OSES Project

Emulating the NXP S32K3X8EVB board with Qemu

Authors:

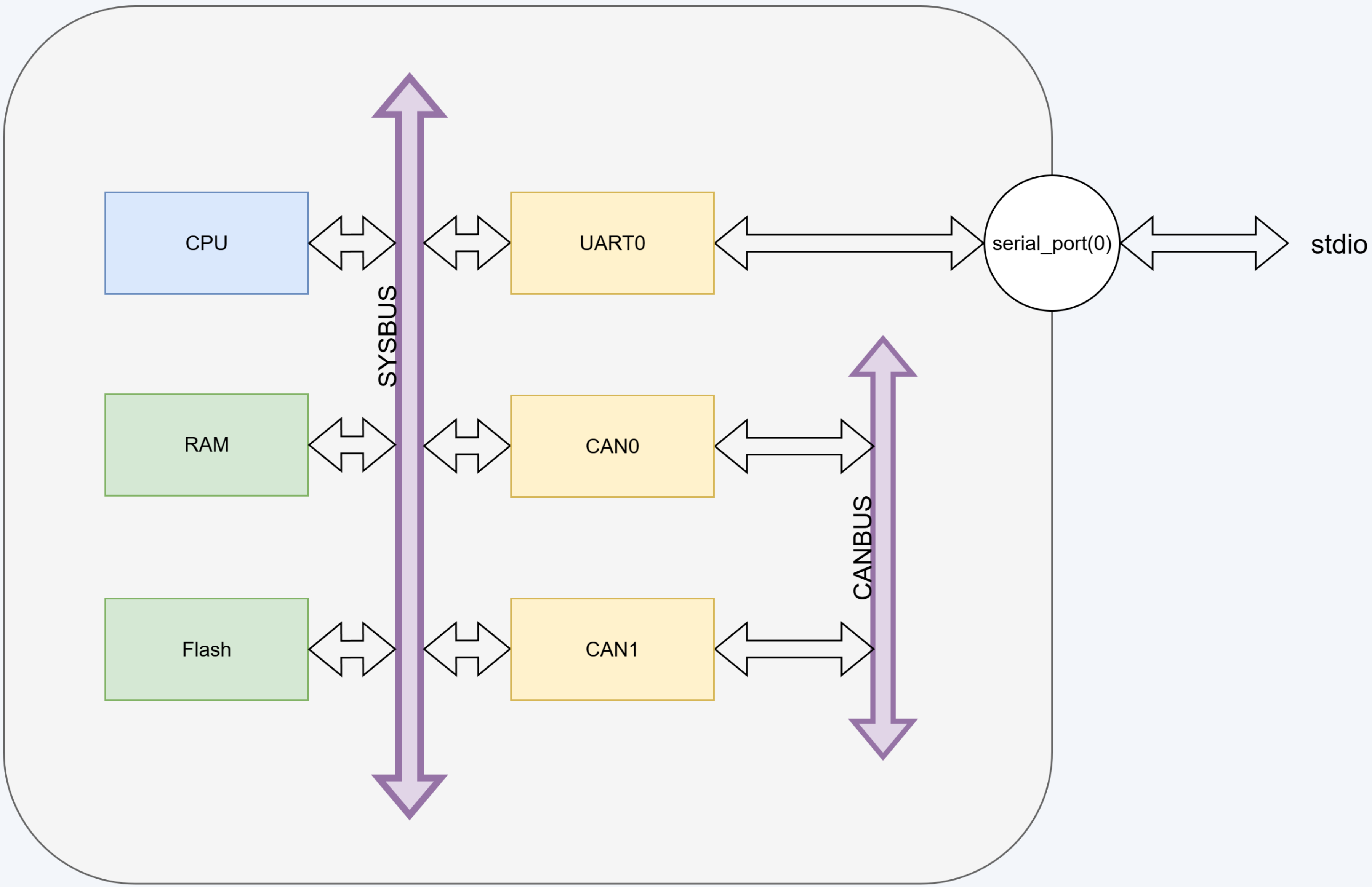
Stefano **Galati** (342146)

Claudio Pio **Perricone** (333942)

Matteo **Ruggeri** (345046)

Gabriele **Arcidiacono** (343444)

The Microcontroller



Part 1: Board Emulation

Introduction

- The NXP S32K3X8EV is based on the microcontroller NXP SK358.
- The microcontroller is based on ARMv7M, there are three Cortex M7 cores, with two configured in lockstep mode.

