Politecnico di Torino

Master degree in Computer Engineering (Embedded Systems)

Operating Systems for Embedded Systems

Design and Implementation of a Virtual S32K3X8EVB Board in QEMU

Authors:
s338247 - Rebecca Burico
s336721 - Yuqi Li

Referents:
Prof. Stefano Di Carlo
Carpegna Alessio
Eftekhari Moghadam Vahid
Magliano Enrico

May 2025

Contents

1 Introduction

The aim of this project is to create a virtualized development environment for the **NXP S32K3X8EVB** board using **QEMU**, enabling full software development and testing without relying on physical hardware. The project combines low-level embedded system programming, hardware abstraction, and real-time operating system (RTOS) integration to build a fully functional simulation of the S32K3X8 platform.

At the heart of this work is the extension of **QEMU**, an open-source hardware emulator, to support the **S32K3X8** microcontroller family based on the **ARM Cortex-M7** architecture. This involves implementing a custom board model (`board.c` and `board.h`), defining memory regions, loading application firmware, and setting up system peripherals according to the reference manual of the real chip.

Parallel to the emulation of the board, we also ported **FreeRTOS** to the virtual board and developed a minimal embedded application that demonstrates scheduling, UART output and memory usage. The firmware is compiled with the ARM GCC toolchain and tested directly on the QEMU-emulated platform.

1.1 Prerequisites

- This project was implemented on **Ubuntu 22.04 LTS**, and it is expected to be compatible with most **Debian-based** Linux distributions.
- The instructions throughout this guide assume the use of the `apt` package manager. Users working on non-Debian systems may need to identify and install the equivalent packages using their distribution's native package manager.

1.2 Requirements

- A Linux-based operating system (e.g., Ubuntu)
- Git for version control
- QEMU for hardware emulation
- FreeRTOS for the real-time operating system
- ARM GCC toolchain for compiling the code
- Ninja build system for building QEMU

1.3 Tool Verification

You can verify if all the necessary tools are installed by running the following commands:

```
# Check for Git
git --version

# Check for QEMU
qemu-system-arm --version

# Check for ARM GCC toolchain
gcc-arm-none-eabi --version

# Check for Ninja build system
ninja --version
```

If any of these commands return an error or indicate that the tool is not installed, follow the configuration instructions detailed below.

Strictly Needed Dependencies

The following packages are essential for building and running QEMU, the ARM GCC toolchain, and the rest of the project:

- **Core Build Tools:**
  - `build-essential` (includes `gcc`, `g++`, `make`)
  - `python3` (required for QEMU build scripts)
  - `python3-pip` (for installing Python-related dependencies, if needed)
- **QEMU-Specific Build Dependencies:** `zlib1g-dev`, `libglib2.0-dev`, `libfdt-dev`, `libpixman-1-dev`, `ninja-build`, `python3-sphinx`, `python3-sphinx-rtd-theme`, `pkg-config`, `libgtk-3-dev`, `libvte-2.91-dev`, `libaio-dev`, `libbluetooth-dev`, `libbrlapi-dev`, `libbz2-dev`, `libcap-dev`, `libcap-ng-dev`, `libcurl4-gnutls-dev`, and `python3-venv`
- **ARM Toolchain:**
  - `gcc-arm-none-eabi` (the ARM GCC compiler)
  - `gdb-multiarch` (a versatile debugger for cross-architecture debugging, including ARM)

2 Setting Up the Project Environment

The following steps describe how to set up the QEMU environment for ARM-based board emulation:

1. **Prepare Project Directory and Clone Repository**

   Choose or create a new directory where you want to clone the project. Let's call this directory `your_project_path`:

```
mkdir your_project_path
cd your_project_path
```

Then, clone the repository and initialize its submodules from the development branch:

```
git clone --recurse-submodules https://baltig.polito.it/eos2024/group1.git
```

2. **Install Dependencies**

   Ensure all required build tools and libraries are installed. Run the following commands:

```
sudo apt update
sudo apt install build-essential zlib1g-dev libglib2.0-dev \
    libfdt-dev libpixman-1-dev ninja-build python3-sphinx \
    python3-sphinx-rtd-theme pkg-config libgtk-3-dev \
    libvte-2.91-dev libaio-dev libbluetooth-dev \
    libbrlapi-dev libbz2-dev libcap-dev libcap-ng-dev \
    libcurl4-gnutls-dev python3-venv \
    gcc-arm-none-eabi cmake git
```

3. **Build and Install QEMU**

   Navigate into the QEMU directory inside the cloned repository:

```
cd group1/qemu
```

   Configure the build system for the ARM softmmu target:

```
./configure --target-list=arm-softmmu
```

   Then compile and install QEMU:

```
make
sudo make install
```

The final directory structure will be organized as follows.

```
- Demo/                    # Contains demo code and FreeRTOS files
  - Firmware/              # Firmware source code and build files
  - FreeRTOS/              # FreeRTOS kernel source (or submodule)
  - Headers/               # Shared project header files
- img/                   # Contains images for documentation (e.g., README)
- materials/              # Contains related reference materials and tools
  - fmstr_uart_s32k358.zip  # FreeMASTER UART S32K358 related files
  - split_rm.zip           # Other material archive
- qemu/             # QEMU source code (includes board/peripheral modifications)
  - hw/                    # QEMU hardware models
  - build/                 # QEMU build output directory (if built here)
  - configure              # QEMU configuration script
  - ...                    # Other QEMU core source, tools, and docs
  - README.rst             # QEMU's own README
- LICENSE                 # Project's software license file
- README.md               # This project description file
```

3 Implementation Detail

3.1 Board

The specific board emulated in this project is the **S32K358EVB**, part of the NXP S32K3 series. Its implementation is guided by the official NXP reference manual to ensure structural accuracy and compatibility with the real hardware.

This emulation involves two key files:

- **board.h** (s32k3x8evb.h) – defines the data structures and constants used to describe the board.
- **board.c** (s32k3_board.c) – contains the implementation logic that initializes and configures the board inside QEMU.

3.1.1 Description of board.h (s32k3x8evb.h)

s32k3x8evb.h defines the virtual hardware layout of the emulated board, including memory sizes, device types, and structure declarations required for QEMU's internal object model.

