AN5617

Application note

Introduction to inter-processor communications for STM32H745/755 and STM32H747/757 MCUs

Introduction

High performance STM32 microcontrollers release constraints on software architecture and open the door to more advanced software solutions. Advanced software applications require that independent components run simultaneously. Microcontrollers of the STM32H745/755 and STM32H747/757 lines feature an asymmetric dual-core architecture. Thus, processing parallelism is guaranteed with two CPUs capable of running different payloads. Nevertheless, there are often tasks that need to communicate with one another in order to share information and ensure correct processing. For these reasons, the inter-processor communication (IPC) layers are needed to link data dependent tasks.

To ease core-to-core interactions and reduce time to market, STM32CubeH7 proposes a set of standard middleware that implements the inter-processor communication channel (IPCC) between the Arm® Cortex®-M7 and the Arm®Cortex®-M4. This application note provides an overview of the dual-core communication technique. It introduces the inter-processor communication channels such as OpenAMP, RPMsg, and FreeRTOS™ as well as the message buffer and custom communication mechanism. It also provides a detailed flowchart with a snippet code example to describe how to use OpenAMP and FreeRTOS™ to create a communication channel between cores.

1 General information

This document applies to STM32H745/755 and STM32H747/757 lines Arm®-based devices.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

The following table present a non-exhaustive list of terms and acronyms used in this document.

Table 1. Terms and accronyms

Term or acronym	Definition
AMP	Asymmetric multi-processing
API	Application programming interface
AXISRAM	Advanced extensible interface SRAM
BKSRAM	Backup SRAM
CMSIS	Cortex® microcontroller software interface standard
CPU	Central processing unit
DTCM	Data tightly coupled memory
EXTI	External interrupt
FreeRTOS	Free real-time operating system
HSEM	Hardware semaphore free interrupt
IPC	Inter-processor communication
IPCC	Inter-processor communication channel
ITCM	Instruction tightly coupled memory
MDMA	Master direct memory access
MPU	Memory protection unit
MW	Middleware
NVIC	Nested vectored interrupt controller
OpenAMP	Open asymmetric multi-processing
RPMsg	Remote processor messaging
RTOS	Real-time operating system
SEV	Send-event instruction
SRAM	Static random access memory
TCM	Tightly moupled memory
TXEV	Transmit event

2 Dual-core communication

When dealing with STM32H745/755 and STM32H747/757 devices dual-core architecture in order to design an application that requires parallel execution, a communication and synchronization mechanism must be applied to ensure that interaction between the two cores occur without unexpected behaviors. This section focus on inter-processor communication techniques and presents some use case examples.

Note: For more details about dual-core architecture of these devices, refer to application note STM32H745/755 and STM32H747/757 lines dual-core architecture (AN5557), available from www.st.com.

2.1 Typical use cases requiring IPCC

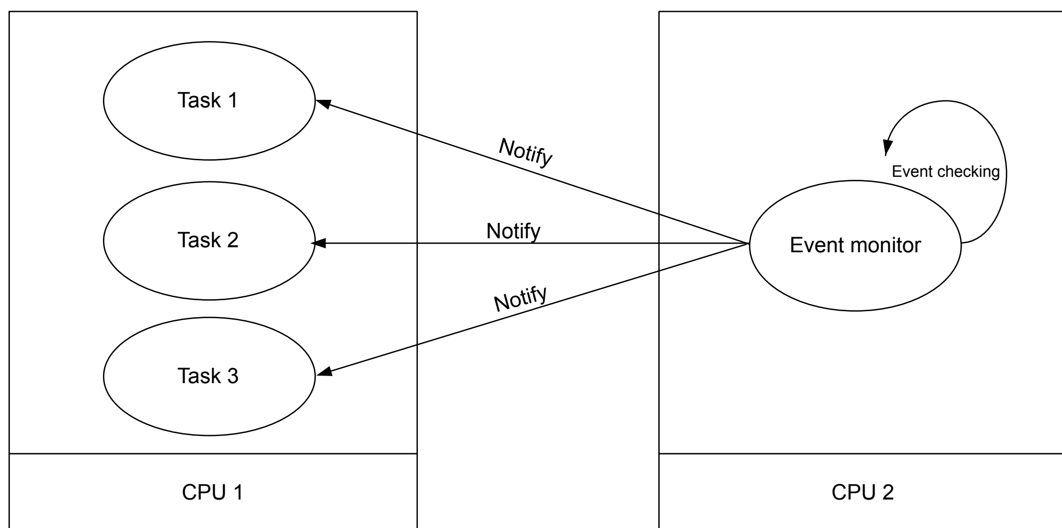
Communication channels are needed in many application examples. This section present a non exhaustive list of inter-processing communication channel examples: event notification, asking for remote service and payload processing.

2.1.1 Event notification

The *event notification* can be used to synchronize the execution of a task on both cores, especially when data processing pipelining is needed. The figure below illustrates a typical use case where one core can run the event monitoring process and sends a notification to wake up the second core to start the requested processing. A practical example of this case is a speech activation pattern detection via CPU2 user interface and its command execution implementation using CPU1.

The notification mechanism can also control application life cycles on CPU1. For example, when no more data are available to process, IPCC can be used to send the right event, allowing to switch CPU1 to low-power mode. To guarantee a safe system state (close opened files, purge log/buffers) before reset execution, CPU1 can be notified upon a reset command reception on CPU2.

