  txb_order_new = txb_move << TXB_SH( to );
  txb_order_new |= ( txb_order ) & ( ~mask_to |
                   ( mask_from << TXB_BF ) );
  txb_order_new |= ( ( (txb_order & mask_to ) & ~mask_from ) << TXB_BF );
  return txb_order_new;
}
```

The implementation of these functions can be found in `ctucanfd_txb.h` header file. The functions are currently implemented only for CTU CAN FD driver, but they might be used for other controllers with up to 8 hardware buffers and the possibility to assign them a different priority. Controllers with more than 8 hardware buffers shall use standard array operations.

5.2.3 Timestamping

CTU CAN FD provides precise hardware timestamping with 10 nanoseconds resolution and a counter with 64 bits. This value is global for all cores in the design, therefore timestamp values can be compared even between separate cores.

The timestamp value is split into two separate registers, CTUCANFD_TIMESTAMP_HIGH providing upper 32 bits and CTUCANFD_TIMESTAMP_LOW providing lower 32 bits. The correct way to read those registers is to start with the upper value, then read the lower value and then again the upper value. If the new upper value does not match the previously read one, an overflow occurred and both upper and lower values should be read again. The simplified implementation of `ctucanfd_get_timestamp()` function can be seen below.

```
upper = ctucanfd_read32( priv, CTUCANFD_TIMESTAMP_HIGH );
lower = ctucanfd_read32( priv, CTUCANFD_TIMESTAMP_LOW );
upper_check = ctucanfd_read32( priv, CTUCANFD_TIMESTAMP_HIGH );
if ( upper != upper_check ) {
  upper = ctucanfd_read32( priv, CTUCANFD_TIMESTAMP_HIGH );
  lower = ctucanfd_read32( priv, CTUCANFD_TIMESTAMP_LOW );
}

*timestamp = ( ( uint64_t )upper << 32) | lower;
```

This function is routed to a common ioctl call `RTEMS_CAN_CHIP_GET_TIMESTAMP` described in section 3.3.9. It is also used to timestamp newly received frames.

Chapter 6
Results and Demonstrations

This chapter provides results and demonstrations of RTEMS CAN/CAN FD stack developed in scope of this thesis and thoroughly described in chapter 3. Various latency profiles were measured to evaluate the stack performance. This chapter also briefly describes used test applications and demonstration with pysimCoder rapid development tool.

All tests and demonstrations were done on educational kit MicroZed APO. This kit is used in many computer science coursers at CTU FEE and is based on MicroZed evaluation kit with Xilinx Zynq-7000 system on chip.

6.1 RTEMS CAN Stack Mutual Latency Profiles

RTEMS CAN stack latencies were partially tested on LaTester infrastructure originally developed for continuous CAN bus latency testing on Linux kernel [10]. In this setup, the device under test (DUT) functions as a CAN gateway: receives messages from CAN bus A and instantly sends it to CAN bus B. The measuring system generates and receives CAN/CAN FD frames and provides precise timestamping measurement. The connection of this framework is available in Figure 6.1.

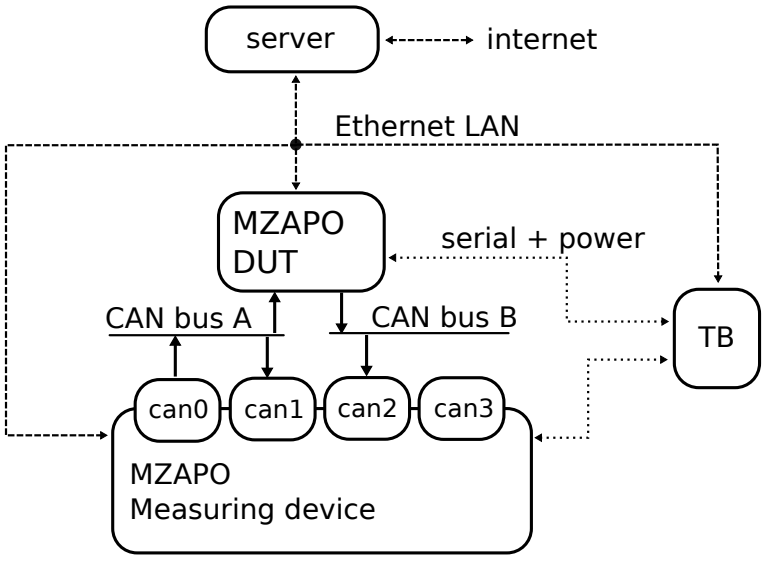

Figure 6.1. CAN LaTester connection scheme (Source: [7]).

