CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering
Department of Radioelectronics

FlexCAN Controller Emulation for QEMU

Bachelor's Thesis

Matyáš Bobek

Supervisor
Ing. Pavel Píša, Ph.D.

2025

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Bobek Matyáš** Personal ID number: **507454**

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

Department / Institute: **Department of Radioelectronics**

Study program: **Open Electronic Systems**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

FlexCAN Controller Emulation for QEMU

Bachelor’s thesis title in Czech:

Emulace kontroléru FlexCAN pro systémový emulátor QEMU

Name and workplace of bachelor’s thesis supervisor:

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

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

Date of bachelor’s thesis assignment: **03.02.2025** Deadline for bachelor thesis submission: _____

Assignment valid until: **20.09.2026**

_____
doc. Ing. Stanislav Vítek, Ph.D.
Head of department’s signature

_____
prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the bachelor’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 bachelor’s thesis, the author must state the names of consultants and include a list of references.

_____
Date of assignment receipt

_____
Student’s signature

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Bobek Matyáš** Personal ID number: **507454**

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

Department / Institute: **Department of Radioelectronics**

Study program: **Open Electronic Systems**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

FlexCAN Controller Emulation for QEMU

Bachelor’s thesis title in Czech:

Emulace kontroléru FlexCAN pro systémový emulátor QEMU

Guidelines:

CAN communication is critical for example in automotive systems and many control units based on NXP ARM devices equipped by FlexCAN peripheral are used in such applications. The serious development of automotive and industrial operating systems and products requires continuous integration platforms and testing which is not blocked by limited availability of the target hardware. The target system emulation (including CAN controllers) is a solution.
1) familiarize with CAN communication protocol, related Linux SocketCAN subsystem and QEMU system emulation
2) study materials for NXP FlexCAN3 CAN controller
3) implement functional emulation of FlexCAN3 controller in QEMU system emulator
4) document the implementation and results and submit project for review

Bibliography / sources:

[1] J. Charvát, „Model of CAN FD Communication Cotroller for QEMU Emulator,“ bachelor thesis, FEL, Czech Technical University in Prague, Prague, 2020. [Online]. Available: https://dspace.cvut.cz/handle/10467/87714
[2] QEMU project , "CAN Bus Emulation Support," https://www.qemu.org/. https://www.qemu.org/docs/master/system/devices/can.html (accessed Jan. 29, 2025).
[3] i.MX 6Dual/6Quad Applications Processor Reference Manual, NXP, Eindhoven, Netherlands, 2020. [Online]. Available: https://www.nxp.com/products/i.MX6Q

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Bobek Matyáš
Personal number: 507454
Programme name: Open Electronic Systems

declare that I have elaborated the bachelor's thesis entitled

FlexCAN Controller Emulation for QEMU

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 18.05.2025

Matyáš Bobek
.....
student's signature

Acknowledgments

I would like to thank my supervisor, Ing. Pavel Píša, Ph.D., for invaluable motivation and bringing me into the world of free software development. This work would not be possible without his guidance.

I would also like to thank my family and friends for continued support.

Abstract

Correct and performant functional emulation of SoCs (system-on-chip) used in embedded devices allows application developers and systems engineers to easily test and debug embedded software and firmware. QEMU project offers a user-level and a whole-system emulator for many targets, including the NXP iMX SoC series, often used in automotive and industrial control units. But its emulation lacks support for the CAN communication bus. This thesis aims to implement a CAN controller peripheral functional emulation for iMX6 SoC-based systems, such as SabreLite development board, model of which is included in QEMU.

Keywords: FlexCAN, QEMU, CAN, emulation, SocketCAN, Linux.

Abstrakt

Korektní a výkonná funkční emulace SoC (system-on-chip) čipů, používaných ve vestavěných zařízeních, umožňuje vývojářům aplikací a systémovým inženýrům jednoduše testovat i ladit vesavěný software a firmware. Projekt QEMU nabízí emulaci v uživatelském módu i emulaci celého systému pro mnoho cílových architektur, včetně série čipů i.MX firmy NXP, často užívaných v automotivních a průmyslových řídících jednotkách. Emulace této série však postrádá podporu pro komunikační sběrnici CAN. Tato práce cílí implmentovat funkční emulaci periferie – CAN kontroléru pro systémy založené na SoC i.MX6, jako je vývojová deska SabreLite, jejíž model je obsažený v QEMU.

Klíčová slova: FlexCAN, QEMU, CAN, emulace, SocketCAN, Linux.

Contents

1 Introduction **2**

2 Analysis **3**

2.1 Motivation . . . 3
2.2 CAN . . . 3
2.2.1 CAN 2.0 . . . 4
2.2.2 CAN FD . . . 5
2.3 FlexCAN . . . 5
2.3.1 Reset . . . 6
2.3.2 Low-Power Modes & Freeze Mode . . . 7
2.4 QEMU . . . 7
2.4.1 Architecture . . . 8
2.4.2 CAN Subsystem . . . 9

3 Implementation **10**

3.1 Overview . . . 10
3.1.1 Migration Support . . . 11
3.1.2 Interrupts . . . 11
3.2 Operation Mode Control . . . 11
3.3 Memory Mapped I/O . . . 12
3.3.1 Load emulation . . . 13
3.3.2 Store emulation . . . 14
3.4 Reset . . . 14
3.5 TX Pipeline . . . 15
3.5.1 Arbitration . . . 16
3.6 RX Pipeline . . . 17
3.6.1 RX Message Buffers . . . 18
3.6.2 RX Serial Message Buffer . . . 19
3.6.3 Rx-FIFO . . . 19

4 Use Instructions and Testing **21**

4.1 Build Guide . . . 21
4.2 Usage and Testing Guide . . . 21
4.2.1 QEMU Virtual Bus . . . 21
4.2.2 Virtual SocketCAN Interface . . . 22

4.2.3 Running SabreLite machine 22
4.2.4 FlexCAN configuration 24
4.2.5 Manual Testing 24

5 Conclusion **27**
5.1 Discussion 28
5.1.1 Future CAN FD Support 29

References **31**

Appendices **34**

A Software Versions **35**

B Used AI Tools **36**

Acronyms

Acronym	Meaning
API	Application Programming Interface
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-rate
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CTRL	Control
ECR	Error Counter Register
FIFO	First-In First-Out
ID	Identifier
IFLAG	Interrupt Flag
IMASK	Interrupt Mask
IP	Intellectual Property
IRQ	Interrupt Request
KVM	Kernel Virtual Machine
MB	Message Buffer or Mailbox
MCR	Master Control Register
MMIO	Memory-Mapped Input-Output
OS	Operating System
RAM	Random Access Memory
RST	Reset
RX	Receive
SMB	Serial Message Buffer
SoC	System on Chip
TTY	Teletypewriter
TX	Transmit
VFS	Virtual File System
VM	Virtual Machine

1 | Introduction

Short development cycles and continuous innovations require system software and applications design, integration, and testing even before target hardware is produced and brought up. Even when hardware is available, it is often much easier to realize routine continuous integration and testing on a general-purpose server (i.e. x86-64-based) than to orchestrate specialized hardware for each test.

In this text, I present a functional FlexCAN peripheral device emulator integrated into the QEMU machine emulator. QEMU and my FlexCAN emulator allow for testing embedded software using FlexCAN for communication with the same operating system, kernel, driver, and possibly the same full binary image, as would be used in production hardware. This way, the maximum number of bugs can be detected during testing and fixed, without using any special hardware, other than an off-the-shelf server or a PC. This minimizes the number of surprise faults and errors when the production hardware is ready.

This work targets the emulation of the FlexCAN core used in i.MX6 series processors for Linux software. The Boundary Devices SABRE Lite i.MX6 development board has been selected as the specimen hardware, as it is already available in QEMU, missing the FlexCAN module emulation.

Document Structure. There is an introduction to CAN, FlexCAN and QEMU in chapter 2. This text does not aim to explain or fully describe any of those technologies but offers an overview of parts that happened to be useful and/or important for this thesis. chapter 3 contains details on how was the device emulator implemented and integrated. chapter 4 shows how to run, test and use the emulator. An assesment of the achieved and unachieved goals may be found in chapter 5 “Conclusion”, together with thoughts and plans for further work.

2 | Analysis

2.1 Motivation

This section will present an example use case of QEMU and the FlexCAN emulator.