- **Memory Definitions:**

```
#define S32K3X8EVB_FLASH_SIZE (4 * MiB)
#define S32K3X8EVB_SRAM_SIZE  (1 * MiB)
#define S32K3_FLASH_BASE 0x00400000
#define S32K3_SRAM0_BASE 0x20400000
```

- **Board State Structure:**

```
typedef struct S32K3X8EVBState {
    MachineState parent_obj;
    ARMv7MState armv7m;
    DeviceState *uart;
    DeviceState *lpspi[S32K3X8_LPSPI_COUNT];
} S32K3X8EVBState;
```

  - `MachineState parent_obj`
    Inherits from QEMU's base machine type, enabling the structure to be registered as a complete board/machine in QEMU.
  - `ARMv7MState armv7m`
    Represents the Cortex-M7 processor used in the S32K3X8 microcontroller. This is configured with properties such as vector table base address and interrupt configuration.
  - `DeviceState *uart`, `*lpspi[S32K3X8_LPSPI_COUNT]`
    Represent pointers to additional peripherals (UART and LPSPI) added to the emulated board. These follow the same device modeling pattern and are connected to the system bus during initialization.

- **Type Definitions and Runtime Checks**: in QEMU, each device or board type must be registered with a unique identifier to allow the emulator to recognize and instantiate it at runtime.

```
#define TYPE_S32K3X8EVB_MACHINE "s32k3x8evb-machine"
DECLARE_INSTANCE_CHECKER(S32K3X8EVBState, S32K3X8EVB_MACHINE,
                         TYPE_S32K3X8EVB_MACHINE)
```

3.1.2 Description of `board.c` (`s32k3_board.c`)

`s32k3_board.c` is responsible for initializing the board and mapping it into QEMU's emulated hardware environment. It performs the following tasks:

- **Memory Mapping:** ITCM, DTCM, Flash, and SRAM regions are defined according to NXP reference manual.

  In table 1 and 2 and figure 1 the base addresses and sizes of each of the memory region of the board are reported.

Memory Region	Start Address	Size
ITCM	0x00000000	64 KB
DTCM	0x20000000	124 KB
DTCM Stack	0x2001F000	4 KB

Table 1: ITCM, DTCM and DTCM stack memory layout

- **Clock Setup:** A 160 MHz clock object (`SYSCLK`) is created and linked to the ARM CPU, mimicking the microcontroller's real frequency.

Table 96. Flash block configuration

Block or address number	Block name	Start address (hex)	End address (hex)	Size	Applicability
0	Code flash memory 0	0040_0000	0047_FFFF	512 KB	S32K310, S32K311
		0040_0000	004F_FFFF	1 MB	S32K3x4, S32K342, S32K312, S32K322, S32K341
		0040_0000	005F_FFFF	2 MB	S32K3x8
1	Code flash memory 1	0048_0000	004F_FFFF	512 KB	S32K311
		0050_0000	005F_FFFF	1 MB	S32K3x4, S32K342, S32K312, S32K322
		0060_0000	007F_FFFF	2 MB	S32K3x8
2	Code flash memory 2	0060_0000	006F_FFFF	1 MB	S32K3x4
		0080_0000	009F_FFFF	2 MB	S32K3x8
3	Code flash memory 3	0070_0000	007F_FFFF	1 MB	S32K3x4
		00A0_0000	00BF_FFFF	2 MB	S32K3x8
4[1]	Data flash memory	1000_0000	1001_FFFF	128 KB	S32K3x8, S32K3x4
2[1]				128 KB	S32K342, S32K312, S32K322, S32K341
		1000_0000	1000_FFFF	64 KB	S32K311, S32K310
0[2]	UTest NVM	1B00_0000	1B00_1FFF	8 KB	S32K3x8, S32K3x4, S32K342, S32K312, S32K322, S32K341, S32K311, S32K310

1. Sector operation should be used for data flash during high voltage operation.
2. The address region number is the same as block number for all the blocks except this one. Address region is called UTest NVM address region in this case.

Figure 1: Flask block configuration from reference manual

SRAM Region	Start Address	Size
SRAM Standby	0x20400000	64 KB
SRAM 0	0x20410000	192 KB
SRAM 1	0x20440000	256 KB
SRAM 2	0x20480000	256 KB

Table 2: SRAM memory layout for the S32K358 board emulation.

- **CPU Initialization:** An ARM Cortex-M7 CPU is instantiated using QEMU's `armv7m` core. The vector table is set to start from the ITCM base (0x00000000), with 240 interrupts and 4 interrupt priority bits.
- **UART Setup:** A Low Power UART device is instantiated and mapped at address `0x4004A000`. It is linked to the host terminal using QEMU's `serial_hd(0)` character backend.
- **LPSPI Setup:** Six SPI peripherals are instantiated, each mapped to its specific base address as described in the reference manual and connected to the corresponding interrupt line.
- **Firmware Loading:** The ELF binary (compiled FreeRTOS application) is loaded at `0x00400000` in Flash memory using `armv7m_load_kernel()`.
- **Board Registration:** Registers the board in QEMU's machine list with a description, name, and initialization function:

```
static const TypeInfo s32k3x8evb_type = {
    .name = MACHINE_TYPE_NAME("s32k3x8evb"),
    .parent = TYPE_MACHINE,
    .instance_size = sizeof(S32K3X8EVBState),
    .class_init = s32k3x8evb_class_init,
};
```

3.2 LPUART

This section describes LPUART emulation. LPUART stands for Low Power Universal Asynchronous Receiver/Transmitter. This emulation is for the NXP S32K3X8 series microcontroller. It operates within the QEMU environment. The emulation copies a hardware LPUART device's behavior. Guest software can then interact with the emulated device. This interaction mimics interaction with real hardware. This process includes several features. It involves register-level I/O. It also includes interrupt generation. Finally, it covers data transmission and reception. A character backend handles this data flow.

Two main files contain the LPUART emulation:

- **s32k3x8_uart.h**: This file defines data structures. It also defines register offsets and bit definitions. Furthermore, it includes type information for the emulated LPUART device.
- **s32k3x8_uart.c**: This file implements important logic. This logic handles register access and FIFO management. It also manages interrupt handling and character device interaction. Additionally, it covers QEMU device model registration.

