Politecnico di Torino

QEMU implementation of NXP S32K3X8EVB board

Group 10

Project Report

Master degree in Computer Engineering: Embedded Systems

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

Authors:
Francesco Mignone
Leonardo Gallina
Andrea Baraldi
Silvia Bonenti
Lorenzo Parata

Abstract

This report presents the development and implementation of a simulation environment for the NXP S32K3X8EVB board using QEMU. The primary objective of this project is to facilitate the testing and development of embedded software applications without the need for physical hardware, thereby reducing costs and increasing accessibility for developers. The report begins with an overview of the NXP S32K3X8EVB board, highlighting its key features and specifications. It then delves into the architecture and capabilities of QEMU, emphasizing its suitability for simulating embedded systems. The core of the report focuses on the step-by-step process of setting up the QEMU environment to emulate the S32K3X8EVB board, including the configuration of necessary peripherals and interfaces (UART and SPI). Furthermore, the report discusses the challenges encountered during the implementation phase and the solutions devised to overcome them.

Finally, the report concludes with an evaluation of the simulation environment's performance and its potential applications in embedded systems development. The successful implementation of this project demonstrates the feasibility of using QEMU for simulating complex embedded systems, paving the way for future advancements in this field. The report is structured to provide a comprehensive understanding of the project, making it a valuable resource for developers and researchers interested in embedded systems simulation. The structure is as follows:

- Chapter 1: Introduction - Provides background information and outlines the objectives of the project.
- Chapter 2: NXP S32K3X8EVB Board Overview - Details the specifications and features of the board.
- Chapter 3: QEMU Overview - Discusses the architecture and capabilities of QEMU.
- Chapter 4: Implementation - Describes the step-by-step process of setting up the simulation environment.
- Chapter 5: FreeRTOS Porting - Explains the process of porting FreeRTOS to the simulated environment.
- Chapter 6: Testing and Conclusion - Evaluates the performance of the simulation environment and summarizes key findings.
- References - Lists the sources and references used throughout the report.
- Appendices - Includes supplementary material such as code snippets and configuration files.

A prior knowledge of embedded systems, real-time operating systems, transmission protocols and virtualization technologies is beneficial for understanding the content of this report.

Contents

CHAPTER 1

Introduction

1.1 Project Objectives

The goal of this project is to create a simulation environment for the NXP S32K3X8EVB board using QEMU. This environment will allow developers to test and develop embedded software applications without the need for physical hardware, thereby reducing costs and increasing accessibility. The specific objectives of the project include:

- Setting up QEMU to emulate the NXP S32K3X8EVB board.
- Configuring necessary peripherals and interfaces, such as UART and SPI, to ensure accurate simulation of the board's functionality.
- Porting FreeRTOS to the simulated environment to enable real-time operating system capabilities.
- Testing the simulation environment with sample applications to validate the implementation.
- Documenting the implementation process, challenges encountered, and solutions devised to overcome them.

In particular the pheripherals that will be implemented are:

- UART (Universal Asynchronous Receiver/Transmitter)
- SPI (Serial Peripheral Interface)

1.2 Tools and Technologies

1.2.1 NXP S32K3X8EVB Board

The NXP S32K3X8EVB board is a development platform designed for automotive and industrial applications. It is based on the S32K3 series of microcontrollers, which feature an ARM Cortex-M7 core. The board includes various peripherals, such as UART, SPI, I2C, ADC, and GPIOs, allowing developers to interface with external devices and sensors. The S32K3X8EVB board is widely used in the automotive industry for applications such as motor control, body electronics, and sensor fusion.

1.2.2 QEMU

QEMU (Quick Emulator) is an open-source machine emulator and virtualizer that allows users to run operating systems and applications for one machine on a different machine. It supports a wide range of architectures, including ARM, x86, MIPS, and PowerPC, making it a versatile tool for simulating various hardware platforms. QEMU operates in two primary modes: full system emulation and user-mode emulation. In full system emulation, QEMU emulates an entire hardware platform, including the CPU, memory, and peripherals, allowing users to run complete operating systems. In user-mode emulation, QEMU can run applications compiled for one architecture on another architecture by translating system calls and library functions. QEMU's flexibility and extensibility make it an ideal choice for simulating embedded systems. It provides a rich set of features, including support for various peripherals, networking capabilities, and the ability to create custom device models. Additionally, QEMU's open-source nature allows developers to modify and extend its functionality to suit their specific needs.

1.2.3 FreeRTOS

FreeRTOS is an open-source real-time operating system (RTOS) designed for embedded systems. It provides a lightweight and efficient kernel that enables multitasking, inter-task communication, and resource management. FreeRTOS is widely used in various applications, including automotive, industrial, and consumer electronics, due to its simplicity, portability, and scalability. It offers a rich set of features, such as task scheduling, semaphores, queues, and timers, allowing developers to create complex real-time applications with ease. FreeRTOS is also highly configurable, enabling users to tailor the kernel to their specific requirements by selecting only the necessary components.

1.2.4 Development Environment

The development environment for this project includes a combination of software tools and libraries to facilitate the implementation and testing of the simulation environment. Key components of the development environment include:

- **QEMU**: The primary tool for emulating the NXP S32K3X8EVB board. The vesion used in this project is QEMU 9.0.0.
- **NXP Toolchain**: A collection of compilers and tools for building embedded applications, including the ARM GCC compiler for compiling code for the ARM Cortex-M7 architecture provided by NXP.
- **FreeRTOS Source Code**: The source code for FreeRTOS, which will be ported to the simulated environment.
- **Debugging Tools**: Tools such as GDB (GNU Debugger) for debugging and testing the simulation environment.
- **Version Control System**: Git is used for managing the source code and tracking changes throughout the development process.

1.3 Project Structure

The project is organized into several key components, each responsible for a specific aspect of the simulation environment. The main components include:

- group10
  - qemu
    - hw
      - arm
        - S32K3X8EVB.c - Board implementation file
        - S32K3X8EVB_MCU.c - MCU implementation file
      - char
        - S32K3X8EVB_UART.c - UART implementation file
      - ssi
        - S32K3X8EVB_SPI.c - SPI implementation file
    - include
      - arm
        - S32K3X8EVB.h - Board header file
        - S32K3X8EVB_MCU.h - MCU header file
      - char
        - S32K3X8EVB_UART.h - UART header file
      - ssi
        - S32K3X8EVB_SPI.h - SPI header file
  - FreeRTOS_App - FreeRTOS application
    - main.c - Main application file
    - FreeRTOSConfig.h - FreeRTOS configuration file
    - pheripherals.c - Function for the pheripherals
    - pheripherals.h - Header file for the pheripherals
    - Makefile - Build file
  - gcc-10.2.0-Earmv7GCC-eabi - NXP toolchain
  - material
    - S32K3.pdf - MCU reference manual
    - S32K3X8EVB-Q289HWUM.pdf - Board user manual
    - S32K3xx_memory_map.xls - Memory and Pheripheral map
  - Docs
    - Source
      - report - Source files for the report
      - presentation - Source files for the presentation
      - Group10_Presentation.pdf
      - Group10_Report.pdf
  - qemu_old - QEMU version 9.0.42 (not complaint to our implementation)

1.4 NXP S32K3X8EVB Board Overview

The S32K3X8EVB-Q289 is an evaluation and development board for general-purpose automotive and industrial applications. Based on the 32-bit Arm Cortex-M7 S32K3 MCU in a 289 MAPBGA package, the S32K3X8EVB-Q289 offers a multicore mode, Hardware Security Engine (HSE), Over-the-Air (OTA) support, advanced connectivity and low power. The S32K3X8EVB-Q289 board is designed to support a wide range of applications, including motor control, body electronics, and sensor fusion. It features a variety of peripherals and interfaces, such as UART, SPI, I2C, ADC, and GPIOs, allowing developers to interface with external devices and sensors.

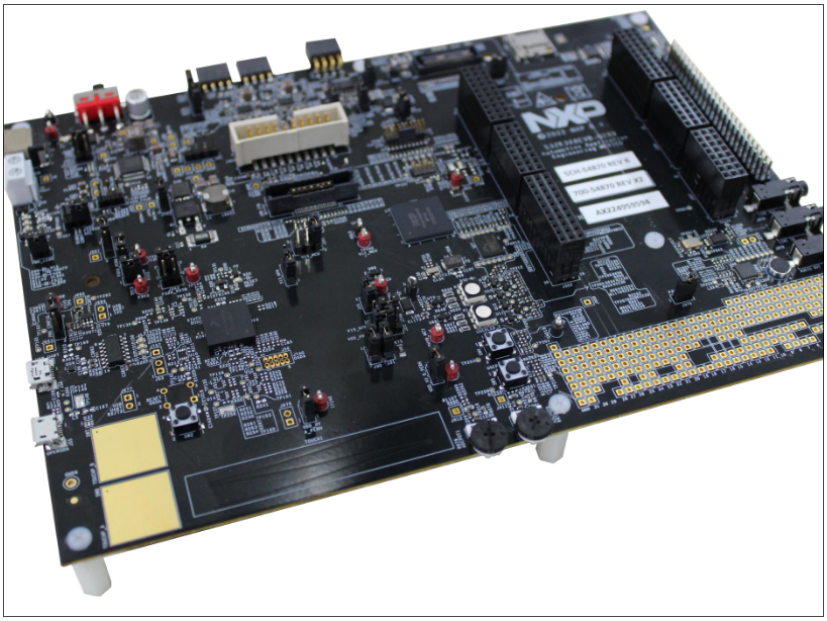

Figure 1.1: NXP S32K3X8EVB Board

1.4.1 Key Addresses

The following table lists some of the key memory-mapped addresses for the NXP S32K3X8EVB board:

- **Flash Memory**:
  - Flash0:
    - **Start Address**: 0x00400000
    - **Size**: 2 MB
  - Flash1:
    - **Start Address**: 0x00600000
    - **Size**: 2 MB
  - Flash2:
    - **Start Address**: 0x00800000
    - **Size**: 2 MB
  - Flash3:
    - **Start Address**: 0x00A00000
    - **Size**: 2 MB

  - Flash4:
    - **Start Address**: 0x10000000
    - **Size**: 128 KB

  Only the first 2 MB of Flash memory has been implemented in QEMU for testing purposes.

