AN14846

Boosting Application Performance with KW47 Dual-Core Architecture

Rev. 1.0 — 10 December 2025 Application note

Document information

Information	Content
Keywords	AN14846, KW47, Narrowband Unit (NBU)
Abstract	This application note describes how to use the dual-core architecture in the KW47 microcontroller to improve performance in generic embedded applications.

1 Introduction

This application note describes how to use the dual-core architecture in the KW47 microcontroller to improve performance in generic embedded applications.

The KW47 microcontroller integrates two Arm Cortex-M33 cores, one of which resides in the Narrowband Unit (NBU). While the NBU typically handles the radio subsystem, its general-purpose core enables broader application use. These core capabilities improve system responsiveness and support efficient multitasking. With this architecture, you can implement communication frameworks that coordinate workloads between the main core and the NBU.

This architecture supports flexible application design, including wireless applications, signal processing, and UI management. This note explains how to implement intercore communication and demonstrates a practical use case involving computation offload and display control.

2 KW47 SoC architecture

The KW47 microcontroller features a dual-core architecture that supports a wide range of wireless and general-purpose applications. The SoC integrates two Arm Cortex-M33 cores: one serves as the main processing unit, and the other resides in the NBU, which is part of the radio subsystem.

The NBU acts as a dedicated compute subsystem for the narrowband radio. It includes its own Cortex-M33 core and a set of peripherals that work alongside the Bluetooth unit to support the Bluetooth Low Energy (LE) protocol. These peripherals include timers, a Messaging Unit (MU), and a dedicated flash memory, all separate from the main SoC flash.

Although both cores share architecture, the main core includes additional features, such as Arm DSP and FPU extensions.

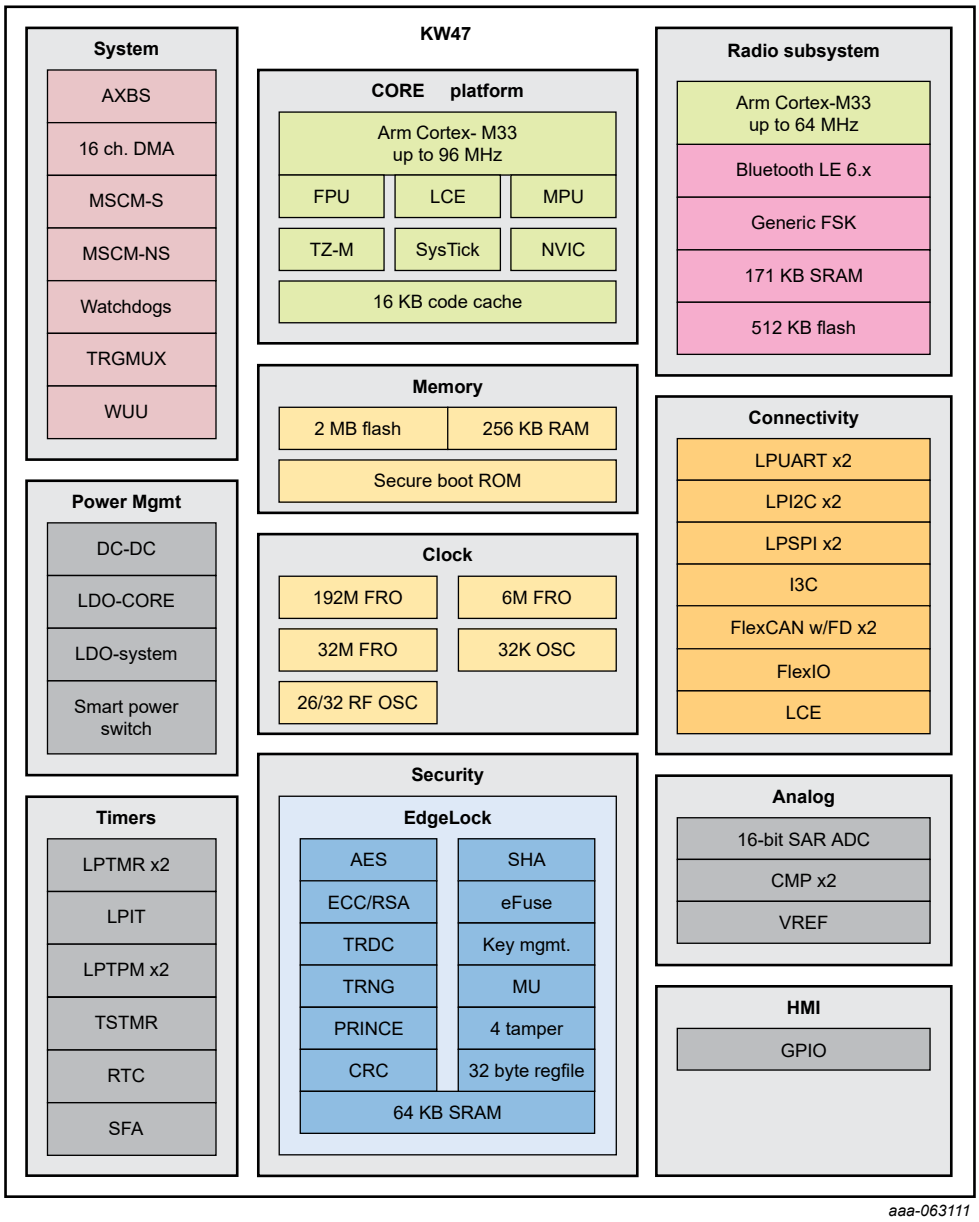

Figure 1. KW47 block diagram

2.1 Using the NBU as a secondary compute core

The NBU typically handles radio-related tasks, but its functionality extends beyond the radio subsystem. Since the NBU includes an Arm Cortex-M33 core, it executes instructions like a general-purpose processor. This capability allows developers to treat the NBU as a secondary computing unit that complements the main core.