The latency measuring starts once CAN frame is sent to the bus and ends once the entire message is received. This is depicted in Figure 6.2. The length of the message on the link at given nominal and data bit rates is subtracted from the latency. The final result is the time it takes the DUT to read received message from CAN bus A controller, process it and forward it into TX buffer of the controller connected to CAN bus B. In other words, it is a time the frame spent in software stack [7] [9]. The advantage

of using the same hardware and framework as for Linux kernel tests is possibility to compare RTEMS and GNU/Linux CAN latencies, respectively character device stack and SocketCAN latencies, in the future.

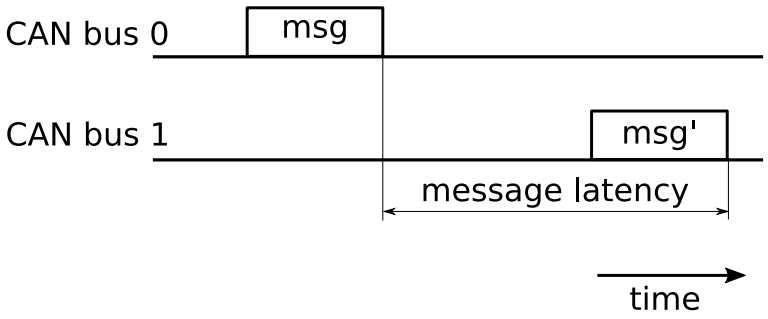

Figure 6.2. CAN message latency measurement (Source: [7]).

The testing framework has several options for which the latencies can be measured. Both standard and FD frames can be tested and modes `once` and `flood` are available. The latter floods the bus, while mode `once` always waits for the message to be received back before sending another one. Measured latencies for system without any other tasks (such as network) can be seen in Figure 6.3.

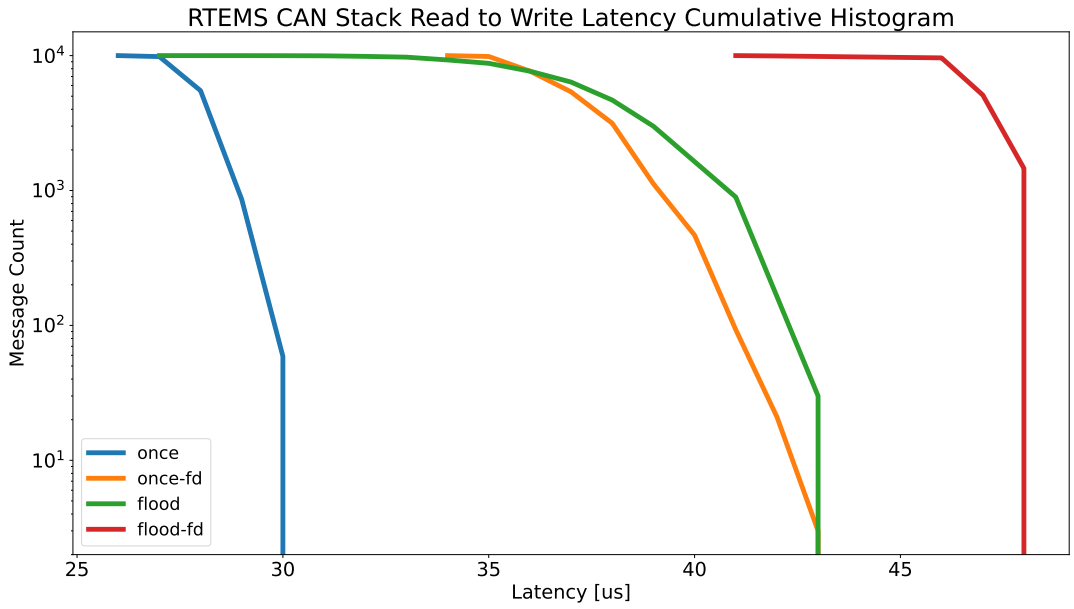

Figure 6.3. RTEMS CAN stack read to write latency profile.