- **SRAM**:
  - SRAM0:
    - **Start Address**: 0x20400000
    - **Size**: 256 KB
  - SRAM1:
    - **Start Address**: 0x20440000
    - **Size**: 256 KB
  - SRAM2:
    - **Start Address**: 0x20480000
    - **Size**: 256 KB

  Only the first 256 KB of SRAM has been implemented in QEMU for testing purposes.

- **UART Base Addresses**:
  - **UART0 Address**: 0x40328000
  - **UART1 Address**: 0x4032C000
  - **UART2 Address**: 0x40330000
  - **UART3 Address**: 0x40334000
  - **UART4 Address**: 0x40338000
  - **UART5 Address**: 0x4033C000
  - **UART6 Address**: 0x40340000
  - **UART7 Address**: 0x40344000
  - **UART8 Address**: 0x4048C000
  - **UART9 Address**: 0x40490000
  - **UART10 Address**: 0x40494000
  - **UART11 Address**: 0x40498000
  - **UART12 Address**: 0x4049C000
  - **UART13 Address**: 0x404A0000
  - **UART14 Address**: 0x404A4000
  - **UART15 Address**: 0x404A8000

  All UART peripherals have been implemented anc connected to QEMU NVIC but only UART0 has been used for testing.

- **SPI Base Addresses**:
  - **SPI0 Address**: 0x40358000
  - **SPI1 Address**: 0x4035C000

- **SPI2 Address**: 0x40360000
- **SPI3 Address**: 0x40364000
- **SPI4 Address**: 0x404BC000
- **SPI5 Address**: 0x404C0000

All SPI peripherals have been implemented anc connected to QEMU NVIC but only SPI0 has been used for testing.

1.5 QEMU Overview

QEMU (Quick Emulator) is an open-source virtualization technology that allows users to run operating systems and applications for one architecture on a different architecture. It achieves this by emulating the hardware of the target architecture, enabling software designed for that architecture to run seamlessly on the host system. QEMU supports a wide range of architectures, including x86, ARM, PowerPC, and MIPS, making it a versatile tool for developers and testers.

Rather than introducing how QEMU works, this section will focus on the components of QEMU that are relevant to our project. For a comprehensive understanding of QEMU's architecture and functionality, readers are encouraged to refer to the official QEMU documentation and other detailed resources QEMU v9.0.4.

1.5.1 QEMU Object Model - QOM

The QEMU Object Model (QOM) is a framework within QEMU that provides a structured way to define and manage the various components and devices that make up a virtual machine. It allows for the creation of complex device hierarchies and relationships, enabling modularity and reusability of code. QOM is built around the concept of "objects", which represent different components of the virtual machine, such as CPUs, memory, and peripherals. Each object can have properties, methods, and events associated with it, allowing for dynamic behavior and interaction between components. QOM provides a set of macros and functions that simplify the process of defining new device types and their interactions. It also supports inheritance, allowing new device types to extend existing ones, promoting code reuse and reducing duplication. In our project, we utilize QOM to define custom devices and their interactions within the virtual machine. This allows us to create a tailored environment that meets our specific requirements while leveraging the existing infrastructure provided by QEMU.

This concept of objects is made possible via two main functions registered in the object class:

- `init()` **- Registration:** This function allows developers to register new object types with QEMU. Each type is defined by a unique name and a set of properties and methods that describe its behavior. This registration process enables QEMU to recognize and manage the new object type within the virtual machine.
- `realize()` **- Instance Creation:** Once an object type is registered, instances of that type can be created using this function. Each instance represents a specific occurrence of the object type, with its own state and configuration. This allows for multiple instances of the same object type to coexist within the virtual machine, each with its own unique characteristics.

1.5.2 Device Emulation

QEMU supports the emulation of a large number of devices from peripherals such network cards and USB devices to integrated systems on a chip (SoCs). Configuration of these is often a source of

confusion so it helps to have an understanding of some of the terms used to describes devices within QEMU:

- **Device Frontend:** A device front end is how a device is presented to the guest. The type of device presented should match the hardware that the guest operating system is expecting to see. All devices can be specified with the –device command line option. Running QEMU with the command line options –device help will list all devices it is aware of. Using the command line –device foo,help will list the additional configuration options available for that device. A front end is often paired with a back end, which describes how the host's resources are used in the emulation.
- **Device Bus:** Most devices will exist on a BUS of some sort. Depending on the machine model you choose (-M foo) a number of buses will have been automatically created. In most cases the BUS a device is attached to can be inferred, for example PCI devices are generally automatically allocated to the next free address of first PCI bus found. However in complicated configurations you can explicitly specify what bus (bus=ID) a device is attached to along with its address (addr=N).
- **Device Backend:** The back end describes how the data from the emulated device will be processed by QEMU. The configuration of the back end is usually specific to the class of device being emulated. For example serial devices will be backed by a –chardev which can redirect the data to a file or socket or some other system. Storage devices are handled by –blockdev which will specify how blocks are handled, for example being stored in a qcow2 file or accessing a raw host disk partition. Back ends can sometimes be stacked to implement features like snapshots.

1.6 QEMU Implementation

The implementation of the board has been divided into several parts for better organization and modularity:

- Board and MCU
- UART
- SPI

1.6.1 Board and MCU

The board is implemented in the file `S32K3x8.c` located in the directory `qemu/hw/arm/`. The file contains the definition of the board. The MCU is defined in the file `S32K3x8_MCU.c` located in the directory `qemu/hw/arm/`. The file contains the definition of the MCU, including its registers, peripherals and memory mapping.

This separation is done to increase modularity in future implementations. The board file can be reused for other MCUs in the S32K3x8 family by simply changing the MCU definition file. We can see that the structure of the board is rather simple as it only contains the MCU object. The MCU object, on the other hand, is more complex as it contains the CPU, memory regions and peripherals.

The actual implementation of the board is done in the function `S32K3X8EVB_init` located in the file `S32K3x8.c`. The function is responsible for initializing the board: connecting the clock to the MCU and initializing the MCU. The function is called when the board is created in QEMU. The MCU implementation is done in the function `S32K3x8_init` located in the file `S32K3x8_MCU.c`. The function is responsible for initializing the MCU: setting up the memory regions, initializing the peripherals and connecting them to the memory regions. The function is called when the MCU is created in QEMU.

```
struct S32K3x8State{
    SysBusDevice parent_obj;

    //CPU Type
    ARMv7MState cpu;

    //Board clock
    Clock *sysclk;

    // Memory Setions
    uint32_t sram0_size;    //SRAM size
    uint32_t flash0_size;   //Flash memory size

    //Memory declaration
    MemoryRegion sram0;
    MemoryRegion flash0;
    MemoryRegion flash_alias;
    MemoryRegion *board_memory;

    MemoryRegion container;

    //Peripherals
    // UART
    S32K3x8UartState uart[NXP_NUM_UARTS];

    // SPI
    S32K3x8SPIState spi[NXP_NUM_SPI];
};
```

Figure 1.2: MCU Object Structure

```
struct S32K3X8EVBMachineState {
    /* Parent machine state. */
    MachineState parent;

    S32K3x8State S32K3X8;
};
```

Figure 1.3: Board Object Structure

```
struct S32K3x8UartState {
    SysBusDevice parent_obj;

    MemoryRegion mmio;

    uint32_t usart_sr;
    uint32_t usart_dr;
    uint32_t usart_brr;
    uint32_t usart_cr1;
    uint32_t usart_cr2;
    uint32_t usart_cr3;
    uint32_t usart_gtpr;

    CharBackend chr;
    qemu_irq irq;
};
```

Figure 1.4: Board Object Structure

1.6.2 UART

The UART peripheral is implemented in the file `S32K3x8_UART.c` located in the directory `qemu/hw/arm/`. The file contains the definition of the UART peripheral, including its registers and memory mapping. The implementation is based on the reference manual of the S32K3x8 MCU. The UART peripheral is defined in the structure `S32K3x8UartState` which contains the registers of the UART peripheral and a pointer to the memory region where the peripheral is mapped. The structure is defined as follows:

The actual functions of the peripheral are implemented in the file `S32K3x8_UART.c`. The functions are responsible for reading and writing to the registers of the UART peripheral, as well as handling interrupts and data transmission. The functions are called when the CPU accesses the memory region where the peripheral is mapped.

The pheripheral is initialized in the function `S32K3x8Uart_init`. The function is responsible for initializing the UART peripheral: setting up the memory region, initializing the registers and connecting the peripheral to the memory region.

When the CPU accesses the memory region where the peripheral is mapped, the functions `S32K3x8Uart_read` and `S32K3x8Uart_write` are called. The functions are responsible for reading and writing to the registers of the UART peripheral. The functions are implemented using a switch case statement to determine which register is being accessed and perform the appropriate action.

The peripheral is connected to a character device using the QEMU CharBackend API. This allows the UART peripheral to send and receive data through a terminal or a file. The connection is established in the function `S32K3x8Uart_init` using the function `qemu_chr_open`. It is initialized and realized in the MCU init and realize functions respectively at the correct addresses.

All 16 UART peripherals have been implemented. The memory addresses and interrupt numbers are defined within the MCU file. The addresses and interrupt numbers are based on the reference manual of the S32K3x8 MCU as seen in Chapter 1.4.1.