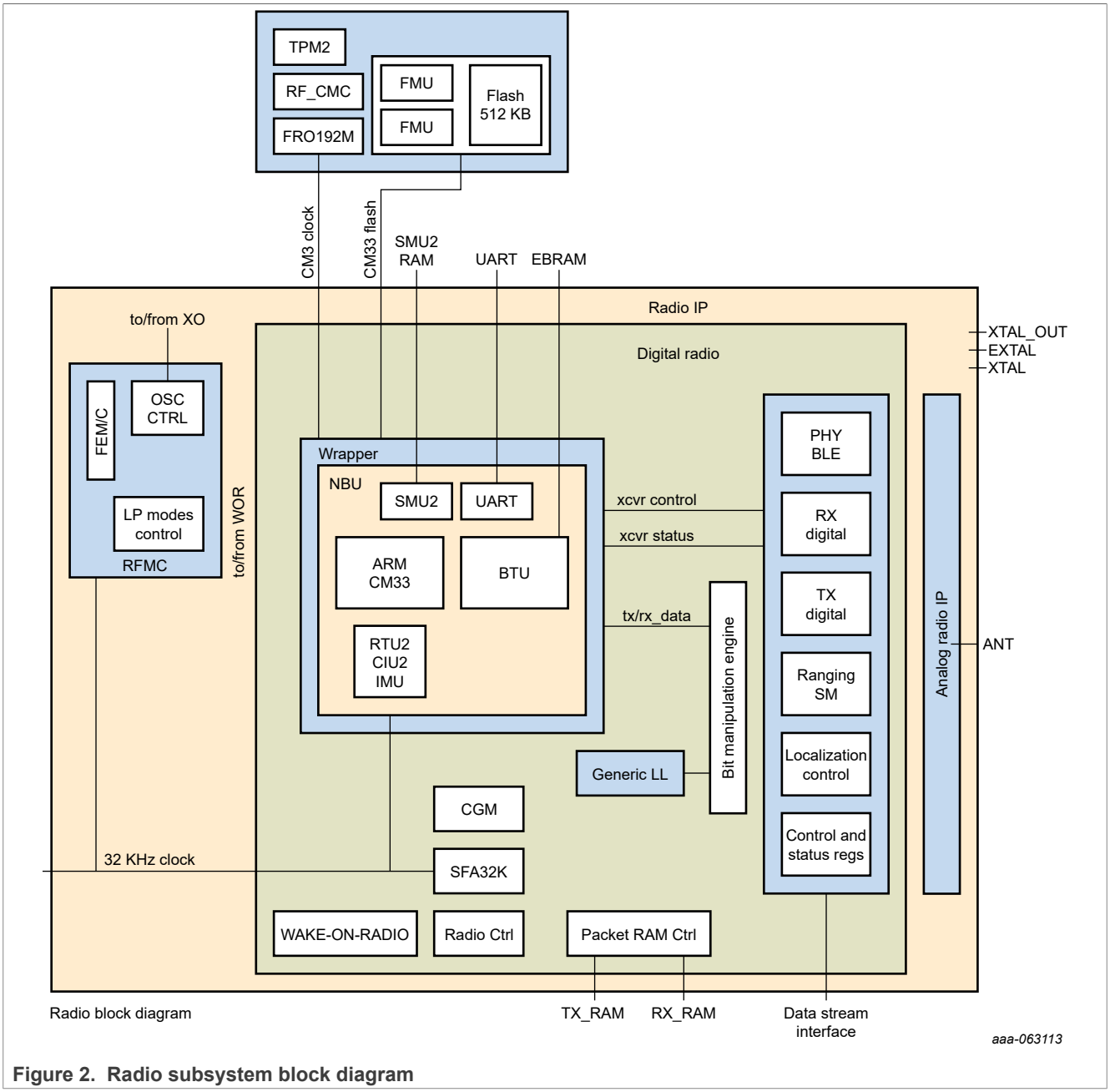

Figure 2. Radio subsystem block diagram

By offloading selected workloads to the NBU, you improve real-time performance and reduce processing bottlenecks on the main core. This approach enables more responsive and efficient application behavior across a wide range of use cases.

3 Practical use case: Fast Fourier Transform (FFT) computation

The FFT is a mathematical algorithm that converts a signal from the time domain into the frequency domain. This transformation helps to identify the frequency components present in a signal, making the FFT useful for audio analysis, vibration monitoring, and wireless signal processing.

The FFT is a resource-intensive operation for a typical microcontroller. It requires many CPU cycles and a great memory bandwidth, especially when processing large data sets or high-resolution signals. In applications where

the main core handles multiple tasks, such as display updates, user input, or protocol management, executing an FFT can introduce delays and degrade the overall performance.

Offloading the FFT computation to the NBU reduces the load of the main core. This approach improves responsiveness and maintains consistent performance across the system. The NBU acts as a general-purpose computing core that handles heavy processing tasks without interfering with time-critical operations on the main core.

This use case highlights the motivation behind using the KW47 dual-core architecture. By distributing workloads intelligently, you can build more efficient and responsive applications without compromising functionality.

4 Main core and NBU interface components

This section describes the hardware and software components that enable the communication between the main core and the NBU in KW47. Together, these elements support a scalable and reliable intercore interaction for both wireless and general-purpose applications.

Consider a Bluetooth LE application as an example. The Host Controller Interface (HCI) defines the communication protocol between the host (main core) and the radio controller on the NBU. It enables the host to send commands, receive events, and exchange data with the controller over a standardized interface.

In MCUXpresso SDK Bluetooth LE examples, the HCI operates using the Multicore SDK (MCSDK) along with KW47 peripherals and a shared SRAM region. This configuration enables the structured communication between the host and controller cores by combining software and hardware resources.

