

# Contents

|       |                                                                   |    |
|-------|-------------------------------------------------------------------|----|
| 1     | Overview                                                          | 2  |
| 1.1   | Introduction                                                      | 2  |
| 1.2   | Description                                                       | 2  |
| 1.3   | Objective                                                         | 2  |
| 2     | NuttX Structure                                                   | 4  |
| 2.1   | What is NuttX                                                     | 4  |
| 2.2   | NSH Shell                                                         | 4  |
| 2.3   | NuttX Repository Structure                                        | 5  |
| 2.3.1 | Overview of NuttX Repositories                                    | 5  |
| 2.3.2 | NuttX OS Repository Structure                                     | 5  |
| 2.3.3 | NuttX Apps Repository Structure                                   | 6  |
| 2.4   | Build System: Make and CMake in NuttX                             | 6  |
| 2.4.1 | Make in NuttX                                                     | 6  |
| 2.4.2 | CMake in NuttX                                                    | 7  |
| 3     | STM32H745I-DISCO Structure                                        | 8  |
| 3.1   | Overview of STM32H745I-DISCO                                      | 8  |
| 3.2   | Microcontroller Core Structure                                    | 9  |
| 3.3   | Board Layout and Components                                       | 10 |
| 4     | Making NuttX run on STM32H745I-DISCO                              | 11 |
| 4.1   | Setting up the Environment                                        | 11 |
| 4.1.1 | Installing Dependencies                                           | 11 |
| 4.1.2 | Cloning NuttX Repositories                                        | 12 |
| 4.2   | Creating a New Build                                              | 12 |
| 4.3   | Building and Flashing                                             | 13 |
| 5     | Per Core NuttX Instance                                           | 14 |
| 5.1   | Create New Configuration                                          | 14 |
| 5.2   | Cortex <sup>®</sup> -M4 Configuration                             | 14 |
| 5.3   | Cortex <sup>®</sup> -M7 Configuration                             | 16 |
| 5.4   | Configuring UART7 for the Cortex <sup>®</sup> -M4 Core            | 17 |
| 5.5   | Modifying the Linker Script                                       | 17 |
| 5.5.1 | Modifying the Existing Linker Script                              | 18 |
| 5.5.2 | Creating a New Linker Script for the Cortex <sup>®</sup> -M4 Core | 18 |
| 5.6   | Editing the CMakeLists.txt File                                   | 19 |
| 5.7   | Building and Flashing                                             | 19 |
| 5.7.1 | Building the NuttX Instances                                      | 19 |
| 5.7.2 | Flashing HEX Files to STM32H745I-DISCO                            | 20 |
| 5.7.3 | Verifying Output                                                  | 20 |
| 6     | OpenAMP Structure                                                 | 22 |
| 6.1   | Overview of OpenAMP                                               | 22 |

|       |                                                        |    |
|-------|--------------------------------------------------------|----|
| 6.3   | Importance in Multi-Core Systems . . . . .             | 23 |
| 7     | RPTun . . . . .                                        | 24 |
| 7.1   | Create New Configuration . . . . .                     | 24 |
| 7.2   | Cortex®-M4 Configuration . . . . .                     | 25 |
| 7.3   | Cortex®-M7 Configuration . . . . .                     | 25 |
| 7.4   | RPMMsg Initialization . . . . .                        | 27 |
| 7.5   | Building and Flashing . . . . .                        | 27 |
| 7.5.1 | Building the NuttX Instances . . . . .                 | 27 |
| 7.5.2 | Flashing HEX Files to STM32H745I-DISCO . . . . .       | 28 |
| 7.5.3 | Verifying Output . . . . .                             | 28 |
| 8     | RPMMsg Ping-Pong . . . . .                             | 30 |
| 8.1   | Creating Custom NuttX Application . . . . .            | 30 |
| 8.2   | Enabling, Building, and Flashing Application . . . . . | 31 |
| 8.2.1 | Enabling Ping Pong Application . . . . .               | 31 |
| 8.2.2 | Building . . . . .                                     | 31 |
| 8.2.3 | Flashing . . . . .                                     | 31 |
| 8.2.4 | Verifying Output . . . . .                             | 32 |
| .1    | Cortex®-M7 Linker Script Condition . . . . .           | 34 |
| .2    | Cortex®-M7 Linker Script Shared Memory . . . . .       | 35 |
| .3    | Cortex®-M4 Linker Script . . . . .                     | 35 |
| .4    | RPMMsg Initialize . . . . .                            | 37 |
| .5    | Ping-Pong Kconfig . . . . .                            | 38 |
| .6    | Ping-Pong C File . . . . .                             | 39 |
| .7    | Ping-Pong CmakeList.txt . . . . .                      | 42 |
|       | References . . . . .                                   | 46 |

# 1 Overview

## 1.1 Introduction

In the realm of embedded systems, the choice of an appropriate real-time operating system (RTOS) is critical for ensuring efficient, reliable, and deterministic performance. NuttX, an RTOS developed by the Apache Software Foundation, stands out for its lightweight design and POSIX-compliant environment, making it suitable for a wide range of microcontroller-based applications. This thesis explores the porting of NuttX to the STM32H745I-DISCO, a development board equipped with dual-core Arm® Cortex®-M4 and Cortex®-M7 processors. This board's dual-core architecture offers a unique opportunity to investigate and harness the parallel processing capabilities within an RTOS environment.

## 1.2 Description

The primary goal of this project is to port the NuttX RTOS to the STM32H745I-DISCO development board, utilizing the full spectrum of the board's peripherals and dual-core architecture. By doing so, this project aims to demonstrate how NuttX can be configured to take full advantage of modern microcontroller capabilities, including inter-core communication and parallel task execution. The successful implementation of NuttX on this platform could serve as a foundation for future embedded projects that require robust, real-time performance.

## 1.3 Objective

The objectives of this project are structured to achieve a comprehensive implementation and evaluation of the NuttX real-time operating system (RTOS) on the STM32H745I-DISCO development board. Specifically, the project aims to accomplish the following:

1. **Port and configure NuttX:** Establish a fully operational NuttX environment on the STM32H745I-DISCO, ensuring that all board peripherals are correctly configured and utilized to their maximum potential.
2. **Set up separate NuttX instances for Cortex<sup>®</sup>-M4 and Cortex<sup>®</sup>-M7 cores:** Enable the dual-core system to operate specific tasks independently on each core, thereby enhancing the board's capability for parallel processing and efficient task management.
3. **Implement Remote Proc Tunnel Drivers (RPTUN):** Develop and integrate RPTUN to facilitate efficient inter-core communication between the Cortex-M4 and Cortex-M7 cores, a critical feature for systems requiring real-time coordination across multiple processing units.
4. **Develop and test a Ping-Pong application:** Create and validate a Ping-Pong application using the RPMsg protocol from the OpenAMP framework. This application will serve as a practical demonstration of inter-core communication, showcasing the effectiveness of the dual-core setup in a real-world scenario.

## 2 NuttX Structure

### 2.1 What is NuttX

NuttX is a real-time operating system (RTOS) developed and maintained by the Apache Software Foundation 2024. It is designed to be highly scalable, supporting a broad range of microcontrollers from 8-bit to 64-bit architectures, which makes it a versatile choice for embedded system applications. The operating system is particularly valued for its small footprint (Apache Software Foundation 2024), POSIX-compliant environment, and extensive support for different architectures and boards (Apache Software Foundation 2024). Due to these characteristics, NuttX has been adopted in various projects, ranging from simple embedded devices to more complex industrial systems.

### 2.2 NSH Shell

The NuttX Shell (NSH) is a lightweight and flexible shell interface provided by NuttX. It offers a simple, command-line interface (CLI) that allows users to interact directly with the operating system, execute commands, manage files, and perform debugging operations (Apache Software Foundation 2024). The NSH shell is essential for testing and developing applications on embedded systems, providing developers with a powerful tool for real-time system management and interaction (Apache Software Foundation 2024). Due to its lightweight design, NSH is particularly suited for use in resource-constrained environments, where a full-featured shell might be too demanding.



Figure 2.1: Apache Software Foundation

## 2.3 NuttX Repository Structure

### 2.3.1 Overview of NuttX Repositories

The NuttX real-time operating system (RTOS) is organized into two distinct repositories: one dedicated to the core operating system and another for additional applications. This separation facilitates modular development and simplifies the management of both the OS itself and the applications that run on it.