3.2.1 The data flow of Lpuart

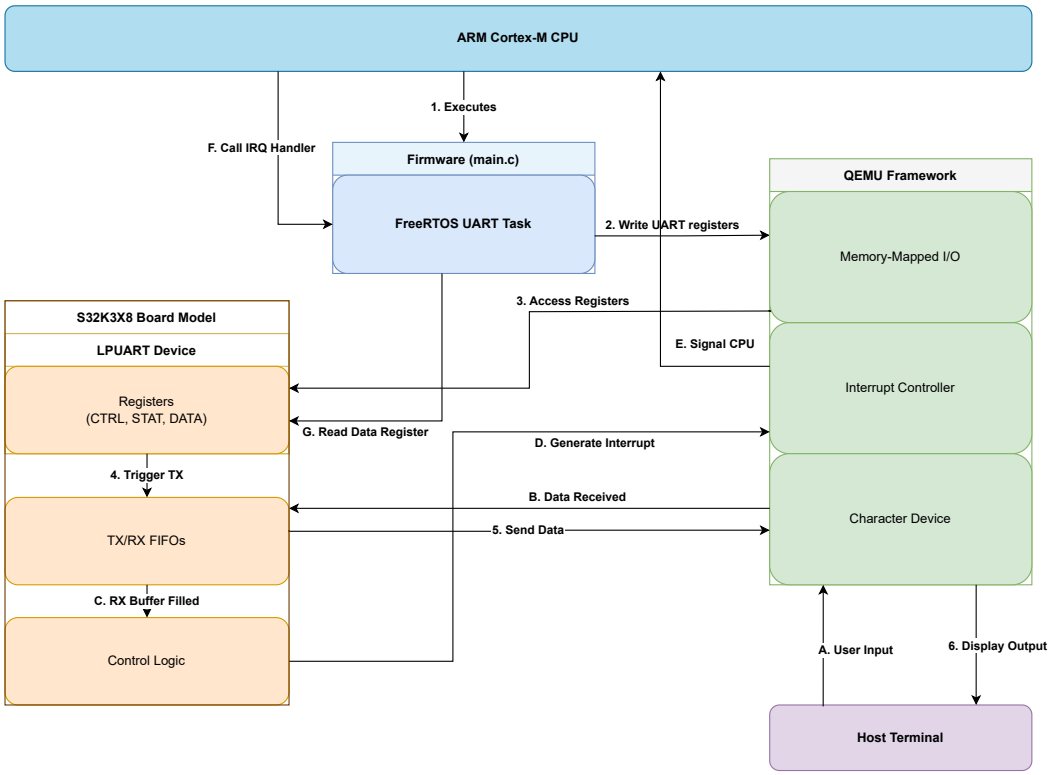

Figure 2: LPUART system integration

Output data stream:

1. The CPU executes FreeRTOS tasks.
2. These tasks write data to UART registers.
3. An MMIO interface accesses these registers.
4. This access triggers a transmission (TX). Data goes into the send FIFO.

5. The FIFO sends data using character devices.
6. The character device displays data on the terminal.

Input data stream:

1. A user enters data at the terminal.
2. A character device receives this data.
3. The data is then placed into the RX FIFO buffer.
4. Control logic detects the received data. This detection generates an interrupt.
5. The interrupt controller sends an interrupt signal to the CPU.
6. The CPU executes an interrupt handler.
7. A task reads the UART data register to retrieve the data.

3.2.2 Description of s32k3x8_uart.h

The header file s32k3x8_uart.h prepares for LPUART emulation. It defines the memory-mapped interface. It also defines the internal state representation.

- **Register Offsets and Bit Definitions:** This file defines macros. These macros represent all LPUART register offsets. Examples include `VERID_OFFSET`, CTRL_OFFSET, and `DATA_OFFSET`. Macros also define bits within these registers. Examples are `CTRL_TE` and `STAT_RDRF`. These macros allow C code to access registers and fields using names.
- **LPUART State Structure (`S32K3X8LPUARTState`):** This structure is a core data structure. It represents the state of an emulated LPUART instance.

```
typedef struct S32K3X8LPUARTState {
    SysBusDevice parent_obj;
    MemoryRegion iomem;
    CharBackend chr;
    qemu_irq irq;      /* Interrupt Request Line */

    uint32_t instance_ID;   /* LPUART Instance ID */
    uint32_t base_addr;     /* Base Address */

    /* FIFO Structure Definition */
    struct {
        uint8_t *data;
        uint32_t size;
        uint32_t read_pos;
        uint32_t write_pos;
        uint32_t count;
```

```
    } rx_fifo, tx_fifo;

    /* Register Group */
    uint32_t verid;    /* Version ID Register */
    uint32_t stat;     /* Status Register */
    uint32_t ctrl;     /* Control Register */
    // ... (and other LPUART registers like BAUD, DATA, FIFO, etc.)

    /* Internal Status Flags */
    bool tx_enabled;
    bool rx_enabled;
} S32K3X8LPUARTState;
```

Key members of this structure include the following:

- `SysBusDevice parent_obj`: This member embeds the QEMU `SysBusDevice` object. This embedding makes the LPUART a device on the system bus.
- `MemoryRegion iomem`: This member represents the LPUART's memory-mapped register block.
- `CharBackend chr`: This is QEMU's character backend interface. It connects the emulated UART to host I/O. Host I/O can be a terminal or a pipe.
- `qemu_irq irq`: This member is the interrupt line. The device asserts this line.
- `rx_fifo`, `tx_fifo`: These are structures for managing FIFOs. They manage the receive and transmit FIFOs. These structures include data buffers, sizes, and current fill states. The default size for these FIFOs is 32 bytes. This size is configurable through properties.
- Register fields (e.g., `stat`, `ctrl`, `data`): These fields hold the current values of emulated LPUART registers.
- Internal flags (`tx_enabled`, `rx_enabled`): These flags track the transmitter's and receiver's operational state.

- **Type Macros:** This section describes type macros.
  - `TYPE_S32E8_LPUART`: This macro defines the string identifier. This identifier is for this QEMU device type.
  - `S32K3X8_LPUART(obj)`: This is a macro. It performs type casting of an object to `S32K3X8LPUARTState`.

3.2.3 Description of `s32k3x8_uart.c`

The C file `s32k3x8_uart.c` holds the LPUART device emulation's functional implementation.

- **Memory-Mapped I/O (MemoryRegionOps):** The s32k3x8_lpuart_ops structure defines interaction with emulated LPUART registers. This structure gives pointers to read and write callback functions.

```
static const MemoryRegionOps s32k3x8_lpuart_ops = {
    .read = s32k3x8_lpuart_read,
    .write = s32k3x8_lpuart_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .impl = {
        .min_access_size = 1,
        .max_access_size = 4,
    },
    .valid = {
        .min_access_size = 1,
        .max_access_size = 4,
    }
};
```

  These callbacks are `s32k3x8_lpuart_read` and `s32k3x8_lpuart_write`. The guest OS invokes them. This happens when the OS accesses the LPUART device's memory region. These callbacks often use a `switch` statement. The statement uses the register offset (`addr`). This handles access to specific registers.

