



# Secure Timeout System NXP S32K3X8EVB

Beamer for the CAOS Project

**Andrea Botticella**

**Fabrizio Emanuel**

**Elia Innocenti**

**Renato Mignone**

**Simone Romano**

February 5, 2025



**Politecnico  
di Torino**



# Table of Contents

## 1 Project Overview

- ▶ Project Overview
- ▶ QEMU Board Emulation
- ▶ FreeRTOS Porting
- ▶ Test Application
- ▶ Memory Protection Unit (MPU) Implementation
- ▶ Conclusion



# Project Overview

## 1 Project Overview

- The assignment consists of FOUR parts:
  - **Part 1: QEMU board emulation**
    - Generating a custom QEMU version to emulate the NXP S32K3X8EVB board.
    - Ensuring that QEMU emulates the proper CPU, memory map, and the peripherals assigned.
  - **Part 2: FreeRTOS porting**
    - Ensuring that FreeRTOS runs on the emulated board.
  - **Part 3: Writing a simple application**
    - Writing a simple application implementing different tasks to test the setup.



# Project Overview

## 1 Project Overview

- The assignment consists of FOUR parts:
  - **Part 4: Documentation and presentation**
    - Creating a tutorial to run and test your code.
    - Documentation of the code.
- What we've actually done:
  - *Secure Timeout System* application on the **NXP S32K3X8EVB board** using **FreeRTOS**, emulated with **QEMU**.
  - Refer to the dedicated markdown files in the repository: `README.md` and `GUIDE.md`. These files contain all the implementation details and the tutorial to replicate the project.



# Table of Contents

## 2 QEMU Board Emulation

- ▶ Project Overview
- ▶ **QEMU Board Emulation**
- ▶ FreeRTOS Porting
- ▶ Test Application
- ▶ Memory Protection Unit (MPU) Implementation
- ▶ Conclusion



# Custom QEMU Version

## 2 QEMU Board Emulation

- Emulate the **NXP S32K3X8EVB** board, which is not natively supported by QEMU.
- Ensure proper emulation of the **CPU (ARM Cortex-M7)**, memory map, and peripherals.





# Technical Details

## 2 QEMU Board Emulation

- Added a new machine model to **QEMU** for the **S32K3X8EVB** board, creating a dedicated .c file.
- Specifically took the **S32K358EVB** board as a reference for implementation.
- Implemented custom initialization routines for memory and peripherals based on its architecture.
- The **S32K358EVB** board specifications we implemented:
  - ARM Cortex-M7 CPU.
  - ~8MB Flash memory, 768KB SRAM, 256KB DTCM, and 128KB ITCM.
  - NVIC with 256 IRQs and 4 priority bits.
  - Multiple peripherals: 16 LPUART, 3 PIT timers, 16 MPU regions.
  - System clock running at 240MHz.



# Memory Regions Initialization

## 2 QEMU Board Emulation

- **Flash Memory:** Configured multiple blocks:
  - Block0: Base Address: 0x00400000, Size: 2 MB
  - Block1: Base Address: 0x00600000, Size: 2 MB
  - Block2: Base Address: 0x00800000, Size: 2 MB
  - Block3: Base Address: 0x00A00000, Size: 2 MB
  - Block4: Base Address: 0x10000000, Size: 128 KB
  - Utest: Base Address: 0x18000000, Size: 8 KB
- **SRAM Memory:**
  - Block0: Base Address: 0x20400000, Size: 256 KB
  - Block1: Base Address: 0x20440000, Size: 256 KB
  - Block2: Base Address: 0x20480000, Size: 256 KB
- **DTCM and ITCM Memory:**
  - DTCMo: Base Address: 0x20000000, Size: 128 KB
  - ITCMo: Base Address: 0x00000000, Size: 64 KB