Another latency measurement was done with enabled networking stack and MicroZed APO board hosting a simple HTTP web server displaying dummy data. This was done to provide some way to stress the networking stack. Priorities of CAN tasks were selected higher compared to networking and web server tasks.

Two types of measurements were done for enabled networking. At first, latencies were measured with enabled networking stack and running web server. Then, the server was stress tested with `siege` tool providing additional load for the networking stack. Since CAN priorities were selected the highest, the expectation for the measurement without web server stress are the latencies approximately the same as without networking stack

at all. These values should get worse if hosted HTTP web server is stress loaded by an external tool.

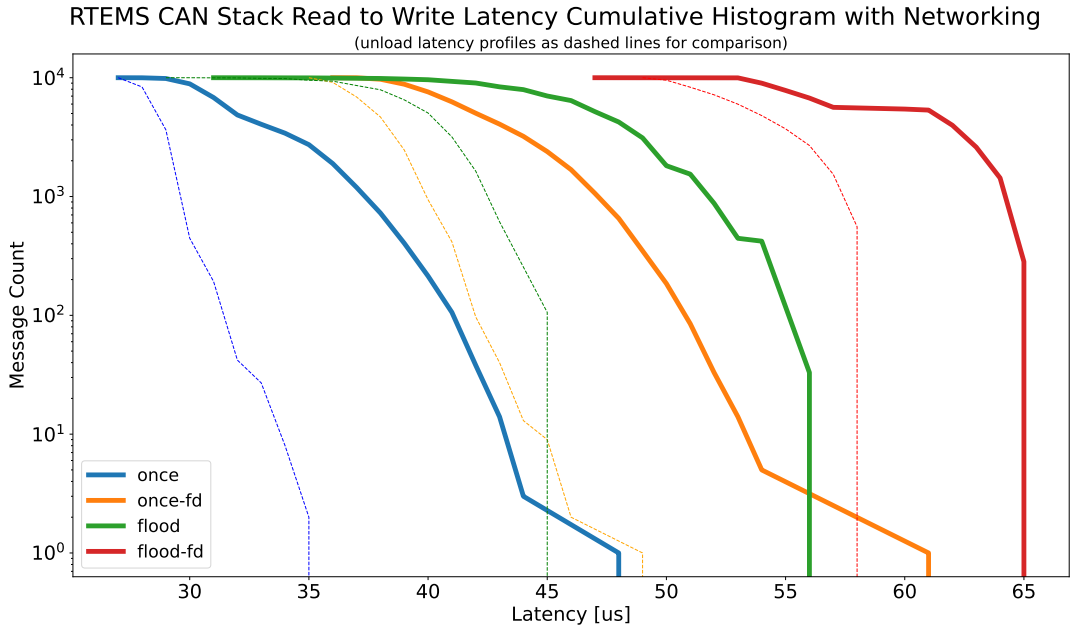

Figure 6.4. RTEMS CAN stack read to write latency profile with enabled networking.

The measurement results can be seen in Figure 6.4. The bold full lines are the data measured with stress loaded HTTP server, dashed thin ones are without external load. It can be seen latencies without load are just slightly worse than the ones with disabled networking presented in Figure 6.3. The results got worse when HTTP server is stress loaded, but still those values are not bad. Maximum latency difference is up to 20 micro seconds, which can be caused by networking critical sections with disabled interrupts, interrupt handling, context switching between cores, competing for cache memory between CAN and network tasks and so on. Overall, the measured latencies, both with and without networking, look really promising.

It should be noted that RTEMS executive has single address space. Therefore, CAN gateway used in these measurements is basically kernel gateway and not user space gateway.

6.2 Dynamic Allocation Demonstration with RTEMS and CTU CAN FD

The ability of dynamic allocation of TX buffers to priority groups, presented and described in chapter 4 was demonstrated on educational kit MicroZed APO. 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.

The goal of the test was to demonstrate the ability of high priority messages to preempt low priority messages located in controller's HW buffers. To achieve this, one controller was fully loading the CAN bus with 8 bytes long messages. These messages had identifier 0x500 that served as middle priority identifier. The second controller

was accessed from two separate applications with one attempting to send messages with 0x700 identifier (low priority message) without delay between messages (basically flood) and second attempting to send 8 byte long messages with 0x20 identifier.