Figure 1. Event notification diagram

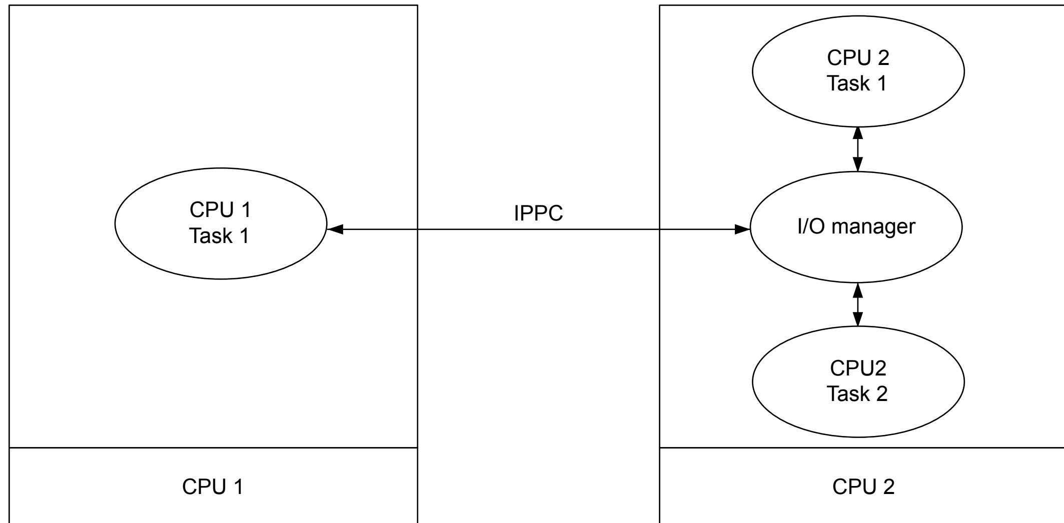


2.1.2 Asking for remote service

In order to reduce concurrency and driver code duplication (including stacks) when using peripherals, some services can be implemented on one of the CPUs. This action helps to reduce code size, and other concerns related to resource sharing. Once a task needs this service, it can make a request via the inter-processor communication channel. Some examples of services are file system management and serial communication interface.

The figure below presents a conventional topology for a shared I/O resource manager (running on CPU2) and access abstraction using communication channels. The application task (CPU Task1) on CPU1 can request CPU2 services and do not have to run a second instance of I/O manager. This implementation reduces the resources allocated to CPU1 and the I/O manager task looks like it is running on the same CPU.

Figure 2. **Remote service request diagram**



2.1.3 Payload processing

A functional split can setup CPU1 to perform computation intensive tasks while real-time tasks (such as sensor acquisition and control) would be located in the second core. This action permits to satisfy an application requirements in term of performance and power.

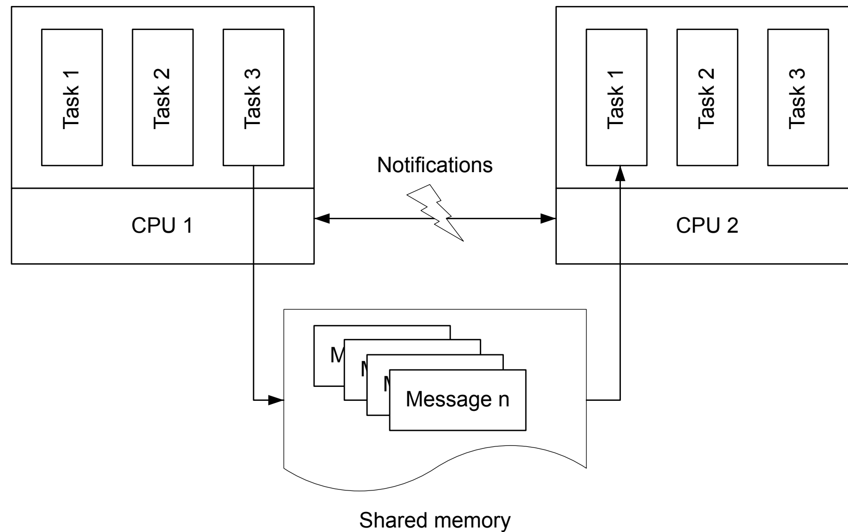
The overall power consumption is reduced by placing CPU1 in low-power mode while CPU2 is preprocessing data (data acquisition, handle connectivity) before waking up CPU1 to run intensive data computation algorithms.

A communication channel can be used to pass processing directives (such as data size, data type or processing type) from CPU2 to CPU1.

2.2 Working techniques

The basic environment required to implement a communication between two CPUs is a shared memory area. As illustrated in th next figure, this technique is used to exchange data and notifications in order to synchronize the access to a shared area.

Figure 3. Inter-processor communication topology



2.2.1 Shared memory

The first step to implement a shared memory area is to choose a memory area that is available and accessible by both cores. STM32H745/755 and STM32H747/757 devices implement a symmetric memory-mapping between the two cores. As presented in the next table, this architecture allows to have ~82% of the SRAM directly accessible by both CPUs and to use it for data exchange.

Some practices imply to map exchange buffers in common power domain (always available to CPU) in order to maintain the shared memory accessible to a CPU while the other one is in low-power mode.

Table 2. STM32H745/755 and STM32H747/757 shared RAM resources

Core	D1 domain			D2 domain			D3 domain	
	ITCM	DTCM	AXISRAM	SRAM1	SRAM2	SRAM3	SRAM4	BKSRAM
Cortex®-M7	Yes		Yes (cacheable)					
Cortex®-M4	Indirect (only via MDMA)		Yes					

The D3 domain implements 64 Kbytes of SRAM and 4 Kbytes of back-up SRAM (BKSRAM). SRAM4 and BKSRAM are accessible to Cortex®-M7 and Cortex®-M4 and they remain available when D1 or D2 domains are in low-power mode. D3 RAM can be used as shared memory for message exchange.

The D1 and D2 RAMs (except TCMs) remain available to implement exchange buffers. The application power efficiency can be impacted as the SRAMs need to be allocated to both CPUs to remain available to read/write data when one of the CPUs is in CSleep or CStop mode.

Note: For more details about dual-core power management refer to application note STM32H747/757 advanced power management (AN5215), available at www.st.com.

When dealing with shared data in memory, it is important to consider caches and data coherency. The developer must define the adequate strategy for the application. Different implementation scenarios can be foreseen:

- Perform runtime cache maintenance using software through the CMSIS functions. Data-cache clears before notifying and using data from the CPU2.

- Use the memory protection unit (MPU) to set a shared memory region as non-cacheable to ensure data coherency and consistency.

Note: For more details about data coherency refer to application note Level 1 cache on STM32F7 Series and STM32H7 Series (AN4839), available at www.st.com.

Note: For more details about memory protection unit (MPU) refer to application note Managing memory protection unit in STM32 MCUs (AN4838), available at www.st.com.