An engineer is developing an i.MX6-based in-car entertainment application software, which uses CAN bus as the interconnect between all digital modules in the car. The specification of the hardware module the software will run on is decided, but the modules are not yet manufactured, or their design is still in its development phase. When the engineer wants to test and/or debug the application, together with a car simulation program, they may use their x86-64-based Linux workstation. See Figure 2.1 for a diagram of this architecture. A car simulation program is available. It will simulate the communication — CAN messages a real car would output. The simulation program can run on the host Linux system. Its CAN communication will attach to a virtual CAN interface(s) using the SocketCAN userspace API. The entertainment system application (i.e. Qt-based) and the OS it will run on, will run in the QEMU ARM system emulator. The QEMU CAN bus will be connected to the host virtual CAN interface over SocketCAN and to the emulated FlexCAN device. For testing, the emulator will run from an emulated flash memory, which will be emulated as I/O to an ordinary disk image file in the host VFS. The file can contain the identical binary image as would be flashed to the target hardware module. No explicit support for the emulator is required from the application and system software development point of view.

2.2 CAN

CAN (Controller Area Network) is a network protocol designed for low-range communication between embedded devices. Originally designed by the German company Bosch intended for use in the automotive industry, where it is commonly used to this day.[27] Today, CAN is also used in "other types of vehicles, from trains to ships, as well as in industrial controls,"[27] and "a variety of industries including building automation, medical, and manufacturing."[10, p. 2]

In the following text, the terms *message*, *identifier* and *data/remote/error frame* will be used as defined in [6]. The standard [6] is based on the original specification [7]. This chapter will present a short introduction to the relevant features of the CAN medium access control layer, corresponding to OSI[5] data-link layer. The physical layer is not of concern for this work.

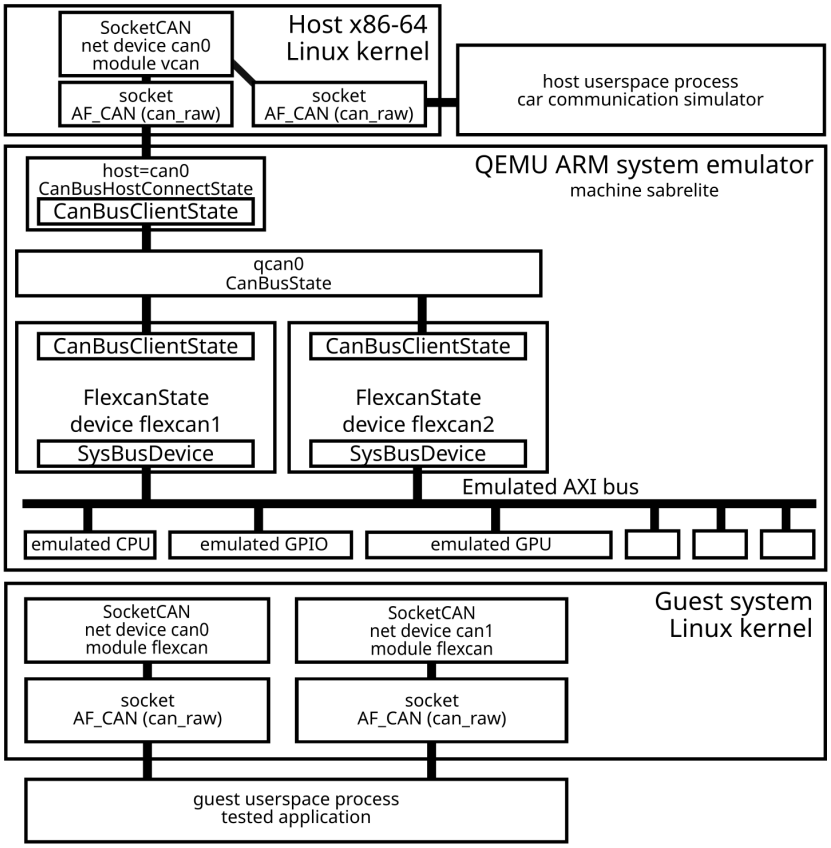

Figure 2.1: Diagram of the described test architecture.

2.2.1 CAN 2.0

Bus Arbitration. "The CAN protocol is a carrier-sense, multiple-access protocol with collision detection."[10] CAN uses arbitration on message priority. The identifier is not only used by higher-level protocols for identifying the message but also as the priority. When multiple nodes begin transmitting, the node transmitting the message, with an identifier of the lowest numeric value,[1] wins the arbitration. The winning node gets to send its frame, while the other nodes must back off and wait for the next arbitration. This process will be called *bus arbitration* in the rest of this text.

Extended Identifiers. The identifier may be either the "standard" 11-bit (CAN 2.0A) or "extended" 29-bit (CAN 2.0B). An extended identifier frame is identified by a recessive `IDE` bit.

[1]That is because the "0" bit is dominant and will override any other node transmitting the recessive "1" on the bus.

2.2.2 CAN FD

CAN FD (Controller Area Network – Flexible Data-rate) is a backward-compatible evolution of CAN 2.0. It offers three main advantages:

- Increased network bandwidth.
- Larger message payloads (up to 64 B compared to the maximum of 8 B for CAN 2.0).
- Improved checksum coverage.[9]

The increased bandwidth is achieved by using faster `DATA` bitrate to part of control bits and data and CRC fields. For the arbitration phase of frame transmission, the bitrate is the same as if it were a CAN 2.0 transmission. If the `BRS` control bit is transmitted recessive, the rest of the frame (until the CRC delimiter and acknowledgment bit) is transmitted at the faster `DATA` rate. "During the arbitration phase, any nodes on the network can be concurrently transmitting. If the CAN-bus layout is good enough to handle a certain nominal bitrate during arbitration, it is possible to use [at least] twice that bitrate for the data bitrate, because the physical signaling is less demanding when there is only one node transmitting."[9]

2.3 FlexCAN

FlexCAN is a CAN controller IP core made by company IPextreme, now owned by Silvaco, in cooperation with NXP Semiconductors.[25][28] This chapter will shortly introduce FlexCAN and some of its interesting features.

Versions. Two versions of FlexCAN are of interest for this work: FlexCAN2 and FlexCAN3. See Table 2.1 for comparison. The main difference is support for CAN FD. In the following text, the term "FlexCAN" refers to version FlexCAN2, unless noted otherwise. See subsection 5.1.1 for details on FlexCAN3 support.

Version	CAN FD support	Example SoC family
FlexCAN2	✗	NXP i.MX53, NXP i.MX6, NXP LS1
FlexCAN3	✓	NXP i.MX8, NXP LX2

Table 2.1: Considered FlexCAN versions.[12, line 4-14]

Structure. FlexCAN is the CAN controller used in NXP i.MX SoCs, although only some of the SoCs in the series have the CAN support feature. The IP core is composed of the following components, with some of their functionality pointed out:

- Bus Interface Unit — Interfaces FlexCAN to SoC buses, clocks, interrupt lines etc.
- Message Buffer RAM block
- Controller Host Interface — Handles message buffer selection — arbitration and filtering.

- Protocol Engine — Encodes/decodes frames between internal representation and the CAN physical wire protocol.[24][2]

In a SoC, the core is connected to a bus, over which the CPU addresses FlexCAN registers. This means the driver running on the CPU controls the core using ordinary loads and stores to the FlexCAN-assigned memory region. For example, in i.MX6 series SoCs, the core is connected to the AIPB (Arm IP Bus), which is then connected to the APB (Advanced Peripheral Bus) via a bridge peripheral. The APB is itself connected to the AHB (Advanced High-performance Bus), which is the main memory transaction switching fabric for the CPUs.

The FlexCAN2 memory-mapped region is 16 kiB long, of which the first 2 532 bytes are used.[1] The region contains configuration registers and the 64 mailbox array. Note that the terms "mailbox" and "message buffer" are used interchangeably in this text.

