Politecnico di Torino

Implementation and Emulation of a new board for qemu

CUGLIARI ALEX S343766
DI PEDE ANTONELLO S347895
D'AMICO DENNIS S343841
LAURETTI MATTIA S338720

Author's address: Cugliari Alex S343766
Di Pede Antonello S347895
D'Amico Dennis S343841
Lauretti Mattia S338720.

ACM XXXX-XXXX/2025/2-ART
https://doi.org/

Contents

0 INTRODUCTION TO QEMU

QEMU (Quick Emulator) is an open source software that acts as an emulator and virtualiser, enabling the execution of operating systems and applications developed for different hardware architectures on a single host. Thanks to QEMU, it is possible to create virtual environments that faithfully reproduce the behaviour of physical hardware, enabling the development, testing and debugging of software in a flexible manner and without the need for dedicated physical machines. QEMU supports numerous architectures, including x86, ARM, PowerPC and MIPS, making it a particularly useful tool in research, development and simulation contexts.

In our project, the aim is to extend QEMU's capabilities to emulate a new board called sk32k3x8evb. In particular, we are focusing on the realisation of a customised UART and GPIO for this board, in order to be able to test and develop hardware-specific applications without the need for the physical board.

1 CORRECT EMULATION IN QEMU OF THE CORTEX-M7

The first thing we need to do in order to work with QEMU is to install it, which can be done safely via terminal by downloading from github. Once downloaded we notice how there are many folders inside them, this is because natively some boards are already emulated. This large number was made possible by the fact that QEMU turns out to be open source and therefore any developer can apply updates. To begin we need to go inside the folder **`hw/`** (hardware). Within this folder are emulations of peripherals, buses, and interfaces. We have focused mainly on arm and char:
`hw/arm:` This directory collects the implementation of hardware models specific to the ARM architecture. Specifically, it contains code that emulates CPUs, boards, peripherals, and other components typical of ARM-based systems, allowing QEMU to simulate the hardware environment and development platforms typical of this architecture.
`hw/char:` The hw/char directory is dedicated to the emulation of character devices, i.e., those devices that handle data streams in text format or character sequences. Within it are implemented templates for serial ports, virtual consoles, and other character-based I/O devices, which are critical for communication, debugging, and interaction with the emulated system.
We chose the arm folder since our board exploits a processor with that architecture namely Arm CORTEX-M7. Before we started reading manuals and online forums we noticed how there were many boards that support this processor, we in particular decided to take a cue from the mps2, another board natively implemented on QEMU (**`hw/arm/mps2.c`**). It should be specified that the within the mps2 code several fpga are generated, the one most similar to our board is the an500, which was also the model we followed.

Basic board emulation

The S32K3X8EVB board is an evaluation board developed by NXP for the S32K3 family of microcontrollers, from which it retains some features, through the use of material found online, including manuals, we began to gather information regarding the architecture, addresses, and various memory size and devices. We completed the search by finding:

Memory Region	Address Range
SRAM0	0x20400000 - 0x2043FFFF
SRAM1	0x20440000 - 0x2047FFFF
SRAM2	0x20480000 - 0x204BFFFF
Flash	0x00400000 - 0x00BFFFFF
ITCM	0x00000000 - 0x00008000
DTCM	0x20000000 - 0x2000F000

Addresses that are generally present inside the board right out of the factory. So once we understood that, first we set the machine name, included it in QEMU, and set its clock (The manual generally says that the clock is 240 MHz).
From here on, we wrote the _init function that encapsulates all the operations needed to initialize the board and the various peripherals. The creation and mapping operations mainly concern FLASH and RAM. All this was always accomplished following the model of the mps2- an500.
Next we initialized the processor and set up superficially how it should handle interrupts, then we connect, virtually, the created clock, memory, and cpu-type to the processor. Finally with `sysbus_realize()` we make the emulated device operational, integrating it into the system so that it can interact properly with the other components of the emulated system.
Finally to conclude with the function _init we have:

- **Loading the kernel:**. The kernel is loaded from the location indicated by the file specified in `machine->kernel_filename` in memory, starting at address `0x20400000`.
- **Configuration of VTOR:** The VTOR register is set to `0x000000`, so that the interrupt table (IVT) is the one located in the ITCM starting at address `0x000000`.