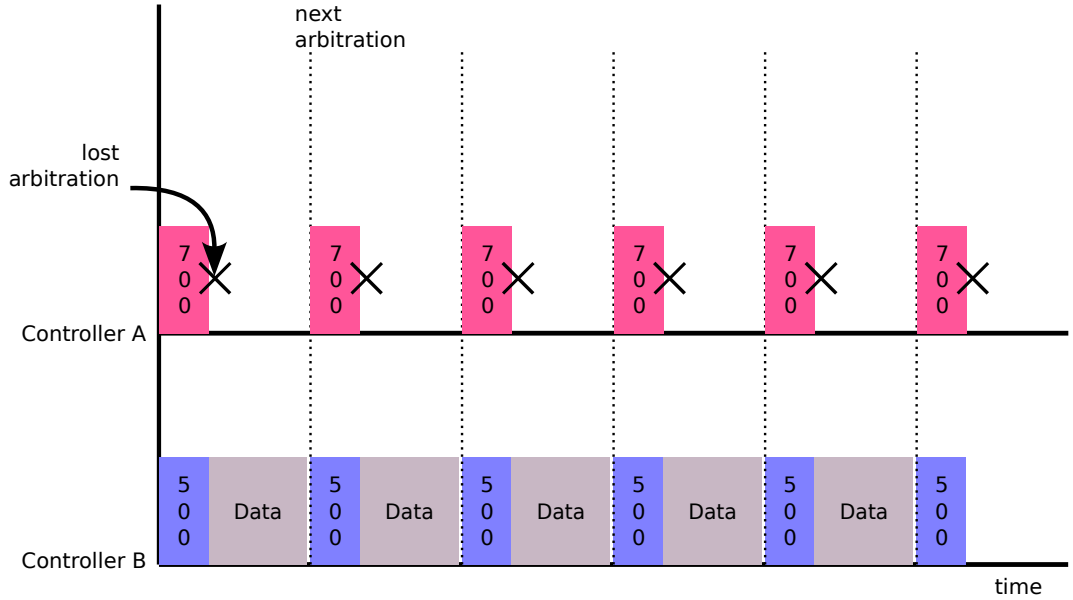

Figure 6.5. CAN bus state without dynamic allocation of TX HW buffers. Priority inversion occurs.

If only traditional driver with FIFOs would be used, the high priority messages would wait indefinitely for low priority ones to leave four TX buffers. Those would of course never leave, because the bus was fully loaded by middle priority ones and thus low priority message would never win the arbitration phase. This situation is depicted in Figure 6.5, where controller A propagates low priority to arbitration phase. These frames of course lose the arbitration to middle priority frames sent from controller B.

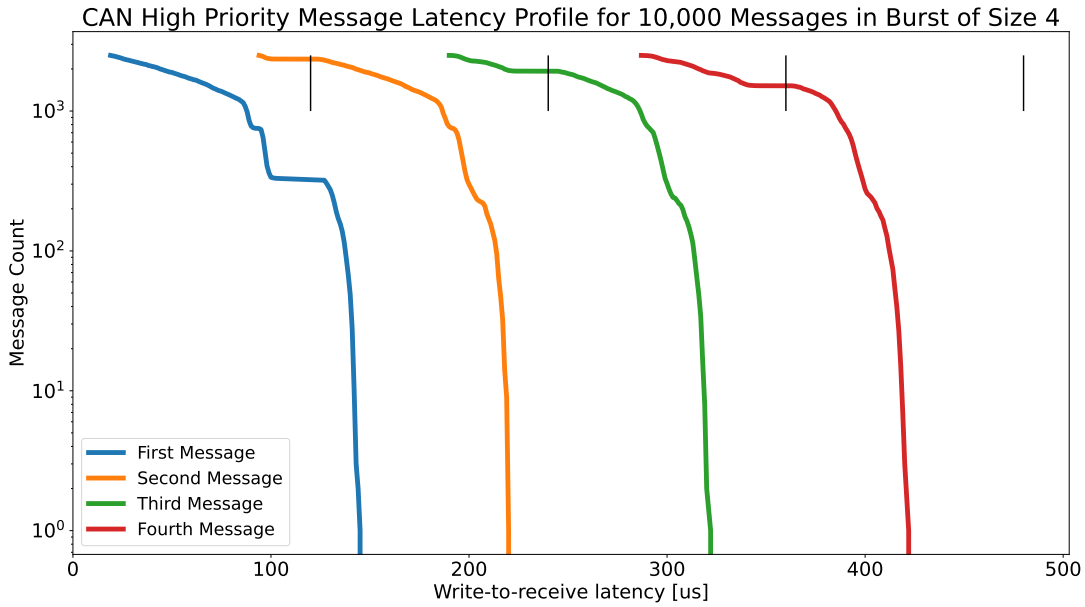

Figure 6.6. High priority message latency profile.