- **Register Access Logic:** This describes the logic for accessing registers.
  - `s32k3x8_lpuart_read(void *opaque, hwaddr addr, unsigned size)`: This function reads a value from a specified register offset. For instance, reading from `DATA_OFFSET` tries to get a byte from the RX FIFO. Reading from `STAT_OFFSET` returns current status flags.
  - `s32k3x8_lpuart_write(void *opaque, hwaddr addr, uint64_t val, unsigned size)`: This function writes a value to a specified register. Writing to `DATA_OFFSET` adds a byte to the TX FIFO. This happens if the FIFO is enabled and has space. The function then tries to send the byte via the character backend. Writing to `CTRL_OFFSET` updates control bits. These bits include transmitter/receiver enable and interrupt enables. Writing to `STAT_OFFSET` can clear certain writable status flags.
- **FIFO Management:** Some static functions manage the RX and TX FIFOs. These functions include:
  - `fifo_reset()`: This function clears a FIFO. It also resets the FIFO's pointers and count.
  - `fifo_is_empty()`, `fifo_is_full()`: These functions check the FIFO's status.
  - `fifo_count()`: This function returns the number of bytes in a FIFO.
  - `fifo_push()`: This function adds a byte to a FIFO.

  - `fifo_pop()`: This function removes a byte from a FIFO. It then returns the byte.
  - `update_uart_status_from_fifo()`: This function updates status register flags. Examples include `STAT_TDRE`, `STAT_TC`, and `STAT_RDRF`. The update depends on the FIFOs' current state and watermark levels.

- **Interrupt Handling (`s32k3x8_lpuart_update_irq`):** This function handles interrupts. It checks the LPUART's status (s->stat) and control (`s->ctrl`) registers. This check determines if an interrupt condition exists. An example condition is RX data available with RIE enabled. Another is an empty TX buffer with TIE enabled. If a condition is met, the function asserts the device's IRQ line. It uses `qemu_set_irq(s->irq, irq_state)` for this. This function is called if a register write might affect interrupt conditions. It is also called if a FIFO state change might affect them.

- **Character Backend Integration:** LPUART emulation needs to interact with QEMU's character device framework. This interaction sends and receives data with the host system.
  - `s32k3x8_lpuart_can_receive()`: This function indicates a status. It shows if the LPUART can accept more data into its RX FIFO.
  - `s32k3x8_lpuart_receive()`: The character backend calls this function. This happens when new data arrives from the host. This data is then put into the RX FIFO.
  - `s32k3x8_lpuart_event()`: This function handles events from the character backend. These events include connection open or close.
  - These handlers are registered with the character backend frontend, `s->chr`. Registration occurs during device realization. The `qemu_chr_fe_set_handlers()` function performs this registration.

- **Device Lifecycle and QOM Registration:** The LPUART device integrates into QEMU's object model (QOM). Several standard functions achieve this integration.
  - **Properties (`s32k3x8_lpuart_properties[]`):** This array defines configurable properties for the device. These properties include the associated character device (`chardev`). They also include the LPUART instance ID (`lpuart_id`). RX/TX FIFO sizes (`rx-size`, `tx-size`) are also properties.
  - **Instance Initialization (`s32k3x8_lpuart_instance_init`):** This function is called when an LPUART object is created. It initializes the memory region (`s->iomem`) for register access. It uses `memory_region_init_io()` for this. It then maps this region as MMIO for the system bus device. The function also initializes the IRQ line using `sysbus_init_irq()`.
  - **Device Realization (`s32k3x8_lpuart_realize`):** This function is called to make the device active. It performs a device reset using `s32k3x8_lpuart_reset()`. It sets up character backend handlers. It also allocates memory for FIFO

data buffers if needed. The reset function initializes all registers. Their values are set to defaults from the hardware reference manual.

- **Class Initialization (`s32k3x8_lpuart_class_init`):** This function is called once. This happens when the LPUART device class is created. It sets the device's realize function (`dc->realize`). It also registers the device's properties.
- **Type Information and Registration:** The `s32k3x8_lpuart_info` structure defines the LPUART device type for QEMU.

```
static const TypeInfo s32k3x8_lpuart_info = {
    .name = TYPE_S32E8_LPUART,
    .parent = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(S32K3X8LPUARTState),
    .instance_init = s32k3x8_lpuart_instance_init,
    .class_init = s32k3x8_lpuart_class_init,
};
```

The `s32k3x8_lpuart_register_types` function registers the device type. It registers with QEMU's type system. This function is invoked automatically at startup. The `type_init()` macro handles this invocation. This process makes the `TYPE_S32E8_LPUART` device available. It can then be instantiated in a QEMU machine model.

3.2.4 LPUART integration into the S32K3X8EVB Board

The LPUART device model (`TYPE_S32E8_LPUART`) must be defined and registered with QEMU before it can be instantiated and integrated into a specific board model. The S32K3X8EVB board serves as an example of such integration, which occurs in the board's initialization code (typically in `s32k3_board.c`).

- **Board State Declaration:**

  The board maintains a state structure called `S32K3X8EVBState`, defined in `s32k3x8evb.h`. This structure includes a pointer to hold the LPUART device instance:

```
typedef struct S32K3X8EVBState {
MachineState parent_obj;
ARMv7MState armv7m;
DeviceState *uart; // LPUART device instance
DeviceState *lpspi[S32K3X8_LPSPI_COUNT];
} S32K3X8EVBState;
```

- **Device Instantiation and Configuration:**

Table 3: LPUART Instance Summary for S32K358EVB

Instance	Base Address	TX/RX size(bytes)
LPUART0	0x40328000	32
LPUART1	0x4032C000	32
LPUART2	0x40330000	8
LPUART3	0x40334000	8
LPUART4	0x40338000	8
LPUART5	0x4033C000	8
LPUART6	0x40340000	8
LPUART7	0x40344000	8
LPUART8	0x4048C000	8
LPUART9	0x40490000	8
LPUART10	0x40494000	8
LPUART11	0x40498000	8
LPUART12	0x4049C000	8
LPUART13	0x404A0000	8
LPUART14	0x404A4000	8
LPUART15	0x404A8000	8

The LPUART device is created and configured during the board initialization function `s32k3x8evb_init` in `s32k3_board.c`:

1. **Create LPUART device instance:**

   ```
   dev = qdev_new(TYPE_S32E8_LPUART);
   ```

2. **Configure the chardev property:** Connect the LPUART device to a QEMU serial frontend (e.g., `serial_hd(0)` for the first host serial port or stdio):

   ```
   qdev_prop_set_chr(dev, "chardev", serial_hd(0));
   ```

3. **Store device reference:** Save the pointer to the created LPUART device in the board's state structure:

   ```
   s->uart = dev;
   ```

4. **Realize the device:** Activate the LPUART device on the system bus:

   ```
   sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_local);
   ```

5. **Memory mapping:** Map the LPUART device's memory-mapped I/O region into the system's address space at the correct base address (`S32K3_UART_BASE = 0x40328000`):

```
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, S32K3_UART_BASE);
```

After completing these steps, the emulated LPUART peripheral is fully integrated into the S32K3X8EVB board model. Guest software can interact with its registers at the specified base address, and interrupts can be processed.