1.6.3 SPI

The SPI peripheral is implemented in the file `S32K3x8_SPI.c` located in the directory `qemu/hw/arm/`. The file contains the definition of the SPI peripheral, including its registers and memory mapping.

```
static void S32K3x8_init(Object  *obj){
    S32K3x8State *s = S32K3x8_MCU(obj);
    //Peripheral initialization

    //UART
    int i =0;
    for ( i = 0; i < NXP_NUM_UARTS; i++) {
       object_initialize_child(obj, "uart[*]", &s->uart[i],
                               TYPE_S32K3x8_UART);
    }

    //cpu initializer
    object_initialize_child(OBJECT(s), "armv7m", &s->cpu,TYPE_ARMV7M);
    //Clock initializer
    s->sysclk = qdev_init_clock_in(DEVICE(s), "sysclk", NULL, NULL,0);
}
```

Figure 1.5: MCU UART init

```
static void S32K3x8_realize(DeviceState *dev_mcu, Error **errp){

...

    SysBusDevice *busdev;
    /* Attach all UARTs and USART controllers */
    int i = 0;
    for (i = 0; i < NXP_NUM_UARTS; i++) {

        dev = DEVICE(&(s->uart[i]));
        //declare the UART new device type
        qdev_prop_set_chr(dev, "chardev", serial_hd(i));
        if (!sysbus_realize(SYS_BUS_DEVICE(&s->uart[i]), errp)) {
            return;
        }
        busdev = SYS_BUS_DEVICE(dev);
        // Map the UART peripheral to memory
        sysbus_mmio_map(busdev, 0, usart_addr[i]);
        // initialize the IRQ table of the CPU with the correct interrupt handler
        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, usart_irq[i]));
    }
...

}
```

Figure 1.6: MCU UART relize

```
struct S32K3x8SPIState {
  /* <private> */
  SysBusDevice parent_obj;

  /* <public> */
  MemoryRegion mmio;

  uint32_t spi_cr1;
  uint32_t spi_cr2;
  uint32_t spi_sr;
  uint32_t spi_dr;
  uint32_t spi_crcpr;
  uint32_t spi_rxcrcr;
  uint32_t spi_txcrcr;
  uint32_t spi_i2scfgr;
  uint32_t spi_i2spr;

  // For testing purposes
  uint32_t test_var;

  qemu_irq irq;
  SSIBus *ssi;
};
```

Figure 1.7: SPI Object Structure

The implementation is based on the reference manual of the S32K3x8 MCU. The SPI peripheral is defined in the structure `S32K3x8SPIState` which contains the registers of the SPI peripheral and a pointer to the memory region where the peripheral is mapped. The structure is defined as follows:

The actual functions of the peripheral are implemented in the file `S32K3x8_SPI.c`. The functions are responsible for reading and writing to the registers of the SPI peripheral, as well as handling interrupts and data transmission. The functions are called when the CPU accesses the memory region where the peripheral is mapped.

The peripheral is initialized in the function `S32K3x8SPI_init`. The function is responsible for initializing the SPI peripheral: setting up the memory region, initializing the registers and connecting the peripheral to the memory region.

The SPI peripheral is connected to an SSIBus using the QEMU SSIBus API. This allows the SPI peripheral to communicate with other SPI devices connected to the bus. The connection is established in the function `S32K3x8SPI_init` using the function `ssi_bus_new`. It is initialized and realized in the MCU init and realize functions respectively at the correct addresses. When the CPU accesses the memory region where the peripheral is mapped, the functions `S32K3x8SPI_read` and `S32K3x8SPI_write` are called. The functions are responsible for reading and writing to the registers of the SPI peripheral. The functions are implemented using a switch case statement to determine which register is being accessed and perform the appropriate action.

Fot testing purposes, a variable named `test_var` has been added to the SPI structure. The variable is used to test the read and write functions of the SPI peripheral and ww will expand in Chapter 1.8.

As for the connection on the MCU it is done as the same as the UART peripheral, as shown in Figure 1.5 and Figure 1.6, but for the SPI peripheral instead.

All 6 SPI peripherals have been implemented. The memory addresses and interrupt numbers are

```
static void S32K3x8_init(Object  *obj){
    S32K3x8State *s = S32K3x8_MCU(obj);
    //Peripheral initialization

    //UART
    int i =0;
    for ( i = 0; i < NXP_NUM_UARTS; i++) {
       object_initialize_child(obj, "uart[*]", &s->uart[i],
                               TYPE_S32K3x8_UART);
    }

    //SPI
    for (i = 0; i < NXP_NUM_SPI; i++) {
        object_initialize_child(obj, "spi[*]", &s->spi[i], TYPE_S32K3x8_SPI);
    }

    //cpu initializer
    object_initialize_child(OBJECT(s), "armv7m", &s->cpu,TYPE_ARMV7M);
    //Clock initializer
    s->sysclk = qdev_init_clock_in(DEVICE(s), "sysclk", NULL, NULL,0);
}
```

Figure 1.8: MCU SPI init

```
static void S32K3x8_realize(DeviceState *dev_mcu, Error **errp){

...

    for (i = 0; i < NXP_NUM_SPI; i++) {
            dev = DEVICE(&(s->spi[i]));
            if (!sysbus_realize(SYS_BUS_DEVICE(&s->spi[i]), errp)) {
                return;
            }
            busdev = SYS_BUS_DEVICE(dev);
            // Map the UART peripheral to memory
            sysbus_mmio_map(busdev, 0, spi_addr[i]);
            // initialize the IRQ table of the CPU with the correct interrupt handler
            sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, spi_irq[i]));
    }

}
```

Figure 1.9: MCU SPI relize

defined within the MCU file. The addresses and interrupt numbers are based on the reference manual of the S32K3x8 MCU as seen in Chapter 1.4.1.

1.7 FreeRTOS Porting

In this chapter, we detail the steps taken to port FreeRTOS to our NXP board. This includes setting up the development environment, configuring the FreeRTOS kernel, and implementing necessary peripheral drivers.

1.7.1 FreeRTOS Linker Script

The linker script is a crucial component in embedded systems development, as it defines how the program's memory is organized. For our NXP board, we created a custom linker script to allocate memory for the FreeRTOS kernel, application code, and data sections. Appendix **??** provides the complete linker script used in our project.

1.7.2 FreeRTOS Configuration

The FreeRTOS configuration file, typically named `FreeRTOSConfig.h`, contains various settings that control the behavior of the FreeRTOS kernel. We modified this file to suit the requirements of our application, including task priorities, stack sizes. Appendix **??** contains the complete configuration file used in our project.

1.7.3 Peripheral Drivers

To interface with the hardware peripherals on the NXP board, we developed custom drivers for UART and SPI communication. These drivers handle the initialization, configuration, and data transfer processes for their respective peripherals. Appendix **??** contains the complete source code for the peripheral drivers.

1.7.4 Test Tasks

To validate the functionality of the FreeRTOS port and the peripheral drivers, we created several test tasks. These tasks perform simple operations such as sending and receiving data over UART and SPI. The main application code that sets up and runs these test tasks is provided in Appendix **??**.

1.8 Testing and Conclusion

In this section we discuss how to compile and run both QEMU and FreeRTOS on the NXP board.

1.8.1 QEMU Compilation and Execution

To compile QEMU with support for the NXP board, follow these steps:

1. Clone the QEMU repository:

```
$ git clone https://baltig.polito.it/eos2024/group10
```

2. Navigate to the buld directory:

```
$ cd qemu/build/
```

3. Run the configuration script:

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

4. Compile QEMU:

```
$ make -j$(nproc)
```

5. Run QEMU with the NXP board:

```
$ ./qemu/build/qemu-system-arm -machine S32K3X8EVB -monitor stdio -m 128M -nographic
```

1.8.2 FreeRTOS Compilation and Execution

To port FreeRTOS to the NXP board, the NXP toolchain has been used and it is included with the repository in the `gcc-10.2.0-Earmv7GCC-eabi` directory.

To compile FreeRTOS for the NXP board a Makefile is provided in the FreeRTOS_App directory. The Makefile is configured to use the NXP toolchain and includes paths to the FreeRTOS kernel and the peripheral drivers.

To compile and run FreeRTOS, follow these steps:

1. Clone the FreeRTOS repository:

```
$ git clone https://github.com/FreeRTOS/FreeRTOS.git --recurse-submodules
```

2. Navigate to the FreeRTOS directory:

```
$ cd FreeRTOS_App
```

3. Compile FreeRTOS:

```
$ make
```

4. Run FreeRTOS on QEMU:

```
$ make qemu_start
```

At the top of the file the path to the FreeRTOS kernel can be modified in case it is cloned in a different directory.

Debugging FreeRTOS with GDB

A Debug environment is provided to debug FreeRTOS running on the NXP board using GDB. To debug FreeRTOS running on the NXP board using GDB, follow these steps:

1. Start QEMU with GDB server:

```
$ make qemu_debug
```

2. Open another terminal and navigate to the FreeRTOS directory:

```
$ cd FreeRTOS_App
```

3. Connect to the QEMU GDB server:

```
$ make gdb_start
```

The Makefile uses `gdb` as the debugger but depending on your system you might need to use `gdb-multiarch` instead or in case of RHEL/Fedora a fork of gdb with ARM support. It is raccomended to have the `gef` plugin installed to have a better debugging experience (GEF).

1.8.3 Conclusion

In conclusion, this project successfully demonstrated the porting of FreeRTOS to an NXP board, including the development of necessary peripheral drivers and test tasks. The steps taken to set up the development environment, configure the FreeRTOS kernel, and implement the drivers were crucial in achieving a functional embedded system. The testing phase validated the functionality of the FreeRTOS port and the peripheral drivers, ensuring reliable communication over UART and SPI. This experience has provided valuable insights into embedded systems development and real-time operating systems, laying a solid foundation for future projects in this domain. The successful execution of FreeRTOS on the NXP board opens up opportunities for further enhancements, such as integrating additional peripherals, optimizing task scheduling, and exploring advanced features of FreeRTOS. Overall, this project has been a significant step towards mastering embedded systems and real-time operating systems, and it sets the stage for more complex applications in the future.

APPENDIX A

QEMU Code

A.1 S32K3X8EVB

A.1.1 Header File

```
#pragma once

#include "qemu/osdep.h"
#include "qapi/error.h"
#include "hw/qdev-properties.h"
#include "hw/boards.h"
#include "hw/arm/boot.h"
#include "sysemu/sysemu.h"
#include "exec/address-spaces.h"
#include "qom/object.h"
#include "hw/qdev-clock.h"
#include "hw/arm/S32K3X8EVB.h"
#include "hw/arm/S32K3x8_MCU.h"

#include "qemu/osdep.h"
#include "qapi/error.h"
#include "hw/boards.h"
#include "hw/qdev-properties.h"
#include "hw/qdev-clock.h"
#include "qemu/error-report.h"
#include "hw/arm/boot.h"

#define S32K3x8_FLASH0_BASE 0x00400000
#define S32K3x8_FLASH0_SIZE 2048*1024

//SRAM
#define S32K3x8_SRAM0_BASE 0x20400000
#define S32K3x8_SRAM0_SIZE 256*1024

#define TYPE_S32K3X8EVB_MACHINE MACHINE_TYPE_NAME("S32K3X8EVB")
```

```
OBJECT_DECLARE_SIMPLE_TYPE(S32K3X8EVBMachineState, S32K3X8EVB_MACHINE)

struct S32K3X8EVBMachineState {
    /* Parent machine state. */
    MachineState parent;

    S32K3x8State S32K3X8;
};
```