# Peripherals and Interrupts Setup

## 2 QEMU Board Emulation

- **NVIC (Nested Vectored Interrupt Controller):**
  - Configured with 4 priority bits and 256 IRQs:
    - 1 Initial Stack Pointer value (-16)
    - 15 System Exceptions
    - 240 External Interrupts
- **LPUART (Low Power UART):**
  - Base Address: 0x4006A000
  - The board has **16 LPUART instances**.  
They are mapped starting from the UART base address.
  - Connected to NVIC and clocked by AIPS\_PLAT\_CLK and AIPS\_SLOW\_CLK.
- **PIT Timers (Periodic Interrupt Timer):**
  - Timer1: Base Address: 0x40037000
  - Timer2: Base Address: 0x40038000
  - Timer3: Base Address: 0x40039000



# System Clocks and Interrupts

## 2 QEMU Board Emulation

- **MPU:** 16 regions.
- **System Clock:**
  - Primary System Clock: 240MHz frequency, 4.16ns period.
  - AIPS Platform Clock: 80MHz
  - AIPS Slow Clock: 40MHz
  - Reference Clock: 1MHz
- **Interrupt Handling:**
  - Configured NVIC to handle exceptions and IRQs.
  - NVIC is connected to system and reference clocks.
  - Interrupt sources include timers, UARTs, and peripheral events.



# Firmware Loading

## 2 QEMU Board Emulation

- Function: `s32k3x8_load_firmware`
- Parameters:
  - `cpu`: The ARM CPU instance.
  - `ms`: The machine state.
  - `flash`: The memory region representing the flash memory.
  - `firmware_filename`: The filename of the firmware to be loaded.
- Functionality:
  - Reads the firmware file and loads its contents into the specified flash memory region.



# Class initialization

## 2 QEMU Board Emulation

- s32k3x8\_class\_init:

```
static void s32k3x8_class_init(ObjectClass *oc, void *data) {  
    MachineClass *mc = MACHINE_CLASS(oc);  
    mc->name = g_strdup("s32k3x8evb");  
    mc->desc = "NXP S32K3X8 EVB (Cortex-M7)";  
    mc->init = s32k3x8_init;  
    mc->default_cpu_type = ARM_CPU_TYPE_NAME("cortex-m7");  
    mc->default_cpus = 1;  
    mc->min_cpus = mc->default_cpus;  
    mc->max_cpus = mc->default_cpus;  
    mc->no_floppy = 1;  
    mc->no_cdrom = 1;  
    mc->no_parallel = 1;  
}
```



# Table of Contents

## 3 FreeRTOS Porting

- ▶ Project Overview
- ▶ QEMU Board Emulation
- ▶ **FreeRTOS Porting**
- ▶ Test Application
- ▶ Memory Protection Unit (MPU) Implementation
- ▶ Conclusion



# Overview

## 3 FreeRTOS Porting

- To test the **FreeRTOS Porting** on **QEMU**, a very simple application was created.
- The application runs a basic task that prints a message every second.
- If everything works correctly, it means that the **FreeRTOS Porting** has been successfully implemented.





# Setting Up FreeRTOS

## 3 FreeRTOS Porting

1. **Cloning** the FreeRTOS repository.
2. Creating the directory **structure**: App/ and App/Peripherals/.
3. Creating and implementing the following files in the App/ directory:
  - s32\_startup.c, s32\_linker.ld
  - FreeRTOSConfig.h
  - Makefile
  - main.c
  - Peripherals/: uart.c, printf-stdarg.c with their respective header files



# Running FreeRTOS on QEMU

## 3 FreeRTOS Porting

- main.c:

```
xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE, NULL,
            mainTASK_PRIORITY, NULL);

void vTask1(void *pvParameters)
{
    (void) pvParameters;

    for (;;)
    {
        printf("Task1 is running...\n");
        vTaskDelay(1000);
    }
}
```





# Running FreeRTOS on QEMU

## 3 FreeRTOS Porting

- Run the **Test**:
  - `cd App && make run`

```
Ready to run the scheduler ...  
Task1 is running ...  
Task1 is running ...  
Task1 is running ...  
Task1 is running ...
```

Figure: FreeRTOS Porting Test.



# Table of Contents

## 4 Test Application

- ▶ Project Overview
- ▶ QEMU Board Emulation
- ▶ FreeRTOS Porting
- ▶ **Test Application**
- ▶ Memory Protection Unit (MPU) Implementation
- ▶ Conclusion



# Secure Timeout System Application

## 4 Test Application

- The application is a simple implementation of a *Secure Timeout System*.
- It consists of **multiple tasks** that simulate events, monitor activities, and handle alerts.
- **Hardware timers** are used to generate **periodic interrupts** for activity detection.