3.3 LPSPI

This section describes the emulation of the **LPSPI** (Low Power Serial Peripheral Interface) peripheral for the NXP S32K3X8 series microcontroller. The emulation reproduces the hardware behavior of the LPSPI module within the QEMU environment, allowing guest software to interact with SPI registers and FIFOs.

Figure 3 provides a visual representation of the LPSPI module as described in the S32K3X8 reference manual. Internally, the peripheral is structured around several key components. First, we have the configuration registers. These are used by the software to set up how the peripheral behaves — for example, whether it's in master or slave mode, what the SPI clock settings are, and how interrupts should be handled.

Then, when the CPU wants to transmit data, it writes it into the TX FIFO. This allows the CPU to send data even if the SPI bus isn't immediately ready, improving efficiency.

Once the data is in the TX FIFO, it's passed into a shift register. This is where the actual bit-by-bit transfer happens. On each clock cycle, a bit is shifted out through the SOUT pin to the external device, and at the same time, a bit is shifted in from the SIN pin into the same register. When a full frame has been transmitted and received, the received data is pushed into the RX FIFO.

The RX FIFO is the counterpart to the TX FIFO. It temporarily stores incoming data until the CPU is ready to read it.

All of this is coordinated by control logic in the LPSPI module. This logic handles the chip select signals, manages transfer completion flags, and triggers interrupts if certain conditions are met — for example, if the TX FIFO is empty or the RX FIFO is full.

In our QEMU model, we simulate this behavior with FIFO buffers, status and control registers, and a loopback mode. This loopback simply echoes the transmitted data back into the receive buffer. It's a useful way to test the functionality without needing a second device.

Two main files implement the LPSPI emulation:

- `s32k358_spi.h`: Defines register offsets, bit masks, and the `S32K3X8LPSPIState` structure.
- `s32k358_spi.c`: Implements register read/write logic, FIFO mechanics, transfer behavior, and QEMU object model integration.

3.3.1 Description of `s32k358_spi.h`

This header file provides definitions for emulating the LPSPI (Low Power Serial Peripheral Interface) module of the S32K3x8 family in QEMU. It includes memory-mapped

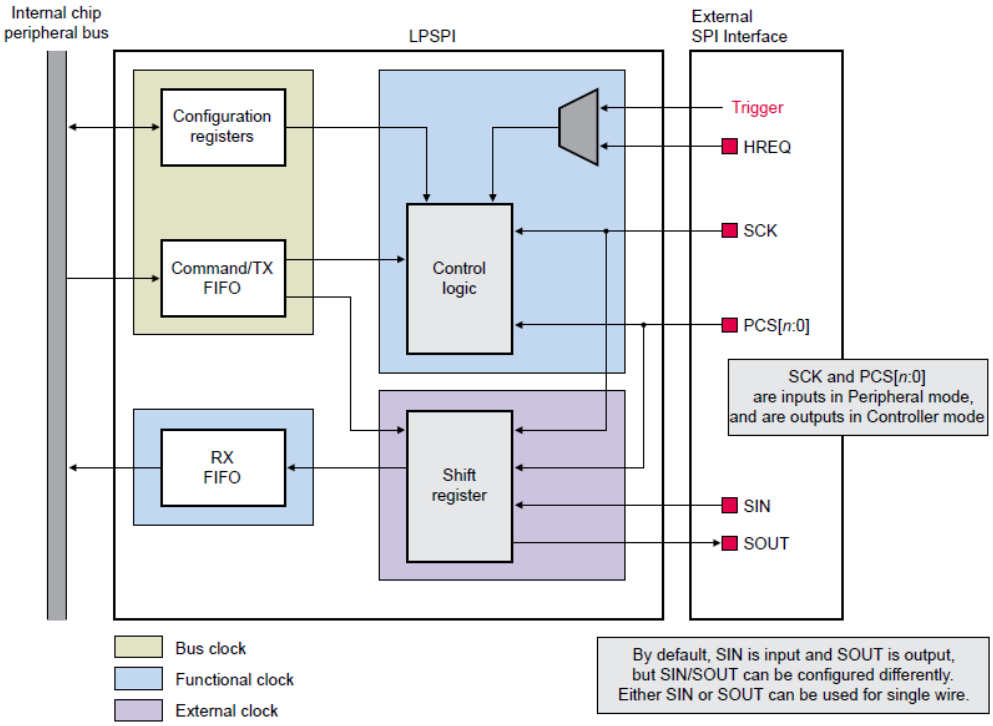

Figure 3: Flask block configuration from reference manual

interface macros, bitfield definitions, and the internal data structure representing the state of the peripheral.

Register Offsets and Bit Definitions: This file defines macros for all LPSPI register offsets. Examples include:

- `LPSPI_VERID_OFFSET`, `LPSPI_CR_OFFSET`, `LPSPI_SR_OFFSET`, etc.

It also defines macros for individual register bitfields:

- Control Register (`CR`): `LPSPI_CR_MEN`, `LPSPI_CR_RST`
- Status Register (`SR`): `LPSPI_SR_TDF`, `LPSPI_SR_RDF`, `LPSPI_SR_MBF`
- Transmit Command Register (`TCR`): `LPSPI_TCR_FRAMESZ_MASK`, `LPSPI_TCR_PRESCALE_MASK`

These macros enable symbolic access to registers and bitfields, making the C implementation more readable and maintainable.

LPSPI State Structure (`S32K3X8LPSPIState`): This structure models the state of an emulated LPSPI peripheral.

```
typedef struct S32K3X8LPSPIState {
    SysBusDevice parent_obj;
    MemoryRegion iomem;
    qemu_irq irq;
```

```
    qemu_irq cs_lines[8];
    uint32_t verid, param, cr, sr, ier, der;
    uint32_t cfgr0, cfgr1, dmr0, dmr1, ccr, ccr1;
    uint32_t fcr, fsr, tcr, tdr, rsr, rdr, rdror;
    struct {
        uint32_t data[4];
        uint32_t level, head, tail;
    } tx_fifo, rx_fifo;
    bool enabled;
    bool master_mode;
    uint32_t transfer_size;
    uint32_t current_cs;
} S32K3X8LPSPIState;
```

Key members include:

- `SysBusDevice parent_obj`: Embeds the QEMU system bus device structure.
- `MemoryRegion iomem`: Represents the memory-mapped interface block.
- `qemu_irq irq, cs_lines[8]`: Handles interrupt signaling and chip select lines.
- `Registers`: Includes all LPSPI register fields (e.g., `cr`, `sr`, `tdr`, `rdr`).
- `tx_fifo, rx_fifo`: Represent 4-word deep FIFO queues used for transmit and receive.
- `enabled, master_mode, transfer_size, current_cs`: Represent internal operational state.