- **NuttX OS Repository:** This repository contains the essential components of the NuttX RTOS, including the kernel, device drivers, and critical libraries. These core elements form the foundation of the operating system, which is adaptable to a wide range of hardware platforms.
- **NuttX Apps Repository:** The applications repository comprises a collection of user-space applications designed to run on the NuttX OS. This repository includes example applications, test programs, and utilities that extend the functionality of the core OS, providing developers with tools to test and demonstrate NuttX's capabilities.

### 2.3.2 NuttX OS Repository Structure

The NuttX OS repository is structured to facilitate the development, configuration, and deployment of the RTOS across different platforms. Key directories within this repository include:

- **arch/:** This directory contains architecture-specific code for the supported processors, such as ARM, x86, and others. Each subdirectory within '**arch/**' corresponds to a different CPU architecture.
- **boards/:** This directory houses board-specific configurations and setup files. Within '**boards/**', each supported development board has its own subdirectory, which includes startup scripts, board-specific drivers, and configuration files.
- **drivers/:** The '**drivers/**' directory contains the device drivers that interface with hardware peripherals like UARTs, SPI, I2C, and others. These drivers are crucial for enabling communication between the hardware and the NuttX kernel.
- **fs/:** The file system directory holds implementations of various file systems that NuttX supports, such as FAT, NFS, and ROMFS.
- **include/:** This directory contains the header files necessary for building NuttX. It includes system-wide headers, as well as architecture-specific ones.
- **sched/:** The scheduler directory manages task scheduling, thread management, and inter-process communication within NuttX.

### 2.3.3 NuttX Apps Repository Structure

The NuttX Apps repository is organized to support a wide range of applications, examples, and tools that can be used with the NuttX OS. Key directories include:

- **examples/**: This directory contains sample applications that demonstrate how to use various NuttX features. These examples are helpful for new users learning how to develop applications for NuttX.
- **system/**: The system directory includes essential system utilities and services that run in user space. Examples include shell utilities, network services, and file manipulation tools.
- **testing/**: The testing directory contains test programs used to verify the functionality of the NuttX OS and its applications. These tests are crucial for ensuring that the system behaves as expected across different hardware platforms.
- **include/**: Similar to the NuttX OS repository, the apps repository also includes an **'include/'** directory with header files necessary for building the applications.
- **misc/**: This directory contains miscellaneous utilities and scripts that support application development.

## 2.4 Build System: Make and CMake in NuttX

NuttX employs both Make and CMake as part of its build system, each serving distinct roles in the configuration and compilation processes.

### 2.4.1 Make in NuttX

Make is the traditional build tool used in NuttX, responsible for compiling the source code according to the rules defined in various Makefiles. These Makefiles are distributed across the NuttX directory structure, with key files located in directories like **'arch/'**, **'boards/'**, and **'drivers/'**. The build process begins with configuring NuttX using the **'make menuconfig'** command, which generates a **'config'** file that guides the subsequent build steps (Apache Software Foundation 2024). Once configured, running **make** in the root directory compiles the OS, producing binaries that can be flashed onto the target hardware (Apache Software Foundation 2024).

## 2.4.2 CMake in NuttX

CMake is increasingly used in NuttX due to its flexibility and portability. CMake generates platform-specific build files from '**CMakeLists.txt**' files scattered throughout the repository. This approach simplifies the build process across various hardware platforms and supports modern development environments (Apache Software Foundation 2024). For example, the command '`cmake -B build -DBOARD_CONFIG=stm32h745i-disco:nsh -GNinja`' sets up a build environment for the STM32H745I-DISCO board, utilizing the Ninja build system for efficient compilation (Apache Software Foundation 2024).

## 3 STM32H745I-DISCO Structure

The STM32H745I-DISCO is a powerful development board designed by STMicroelectronics, featuring a dual-core microcontroller that combines an Arm® Cortex®-M4 and an Arm® Cortex®-M7 processor. Understanding the structure and capabilities of this board is essential for effectively porting and running the NuttX RTOS.

### 3.1 Overview of STM32H745I-DISCO

The STM32H745I-DISCO is specifically engineered to facilitate the development of applications that require high-performance processing and real-time capabilities. It offers a wide array of features, including a high-resolution touchscreen, multiple communication interfaces (like USB, Ethernet, and CAN), and extensive GPIO options, making it an ideal platform for complex embedded systems (STMicroelectronics 2024). *Figure 3.1 on page 9 illustrates the STM32H745I-DISCO development board, highlighting its key components and layout.*

- **Dual-Core Architecture:** The STM32H745I-DISCO's standout feature is its dual-core design, with an Arm® Cortex®-M7 core running at up to 480 MHz for high-performance tasks and a Cortex®-M4 core at 240 MHz for efficient real-time control. This setup allows parallel processing, where the M7 core handles demanding computations, and the M4 core manages real-time operations and communication (STMicroelectronics 2024).
- **Peripheral Support:** The board is equipped with numerous peripherals, including high-speed USB, Ethernet, and multiple UARTs, SPI, I2C, and GPIOs. These peripherals are crucial for developing complex applications that require extensive input/output operations (STMicroelectronics 2024).
- **Memory and Storage:** The STM32H745I-DISCO includes a substantial amount of on-chip memory, including SRAM, Flash, and support for external memory via a QSPI interface (STMicroelectronics 2024). This memory structure is essential for running an RTOS like NuttX, which requires sufficient resources for both the OS and user applications.



Figure 3.1: STM32H745I-DISCO Development Board

## 3.2 Microcontroller Core Structure

The microcontroller unit (MCU) on the STM32H745I-DISCO board is the STM32H745XI, which incorporates the dual-core setup. Here's a closer look at each core's capabilities and roles:

- **Cortex<sup>®</sup>-M7 Core:**
  - **High-Performance Processing:** The Cortex-M7 core is designed for tasks requiring significant computational power, such as digital signal processing, high-speed data manipulation, and complex algorithms. It features a double-precision floating-point unit (FPU) and an L1 cache, which help to accelerate these tasks (STMicroelectronics 2024).
  - **Real-Time Performance:** Despite its high processing power, the Cortex-M7 core also supports real-time operations, making it suitable for time-sensitive applications where both speed and precision are critical (STMicroelectronics 2024).
- **Cortex<sup>®</sup>-M4 Core:**
  - **Real-Time Control:** The Cortex-M4 core is optimized for real-time control tasks. It includes a single-precision FPU and an adaptive real-time accelerator (ART Accelerator), which enhance its performance in handling control algorithms, sensor data processing, and communication protocols (STMicroelectronics 2024).
  - **Energy Efficiency:** The M4 core is designed to be energy-efficient, making it ideal for tasks that need to be always active, such as monitoring sensors or handling low-latency communication with other devices (STMicroelectronics 2024).

- **Inter-Core Communication:** The STM32H745XI MCU allows for efficient communication between the M7 and M4 cores. This inter-core communication is a critical feature for developing complex embedded systems, enabling the cores to collaborate on tasks while maintaining separate responsibilities (STMicroelectronics 2024).

### 3.3 Board Layout and Components

A thorough understanding of the STM32H745I-DISCO board's physical layout and components is crucial for effective development and deployment of embedded systems. This section provides an overview of the key connectors, interfaces, and peripherals that are integral to the board's operation.

- **Connectors and Interfaces:** The STM32H745I-DISCO board is equipped with multiple connectors that support a wide range of peripherals. These include Arduino Uno V3 connectors, facilitating expansion with additional modules or sensors, and a USB OTG connector, which allows for flexible USB device and host configurations. The board also includes an Ethernet jack and audio interfaces, enhancing its capability for networked and multimedia applications. (STMicroelectronics 2024).
- **Display and Input:** The board features a 4.3-inch color TFT LCD with capacitive touch capabilities. This display, driven by the Cortex<sup>®</sup>-M7 core, is optimized for the development of graphical user interfaces (GUIs), offering a responsive and interactive visual output that is ideal for applications requiring real-time data visualization. (STMicroelectronics 2024).
- **Debugging and Programming Interfaces:** Integrated into the STM32H745I-DISCO board is an ST-LINK/V3 debugger/programmer. This tool simplifies the process of uploading code and debugging applications directly on the hardware, making development more efficient and accessible. (STMicroelectronics 2024).