2.2.2 Notification

Using notification or interrupt request from one core to the other helps to ensure consistency and to reduce polling time for new message.

The STM32H745/755 and STM32H747/757 dual-core devices offer plenty of solutions to implement the notification mechanism. To generate an interrupt on one side or the other, the software can use the hardware semaphore free interrupt (HSEM), the EXTI software interrupt and event registers (EXTI) or the CPU send-event instruction (SEV). HSEM, EXTI and SEV allow also to wake up the CPUs and their respective domains from low-power mode to handle the incoming messages.

3 STM32 inter-processor communication channels

This section presents general information about inter-processor communication libraries available with the STM32CubeH7 firmware: OpenAMP and RPmsg APIs and FreeRTOS™ message buffer and stream buffers APIs. It also presents some basic IPCC using STM32H745/755 and STM32H747/757 hardware resources.

3.1 OpenAMP and RPMsg

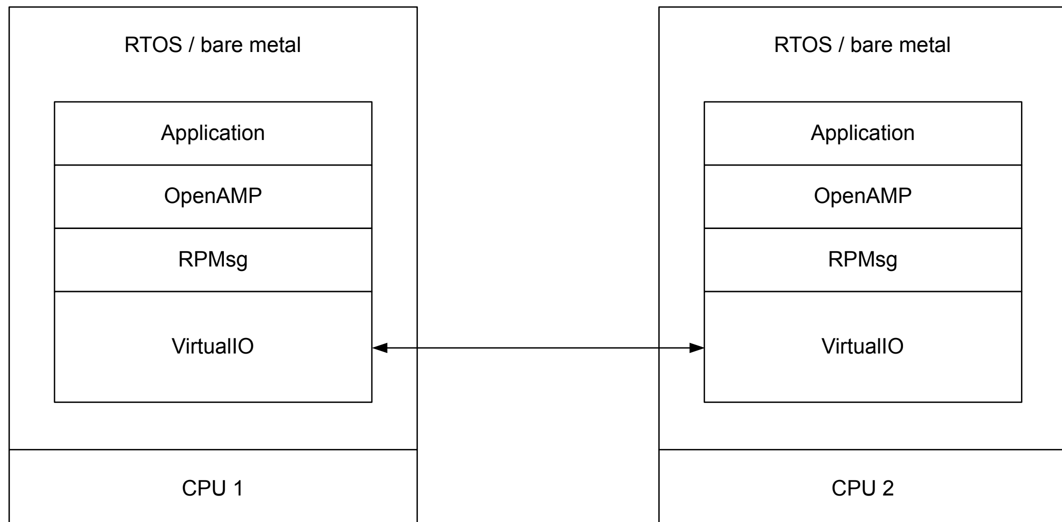
This section gives information about OpenAMP (open asymmetric multi-processing) and about RPMsg (remote processor messaging) libraries.

OpenAMP is a framework that provides the required software components to enable the development of applications for asymmetric multi-processing (AMP) systems. It standardizes the interactions between operating environments in a heterogeneous embedded system through open-source components such as remoteproc and RPMsg).

RPMsg is a component of the OpenAMP framework. It allows inter-processor communication between applications running on different CPUs. It is a virtual I/O-based messaging library that enables RTOS and bare metal applications on a master processor to interact with remote CPU firmware and communicate with them using standard APIs. Master and slave terminology is defined with OpenAMP framework.

The virtual I/O implements standards for the management of the shared memory. OpenAMP uses information published through the remote processor firmware resource table to allocate system resources and create a virtual I/O device. An example of RPMsg implementation in RTOS or bare metal environment is presented in the following figure.

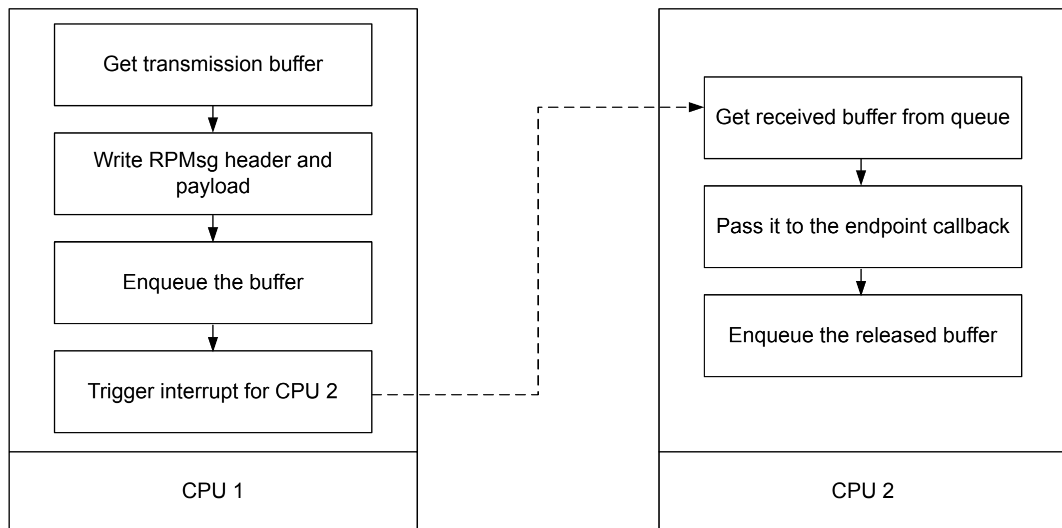
Figure 4. OpenAMP and RPMsg implementation layers



The sequence that is used to send message from CPU1 to CPU2 is the following:

CPU1 which is the master core sending data allocates buffers from shared memory used for transmission, writes RPMsg header and data payload to it. Finally, it enqueues the buffers in the ring buffer to make it available for CPU2. This sequence is illustrated below:

Figure 5. **CPU1 to CPU2 message passing flow**



3.2 FreeRTOS message buffer and stream buffer

Starting from version 10.0.0, FreeRTOS™ implements inter-processor communication APIs. the following sections describe message buffers and stream buffers APIs.