Type Macros:

- `TYPE_S32K3X8_LPSPI`: Defines the QEMU type string for this device.
- `S32K3X8_LPSPI(obj)`: Casts a generic object to `S32K3X8LPSPIState`.

Function Declaration:

- `void s32k3x8_lpspi_register_types(void);`
  Registers the LPSPI type within QEMU's object model.

3.3.2 Description of `s32k358_spi.c`

The C file `s32k358_spi.c` implements the functionality for the emulated LPSPI (Low Power Serial Peripheral Interface) peripheral.

Memory-Mapped I/O (`MemoryRegionOps`): The `s32k3x8_lpspi_ops` structure defines the interface to access LPSPI registers:

```
static const MemoryRegionOps s32k3x8_lpspi_ops = {
    .read = s32k3x8_lpspi_read,
    .write = s32k3x8_lpspi_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .impl = {
        .min_access_size = 1,
        .max_access_size = 4,
    },
    .valid = {
        .min_access_size = 1,
        .max_access_size = 4,
    }
};
```

These callbacks are invoked by the guest OS when accessing memory-mapped registers.

Register Access Logic:

- `s32k3x8_lpspi_read()`: Reads register values based on offsets. Handles status flags and FIFO output (e.g., `RDR` and `RSR`).
- `s32k3x8_lpspi_write()`: Writes register values and controls behavior such as enabling the module, resetting FIFOs, and pushing data into TX FIFO.

FIFO Management: The RX and TX FIFOs are managed by static functions:

- `fifo_reset()`: Clears and reinitializes FIFO buffers.
- `fifo_is_empty()`, `fifo_is_full()`: Check FIFO state.
- `fifo_push()`, `fifo_pop()`: Insert or extract data from FIFO.

Transfer and Status Updates:

- `s32k3x8_lpspi_do_transfer()`: Simulates a loopback transfer by moving data from TX to RX FIFO. Updates completion flags.
- `s32k3x8_lpspi_update_status()`: Refreshes the status register (`SR`) and FIFO status register (`FSR`) based on FIFO conditions.

Interrupt Handling (`s32k3x8_lpspi_update_irq`): Checks interrupt enable flags (`IER`) and active conditions (`SR`) to determine whether to assert the IRQ line using `qemu_set_irq()`.

Device Lifecycle and QOM Integration:

- `s32k3x8_lpspi_reset()`: Sets default register values and clears state.
- `s32k3x8_lpspi_instance_init()`: Initializes MMIO region and IRQs.
- `s32k3x8_lpspi_realize()`: Finalizes device instantiation.
- `s32k3x8_lpspi_class_init()`: Configures class with a description and realize function.

Type Registration: The device is registered in QEMU's type system via:

```
static const TypeInfo s32k3x8_lpspi_info = {
    .name = TYPE_S32K3X8_LPSPI,
    .parent = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(S32K3X8LPSPIState),
    .instance_init = s32k3x8_lpspi_instance_init,
    .class_init = s32k3x8_lpspi_class_init,
};

void s32k3x8_lpspi_register_types(void) {
    type_register_static(&s32k3x8_lpspi_info);
}

type_init(s32k3x8_lpspi_register_types)
```

This makes the `TYPE_S32K3X8_LPSPI` device type available for use in QEMU system models.

3.3.3 LPSPI Integration into the S32K3X8EVB Board

The **LPSPI** device model (`TYPE_S32K3X8_LPSPI`) must be defined and registered with QEMU before it can be instantiated and integrated into a specific board model. In this case, the `S32K3X8EVB` development board demonstrates the integration of multiple LPSPI instances through its board initialization routine in the `s32k3x8evb_init()` function.

- **Board State Declaration**

  The board maintains a state structure called `S32K3X8EVBState`, defined in `s32k3x8evb.h`. This structure holds pointers to all connected peripherals, including an array of LPSPI device instances.

```
typedef struct S32K3X8EVBState {
MachineState parent_obj;
ARMv7MState armv7m;
DeviceState *uart;
DeviceState *lpspi[S32K3X8_LPSPI_COUNT]; // LPSPI device instance
```

```
} S32K3X8EVBState;
```

This design allows the board to instantiate and manage up to six independent LPSPI controllers as defined by the hardware reference manual (LPSPI0 to LPSPI5).

- **Device Instantiation and Configuration**

  All six LPSPI peripherals are instantiated in the `s32k3x8_init_lpspi()` function. This function is invoked inside the main `s32k3x8evb_init()` board initialization sequence.

  1. **Create LPSPI device instances**
     Each instance is created dynamically using:

     ```
     dev = qdev_new(TYPE_S32K3X8_LPSPI);
     ```

  2. **Store device reference in board state**
     Each created instance is assigned to the corresponding entry in the `lpspi[]` array:

     ```
     s->lpspi[i] = dev;
     ```

  3. **Realize the device on the system bus**
     Each device is finalized and made operational using:

     ```
     sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_abort);
     ```

  4. **Memory-map the device**
     Each LPSPI instance is assigned a specific MMIO base address as defined by the S32K3 reference manual:

     ```
     sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, S32K3_LPSPIi_BASE);
     ```

  5. **Connect interrupts**
     Each LPSPI device is connected to its corresponding NVIC interrupt line:

     ```
     sysbus_connect_irq(SYS_BUS_DEVICE(dev), 0,
         qdev_get_gpio_in(DEVICE(armv7m), S32K3_LPSPIi_IRQ));
     ```

  This setup is repeated for each of the six LPSPI instances (LPSPI0 to LPSPI5), enabling individual access, memory-mapped communication, and interrupt handling. Table 4 shows the instantiated LPSPI peripherals along with their base address and IRQ number.