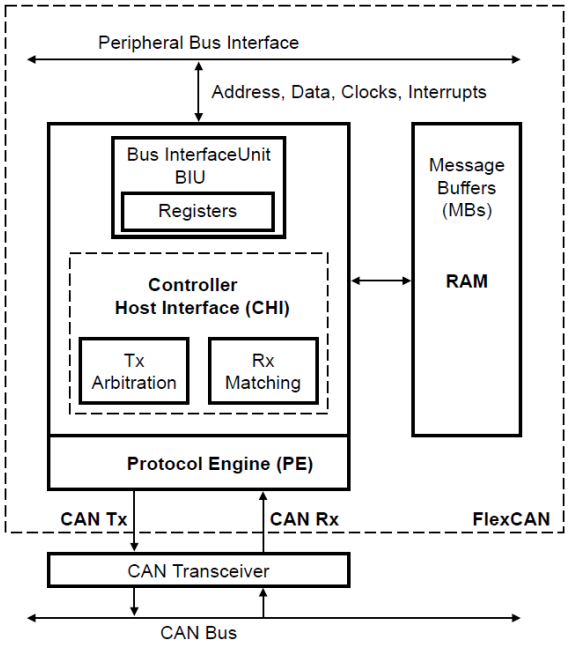

Figure 2.2: Schema of the FlexCAN IP core structure.[2]

2.3.1 Reset

Hardware reset is functionality present in almost any digital device since the state of all semiconductor-based latches, flip-flops and volatile memories is usually undefined after power-up. In a SoC peripheral, reset should clear visible and hidden/inner state and set it to a well-defined configuration. For FlexCAN, that configuration is quite simple – all registers except for the Master Control Register (MCR) are zeroed. That also includes interrupt flags and therefore the interrupt request line should be set to its inactive state. Functionally, the modules power on in freeze mode with FIFO disabled (for backward compatibility).

Additionally, the module can be soft-reset by setting the `MCR[SOFT_RST]` bit. This also resets

the module to freeze mode with FIFO disabled, but crucially does not clear the mailboxes, the bit-timing configuration in `CTRL1` and `CTRL2` registers and all queue and message buffer filtering registers.

2.3.2 Low-Power Modes & Freeze Mode

Freeze Mode. When FlexCAN is reset, it is configured into so-called "freeze mode". In this mode, the module is unable to receive nor transmit frames. Many configuration registers can only be written in freeze mode, and drivers are expected to configure FlexCAN in freeze mode.

Low-Power Modes. For power saving, FlexCAN also offers "low-power" modes: module disable mode and stop mode. Of course, when CAN capability is not required at all, the whole module can have its clock centrally disabled, using SoC clock and power management facilities, but that is out of the scope for this work. Stop mode is requested globally on the SoC level and is not considered in this work.

Module Disable Mode. In module disable mode, both the Protocol Engine and Controller Host Interface (as in Figure 2.2) have their clocks disabled, while the bus interface unit is left powered on, so many registers can be read or written to. Although the bus interface unit is powered up, some of the registers and the entire message buffer area may or "may not be accessed when the module is in Disable Mode depending on how FlexCAN RAM is integrated into the Arm."[1, p. 1158] Module disable mode can be entered from freeze mode and has "priority" over it. The module returns back to freeze mode on wake up. FlexCAN has a "Self Wake" mechanism: waking up the module and the CPU after detecting a recessive to dominant transition on the CAN bus. The frame that woke the module up must not be received.[1]

In further text, "normal mode" means powered on FlexCAN in neither freeze mode nor any of the low-power modes, unless specified otherwise. This is equivalent to `MCR[NOT_RDY]` read-only flag being cleared. Note the i.MX6 reference manual[1] uses a different definition of normal mode.

2.4 QEMU

QEMU is an open-source computer emulator. QEMU supports two emulation modes:

1. System-mode — Emulation of an entire machine, which will run an entire OS.
2. User-mode — Emulates CPU and OS interface. Allows one to run binaries compiled for different processor architectures on native OS as ordinary processes. Only Linux and BSD are supported.[16]

This work uses system mode emulation exclusively. In system-mode, QEMU can also be used for hardware-accelerated virtualization; concerning the target platform of this work (32-bit ARM),

there is KVM[2] support for ARM since ARMv7, but QEMU supports KVM only on 64-bit ARM. There might be the option of running a 32-bit ARM guest on a 64-bit ARM under KVM, but that is out of the scope of this work.[19][17][29]

2.4.1 Architecture

QEMU supports lots of different combinations of host OS and architectures, guest OS and architectures, emulated devices, host device drivers etc. QEMU codebase is large and is divided into logical parts called subsystems. The device subsystem and CAN subsystem are of most importance to this work.[15][14]

QEMU uses a GLib-like object model called QOM (QEMU Object Model). The model (a layer built on top of it called QEMU device API) is used for encapsulating emulated devices.

VM Migration

QEMU allows its user to pause VM execution, serialize its whole state (RAM, CPU architectural state and the current state of all peripheral devices), and then launch a different instance of QEMU, potentially on a different host computer or even a newer version of QEMU, which will then deserialize the saved state and run the machine. This feature is called machine migration. The state is serialized to a stream, which can be saved, but also can be streamed directly into a new instance of QEMU. This allows for little downtime when upgrading QEMU to a newer version, among other things.[18]

The guest OS does not have to have any explicit support for migration. This thesis adds a device to QEMU, and the new device has to support it.

IRQ handling

In QEMU, IRQ lines are represented by the type `qemu_irq`, which is a pointer to an internal interrupt line representation. A device allocates interrupt lines for all its interrupt outputs. It then stores the `qemu_irq` references in its own state structure. The upper layer then usually connects the virtual interrupt line to a virtual interrupt controller.

For example in FlexCAN implementation, the i.MX6 machine base class[3] connects the interrupt line to a virtual Generic Interrupt Controller (the "standard interrupt controller for Arm Cortex-A and Arm Cortex-R profile processors"[30]) using function `sysbus_connect_irq`.

The interrupt line is then controlled using the function:

```
void qemu_set_irq(qemu_irq irq, int level);
```

A `level` of 1 means there is an interrupt request pending.

[2] Kernel Virtual Machine; Linux hardware-accelerated virtualization framework.

[3] `struct FslIMX6State` defined in `include/hw/arm/fsl-imx6.h`

2.4.2 CAN Subsystem

QEMU provides a modular subsystem for CAN networking support. It consists of *clients* possibly connected to *buses*. Every client is connected to up to one bus. The bus is a collection of references to connected clients. When a client sends a frame (using the function `can_bus_client_send`), the frame is immediately passed to every other client connected to the same bus. The subsystem uses structure `qemu_can_frame` for CAN frame representation. The structure is identical (and binary compatible) to the one used by the Linux SocketCAN subsystem. It contains the following fields:[14]

- Field `qemu_canid_t can_id`. 32-bit integer storing the CAN frame destination ID along with flags for extended ID, remote request frames or error frame.
- Field `uint8_t can_dlc`. Data length code, storing the number of data bytes in frame. Ranges from 0 to 8 (inclusive) on CAN 2.0B.
- Field `uint8_t flags`. Not used by FlexCAN. Used to identify CAN FD frames and store related flags.
- Field `uint8_t data[64]`. Stores frame data. Only the first `can_dlc` bytes are valid. The array size is 64 bytes to support CAN FD (see subsection 2.2.2).

When a CAN frame is transmitted onto the virtual bus, it is passed onto all connected bus clients, using two functions called `can_receive` and `receive`. Those functions are passed to the CAN engine using struct `CanBusClientInfo`:

```
struct CanBusClientInfo {
    bool (*can_receive)(CanBusClientState *);
    ssize_t (*receive)(CanBusClientState *,
        const struct qemu_can_frame *frames, size_t frames_cnt);
};
```

When a frame is sent to the bus, the engine calls `can_receive` first, which is meant to test whether the client is ready to receive frames. If it returns `true`, it will then call `receive` with a vector of frames to be received, meaning that `frames` is `frames_cnt`-element array.[14]

3 | Implementation

3.1 Overview

The state of the FlexCAN emulator is stored in structure `FlexcanState`. It has the following fields:

- Field `SysBusDevice parent_obj`. Since FlexCAN is an MMIO peripheral, it inherits from `SysBusDevice`, from QEMU device API.
- Field `MemoryRegion iomem`. A descriptor of the MMIO memory region assigned to FlexCAN.
- Field `qemu_irq irq`. FlexCAN interrupt line descriptor.
- Field `CanBusState* canbus`. Reference to the virtual CAN bus FlexCAN is connected to. `NULL` if not connected to any bus.
- Field `CanBusClientState bus_client`. A descriptor of the connection of FlexCAN and the virtual CAN bus. Only valid if `canbus` is not `NULL`.
- An anonymous union of `FlexcanRegs regs` and `uint32_t regs_raw[]`. Stores FlexCAN registers in the same offsets as they are located in the MMIO region.
- Field `int32_t locked_mbidx`. Index of the locked message buffer. If no message buffer is locked, the value is `FLEXCAN_NO_MB_LOCKED`.
- Field `int32_t smb_target_mbidx`. If not `FLEXCAN_SMB_EMPTY`, indicates that the SMB (Serial Message Buffer) is filled and stores the index of the target mailbox. If the SMB is non-empty, this field should be equal to `locked_mbidx`.