These components, along with the board's extensive peripheral support and dual-core architecture, provide a versatile platform for developing sophisticated embedded systems that leverage the full potential of NuttX RTOS.

## 4 Making NuttX run on STM32H745I-DISCO

### 4.1 Setting up the Environment

To deploy NuttX on the STM32H745I-DISCO, it is necessary to configure the development environment by installing required dependencies, setting up the ARM toolchain, and cloning the NuttX repositories.

#### 4.1.1 Installing Dependencies

Install essential packages on a Debian-based system:

```
sudo apt install \
bison flex gettext texinfo libncurses5-dev \
libncursesw5-dev xxd gperf automake libtool \
pkg-config build-essential gperf genromfs \
libgmp-dev libmpc-dev libmpfr-dev libisl-dev \
binutils-dev libelf-dev libexpat-dev \
gcc-multilib g++-multilib picocom \
u-boot-tools util-linux
```

Next, install the KConfig frontend:

```
sudo apt install kconfig-frontends
pip install kconfiglib
```

Finally, install the ARM toolchain:

```
sudo apt install gcc-arm-none-eabi binutils-arm-
none-eabi
```

### 4.1.2 Cloning NuttX Repositories

Clone the NuttX OS and Apps repositories using:

```
git clone https://github.com/apache/nuttX.git
    nuttX
git clone https://github.com/apache/nuttX-apps
    apps
```

(Apache Software Foundation 2024)

## 4.2 Creating a New Build

Once the environment is set up, the next step is to create a build configuration specifically tailored for the STM32H745I-DISCO board.

1. **Configuring the Build:** Change to the '**nuttX**/' directory and run the following command to configure the build for the STM32H745I-DISCO:

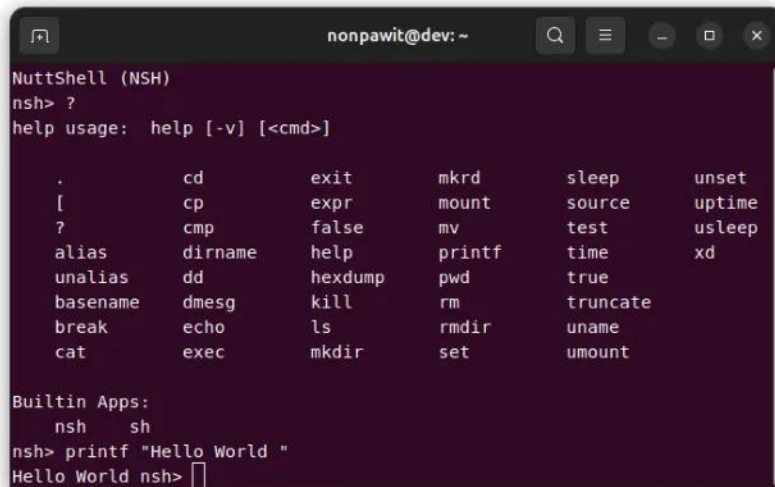
```
cd nuttX
cmake -B build -DBOARD_CONFIG=stm32h745i-disco:
      nsh -GNinja
```

Here, specifies the directory for the build files, and '**-DBOARD\\_CONFIG=stm32h745i-disco:nsh**' tells NuttX to use the configuration for the STM32H745I-DISCO board with the '**nsh**' (NuttX Shell) setup.

2. **Enabling SMPS Supply Mode:** During the development process, SMPS (Switched-Mode Power Supply) mode might need to be enabled, which is not enabled by default at the time of development. This can be done using '**menuconfig**':

```
cmake --build build -t menuconfig
```

Navigate to '**System Type > Use direct SMPS supply mode**' and enable it. This ensures the board use SMPS during operation.



```

nonpawit@dev: ~
NuttShell (NSH)
nsh> ?
help usage:  help [-v] [<cmd>]

.      cd      exit      mkrd      sleep      unset
[      cp      expr      mount     source     uptime
?      cmp      false     mv        test       usleep
alias  dirname  help      printf    time       xd
unalias dd      hexdump   pwd       true
basename dmesg    kill      rm        truncate
break   echo     ls        rmdir     uname
cat     exec     mkdir     set       umount

Builtin Apps:
nsh  sh
nsh> printf "Hello World "
Hello World nsh>

```

Figure 4.1: NuttX NSH Console

## 4.3 Building and Flashing

After configuring the build, the next steps are to compile the code and flash it onto the STM32H745I-DISCO.

1. **Building the Code:** To compile the NuttX code, run:

```
cmake --build build
```

This command compiles the code according to the configuration and generates the necessary binaries.

2. **Flashing the Firmware:** Use the STM32CubeProgrammer tool to flash the compiled NuttX firmware onto the STM32H745I- DISCO. The HEX file generated in the build process can be flashed with STM32CubeProgrammer.
3. **Verifying the Installation:** Once NuttX is running on the STM32H745I-DISCO, you can connect to the board using a serial console. For instance, on Linux, you can use screen:

```
screen /dev/ttyACM0 115200
```

This connects to the board's NSH shell, where you can interact with NuttX by entering commands like '?' to list available commands or 'printf "Hello World"' to test output. *Figure 4.1 on page 13 shows the NuttX sample interactions.*

## 5 Per Core NuttX Instance

### 5.1 Create New Configuration

To fully utilize the dual-core architecture, it is necessary to create distinct NuttX configurations for each core. This involves duplicating the default configuration and modifying it for the specific needs of the cortex<sup>®</sup>-M4 and cortex<sup>®</sup>-M7 cores.

1. **Duplicate the Default Configuration:** Start by copying the default 'nsh' configuration to create separate configurations for each core:

```
cp -r boards/arm/stm32h7/stm32h745i-disco/  
  configs/nsh boards/arm/stm32h7/stm32h745i-  
  disco/configs/nsh_cm4  
cp -r boards/arm/stm32h7/stm32h745i-disco/  
  configs/nsh boards/arm/stm32h7/stm32h745i-  
  disco/configs/nsh_cm7
```

2. **Modify the Configuration Files:** Each configuration must be tailored to the specific core by adjusting the 'defconfig' files.

### 5.2 Cortex<sup>®</sup>-M4 Configuration

Configuring the Cortex<sup>®</sup>-M4 core requires several critical adjustments to ensure that it operates independently from the Cortex<sup>®</sup>-M7 core. These adjustments involve modifying the default configuration to allocate resources appropriately and enable essential features for the Cortex<sup>®</sup>-M4.

1. **Disable Architecture-Specific Features:** The Cortex<sup>®</sup>-M4 should not control LEDs which typically managed by the M7 core. In the defconfig file for 'nsh\_cm4', disable the LED control:

```
# CONFIG_ARCH_LEDS is not set
```

2. **Enable Cortex<sup>®</sup>-M4 Core and Enable Upper-Half Serial by Default:** Ensure that the Cortex<sup>®</sup>-M4 core is selected as the active core and that the upper-half serial is enabled by default. This is done by removing the following lines from the defconfig file:

```
# CONFIG_STANDARD_SERIAL is not set
# CONFIG_STM32H7_CORTEXM4_ENABLED is not set
```

3. **Select Cortex<sup>®</sup>-M4 Instead of Cortex<sup>®</sup>-M7:** Modify the configuration to use the cortex<sup>®</sup>-M4 core:

```
CONFIG_ARCH_CHIP_STM32H7_CORTEXM7=y
```

should be replaced with:

```
CONFIG_ARCH_CHIP_STM32H7_CORTEXM4=y
```

4. **Disable Cachine for Cortex<sup>®</sup>-M7:** Since the Cortex<sup>®</sup>-M7 is not enabled, caching specific to the Cortex<sup>®</sup>-M7 must be disabled by removing the following lines:

```
CONFIG_ARMV7M_DCACHE=y
CONFIG_ARMV7M_DCACHE_WRITETHROUGH=y
CONFIG_ARMV7M_DTCM=y
CONFIG_ARMV7M_ICACHE=y
```

5. **Enable C++ Compiler and Initialization:** To support C++ code, enable the C++ compiler and initialization by adding the following lines:

```
CONFIG_HAVE_CXX=y
CONFIG_HAVE_CXXINITIALIZE=y
```

6. **Adjust Memory Regions:** The memory regions should be configured to avoid conflicts with the Cortex<sup>®</sup>-M7 core:

```
CONFIG_RAM_START=0x10000000
CONFIG_MM_REGIONS=1
```

7. **Set NSH Prompt:** Set a unique prompt for the Cortex<sup>®</sup>-M4 NSH shell:

```
CONFIG_NSH_PROMPT_STRING="nsh - cm4> "
```

8. **Enable UART7:** Since USART3 which is connected to ST-LINKV3 is being used by Cortex<sup>®</sup>-M7 core.

```
CONFIG_STM32H7_USART3=y
CONFIG_USART3_SERIAL_CONSOLE=y
```

should be replaced with:

```
CONFIG_STM32H7_UART7=y
CONFIG_UART7_SERIAL_CONSOLE=y
```

9. **Enable SMPS:** Check if the configuration has SMPS enabled or not, if not add the following line.

```
CONFIG_STM32H7_PWR_DIRECT_SMPS_SUPPLY=y
```

These adjustments ensure that the Cortex<sup>®</sup>-M4 core operates with its designated resources and settings, independent of the Cortex<sup>®</sup>-M7 core.