# Task Implementation

## 4 Test Application

- **Event Task:**

- Periodically generates events that can be either user activities or suspicious activities.
- Uses a pseudo-random number generator to decide the type of event.
- Logs the generated event and updates the respective counters.

```
[EVENT SIMULATOR] ——— New Cycle Started —————  
[EVENT SIMULATOR] Generated: User Activity      | Count: 1
```

```
[EVENT SIMULATOR] ——— New Cycle Started —————  
[EVENT SIMULATOR] Generated: Security Event    | Count: 1
```

**Figure:** Generation of a user activity and a suspicious activity.



# Hardware Timer Initialization

## 4 Test Application

- **Timer 0:**
  - Configured to generate periodic interrupts.
  - Interrupt handler checks for **user activities** and sets the user activity detection flag.
- **Timer 1:**
  - Configured to generate periodic interrupts.
  - Interrupt handler checks for **suspicious activities** and sets the suspicious activity detection flag.



# Task Implementation

## 4 Test Application

- **Monitor Task:**

- Checks for user activity detection.
- Logs the status of user activity.
- Resets the user activity detection flag after logging.

- **Alert Task:**

- Checks for suspicious activity detection.
- Logs the status of the system security.
- Initiates security protocols if suspicious activity is detected.
- Resets the suspicious activity detection flag after logging.



# Implementation Details

## 4 Test Application

- **Global Variables:**

- Four main **flags**:

- `userActivity`, `userActivityDetection`,  
`suspiciousActivity`, `suspiciousActivityDetection`

- **Task Priorities:**

- Event Task has the highest priority to ensure timely event generation.
  - Monitor Task and Alert Task have lower priorities.

- **Timer Frequency:**

- Timer 0 and Timer 1 are configured to generate periodic interrupts at a frequency of 2 Hz.



# Implementation Details

## 4 Test Application

- **Task Priorities:**

```
// filepath: /App/SecureTimeoutSystem/secure_timeout_system.c
#define MONITOR_TASK_PRIORITY (tskIDLE_PRIORITY + 2)
#define ALERT_TASK_PRIORITY   (tskIDLE_PRIORITY + 3)
#define EVENT_TASK_PRIORITY   (tskIDLE_PRIORITY + 4)
```

- **Timer Frequency:**

```
// filepath: /App/Peripherals/IntTimer.c
#define tmrTIMER_0_FREQUENCY (2UL)
#define tmrTIMER_1_FREQUENCY (2UL)
```





# Run Example

## 4 Test Application

```
[EVENT SIMULATOR] ——— New Cycle Started ———
[EVENT SIMULATOR] Generated: Security Event | Count: 1

Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
[SECURITY ALERT] Suspicious activity detected | Status: ALARM
[SECURITY ALERT] Initiating security protocols...
[USER MONITOR] No activity | Status: IDLE
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
[SECURITY ALERT] Suspicious activity detected | Status: ALARM
[SECURITY ALERT] Initiating security protocols...
[USER MONITOR] No activity | Status: IDLE
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
[SECURITY ALERT] Suspicious activity detected | Status: ALARM
[SECURITY ALERT] Initiating security protocols...
[USER MONITOR] No activity | Status: IDLE
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
[SECURITY ALERT] Suspicious activity detected | Status: ALARM
[SECURITY ALERT] Initiating security protocols...
[USER MONITOR] No activity | Status: IDLE
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
[SECURITY ALERT] Suspicious activity detected | Status: ALARM
[SECURITY ALERT] Initiating security protocols...
[USER MONITOR] No activity | Status: IDLE
Timer 0 Interrupt: looking for user activities ...
Timer 1 Interrupt: looking for suspicious activities ...
```

Alt+F2 to open

How can I see the result of the terminal with you?

Alt+F2 to open

Alt+F2 to open

To take a screenshot of the terminal in Windows, you can use the following steps:

1. Open the Terminal: Open the terminal window that you want to take a screenshot of.

2. Take the Screenshot:

Using Keyboard Shortcut: Press **Alt+F2** to open the

terminal, and then press the **Alt+F2** shortcut to

take a screenshot of the terminal.