Aliasing. FlexCAN registers are stored as array of `uint32_t`, but it is often useful to view them as structures (aggregates) of named fields. Structures `FlexcanRegs`, `FlexcanRegsMessageBuffer` and `FlexcanRegsRXFifo` are used for this very purpose. When using pointers of different types which address the same (overlapping) region of memory, the optimizing C compiler might presume the pointed-to memory regions do not overlap and generate an invalid machine code. This is called aliasing and the C standard specifies under which conditions exactly might pointers alias. In this context, the types are an array of `uint32_t` and an aggregate of `uint32_t`.

An aggregate containing a value of type A may alias with an array of the same type $A[]$, as written in the C standard:

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)[4, sec. 6.5]

Therefore the program using the said structures will be correctly compiled.

3.1.1 Migration Support

As the whole device state is saved within the `uint32_t` register array and a few more 32-bit integers in structure `FlexcanState`, the whole device can be migrated by saving said integers. QEMU allows for the implementation of the (de)serialization by filling a `VMStateDescription` structure with a description of the state object fields.

The CAN subsystem link objects (`canbus` and `bus_client`) do not have to be saved, as they will be created when the new instance launches in quite the same manner as they were created originally.

3.1.2 Interrupts

FlexCAN has 70 interrupt sources, listed here with their flag and mask registers: 32 in `IFLAG1` masked by `IMASK1`, 32 in `IFLAG2` masked by `IMASK2` (one for every one of the 64 mailboxes) and 6 additional in `ECR` with masks in `MCR`. The 6 `ECR` interrupts cannot be raised by the implementation and are always cleared (inactive). The `IFLAG`*n* registers are updated by TX and RX implementations. The function `flexcan_irq_update` updates the IRQ output line based on all interrupt sources. It bitwise ANDs the flags with their respective masks and then ORs all the resulting bits together. The result is passed as a 1 or a 0 to `qemu_set_irq` (see subsubsection 2.4.1), which sets the virtual IRQ line accordingly. The IRQ update function is called:

- At the end of all store memory transactions.
- At the end of the frame receive callback (`flexcan_receive`).
- After a load memory transaction, which caused a mailbox to be unlocked (paragraph 3.3.1).
- During reset.

3.2 Operation Mode Control

Transitions between normal mode, freeze mode and module disable mode (see subsection 2.3.2) can only happen during a store to `MCR` register. Those stores are processed by the function `flexcan_set_mcr`. Two approaches to mode transitions were considered:

1. Detecting transitions by comparing the previous and newly set `MCR` value.

2. Stateless decisions based only on the newly set value.

The transition-based approach is certainly more powerful and allows for taking action exactly in the moment of the transition. This is required for example when enabling and disabling Rx-FIFO, because the FIFO mailbox area and related interrupt flags need to be cleared for proper emulated FIFO function. Meanwhile, in the current implementation state, there is no such action required for transitions between module disable mode, freeze mode and normal mode.

In the implementation, module disable mode transitions are detected, but not used. Freeze mode transitions are not detected and use approach 2. These transitions only need to set their respective acknowledgment flags, which are used not only by the guest driver but internally by the device emulator too. The `MCR[NOT_RDY]` flag is of most use in the emulator. When set, the module can neither transmit nor receive frames.

3.3 Memory Mapped I/O

FlexCAN is controlled by reading/writing to registers mapped to CPU memory space. QEMU represents those kind of devices via `TYPE_SYS_BUS_DEVICE` base QOM class. FlexCAN emulator does not need to know the offset of the region in CPU address space; it operates upon relative offsets only. The base offset is assigned by the upper layer in `fsl-imx6.c`. Supported operations are specified using structure `MemoryRegionOps` like so:

```
static const struct MemoryRegionOps flexcan_ops = {
    .read = flexcan_mem_read,
    .write = flexcan_mem_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
    .valid = {
        .min_access_size = 1,
        .max_access_size = 4,
        .unaligned = false,
        .accepts = flexcan_mem_accepts
    },
    .impl = {
        .min_access_size = 4,
        .max_access_size = 4,
        .unaligned = false
    },
};
```

The `valid` field specifies which size and alignment of memory accesses are valid for the CPU to make. The `impl` field specifies which size and alignment of memory accesses are supported by the device implementation. Only 32-bit aligned 32-bit loads and stores are supported by the FlexCAN implementation. QEMU memory subsystem automatically converts any smaller

operation into a load/store pair and a misaligned access into two load/store pairs.

Endianness. The endianness of `DEVICE_NATIVE_ENDIAN` means that memory operations are converted from guest endianness to host endianness. For example, if QEMU emulating a 32-bit ARM machine with FlexCAN was running on a big-endian SPARC V8 host processor, QEMU would byte-swap the result of a 32-bit word load (`flexcan_mem_read` return value) made from the emulated guest CPU to the FlexCAN region, converting it from big-endian to little-endian.

User-mode access. When the `MCR[SUPV]` bit is cleared, some registers can be accessed even when the CPU is in the user processor mode.[1] Higher privilege levels have full access to FlexCAN. Source [1] is not very clear on what registers can be accessed in user mode, as it only mentions `MCR` are inaccessible. Bit `CTRL2[WRMFRZ]` can also "enable unrestricted write access to FlexCAN memory."[1] It is, however, also unknown to which registers it applies, and therefore it is left ignored.

Before any memory operation in the FlexCAN region, the function `flexcan_mem_accepts` is called. It checks the `MCR[SUPV]` bit and rejects user mode operations if the bit is set or the access target is `MCR`. It also checks that the relative address is in the range of the FlexCAN2 registers. If any check fails, the invalid access is rejected and will cause a memory violation exception in the CPU.

3.3.1 Load emulation

Emulation of loads from the FlexCAN memory region is implemented in function `flexcan_mem_read`. The function essentially loads the register at the given offset in the register array and returns it.

Message Buffer Lock Mechanism. The only additional `flexcan_mem_read` behavior is handling locking and unlocking message buffers.

At most one message buffer can be locked at a time. Locking and unlocking are implemented in functions:

```
static void flexcan_mb_lock(FlexcanState *s, int mbidx);
static void flexcan_mb_unlock(FlexcanState *s);
```

Unlocking happens in the following conditions:

1. Loading control word of a different (unlocked) mailbox. This simultaneously locks the other mailbox.
2. Loading `TIMER` register.
3. Store to the control word of the locked mailbox.

[1] PL0, ARMv7 terminology, corresponding to EL0 in ARMv8 or ring 3 in x86.

Conditions 1 and 2 are implemented in `flexcan_mem_read`. Condition 3 is implemented in the transmission program (section 3.5). [1, sec. 26.6.7.3]

3.3.2 Store emulation

Store emulation is implemented in function `flexcan_mem_write`. Some registers are simply updated in the register array in `FlexcanState`, while others need special attention.

Write Masks. Not all registers can be written to. In FlexCAN, there are whole read-only registers but also registers with mixed read-only and writable bit fields. To correctly support the read-only fields, the implementation uses a write mask — a constant compile-time defined instance of `FlexcanRegs`, where the registers are filled with set bits for writable fields and cleared bits for read-only fields. During a store emulation, the mask is applied to the new value in the following way: A 32-bit naturally aligned store of value `VALUE` to register `REG` with `MASK` (all unsigned 32-bit integers) will be applied as:

$$\texttt{REG} := \texttt{VALUE} \cdot \texttt{MASK} + \texttt{REG} \cdot \overline{\texttt{MASK}}$$

where the operations are bitwise AND ($\cdot$), OR ($+$) and one's complement ($\overline{x}$).

Freeze Mode. Some fields and registers "can only be written in Freeze mode as it is blocked by hardware in other modes."[1] This is the case for the following registers: `CTRL2`, `ECR`, `RXMGMASK`, `RX14MASK`, `RX15MASK`, `RXFGMASK` and `RXIMR`*n* for every *n*. The same applies to certain fields in `MCR` and `CTRL1` registers. The emulator blocks writes to those fields and registers outside of freeze mode and they act as a no-op.

Other Singular Cases. Writes into `MCR` are handled by function `flexcan_set_mcr` as described in section 3.2. `IFLAG1` and `IFLAG2` registers contain interrupt flags, which are to be reset (cleared) when one is written to the specific flag bit. Also, in Rx-FIFO mode, resetting the Rx-FIFO `AVAILABLE` interrupt flag de-queues element from the queue; see paragraph 3.6.3 for details. Writes into mailbox control word trigger TX code, see section 3.5.

3.4 Reset

FlexCAN device emulator implements resetting using the new `ResettableClass` multi-phase reset interface.[21] It uses two phases: `enter` and `hold`.