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. These messages were passed to the controller through the queue with higher priority and filter set to match only 0x20 identifier. The latency profile in Figure 6.6 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 and delivered in frame's timestamp field to the test application. The TX time is read from the CTU CAN FD IP core by `RTEMS_CAN_CHIP_GET_TIMESTAMP` ioctl call (refer to section 3.3.9) exactly before frame write operation.

The black vertical lines represent the length of one 8-byte long message transmission (about 130 microseconds). It can be seen the first message is transmitted to the network almost immediately in the best case. The delay is caused by framework latency (time it takes to propagate the frame through queues to the controller and save it to HW buffer, see section 6.1). The worst case is given by an interval corresponding to the blocking of transmission by an already started middle priority message transmission that occupies the bus. Subsequent three messages follow immediately after the previous ones.

From bus point of view, high priority frames are now propagated to the arbitration phase correctly and win the arbitration against frame from controller B. This can be seen in Figure 6.7.

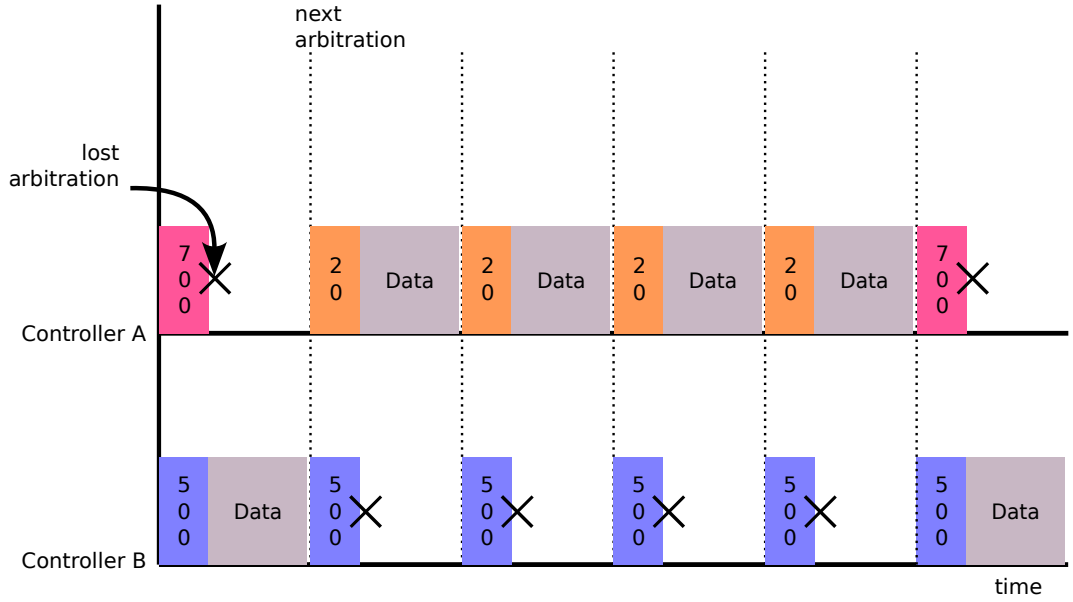

Figure 6.7. CAN bus state with dynamic allocation of TX HW buffers.

6.3 Stack Functionality Demonstration with pysimCoder

The framework was also tested against pysimCoder rapid application development tool. It is developed by Professor Roberto Bucher as an open source project with the goal to provide alternative to proprietary Matlab/Simulink for educational purposes and possibly even for simple industrial applications [11].

There is an active development effort towards pysimCoder at CTU, including support for NuttX operating system or run time monitoring and tuning of model parameters. In this demonstration, pysimCoder was ported to RTEMS and simple application demonstrating CAN frames send and receive was created.