3.2.1 Message buffers

Message buffers allow variable-length discrete messages to be passed from an interrupt service routine to a task, or from one task to another. For example, messages of length 10, 20, and 123 bytes can all be written to, and read from, the same message buffer. A 10-byte message can only be read as a 10-byte message, not as individual bytes. Message buffers are built on top of the stream buffer implementation.

Message buffer requires a word to save the payload data size. The length is stored in a variable which is typically 4 bytes on a 32-bit architecture. Therefore, writing a 10-byte message into a message buffer consumes 14 bytes of buffer space. Likewise, writing a 100-byte message into a message buffer uses 104 bytes of buffer space.

3.2.2 Stream buffers

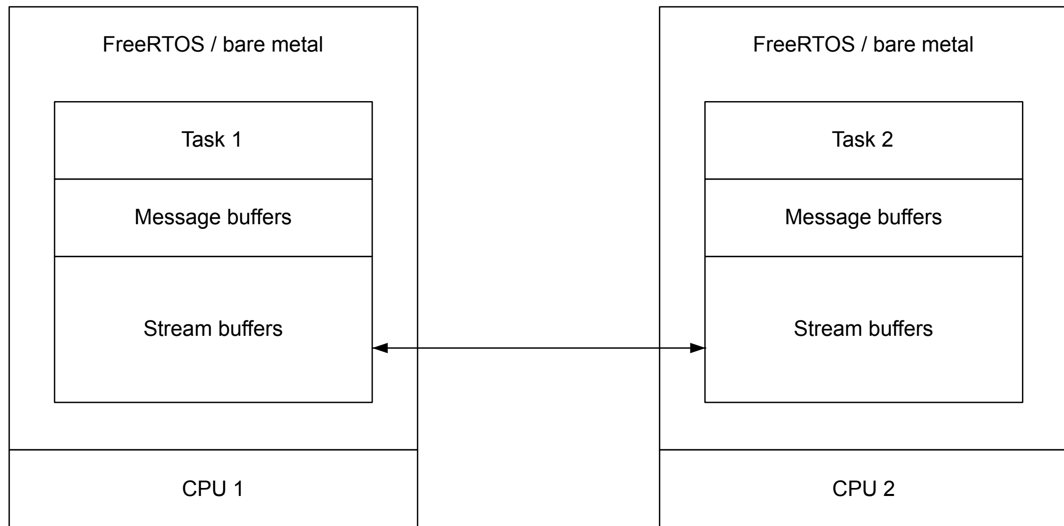
Stream buffers allow a stream of bytes to be passed from an interrupt service routine to a task, or from one task to another, as shown in the figure below. A byte-stream can be of arbitrary length and does not necessarily have a beginning or an end. Any number of bytes can be written at one time, and any number of bytes can be read at one time.

The stream buffer implementation uses speed and footprint optimized notifications. Therefore, calling a stream buffer API that places the calling task into the blocked state can change the notification state and value of the calling task .

Unlike most other FreeRTOS™ communications primitives, they are optimised for single-reader single-writer scenarios. It is not safe to have multiple writers or readers.

When passing data from one CPU to another on dual-core microcontroller, data is passed by copy: the data is copied into the buffer by the sender and copied out of the buffer by the reader.

Figure 6. Message buffer layers



3.3 Using STM32H745/755 and STM32H747/757 hardware resources

A lightweight inter-processor communication channel can be implemented using the available peripherals built into the STM32H745/755 or STM32H747/757 device. For example, interrupt lines can be used to signal events or to communicate the availability of data. Other peripherals such as DMA channels can be used to transfer data and generate an end of transfer notification from one side to the other. This solution is a customized solution and is not as generic as OpenAMP or FreeRTOS™ remote messaging implementation.

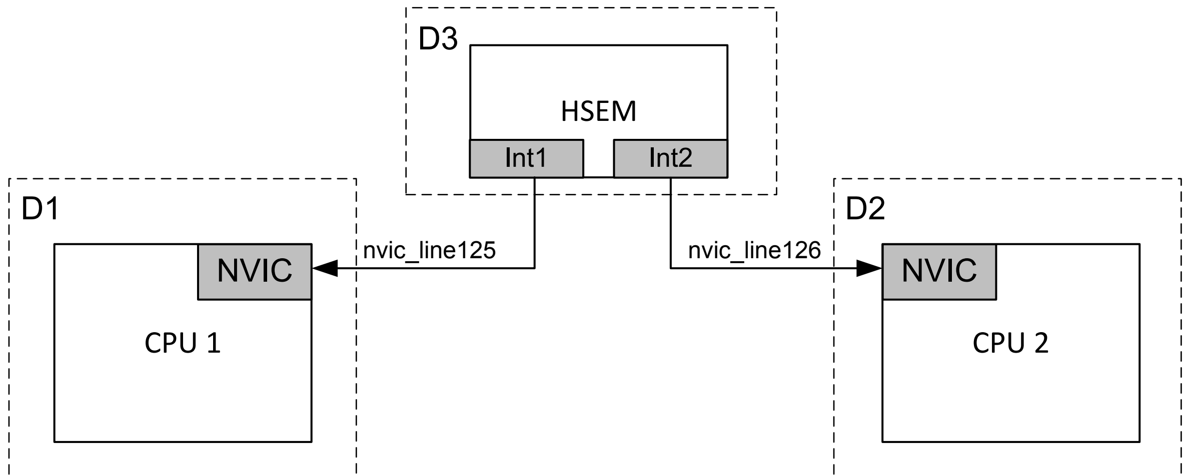
3.3.1 Hardware semaphore

Hardware semaphore peripherals implemented on the STM32 microcontrollers allow the implementation of notifications and event exchange between two CPUs (see figure below). Two interrupt lines are available from the HSEM peripheral to the CPU1 and the CPU2 interrupt controllers. The HSEM notification features can permit software developers to implement event exchanges between cores.

HSEM peripheral is also connected to the EXTI controller in the STM32H7 series products, this connection also allows to manage notification from CPUx to CPUy while the CPUy is in low-power mode.

HSEM can be used to convey simple notification event between CPUs or it can be used by firmware developers to indicate to tasks running on the other CPU that new data is available in a shared memory structure.