In `enter` phase, the state of the module is itself reset. Since in `enter` phase, the reset function cannot mutate any external state, the IRQ line cannot be cleared. That is the only reason the `hold` phase is also required. The `exit` phase would be used if the device interacted with external QOM objects in some complex way. That is not the case for FlexCAN, therefore it is not used.

The `enter` phase is implemented by first clearing the whole register memory. This is to mimic real hardware behavior, prevent the use of uninitialized memory and potential insecure leakage of sensitive information to the guest machine. After that, it calls the function shared with the soft-reset code pathway: `flexcan_reset_local_state`. This function sets the module state to the after-reset configuration. Because of the sharing, it once again clears registers that soft-reset would clear. That is acceptable because it allows for simpler program and greater code reuse and the reset code runs once in the lifecycle of a QEMU execution for the expected use cases.

3.5 TX Pipeline

The reference manual requires the user to first inactive the mailbox, store message ID and data, and only then enable transmission by writing transmission request message buffer code into the control word of the mailbox. As the control word is written to last, the transmission code only needs to execute on a write to the control word of any mailbox.

Transmission logic is fully contained in function

```
static void flexcan_mb_write(FlexcanState *s, int mbid);
```

which is passed the mailbox index as a parameter. The index is immediately checked not to be in the range of Rx-FIFO mailboxes or filters. If a `DATA` message buffer code is detected, the function continues on sending the message. Next, the function is to move the mailbox payload to structure `qemu_can_frame`, which is used to transfer frames in the QEMU CAN subsystem.

Endianness. Payload data are stored in native endianness 32-bit words in FlexCAN mailbox, but `qemu_can_frame` structure holds them as an array of bytes. That means that a simple word-wise copy would result in an incorrect byte order if running on a little-endian host machine. Therefore the copying is done using a byte-swapping function from QEMU bit-wise operations library:

```
stl_be_p(&frame_data[i], mb->data[i])
```

This function stores the word passed in the second argument (in native endianness) to the address pointed at by the first argument in big-endian byte order. That results in correct ordering in the `frame.data` array, as can be seen in Figure 3.1. Note that `frame_data` is just `frame.data` byte array cast as `uint32_t *`. The cast is always valid as the `frame.data` array is defined to have an alignment of 8 B – meaning that reading it in 32-bit words will be valid even on platforms requiring aligned loads.

Transmission and Self-Reception Modes. When the QEMU frame structure is filled-in, the frame can be sent into the QEMU virtual CAN bus and/or self-received depending on module configuration, see Table 3.1. Self-reception is implemented by calling the receive function with the currently created `qemu_can_frame` structure for it can share the same program path as ordinary virtual bus reception.

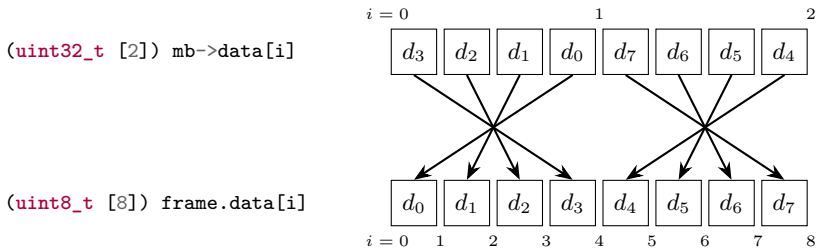

Figure 3.1: Copying payload data from message buffer to QEMU frame structure on little-endian host machine

Mode	Bus Transmission	Self-Reception
Normal Mode[2]	✓	✗
`MCR[SRX_DIS]` Cleared	✓	✓
Loopback Mode	✗	✓

Table 3.1: Transmit behavior under different configurations

Setting both `MCR[SRX_DIS]` and `CTRL1[LPB]` (thus enabling loopback mode with no self-reception) will cause no CAN frames to be self-received nor transmitted to the bus. Zephyr FlexCAN driver explicitly clears `MCR[SRX_DIS]` for loopback mode and the Linux driver contains the following comment: [13, line 423-425] [11, lines 1503-1514]

```
/** [redacted]
 *
 * NOTE: In loopback mode, the CAN_MCR[SRXDIS] cannot be
 *       asserted because this will impede the self reception
 *       of a transmitted message. This is not documented in
 *       earlier versions of flexcan block guide.
 *
 * [redacted]
 */
```

3.5.1 Arbitration

In a real-world scenario, only a finite amount of bytes can be transmitted per a unit of time. Also, if the bus arbitration[3] fails, it must retry transmission. This means there might be more than one TX buffer active, so FlexCAN needs a mechanism to select which message to transmit first. This is achieved by using a priority-based arbitration scheme, where the message with the highest priority is transmitted first. However, since the QEMU CAN subsystem only supports

[3] See paragraph 2.2.1

immediate transmission with no bus arbitration and no line delays, the emulation does not need to implement the arbitration mechanism. This greatly simplifies the transmission emulation code, as can be seen above.[1, p. 1135]

Note that as transmission can happen only after a write to the message buffer control word, the implementation cannot handle a mailbox becoming enabled after a transmit frame was written to it when it was disabled. This happens in the following cases:

1. Module exits from freeze mode or low-power mode to normal operation and any message buffer contains an active TX frame.
2. `MCR[MAXMB]` is increased and any of the formerly disabled message buffers contains an active TX frame.
3. An active TX message buffer is written in listen-only mode and the module exits the listen-only mode into a transmission-enabled mode.

For all but the first case, the module must also transition through freeze mode, as the related configuration registers cannot be written to outside freeze mode. Those cases are nevertheless mentioned for clarity. This is accepted as an unimplemented edge case by the author, because no target driver will do this.

In the short term, this can be addressed by iterating over all message buffers and processing the active TX ones, in order, or at least printing `LOG_UNIMP` warnings to the user. Implementing the complete arbitration scheme for handling this problem would not only solve it, but it would prepare the driver for the implementation of bus arbitration and line delays in the QEMU CAN subsystem. It would also allow for emulating the `ESR2` register, which allows the CPU to access some of the TX arbitration results. This register is not used by any target driver.

3.6 RX Pipeline

FlexCAN supports two structures for receiving messages: Message Buffers and Rx-FIFO. Both allow for advanced filtering and prioritization of incoming frames. This causes the receiving program pathway to be much more complex than for transmission.

Overall Datapath. Each to-be-received frame given to `flexcan_receive` is, after checking for unsupported features[4], dispatched to one of the reception mechanisms:

1. Message buffer based, implemented in function `flexcan_mb_rx`, or
2. FIFO based, implemented in function `flexcan_fifo_rx`.

If FIFO is disabled, the frame is routed directly to `flexcan_mb_rx`. Otherwise, the implementation needs to try both mechanisms, whose order is defined by configuration bits `CTRL2[MRP]` and

[4]Error frames and CAN FD frames are not supported. Both are dropped and ignored.

MCR[IRMQ]. [1, tbl. 26-15]. See Table 3.2 for the implemented ordering. The return codes of

CTRL2[MRP]	1st attempt	2nd attempt
0	FIFO	Message buffer
1	Message buffer	FIFO

Table 3.2: implemented dispatch order for enabled FIFO.

both receive mechanisms are shown in Table 3.3.

Value	Meaning
FLEXCAN_RX_SEARCH_ACCEPT	Frame stored successfully.
FLEXCAN_RX_SEARCH_RETRY	Mechanism rejected the frame and flexcan_receive should try the other one.
FLEXCAN_RX_SEARCH_DROPPED	Mechanism rejected the frame and the frame should be dropped.

Table 3.3: Return codes of flexcan_mb_rx and flexcan_fifo_rx.

3.6.1 RX Message Buffers

When trying to receive into a reception message buffer, the best candidate buffer must be selected. Program flow enters function flexcan_mb_rx, which implements the search over available message buffers. First, it clears the SMB, as the currently received frame would override it in hardware. Then it iterates serially over enabled message buffers; utility functions are used for getting the start and end pointers. For each message buffer the flexcan_mb_rx_check_mb function is called, which will determine whether said message buffer:

1. matches the incoming frame and is free-to-receive returning FLEXCAN_CHECK_MB_MATCH, or
2. matches the incoming frame but is not free-to-receive due to being locked, returning FLEXCAN_CHECK_MB_MATCH_LOCKED, or
3. matches the incoming frame but is not free-to-receive for other reason, returning FLEXCAN_CHECK_MB_MATCH_NON_FREE, or
4. does not match the incoming frame, returning FLEXCAN_CHECK_MB_NIL.