Although Bluetooth LE is a typical use case for KW47, the same Inter-Processor Communication (IPC) principles apply to any application that benefits from task separation or computing offload. You can use the NBU as a general-purpose processor, independent of the radio subsystem. By implementing custom intercore protocols over the MCSDK, applications can delegate the heavy or time-sensitive tasks to the NBU. This flexibility allows the microcontroller to support a wide range of use cases beyond wireless communication.

5 Dedicated IPC hardware

KW47 supports the dedicated hardware for the IPC between the main core and the NBU. The Message Unit (MU) peripheral enables the signaling through dedicated interrupts routed to both cores and includes a set of registers for passing control, values, and status flags.

In addition to the MU, both cores can access a shared section of SoC SRAM, known as SMU2. This memory region is addressable by both cores, allowing the data exchange. It can be used by software frameworks to implement structured communication.

While the MU and SMU2 offer the basic infrastructure for the IPC, the MCSDK enhances this setup with a robust software layer. The MCSDK enables reliable, scalable, and maintainable intercore communication beyond what raw hardware access can provide.

5.1 MU

The MU enables both cores in KW47 to exchange control signals, status flags, and data through a dedicated hardware interface. It supports intercore signaling using interrupts and register-based communication. Each core accesses its own set of MU-facing registers, which mirror the registers of the opposite core. This design ensures a reliable synchronization, even when the cores operate on independent clock domains.

KW47 implements a custom version of the MU, known as the Inter-CPU Message Unit (IMU), for radio subsystem interfacing. The IMU provides the same foundational capabilities as the standard MU but it supports the specific communication requirements between the main core and the NBU.

Both MU and IMU drivers are available as part of the MCUXpresso SDK, enabling you to integrate intercore communication seamlessly into your applications. These drivers abstract the hardware details and provide a consistent API for event signaling, message passing, and coordination between cores.

5.1.1 Functional description

The IMU enables communication and coordination between two cores, processor CPU1 and processor CPU2, by exchanging data, control, and status information. Each core can also use the IMU to wake the other core through interrupt signaling. The IMU registers are accessible to both cores through the CIU2 interface, which resides in the NBU domain. Both cores can address the CIU2 through memory-mapped interfaces:

- CIU2 NBU address: 0xA800_8000
- CIU2 main core address: 0x4894_8000

The following two instances of IMU reside within the CIU2:

- The IMU instance facing the main core is located at an offset of 0x1D4 from the CIU2 base address.
- The IMU instance facing the NBU core is located at an offset of 0x1E8 from the CIU2 base address.

Each core interacts with its own set of IMU-facing registers. These registers mirror the state of the opposite core's interface, enabling synchronized communication even when the cores operate on independent clock domains. The IMU drivers provided in the MCUXpresso SDK abstract these hardware details and offer APIs for intercore messaging.

The IMU works with the shared memory to support flexible messaging schemes. You can define your own messaging protocols. For example, a message might indicate that a data block of *n* words has been written at offset *x* or that a previously sent block has been read. The messaging logic remains independent of the shared memory structure and requires minimal software overhead.

The IMU provides symmetric messaging mechanisms, available on both sides of the interface:

- **FIFO buffer**: Each IMU instance includes a FIFO buffer that holds up to 16 32-bit words. This buffer stores messages pushed by the local core and read by the remote core.
- **WR_MSG register**: Writes a 32-bit word into the FIFO. This operation queues a message for the remote core to read.
- **RD_MSG Register**: Reads a 32-bit word from the FIFO. This operation retrieves a message previously sent by the remote core.
- **MSG_FIFO_STATUS Register**: Reports the status of the FIFO. It indicates whether the FIFO is full or empty and provides the count of messages currently stored.
- **MSG_FIFO_CNTL Register**: Provides control functions for the FIFO. It includes bits for flushing the FIFO, locking the FIFO to prevent writes, clearing interrupt flags from the remote core, and setting a watermark level during initialization.
- **RD_MSG_DBG Register**: Holds the last 32-bit word read by the remote core. This register allows the sender to verify the message consumption.

To ensure a proper synchronization:

- When a core writes to the remote peer, messages are pushed into the FIFO using WR_MSG. If the FIFO is locked and interrupt signaling is enabled on the remote core, the write operation triggers an interrupt.
- The interrupt vector number for this signaling is:
  - 14 for the NBU NVIC
  - 12 for the main core NVIC

5.1.2 IMU registers description

- CIU2 base address - main core (CPU1) address space: 4894_8000h

- CIU2 base address - NBU core (CPU2) address space: A800_8000h

Table 1. IMU registers - CIU2 base address - NBU core (CPU2) address space: A800_8000h

Offset	Register	Width (bits)
1D4h	CIU2_IMU_CPU1_WR_MSG_TO_CPU2	32
1D8h	CIU2_IMU_CPU1_RD_MSG_TO_CPU2	32
1DCh	CIU2_IMU_CPU1_CPU2_MSG_FIFO_STATUS	32
1E0h	CIU2_IMU_CPU1_CPU2_MSG_FIFO_CNTL	32
1E4h	CIU2_IMU_CPU2_RD_MSG_FROM_CPU1_VAL_DBG	32
1E8h	CIU2_IMU_CPU2_WR_MSG_TO_CPU1	32
1ECh	CIU2_IMU_CPU2_RD_MSG_TO_CPU1	32
1F0h	CIU2_IMU_CPU2_CPU1_MSG_FIFO_STATUS	32
1F4h	CIU2_IMU_CPU2_CPU1_MSG_FIFO_CNTL	32
1F8h	CIU2_IMU_CPU1_RD_MSG_FROM_CPU2_VAL_DBG	32

5.2 SMU2

SMU2 refers to a dedicated region of system SRAM used for intercore data exchange between the main core and the NBU. This memory is physically shared but logically mapped into each core's address space. SMU2 is part of a larger SRAM pool that the NBU uses for general-purpose data storage, called DMEM. Within this pool, a specific portion, designated as SMU, is accessible by both cores and reserved for interprocessor communication. This shared access enables data exchange. The remaining DMEM portion is used by the NBU for internal processing and storage.