These operations are essential to prepare the ARMv7-M processor for kernel execution and to ensure that interrupts are handled correctly by routing requests to the correct handler table.
Speaking of the last functions we also have :

- **`S32K3X8EVB_class_init:`** Configures S32K3X8EVB machine-specific properties (description, default CPU, RAM, initialization function, etc.) inheriting from the generic machine type.

- **`S32K3X8EVB_info`:** Provides all the meta information (name, size, class, initialization function) needed to define the new machine type in the QEMU type system.
- **`s32k3x8evb_machine_init`:** Registers the new machine type in QEMU, making it available for use.

. Together, these functions integrate the new machine type S32K3X8EVB into the QEMU framework, enabling its proper configuration and instance in the emulated system.

```
Antonello@UbuntuCAOS:~/Desktop/qemu_nxp_s32k3x8evb/build$ ./qemu-system-arm -M s32k3x8evb -nographic -S -s -kernel test.elf
QEMU 9.1.91 monitor - type 'help' for more information
(qemu) info mtree
address-space: cpu-memory-0
  0000000000000000-ffffffffffffffff (prio 0, i/o): armv7m-container
    0000000000000000-ffffffffffffffff (prio -1, i/o): system
      0000000000000000-0000000000007fff (prio 0, ram): s32k3x8evb.itcm
      0000000000400000-0000000000bfffff (prio 0, rom): s32k3x8evb.rom
      0000000020000000-000000002000efff (prio 0, ram): s32k3x8evb.dtcm
      0000000020400000-000000002043ffff (prio 0, ram): s32k3x8evb.sram0
      0000000020440000-000000002047ffff (prio 0, ram): s32k3x8evb.sram1
      0000000020480000-00000000204bffff (prio 0, ram): s32k3x8evb.sram2
      0000000030200000-0000000030200fff (prio -1000, i/o): cmsdk-ahb-gpio
      0000000040328000-0000000040328fff (prio 0, i/o): s32k3x8evb_uart
    00000000e0000000-00000000e00fffff (prio -1, i/o): nvic-default
    00000000e000e000-00000000e000efff (prio 0, i/o): nvic_sysregs
    00000000e000e010-00000000e000e0ef (prio 1, i/o): v7m_systick

address-space: memory
  0000000000000000-ffffffffffffffff (prio -1, i/o): system
    0000000000000000-0000000000007fff (prio 0, ram): s32k3x8evb.itcm
    0000000000400000-0000000000bfffff (prio 0, rom): s32k3x8evb.rom
    0000000020000000-000000002000efff (prio 0, ram): s32k3x8evb.dtcm
    0000000020400000-000000002043ffff (prio 0, ram): s32k3x8evb.sram0
    0000000020440000-000000002047ffff (prio 0, ram): s32k3x8evb.sram1
    0000000020480000-00000000204bffff (prio 0, ram): s32k3x8evb.sram2
    0000000030200000-0000000030200fff (prio -1000, i/o): cmsdk-ahb-gpio
    0000000040328000-0000000040328fff (prio 0, i/o): s32k3x8evb_uart

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io
```

Fig. 1. Output of info mtree

2 MAPPING OF UART AND GPIO

What are they?

The UART (Universal Asynchronous Receiver/Transmitter) is a fundamental hardware component for asynchronous serial communication. It enables the transmission and reception of data over a serial connection, without the need for a shared clock signal between communicating devices. This mechanism allows simple and efficient communication to be established between the microcontroller and other devices, such as computers, communication modules or sensors.
In the emulated environment of QEMU, the implementation of the UART allows the behavior of a serial port to be simulated, which is useful for:

- Send commands and debug data to the system.
- Receive responses from emulated external devices.
- Test proper error handling and transmission synchronization.

GPIOs (General-Purpose Input/Output) represent programmable pins found in microcontrollers that are used to interface with the outside world. These pins can be configured either as inputs (to read the status of sensors, buttons or other digital signals) or as outputs (to control actuators, LEDs, motors, relays, etc.).
The emulation of GPIOs in QEMU allows external events to be simulated and verified:

- The correct reading of input signals from external devices.
- The ability of the system to send output signals to control hardware components.
- The interaction between GPIOs and other features of the emulated core.

What have we done?

