POLITECNICO DI TORINO

S32K3X8EVB Board
Technical Documentation

A project of Computer Architecture and Operating Systems

Authors:

Matteo Failla
Antonio Fortunato
Lorenzo di Maio
Gianmarco Michelini

Date: January 16, 2025

Contents

Chapter 1

Introduction

Integrating a new board into QEMU is a complex but essential process for emulating embedded hardware in a virtual environment. This project documents the addition of the NXP S32K3X8EVB board, based on the S32K358 System-on-Chip (SoC), into the QEMU ecosystem. The NXP S32K3X8EVB board features an ARM Cortex-M7 CPU and includes integrated peripherals such as LPUART (Low Power Universal Asynchronous Receiver-Transmitter) and PIT (Periodic Interrupt Timer).

This work follows several key phases:

1. Board Definition: Initial board configuration, including the implementation of the board initialization file, definition of system clocks, and creation of the fixed-frequency clock (SYSCLK) set to 48 MHz.

2. SoC Implementation: Creation of the SoC structure, including memory regions (such as flash memory), peripherals (including LPUART and PIT), and IRQ (Interrupt Request) lines for handling interrupts from the peripherals.

3. QEMU Integration: Modifying QEMU configuration files to include the new board and its corresponding SoC, including Kconfig entries for enabling the board in ARM architectures.

4. Peripheral Configuration: Detailed setup of LPUART modules for serial communication and PIT for time-based operations such as periodic interrupts and task scheduling.

Through this process, it has been possible to accurately simulate the NXP S32K3X8EVB board within QEMU.

Chapter 2

Before Adding the Board to QEMU

Adding a new board to QEMU involves multiple steps. This chapter outlines the detailed process we followed to integrate the *S32K3X8EVB board*, based on *S32K358 SoC*, into QEMU. Our implementation leveraged QEMU's modular architecture, reusing pre-existing CPU models (Cortex-M7) while defining the board, SoC, peripherals, and memory mappings required to emulate this embedded platform.

2.1 What is a System-on-Chip?

Referred as SoC.

A System-on-Chip integrates all major components of a computing system, such as the CPU, memory, I/O peripherals, and other essential controllers, into a single silicon chip. In embedded systems, the SoC is the heart of the hardware platform, providing the necessary computational and I/O capabilities tailored for specific tasks.

For example:

- The *CPU* (Central Processing Unit) handles the computational logic.
- Integrated *peripherals* like UARTs, timers, and memory controllers enable communication, timekeeping, and data storage.

For the project, the System on Chip (SoC) used is the *S32K358 SoC*. It comes pre-configured with a Cortex-M7 CPU and includes additional peripherals such as UART and timers.

2.2 Relationship between IRQs and Peripherals

IRQs (Interrupt Requests) are signals generated by peripherals to notify the CPU of events that require immediate attention.

Timers and UARTs are examples of peripherals that generate IRQs. A timer is able to generate IRQs to signal the end of a countdown or periodic event. For example, A Periodic Interrupt Timer (PIT) signals the CPU for periodic tasks, such as updating a clock or managing real-time scheduling. Similarly, a UART generates IRQs to signal the arrival of new data or the completion of a transmission. For example, when a UART receives data from a serial line, it triggers an IRQ to notify the CPU to read the data buffer.

Each peripheral is assigned a specific IRQ line (unique lines connecting the peripheral to the CPU or interrupt controller) and an interrupt vector (a unique identifier in the vector table pointing to the Interrupt Service Routine for that peripheral).

Chapter 3

Adding the Board to QEMU

A board in QEMU represents the hardware platform that uses a SoC. This chapter explains the steps and considerations involved in implementing a new board within the QEMU emulator. By simulating a board, developers can create and test software in a virtual environment that mimics the behavior of real hardware.

3.1 Board Initialization

The first step in adding a new board to QEMU is to define a board initialization file (i.e. `hw/arm/s32k3x8evb.c`, relative to the QEMU source root). This file specifies the clock configuration, the SoC to use and the process for loading the firmware or kernel into memory.