Defining a new board

1. The board is defined as a **QEMU** machine, composed by **`MachineClass`** and **`MachineState`**, which are structures defined by QEMU to emulate a complete system:
   - **`MachineClass`** abstracts and encapsulates the emulated platform
   - **`MachineState`** tracks the instance of a machine
2. The microcontroller and all the other peripherals are defined as QEMU devices and treated like objects, with the following structures:
   - **`DeviceClass`** and **`DeviceState`**: structures to emulate a particular device, such as a microcontroller

Microcontroller

The microcontroller is composed by:

- A **`Sysbus`** : a generic virtual bus realized by QEMU used to interconnect all the devices
- The Cortex-M7 and its instruction set
- A **`MemoryRegion`**, an abstract way to represent the physical memory needed by the device
- A **`MemoryRegion`** Alias, needed to correctly emulate the load of a kernel

Testing and FreeRTOS

Finally we had to test the functionalities of the board:

1. First we tried with a simple bare-metal application:
    - We developed a suitable linker to define all the memory sections of the board, and the entry point of the program;
    - We coded a startup file, with the correct interrupt vector table, and a reset and default handler;
    - We also wrote a new Makefile with option for debugging.
2. Then was the turn of FreeRTOS:
    - We modified the configuration files;
    - We wrote a new main file including the FreeRTOS functionalities.

Multicore

Last step for the board was to add the additional cores.

The major problems where:

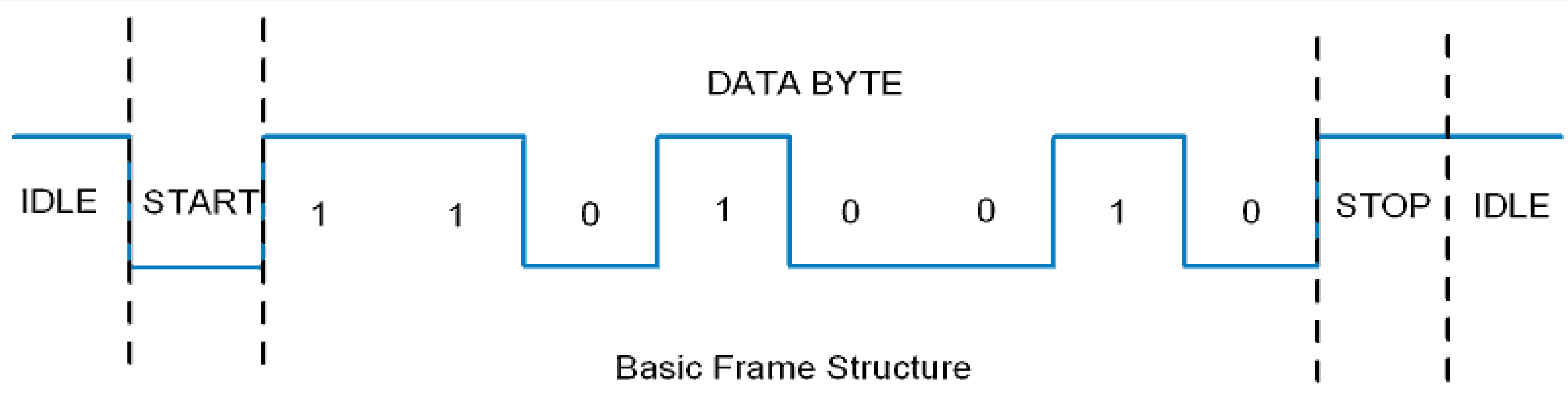
- Changing how the memories where define, and make some private for each core, and some shared
- Understanding how the boot sequence would work
- Making the interrupt working on the cores

Part 2: Peripherals emulation

In this part we discuss the emulation of UART and CAN peripherals.