## 5.3 Cortex<sup>®</sup>-M7 Configuration

The configuration for the Cortex<sup>®</sup>-M7 core also requires specific adjustments to leverage its processing capabilities while operating independently from the Cortex<sup>®</sup>-M4 core.

1. **Enable Cortex<sup>®</sup>-M4 Core via Cortex<sup>®</sup>-M7:** Remove the following line to ensure that the CM4 core is enabled:

```
# CONFIG_STM32H7_CORTEXM4_ENABLED is not set
```

2. **Disable Debug Features:** To streamline the configuration, disable unnecessary debug features by removing the following line:

```
CONFIG_DEBUG_FEATURES=y
```

3. **Enable C++ Compiler and Initialization:** Similar to the Cortex<sup>®</sup>-M4, enable the C++ compiler and initialization:

```
CONFIG_HAVE_CXX=y
CONFIG_HAVE_CXXINITIALIZE=y
```

4. **Set NSH Prompt:** Set a unique prompt for the Cortex<sup>®</sup>-M7 NSH shell:

```
CONFIG_NSH_PROMPT_STRING="nsh - cm4 > "
```

5. **Enable SMPS:** Check if the configuration has SMPS enabled or not, if not add the following line.

```
CONFIG_STM32H7_PWR_DIRECT_SMPS_SUPPLY=y
```

## 5.4 Configuring UART7 for the Cortex®-M4 Core

In a dual-core setup, UART communication is often divided between the two cores to facilitate independent operations. The default NuttX configuration assigns USART3 to the Cortex®-M7 core. Therefore, UART7 should be configured for the Cortex®-M4 core.

To achieve this, modifications must be made to the 'board.h' file, which is located in the following directory:

```
nuttX/boards/arm/stm32h7/stm32h745i-disco/include/
board.h
```

By adding the following line in alternate function pin selections section, UART7 will be enabled and recognised by NuttX.

```
#define GPIO_UART7_RX    (GPIO_UART7_RX_1 |
    GPIO_SPEED_100MHz)
#define GPIO_UART7_TX    (GPIO_UART7_TX_2 |
    GPIO_SPEED_100MHz)
```

## 5.5 Modifying the Linker Script

In a dual-core system such as the STM32H745I-DISCO, it is essential to modify the linker script to allocate distinct memory regions for each core. This ensures that the Cortex®-M4 and Cortex®-M7 cores operate independently while sharing certain resources as needed.

### 5.5.1 Modifying the Existing Linker Script

The default linker script ('flash.ld') provided by NuttX is designed for single-core configurations. To adapt it for dual-core operation, conditions must be introduced that handle memory allocation separately for each core.

1. **Locate the Linker Script:** The linker script is located in the following directory:

```
nuttX/boards/arm/stm32h7/stm32h745i-disco/
scripts/flash.ld
```

2. **Modify the Linker Script:** To enable the dual-core setup, modify the 'flash.ld' file to conditionally allocate memory regions based on the active core. This includes specifying the flash memory and SRAM allocations: see Appendix .1, as well as setting up shared memory for inter-core communication: Appendix .2.

This modification divides the memory between the two cores, with shared memory specifically allocated for inter-core communication.

### 5.5.2 Creating a New Linker Script for the Cortex®-M4 Core

Given that the default linker script is primarily configured for the Cortex®-M7 core, a new linker script needs to be created for the Cortex®-M4 core. This script ensures that the Cortex®-M4 core is assigned its own memory space without conflicting with the Cortex®-M7 core's memory allocations.

1. **Create a New Linker Script:** A new linker script, 'flash\_cm4.ld', should be created in the same directory as 'flash.ld':

```
nuttX/boards/arm/stm32h7/stm32h745i-disco/
scripts/flash_cm4.ld
```

2. **Linker Script for the Cortex®-M4 Core:** The complete linker script that defines the memory regions specific to the Cortex®-M4 core is provided in Appendix .3. This ensures that the Cortex®-M4 core operates independently and efficiently.

## 5.6 Editing the CMakeLists.txt File

The development process primarily relies on CMake for compiling, all necessary compiler configuration changes will be made within the CMake framework. The STM32H745I-DISCO NuttX CMake configuration file is located at:

```
nuttx/boards/arm/stm32h7/stm32h745i-disco/src/
CMakeLists.txt
```

The primary modification required in this file is to select the appropriate linker script based on the core for which CMake is compiling. By replacing the following lines, CMake will correctly compile using the core-specific linker script:

```
if(CONFIG_ARCH_CHIP_STM32H7_CORTEXM7)
    set_property(GLOBAL PROPERTY LD_SCRIPT
        "${NUTTX_BOARD_DIR}/scripts/flash.ld")
else()
    set_property(GLOBAL PROPERTY LD_SCRIPT
        "${NUTTX_BOARD_DIR}/scripts/flash_m4.ld")
endif()
```

This conditional statement instructs CMake to use ‘flash.ld’ when compiling for the Cortex<sup>®</sup>-M7 core and ‘flash\_cm4.ld’ when compiling for the Cortex<sup>®</sup>-M4 core. This ensures that each core is linked with its specific memory configuration, as defined in the corresponding linker script.

## 5.7 Building and Flashing

### 5.7.1 Building the NuttX Instances

To generate the NuttX operating system binaries for each core, the following steps should be executed:

1. **Building the NuttX Instance for the Cortex<sup>®</sup>-M4 Core:** Navigate to the nuttx directory and execute the following commands to create and compile the NuttX instance tailored for the Cortex<sup>®</sup>-M4 core:

```
cmake -B build_cm4 -DBOARD_CONFIG=
      stm32h745i-disco:nsh_cm4 -GNinja
cmake --build build_cm4
```

2. **Building the NuttX Instance for the Cortex<sup>®</sup>-M7 Core:** Similarly, execute the following commands to compile the NuttX instance for the Cortex<sup>®</sup>-M7 core:

```
cmake -B build_cm7 -DBOARD_CONFIG=
      stm32h745i-disco:nsh_cm7 -GNinja
cmake --build build_cm7
```

By utilizing CMake in conjunction with the Ninja build system, these commands efficiently compile the NuttX code into distinct binaries for each core, reflecting their specific configurations.

### 5.7.2 Flashing HEX Files to STM32H745I-DISCO

Once the NuttX instances have been successfully built, the next critical step is to flash the generated .hex files onto the STM32H745I-DISCO board using STM32CubeProgrammer, starting with the Cortex<sup>®</sup>-M7 core, followed by the Cortex<sup>®</sup>-M4 core, ensuring successful completion and avoiding memory conflicts.

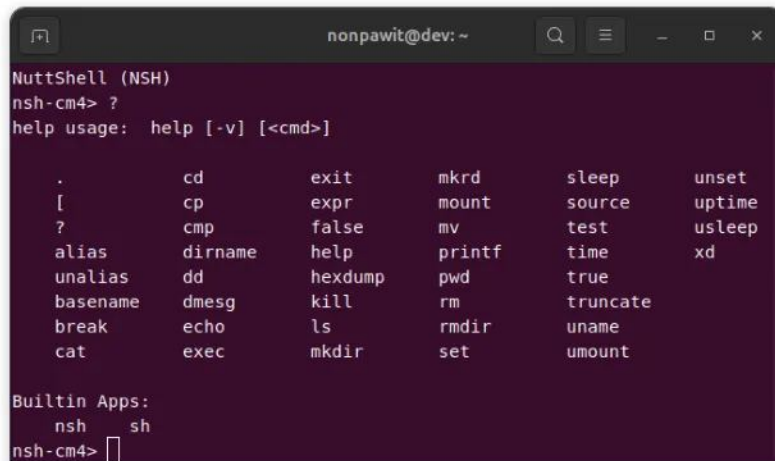
### 5.7.3 Verifying Output

Post-flashing, it is imperative to verify that each NuttX instance is functioning as intended.