3. Save the Screenshot: The screenshot will be saved to your

default screenshot location. You can find the screenshot in your

default screenshot location.

4. Include the Screenshot in Your LaTeX Document: If you

want to include the screenshot in your LaTeX document, you

can use the `\includegraphics` command to include the screenshot.

Here is an example of how to include the screenshot in your LaTeX

document:

```
\includegraphics[width=10cm]{screenshot.png}
```

Replace `screenshot.png` with the path to your screenshot file.

That's all! You can now include the screenshot of the terminal

in your LaTeX document.

How can I see the result of the terminal with you?



# Table of Contents

## 5 Memory Protection Unit (MPU) Implementation

- ▶ Project Overview
- ▶ QEMU Board Emulation
- ▶ FreeRTOS Porting
- ▶ Test Application
- ▶ **Memory Protection Unit (MPU) Implementation**
- ▶ Conclusion



# Overview

## 5 Memory Protection Unit (MPU) Implementation

- The MPU enhances security by restricting memory access based on region configurations.
- The ARM Cortex-M7 processor supports up to 16 MPU regions.
- FreeRTOS provides built-in MPU support for ARM Cortex-M4, which can be theoretically adapted for Cortex-M7.
  - **Errata 837070:** Requires workarounds for Cortex-M7 ropo and rop1 revisions.



# Theoretical Steps for Implementation

## 5 Memory Protection Unit (MPU) Implementation

- **Define MPU Region Count in `FreeRTOSConfig.h`:**
  - Set `configENABLE_MPU` to 1.
  - Set `configTOTAL_MPU_REGIONS` to 16.
  - Set `configENABLE_ERRATA_837070_WORKAROUND` to 1.
- **Enable Errata Workaround:** Apply fix for Cortex-M7 ropo and rop1 by modifying `port.c`.
- **Integrate FreeRTOS Changes:** Adapt `ARM_CM4_MPU/port.c` to support Cortex-M7.



# MPU Configuration

## 5 Memory Protection Unit (MPU) Implementation

- Each MPU region is configured with:
  - Base address
  - Region size
  - Access permissions (privileged/unprivileged, read/write/execute)
- Enables separation of kernel and user-mode tasks.





# MPU in QEMU

## 5 Memory Protection Unit (MPU) Implementation

- **Device Tree (qtree)**
  - Used qtree to inspect the device tree of the board.
  - Found two MPU regions, each containing 8 blocks.
- **Script Execution**
  - Ran a script to determine how many MPU regions exist in a block.

```
#define MPU_TYPER (*(volatile unsigned int*)0xE000ED90)

void check_mpu() {
    printf("MPU_TYPER: 0x%08X\n", MPU_TYPER);
}
```

Figure: MPU Region Detection Script



# MPU in QEMU

## 5 Memory Protection Unit (MPU) Implementation

- **Result Analysis**

- The script returned 0x00000800.
- **Bit 0 (MPU Present Bit) = 0**
  - Some Cortex-M chips use this bit to indicate MPU presence.
  - For Cortex-M7 (S32K3 series), this bit is always 0.
- **Bits 15:8 (DREGION) = 8**
  - MPU supports 8 regions.

```
MPU_TYPER: 0x00000800
```

Figure: MPU Detection Result



# Table of Contents

## 6 Conclusion

- ▶ Project Overview
- ▶ QEMU Board Emulation
- ▶ FreeRTOS Porting
- ▶ Test Application
- ▶ Memory Protection Unit (MPU) Implementation
- ▶ Conclusion





# Conclusion

## 6 Conclusion

- The `s32k3x8evb_board.c` file plays a crucial role in the emulation of the **NXP S32K3X8EVB board** within **QEMU**.
- It provides the necessary functions to load firmware, initialize memory regions, set up hardware components, and manage system clocks and interrupts.
- All the implementations and detailed information about the project are contained in the repository.
- Repository link: <https://baltig.polito.it/caos2024/group2.git>



*Thank you for listening!*  
*Any questions?*