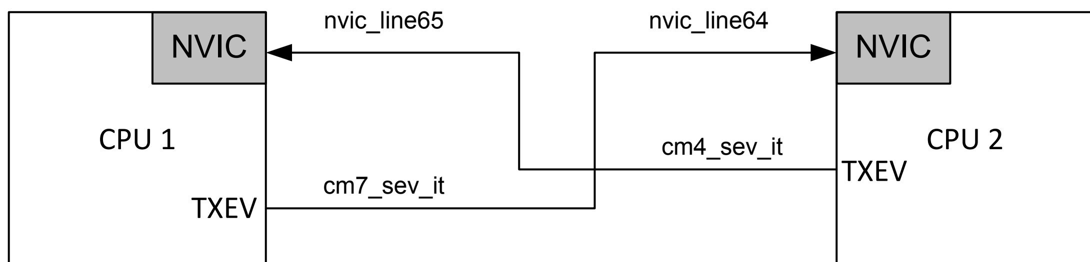
Figure 7. Hardware semaphore interrupt lines



3.3.2 EXTI controller and send-event instruction

The STM32H745/755 and STM32H747/757 lines devices architecture and internal connections allow software developers to implement a notification mechanism. CPU1 event output signal (TXEV) is connected to the EXTI controller which allows to trigger interrupts or events on CPU2. This connection is available in both directions; from CPU1 to CPU2 and from CPU2 to CPU1, as illustrated in the figure below. The event output-signal can be triggered by simply executing the SEV(send-event instruction) in the firmware. EXTI offers the capability to mask and unmask events and interrupts from the TXEV signal depending on the application needs.

Figure 8. SEV notification mechanism



Note: The TXEV from CPU1 pulse duration is equal to 512 AHB cycles, this duration needs to be taken in consideration by the firmware running on CPU2. The 512 AHB cycles pulse duration allows CPU2 to correctly latch the event whatever the clock ratio between both CPUs.

3.3.3 DMAs and MDMA

In addition to hardware semaphore peripherals and event out interconnections, DMAs or MDMA can help to implement communication between tasks running on CPU1 and tasks running on CPU2. The DMA memory to memory mode work as a transfer channel to copy data from one CPU SRAM buffer to a shared buffer (intended for CPU2), and the transfer-complete interrupt can be used to notify CPU2 that a new data was written into the shared buffer.

The advantage of using DMA in this method is that the data copy is performed by the DMA, thus offloading the CPU. With more than two DMA controllers available in the product, dedicated channels can be defined during application resource partitioning to perform inter-processor communication.

4 STM32CubeH7 examples

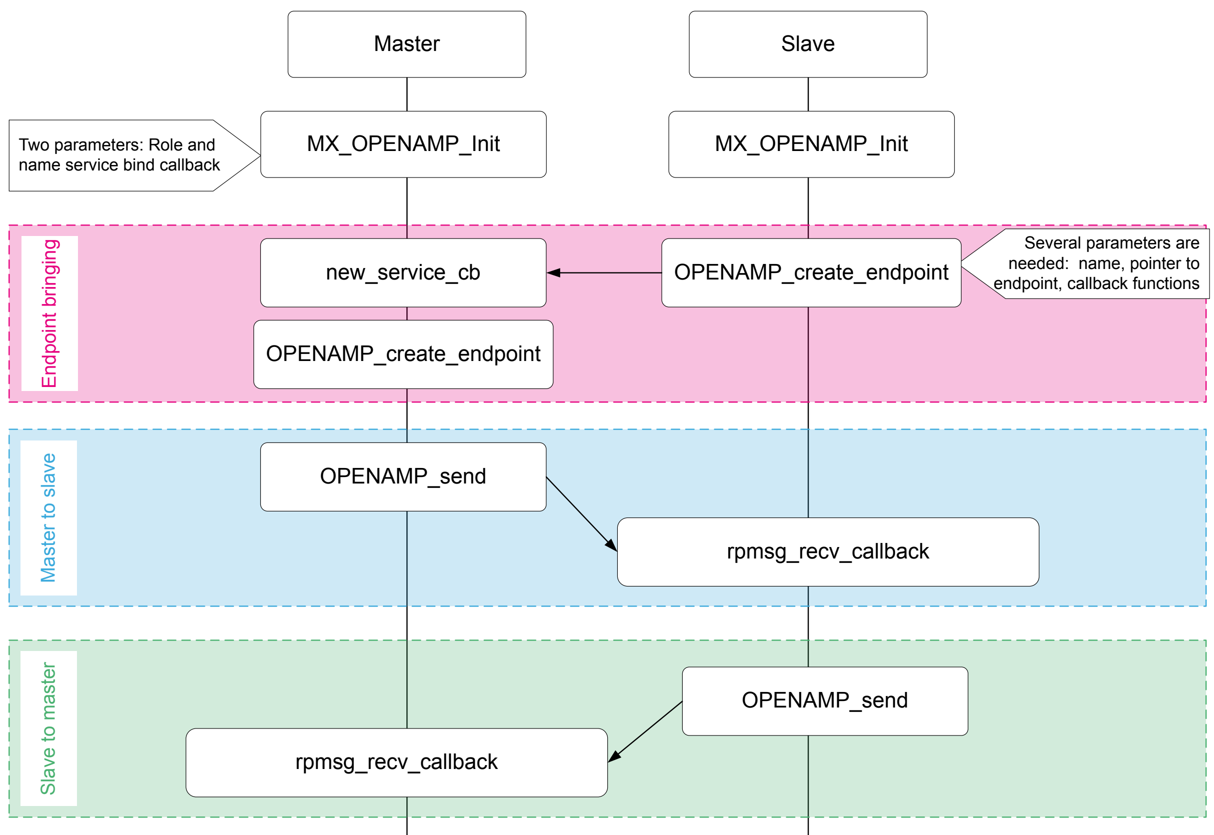
STM32CubeH7 packages include several examples for dual-core applications, including inter-processor communication using OpenAMP and FreeRTOS™ message buffer. The following sections describe some examples using OpenAMP and FreeRTOS™ and other using the hardware semaphore.