1. **Connect to Serial Consoles:** Open two terminal sessions, one for each core. Connect Cortex<sup>®</sup>-M4 via external USB to UART converter.

```
screen /dev/ttyACM0 115200 // For CM7
screen /dev/ttyUSB0 115200 // For CM4
```

2. **Validating NSH Shell:** The result of two terminals output can be seen in figure 5.1 on page 21 and figure 5.2 on page 21.

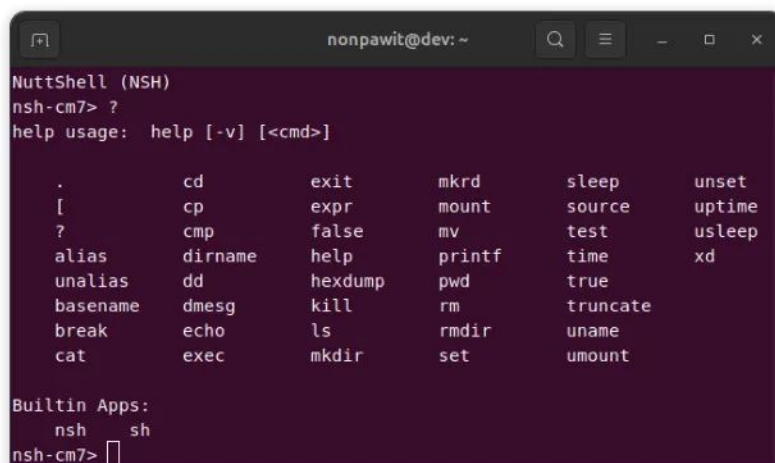


```
NuttShell (NSH)
nsh-cm4> ?
help usage:  help [-v] [<cmd>]

.          cd          exit          mkrd          sleep          unset
[          cp          expr          mount         source         uptime
?          cmp          false         mv            test           usleep
alias      dirname     help        printf        time           xd
unalias    dd           hexdump     pwd           true
basename   dmesg          kill        rm            truncate
break      echo          ls          rmdir         uname
cat        exec          mkdir       set           umount

Builtin Apps:
    nsh      sh
nsh-cm4> 
```

Figure 5.1: NSH CM4 Shell



```
NuttShell (NSH)
nsh-cm7> ?
help usage:  help [-v] [<cmd>]

.          cd          exit          mkrd          sleep          unset
[          cp          expr          mount         source         uptime
?          cmp          false         mv            test           usleep
alias      dirname     help        printf        time           xd
unalias    dd           hexdump     pwd           true
basename   dmesg          kill        rm            truncate
break      echo          ls          rmdir         uname
cat        exec          mkdir       set           umount

Builtin Apps:
    nsh      sh
nsh-cm7> 
```

Figure 5.2: NSH CM7 Shell

## 6 OpenAMP Structure

### 6.1 Overview of OpenAMP

OpenAMP (Open Asymmetric Multi-Processing) is a framework designed to facilitate communication and resource sharing between multiple processors or cores in a system. It enables the development of applications that can run across different cores, typically in heterogeneous multi-core systems, where different cores may be running different operating systems or instances of the same operating system. (OpenAMP Project 2024)

The primary goal of OpenAMP is to simplify inter-core communication by providing standardized APIs and mechanisms, allowing developers to focus on application logic rather than the complexities of multi-core synchronization and messaging. (OpenAMP Project 2024)

### 6.2 Structure of OpenAMP

#### 6.2.1 Core Components

- **RemoteProc (Remote Processor Framework):** Manages the life cycle of remote processors, including booting, shutting down, and loading firmware. It is responsible for initializing communication channels between cores. (OpenAMP Project 2024)
- **RPMMsg (Remote Processor Messaging):** A messaging protocol built on top of Virtio, enabling high-level communication between cores. It provides a simple API for sending and receiving messages between processors. (OpenAMP Project 2024)
- **Virtio:** A standardized abstraction layer that facilitates communication between virtualized systems. In the context of OpenAMP, Virtio is used to establish communication channels over which RPMMsg messages are exchanged. (OpenAMP Project 2024)

### 6.2.2 OpenAMP Software Stack

The OpenAMP software stack is layered to separate the application logic from the underlying hardware specifics. The stack typically consists of:

- **Application Layer:** Where user-defined processes run, utilizing the OpenAMP APIs to communicate with remote cores. (OpenAMP Project 2024)
- **Middleware:** Includes RPMsg and Virtio, which handle the communication protocols and virtual channels. (OpenAMP Project 2024)
- **Hardware Abstraction Layer (HAL):** Manages the interaction with the physical hardware, providing a consistent interface for the middleware to operate across different platforms. (OpenAMP Project 2024)

## 6.3 Importance in Multi-Core Systems

OpenAMP is particularly valuable in embedded systems like the STM32H745I-DISCO, where it enables efficient communication between the Cortex<sup>®</sup>-M4 and Cortex<sup>®</sup>-M7 cores. By abstracting the complexities of inter-core communication, OpenAMP allows developers to focus on optimizing performance and functionality in multi-core applications. (OpenAMP Project 2024)

## 7 RPTun

RPTun (Remote Processor Tunneling) enables seamless remote tunneling between processors in a multi-core system. On the STM32H745I-DISCO, integrating RPTun with NuttX facilitates communication between the Cortex<sup>®</sup>-M7 and Cortex<sup>®</sup>-M4 cores, allowing access to both cores' shells via a single USB interface. This chapter details the steps required to configure and deploy RPTun, enabling efficient remote tunneling and streamlined core management within the dual-core NuttX environment.

### 7.1 Create New Configuration

To begin integrating RPTun with the NuttX environment on the STM32H745I-DISCO board, the first step involves creating distinct configurations for the Cortex<sup>®</sup>-M7 and Cortex<sup>®</sup>-M4 cores. This is achieved by duplicating the existing `nsh_cm4` and `nsh_cm7` configurations, which serve as the foundational configurations for each core.

1. **Duplicate the Default Configuration:** Start by copying the 'nsh\_cm4' and 'nsh\_cm7' configuration to create separate configurations for each core:

```
cp -r boards/arm/stm32h7/stm32h745i-disco/
  configs/nsh_cm4 boards/arm/stm32h7/
  stm32h745i-disco/configs/nsh_cm4_rptun
cp -r boards/arm/stm32h7/stm32h745i-disco/
  configs/nsh_cm7 boards/arm/stm32h7/
  stm32h745i-disco/configs/nsh_cm7_rptun
```

This approach ensures that the new configurations are tailored specifically for enabling RPTun functionality, setting the stage for subsequent configuration and integration steps.

## 7.2 Cortex<sup>®</sup>-M4 Configuration

To properly configure the Cortex<sup>®</sup>-M4 core for integration with RPTun, several specific adjustments need to be made to the NuttX configuration files. These adjustments ensure that the Cortex<sup>®</sup>-M4 core operates independently while enabling effective communication with the Cortex<sup>®</sup>-M7 core through RPTun.

1. **Enable Simple Address:** The configuration will include simple address environment by adding the following line.

```
CONFIG_DEV_SIMPLE_ADDRENV=y
```

2. **Enable Architecture-Specific Initialization:** To ensure the correct initialization of the Cortex<sup>®</sup>-M4 core, include the architecture initialization option:

```
CONFIG_NSH_ARCHINIT=y
```

3. **Enable RPTun and RPTun:** Enabling RPTun framework and associated messaging protocols by adding the following lines.

```
CONFIG_RPMSG_PING=y
CONFIG_RPMSG_UART=y
CONFIG_RPMSG_UART_CONSOLE=y
CONFIG_RPTUN=y
```

4. **Disable UART7:** By enabling RPTun, Cortex<sup>®</sup>-M4 shell can be accessed via RPTun instead of UART7. Remove the following line:

```
CONFIG_STM32H7_UART7=y
CONFIG_UART7_SERIAL_CONSOLE=y
```

These configuration changes prepare the Cortex<sup>®</sup>-M4 core for remote processor tunneling, ensuring that it can efficiently communicate with the Cortex<sup>®</sup>-M7 core while being accessed through a unified USB interface.