For detailed information on the board initialization process, refer to the following steps:

- **Clock Initialization:** Create a fixed-frequency clock (SYSCLK) using `clock_new()` and set it to 48 MHz (defined by `SYSCLK_FRQ`), based on the S32K358 hardware specifications. Connect the clock to the SoC using `qdev_connect_clock_in()`, ensuring it serves as the primary clock source for the system.

- **SoC Instantiation:** Instantiate the S32K358 SoC using `qdev_new(TYPE_S32K358_SOC)`. Add the SoC to the machine's object hierarchy with `object_property_add_child()` and associate it with the SYSCLK. Use `sysbus_realize_and_unref()` to finalize the creation and registration of the SoC within QEMU's system bus.

- **Firmware Loading:** If a firmware file is specified (`machine->kernel_filename`), it is loaded into the system's memory using `armv7m_load_kernel()`. This function places the firmware into the SoC's flash memory region (up to `FLASH_SIZE`).

- **Board Registration:** Register the board in QEMU by defining the board name (`s32k3x8evb`) and associating it with the board initialization function (`s32k3x8evb_machine_init()`) using the `DEFINE_MACHINE()` macro. This function specifies the board's description (`mc->desc`) and valid CPU types (i.e., cortex-m7).

- **Logging; Debug Information:** Use the `qemu_log()` function to provide debug information, such as the clock frequency and the successful realization of the SoC.

3.2 Board Integration with QEMU

The board is integrated into QEMU's build system by creating a *Kconfig* entry (located in `hw/arm/Kconfig`) for the board, specifying its dependencies (e.g., the SoC). This integration enables the board for ARM architectures, making it selectable during the configuration process.

The Kconfig entry for the S32K3X8EVB board and the S32K358 SoC is as follows:

```
config S32K3X8EVB
    bool
    default y
    depends on TCG && ARM
    select S32K358_SOC

config S32K358_SOC
    bool
    select ARM_V7M
```

3.3 Leveraging the Pre-Implemented CPU

QEMU provides pre-implemented CPU models that can be used for new boards with minimal customization. For the S32K3X8EVB board, the ARM Cortex-M7 CPU was utilized. The CPU type is specified during the SoC initialization, ensuring that the correct CPU model is integrated into the board. Additionally, CPU-specific properties such as the number of IRQ lines (*num-irq*), priority bits (*num-prio-bits*), and bit-banding support (e.g., enabling *bit-banding*) are configured to align with the Cortex-M7 CPU's specifications.

3.4 SoC Implementation

The SoC implementation in QEMU involves defining the SoC's memory regions, peripherals, and IRQ assignments. The SoC is encapsulated in a header file (e.g., `s32k358_soc.h`, located at `include/hw/arm/s32k358_soc.h` relative to the QEMU source root) that provides the foundational definitions and structures needed to emulate the S32K358 SoC in QEMU. It encapsulates the SoC's core features, including memory regions, peripherals, and IRQ assignments.

```
SRAM:
- Base Address: 0x20400000
- Size: 256 KB (SRAM_SIZE = 0x00040000)

Flash Memory (Defined as ROM):
- Base Address: 0x00400000
- Size: 2 MB (FLASH_SIZE = 0x00200000)
The flash memory is defined along with an alias region,
            likely intended for memory remapping purposes.

CPU: ARM Cortex-M7 core, configured with 240 IRQ lines,
                    4 priority bits and bit-banding enabled.

Peripherals:
- LPUARTs: Realizes 8 LPUART modules,
                each connected to a serial device,
                and maps MMIO (Memory-Mapped I/O) and interrupts.
            UART IRQs (LPUART0_IRQ to LPUART7_IRQ) range from 141 to 148.
- PIT: Configures the periodic interrupt timer
```

```
            and links it to the system clock and CPU.
            PIT IRQ is set to 96.
```

Inside this file, we can also find the SoC's state structure (`S32K358State`), which integrates the CPU, memory regions, peripherals, and clock, enabling accurate hardware emulation.

```
struct S32K358State {
    SysBusDevice parent_obj;

    /* ARMv7M CPU */
    ARMv7MState armv7m;

    /* Peripherals */
    S32K358LPUARTState lpuart[NUM_LPUART];
    S32K358PITState pit;

    /* Memory regions */
    MemoryRegion flash;
    MemoryRegion flash_alias;
    MemoryRegion sram;

    Clock *sysclk;
};
```

3.5 SoC Integration with QEMU

The SoC is declared as a QEMU device type (`TYPE_S32K358_SOC`), enabling its instantiation and integration with other components, such as the board (`s32k3x8evb`) or specific peripherals. This declaration follows QEMU's QOM (QEMU Object Model) framework, which standardizes object creation and property management.