SMU2 supports up to 160 kB of shared memory, accessible by both cores through distinct address mappings:

- NBU address range: 0xB000_0000 to 0xB002_7FFF
- Main core address start: 0x489C_0000 to 0x489E_7FFF

5.2.1 NBU SMU/DMEM multiplexing

The NBU SMU2 memory configuration is flexible and depends on the system-level allocation between SMU and DMEM. The SoC includes multiple SRAM blocks that can be dynamically assigned to either SMU or DMEM, based on application requirements. The default configuration allocates 80 kB to SMU and 80 kB to DMEM, but the total available memory can reach up to 160 kB for SMU when fully allocated.

The SRAM is organized into 4 × 8 kB, 4 × 16 kB, and 2 × 32 kB blocks. Each block can be independently assigned to either SMU or DMEM. This allocation is controlled by the RAM_MUX_CTRL[SMU_MEM_SEL] register. To modify this register, the application must first unlock access by writing 0x5 to the RAM_MUX_CTRL[UNLOCK] field.

Memory reassignment should only occur during system initialization. The application must ensure that there are no active transactions on SMU or DMEM before modifying the configuration. This restriction helps to prevent data corruption and ensures consistent memory behavior across both cores.

Table 2 shows the supported configurations. Any other configurations are not supported:

Table 2. RF SMU/DMEM valid multiplexing options

SMU_MEM_SEL[9:0]	SMU space (kB)	DMEM space (kB)
3E0h	80 kB	80 kB

Table 2. RF SMU/DMEM valid multiplexing options*...continued*

SMU_MEM_SEL[9:0]	SMU space (kB)	DMEM space (kB)
3F0h	64 kB	96 kB
3F8h	32 kB	128 kB
3FCh	16 kB	144 kB
3FEh	8 kB	152 kB
3FFh	-	160 kB
3C0h	96 kB	64 kB
380h	128 kB	32 kB
300h	144 kB	16 kB
200h	152 kB	8 kB
000h	160 kB	-

The RAM_MUX_CTRL register, used to configure the memory allocation between SMU and DMEM, is part of the RF Core Mode Controller (RF_CMC) module. This module is memory-mapped at base address 0x4898_3000, with RAM_MUX_CTRL at offset 0x18.

- **RAM_MUX_CTRL[UNLOCK]**: Writing this bit field to 3‘b101 unlocks the SMU_MEM_SEL for a write operation.
- **RAM_MUX_CTRL[SMU_MEM_SEL]**: Controls whether the SRAM blocks are attached to DMEM or SMU. See Table 2 for valid configuration options. Any values not listed in Table 2 are invalid.

5.2.2 SMU linker considerations

When configuring SMU2 for intercore communication, the linker files for both the main core and the NBU must reflect the actual memory size and address mapping defined by the SMU and DMEM allocation. The memory regions assigned to the SMU must match across both projects to ensure consistent access and avoid overlap with other memory sections. Any mismatch between the hardware configuration and linker definitions can lead to undefined behavior or data corruption. Align the linker memory segments with the active configuration set by RAM_MUX_CTRL[SMU_MEM_SEL], considering the total allocated size and base addresses visible to each core.

The shared memory start address is then exported from the linker file to the application.

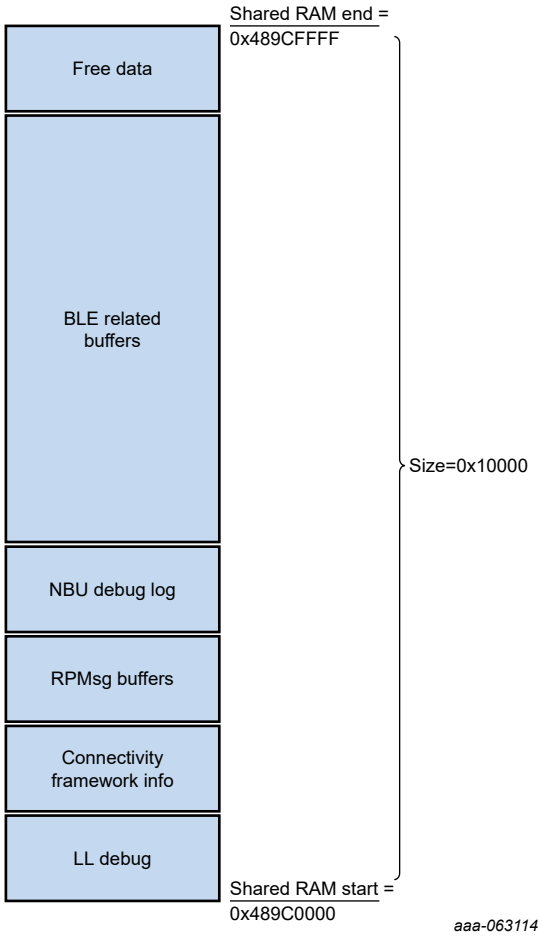

Figure 3. NBU shared RAM allocation example

5.3 IPC using low-level drivers

KW47 includes peripheral modules for intercore communication. The IMU peripheral provides a low-level IPC mechanism that allows both cores to exchange control values through dedicated registers and trigger interrupts for event signaling. This setup enables fast and lightweight communication using the MCUXpresso SDK IMU driver APIs.

While IMU and SMU2 are sufficient for basic IPC, building a robust messaging system requires structured software support. The MCSDK offers this capability and serves as the preferred MCUXpresso SDK messaging library. It enables the message exchange between multiple endpoints on each core using ring buffers in the shared memory. This design avoids locking mechanisms and supports asynchronous communication. RPMsg-Lite runs on top of the IMU and shared memory infrastructure, providing a scalable and maintainable solution for intercore messaging.