## 7.3 Cortex<sup>®</sup>-M7 Configuration

To effectively integrate RPTun with the Cortex<sup>®</sup>-M7 core, specific configurations must be implemented to ensure robust and independent operation while enabling communication with the Cortex<sup>®</sup>-M4 core.

1. **Enable Simple Address:** Begin by enabling the simple address environment, which simplifies the memory addressing required by the core.

```
CONFIG_DEV_SIMPLE_ADDRENV=y
```

2. **Enable Specific Architecture Init:** This setting ensures that the Cortex<sup>®</sup>-M7 core is properly initialized during the boot process:

```
CONFIG_NSH_ARCHINIT=y
```

3. **Set MPU Region to 8:** Configure the Memory Protection Unit (MPU) to manage up to eight distinct memory regions, enhancing the security and reliability of memory operations.

```
CONFIG_ARM_MPU_NREGIONS=8
```

4. **Enable OpenAMP Cache:** Activate the OpenAMP cache to improve the performance of inter-core communications by caching messages.

```
CONFIG_OPENAMP_CACHE=y
```

5. **Enable RPSmsg and RPTun:** These configurations enable the RPSmsg protocol and the RPTun framework, both of which are essential for facilitating efficient communication between the two cores.

```
CONFIG_RPSMSG_PING=y
```

```
CONFIG_RPSMSG_UART=y
```

```
CONFIG_RPTUN=y
```

6. **Enable CU Minimal Serial Terminal:** To allow access to the Cortex<sup>®</sup>-M4 via the '/texttt/dev/ttyproxy' terminal, enable the minimal serial terminal functionality.

```
CONFIG_SYSTEM_CUTERM=y
```

These configurations ensure that the Cortex<sup>®</sup>-M7 core is fully prepared to operate within the RPTun framework, enabling effective communication and resource sharing with the Cortex<sup>®</sup>-M4 core, while maintaining the performance and security required for high-priority tasks.

## 7.4 RPMsg Initialization

To enable RPMsg communication between the Cortex<sup>®</sup>-M4 and Cortex<sup>®</sup>-M7 cores, it is necessary to configure the 'stm32\_bringup.c' file within the NuttX environment. The specific code required to initialize RPMsg for UART should be inserted directly into the 'stm32\_bringup.c' file, which is located in the following directory:

```
nuttX/boards/arm/stm32h7/stm32h745i-disco/src/
  stm32_bringup.c
```

The code snippet can be found in Appendix .4.

The code initializes the RPMsg UART interface, facilitating communication between the two cores through a virtual serial port. This configuration is critical for enabling the remote tunneling of the NSH shell via RPTun, thereby allowing for streamlined core management through a single USB interface.

## 7.5 Building and Flashing

After configuring the NuttX environment for both the Cortex<sup>®</sup>-M4 and Cortex<sup>®</sup>-M7 cores, the next steps involve building the NuttX instances and flashing them onto the STM32H745I-DISCO board.

### 7.5.1 Building the NuttX Instances

To generate the NuttX operating system binaries for each core"

1. **Building the NuttX Instance for the Cortex<sup>®</sup>-M4 Core:** Navigate to the nuttx directory and execute the following commands to create and compile the NuttX instance tailored for the Cortex<sup>®</sup>-M4 core:

```
cmake -B build_cm4_rptun -DBOARD_CONFIG=
  stm32h745i-disco:nsh_cm4_rptun -GNinja
cmake --build build_cm4_rptun
```

2. **Building the NuttX Instance for the Cortex<sup>®</sup>-M7 Core:** Similarly, execute the following commands to compile the NuttX instance for the Cortex<sup>®</sup>-M7 core:

```
cmake -B build_cm7_rptun -DBOARD_CONFIG=
      stm32h745i-disco:nsh_cm7_rptun -GNinja
cmake --build build_cm7_rptun
```

By utilizing CMake along with the Ninja build system, these commands efficiently compile the NuttX code into distinct binaries for each core, reflecting their specific configurations.

## 7.5.2 Flashing HEX Files to STM32H745I-DISCO

Once the NuttX instances have been successfully built, the next critical step is to flash the generated '.hex' files onto the STM32H745I-DISCO board using STM32CubeProgrammer, starting with the Cortex<sup>®</sup>-M7 core, followed by the Cortex<sup>®</sup>-M4 core, ensuring successful completion and avoiding memory conflicts.

## 7.5.3 Verifying Output

Post-flashing, it is crucial to verify that each NuttX instance is functioning as intended.

1. **Connect to Serial Consoles:** Open a terminal for Cortex<sup>®</sup>-M7 core.

```
screen /dev/ttyACM0 115200
```

2. **Check for Device Appearance:** Verify that the RPTun device appears in the device list by running:

```
ls dev/
```

If the setup is correct, '/dev/ttyproxy' should be listed among the devices, as can be seen in figure 7.1 on page 29.

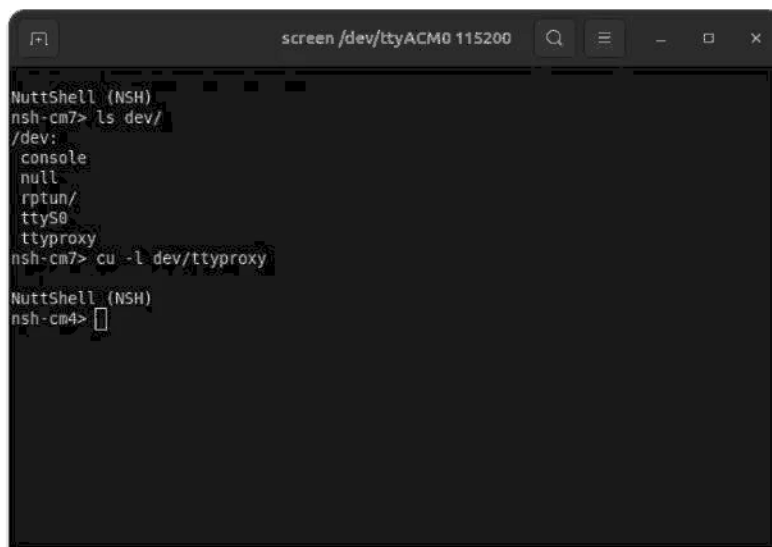
3. **Tunnel to Cortex<sup>®</sup>-M4 NSH Shell:** By running the following command, NuttX will tunnel the shell to NSH-CM4 shell. As can be seen in figure 7.2 on page 29.

```
cu -l dev/ttyproxy
```



```
screen /dev/ttyACM0 115200
NuttShell (NSH)
nsh-cm7> ls dev/
/dev:
console
null
rptun/
ttyS0
ttyproxy
nsh-cm7>
```

Figure 7.1: NSH RPTun Device List



```
screen /dev/ttyACM0 115200
NuttShell (NSH)
nsh-cm7> ls dev/
/dev:
console
null
rptun/
ttyS0
ttyproxy
nsh-cm7> cu -l dev/ttyproxy
NuttShell (NSH)
nsh-cm4>
```

Figure 7.2: NSH CM4 Shell Tunneled

## 8 RPMsg Ping-Pong

Testing inter-core communication between the Cortex<sup>®</sup>-M4 and Cortex<sup>®</sup>-M7 cores is essential for understanding the RPMsg protocol. The RPMsg Ping-Pong application provides a practical method to validate and observe the message exchange between the two cores, offering insights into the protocol's performance and reliability within the STM32H745I-DISCO environment.

### 8.1 Creating Custom NuttX Application

To implement the Ping-Pong test application, it is necessary to establish a new application directory within the NuttX framework. This directory should be created in the following path:

```
apps/custom_apps
```

The process involves the following steps:

1. **Create the CMakeLists.txt File:** To ensure that the NuttX OS can detect and integrate the new application directory, you need to create a 'CMakeLists.txt' file within the new application directory. Add the following lines to include the current subdirectory in the NuttX application directory.

```
nuttx_add_subdirectory()  
nuttx_generate_kconfig(MENUESC "Custom Apps")
```

2. **Create the Application Directory:** Within the custom application directory, create a new folder to house the Ping-Pong application files. NuttX requires a minimum of three essential files for the application to be recognized and enabled: 'CMakeLists.txt', 'Kconfig', and the main application file.
3. **Develop the Main Application File:** In the development phase, the main application file was named 'rpmmsg\_test\_ping.c'. This file contains the code that registers a callback for RPMsg, handling both device creation and destruction

callbacks. Once the RMPMsg device is created, an endpoint callback is established for each received data packet, which processes and compares the messages. Additionally, it checks if 'SEND\_PING' is enabled via menuconfig; if so, it sends the ping message. The full code for the main application file is provided in Appendix .6.