Create a source file (e.g., *.../hw/arm/s32k358_soc.c*) to model the S32K358 SoC, enabling QEMU to simulate its hardware features. Initialize memory regions (e.g., Flash and SRAM) and connect peripherals (e.g., LPUARTs and PIT) to memory-mapped I/O and IRQ lines of the S32K358 SoC. This file also configures the clock and ensures that peripherals are correctly mapped into the system memory.

The SoC integrates these components into a cohesive unit, representing the real hardware architecture.

Key functions:

- `s32k358_soc_initfn()`: Initializes the SoC, including the ARMv7M CPU core, PIT, and 8 LPUART instances. It also creates the system clock (`qdev_init_clock_in`).
- `s32k358_soc_realize()`: Finalizes the SoC setup and integrates it into the QEMU emulation environment. This function validates the system clock, sets up memory regions, realizes the CPU, LPUARTs, and PIT, and connects them to the system bus.

3.6 Configuring Peripherals

The LPUARTs and PIT are essential peripherals in the S32K358 SoC, providing serial communication and timing functionality, respectively.

3.6.1 LPUART Modules Configuration

The Low Power Universal Asynchronous Receiver-Transmitter (LPUART) modules are crucial for serial communication between the emulated system and external devices.

The file `include/hw/char/s32k358_lpuart.h` (relative to the QEMU source root) models the LPUART hardware for the S32K358 SoC. This file includes the necessary register offsets, bit definitions, and the structure to model the LPUART hardware and its interaction with other QEMU components.

The registers:

```
- Baud Rate Register (LPUART_BAUD): Sets the baud rate for communication.
- Status Register (LPUART_STAT): Provides status information about the UART.
- Control Register (LPUART_CTRL): Controls the operation of the UART.
- Data Register (LPUART_DATA): Reads or writes data to/from the UART.
```

The include file also contains the state structure (`S32K358LPUARTState`), which encapsulates the LPUART's components:

```
- Memory-mapped I/O regions for communication with the LPUART.
- IRQ line for interrupt handling.
- Baud rate, status, control, and data registers to store the UART's state.
- Character backend for serial communication.
```

The `hw/char/s32k358_lpuart.c` file (relative to the QEMU source root) contains the implementation of the LPUART modules, including initialization, memory-mapped I/O, register operations, and IRQ handling for each UART module.

Main functions:

- `s32k358_lpuart_init()`: Initializes the LPUART modules, sets up the MMIO regions, and connects the UART to the serial devices.

- `s32k358_lpuart_realize()`: Finalizes the configuration of the LPUART by setting default values for its registers (baud rate, control, . . .). It also sets up the UART's receive and transmit handlers (`qemu_chr_fe_set_handlers`), enabling communication between the device and external systems.

- `s32k358_lpuart_read()` and `s32k358_lpuart_write()`: Handle read and write operations on the UART's data register, enabling interaction with the emulated LPUART hardware through memory-mapped I/O.

- `s32k358_lpuart_can_receive()` and `s32k358_lpuart_receive()`: Check if the UART can receive data (by verifying the receiver flag `RDRF`) and handle the reception of data by updating the data register, setting the receive flag (`RDRF`), and triggering an interrupt via `qemu_set_irq`, respectively. This ensures proper communication between the UART and external devices.

IRQ Handling: When the LPUART triggers an interrupt (e.g., after data is received or transmitted), an IRQ is signaled to the CPU. The `qemu_set_irq` function is used to propagate the interrupt and notify the CPU.

Throughout the module, *qemu_log* and custom debug macros are used for logging UART operations, helping track the initialization process, interrupt handling, and I/O operations. This aids in debugging the UART behavior when interacting with QEMU.