4.1 OpenAMP

The next figure shows an example that defines CPU1 as a master and CPU2 as a slave. After system initialization both CPUs run the MX_OPENAMP_Init() function, which is responsible for the OpenAMP framework initialization. CPU2 (slave) is responsible for endpoint creation, this is done using the OPENAMP_create_endpoint() API.

As soon as the endpoint is created, the master CPU (CPU1) is informed that a link is available. This notification is done through a callback passed as a parameter to MX_OPENAMP_Init() function. The master CPU is ready to complete endpoints binding. After successful binding, the master CPU sends its first data using the OPENAMP_send() function, this results in a notification on the slave side, where the slave rpmsg_recv_callback() is executed, and the data is copied to the application running on the CPU2 side. CPU2 is also able to send messages to the master CPU using the OPENAMP_send() function.

Figure 9. OpenAMP example



This figure shows a flowchart with snippet code for OpenAMP using hardware semaphore notification.

Figure 10. OpenAMP example using hardware semaphore notification

CM4

semaphore Notification enable
HAL_HSEM_ActivateNotification()

Inilitize the mailbox use notify the other core on new message
MAILBOX_Init()

Inilitize OpenAmp and libmetal libraries, then check the API return
status = MX_OPENAMP_Init()

No

HAL_OK

create an endpoint for rmpsg communication and check the return value of
status= OPENAMP_create_endpoint()

Status<0

Initialization error

End

Status>=0

Main loop

No

If mainloop is finished

Yes

Deinitialize OpenAMP
OPENAMP_DeInit()

End

Send a notification using semaphore from CM7 to CM4

Receive the massage from the remote CPU
OPENAMP_check_for_message

Send a data from CM4 to CM7
OPENAMP_send()

CM7

Define used semaphore
osSemaphoreDef()

Create the semaphore
osSemaphoreCreate()

Start scheduler
osKernelStart()

Initialize the mailbox use notify the other core on new message
MAILBOX_Init()

Initialize the rpmsg endpoint to set default addresses
rpmsg_init_ept()

Inilitize OpenAmp and libmetal libraries, then check the API return
status = MX_OPENAMP_Init()

No

Error

End

HAL_OK

Send a data from CM7 to CM4
OPENAMP_send()

Notification

OPENAMP_check_for_message

Save received message

Yes

Do you want to send new data?

No

Deinitialize OpenAMP
OPENAMP_DeInit()

Interrupt path

Principal path

The following tables describe the OpenAMP APIs and the main files presented above.

Table 3. OpenAMP APIs description

Function name	Description
MX_OPENAMP_Init	OpenAMP framework initialization
OPENAMP_create_endpoint	Endpoint creation
Notification callbacks	Different functions called when messages are received
OPENAMP_send	Send message
OPENAMP_check_for_message	Receive message

Table 4. Main files implementation description

File name	Description
main_cm7[.c/.h]	Main CM7 program that implements the example
main_cm4[.c/.h]	Main CM4 program that implements the example
stm32h7xx_it.c	CMx interrupt handlers
rsc_table[.c/.h]	Resource_table for OpenAMP
mbox_hsem[.c/.h]	HSEM mailbox interface implementation
openamp[.c/.h]	OpenAMP API wrapper interface
openamp_conf.h	Configuration file for OpenAMP MW

OpenAMP Linker

For OpenAMP some memory should be allocated to the shared resource table and shared buffer. This can be done in the linker file for both CPUs using the syntax presented in table below:

Table 5. OpenAMP Linker configuration

Tools	Linker configuration
IAR	define region RSC_TAB_region = mem:[from 0x38000000 to 0x380003FF]; place in RSC_TAB_region { readwrite section .resource_table }; define symbol __OPENAMP_region_start__ = BASE_ADDRESS; (0x38000400 for example) define symbol __OPENAMP_region_size__ = MEM_SIZE; (0xB000 as example) export symbol __OPENAMP_region_start__; export symbol __OPENAMP_region_size__;
MDK-ARM	LR_IROM1 .... { ... __OpenAMP_SHMEM__ 0x38000400 EMPTY 0x0000B000 {} ; Shared Memory area used by OpenAMP

GCC	MEMORY
	{
	...
	OPEN_AMP_SHMEM (xrw) : ORIGIN = 0x38000400, LENGTH = 63K
	}
	__OPENAMP_region_start__ = ORIGIN(OPEN_AMP_SHMEM);
	__OPENAMP_region_end__ = ORIGIN(OPEN_AMP_SHMEM) + LENGTH(OPEN_AMP_SHMEM);

Tools	Linker configuration
	using the LENGTH(OPEN_AMP_SHMEM) to set the SHM_SIZE lead to a crash thus we use the start and end address.

Note: Refer to any linker files provided in dual-core STM32CubeH7.

4.2 FreeRTOS message buffer

This example shows how to use FreeRTOS™ message buffers to pass data from one core to another. Each core has his own FreeRTOS™ instance.

Assumptions:

- The core that sends the data is the Cortex®-M7 referred to as core 1.
- The core that receives the data is the Cortex®-M4 referred to as core 2.

Example:

1. The task implemented by prvCore1Task() runs on core 1. Two instances of the task implemented by prvCore2Tasks() run on core 2.
2. prvCore1Task() sends messages via message buffers to both instances of prvCore2Tasks(), one message buffer per channel.
3. A third message buffer is used to pass the handle of the message buffer written to by core 1 to an interrupt service routine that is triggered by core 1 but executed on core 2.
4. Core to core communication via message buffer requires the message buffers to be at an address known to both cores within shared memory.

The following table describes the FreeRTOS message buffers APIs.