First of all, we also took our cue from the arm manuals and the mps2-an500 board's code in particular we found these links very useful:

- **S32K3 Memories Guide**
- **APB-UART**
- **S32K388EVB-Q289_HWUM.pdf - (on github)**
- **S32K3XXRM.pdf - (on github)**

First of all, the first thing to do in order to successfully implement the uart is to start working within the '**`hw/arm/`**' folder and create the configuration files for our new uart. Inside the folder, we create the files '**`s32k3x8evb_uart.c`**' and '**`s32k3x8evb_uart.h`**' and start to insert all functions required to correctly initialise the uart. In order, what we did was :
In order what we did was :

- The '**`s32k3x8evb_uart_realize()`**' function is the part of the code that allows the UART to be initialised with the various default values. First the baud rate, a number representing the transmission speed, is set. Next, the control registers are set, and then the ouput is set to stdout for data transmission. The memory regions are then connected to the bus and a line is created for IRQ (interrupt request). Finally, the memory for the UART is mapped.

- **' s32k3x8evb_uart_read()**' functions allow reading and are invoked by QEMU when the guest attempts to perform an operation of this type. In general the function outputs the contents of the registers according to the address taken as input (addr), within it we find a switch-case where the address check takes place such as :

Address	Returned Variable
`0x10`	`uartDev->uartBaud`
`0x18`	`uartDev->uartCtrl`
`0x14`	`uartDev->uartStat`
`0x1C`	`uartDev->uartData`
Default: Value 0 (invalid Address)	

- functions "**`s32k3x8evb_uart_write()`**" is invoked when the system writes to an address mapped by the UART. It initially makes a log for possible debugging, then a switch-case similar to the read function with the following features:

Address	Action
0x10	Updates the Baud Rate register by setting 'uartDev->uartBaud' to the written value.
0x18	Updates the Control register by setting 'uartDev->uartCtrl' to the written value and enables/disables interrupts by checking the '**UART_CTRL_INTERRUPT_ENABLE**' bit.
0x1C	Updates the Data register for transmission by setting 'uartDev->uartData' to the written value. Logs the transmitted character. If an output stream is set, writes the character to the stream and flushes it. Sets the '**TX_COMPLETE**' flag in 'uartDev->uartStat' and triggers the transmit interrupt.
0x14	Clears the '**TX_COMPLETE**' flag in the Status register by removing it from 'uartDev->uartStat'.
Default: Invalid address. Generates a log message indicating the error.	

- the '**uart_tx_interrupt()**' function is responsible for checking whether the conditions for activating the 'interrupt (TX)' are met and, if so, for activating the interrupt. It checks whether the interrupts for the UART are enabled and checks whether the "transmission complete" flag is set in the UART status register with the 'AND' operator bit by bit. Finally it writes a log message.

- The file "**s32k3x8evb_uart.h**" contains all of the prototypes of variables and functions that were later created within the ".c" file. One of the main pieces of information is the addresses we chose:

Macro	Value	info
S32K3X8EVB_UART0_BASE	0x40328000	Custom UART base address
S32K3X8EVB_UART_BAUD	(S32K3X8EVB_UART0_BASE + 0x10)	Baud Rate Register
S32K3X8EVB_UART_STAT	(S32K3X8EVB_UART0_BASE + 0x14)	Status Register
S32K3X8EVB_UART_CTRL	(S32K3X8EVB_UART0_BASE + 0x18)	Control Register
S32K3X8EVB_UART_DATA	(S32K3X8EVB_UART0_BASE + 0x1C)	Data Register

Going forward with the creation of the uart, we must move to the folder where our 'main' is currently located (in our case within the **FreeRTOS Project/ Demo/**) and create the basic functions that will be used by the main.
Within this folder there are two files 'uart.c' and 'uart.h' and in them are defined:

- the function '**my_uart_init()**' which initialises the uart with a certain baud rate and enable the interrupt.
- the function '**my_uart_transmit_char()**' which allows a single character to be transmitted via the UART. Specifically, pointers to the UART registers are defined, then the transmitter is waited for ready, in fact the function enters a while loop that continues to iterate until the **UART_FL_TX_DONE** flag is set in the status register. Then the "**my_stat_reg**" flag is reset and The character to be transmitted is written to the data register ("**my_dat_reg**"). Finally, after sending the character, the function again enters a while loop to wait for the **UART_FL_TX_DONE** flag to be reset.
- the function "**my_uart_printf()**" reads the string and uses a for loop in which it breaks down the string character by character and has them sent individually by the previously created function "**my_uart_transmit_char()**"

This set of operations and function allows the UART to be integrated into the emulated system, making it operational and ready to interact with the CPU via interrupts, while the GPIO device is registered as a placeholder to be made functional later. Unfortunately, despite all the effort and work put into finding information and writing code, we were not able to make the GPIO fully functional, but only to map it to its memory address.