5.4 Multicore SDK

The MCSDK is a software development kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

Figure 4 highlights the layers and main software components of the MCSDK:

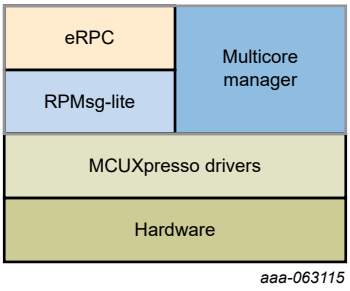

Figure 4. MCSDK architecture

The MCSDK consists of the following software components:

- Embedded Remote Procedure Call (eRPC): This component implements a transparent function call interface to remote services (running on a different core). Although the eRPC is part of the MCSDK, it is not in the scope of this document.
- Multicore Manager (MCMGR): This library maintains information about all cores and starts up secondary/auxiliary cores.
- Remote Processor Messaging - Lite (RPMsg-Lite): Interprocessor communication library.

5.4.1 MCMGR

The MCMGR software library provides several services for multicore systems. The main MCMGR features are as follows:

- Maintains information about all cores in system
- Secondary/auxiliary cores startup and shutdown
- Remote core monitoring and event handling

The MCMGR library is in the *mcuxsdk/middleware/multicore/mcmgr* folder. For detailed information about the MCMGR library, see the documentation available here.

5.4.2 Remote Processor Messaging Lite (RPMsg-Lite)

RPMsg-Lite is an open-source component developed by NXP Semiconductors and released under the BSD-compatible license. It is a lightweight implementation of the RPMsg protocol.

The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system. RPMsg-Lite offers a lightweight code size, simple APIs, and component modularity.

The main RPMsg protocol features are as follows:

- Shared memory interprocessor communication
- Virtual IO-based messaging bus
- Application-defined messages sent between endpoints
- Portable to different environments and platforms
- Available in Upstream Linux OS

The RPMsg-Lite implementation consists of three software components, with two considered optional. The core functionality resides in *rpmsg_lite.c*, which handles basic message transport and endpoint management. The optional components include *rpmsg_queue.c*, which adds support for blocking receive operations, and *rpmsg_ns.c*, which enables dynamic creation and deletion of named endpoints.

The underlying transport layer is implemented in *virtqueue.c*. This file defines the shared memory model and internal structures, such as vring and virtqueue, which manage buffer queues and message flow. This layer serves as the media access abstraction for RPMsg-Lite.

The porting layer is divided into two parts: the environment layer and the platform layer. The environment layer adapts RPMsg-Lite to the operating context. For example, *rpmsg_env_bm.c* supports bare-metal environments, while *rpmsg_env_freertos.c* targets FreeRTOS. Only the file matching of the selected environment is included in the application project. The platform layer, implemented in *rpmsg_platform.c*, provides low-level functions for interrupt control and signaling. It handles enabling, disabling, and triggering interrupts used by the messaging system.

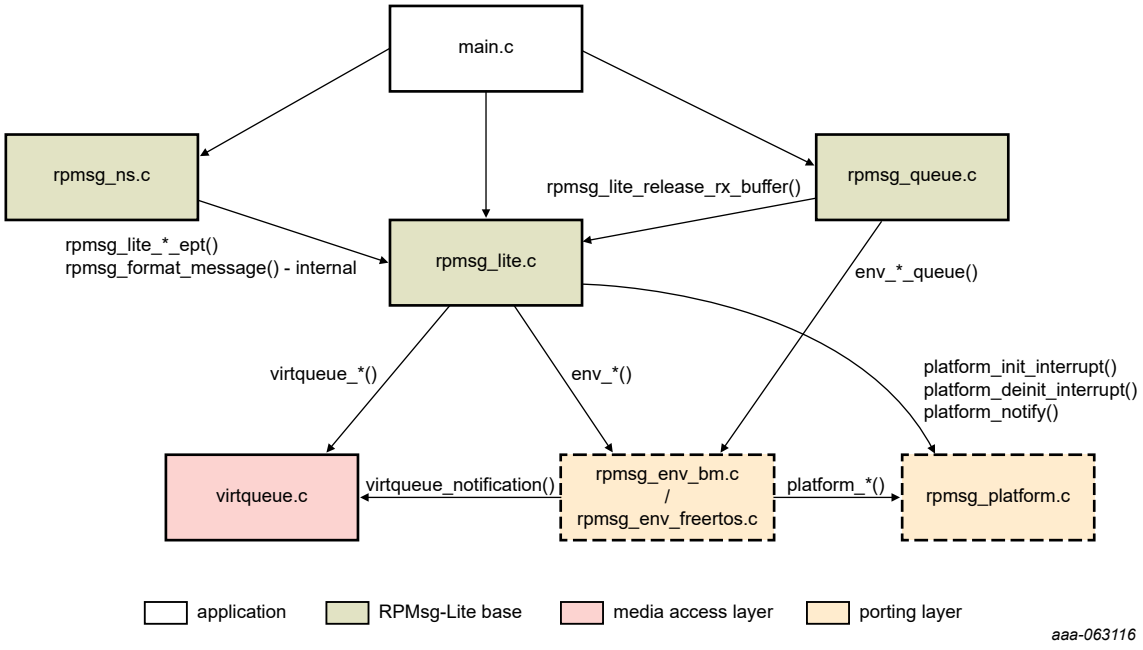

Figure 5. RPMsg-Lite architecture

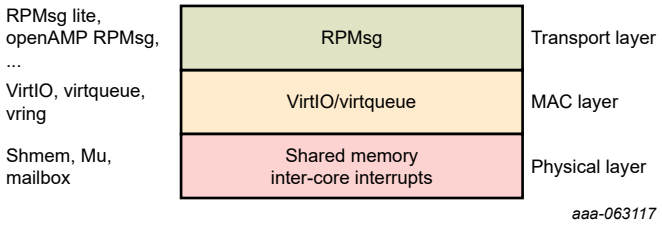

Figure 6. RPMsg-Lite core subcomponents

The RPMsg-Lite library is in the *mcuxsdk/middleware/multicore/rpmsg-lite* folder. For detailed information about the RPMsg-Lite, see the RPMsg-Lite documentation available here.