Table 6. FreeRTOS message buffers APIs

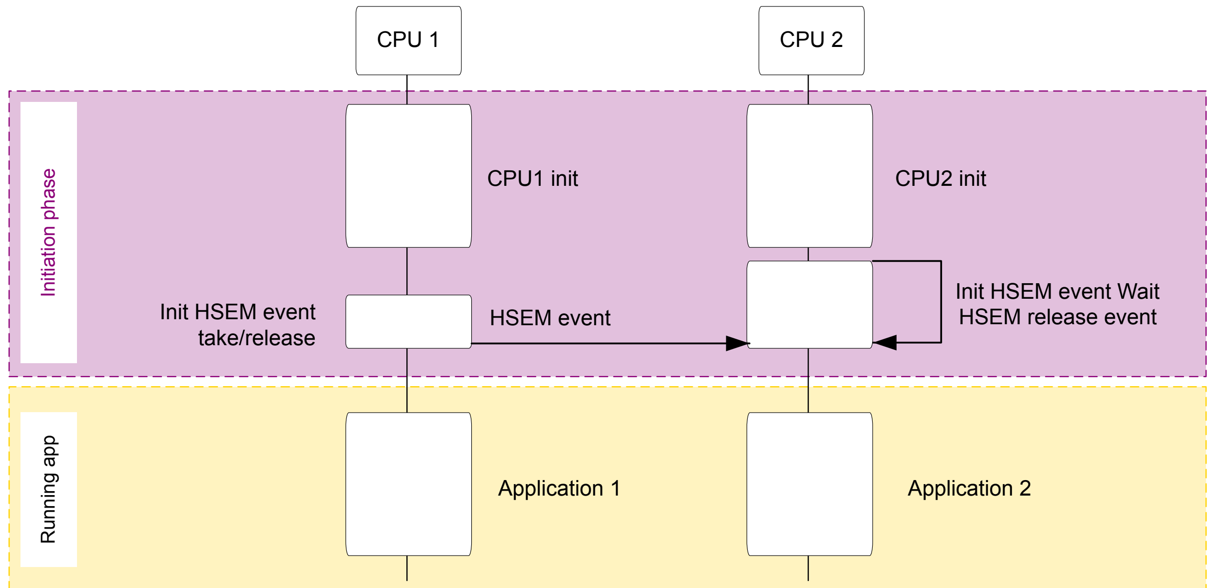
Function name	Description
xControlMessageBuffer	Create control message buffer
xDataMessageBuffers	Create data message buffer
xMessageBufferSend	Send message
xMessageBufferReceiveFromISR	Receive message
xMessageBufferSendCompletedFromISR	Send notification
sbSEND_COMPLETED	Send notification between cores

4.3 Using hardware semaphore

A basic example showing how to use the hardware semaphore to implement event transmission between CPUs is present in the project template.

The next figure shows the synchronization mechanism using the hardware semaphore. When both cores are programmed to boot in parallel, a synchronization is implemented within the template code to allow CPU2 to know that CPU1 has completed the system initialization from its side.

Figure 11. Synchronization mechanism using hardware semaphore



4.4 EXTI notification example

This section presents how to use the EXTI to implement event transmission between CPUs.

The first figure shows the synchronization mechanism using the EXTI and the second figure presents a flowchart with snippet code for EXTI interrupt notification.

Figure 12. Synchronization mechanism using EXTI

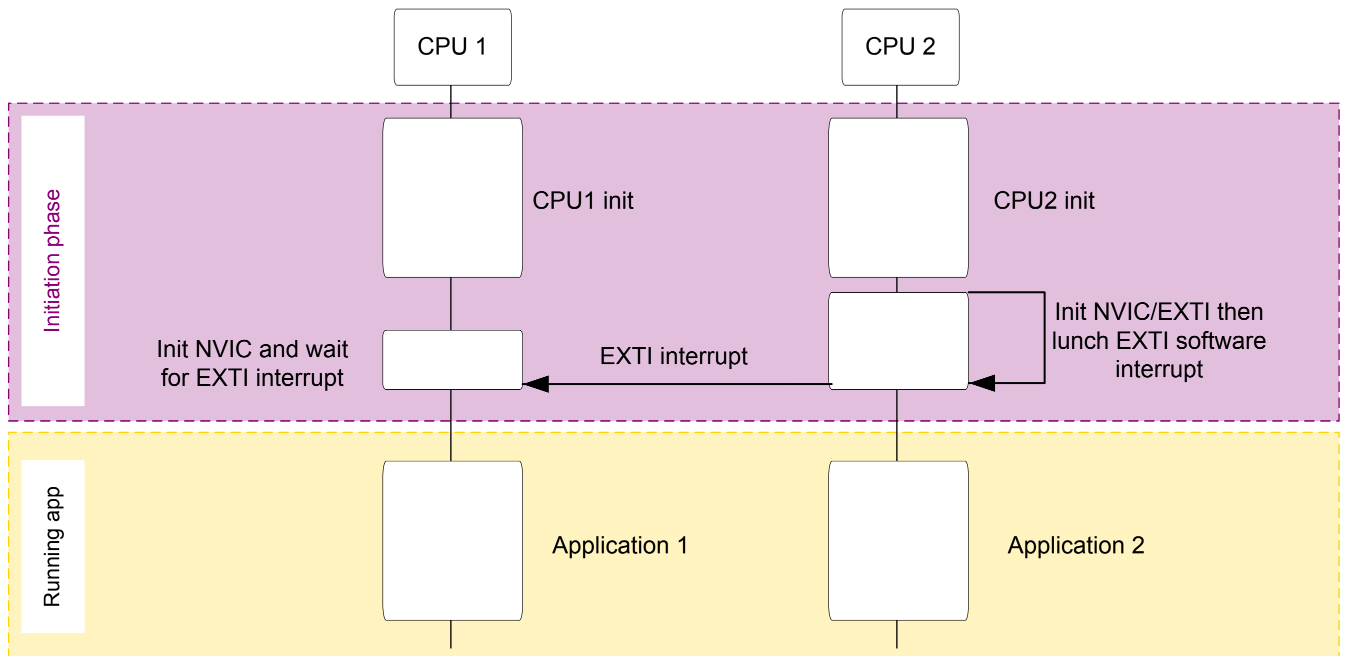
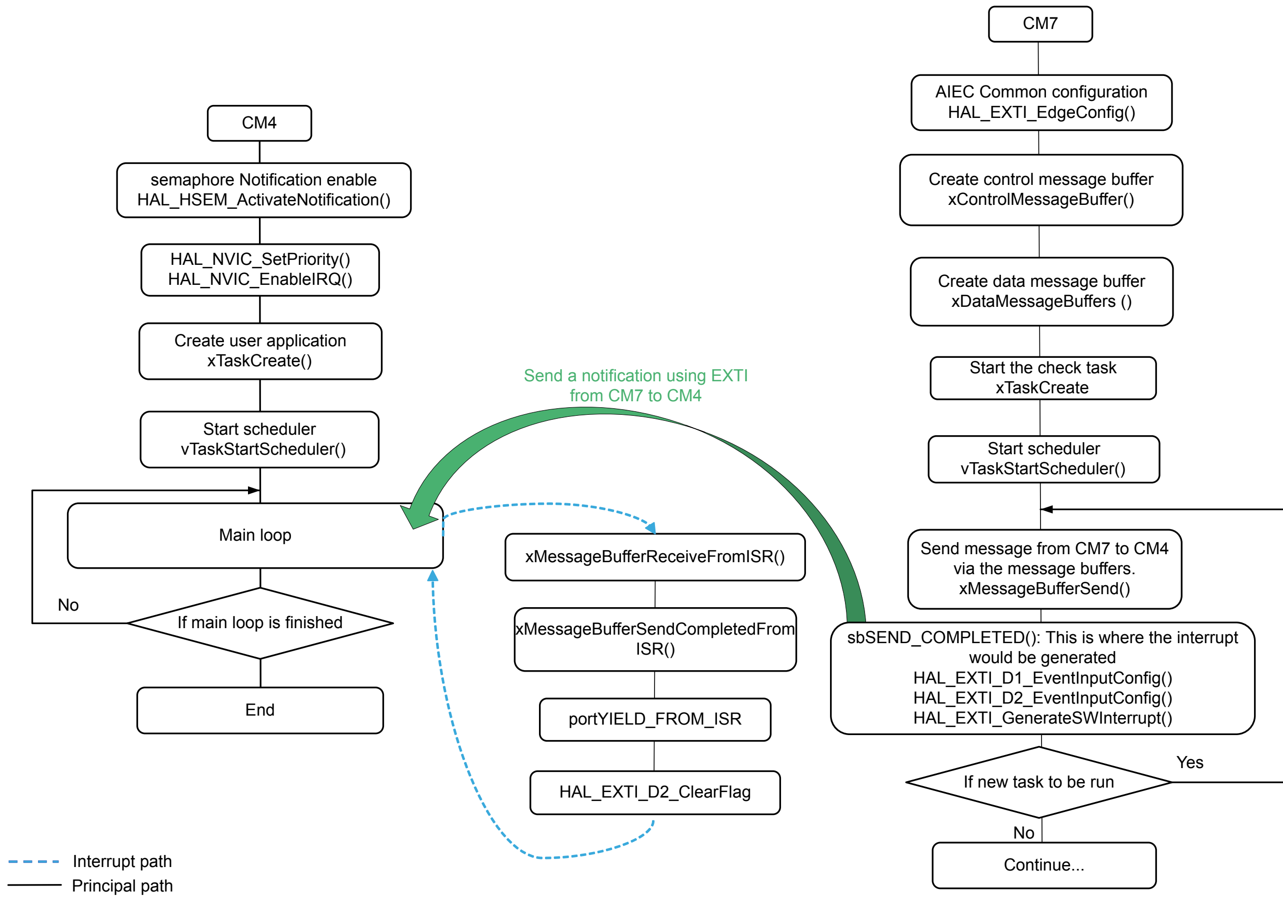