Source File

```
#include "qemu/osdep.h"
#include "qapi/error.h"
#include "hw/qdev-properties.h"
#include "hw/boards.h"
#include "hw/arm/boot.h"
#include "sysemu/sysemu.h"
#include "exec/address-spaces.h"
#include "qom/object.h"
#include "hw/qdev-clock.h"
#include "hw/arm/S32K3X8EVB.h"
#include "hw/arm/S32K3x8_MCU.h"

#include "qemu/osdep.h"
#include "qapi/error.h"
#include "hw/boards.h"
#include "hw/qdev-properties.h"
#include "hw/qdev-clock.h"
#include "qemu/error-report.h"
#include "hw/arm/boot.h"


static void S32K3X8EVB_init(MachineState *machine){
    DeviceState *dev;
    Clock *sysclk;
    // Clock instantiation
    /* This clock doesn't need migration because it is fixed-frequency */
    sysclk = clock_new(OBJECT(machine), "SYSCLK");
    clock_set_hz(sysclk, HCLK_FRQ);

    // MCU initalization and realization
    dev = qdev_new(TYPE_S32K3x8_MCU);
    object_property_add_child(OBJECT(machine), "soc", OBJECT(dev));
    // Clock connection to the MCU
    qdev_connect_clock_in(dev, "sysclk", sysclk);
    // Attach the MCU to the "Bus"
    sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
    armv7m_load_kernel(ARM_CPU(first_cpu), machine->kernel_filename,
                       0, S32K3x8_FLASH0_SIZE);
}
```

```
// QEMU machine class initalization
static void S32K3X8EVB_machine_class_init(ObjectClass *oc, void *data)
{
    MachineClass *mc = MACHINE_CLASS(oc);

    mc->desc = "NXP S32K3X8EVB-Q289 (Cortex-M7)";
    mc->init = S32K3X8EVB_init;
    mc->max_cpus = 1;
}

/*
  The following structure describes the machine class type:
    - It sets the machine name ("S32K3X8EVB").
    - It sets the parent class type (a QEMU machine).
    - It sets the instance size.
    - It sets the class initialization callback.
*/
static const TypeInfo S32K3X8EVB_info = {
    .name = TYPE_S32K3X8EVB_MACHINE,
    .parent = TYPE_MACHINE,
    .instance_size = sizeof(S32K3X8EVBMachineState),
    .class_init = S32K3X8EVB_machine_class_init,
};

// QEMU registration of the board
static void S32K3X8EVB_machine_init(void){
    type_register_static(&S32K3X8EVB_info);
}

type_init(S32K3X8EVB_machine_init);
```

A.2 S32K3X8EVB - MCU

A.2.1 Header File

```
#pragma once

#include <glib.h>
#include "qemu/atomic.h"
#include "glibconfig.h"
#include "hw/sysbus.h"
#include "hw/arm/armv7m.h"
#include "hw/char/S32K_uart.h"
#include "hw/clock.h"
#include "qemu/typedefs.h"
```

```
#include "qom/object.h"
#include "hw/arm/S32K3X8EVB.h"
#include "hw/or-irq.h"
#include "hw/arm/armv7m.h"
#include "include/hw/ssi/S32K3x8_spi.h"
#include "qom/object.h"

// Board clock frequency
#define HCLK_FRQ 240000000

// UART
#define NXP_NUM_UARTS 16
// SPI
#define NXP_NUM_SPI 6


#define TYPE_S32K3x8_MCU "S32K3x8_MCU"
OBJECT_DECLARE_SIMPLE_TYPE(S32K3x8State, S32K3x8_MCU)


// S32K3x8 Board class
struct S32K3x8State{
    SysBusDevice parent_obj;

    //CPU Type
    ARMv7MState cpu;

    //Board clock
    Clock *sysclk;

    // Memory Setions
    uint32_t sram0_size;    //SRAM size
    uint32_t flash0_size;   //Flash memory size

    //Memory declaration
    MemoryRegion sram0;
    MemoryRegion flash0;
    MemoryRegion flash_alias;
    MemoryRegion *board_memory;

    // We could also initialize and realize
    // this container to not use dirrectly map all
    // the peripherals or memory regions directly in the
    // QEMU memory
    MemoryRegion container;

    //Peripherals
    // UART
    S32K3x8UartState uart[NXP_NUM_UARTS];
```

```
    // SPI
    S32K3x8SPIState spi[NXP_NUM_SPI];
};
```

Source File

```
#include <stdint.h>
#include <glib.h>
#include "include/hw/arm/S32K3X8EVB.h"
#include "hw/char/S32K_uart.h"
#include "hw/arm/S32K3x8_MCU.h"
#include "hw/arm/S32K3X8EVB.h"
#include "include/hw/ssi/S32K3x8_spi.h"
#include "hw/sysbus.h"
#include "qemu/osdep.h"
#include "qapi/error.h"
#include "qemu/module.h"
#include "hw/arm/boot.h"
#include "exec/address-spaces.h"
#include "hw/qdev-properties.h"
#include "hw/qdev-clock.h"
#include "qemu/typedefs.h"
#include "sysemu/sysemu.h"

#include "qemu/osdep.h"
#include "qapi/error.h"
#include "exec/address-spaces.h"
#include "sysemu/sysemu.h"
#include "hw/qdev-clock.h"
#include "hw/misc/unimp.h"

//----------Peripherals------------
//UARTs
static const uint32_t usart_addr[NXP_NUM_UARTS] = {
                            0x40328000, //Uart 0
                            0x4032C000, //Uart 1
                            0x40330000, //Uart 2
                            0x40334000, //Uart 3
                            0x40338000, //Uart 4
                            0x4033C000, //Uart 5
                            0x40340000, //Uart 6
                            0x40344000, //Uart 7
                            0x4048C000, //Uart 8
                            0x40490000, //Uart 9
                            0x40494000, //Uart 10
                            0x40498000, //Uart 11
                            0x4049C000, //Uart 12
                            0x404A0000, //Uart 13
```

```
                            0x404A4000, //Uart 14
                            0x404A8000, //Uart 15
                            };
// IRQ
static const int usart_irq[NXP_NUM_UARTS] = {   141, //Uart 0
                                                142, //Uart 1
                                                143, //Uart 2
                                                144, //Uart 3
                                                145, //Uart 4
                                                146, //Uart 5
                                                147, //Uart 6
                                                148, //Uart 7
                                                149, //Uart 8
                                                150, //Uart 9
                                                151, //Uart 10
                                                152, //Uart 11
                                                153, //Uart 12
                                                154, //Uart 13
                                                155, //Uart 14
                                                156  //Uart 15
                                            };

//------- END UARTs ----------

//------- SPIs ----------
static const uint32_t spi_addr[NXP_NUM_SPI] = {
    0x40358000, //SPI 0
    0x4035C000, //SPI 1
    0x40360000, //SPI 2
    0x40364000, //SPI 3
    0x404BC000, //SPI 4
    0x404C0000  //SPI 5
};

static const int spi_irq[NXP_NUM_SPI] = {
    35, //SPI 0
    36, //SPI 1
    37, //SPI 2
    38, //SPI 3
    39, //SPI 4
    40  //SPI 5
};

//------- END SPIs ----------


//------- MCU initialization -----------
static void S32K3x8_init(Object  *obj){
    S32K3x8State *s = S32K3x8_MCU(obj);
```

```
    //Peripheral initialization

    //UART
    int i =0;
    for ( i = 0; i < NXP_NUM_UARTS; i++) {
       object_initialize_child(obj, "uart[*]", &s->uart[i],
                               TYPE_S32K3x8_UART);
    }

    //SPI
    for (i = 0; i < NXP_NUM_SPI; i++) {
        object_initialize_child(obj, "spi[*]", &s->spi[i], TYPE_S32K3x8_SPI);
    }

    //cpu initializer
    object_initialize_child(OBJECT(s), "armv7m", &s->cpu,TYPE_ARMV7M);
    //Clock initializer
    s->sysclk = qdev_init_clock_in(DEVICE(s), "sysclk", NULL, NULL,0);
}

//------- MCU initialization -----------

//------- MCU realization --------------
static void S32K3x8_realize(DeviceState *dev_mcu, Error **errp){
    S32K3x8State *s = S32K3x8_MCU(dev_mcu);

    // Memory
    DeviceState *dev, *armv7m;
    MemoryRegion *system_memory = get_system_memory();
    Error *err = NULL;

    /*if (!s->board_memory) {*/
    /*    error_setg(errp, "memory property was not set");*/
    /*    return;*/
    /*}*/

    /*if (!clock_has_source(s->sysclk)) {*/
    /*    error_setg(errp, "sysclk clock must not be wired up by the board
    ↪  code");*/
    /*    return;*/
    /*}*/

    //Memory Initializer
    // ------------ FLASH ---------------

    // Init memory addresses
    memory_region_init_rom(
                           &s->flash0,
                           OBJECT(dev_mcu),
```

```
                        "S32K3.flash",
                        S32K3x8_FLASH0_SIZE,
                        &err
                      );
    if (err != NULL) {
        error_propagate(errp, err);
        return;
    }

    memory_region_init_alias(
                              &s->flash_alias,
                              OBJECT(dev_mcu),
                              "S32K3.flash.alias",
                              &s->flash0, 0,
                              S32K3x8_FLASH0_SIZE
                             );
    // ------------ END FLASH ------------

    // ------------ RAM -----------------
    memory_region_add_subregion(
                              system_memory,
                              S32K3x8_FLASH0_BASE,
                              &s->flash0
                              );
    memory_region_add_subregion(
                              system_memory,
                              0,
                              &s->flash_alias);
    memory_region_init_ram(
                        &s->sram0,
                        NULL,
                        "S32K3.sram",
                        S32K3x8_SRAM0_SIZE,
                        &err
                       );

    if (err != NULL) {
        error_propagate(errp, err);
        return;
    }

    memory_region_add_subregion(
                              system_memory,
                              S32K3x8_SRAM0_BASE,
                              &s->sram0
                              );
    // ------------ END RAM --------------

    // Attach the parent clock to the MCU
```

```
clock_set_hz(s->sysclk, HCLK_FRQ);

// CPU class realization
armv7m = DEVICE(&s->cpu);

// ARM-M7 interrupt setup
qdev_prop_set_uint32(armv7m, "num-irq", 256);
/*qdev_prop_set_uint8(armv7m, "num-prio-bits", 4);*/

// CPU description (Useful for QEMU info tree)
qdev_prop_set_string(armv7m, "cpu-type", ARM_CPU_TYPE_NAME("cortex-m7"));
/*qdev_prop_set_bit(armv7m, "enable-bitband", true);*/

// Attach the clock to the CPU
qdev_connect_clock_in(armv7m, "cpuclk", s->sysclk);\

// Attach the memory regions to the CPU
object_property_set_link(
                            OBJECT(&s->cpu),
                            "memory",
                            OBJECT(system_memory),
                            &error_abort
                        );
if (!sysbus_realize(SYS_BUS_DEVICE(&s->cpu),errp)) {
   return;
}


// ----------------- UARTs Realization -----------------
SysBusDevice *busdev;
/* Attach all UARTs and USART controllers */
int i = 0;
for (i = 0; i < NXP_NUM_UARTS; i++) {

    dev = DEVICE(&(s->uart[i]));
    //declare the UART new device type
    qdev_prop_set_chr(dev, "chardev", serial_hd(i));
    if (!sysbus_realize(SYS_BUS_DEVICE(&s->uart[i]), errp)) {
        return;
    }
    busdev = SYS_BUS_DEVICE(dev);
    // Map the UART peripheral to memory
    sysbus_mmio_map(busdev, 0, usart_addr[i]);
    // initialize the IRQ table of the CPU with the correct interrupt handler
    sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, usart_irq[i]));
}

// ----------------- END UARTs Realization -----------------
```

```
    // ----------------- SPIs Realization -----------------
    for (i = 0; i < NXP_NUM_SPI; i++) {
        dev = DEVICE(&(s->spi[i]));
        if (!sysbus_realize(SYS_BUS_DEVICE(&s->spi[i]), errp)) {
            return;
        }
        busdev = SYS_BUS_DEVICE(dev);
        // Map the UART peripheral to memory
        sysbus_mmio_map(busdev, 0, spi_addr[i]);
        // initialize the IRQ table of the CPU with the correct interrupt handler
        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, spi_irq[i]));
    }
    // ----------------- END SPIs Realization -----------------
}

//------- END MCU realization --------------

// ------------- MCU properties --------------

static Property S32K3x8_properties[] = {
    DEFINE_PROP_LINK("memory", S32K3x8State, board_memory, TYPE_MEMORY_REGION,
    ↪ MemoryRegion *),
    DEFINE_PROP_UINT32("sram0-size", S32K3x8State, sram0_size, S32K3x8_SRAM0_SIZE),
    DEFINE_PROP_UINT32("flash0-size", S32K3x8State, flash0_size,
    ↪ S32K3x8_FLASH0_SIZE),
    DEFINE_PROP_END_OF_LIST(),
};

// BOARD INIT
static void S32K3x8_class_init(ObjectClass *klass, void *data){
    DeviceClass *dc = DEVICE_CLASS(klass);

    dc->realize = S32K3x8_realize;
    device_class_set_props(dc, S32K3x8_properties);
}

static const TypeInfo S32K3x8_info = {
            .name = TYPE_S32K3x8_MCU,
            .parent = TYPE_SYS_BUS_DEVICE,
            .instance_size = sizeof(S32K3x8State),
            .instance_init = S32K3x8_init,
            .class_init = S32K3x8_class_init,
};

static void S32K3x8_types(void){
    type_register_static(&S32K3x8_info);
}

type_init(S32K3x8_types);
```