The matching logic is left as a stub, all RX message buffers therefore accept all frames. The order of selection of the matching structure is somewhat flexible, it changes depending on whether FIFO is enabled and some other registers. Ordering for the case of disabled FIFO is fully implemented, shown below in descending order of priority:

1. First free message buffer.
2. Last non-free message buffer. This applies only when bit MCR[IRMQ] is set.
3. If not matched, the frame is dropped.

Note that "first" and "last" above mean the lowest/highest buffer index respectively.

In case 2, a message buffer might not be free-to-receive because it is locked by the Message Buffer Lock Mechanism (paragraph 3.3.1). Handling of this case is explained in subsection 3.6.2.

3.6.2 RX Serial Message Buffer

If a locked mailbox matches a frame (see subsection 3.6.1), the implementation does not drop the frame, but moves it into the emulated counterpart of the SMB (Serial Message Buffer). The implementation of SMB uses an unused member of structure `FlexcanRegs`, `rx_smb0`. An additional field `locked_mbidx` in `struct FlexcanState` is used for this logic.

The frame is moved-in into the SMB as if it were a normal message buffer. The control word of the SMB structure is copied from the target message buffer beforehand so that the `flexcan_mb_move_in` move-in procedure sets the correct status code (eg. `OVERRUN`). Field `locked_mbidx` in the device state is set to the index of the target mailbox.

When a mailbox unlock condition arises, `flexcan_mb_unlock` function checks if the currently locked mailbox is the one to which `locked_mbidx` points and therefore the frame should be moved-in. In that case, the SMB contents are copied to the target mailbox, the SMB is cleared and the interrupt flag of the target mailbox is set.

If any other frame is received when a frame is waiting in the SMB, the SMB is cleared and the waiting frame is lost.

3.6.3 Rx-FIFO

When FIFO is enabled, by setting the `MCR[RFEN]` bit, the first 6 mailboxes and their respective interrupt flags are repurposed into queue slots, where the mailbox with index 0 is the front of the queue. The entire state of the FIFO is contained in the memory space of the 6 mailboxes and the interrupt flags. In the following text, each of the 6 mailboxes will be called a queue *slot*. In addition, a number of the mailboxes following the queue slots are used for holding filtering rules for FIFO reception. Their number is set using `CTRL2[RFFN]` field, ranging from 2 to 32, where each mailbox has space for 4 filters, meaning there can be up to 128 filters.

The queue slots have the same layout as a message buffer. An empty slot is identified by a zeroed-out control word. The queue is therefore empty when the front slot control word is zeroed and full if the back (sixth) slot has a non-zero control word.

Queue push. The FIFO reception pathway starts in function `flexcan_fifo_rx`, which starts with a filtering stub. The filters can be of different types, selected using `MCR[IDAM]` field. Only the type "`D`" is implemented – it filters out all frames. Other filtering is left as a stub as it is not used by the target driver. All frames pass and are received.

If a frame passes the filter, it is then pushed into the queue, given the queue is not full. For this, the function `flexcan_fifo_find_free_slot` scans the slots and returns a pointer to the first empty one, or `NULL` otherwise. If an empty slot is found, the frame is moved-in using `flexcan_mb_move_in` function. Before the move-in, the control word was zeroed out, as that identifies an empty slot. Therefore the `CODE` field in the control word will be set to `FULL`.

Irrespective of moving-in or the queue being empty, the `flexcan_fifo_push` needs to be called. Its name might perhaps be a little confusing, as the frame was already pushed (moved-in) into the queue, but this function updates the FIFO IRQs after the push. It takes the pointer returned by `flexcan_fifo_find_free_slot` and uses it to find whether

1. the pointer is `NULL` and the queue was full, in which case the `OVERFLOW` interrupt flag is set, or
2. the pointer is not `NULL` and a frame was moved-in and the `AVAILABLE` interrupt flag is set, and
3. whether the occupancy of the queue rose from 4 to 5, in which case the `WARN` interrupt flag is set.

Queue pop. A frame is popped from the queue when the `AVAILABLE` interrupt flag is reset. To do that, the `flexcan_fifo_pop` function moves the top 5 slots one slot down using `memmove` and sets the appropriate interrupt flag according to whether the queue is left empty or not.

Performance. As written above, linear time algorithms are used to manipulate the queue, like scan for the first free slot or `memmove` usage for de-queue. Their use is justified by the small size of the FIFO, $6 \cdot 16\,\mathrm{B} = 96\,\mathrm{B}$. The whole queue comfortably fits into two 64 B cache lines. This implementation, compared to cyclic queue, results in

- Somewhat simpler FIFO manipulation code.
- Less superscalar dependencies between operations in the de-queue code.
- No need for special handling of the queue memory in `flexcan_mem_read`.
- No additional state outside the mailboxes and interrupts.

4 | Use Instructions and Testing

4.1 Build Guide

To use the FlexCAN emulator, build QEMU system emulator for the 32-bit ARM (AArch32) architecture (executable `qemu-system-arm`, target[1] `arm-softmmu`). FlexCAN is built in by default. To speed up the build, disable unnecessary features and guest architectures. List features by:

```
$ $QEMU_SOURCE_DIR/configure --help
Usage: configure [options]
Options: [defaults in brackets after descriptions]

Standard options:
  --help                   print this message

...
```

Recommended configuration command is:

```
$ $QEMU_SOURCE_DIR/configure  --disable-xen --enable-kvm \
    --enable-virtfs --enable-linux-aio \
    --disable-libnfs --enable-libusb --enable-vde --enable-spice \
    --disable-strip --enable-modules --disable-slirp --disable-fuse \
    --enable-install-blobs --enable-trace-backends=simple \
    --target-list=arm-softmmu
```

After configuring, simply use GNU Make default target to build QEMU. [22]

4.2 Usage and Testing Guide

4.2.1 QEMU Virtual Bus

Virtual bus (see subsection 2.4.2) of name `CAN_VBUS_ID` is created using

```
--object can-bus,id=CAN_VBUS_ID
```

command line options. Any number of virtual buses can be created by repeating this option, as

[1] As in QEMU `configure` script sense.

long as they have unique names.

Host SocketCAN Bridge. If a connection between the QEMU virtual CAN bus and a physical host system CAN interface is required, a host interface `HOST_CAN_IF` can be connected to virtual bus `CAN_VBUS_ID` using

```
-object can-host-socketcan,if=HOST_CAN_IF, \
    canbus=CAN_VBUS_ID,id=SOCKETCAN_OBJ_ID
```

command line options,[2] creating a bridge between the host interface and the virtual bus. Note `SOCKETCAN_OBJ_ID` is just a unique name for the bridge object. The name is mostly unimportant.

4.2.2 Virtual SocketCAN Interface

If no physical host CAN interface is present, a virtual CAN interface can be created. This is very useful in cases like:

- A CAN bus connection between multiple virtual machines (QEMU instances) is required.
- CAN communication between software running on the host system and on the VM is required.

If not already loaded, SocketCAN kernel modules must be loaded in using `modprobe`:

```
$ modprobe can-raw vcan;
```

The virtual CAN interface can be created using `iproute2` tools like so (example output included):

```
$ ip link add dev can0 type vcan; # create vcan interface named can0
$ ip link set can0 up; # start the virtual interface
$ ip link show can0; # show details of the interface
16: can0: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN mode
DEFAULT group default qlen 1000 link/can
```

These commands require system network administration capabilities available (and `modprobe` requires system admin capability), run them as *root*. QEMU can connect to this interface just like to any other CAN interface, using the parameters described in the paragraph above.