Figure 13. EXTI interrupt notification example



5 Conclusion

Multiple core microcontrollers enhance the possibility for developers to successfully solve complex problems. It is important to select the adequate inter-processor communication channel when separating the workload between the cores. Different solutions are possible, starting from standardized communication interfaces to application-specific mechanisms.

Revision history

Table 7. Document revision history

Date	Version	Changes
25-Feb-2021	1	Initial release.
28-Jan-2026	2	Updated document title.

Contents

1 General information 2
2 Dual-core communication 3
2.1 Typical use cases requiring IPCC 3
2.1.1 Event notification 3
2.1.2 Asking for remote service 3
2.1.3 Payload processing 4
2.2 Working techniques 5
2.2.1 Shared memory 5
2.2.2 Notification 6
3 STM32 inter-processor communication channels 7
3.1 OpenAMP and RPMsg 7
3.2 FreeRTOS message buffer and stream buffer 8
3.2.1 Message buffers 8
3.2.2 Stream buffers 8
3.3 Using STM32H745/755 and STM32H747/757 hardware resources 9
3.3.1 Hardware semaphore 9
3.3.2 EXTI controller and send-event instruction 10
3.3.3 DMAs and MDMA 10
4 STM32CubeH7 examples 12
4.1 OpenAMP 12
4.2 FreeRTOS message buffer 15
4.3 Using hardware semaphore 15
4.4 EXTI notification example 17
5 Conclusion 19
Revision history 20
List of tables 22
List of figures 23

List of tables

Table 1. Terms and accronyms 2
Table 2. STM32H745/755 and STM32H747/757 shared RAM resources 5
Table 3. OpenAMP APIs description 14
Table 4. Main files implementation description 14
Table 5. OpenAMP Linker configuration 14
Table 6. FreeRTOS message buffers APIs 15
Table 7. Document revision history 20

List of figures

Figure 1. Event notification diagram . . . 3
Figure 2. Remote service request diagram . . . 4
Figure 3. Inter-processor communication topology . . . 5
Figure 4. OpenAMP and RPMsg implementation layers . . . 7
Figure 5. CPU1 to CPU2 message passing flow . . . 8
Figure 6. Message buffer layers . . . 9
Figure 7. Hardware semaphore interrupt lines . . . 10
Figure 8. SEV notification mechanism . . . 10
Figure 9. OpenAMP example . . . 12
Figure 10. OpenAMP example using hardware semaphore notification . . . 13
Figure 11. Synchronization mechanism using hardware semaphore . . . 16
Figure 12. Synchronization mechanism using EXTI . . . 17
Figure 13. EXTI interrupt notification example . . . 18

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

The purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of the purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

If the purchasers identify an ST product that meets their functional and performance requirements but that is not designated for the purchasers' market segment, the purchasers shall contact ST for more information.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2026 STMicroelectronics – All rights reserved