A.3 S32K3X8EVB - UART

A.3.1 Header File

```
#ifndef HW_S32K3x8_USART
#define HW_S32K3x8_USART

#include "qom/object.h"
#include "hw/sysbus.h"
#include "chardev/char-fe.h"

#define USART_SR   0x00
#define USART_DR   0x04
#define USART_BRR  0x08
#define USART_CR1  0x0C
#define USART_CR2  0x10
#define USART_CR3  0x14
#define USART_GTPR 0x18

#define USART_SR_RESET (USART_SR_TXE | USART_SR_TC)

#define USART_SR_TXE  (1 << 7)
#define USART_SR_TC   (1 << 6)
#define USART_SR_RXNE (1 << 5)

#define USART_CR1_UE     (1 << 13)
#define USART_CR1_TXEIE  (1 << 7)
#define USART_CR1_TCEIE  (1 << 6)
#define USART_CR1_RXNEIE (1 << 5)
#define USART_CR1_TE     (1 << 3)
#define USART_CR1_RE     (1 << 2)

#define TYPE_S32K3x8_UART "S32K3x8_UART"
OBJECT_DECLARE_SIMPLE_TYPE(S32K3x8UartState, S32K3x8_UART)

struct S32K3x8UartState {
    SysBusDevice parent_obj;

    MemoryRegion mmio;

    uint32_t usart_sr;
    uint32_t usart_dr;
    uint32_t usart_brr;
    uint32_t usart_cr1;
    uint32_t usart_cr2;
    uint32_t usart_cr3;
    uint32_t usart_gtpr;

    CharBackend chr;
```

```
    qemu_irq irq;
};
#endif /* HW_STM32F2XX_USART_H */
```

Source File

```
#include "qemu/osdep.h"
#include "hw/char/S32K_uart.h"
#include "hw/irq.h"
#include "hw/qdev-properties.h"
#include "hw/qdev-properties-system.h"
#include "qemu/log.h"
#include "qemu/module.h"

#ifndef NXP_USART_ERR_DEBUG
#define NXP_USART_ERR_DEBUG 0
#endif

#define DB_PRINT_L(lvl, fmt, args...) do { \
    if (NXP_USART_ERR_DEBUG >= lvl) { \
        qemu_log("%s: " fmt, __func__, ## args); \
    } \
} while (0)

#define DB_PRINT(fmt, args...) DB_PRINT_L(1, fmt, ## args)

static int s32k3x8_usart_can_receive(void *opaque)
{
    S32K3x8UartState *s = opaque;

    if (!(s->usart_sr & USART_SR_RXNE)) {
        return 1;
    }

    return 0;
}

static void s32k3x8_update_irq(S32K3x8UartState *s)
{
    uint32_t mask = s->usart_sr & s->usart_cr1;

    if (mask & (USART_SR_TXE | USART_SR_TC | USART_SR_RXNE)) {
        qemu_set_irq(s->irq, 1);
    } else {
        qemu_set_irq(s->irq, 0);
    }
}

static void s32k3x8_usart_receive(void *opaque, const uint8_t *buf, int size)
{
```

```
    S32K3x8UartState *s = opaque;

    // USART not enabled - drop the chars
    if (!(s->usart_cr1 & USART_CR1_UE && s->usart_cr1 & USART_CR1_RE)) {
        DB_PRINT("Dropping the chars\n");
        return;
    }

    s->usart_dr = *buf;
    s->usart_sr |= USART_SR_RXNE;

    s32k3x8_update_irq(s);

    DB_PRINT("Receiving: %c\n", s->usart_dr);
}

static void s32k3x8_usart_reset(DeviceState *dev)
{
    S32K3x8UartState *s = S32K3x8_UART(dev);

    s->usart_sr = USART_SR_RESET;
    s->usart_dr = 0x00000000;
    s->usart_brr = 0x00000000;
    s->usart_cr1 = 0x00000000;
    s->usart_cr2 = 0x00000000;
    s->usart_cr3 = 0x00000000;
    s->usart_gtpr = 0x00000000;

    s32k3x8_update_irq(s);
}


static uint64_t s32k3x8_usart_read(void *opaque, hwaddr addr,
                                   unsigned int size)
{
    S32K3x8UartState *s = opaque;
    uint64_t retvalue;

    DB_PRINT("Read 0x%"HWADDR_PRIx"\n", addr);

    switch (addr) {
    case USART_SR:
        retvalue = s->usart_sr;
        qemu_chr_fe_accept_input(&s->chr);
        return retvalue;
    case USART_DR:
        DB_PRINT("Value: 0x%" PRIx32 ", %c\n", s->usart_dr, (char) s->usart_dr);
        retvalue = s->usart_dr & 0x3FF;
        s->usart_sr &= ~USART_SR_RXNE;
```

```
        qemu_chr_fe_accept_input(&s->chr);
        s32k3x8_update_irq(s);
        return retvalue;
    case USART_BRR:
        return s->usart_brr;
    case USART_CR1:
        return s->usart_cr1;
    case USART_CR2:
        return s->usart_cr2;
    case USART_CR3:
        return s->usart_cr3;
    case USART_GTPR:
        return s->usart_gtpr;
    default:
        qemu_log_mask(LOG_GUEST_ERROR,
                      "%s: Bad offset 0x%"HWADDR_PRIx"\n", __func__, addr);
        return 0;
    }

    return 0;
}

static void s32k3x8_usart_write(void *opaque, hwaddr addr,
                                uint64_t val64, unsigned int size)
{
    S32K3x8UartState *s = opaque;
    uint32_t value = val64;
    unsigned char ch;

    DB_PRINT("Write 0x%" PRIx32 ", 0x%"HWADDR_PRIx"\n", value, addr);

    switch (addr) {
    case USART_SR:
        if (value <= 0x3FF) {
            s->usart_sr = value | USART_SR_TXE;
        } else {
            s->usart_sr &= value;
        }
        s32k3x8_update_irq(s);
        return;
    case USART_DR:
        if (value < 0xF000) {
            ch = value;
            qemu_chr_fe_write_all(&s->chr, &ch, 1);
            s->usart_sr |= USART_SR_TC;
            s32k3x8_update_irq(s);
        }
        return;
    case USART_BRR:
```

```
        s->usart_brr = value;
        return;
    case USART_CR1:
        s->usart_cr1 = value;
        s32k3x8_update_irq(s);
        return;
    case USART_CR2:
        s->usart_cr2 = value;
        return;
    case USART_CR3:
        s->usart_cr3 = value;
        return;
    case USART_GTPR:
        s->usart_gtpr = value;
        return;
    default:
        qemu_log_mask(LOG_GUEST_ERROR,
                      "%s: Bad offset 0x%"HWADDR_PRIx"\n", __func__, addr);
    }
}


// --------- UART initialization and realization ----------
static const MemoryRegionOps s32k3x8_usart_ops = {
    .read = s32k3x8_usart_read,
    .write = s32k3x8_usart_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
};

static Property s32k3x8_usart_properties[] = {
    DEFINE_PROP_CHR("chardev", S32K3x8UartState, chr),
    DEFINE_PROP_END_OF_LIST(),
};

static void s32k3x8_usart_init(Object *obj)
{
    S32K3x8UartState *s = S32K3x8_UART(obj);

    sysbus_init_irq(SYS_BUS_DEVICE(obj), &s->irq);
    memory_region_init_io(
                            &s->mmio,
                            obj,
                            &s32k3x8_usart_ops,
                            s,
                            TYPE_S32K3x8_UART,
                            0x400
                        );
    sysbus_init_mmio(SYS_BUS_DEVICE(obj), &s->mmio);
}
```

```
static void s32k3x8_usart_realize(DeviceState *dev, Error **errp)
{
    S32K3x8UartState *s = S32K3x8_UART(dev);

    qemu_chr_fe_set_handlers(
                                &s->chr,
                                s32k3x8_usart_can_receive,
                                s32k3x8_usart_receive,
                                NULL,
                                NULL,
                                s,
                                NULL,
                                true
                            );
}

static void s32k3x8_usart_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);

    device_class_set_legacy_reset(dc, s32k3x8_usart_reset);
    device_class_set_props(dc, s32k3x8_usart_properties);
    dc->realize = s32k3x8_usart_realize;
}

static const TypeInfo s32k3x8_usart_info = {
    .name          = TYPE_S32K3x8_UART,
    .parent        = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(S32K3x8UartState),
    .instance_init = s32k3x8_usart_init,
    .class_init    = s32k3x8_usart_class_init,
};

static void s32k3x8_usart_register_types(void)
{
    type_register_static(&s32k3x8_usart_info);
}

type_init(s32k3x8_usart_register_types)
```