5.4.2.1 RPMsg-Lite linker considerations

RPMsg-Lite relies on the shared memory region to exchange messages between cores. This region holds the underlying data transport layer and must reside within the shared SRAM section. The linker file must define this region with correct alignment and placement.

Both projects must declare the RPMsg shared memory region consistently. The master side, typically the main core, initializes the shared memory using the *rpmsg_lite_master_init* API.

This initialization API sets up the shared memory region used by the RPMsg transport layer. This API requires two parameters: the base address and the size of the shared memory. The memory region must be large enough to store RPMsg structures, such as vrings and buffers.

In most MCUXpresso SDK implementations, the base address is defined in the linker file and exported for the initialization API usage. The shared memory size is configured using the SH_MEM_TOTAL_SIZE definition in the *rpmsg_config.h* configuration file.

Both the base address and the memory size must match the values defined in the linker script. Any mismatch between configuration and linker definitions may cause unexpected behavior. The following code snippets illustrate key aspects of the configuration.

The first snippet shows a typical linker script definition for the NBU shared RAM region.

The second snippet demonstrates the RPMsg-Lite master initialization, where the transport layer is configured to use the same memory region.

```
/* Shall be aligned with other core firmware */
define symbol m_shared_ram_start                = 0x489C0000;
define symbol m_shared_ram_size                 = 0x10000;
define symbol m_shared_ram_end                  = m_shared_ram_start +
 m_shared_ram_size - 1;
/* Section allowing to share debug info to CM33 (LL error/warnings).*/
define exported symbol m_sqram_debug_start      = m_shared_ram_start;
define symbol m_sqram_debug_size                = 0x200;
define symbol m_sqram_debug_end                 = m_shared_ram_start +
 m_sqram_debug_size - 1;
/* Section used to share platform/services (SFC, information on both cores */
define symbol framework_info_sh_mem_start       = m_sqram_debug_end + 1;
define symbol framework_info_sh_mem_size        = 0x20;
define symbol framework_info_sh_mem_end         = framework_info_sh_mem_start +
 framework_info_sh_mem_size - 1;
/* Section used for the sfc module intercore debug */
define exported symbol m_sfc_log_start          = framework_info_sh_mem_start;
define symbol m_sfc_log_size                    = 0x04;
/* Section used to share a flag on both cores for lowpower purpose */
define exported symbol m_lowpower_flag_start    = framework_info_sh_mem_start +
 m_sfc_log_size;
define symbol m_lowpower_flag_size              = 0x04;
define symbol m_lowpower_flag_end               = m_lowpower_flag_start +
 m_lowpower_flag_size - 1;
/* RPMSG section for intercore communication */
define exported symbol rpmsg_sh_mem_size        = 0x1100;
define exported symbol rpmsg_sh_mem_start       = framework_info_sh_mem_end + 1;
define exported symbol rpmsg_sh_mem_end         = rpmsg_sh_mem_start +
 rpmsg_sh_mem_size - 1;
/* Define NBU shared memory region with previously defined memory limits */
define region rpmsg_sh_mem_region               = mem:[from rpmsg_sh_mem_start
 to rpmsg_sh_mem_end];
/* Define shared memory size for RPMsg. This values must match with linker */
#define SH_MEM_TOTAL_SIZE (4352U) /* 0x1110. This size matches with linker */
/* Declare rpmsg_lite_base as memory pool for RPMsg-Lite object allocation.
 Place this memory pool on rpmsg_sh_mem_section declared on linker file.
 Placement directives keywords might change depending linker toolchain choice */
#if defined(__ICCARM__) /* IAR Workbench */
#pragma location = "rpmsg_sh_mem_section"
static char rpmsg_lite_base[SH_MEM_TOTAL_SIZE];
static char rpmsg_lite_base[SH_MEM_TOTAL_SIZE]
  __attribute__((section("rpmsg_sh_mem_section")));
#elif defined(__GNUC__)
```

```
static char rpmsg_lite_base[SH_MEM_TOTAL_SIZE] __attribute__((section(".noinit.
$rpmsg_sh_mem")));
#else
#error "RPMsg: Please provide your definition of rpmsg_lite_base[]!"
#endif
/* RPMsg initialization. This is typically done by master (main core). Provide
 pointer to RPMsg share memory along with size in bytes */
s_rpmsgContext = rpmsg_lite_master_init((void *)rpmsg_lite_base,
 SH_MEM_TOTAL_SIZE, RPMSG_LITE_LINK_ID, RL_NO_FLAGS);
```

6 Integrating a user-defined application with Bluetooth LE firmware

The KW47 microcontroller executes the Bluetooth LE link layer and radio-related tasks in the NBU. In some cases, Bluetooth LE activity remains low or inactive, such as when the radio is disabled or idle. If the target use case allows it, integrate a custom application into the NBU Bluetooth LE firmware.

The application should be correctly architected to run alongside the Bluetooth LE firmware on the same core. Avoid race conditions, deadlocks, and other runtime issues by designing the application with clear task boundaries. Task priorities must be configured to prevent interference between Bluetooth LE operations and user-defined logic. For this, use RTOS objects, such as mutex and semaphores, for correct task scheduling.

Careful scheduling and resource management ensure reliable operation. The Bluetooth LE firmware tasks must retain priority over user tasks when radio activity resumes. Developers must validate the integration under realistic load conditions to confirm system stability.