3 FREERTOS PROJECT RUNNING ON QEMU

What is FreeRTOS

FreeRTOS is an open source real-time operating system (RTOS) designed specifically for microcontrollers and embedded systems with limited resources. Its light and efficient kernel allows deterministic handling of multitasking, interrupts, synchronization between tasks, and memory management, which are fundamental for real-time applications.
In our project, FreeRTOS runs inside QEMU, taking advantage of the power of emulation to simulate a full hardware environment without needing the physical board. The integration of FreeRTOS with QEMU permits rapid and flexible development and testing of real-time applications. Compilation and design optimization take place using the NXP Toolchain, which offers specific tools for ARM Cortex-based platforms, ensuring optimal performance and tight integration with the FreeRTOS kernel.

Project composition

Our project consists of many different files, in particular :

- **`main.c`**:
  - Contains the main function, (the `main.c()`) which is the first function to be executed.
  - In this file, the hardware peripherals are configured and initialized, but they have been mapped in the s32k3x8evb.c file.
  - Then at the beginning a task is created and the FreeRTOS scheduler is started to handle concurrent execution (vTaskStartScheduler();).
- **`startup.c`**:
  - Contains the initialization code executed before the call to `main()`.
  - Takes care of setting the stack pointer and initializing the memory.
  - Configures the interrupt table and prepares the hardware (in this case emulated) to execute the application code.
- **`syscalls.c`**:
  - Implements system calls (syscalls) needed for the low-level operations.
  - Generally defines functions like `_write`, `_read`, `_close`, etc., which are useful for retargeting standard I/O functions (e.g., `printf` and `scanf`) to hardware peripherals, but in our case we have left some of them as temporary.
- **`linker.ld`**:
  - This is the linker script that defines how the maps code and data should be organized and mapped in the microcontroller's memory.
  - Specifies the memory regions (e.g. Flash for the code and RAM for the data) and the different sections (`.text`, `.data`, `.bss`, etc.), while also taking into account information gained from the manuals so that the software can be linked to the hardware.
  - Ensures that the program is properly placed in memory for the correct execution.

4 MPU

What is MPU?

The MPU (Memory Protection Unit) is a hardware component found in many microcontrollers and embedded systems, whose main purpose is to protect memory and improve system security. Its main functions are:

- **Definition of memory regions**. The MPU allows the memory space to be divided into distinct regions. Specific attributes can be defined for each region, such as read, write and execute permissions.
- **Access control:** The configuration of regions makes it possible to ensure that only authorized code can access certain areas of memory, thereby preventing accidental errors or malicious attacks.
- **System Isolation:** By separating system (or kernel) code from application code, the MPU helps maintain system integrity and stability. This isolation is critical in real-time environments, where an error in one task could compromise the entire application.
- **Security Support:** By configuring the memory , the MPU reduces the attack surface, limiting the risk that bugs or exploits could compromise the operation of the system.

In summary, the MPU is a very important resource for ensuring secure and controlled memory management, isolating the different software components and protecting the system from unauthorised access or errors that could lead to unpredictable behaviour.
In our project, the Memory Protection Unit or MPU tries to emulate the memory access management functionality of the s32k3x8evb-q289 board within QEMU. The MPU manages memory permissions and accesses, providing a protection barrier for memory that is essential for system security. Our implementation, in particular, focused on emulating the MPU's privilege-based memory access control capabilities, specifically the ability to restrict access to protected memory regions. One of our first approaches involved the creation of software that emulated the main functionalities of the MPU, focusing on configuration of distinct memory regions with different access permissions and the validation of memory protection mechanisms through controlled accesses.
It was later extended to include modifications to the main components of QEMU to achieve a more authentic hardware emulation. The project focused in particular on the implementation of memory access validation, privilege level control and error generation mechanisms that attempt to emulate the behaviour of the MPU's physical hardware.