UART

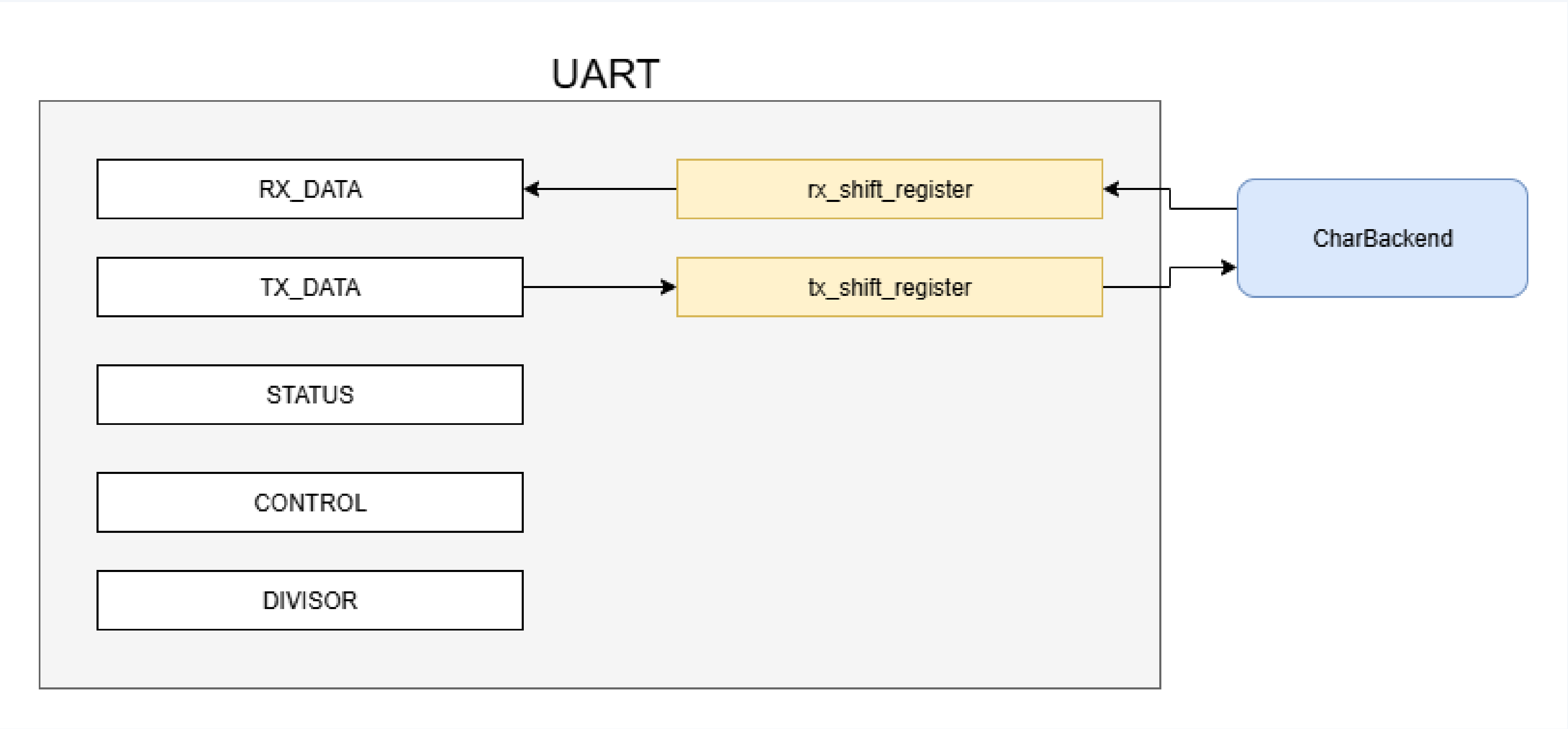
UART (Universal Asynchronous Receiver Transmitter) is a device used for full duplex point-to-point serial communication.



Altera Nios II UART Peripheral

- 1 RX shift register + 1 TX shift register
- No FIFOs
- STATUS register:
  - **PE**
  - **FE**
  - **BRK**
  - **ROE**
  - **TOE**
  - **TMT**
  - **TRDY**
  - **RRDY**
  - **E**
- Control register for interrupt activation
- 115200, 8E1 with configurable baudrate

Schematic view of our UART

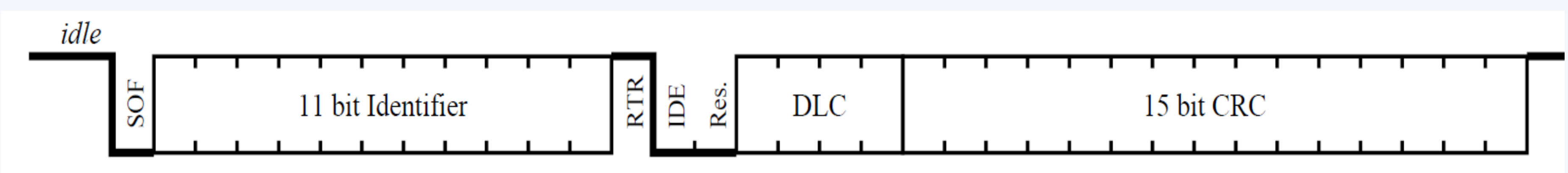


INSTANTIATING A UART DEVICE

- Implementation requires creating `uart.c` and `uart.h`
- Peripheral state includes registers, IRQ, and Character Backend
- The UART device logic handles:
  - Transmission and reception from CharBackend
  - Changing transmission baudrate
  - Behaviour of STATUS register
  - Interrupt