4. **Configure the Kconfig File:** The Kconfig file is used to configure the 'menuconfig' display. It includes the application name, dependencies, tooltips, and a boolean option to enable or disable the ping sending feature. The full code for the 'Kconfig' is provided in Appendix .5.
5. **Set Up the CMakeLists.txt File:** Finally, ensure that the 'CMakeLists.txt' file is correctly set up to add the application to the NuttX directory and enable it. Detailed instructions for setting up this file can be found in Appendix .7.

## 8.2 Enabling, Building, and Flashing Application

### 8.2.1 Enabling Ping Pong Application

To enable the STM32H745I-DISCO board to run the Ping-Pong application, the application must be enabled within 'menuconfig'. This can be done by navigating to:

```
Application Configuration > Custom Apps > Rpmmsg
ping pong example
```

For the Cortex<sup>®</sup>-M4 core, ensure that the 'Send Ping buffer' option is enabled.

### 8.2.2 Building

To build the application, run the following commands:

```
cmake --build build/cm4_rptun
cmake --build build/cm7_rptun
```

### 8.2.3 Flashing

Once the NuttX instances have been successfully built, the next critical step is to flash the generated '.hex' files onto the STM32H745I-DISCO board using STM32CubeProgrammer,

starting with the Cortex<sup>®</sup>-M7 core, followed by the Cortex<sup>®</sup>-M4 core, ensuring successful completion and avoiding memory conflicts.

### 8.2.4 Verifying Output

After flashing, it is crucial to verify that each NuttX instance is functioning as intended:

1. **Connect to Serial Consoles:** Open a terminal for Cortex<sup>®</sup>-M7 core using the command.

```
screen /dev/ttyACM0 115200
```

2. **Running Ping Pong Application on Cortex<sup>®</sup>-M7 Core:** Execute the following command to run the Ping-Pong application.

```
rpmsg_test_ping
```

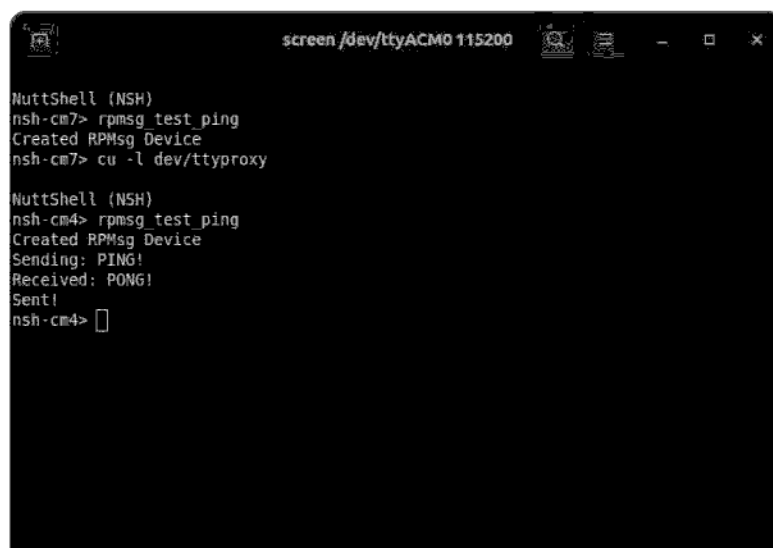
3. **Tunnel to Cortex<sup>®</sup>-M4 NSH Shell:** By running the following command, NuttX will tunnel the shell to NSH-CM4 shell.

```
cu -l dev/ttyproxy
```

4. **Running Ping Pong Application on Cortex<sup>®</sup>-M4 Core:** Execute the Ping-Pong application on the Cortex<sup>®</sup>-M4 core by running.

```
rpmsg_test_ping
```

The results of the application can be seen in figure 8.1 on page 33, which illustrates the successful execution of the RPMsg Ping-Pong test.



```
screen /dev/ttyACM0 115200

NuttShell (NSH)
nsh-cm7> rpmsg_test_ping
Created RPMsg Device
nsh-cm7> cu -l dev/ttyproxy

NuttShell (NSH)
nsh-cm4> rpmsg_test_ping
Created RPMsg Device
Sending: PING!
Received: PONG!
Sent!
nsh-cm4> 
```

Figure 8.1: RPMsg Ping Test

## .1 Cortex<sup>®</sup>-M7 Linker Script Condition

```
#include <nutttx/config.h>

#ifndef CONFIG_STM32H7_CORTEXM4_ENABLED
MEMORY
{
    itcm   (rwx) : ORIGIN = 0x00000000, LENGTH = 64K
    flash (rx)  : ORIGIN = 0x08000000, LENGTH = 2048K
    dtcm1  (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
    dtcm2  (rwx) : ORIGIN = 0x20010000, LENGTH = 64K
    sram   (rwx) : ORIGIN = 0x24000000, LENGTH = 512K
    sram1  (rwx) : ORIGIN = 0x30000000, LENGTH = 128K
    sram2  (rwx) : ORIGIN = 0x30020000, LENGTH = 128K
    sram3  (rwx) : ORIGIN = 0x30040000, LENGTH = 32K
    sram4  (rwx) : ORIGIN = 0x38000000, LENGTH = 64K
    bbram  (rwx) : ORIGIN = 0x38800000, LENGTH = 4K
}
#else
MEMORY
{
    itcm   (rwx) : ORIGIN = 0x00000000, LENGTH = 64K
    flash (rx)  : ORIGIN = 0x08000000, LENGTH =
        CONFIG_STM32H7_CORTEXM7_FLASH_SIZE
    dtcm1  (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
    dtcm2  (rwx) : ORIGIN = 0x20010000, LENGTH = 64K
    sram   (rwx) : ORIGIN = 0x24000000, LENGTH = 512K
    shmem  (rwx) : ORIGIN = 0x30040000, LENGTH = 32K
    sram4  (rwx) : ORIGIN = 0x38000000, LENGTH = 64K
    bbram  (rwx) : ORIGIN = 0x38800000, LENGTH = 4K
}
#endif
```

## .2 Cortex<sup>®</sup>-M7 Linker Script Shared Memory

```
.shmem :
{
    . = ALIGN(4);
    *(.shmem);
    KEEP(*(.shmem))
} > shmem
```

## .3 Cortex<sup>®</sup>-M4 Linker Script

```
#include <nutttx/config.h>

#define STM32M4_FLASH_START (0x08000000 +
    CONFIG_STM32H7_CORTEXM7_FLASH_SIZE)

MEMORY
{
    flash (rx) :
        ORIGIN = STM32M4_FLASH_START,
        LENGTH = (2048K -
            CONFIG_STM32H7_CORTEXM7_FLASH_SIZE)

    sram (rwx) : ORIGIN = 0x10000000, LENGTH = 128K
        + 128K
    shmem (rwx) : ORIGIN = 0x10040000, LENGTH = 32K
}

OUTPUT_ARCH(arm)
EXTERN(_vectors)
ENTRY(_stext)
SECTIONS
{
    .text :
    {
        _stext = ABSOLUTE(.);
        *(.vectors)
    }
```

```

*(.text .text.*)
*(.fixup)
*(.gnu.warning)
*(.rodata .rodata.*)
*(.gnu.linkonce.t.*)
*(.glue_7)
*(.glue_7t)
*(.got)
*(.gcc_except_table)
*(.gnu.linkonce.r.*)
_etext = ABSOLUTE(.);
} > flash

.init_section :
{
    _sinit = ABSOLUTE(.);
    *(.init_array .init_array.*)
    _einit = ABSOLUTE(.);
} > flash

.ARM.extab :
{
    *(.ARM.extab*)
} > flash

__exidx_start = ABSOLUTE(.);
.ARM.exidx :
{
    *(.ARM.exidx*)
} > flash
__exidx_end = ABSOLUTE(.);

_eronly = ABSOLUTE(.);

.data :
{
    _sdata = ABSOLUTE(.);
    *(.data .data.*)
    *(.gnu.linkonce.d.*)
    CONSTRUCTORS