A.4 S32K3X8EVB - SPI

A.4.1 Header File

```
#ifndef HW_STM32F2XX_SPI_H
#define HW_STM32F2XX_SPI_H

#include "hw/ssi/ssi.h"
```

```
#include "hw/sysbus.h"
#include "qom/object.h"

// TODO: finish setting al lthe register form the datasheet
#define S32_SPI_CR1 0x00
#define S32_SPI_CR2 0x04
#define S32_SPI_SR 0x08
#define S32_SPI_DR 0x0C
#define S32_SPI_CRCPR 0x10
#define S32_SPI_RXCRCR 0x14
#define S32_SPI_TXCRCR 0x18
#define S32_SPI_I2SCFGR 0x1C
#define S32_SPI_I2SPR 0x20

#define S32_SPI_CR1_SPE (1 << 6)
#define S32_SPI_CR1_MSTR (1 << 2)

#define S32_SPI_SR_RXNE 1

#define TYPE_S32K3x8_SPI "S32K3x8_SPI"
OBJECT_DECLARE_SIMPLE_TYPE(S32K3x8SPIState, S32K3x8_SPI)

struct S32K3x8SPIState {
  /* <private> */
  SysBusDevice parent_obj;

  /* <public> */
  MemoryRegion mmio;

  uint32_t spi_cr1;
  uint32_t spi_cr2;
  uint32_t spi_sr;
  uint32_t spi_dr;
  uint32_t spi_crcpr;
  uint32_t spi_rxcrcr;
  uint32_t spi_txcrcr;
  uint32_t spi_i2scfgr;
  uint32_t spi_i2spr;

  // For testing purposes
  uint32_t test_var;

  qemu_irq irq;
  SSIBus *ssi;
};

#endif
```

Source File

```
#include "qemu/osdep.h"
#include "qemu/log.h"
#include "qemu/module.h"
#include "hw/ssi/S32K3x8_spi.h"
#include "migration/vmstate.h"

#ifndef S32_SPI_ERR_DEBUG
#define S32_SPI_ERR_DEBUG 0
#endif

#define DB_PRINT_L(lvl, fmt, args...) do { \
    if (S32_SPI_ERR_DEBUG >= lvl) { \
        qemu_log("%s: " fmt, __func__, ## args); \
    } \
} while (0)

#define DB_PRINT(fmt, args...) DB_PRINT_L(1, fmt, ## args)

static void S32Kx8_spi_reset(DeviceState *dev)
{
    S32K3x8SPIState *s = S32K3x8_SPI(dev);

    s->spi_cr1 = 0x00000000;
    s->spi_cr2 = 0x00000000;
    s->spi_sr = 0x0000000A;
    s->spi_dr = 0x0000000C;
    s->spi_crcpr = 0x00000007;
    s->spi_rxcrcr = 0x00000000;
    s->spi_txcrcr = 0x00000000;
    s->spi_i2scfgr = 0x00000000;
    s->spi_i2spr = 0x00000002;
}

// Function to handle SPI data transfer
static void S32Kx8_spi_transfer(S32K3x8SPIState *s)
{
    DB_PRINT("Data to send: 0x%x\n", s->spi_dr);

    s->spi_dr = ssi_transfer(s->ssi, s->spi_dr);
    // Clear the RXNE flag and set it again to indicate that new data is available
    s->spi_sr |= S32_SPI_SR_RXNE;

    DB_PRINT("Data received: 0x%x\n", s->spi_dr);
}

// Memory-mapped I/O read
// When reading from the data register, perform a transfer
```

```
static uint64_t S32Kx8_spi_read(void *opaque, hwaddr addr,
                                unsigned int size)
{
    S32K3x8SPIState *s = opaque;

    DB_PRINT("Address: 0x%" HWADDR_PRIx "\n", addr);

    switch (addr) {
    case S32_SPI_CR1:
        return s->spi_cr1;
    case S32_SPI_CR2:
        qemu_log_mask(LOG_UNIMP, "%s: Interrupts and DMA are not implemented\n",
                      __func__);
        return s->spi_cr2;
    case S32_SPI_SR:
        return s->spi_sr;
    case S32_SPI_DR:
        S32Kx8_spi_transfer(s);
        s->spi_sr &= ~S32_SPI_SR_RXNE;
        // s->spi_dr |= 0xFFFFFF66;
        // For testing purposes, increment the value read each time
        s->spi_dr = s->test_var + 0x01;
        return s->spi_dr;
    case S32_SPI_CRCPR:
        return s->spi_crcpr;
    case S32_SPI_RXCRCR:
        return s->spi_rxcrcr;
    case S32_SPI_TXCRCR:
        return s->spi_txcrcr;
    case S32_SPI_I2SCFGR:
        return s->spi_i2scfgr;
    case S32_SPI_I2SPR:
        return s->spi_i2spr;
    default:
        qemu_log_mask(LOG_GUEST_ERROR, "%s: Bad offset 0x%" HWADDR_PRIx "\n",
                      __func__, addr);
    }

    return 0;
}

// Memory-mapped I/O write
// Writing to the data register initiates a transfer
static void S32Kx8_spi_write(void *opaque, hwaddr addr,
                             uint64_t val64, unsigned int size)
{
    S32K3x8SPIState *s = opaque;
    uint32_t value = val64;
```

```
    DB_PRINT("Address: 0x%" HWADDR_PRIx ", Value: 0x%x\n", addr, value);

    switch (addr) {
    case S32_SPI_CR1:
        s->spi_cr1 = value;
        return;
    case S32_SPI_CR2:
        qemu_log_mask(LOG_UNIMP, "%s: " \
                      "Interrupts and DMA are not implemented\n", __func__);
        s->spi_cr2 = value;
        return;
    case S32_SPI_SR:
        return;
    case S32_SPI_DR:
        s->spi_dr = value;
        s->test_var = value; // Reset test_var on write
        S32Kx8_spi_transfer(s);
        return;
    case S32_SPI_CRCPR:
        s->spi_crcpr = value;
        return;
    case S32_SPI_RXCRCR:
        qemu_log_mask(LOG_GUEST_ERROR, "%s: Read only register: " \
                      "0x%" HWADDR_PRIx "\n", __func__, addr);
        return;
    case S32_SPI_TXCRCR:
        qemu_log_mask(LOG_GUEST_ERROR, "%s: Read only register: " \
                      "0x%" HWADDR_PRIx "\n", __func__, addr);
        return;
    case S32_SPI_I2SCFGR:
        s->spi_i2scfgr = value;
        return;
    case S32_SPI_I2SPR:
        s->spi_i2spr = value;
        return;
    default:
        qemu_log_mask(LOG_GUEST_ERROR,
                      "%s: Bad offset 0x%" HWADDR_PRIx "\n", __func__, addr);
    }
}

static const MemoryRegionOps S32Kx8_spi_ops = {
    .read = S32Kx8_spi_read,
    .write = S32Kx8_spi_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
};

static const VMStateDescription vmstate_S32Kx8_spi = {
    .name = TYPE_S32K3x8_SPI,
```

```
    .version_id = 1,
    .minimum_version_id = 1,
    .fields = (const VMStateField[]) {
        VMSTATE_UINT32(spi_cr1, S32K3x8SPIState),
        VMSTATE_UINT32(spi_cr2, S32K3x8SPIState),
        VMSTATE_UINT32(spi_sr, S32K3x8SPIState),
        VMSTATE_UINT32(spi_dr, S32K3x8SPIState),
        VMSTATE_UINT32(spi_crcpr, S32K3x8SPIState),
        VMSTATE_UINT32(spi_rxcrcr, S32K3x8SPIState),
        VMSTATE_UINT32(spi_txcrcr, S32K3x8SPIState),
        VMSTATE_UINT32(spi_i2scfgr, S32K3x8SPIState),
        VMSTATE_UINT32(spi_i2spr, S32K3x8SPIState),
        VMSTATE_END_OF_LIST()
    }
};

static void S32Kx8_spi_init(Object *obj)
{
    S32K3x8SPIState *s = S32K3x8_SPI(obj);
    DeviceState *dev = DEVICE(obj);

    memory_region_init_io(&s->mmio, obj, &S32Kx8_spi_ops, s,
                          TYPE_S32K3x8_SPI, 0x400);
    sysbus_init_mmio(SYS_BUS_DEVICE(obj), &s->mmio);

    sysbus_init_irq(SYS_BUS_DEVICE(obj), &s->irq);

    s->ssi = ssi_create_bus(dev, "ssi");
}

static void S32Kx8_spi_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);

    device_class_set_legacy_reset(dc, S32Kx8_spi_reset);
    dc->vmsd = &vmstate_S32Kx8_spi;
}

static const TypeInfo S32Kx8_spi_info = {
    .name          = TYPE_S32K3x8_SPI,
    .parent        = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(S32K3x8SPIState),
    .instance_init = S32Kx8_spi_init,
    .class_init    = S32Kx8_spi_class_init,
};

static void S32Kx8_spi_register_types(void)
{
    type_register_static(&S32Kx8_spi_info);
```

```
}

type_init(S32Kx8_spi_register_types)
```