4.2.3 Running SabreLite machine

Use the Linux kernel with one of the upstreamed SabreLite device trees. One is located in file

```
arch/arm/boot/dts/nxp/imx/imx6q-sabrelite.dts
```

[2] The second command line argument is broken into two lines. The break is marked with a backslash. It is a single argument with no spaces and no backslash.

for a SabreLite with a quad-core processor. While the `sabrelite` machine in QEMU will have two FlexCANs emulated, `flexcan1` and `flexcan2`, the said device tree only describes the former FlexCAN. The device tree needs to be patched to enable support for the second FlexCAN module. The following diff can be applied to enable it (the diff file is also an attachment to this text):

```
diff --git a/arch/arm/boot/dts/nxp/imx/imx6qdl-sabrelite.dtsi
↪  b/arch/arm/boot/dts/nxp/imx/imx6qdl-sabrelite.dtsi
index 9c502bf77d0b..47417a8bdd92 100644
--- a/arch/arm/boot/dts/nxp/imx/imx6qdl-sabrelite.dtsi
+++ b/arch/arm/boot/dts/nxp/imx/imx6qdl-sabrelite.dtsi
@@ -256,6 +256,13 @@ &can1 {
         status = "okay";
 };

+&can2 {
+        pinctrl-names = "default";
+        pinctrl-0 = <&pinctrl_can2>;
+        xceiver-supply = <&reg_can_xcvr>;
+        status = "okay";
+};
+
 &clks {
         assigned-clocks = <&clks IMX6QDL_CLK_LDB_DI0_SEL>,
                           <&clks IMX6QDL_CLK_LDB_DI1_SEL>;
@@ -413,6 +420,13 @@ MX6QDL_PAD_KEY_ROW2__FLEXCAN1_RX        0x1b0b0
                         >;
                 };

+                pinctrl_can2: can2grp {
+                        fsl,pins = <
+                                MX6QDL_PAD_KEY_COL4__FLEXCAN2_TX  0x17059
+                                MX6QDL_PAD_KEY_ROW4__FLEXCAN2_RX  0x17059
+                        >;
+                };
+
                 pinctrl_can_xcvr: can-xcvrgrp {
                         fsl,pins = <
                                 /* Flexcan XCVR enable */
```

A compiled Linux kernel of version v6.12 is provided as an attachment in file `flexcan-scripts-and-patches.tar.xz`, with the patched compiled device tree.

FlexCANs are connected to QEMU CAN buses by passing the bus IDs as machine properties: property `canbus0` for bus `flexcan1` and `canbus1` for `flexcan2`.

Below a command to run QEMU emulating quad-core Sabrelite development board with 1 GiB of RAM is found, with both FlexCANs connected to a single QEMU CAN bus (called `qcan0`), bridged to host system `can0` interface. It bidirectionally connects the terminal to a serial port,

which is by default used for `printk` logging from Linux kernel. It is possible to start a TTY on the serial to access guest system shell directly from the terminal QEMU was launched from. Press `Ctrl+A` and then `h` to show possible QEMU controls. Press `Ctrl+A` and then `x` to kill the emulator and exit. Script `qemu-run-saberlite` for running QEMU this way is attached. In the command, `$LINUX` is a placeholder for a path to a compiled ARM Linux build directory. Similarly, `$INITRD_CPIO` holds for a path to a CPIO archive holding the initial RAM filesystem.

```
$ qemu-system-arm -M sabrelite -smp 4 -m 1G \
    -display none -serial null -serial mon:stdio -nographic \
    -object can-bus,id=qcan0 \
    -machine canbus0=qcan0 -machine canbus1=qcan0 \
    -object can-host-socketcan,if=can0,canbus=qcan0,id=qcan0-socketcan \
    -kernel $LINUX/arch/arm/boot/zImage \
    -dtb $LINUX/arch/arm/boot/dts/nxp/imx/imx6q-sabrelite.dtb \
    -initrd $INITRD_CPIO -append "root=/dev/ram"
```

4.2.4 FlexCAN configuration

The attached initial RAM filesystem CPIO archive (`ramdisk.cpio`) can be used as a minimal userspace. To use it, pass its filepath as the argument to the `-initrd` QEMU command line option. The initialization script in the RAM disk will automatically configure and enable `can0` and `can1` FlexCAN interfaces (it expects the patched device tree from subsection 4.2.3). If using different guest userspace, configure and enable the FlexCAN module using:

```
$ ip link set can0 type can bitrate 1000000 # the bitrate is ignored, but
↪  needs to be set
$ ip link set can0 up # enable the interface
```

If using the second FlexCAN too, run the same commands for `can1`.

4.2.5 Manual Testing

Programs from the CAN-utils[3] package can be used to test CAN communication between the VM and the host or another VM. Two programs are presented: one for transmitting and the other for receiving frames.

For testing, run one of them on the VM, and another on the host (or another VM). Run the receiver program first and exit the transmitting program first, so no frames are missed. Simply compare their outputs. If they are not equal, something was wrong, either were some frames lost, or there was an error in the emulator. Should the FlexCAN emulator receive CAN FD frames, it is expected to ignore them.

[3] https://github.com/linux-can/can-utils

As this form of testing is quite laborious, automated testing has been also included, using the QTest framework. It can be used in CI pipelines.

Transmitting Frames. Program `cangen` can be used to generate random messages and transmit them over a CAN interface. The basic usage (with example output) for transmitting on interface `can0` is:

```
$ cangen -v -v can0
can0  58A   [5]  70 93 98 31 80
can0  3C3   [8]  91 76 0E 3D DB 19 BF 57
can0  3A0   [4]  6D D9 13 3D
can0  3B3   [8]  ED 7D 6A 7B E0 06 4F 42
can0  509   [0]
```

The two `-v` flags cause the program to print every message sent. That is useful for comparing the sent messages to the messages received on the other end of the bus. Additional useful flags are:

- `-e` Generated frames will have the extended identifier instead of the standard (see paragraph 2.2.1).
- `-R` Generate remote request frames.
- `-f`, `-b`, `-E` Generate multiple kinds on CAN FD frames.
- `-n 100` Transmit exactly 100 frames.
- `-g 250` Transmit frames 250 ms apart.

Receiving Frames. For printing received frames, the `candump` program can be used. The format of the printed frames is consistent with `cangen`. For printing frames received on interface `can0`, it can be used this way:

```
$ candump can0
can0  58A   [5]  70 93 98 31 80
can0  3C3   [8]  91 76 0E 3D DB 19 BF 57
can0  3A0   [4]  6D D9 13 3D
can0  3B3   [8]  ED 7D 6A 7B E0 06 4F 42
can0  509   [0]
```

Guest-only testing. If the second FlexCAN is added to the device tree (see subsection 4.2.3) and up, communication between the two FlexCAN instances can be tested. Connect both to a single QEMU CAN bus using the following command line parameters:

```
$ qemu-system-arm ... \
    -object can-bus,id=qcan0 \
    -machine canbus0=qcan0 -machine canbus1=qcan0
```

And then run `candump` in background and `cangen` in foreground in a single TTY:

```
$ candump can0 &
$ cangen -v -v -e can1
can1  0CF0FA97   [8]  D3 8B DD 1F 25 F1 C5 1B
can0  0CF0FA97   [8]  D3 8B DD 1F 25 F1 C5 1B
can1  04FC3E04   [1]  4C
can0  04FC3E04   [1]  4C
```

5 | Conclusion

A FlexCAN functional emulator has been developed and integrated into QEMU, with full support of the following features:

- Transmitting data frames.
- Receiving data frames into message buffers. Searching for empty message buffers. Waiting for the message buffer to get unlocked before moving in the frame.
- Receiving data frames in Rx-FIFO. Correct queueing and de-queueing. Availability, warning and overflow interrupts.
- Message buffer locking mechanism.
- All interrupt sources are considered.
- Unpriviledged access to some FlexCAN registers in "user mode".
- Loopback mode & listen-only mode.
- Freeze mode and module disable low-power mode. Transitions between operation modes.

These features are partially supported:

- Serviced (`SRV`) flag on message buffers and proper use of `FULL` and `OVERRUN` codes.
- Remote request frames, supported only if received as ordinary frames (eg. bit `CTRL2[RRS]` is set).
- Order of reception mechanisms — in some situations, the Rx-FIFO should be, for example, tried first before moving the frame to a not-free-to-receive message buffer. This emulator implements a simplified mechanism — it tries Rx-FIFO and then message buffers, or the other way around (depending on configuration), but does not interleave the mechanisms as in the example.

These features are not supported:

- Searching for a receiving message buffer based on its ID field and related masks. The filters are ignored and all frames are received.
- Rx-FIFO ID filter table is ignored and every frame is received.
- Automated remote request frame responses.
- Transmitting frames from mailboxes written during freeze or low-power mode.

- "Self Wake Up" feature.
- Free-running timer and timestamping frames.
- Stop mode.

All key features used by the Linux FlexCAN driver have been implemented and tested. That means the guest application can use the SocketCAN interface of the emulated kernel to send and receive CAN frames, including both data frames and remote request frames. Standard and extended IDs are supported.

Many features are tested automatically in programmed integration QTest tests.

5.1 Discussion

The Rx-FIFO is implemented, apart from filtering, even though the upstream Linux kernel cannot be configured to use Rx-FIFO on i.MX6 without editing its source code.