7 Custom application intercore interface

The custom application intercore interface implements a lightweight software layer that manages communication between cores. This layer performs three main functions:

1. Initializes the RPMsg-Lite endpoint for the application on the remote core
2. Handles application-specific messages received from the remote core
3. Sends application-specific messages to the remote core

This interface enables modular and scalable intercore communication. It abstracts the RPMsg-Lite transport details and provides a clean mechanism for exchanging data between the main core and the remote core. Ensure that message formats and endpoint configurations remain consistent across both cores to maintain reliable operation.

The following code snippets illustrate an example of endpoint configuration. The first snippet shows the endpoint configuration for the main core side and the second snippet shows the same for the NBU side.

```
#include “fsl_adapter_rpmsg.h”
static RPMSG_HANDLE_DEFINE(AppIcIntf_RpmsgHandle);
/* Main core RPMsg EPT configuration */
const static hal_rpmsg_config_t AppIcIntf_Rpmsg = {
.local_addr = 190,
.remote_addr = 180,
.callback = AppIcIntf_HandleMessageRx,
.param = NULL,
```

```
};
#include “fsl_adapter_rpmsg.h”
static RPMSG_HANDLE_DEFINE(AppIcIntf_RpmsgHandle);
/* NBU RPMsg EPT configuration */
const static hal_rpmsg_config_t AppIcIntf_Rpmsg = {
```

```
.local_addr = 180,
.remote_addr = 190,
};
```

8 How to register a user-defined application to use the intercore interface

To enable user-defined logic over the intercore communication layer, the application must establish the intercore link. This is done through the *AppIcIntf_InitLink* API.

Call this API during system initialization, after a successful initialization of RPMsg-Lite and MCMGR. The main core must also bring the NBU out of reset before invoking this API. Failing to meet these conditions results in a link-setup failure or undefined behavior.

When the link is initialized, the application can register its own message handlers and begin exchanging data with the remote core. This setup provides a clean entry point for extending intercore functionality with custom logic.

To add application-specific messages to the intercore interface, the developer must extend the *eAppIcIntfMsgType_t* enumeration. Each entry in this enumeration acts as a unique identifier for an intercore application message.

After defining the message type, register a corresponding callback function. This is done by assigning the callback implementation to the appropriate index in the *AppIcIntf_RxApplicationCallbackService* function pointer array. The index must match the enumeration value defined for the message type. The following code snippet shows an example of FFT application registration to the intercore interface component.

```
/* Developer may add a desired application ID to enumeration */
typedef enum
{
    gAppIcIntf_FFTApplication_c = 0x0,
    gAppIcIntfLast_c
}eAppIcIntfMsgType_t;
typedef void (*AppInIntf_Cbk_t)(uint8_t*, uint32_t);
/* Register here any generic application service callbacks request to remote
 core */
/* Array of function pointers used in AppIcIntf_HandleMessageRx() */
static AppInIntf_Cbk_t AppIcIntf_RxApplicationCallbackService[gAppIcIntfLast_c]
 =
{
    AppNbuFFT_HandleFFTRx, /* Index 0. Correctly matches
gAppIcIntf_FFTApplication_c enumeration value */
};
```

By registering both the message type and its callback, enable the interface to route incoming messages to the correct handler. This approach supports the integration of multiple generic applications into the intercore communication layer with minimal overhead.

When the remote core sends an application intercore message, it triggers an Interrupt Service Routine (ISR) on the receiving core. During this ISR, the *AppIcIntf_HandleMessageRx* callback executes. This API receives a buffer containing the bytes sent by the remote core.

The callback can execute directly in the ISR context, as supported by MCSDK. Alternatively, the system can defer execution through an RTOS event mechanism. This flexibility allows developers to balance the responsiveness and system load based on application requirements.

The *AppIcIntf_HandleMessageRx* callback API unpacks the first byte in the buffer to identify the target application logic. This byte must match one of the values defined in the *eAppIcIntfMsgType_t* enumeration. The

API uses this identifier to select the correct callback from the *AppIcIntf_RxApplicationCallbackService* function pointer array.

To send a generic application message to the remote core, use the *AppIcIntf_MessageTx* API. This API takes three parameters: a pointer to the buffer containing the payload, the size of the payload in bytes, and the application message type defined in the *eAppIcIntfMsgType_t* enumeration.

The API packs the application message type into the first byte of the transmission buffer. The remaining bytes contain the payload data. After packing, the core initiates the data exchange with the remote core through the RPMsg-Lite transport layer.

This mechanism ensures that the remote core can identify the message type and route it to the correct application callback. Figure 7 shows an example application sequence flow between cores.