Table 4: LPSPI Instance Summary for S32K3X8EVB

Instance	Base Address	IRQ Number
LPSPI0	0x40358000	69
LPSPI1	0x4035C000	70
LPSPI2	0x40360000	71
LPSPI3	0x40364000	72
LPSPI4	0x404BC000	73
LPSPI5	0x404C0000	74

4 Integration of the board in QEMU

1. **Place Source Files**

   - Save `s32k3_board.c` and `s32k3x8evb.h` in:

     `qemu/hw/arm/`

   - Save `s32k3_uart.c` and `s32k3_uart.h` in:

     `qemu/hw/char/`

   - Save `s32k3_lpspi.c` and `s32k3_lpspi.h` in:

     `qemu/hw/ssi/`

2. **Modify `meson.build` Files**

   - In `hw/arm/meson.build`:

     ```
     arm_ss.add(when: 'CONFIG_S32K3X8EVB', if_true: files('s32k3_board.c'))
     ```

   - In `hw/char/meson.build`:

     ```
     softmmu_ss.add(when: 'CONFIG_S32K3_UART', if_true: files('s32k3_uart.c'))
     ```

   - In `hw/ssi/meson.build`:

     ```
     softmmu_ss.add(when: 'CONFIG_S32K3_LPSPI', if_true: files('s32k3_lpspi.c'))
     ```

3. **Update `Kconfig` Files**

   - In `hw/arm/Kconfig`:

```
config S32K3X8_MCU
  bool
  default y
  select ARM_V7M

config S32K3X8EVB
  bool
  default y
  select S32K3X8_MCU
  select S32K3_UART
  select S32K3_LPSPI
```

- In `hw/char/Kconfig`:

```
config S32K3_UART
  bool
  default n
```

- In `hw/ssi/Kconfig`:

```
config S32K3_LPSPI
  bool
  default n
```

4. **Modify `default.mak`**

   Edit `configs/devices/arm-softmmu/default.mak` and add:

```
CONFIG_S32K3X8EVB=y
CONFIG_S32K3_UART=y
CONFIG_S32K3_LPSPI=y
```

5. **Configure QEMU**

```
./configure --target-list=arm-softmmu
```

   If you encounter errors, temporarily comment out the added lines in `default.mak`, reconfigure, and re-enable them afterward.

6. **Build QEMU**

```
make
```

7. **Install QEMU**

```
sudo make install
```

8. **Verify Board Installation**

   ```
   qemu-system-arm -M ?
   ```

   Check that `s32k3x8evb` appears in the list of supported machines.

5 FreeRTOS Porting and Demo

This section explains key aspects of FreeRTOS porting. The porting is for the emulated S32K3X8EVB platform. It also shows a simple application example. The information comes mainly from several files. These files are `FreeRTOSConfig.h`, `main.c`, `startup_ARMCM7.c`, and `system_ARMCM7.c`.

5.1 FreeRTOS Porting

FreeRTOS Configuration (`FreeRTOSConfig.h`) The FreeRTOSConfig.h file provides definitions. These are application-specific definitions. The FreeRTOS kernel needs them for the S32K3X8 target.

- **Core Target Settings:** This details core settings for the target.
  - `configPRIO_BITS`: This is set to 4. It defines the number of priority bits for tasks. These tasks are on the S32K3X8.
  - `configCPU_CLOCK_HZ`: This defines the CPU clock frequency. It is set to 160,000,000 Hz (160MHz).
  - `configTICK_RATE_HZ`: This configures the FreeRTOS tick interrupt frequency. It is set to 1000 Hz. This results in a 1ms tick period.
  - `configUSE_PREEMPTION`: This is enabled (1). It allows preemptive multitasking.
  - `configUSE_PORT_OPTIMISED_TASK_SELECTION`: This is enabled (1). It uses Cortex-M7 specific optimizations for task selection.
- **Memory Management:** This section covers memory settings.
  - `configTOTAL_HEAP_SIZE`: This specifies the total memory for the FreeRTOS heap. It is set to 128KB.
  - `configMINIMAL_STACK_SIZE`: This defines the minimum stack size for a task. The size is in words. It is set to 256 words.
  - `configSUPPORT_DYNAMIC_ALLOCATION`: This is enabled (1). It allows dynamic memory allocation for FreeRTOS objects.
- **Interrupt Priorities:** This details interrupt priority settings.

  - `configKERNEL_INTERRUPT_PRIORITY`: This sets the lowest interrupt priority. The kernel can manage interrupts at this priority. For S32K3X8, there are 4 priority bits. So, this is set to $15 \ll (8 - 4)$. This value is the numerically highest priority value. It means the lowest urgency. FreeRTOS uses this for its own interrupts, like the tick interrupt.
  - `configMAX_SYSCALL_INTERRUPT_PRIORITY`: This defines the highest interrupt priority. FreeRTOS API functions can be called safely from this priority. For S32K3X8, it is set to $5 \ll (8 - 4)$. Interrupts at or below this priority can call FreeRTOS API functions. "Below" here means a numerically higher or equal value.

- **Assertion:** `configASSERT(x)` is defined. It traps errors during development. It does this by disabling interrupts. It also enters an infinite loop if the assertion fails.

Startup and System Initialization (`startup_ARMCM7.c` and `system_ARMCM7.c`)
Proper startup is crucial for FreeRTOS. System initialization is also crucial. They ensure FreeRTOS functions correctly.

- **Vector Table (`startup_ARMCM7.c`):** The vector table contains pointers. These pointers are for exception and interrupt handlers. The table is at the beginning of ITCM. For FreeRTOS, certain handlers are essential. They are assigned in the `__Vectors` array. These handlers include:
  - SVCall Handler: `vPortSVCHandler`.
  - PendSV Handler: `xPortPendSVHandler`.
  - SysTick Handler: `xPortSysTickHandler`.

  The initial stack pointer is also set from this table.

- **Reset Handler (`startup_ARMCM7.c`):** The `Reset_Handler` is the entry point. This occurs after a system reset. Its responsibilities include the following:
  - It uses a loop to copy the `__Vectors` array to `ITCM_BASE` address. The copy size is calculated as `sizeof(__Vectors) / sizeof(uint32_t)`.
  - It updates the SCB Vector Table Offset Register (`SCB_VTOR_ADDR`) to point to the new location in ITCM. This is set to `ITCM_BASE`.
  - This relocation improves interrupt response performance by placing vectors in faster memory.

- **System Initialization (`system_ARMCM7.c`):** The `SystemInit()` function performs basic hardware setup. This function is part of the CMSIS standard.

5.2 LPUART Implementation

LPUART Driver (`LPUART.h` and related functions) The LPUART driver provides serial communication for debug output and system monitoring.