- Test the emulator under heavy guest CPU load and/or high frame throughput. The driver would have less time to service the interrupts, which would drive the emulator towards edge cases, like overflowing FIFO or full message buffers. Additionally, the drop rate could be compared to other emulated CAN devices, and, if automated, would make an interesting test for performance regressions.
- Test the emulator with other drivers. Diverse usage of the emulator will lead to more bugs discovered and fixed.
- Test the emulator by fuzzing the memory-mapped IO. This means essentially writing random data into the memory-mapped registers, and waiting for the instance to, for example, crash due to segmentation fault. That should not happen with any input and it would mean a memory violation bug is present, which could allow malicious guest to crash the VM or even gain code execution in QEMU and take control of the host machine.[1]

Support for the Zephyr driver has not been tested but is expected to work properly when not using its filtering features.

FlexCAN documentation specifies an unrestricted access should behave "as though the access was done to an unimplemented register location". The implementation will cause both (unrestricted access or access to unimplemented registers) to generate a memory violation exception.

Depending on the bit timing configuration, FlexCAN may not be able to scan all filters or message buffers in time when there are frames transmitted directly after each other. Emulating this would require the QEMU CAN subsystem to implement finite virtual bus bitrate.

[1] This is not that important, given QEMU security model does not treat this as a considered attack vector during non-virtualized emulation (which is the expected use case).[20]

After discussion with a QEMU CAN subsystem maintainer, support for error frames on virtual buses needs further work and FlexCAN implementation ignores them.

5.1.1 Future CAN FD Support

In this section, a plan for CAN FD support is presented. As i.MX8 series support is planned to be added to QEMU in the medium-term future, support for FlexCAN3 would make i.MX8 emulation more complete.

QEMU CAN subsystem already supports FD frames. I will list the main differences between FlexCAN2 and FlexCAN3, which would, in my opinion, require larger, non-trivial changes to the current implementation:

1. Variable mailbox size. When CAN FD is enabled, each half of the message buffer RAM block can be partitioned in 4 different ways, allowing the half to store from 7 message buffers of up to 64 B payloads to 32 message buffers of up to 8 B payloads. This would require deeper changes to handling message buffers. For example, the start offsets of the message buffers would have to be dynamically computed. A simple iteration over all message buffers would depend on the partitioning.
2. DMA functionality. Enables DMA transfer of Rx-FIFO received frames to memory. Enabled by `MCR[DMA]`.

References

[1] NXP SEMICONDUCTORS. *i.MX 6Dual/6Quad Applications Processor Reference Manual*. Manual, rev. 6. Eindhoven, Netherlands, May 2020. 5 642 pp.

[2] NXP SEMICONDUCTORS. *S32K3xx Communication Modules: FlexCAN*. Online. 2021. [Accessed 12 May 2025]. Available from: https://www.nxp.com/webapp/Download?colCode=17_S32K3XX_COMMUNICATION_MODULES_FLEXCAN_TRAINING.

[3] CHARVÁT, Jan. *Model of CAN FD Communication Controller for QEMU Emulator*. Online, bachelor's thesis. Prague: Czech Technical University in Prague, 2020. Available from: https://dspace.cvut.cz/handle/10467/87714.

[4] ISO/IEC 9899:2011. *Information technology – Programming languages – C*. Geneva: International Electrotechnical Commission, 2011.

[5] ISO 7498-1:1994. *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. Geneva: International Organization for Standardization, November 1994.

[6] ISO 11898:2003. *Road vehicles — Controller area network (CAN)*. Geneva: International Organization for Standardization, December 2003.

[7] CAN IN AUTOMATION. *CAN Specification 2.0*. 1999.

[8] KVASER. *Controller Area Network (CAN BUS) Protocol*. Online. May 2024. [Accessed 9 May 2025]. Available from: https://kvaser.com/can-protocol-tutorial.

[9] KVASER. *CAN FD Protocol Tutorial*. Online. May 2024. [Accessed 9 May 2025]. Available from: https://kvaser.com/can-fd-protocol-tutorial.

[10] CORRIGAN, Steve. *Introduction to the Controller Area Network (CAN)*. Online. Dallas: Texas Instruments, May 2016. [Accessed 10 May 2025]. Available from: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1746881015496.

[11] KLEINE-BUDDE, Marc *et al. Linux – FlexCAN driver – drivers/net/can/flexcan/flexcan-core.c*. Source code file, online. Version 6.12. [Accessed 11 May 2025]. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/can/flexcan/flexcan-core.c?h=v6.12.

[12] KLEINE-BUDDE, Marc *et al. Linux – FlexCAN driver – drivers/net/can/flexcan/flexcan.h*. Source code file, online. Version 6.12. [Accessed

11 May 2025]. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/can/flexcan/flexcan.h?h=v6.12.

[13] VESTAS WIND SYSTEMS. *Zephyr – FlexCAN driver – drivers/can/can_mcux_flexcan.c*. Source code file, online. Version 4.1.0. [Accessed 11 May 2025]. Available from: https://github.com/zephyrproject-rtos/zephyr/blob/v4.1.0/drivers/can/can_mcux_flexcan.c.

[14] *CAN Bus Emulation Support*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/system/devices/can.html.

[15] *System Emulation — Device Emulation*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/system/device-emulation.html.

[16] *QEMU User space emulator*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/user/main.html.

[17] *System Emulation — Introduction*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/system/introduction.html.

[18] *Internal Subsystem Information — Migration framework*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/devel/migration/main.html.

[19] *Supported build platforms*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/about/build-platforms.html.

[20] *Security*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/system/security.html#security.

[21] *Reset in QEMU: the Resettable interface*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/devel/reset.html.

[22] *The QEMU build system architecture*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemuorg/docs/master/devel/build-system.html.

[23] *The QEMU Object Model (QOM)*. Online. In: QEMU Documentation. [Accessed 11 May 2025]. Available from: https://www.qemu.org/docs/master/devel/qom.html.

[24] SILVACO. *FlexCAN Controller*. Online. Product brief, 15 Apr 2024 [Accessed 11 May 2025]. Available from: https://silvaco.com/wp-content/uploads/product/ip/pdf/70017_FlexCAN_brief.pdf.

[25] SILVACO. *Silvaco Enters IP Market With Acquisition of IPextreme*. Online, press release, 3 Jun 2016. [Archived 30 Aug 2019, archive accessed 11 May 2025]. Available from: https://silvaco.com/news/pressreleases/2016_06_03_01.html. Archived in the Wayback Machine at archive.org.

[26] SILVACO. *Silvaco Expands Automotive IP Portfolio With Release of CAN FD Core*. Online, press release, 28 Jun 2016. [Archived 23 Mar 2025, archive accessed 11 May 2025]. Available from: https://silvaco.com/news/silvaco-expands-automotive-ip-portfolio-with-release-of-can-fd-core. Archived in the Wayback Machine at archive.org.

[27] CAN IN AUTOMATION. *History of the CAN technology*. Online. [Accessed 10 May 2025]. Available from: https://www.can-cia.org/can-knowledge/history-of-can-technology

[28] EMERICH, Annegret; ZELTWANGER, Holger. *CAN FD chips: different buffer features*. Online. *CAN Newsletter*. September 2016, pp. 4–7. [Accessed 11 May 2025]. Available from: https://www.can-cia.org/fileadmin/cia/documents/publications/cnlm/previous_issues/16-3_cnlm.pdf.

[29] LINUX FOUNDATION. *Processor support — KVM*. Online. [Accessed 11 May 2025]. Available from: https://linux-kvm.org/page/Processor_support.

[30] ARM HOLDINGS. *Learn the architecture — Generic Interrupt Controller v3 and v4, Overview*. Online. [Accessed 14 May 2025]. Available from: https://developer.arm.com/documentation/198123/0302/What-is-a-Generic-Interrupt-Controller-.

[31] ARM HOLDINGS. *ARM Cortex-A57 MPCore Processor Technical Reference Manual*. Online. [Accessed 11 May 2025]. Available from: https://developer.arm.com/documentation/ddi0488/d/programmers-model/armv8-architecture-concepts/aarch32-execution-modes.

Appendices

A | Software Versions

The versions of software used are:

- Linux kernel — version 6.12
- Zephyr kernel — version 4.1.0
- QEMU — Git VCS commit `e28fbd1c525db21f0502b85517f49504c9f9dcd8`
- CAN utils[1] — version 2023.03
- GCC — version 15.1.1

Linux and Zephyr versions were the latest stable releases at the time of development. This work is based upon the QEMU commit. It was the head of QEMU main development branch at the time of development.

[1] https://github.com/linux-can/can-utils

B | Used AI Tools

Google Gemini Deep Research was used for FlexCAN history and origins research. Google Gemini, OpenAI ChatGPT and Grammarly were used to check this text for lexical and grammatical errors. No part of this text and any created and/or modified source code was written by a large language model.