The initial version of common CAN bus interface was also introduced for this demonstration. Application generated by pysimCoder consists of multiple blocks. Target specific blocks, motor control block for example, may use CAN bus to communicate with the hardware and there may be more than one of these blocks in the diagram. In this case, each block usually listens to different CAN frames (different identifier). Some common pysimCoder interface, capable of registering messages to be received, is useful in this case.

In the developed interface, pysimCoder's blocks registers identifier to receive during their initialization. This creates one slot for the message with given identifier in pysimCoder's common interface. Block can then read from this slot during its periodically called in/out function. CAN frame read/write is implemented in target specific code with frame reception done in a separate thread. Frame transmission is performed as blocking write.

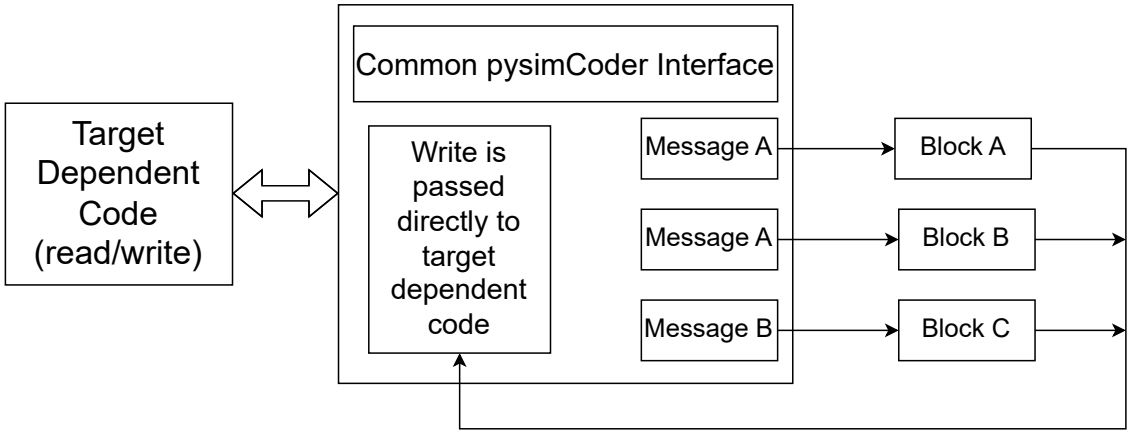

Figure 6.8. Visualization of pysimCoder common CAN interface.

Visualization of the interface can be seen in Figure 6.8. In this example, both blocks A and B want to read the frame with identifier A. Therefore, the interface creates two slots and saves the message in both of them when received. Frame transmission from blocks to network also passes through the interface for code unification, but it is passed directly to the target dependent code. Since CAN frame structure differs from system to system, pysimCoder just passes the desired identifier, data and optionally flags. Target dependent code then assigns these values to its CAN frame.

The common CAN interface for pysimCoder's blocks is still under development and not yet ready to merged into mainline. At the moment, only RTEMS is partially ported to this interface with other systems (Linux and NuttX) still using the old interface.

6.4 OrtCAN CAN/CANopen

An experiment to include new RTEMS CAN/CAN FD API option into OCERA OrtCAN CAN/CANopen infrastructure has been implemented by Pavel Píša in `ortcan-vca/libvca/vca_irtems.c`[1]. The code can be build with actual RTEMS even together with PMSM control application using RVapo RISC-V coprocessor[2] and testing is planned in the following months.

[1] `https://sourceforge.net/p/ortcan/ortcan-vca/ci/master/tree/libvca/vca_irtems.c`
[2] `https://gitlab.fel.cvut.cz/otrees/fpga/rvapo-vhdl`

6.5 Documentation

Documentation is automatically generated by a tool named Doxygen. It extracts information from comments in source files and headers and generates description of structures, functions, defines, files and so on. The descriptions with links are very useful when going through used functions and arguments for example. The link to the generated documentation can be found in project repository[3].

Another piece of created documentation is a user manual. It refers to how to use the infrastructure from user point of view and also how to port a new driver to it[4].

6.6 Test Applications

Several test applications were written during the stack implementation to test the principle and measure performance. These are all linked during build steps described in sections 2.3 and 2.4. This section introduces them and informs the readed how to use them to reproduce thesis's results.