```

```

    . = ALIGN(4);
    _edata = ABSOLUTE(.);
} > sram AT > flash

.bss :
{
    _sbss = ABSOLUTE(.);
    *(.bss .bss.*)
    *(.gnu.linkonce.b.*)
    *(COMMON)
    . = ALIGN(4);
    _ebss = ABSOLUTE(.);
} > sram

.stab 0 : { *(.stab) }
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }
.debug_abbrev 0 : { *(.debug_abbrev) }
.debug_info 0 : { *(.debug_info) }
.debug_line 0 : { *(.debug_line) }
.debug_pubnames 0 : { *(.debug_pubnames) }
.debug_aranges 0 : { *(.debug_aranges) }
}

```

## .4 RPMsg Initialize

```

#ifdef CONFIG_RPTUN
#   include "stm32_rptun.h"
#endif

#ifdef CONFIG_RPMMSG_UART
#   include <nutttx/serial/uart_rpmsg.h>
#endif

```

```

#ifdef CONFIG_RPMSG_UART
void rpmsg_serialinit(void)
{
#ifdef CONFIG_ARCH_CHIP_STM32H7_CORTEXM7
    uart_rpmsg_init("cm4", "proxy", 4096, false);
#endif

#ifdef CONFIG_ARCH_CHIP_STM32H7_CORTEXM4
#ifdef CONFIG_RPMSG_UART_CONSOLE
    uart_rpmsg_init("cm7", "proxy", 4096, true);
#else
    uart_rpmsg_init("cm7", "proxy", 4096, false);
#endif
#endif
}
#endif

#ifdef CONFIG_RPTUN
#ifdef CONFIG_ARCH_CHIP_STM32H7_CORTEXM7
    stm32_rptun_init("cm4");
#else
    stm32_rptun_init("cm7");
#endif
#endif

```

## .5 Ping-Pong Kconfig

```

config RPMSG\_TEST\_PING\_PONG
    tristate "Rpmsg ping pong example"
    depends on RPMSG
    default n
    --help--
        Enable the rpmsg ping pong example

if RPMSG_TEST_PING_PONG

config RPMSG_SEND_PING
    bool "Send Ping buffer"

```

```

    default no

endif # RPMSG_TEST_PING_PONG

```

## .6 Ping-Pong C File

```

#include <nutttx/config.h>

#include <nutttx/rpmsg/rpmsg.h>
#include <nutttx/kmalloc.h>

#include <stdio.h>
#include <sys/param.h>

#define CUSTOM_EPT_NAME          "ping_ept"
#define CUSTOM_RPMSG_PING_SEND   1
#define CUSTOM_RPMSG_PING_ACK    2

begin_packed_struct struct rpmsg_msg_s
{
    uint32_t cmd;
    uint32_t len;
    uint64_t cookie;
}end_packed_struct;

static int ping_rpmsg_ept_cb(FAR struct
    rpmsg_endpoint *ept,
                                FAR void *data,
                                size_t len,
                                uint32_t src,
                                FAR void *priv)
{
    FAR struct rpmsg_msg_s *msg = data;
    FAR sem_t *sem = (FAR sem_t *) (uintptr_t) msg->
        cookie;

    if (msg->cmd == CUSTOM_RPMSG_PING_SEND)
    {

```

```

        msg->cmd = CUSTOM_RPMSG_PING_ACK;
        rpmsg_send(ept, msg, len);
    }
    else if (msg->cmd == CUSTOM_RPMSG_PING_ACK)
    {
        printf("Received: PONG!\n");
        nxsem_post(sem);
    }

    return 0;
}

static void ping_rpmsg_device_created(FAR struct
    rpmsg_device *rdev,
                                     FAR void *
                                     priv)
{
    printf("Created RPMsg Device\n");
    rpmsg_create_ept(priv, rdev, CUSTOM_EPT_NAME,
                    RPMSG_ADDR_ANY, RPMSG_ADDR_ANY,
                    ping_rpmsg_ept_cb, NULL);
}

static void ping_rpmsg_device_destroy(FAR struct
    rpmsg_device *rdev,
                                     FAR void *
                                     priv)
{
    printf("Destroyed RPMsg Device\n");
    rpmsg_destroy_ept(priv);
}

static int rpmsg_ping_once(FAR struct
    rpmsg_endpoint *ept,
                        int len, uint32_t *
                        buf_len)
{
    FAR struct rpmsg_msg_s *msg;
    int ret;

```

```

msg = rpmsg_get_tx_payload_buffer(ept, buf_len,
    true);
if (!msg)
{
    return -ENOMEM;
}

len = MAX(len, sizeof(struct rpmsg_msg_s));
len = MIN(len, *buf_len);

sem_t sem;

msg->cmd      = CUSTOM_RPMSG_PING_SEND;
msg->len      = len;
msg->cookie   = (uintptr_t)&sem;

nxsem_init(&sem, 0, 0);

ret = rpmsg_send_nocopy(ept, msg, len);
if (ret >= 0)
{
    nxsem_wait_uninterruptible(&sem);
}

nxsem_destroy(&sem);

return ret;
}

int main(int argc, char *argv[])
{
    FAR struct rpmsg_endpoint *rpmsg_ping_ept;
    int ret = -ENOMEM;

    rpmsg_ping_ept = kmm_zalloc(sizeof(struct
        rpmsg_endpoint));
    if (!rpmsg_ping_ept)
    {
        return ret;
    }

```

```

    ret = rpmsg_register_callback(rpmsg_ping_ept,
                                   ping_rpmsg_device_created,
                                   ,
                                   ping_rpmsg_device_destroy,
                                   ,
                                   NULL,
                                   NULL);

#ifdef CONFIG_RPMSG_SEND_PING
    uint32_t buf_len = 0;
    printf("Sending: PING!\n");
    rpmsg_ping_once(rpmsg_ping_ept, 0, &buf_len);
    printf("Sent!\n");
#endif

    return ret;
}

```

## .7 Ping-Pong CmakeList.txt

```

if(CONFIG_RPMSG_TEST_PING_PONG)
    nuttx_add_application(
        NAME
        rpmsg_test_ping
        PRIORITY
        ${CONFIG_RPMSG_TEST_PING_PONGPRIORITY}
        STACKSIZE
        ${CONFIG_DEFAULT_TASK_STACKSIZE}
        MODULE
        ${CONFIG_RPMSG_TEST_PING_PONG}
        SRCS
        rpmsg_test_ping.c)
endif()

```

## Declaration concerning the final project

I herewith affirm that I have drafted the submitted dissertation independently and have not used any other sources or resources than those declared therein. Sources quoted literally or substantively have been cited according to the acknowledged rules of scientific publication. Furthermore, I declare that the dissertation has not been submitted elsewhere. I have read and understood the explanatory note on fraud in examination procedures at the University of Applied Sciences Augsburg. I affirm that the submitted dissertation does not include any plagiarism or any texts or figures created by other persons acting on my instructions.

Augsburg, December 13, 2024

---

Nonpawit Ekburanawat

# List of Figures

|     |                                              |    |
|-----|----------------------------------------------|----|
| 2.1 | Apache Software Foundation . . . . .         | 4  |
| 3.1 | STM32H745I-DISCO Development Board . . . . . | 9  |
| 4.1 | NuttX NSH Console . . . . .                  | 13 |
| 5.1 | NSH CM4 Shell . . . . .                      | 21 |
| 5.2 | NSH CM7 Shell . . . . .                      | 21 |
| 7.1 | NSH RPTun Device List . . . . .              | 29 |
| 7.2 | NSH CM4 Shell Tunneled . . . . .             | 29 |
| 8.1 | RPMsg Ping Test . . . . .                    | 33 |

## List of Tables

# References

Apache Software Foundation (2024). *NuttX RTOS Documentation*. URL: <https://nuttx.apache.org/> (cit. on pp. 4, 6, 7, 12).

OpenAMP Project (2024). *OpenAMP Documentation*. URL: <https://openamp.readthedocs.io/> (cit. on pp. 22, 23).

STMicroelectronics (2024). *Discovery kits with STM32H745XI and STM32H750XB Microcontrollers* (cit. on pp. 8–10).

# Stichwortverzeichnis

Grafik, 4, 9, 13, 21, 29, 33

jpg, 4, 9, 13, 21, 29, 33