3.6.2 Timer Configuration

The Periodic Interrupt Timer (PIT) is a crucial peripheral in the S32K358 SoC, providing periodic interrupts to the CPU for time-based operations, such as periodic events or task scheduling. The `hw/timer/s32k358_pit.c` and `include/hw/timer/s32k358_pit.h` files (relative to the QEMU source root) in QEMU emulate the PIT device, managing its configuration, timer functionality, and interrupt handling. The registers:

```
- Module Control Register (PIT_MCR):
        Enables/disables the PIT and controls its operation.
- Timer Load Value Register (PIT_LDVAL0):
        Sets the timer's initial value.
- Current Timer Value Register (PIT_CVAL0):
        Stores the current timer value.
- Timer Control Register (PIT_TCTRL0):
        Controls the timer enable (TEN) and interrupt enable (TIE) settings.
- Timer Flag Register (PIT_TFLG0):
        Indicates the timer's status.
```

The include file also defines the state structure (`S32K358PITState`), which encapsulates the PIT's components:

```
- Memory-mapped I/O regions for communication with the PIT registers.
- IRQ line for interrupt handling.
- PIT Clock and timer state (initialized by the ptimer framework)
        to manage the PIT's operation.
```

Beware: *The PIT module does not have an integrated prescaler, so we need to interact with the system clock.*

The `hw/timer/s32k358_pit.c` file contains the implementation of the PIT module, including initialization, memory-mapped I/O, register operations, and IRQ handling for the timer. Main Functions:

- `s32k358_pit_init()`: Initializes the PIT module, sets up the MMIO regions, and connects the PIT to the system clock. The PIT is clocked by the `pclk` (periodic clock), which is connected via the `qdev_init_clock_in` function. This ensures the PIT operates at the correct frequency and generates interrupts at the expected intervals.

- `s32k358_pit_realize()`: Finalizes the PIT configuration. It ensures that the clock is connected, initializes the timer, and configures the interrupt behavior.

- `s32k358_pit_read()` and `s32k358_pit_write()`: Handle read and write operations on the PIT registers, allowing the CPU to control the timer's operation.

- `s32k358_pit_tick()`: The callback function that is triggered when the PIT timer reaches zero, signaling an interrupt. It checks if the interrupt flag (`TIE`) is enabled. If so, it sets the interrupt flag (`TFLG0`) and triggers the IRQ line, notifying the CPU of the interrupt.

- `s32k358_pit_clk_update()`: Updates the PIT's timer period based on the system clock (`pclk`). The function recalculates the period of the timer based on the current system clock frequency using `ptimer_set_period_from_clock`. This ensures that the timer's period is adjusted according to the clock frequency, enabling accurate timing for periodic events.

IRQ Handling: The PIT can generate interrupts when the timer expires. The `qemu_irq_pulse` function triggers the interrupt, which is then processed by the interrupt controller. The PIT's interrupt line is linked to IRQ line 96, which is handled by the NVIC (Nested Vectored Interrupt Controller) in the ARM Cortex-M7 CPU.

Debugging and logging mechanisms are provided in the module using `qemu_log` and debug macros. These logs help track the PIT's state and behavior, which is particularly useful for ensuring the timer operates correctly and the interrupts are triggered as expected.

In order to enable the PIT (Periodic Interrupt Timer) module, set the `MC_ME` module[1], specifically by configuring the register `PRTN0_COFB1_CLKEN` at `REQ44`.