MPU Register Overview

The MPU implementation is controlled through a set of memory-mapped registers. The following table details the key registers and their functionalities:

Register	Address	Description	Access
MPU_TYPE	0xE000ED90	Provides information about the MPU implementation	Read-only
MPU_CTRL	0xE000ED94	Controls the operation of the MPU	Read/Write
MPU_RNR	0xE000ED98	Selects the region to be configured	Read/Write
MPU_RBAR	0xE000ED9C	Specifies the base address of the selected region	Read/Write
MPU_RASR	0xE000EDA0	Defines the size and attributes of the selected region	Read/Write

MPU Control Register (MPU_CTRL) Bit Fields

Bit	Name	Description
0	ENABLE	Enables the MPU.
1	HFNMIENA	Enables MPU during hard fault, NMI, and FAULTMASK handlers.
2	PRIVDEFENA	Enables privileged software access to the default memory map.

Architectural Overview

In the initial phase, we created a dedicated directory 'MPU_project' containing a draft of the complete software to demonstrate the functionality of the MPU. This 'proof of concept' was structured around several key components: a memory management system defined in mpu.c and mpu.h, supporting configuration files (linker.ld, startup.c) and system interface components (syscalls.c). The main functionality of the MPU was implemented by means of memory-mapped registers, namely MPU_TYPE, MPU_CTRL, MPU_RNR, MPU_RBAR and MPU_RASR, mapped to the respective ARM Cortex-M7 addresses starting at 0xE000ED90, addresses always taken from the manuals.

The architecture of the implementation was, in fact, created on two primary operational aspects:
First, the region configuration system allowed the definition of memory regions that protected with specific access permissions. This was implemented within the MPU_Init() function, within the mpu.c file. Subsequently, the privilege management system was implemented through changes to the control registers.
For the integration phase of QEMU, we modified the directories hw/arm and target/arm. The purpose of these changes was to correctly manage memory access attempts according to the configured MPU regions and the current privilege level of the processor.

Software-Based Demonstration Phase

The first phase focused on the creation of a demonstration framework for MPU. The initial implementation used memory-mapped registers to configure protection regions and implemented privilege-based access controls. The framework included MPU initialisation logic, region configuration and a memory error handling system. Although this approach succeeded in demonstrating the conceptual operation of an MPU, unfortunately, it was unable to provide true memory protection at the hardware level, as the validation of memory accesses occurred at the software level rather than through hardware emulation.

The second phase shifted the focus to hardware-level emulation through direct integration with QEMU. The implementation included a dedicated MPU device (cortexm7_mpu.c/h) that managed the MPU's register interface and access control logic, along with helper functions (cortexm7_mpu_helper.c) to try and integrate better.

Unfortunately, the custom implementation proved somewhat challenging, so a third approach was attempted, exploiting the existing ARMv7-M MPU support in QEMU in the hw/intc/ directory. This approach aimed to utilise the memory protection mechanisms that were already present in QEMU, instead of having to implement them from scratch.

Current Status and Future Development

The current implementation status highlights several areas for future development:

- **Memory Access Validation**
  - Integration with QEMU's CPU emulation layer
  - Proper handling of different access types
  - Accurate privilege level verification
- **Fault Handling**
  - Implementation of proper fault generation mechanisms
  - Integration with the ARM exception model
  - Accurate fault status reporting
- **Integration Improvements**
  - Better integration with existing board emulation
  - Enhanced debugging capabilities
  - Performance optimization

Code Structure

The MPU part was implemented on a separate branch called "MPU". The following table outlines the key files and their roles in the implementation:

File	Location	Purpose
cortexm7_mpu.c	hw/arm/	Core MPU device implementation
cortexm7_mpu.h	hw/arm/	MPU device header definitions
cortexm7_mpu_helper.c	target/arm/	CPU integration helpers
s32k3x8evb.c	hw/arm/	Board-level integration
mpu.c	MPU_project/	Software demonstration implementation
mpu.h	MPU_project/	Software demonstration headers

Conclusion

Although the current implementation does not yet fully emulate the functionality of the MPU, the development process has provided valuable insights into both the intricacies of hardware emulation and the whole range of components that enable QEMU to perfectly emulate a real board.