APPENDIX B

FreeRTOS Code

B.1 Linker Script

```
/*
 * FreeRTOS V202212.00
 * Copyright (C) 2020 Amazon.com, Inc. or its affiliates.  All Rights Reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy of
 * this software and associated documentation files (the "Software"), to deal in
 * the Software without restriction, including without limitation the rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
 * the Software, and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 * https://www.FreeRTOS.org
 * https://github.com/FreeRTOS
 *
 */

MEMORY
{
    FLASH (xr) : ORIGIN = 0x00400000, LENGTH = 2048K /* to 0x00003FFF =
    ↪  0x007FFFFF*/
    RAM (rw)  : ORIGIN = 0x20400000, LENGTH = 256K /* to 0x21FFFFFF = 0xFFFFFF */
}
```

```
ENTRY(Reset_Handler)

_Min_Heap_Size = 0x8 ;         /* Not used as building heap_4.c */
_Min_Stack_Size = 0x400 ;        /* Required amount of stack.  Used by main(), then
↪  re-used as the interrupt stack after the kernel starts. */
_estack = ORIGIN(RAM) + LENGTH(RAM);


SECTIONS
{
    .isr_vector :
    {
        __vector_table = .;
        KEEP(*(.isr_vector))
        . = ALIGN(4);
    } > FLASH

    .text :
    {
        *(.text)
        *(.rodata*)
        *(.constdata*)
        _etext = .;
        _sidata = .;
    } > FLASH

    .data :
    {
        . = ALIGN(8);
        _data = .;
        _sdata = .;
        *(vtable)
        *(.data)
        _edata = .;
    } > RAM

    .bss :
    {
        . = ALIGN(8);
        _bss = .;
        _sbss = .;
        *(.bss)
        _ebss = .;
    } > RAM

    .heap :
    {
        . = ALIGN(8);
        PROVIDE ( end = . );
```

```
        PROVIDE ( _end = . );
        _heap_bottom = .;
        . = . + _Min_Heap_Size;
        _heap_top = .;
        . = . + _Min_Stack_Size;
        . = ALIGN(8);
    } >RAM

    /* Set stack top to end of RAM, and stack limit move down by
     * size of stack_dummy section */
    __StackTop = ORIGIN(RAM) + LENGTH(RAM);
    __StackLimit = __StackTop - _Min_Stack_Size;
    PROVIDE(__stack = __StackTop);

  /* Check if data + heap + stack exceeds RAM limit */
  ASSERT(__StackLimit >= _heap_top, "region RAM overflowed with stack")

}
```

B.2 Peripheral Drivers

B.2.1 Header File

```
#ifndef __PRINTF__
#define __PRINTF__

#include "FreeRTOS.h"
#include <stdint.h>

//UART
#define UART0_ADDRESS (0x40328000UL)
#define UART0_DATA UART0_ADDRESS + 4UL
#define UART0_STATE UART0_ADDRESS + 0UL
#define UART0_CTRL UART0_ADDRESS + 0xCUL
#define UART0_BAUDDIV UART0_ADDRESS + 8UL

void UART_init(void);
void uart_printf(const char *fmt, ...);


// SPI
// LPSPI0 base pointer
#define LPSPI0_BASE (0x40358000UL)

// #define LPSPI0_CTRL (*((volatile uint32_t *)(LPSPI0_BASE + 0x0UL)))

// LPSPI0 registers
#define LPSPI0_CTRL (LPSPI0_BASE + 0x0UL)
#define LPSPI0_CTRL2 (LPSPI0_BASE + 0x04UL)
```

```
#define LPSPI0_SR (LPSPI0_BASE + 0x08UL)
#define LPSPI0_DR (LPSPI0_BASE + 0xCUL)
#define LPSPI0_CRCPR (LPSPI0_BASE + 0x10UL)
#define LPSPI0_RXCRCR (LPSPI0_BASE + 0x14UL)
#define LPSPI0_TXCRCR (LPSPI0_BASE + 0x18UL)
#define LPSPI0_I2SCFGR (LPSPI0_BASE + 0x1CUL)
#define LPSPI0_I2SPR (LPSPI0_BASE + 0x20UL)

void SPI_init(void);
void SPI_status(void);
void SPI_write(uint8_t);
void SPI_get(uint8_t *);
#endif
```

Source File

```
#include "peripherals.h"
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>

// UART
// UART Initialization
void UART_init(void) {
  // Set baud rate
  *((uint32_t *)(UART0_BAUDDIV)) = 16;
  // Enable transmitter and receiver
  *((uint32_t *)(UART0_CTRL)) = 1;
}

static void uart_send_char(char c)
{
  *((uint32_t *)(UART0_DATA)) = (unsigned int)c;
}

static void uart_send_string(const char *s)
{
    while (*s) {
        uart_send_char(*s++);
    }
}

void uart_printf(const char *fmt, ...)
{
    char buffer[128];
    char *p = buffer;
    va_list args;
    va_start(args, fmt);

    while (*fmt) {
```

```
    if (*fmt == '%') {
        fmt++;
        switch (*fmt) {
            case 'c': {
                char c = (char)va_arg(args, int);
                *p++ = c;
                break;
            }
            case 's': {
                char *str = va_arg(args, char*);
                while (*str) *p++ = *str++;
                break;
            }
            case 'd': {
                int val = va_arg(args, int);
                char num[16];
                bool neg = false;
                if (val < 0) { neg = true; val = -val; }
                int i = 0;
                do { num[i++] = (val % 10) + '0'; val /= 10; } while (val > 0);
                if (neg) *p++ = '-';
                while (i--) *p++ = num[i];
                break;
            }
            case 'x':
            case 'X': {
                unsigned val = va_arg(args, unsigned);
                char num[16];
                int i = 0;
                do {
                    int digit = val % 16;
                    num[i++] = (digit < 10) ? '0' + digit : ((*fmt == 'x') ? 'a'
                    ↪  : 'A') + (digit - 10);
                    val /= 16;
                } while (val > 0);
                *p++ = '0';
                *p++ = 'x';
                while (i--) *p++ = num[i];
                break;
            }
            default:
                *p++ = '%';
                *p++ = *fmt;
        }
    } else {
        *p++ = *fmt;
    }
    fmt++;
}
```

```
    *p = '\0';

    va_end(args);
    uart_send_string(buffer);
}

// SPI

// SPI Initialization
void SPI_init(void) {
  *((uint32_t *)(LPSPI0_CTRL)) = 1;
  *((uint32_t *)(LPSPI0_CRCPR)) = 7;
  *((uint32_t *)(LPSPI0_CTRL2)) = 0;        // Default configuration
  *((uint32_t *)(LPSPI0_I2SCFGR)) = 0;      // Disable I2S
  *((uint32_t *)(LPSPI0_I2SPR)) = 0;        // Default configuration
  *((uint32_t *)(LPSPI0_CTRL)) |= (1 << 6); // Enable SPI
  *((uint32_t *)(LPSPI0_CTRL)) |= (1 << 2); // Master mode
}

// SPI all register status read
void SPI_status(void) {
  uart_printf("---------- SPI Registers Status ----------\n");
  uart_printf("LPSPI0_CTRL: %X\n", *((uint32_t *)(LPSPI0_CTRL)));
  uart_printf("LPSPI0_CTRL2: %x\n", *((uint32_t *)(LPSPI0_CTRL2)));
  uart_printf("LPSPI0_SR: %X\n", *((uint32_t *)(LPSPI0_SR)));
  uart_printf("LPSPI0_DR: %X\n", *((uint32_t *)(LPSPI0_DR)));
  uart_printf("LPSPI0_CRCPR: %X\n", *((uint32_t *)(LPSPI0_CRCPR)));
  uart_printf("LPSPI0_RXCRCR: %X\n", *((uint32_t *)(LPSPI0_RXCRCR)));
  uart_printf("LPSPI0_TXCRCR: %X\n", *((uint32_t *)(LPSPI0_TXCRCR)));
  uart_printf("LPSPI0_I2SCFGR: %X\n", *((uint32_t *)(LPSPI0_I2SCFGR)));
  uart_printf("LPSPI0_I2SPR: %X\n", *((uint32_t *)(LPSPI0_I2SPR)));
  uart_printf("-----------------------------------------\n");
}

// SPI Write
void SPI_write(uint8_t s) {
  while (!(*((uint32_t *)(LPSPI0_SR)) & 0x2))
    ; // Wait until TX ready

    *((uint32_t *)(LPSPI0_DR)) = s;
}

// SPI Read
void SPI_get(uint8_t *s) {
  while (!(*((uint32_t *)(LPSPI0_SR)) & 0x1))
    ; // Wait until RX ready

  *s = (uint8_t)(*((uint32_t *)(LPSPI0_DR)) & 0xFF);
}
```

B.3 FreeRTOS Main

```
#include "FreeRTOS.h"
#include "projdefs.h"
#include "queue.h"
#include "semphr.h"
#include "task.h"
#include "peripherals.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define mainTASK_PRIORITY (tskIDLE_PRIORITY + 2)

void UART_test(void *pvParameters);
void SPI_test(void *pvParameters);

int main(int argc, char **argv) {

  (void)argc;
  (void)argv;

  // Peripherals initialization
  UART_init();
  SPI_init();

  // Test Tasks creation
  xTaskCreate(UART_test, "UART_test", configMINIMAL_STACK_SIZE, NULL,
    mainTASK_PRIORITY, NULL);

  xTaskCreate(SPI_test, "SPI_test", configMINIMAL_STACK_SIZE, NULL,
              mainTASK_PRIORITY, NULL);

  vTaskStartScheduler();
  for (;;)
    ;
}
void UART_test(void *pvParameters) {

  (void)pvParameters;

  uart_printf("------------ Starting UART Test ------------\n");

    uart_printf("Hello from UART of Group10!\n");

  uart_printf("------------ Ending UART Test ------------\n");

  vTaskDelete(NULL);
```

```
}

void SPI_test(void *pvParameters) {

  (void)pvParameters;
  uart_printf("------------- Starting SPI Test -------------\n");

  SPI_status();

  uint32_t pippo = 0x00;
  uint32_t pluto = 0x00;
  for (int i=0; i<10; ++i) {
    uart_printf("---------- Write Test -------------\n");

    uart_printf("Sending: %x\n", pippo);
    SPI_write((uint8_t)pippo);

    uart_printf("---------- Read Test -------------\n");

    SPI_get((uint8_t *)&pluto);
    pippo = pluto;

    uart_printf("SPI read: %x\n", pluto);
    vTaskDelay(pdMS_TO_TICKS(1000));
  }

  SPI_status();

  uart_printf("---------- Ending SPI Test -------------\n");
}
```