- **Hardware Definitions (`LPUART.h`):**
  - Base address: `S32K3_UART_BASE` at `0x4004A000`.
  - Register offsets: `GLOBAL_OFFSET`, `BAUD_OFFSET`, `STAT_OFFSET`, `CTRL_OFFSET`, `DATA_OFFSET`, `FIFO_OFFSET`, `WATER_OFFSET`.
  - Status bits: `STAT_TC` (transmission complete), `STAT_TDRE` (transmit data register empty), `STAT_RDRF` (receive data register full).
  - Control bits: `CTRL_TE` (transmit enable), `CTRL_RE` (receive enable), `CTRL_TIE` (transmit interrupt enable), `CTRL_RIE` (receive interrupt enable).
  - The `UART_REG(offset)` macro provides register access.
- **Initialization (`uart_init()` in `startup_ARMCM7.c`):**
  - This function sets up UART hardware and FreeRTOS integration.
  - It resets the UART module using `GLOBAL_RST`.
  - It enables the UART using `GLOBAL_ENABLE`.
  - It enables transmit and receive FIFOs.
  - It enables both transmitter and receiver.
  - It creates FreeRTOS queues for interrupt-driven communication.
  - It enables receive interrupts for future use.
- **Basic Functions (`main.c`):**
  - `uart_send_byte()`: Waits for `STAT_TDRE` flag. Then writes one byte to `DATA_OFFSET`.
  - `uart_send_string()`: Sends string character by character. Waits for `STAT_TC` when done.
  - These functions use polling method, not interrupts.
- **Interrupt Handler (`LPUART_Handler()` in `startup_ARMCM7.c`):**
  - Handles receive interrupts (`STAT_RDRF`).
  - Handles transmit interrupts (`STAT_TDRE`).
  - Uses FreeRTOS queues to pass data between interrupt and tasks.
  - Currently not used by main application tasks.

5.3 LPSPI Implementation

LPSPI Driver (`LPSPI.h` and `LPSPI.c`) The LPSPI driver provides SPI communication with complete register-level control and testing functions.

- **Hardware Definitions (`LPSPI.h`):**
  - Base addresses for LPSPI0-3. LPSPI0 at `0x40358000` is used.
  - Register offsets: `CR_OFFSET` (control), `SR_OFFSET` (status), `CFGR1_OFFSET` (configuration), `TCR_OFFSET` (transmit command), `TDR_OFFSET` (transmit data), `RDR_OFFSET` (receive data).
  - Control bits: `CR_MEN` (module enable), `CR_RST` (software reset), `CR_RTF` (reset TX FIFO), `CR_RRF` (reset RX FIFO).
  - Status bits: `SR_TDF` (transmit data flag), `SR_RDF` (receive data flag), `SR_TCF` (transfer complete flag), `SR_MBF` (module busy flag).
  - Configuration bits: `CFGR1_MASTER` (master mode).
  - The `LPSPI_REG(offset)` macro provides register access.
- **Driver Functions (`LPSPI.c`):**
  - `lpspi_init()`: Complete SPI initialization sequence. Does software reset, sets master mode, configures clock, sets 8-bit transfer mode, enables module.
  - `lpspi_transmit_byte()`: Sends one byte. Waits for TX ready, sends data, waits for transfer complete.
  - `lpspi_receive_byte()`: Receives one byte. Waits for RX data ready, reads data.
  - `lpspi_transfer_byte()`: Does both send and receive in one operation.
  - `lpspi_reset_fifos()`: Clears both TX and RX FIFOs.
  - `lpspi_get_status()`: Returns status register value.
  - `lpspi_is_busy()`: Checks if SPI module is busy.
- **Test Functions (`LPSPI.c`):**
  - `lpspi_loopback_test()`: Tests SPI with fixed data patterns. Uses array `{0xAA, 0x55, 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC}`. Checks if received data matches sent data. Reports results via UART.
  - `lpspi_status_check()`: Reads all important SPI registers. Shows register values and decoded status bits. Useful for debugging.
- **Helper Functions (`LPSPI.c`):**
  - `lpspi_delay_us()`: Simple delay function using busy wait.
  - `lpspi_wait_flag()`: Waits for status flag with timeout. Prevents infinite loops.

5.4 Demo Application

Application Tasks (`main.c`) The demo shows both LPUART and LPSPI working together in FreeRTOS environment.

- **Initialization:**
  - `main()` calls both `uart_init()` and `lpspi_init()`.
  - Both peripherals are ready before tasks start.
- **LPUART Tasks:**
  - `TxTask1`: Priority `mainTASK_PRIORITY - 2`. Sends "Hello task1" message every 3 seconds.
  - `TxTask2`: Priority `mainTASK_PRIORITY - 1`. Sends "Hello task2" message every 3 seconds.
  - `UartStatusTask`: Priority `mainTASK_PRIORITY - 3`. Checks UART registers every 10 seconds. Shows `STAT_OFFSET` and `FIFO_OFFSET` values. Reports status of `TDRE`, `TC`, `RDRF` bits.
- **LPSPI Task:**
  - `SimpleSpiTestTask`: Priority `mainTASK_PRIORITY - 4`. Runs every 10 seconds.
  - First reads SPI registers (`VERID`, `PARAM`, `SR`) for basic check.
  - Calls `lpspi_status_check()` for detailed register dump.
  - Calls `lpspi_loopback_test()` for pattern testing.
  - Does extra data pattern test with values `{0x00, 0xFF, 0xAA, 0x55}`.
  - Does counter test using task count as test data.
  - All results go to UART for monitoring.
- **Task Management:**
  - All tasks created with `xTaskCreate()`.
  - Error checking for each task creation.
  - Different priorities prevent conflicts.
  - `vTaskStartScheduler()` starts FreeRTOS.

System Features The complete system demonstrates:

- **LPUART Features:** Debug output, status monitoring, interrupt infrastructure (ready but not used).
- **LPSPI Features:** Complete SPI driver, loopback testing, register debugging, multiple test patterns.
- **FreeRTOS Integration:** Multiple tasks, different priorities, proper timing, error handling.

6 Conclusion

In this project, we have successfully designed and implemented a virtual S32K3X8EVB board model within the QEMU emulator, replicating essential hardware components based on the NXP S32K358 microcontroller. Through careful modeling of memory regions, interrupt handling, and key peripherals such as LPUART and LPSPI, we achieved an emulation environment that supports full software development without the need for physical hardware.

The integration of the LPUART module allowed robust simulation of serial communication between guest firmware and host terminal, supporting both transmission and reception with FIFO and interrupt support. Furthermore, our LPSPI implementation accurately reproduced master/slave operation, data transfer logic, and FIFO-based buffering, enabling SPI communication and loopback testing. Each of the six LPSPI instances was instantiated and correctly mapped, in line with the hardware reference manual.

We also demonstrated the viability of this virtual board by porting FreeRTOS and running a simple multitasking application, proving the operational stability of the system. The modular approach adopted for board modeling and peripheral integration ensures scalability and can support future extensions.

Overall, the work showcases the effectiveness of QEMU as a platform for embedded system emulation, particularly in educational and development contexts where access to hardware may be limited. Our virtual board is now a practical tool for real-time software prototyping and testing.