6.6.1 can_register

This application registers CAN device/controller into `dev/canX` namespace. Simple iteration from zero is used, therefore first registered device will get `dev/can0`, second `dev/can1` and so on. Type of device to register is selected with `-t` parameter; it can be either `ctucanfd` or `virtual` target.

Example: Initialization of CTU CAN FD driver would be called as

```
can_register -t ctucanfd
```

This would register two CTU CAN FD controllers under `dev/can0` and `dev/can1` (suppose these are the first two registered controllers).

This is a process that would usually be handled from board support layer in RTEMS based on values obtained from device tree. However, since the stack is not merged to upstream at the time of writing this thesis, all tests were done from application layer against clean RTEMS build. Therefore, this application substitutes board level initialization for CAN device.

6.6.2 can_list_registered

This application can be used to list all devices previously registered with `can_register` application.

6.6.3 can_set_test_dev

It is necessary to set which device drivers shall be used for testing. Application `can_set_test_dev` is used for this purpose. It takes to device drivers as input arguments and uses them for subsequent test applications.

Example: Previously initialized CTU CAN FD controller can be set for tests as

```
can_set_test_dev dev/can0 dev/can1
```

[3] https://otrees.pages.fel.cvut.cz/rtems/rtems-canfd/doc/doxygen/html/index.html
[4] https://otrees.pages.fel.cvut.cz/rtems/rtems-canfd/doc/can/can-html/can.html

The order of parameters matters. In case of 1 way test – please refer to section 6.6.4 – first parameter is used as a sender and second one as a receiver. For 2 way test – refer to section 6.6.5 – both devices send and receive. It is necessary to specify both parameters even if they are the same devices (as it is for virtual target for example).

6.6.4 can_1way

This application performs one way test where one device acts as a sender and other one as a receiver. Used devices are specified by `can_set_test_dev` application.

```
can_1w [count] [delay] [burst]
```

It sends `[count]` number of messages with burst size `[burst]` and delay between bursts specified by `[delay]` arguments.

6.6.5 can_2way

This test is the similar to the previous one, except frame transmission is performed in both ways. Once again, parameters `[count]`, `[delay]` and `[burst]` can be specified. This applies to both devices in this case.

```
can_2w [count] [delay] [burst]
```

6.6.6 can_gateway

This application implements CAN gateway – an interface between two CAN buses, possibly with different bit timing settings. It receives messages on the first device defined with `can_set_test_dev` application and sends it to the seconds one.

This application is used for mutual latency testing described in section 6.1. Since RTEMS is a single address space operating system, the application also acts as a kernel gateway, although implemented in application space.

6.6.7 can_latency

This application performs latency testing with priority inversion occuring on the bus. One device is flooding the bus with middle priority frames while the other is trying to send mix of low and high priority messages. Please refer to chapter 4 and section 6.2 for further description.

Chapter 7
Conclusion

My work done in the scope of this thesis provides RTEMS with long needed full-featured CAN/CAN FD stack. This brings many new possibilities for CAN bus usage in RTEMS such as unified approach or POSIX interface. Many new features, including priority classes, polling functions, operations on FIFO queues, error reporting, controller configuration and control, and others, were also implemented. These can further enhance CAN bus performance in many applications and implementations even in other operating systems if ported to.

The latency measurements provided in Section 6.1 indicates the stack can perform well within commonly expected latencies even when stressed with networking stack. It shall be noted the measurement was done on a single hardware and architecture, future measurements with different controllers or MCU architectures may also provide interesting results.

Chapter 4 dealt with the common problem of priority inversion that may occur on CAN bus during the arbitration phase. The algorithm, extending the common solution of multiple priority classes, was extended by dynamic allocation of hardware transmission buffers to multiple priority groups. The proposed solution allows to utilize all hardware TX buffers while preserving the correct transmission order. This algorithm and its results were presented at the 18th international CAN Conference in 2024 [1].

There is a lot of possible future work for upcoming students once the stack gets approaved by RTEMS maintainers and is merged into project upstream – this process is already underway with an initial positive response on stack basic principles and operations. RTEMS would benefit from implementing other controllers to the presented infrastructure like SJA1000, FlexCAN, or MCAN to name a few. Another future development goal might be the stack extension to CAN XL format.