Configure the Clock

1. Enable the FIRC clock as the primary source (`MC_CGM.MUX_0_CSC[SELSTAT]`):
   - The `FIRC` (Fast Internal Reference Clock) is a high-speed, internal clock that is typically used as the main clock source for many peripherals in the system. The `SELSTAT` bit in the `MC_CGM.MUX_0_CSC` register determines which clock source is selected. To use the FIRC as the primary clock, this bit needs to be set to the value corresponding to the FIRC source.
2. Set the divider (`HSE_B.CONFIG_REG_GPR[FIRC_DIV_SEL]`):
   - After selecting the FIRC as the clock source, you can set the divider to adjust the clock frequency for other system components. The `FIRC_DIV_SEL` bit field in the `HSE_B.CONFIG_REG_GPR` register allows you to select the appropriate divider value. This is important for ensuring the clock frequency is within the required range for system performance.
3. Select the appropriate mode for `CORE_CLK`, in order to configure the source clock for the `pit0` timer:
   - The `CORE_CLK` is the main clock used by the CPU and other critical components. The configuration of the `CORE_CLK` determines which clock source will drive the `pit0` (Periodic Interrupt Timer). Setting the `CORE_CLK` mode ensures the proper source is used for the timer, which is crucial for accurate timing operations in the system.
4. Select the proper divider `MC_CGM.MUX_0_DC_0[DIV]` to be configured as prescaler. Configure the prescaler in order to have a defined clock frequency:
   - A prescaler divides the clock frequency before it is used by the PIT. This allows the PIT to run at a slower rate than the system clock, which is necessary to achieve the desired interrupt frequency. The `DIV` field in the `MC_CGM.MUX_0_DC_0` register determines the prescaler value. By configuring the prescaler, you ensure that the PIT has an appropriate clock frequency for accurate periodic interrupts.
5. Link the timer's interrupt line to the ARM Cortex-M7 CPU for accurate emulation:
   - In an ARM Cortex-M7 system, interrupts are managed by the *NVIC* (Nested Vectored Interrupt Controller). You need to link the timer's interrupt output to the NVIC so that it is properly handled when triggered by the timer.

[1]System Control Module (MC_ME) provides a control interface to enable or disable clocks for various peripherals within the SoC.

- This involves configuring the interrupt source in the NVIC, ensuring that the timer triggers the interrupt correctly.

6. Add the reference to the library in the Makefile:

   - You must include the necessary library or header file for managing NVIC-related functions. In this case, the line `#include "hw/intc/armv7m_nvic.h"` should be added to the Makefile to enable the NVIC management functions.

7. Define NVIC registers (`ISER3`, `ICER3`, `ISPR3`, `ICPR0`, `IABR`, `IPR`) to handle IRQ 96:

   - In order to manage interrupts in the ARM Cortex-M7, you need to define and configure several NVIC registers.
   - These registers are used to enable, disable, and configure the priority of interrupts, including IRQ 96, which is likely the interrupt line for the PIT (Periodic Interrupt Timer).

3.7 Editing Build Files

After implementing the board and SoC, the next step is to edit the build files to include the new board configuration in the QEMU build system. This involves updating the `mason.build` file to include the board and SoC source files and the `Kconfig` file to enable the board based on the SoC selection.

- Add the board (*s32k3x8evb.c*) and the SoC (*s32k358_soc.c*) to the *hw/arm/mason.build* file (relative to the QEMU source root). This ensures that these files are included in the list of source files to be compiled when building the project for an ARM-based target.
- Enable the board-SoC dependency in the *hw/arm/Kconfig* file (relative to the QEMU source root) to ensure that the board is only available when the corresponding SoC is selected.

3.8 Building QEMU

After completing the board/SoC setup.

- In the QEMU source directory, run the following commands to configure and build QEMU for the ARM target:

  ```
  ./configure --target-list=arm-softmmu
  make -j$(nproc)
  ```

  Maybe will be asked to install some dependencies:

  ```
  sudo apt update
  sudo apt install python3-venv
  sudo apt install ninja-build
  sudo apt install pkg-config
  sudo apt install libglib2.0-dev
  sudo apt install bzip2 python3-pip libfdt-dev
  sudo apt install libpixman-1-dev zlib1g-dev
  ```

- In the Demo directory of FreeRTOS, run the following command to build the firmware:

```
Build the environment:
    make [all]

To run the demo project:
    make qemu_start

To debug:
    terminal1: make qemu_debug
    terminal2: make gdb-start
            (and then insert the command "target remote localhost:1234")
```

We verified functionality by running sample firmware and checking peripheral interactions through a serial terminal. The UART and PIT peripherals were tested to ensure proper communication and timing behavior, respectively. The successful integration of the NXP S32K3X8EVB board into QEMU demonstrates the accurate emulation of the board's hardware features within a virtual environment.