CAN

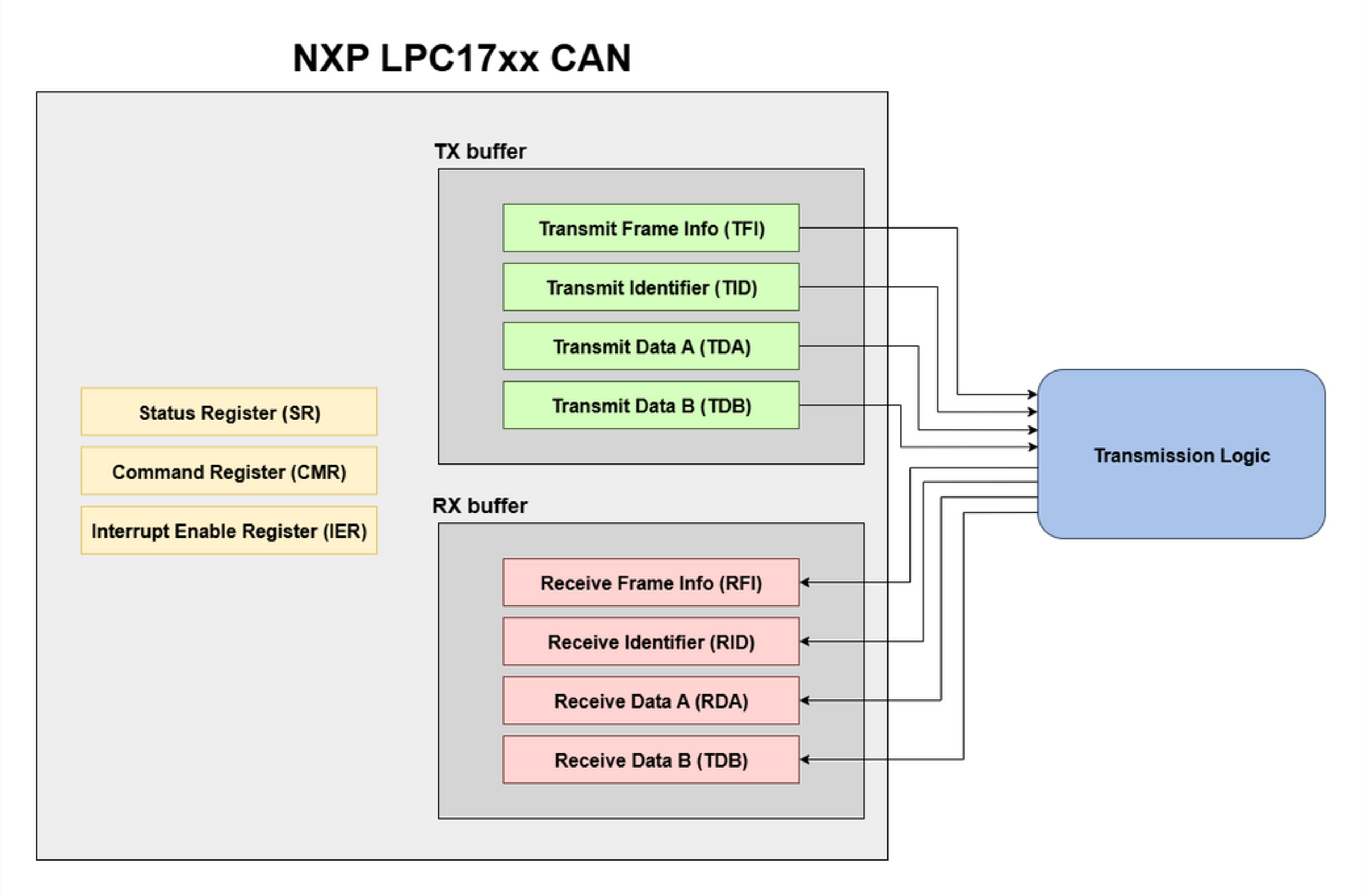
- CAN (Controller Area Network) is a shared bus protocol.
- Unlike UART (point-to-point), CAN uses a bus where all nodes receive frames.
- In Qemu, a CAN bus object is instantiated and CAN peripherals are connected to it.



LPC17xx CAN

- Based on NXP LPC17xx CAN controller (simplified).
- Registers: TX buffer (`TFI`, `TID`, `TDA`, `TDB`), RX buffer (`RFI`, `RID`, `RDA`, `RDB`), `SR`, `CMR`, `IER`.
- Transmission: data written to TX buffer, a bit in `SR` starts transmission.
- Reception: frames copied to RX buffer, interrupts optionally raised on events.

Schematic view of our CAN



INSTANTIATING A CAN DEVICE

- Implementation requires creating `can.c` and `can.h`.
- Peripheral state includes registers, IRQ, and CAN bus connection.
- The CAN device logic handles transmit, receive, interrupts, and bus connection.

Example application