Appendix A
Source Code

This appendix lists the links to the source code and GIT repositories related to the project. The ultimate goal is to merge the infrastructure and CTU CAN FD controller to RTEMS mainline, but the implementation is still under review at the time of the thesis submission.

This also means there might be some changes to the code based on the review. The basic infrastructure API was already reviewed with a positive results, therefore changes in this part are not expected, however the user should verify the current state of RTEMS CAN stack in the official RTEMS documentation after reading this thesis.

The current implementation of both CAN/CAN FD stack and CTU CAN FD controller can be found as a public repository at CTU FEE GitLab page [1]. The stack is implemented in `lib/candrv` directory, test applications are located in `rtems_can_test` directory.

[1] `https://gitlab.fel.cvut.cz/otrees/rtems/rtems-canfd`

Appendix B
Glosary

API	Application Programming Interface
APO	Computer Architecture Course at CTU FEE
BSD	Berkeley Software Distribution
BSP	Board Support Package
CAN	Controller Area Network
CAN CC	Classical CAN
CAN FD	Controller Area Network Flexible Data-Rate
CAN XL	Controller Area Network Extended Data-Field Length
CSMA/CR	Carrier-Sense Multiple Access with Collision Resolution
CTU	Czech Technical University in Prague
DUT	Device Under Test
FEE	Faculty of Electrical Engineering
FIFO	First In, First Out
HW	Hardware
IOCTL	Input/Output Control
POSIX	Portable Operating System Interface
RTEMS	Real-Time Executive for Multiprocessor Systems
RX	Receive
SMP	Symmetric Multiprocessing
SW	Software
TX	Transmit
VHDL	VHSIC Hardware Description Language

References

[1] Lenc, Michal, and Pavel Píša. Scheduling of CAN frame transmission when multiple FIFOs with assigned priorities are used in RTOS drivers. *2024 18th international CAN Conference in 2024*. 2024, pp. 105-110.

[2] RTEMS Project. *Historical Timeline*. [cit. 2024-04-03]. Available from `https://devel.rtems.org/wiki/History/Timeline`.

[3] Bloom, Gedare, and Joel Sherrill. Scheduling and thread management with RTEMS. *SIGBED Rev.* New York, NY, USA: Association for Computing Machinery, feb, 2014, Vol. 11, No. 1, pp. 20–25. Available from DOI 10.1145/2597457.2597459. Available from `https://doi.org/10.1145/2597457.2597459`.

[4] RTEMS Project. *Task Definition*. [cit. 2024-04-19]. Available from `https://docs.rtems.org/branches/master/c-user/task/background.html`.

[5] Aliwa, Emad, Omer Rana, Charith Perera, and Peter Burnap. Cyberattacks and Countermeasures For In-Vehicle Networks. 04, 2020.

[6] Píša, Pavel. *Linux/RT-Linux CAN Driver (LinCAN)*. [cit. 2024-25-02]. Available from `https://cmp.felk.cvut.cz/~pisa/can/doc/lincandoc-0.3.pdf`.

[7] Vasilevski, Matěj. CAN Bus Latency Test Automation for Continuous Testing and Evaluation. *master's thesis*. CTU FEE, 2022. Available from `https://dspace.cvut.cz/bitstream/handle/10467/101450/F3-DP-2022-Vasilevski-Matej-vasilmat.pdf`.

[8] Ille, Ondrej, Jiří Novák, Pavel Píša, and Matěj Vasilevski. CAN FD open-source IP core. *CAN Newsletter*. 2022, Vol. 3/2022, pp. 39-41.

[9] Píša, Pavel, Jiří Novák, Pavel Hronek, and Matěj Vasilevski. Continuous CAN Bus Subsystem Latency Evaluation and Stress Testing on GNU/Linux-Based Systems. *embedded world Conference 2024*. 2024.

[10] Jeřábek, Martin. Open-source and Open-hardware CAN FD Protocol Support. *master's thesis*. CTU FEE, 2019. Available from `hhttps://dspace.cvut.cz/handle/10467/80366`.

[11] Lenc, Michal, Pavel Píša, and Roberto Bucher. pysimCoder – Open-Source Rapid Control Prototyping for GNU/Linux and NuttX. *2023 24th International Conference on Process Control (PC)*. 2023, pp. 102-107. Available from DOI 10.1109/PC58330.2023.10217596.