B.4 FreeRTOSConfig

```
/*
 * FreeRTOS V202212.00
 * Copyright (C) 2020 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy of
 * this software and associated documentation files (the "Software"), to deal in
 * the Software without restriction, including without limitation the rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
 * the Software, and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
```

```
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 * https://www.FreeRTOS.org
 * https://github.com/FreeRTOS
 *
 */

#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/*-----------------------------------------------------------
* Application specific definitions.
*
* These definitions should be adjusted for your particular hardware and
* application requirements.
*
* THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
* FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
*
* See http://www.freertos.org/a00110.html
*----------------------------------------------------------*/

#define configUSE_TRACE_FACILITY                0
#define configGENERATE_RUN_TIME_STATS           0

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCPU_CLOCK_HZ                      ( ( unsigned long ) 24000000 )
#define configTICK_RATE_HZ                      ( ( TickType_t ) 1000 )
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 80 )
#define configTOTAL_HEAP_SIZE                   ( ( size_t ) ( 60 * 1024 ) )
#define configMAX_TASK_NAME_LEN                 ( 12 )
#define configUSE_TRACE_FACILITY                0
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 0
#define configUSE_CO_ROUTINES                   0
#define configUSE_MUTEXES                       1
#define configUSE_RECURSIVE_MUTEXES             1
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_QUEUE_SETS                    1
#define configUSE_COUNTING_SEMAPHORES           1

#define configMAX_PRIORITIES                    ( 9UL )
#define configMAX_CO_ROUTINE_PRIORITIES         ( 2 )
```

```
#define configQUEUE_REGISTRY_SIZE               10
#define configSUPPORT_STATIC_ALLOCATION         0

/* Timer related defines. */
#define configUSE_TIMERS                        0
#define configTIMER_TASK_PRIORITY               ( configMAX_PRIORITIES - 4 )
#define configTIMER_QUEUE_LENGTH                20
#define configTIMER_TASK_STACK_DEPTH            ( configMINIMAL_STACK_SIZE * 2 )

#define configUSE_TASK_NOTIFICATIONS            1
#define configTASK_NOTIFICATION_ARRAY_ENTRIES   3

/* Set the following definitions to 1 to include the API function, or zero
 * to exclude the API function. */

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           0
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_uxTaskGetStackHighWaterMark     1
#define INCLUDE_xTaskGetSchedulerState          1
#define INCLUDE_xTimerGetTimerDaemonTaskHandle  1
#define INCLUDE_xTaskGetIdleTaskHandle          1
#define INCLUDE_xSemaphoreGetMutexHolder        1
#define INCLUDE_eTaskGetState                   1
#define INCLUDE_xTimerPendFunctionCall          1
#define INCLUDE_xTaskAbortDelay                 1
#define INCLUDE_xTaskGetHandle                  1

/* This demo makes use of one or more example stats formatting functions. These
 * format the raw data provided by the uxTaskGetSystemState() function in to human
 * readable ASCII form.  See the notes in the implementation of vTaskList() within
 * FreeRTOS/Source/tasks.c for limitations. */
#define configUSE_STATS_FORMATTING_FUNCTIONS    0

#define configKERNEL_INTERRUPT_PRIORITY         ( 255 )        /* All eight bits
↪  as QEMU doesn't model the priority bits. */

#ifndef __IASMARM__ /* Prevent C code being included in IAR asm files. */
        #define configASSERT( x ) if( ( x ) == 0 ) while(1);
#endif


/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
 * See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY            ( 4 )
```

```
/* Use the Cortex-M3 optimised task selection rather than the generic C code
 * version. */
#define configUSE_PORT_OPTIMISED_TASK_SELECTION          1

/* The Win32 target is capable of running all the tests tasks at the same
 * time. */
#define configRUN_ADDITIONAL_TESTS                       1

/* The test that checks the trigger level on stream buffers requires an
 * allowable margin of error on slower processors (slower than the Win32
 * machine on which the test is developed). */
#define configSTREAM_BUFFER_TRIGGER_LEVEL_TEST_MARGIN    4

#define intqHIGHER_PRIORITY      ( configMAX_PRIORITIES - 5 )
#define bktPRIMARY_PRIORITY      ( configMAX_PRIORITIES - 3 )
#define bktSECONDARY_PRIORITY    ( configMAX_PRIORITIES - 4 )

#define configENABLE_BACKWARD_COMPATIBILITY 0

#endif /* FREERTOS_CONFIG_H */
```

B.5 Makefile

```
# The directory that contains FreeRTOS source code
FREERTOS_ROOT := /home/chuck/repos/FreeRTOS/FreeRTOS/

# Demo code
DEMO_PROJECT := .

# FreeRTOS kernel
KERNEL_DIR := $(FREERTOS_ROOT)/Source
KERNEL_PORT_DIR := $(KERNEL_DIR)/portable/GCC/ARM_CM3

# Where to store all the generated files (objects, elf and map)
OUTPUT_DIR := ./build

DEMO_NAME := demo
ELF := $(OUTPUT_DIR)/$(DEMO_NAME).elf
MAP := $(OUTPUT_DIR)/$(DEMO_NAME).map

# NXP toolchain
CC := ../gcc-10.2.0-Earmv7GCC-eabi/x86_64-linux/bin/arm-none-eabi-gcc
LD := ../gcc-10.2.0-Earmv7GCC-eabi/x86_64-linux/bin/arm-none-eabi-gcc
SIZE := ../gcc-10.2.0-Earmv7GCC-eabi/x86_64-linux/bin/arm-none-eabi-size

# Target embedded board and CPU
MACHINE := S32K3X8EVB
CPU := cortex-m7
```

```
# Use -s to connect to gdb port 1234 and -S to wait before executing
QEMU_FLAGS_DBG = -s -S

INCLUDE_DIRS = -I$(KERNEL_DIR)/include -I$(KERNEL_PORT_DIR)
INCLUDE_DIRS += -I$(DEMO_PROJECT)

VPATH += $(KERNEL_DIR) $(KERNEL_PORT_DIR) $(KERNEL_DIR)/portable/MemMang
VPATH += $(DEMO_PROJECT)

# Include paths. See INCLUDE_DIRS
CFLAGS = $(INCLUDE_DIRS)

# Don't include standard libraries
CFLAGS += -ffreestanding

# Target ARM cortex M3 processor
CFLAGS += -mcpu=$(CPU)
# Use the thumb 16 bits instruction set. Required for cortex-M
CFLAGS += -mthumb

# Print all the most common warnings
CFLAGS += -Wall

# Print extra warnings, not included in -Wall
CFLAGS += -Wextra

# Print warnings about variable shadowing (var name = outer scope var name)
CFLAGS += -Wshadow

# Include debug information (-g) with maximum level of detail (3)
CFLAGS += -g3

# Optimize (-O) the size (s) of the generated executable
CFLAGS += -Os

# Place each function in a dedicated section (see -gc-sections in LDFLAGS)
CFLAGS += -ffunction-sections

# Place each variable in a dedicated section (see -gc-sections in LDFLAGS)
CFLAGS += -fdata-sections

# Specify the linker script
LDFLAGS = -T ./S32K3.ld

# Don't use the standard start-up files
LDFLAGS += -nostartfiles

# Optimize compilation to reduce memory
```

```
LDFLAGS += -specs=nano.specs

# Optimize compilation targeting a system without any operating system
LDFLAGS += -specs=nosys.specs

# Generate map file with memory layout. Pass option to the linker (-Xlinker)
LDFLAGS += -Xlinker -Map=$(MAP)

# Remove unused sections. Used with -ffunction-sections and -fdata-sections
# optimized memory. Pass option to the linker (-Xlinker)
LDFLAGS += -Xlinker --gc-sections

# Link specifically for cortex M3 and thumb instruction set
LDFLAGS += -mcpu=$(CPU) -mthumb

# Kernel files
SOURCE_FILES += $(KERNEL_DIR)/list.c
SOURCE_FILES += $(KERNEL_DIR)/tasks.c
SOURCE_FILES += $(KERNEL_DIR)/queue.c
SOURCE_FILES += $(KERNEL_DIR)/portable/MemMang/heap_4.c
SOURCE_FILES += $(KERNEL_DIR)/portable/GCC/ARM_CM3/port.c

# Demo files
SOURCE_FILES += $(DEMO_PROJECT)/main.c
SOURCE_FILES += $(DEMO_PROJECT)/peripherals.c

# Start-up code
SOURCE_FILES += ./startup.c

# Create list of object files with the same names of the sources
OBJS = $(SOURCE_FILES:%.c=%.o)

# Remove path from object filename
OBJS_NOPATH = $(notdir $(OBJS))

# Prepend output dir to object filenames
OBJS_OUTPUT = $(OBJS_NOPATH:%.o=$(OUTPUT_DIR)/%.o)

all: $(ELF)

$(ELF): $(OBJS_OUTPUT) ./S32K3.ld Makefile
        echo "\n\n--- Final linking ---\n"
        $(LD) $(LDFLAGS) $(OBJS_OUTPUT) -o $(ELF)
        $(SIZE) $(ELF)

$(OUTPUT_DIR)/%.o : %.c  Makefile $(OUTPUT_DIR)
        $(CC) $(CFLAGS) -c $< -o $@

$(OUTPUT_DIR):
```

```
	mkdir -p $(OUTPUT_DIR)

cleanobj:
	rm -f $(OUTPUT_DIR)/*.o

clean:
	rm -rf £(ELF) £(MAP) £(OUTPUT_DIR)/*.o £(OUTPUT_DIR)

qemu_start:
	../qemu/build/qemu-system-arm -machine £(MACHINE) -cpu £(CPU) -kernel \
		£(ELF) -monitor none -nographic -serial stdio -d strace -d
		↪  trace:"memory_region_ops_*" -D ./log.txt

qemu_debug:
	../qemu/build/qemu-system-arm -machine £(MACHINE) -cpu £(CPU) -kernel \
	£(ELF) -monitor none -nographic -serial stdio £(QEMU_FLAGS_DBG) -m 128M
	↪  -append "root=/dev/sda console=ttyS0" -d trace:"memory_region_ops_read"
	↪  -D ./log.txt

gdb_start:
	gdb £(ELF) -ex "target remote :1234"
```