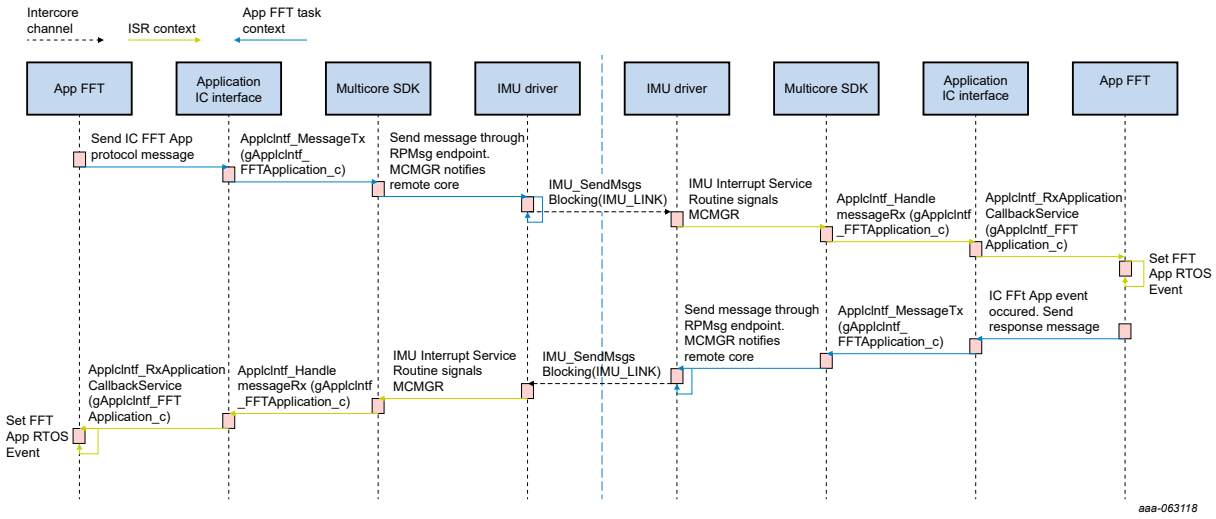

Figure 7. Inter-core application sequence diagram

9 Proof-of-concept application demonstrating NBU-assisted performance enhancement

This chapter presents a proof-of-concept application designed to evaluate how the NBU can improve overall system performance when integrated with user-defined logic. The application runs on the KW47-EVK platform and uses the LCD-PAR-S035 display module. This module features a 3.5-inch 480 × 320 IPS TFT panel with wide viewing angles and supports 5-point capacitive touch input. The display connects to the main core through a Pmod interface, using SPI for screen control and I2C for touch-panel communication.

The main core drives the display and continuously renders animations using the LVGL graphics library. LVGL also renders four interactive buttons on the LCD-PAR-S035 TFT module. These buttons appear as part of the graphical interface and respond to user input through the touch panel connected via I2C. Each button provides a specific function and can be pressed directly on the screen to trigger its associated action:

- **Red label**: Triggers an FFT computation.
- **Green label**: Toggles the FFT computation source between the main core and the NBU. When the NBU is selected, the input data is sent through the intercore protocol, processed by the NBU, and the output is returned to the main core. When the main core is selected, it computes the FFT locally without offloading.
- **Blue label**: Controls the Bluetooth LE status. Pressing it starts the Bluetooth LE wireless UART advertising. If connected to a mobile device using the NXP IoT Toolbox application, the system sends the FFT computation/ reception time over Bluetooth LE.

- **Yellow label**: Increases the number of FFT iterations in steps of 10. This button serves for debugging purposes only. It intentionally increases CPU load to affect system responsiveness. When the number of iterations exceeds 1, the FFT output becomes unreliable due to in-place computation by the CMSIS DSP library.

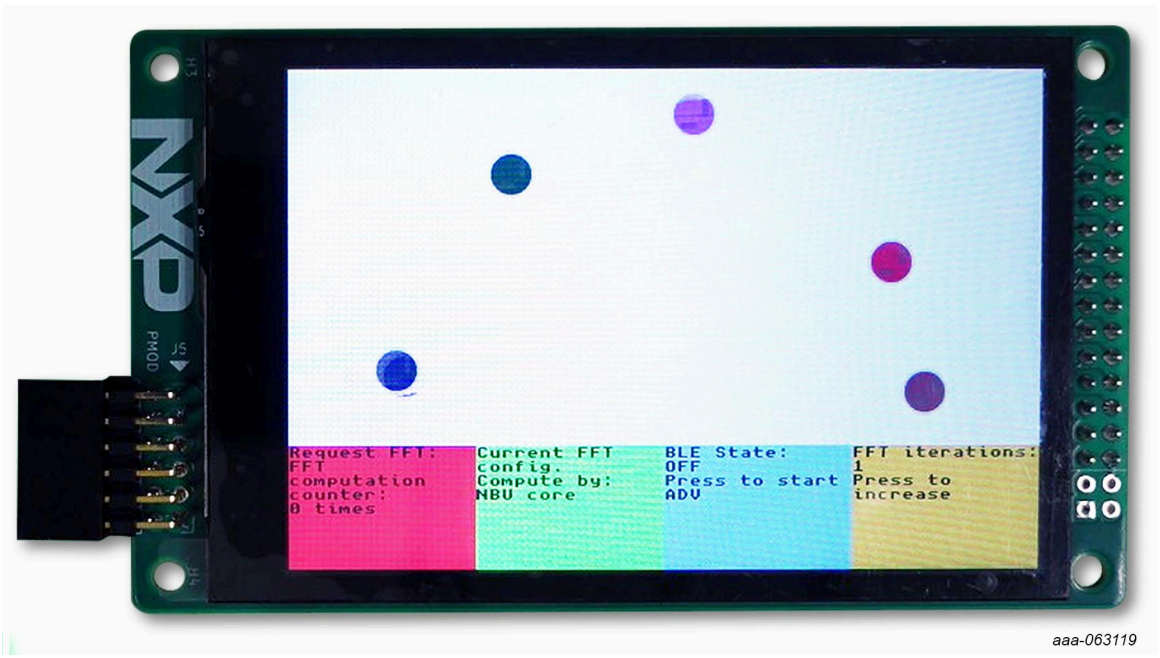

Figure 8. Proof-of-concept application user interface

This setup enables you to observe how the system responsiveness changes under varying CPU loads. By increasing FFT iterations and toggling the computation source, the impact on LVGL animation performance becomes visible. This behavior supports the claim that offloading tasks to the NBU can improve application responsiveness and reduce main core workload.

The FFT functionality is integrated into the MCUXpresso SDK wireless UART example on the main core and into the NBU Bluetooth LE firmware on the remote core. This integration demonstrates how to extend existing SDK examples to add custom application logic on both cores. Starting from a working Bluetooth LE communication baseline simplifies the development and allows faster prototyping of intercore applications.

10 Introduction to LVGL graphics library

LVGL (Light and Versatile Graphics Library) is an open-source graphics library optimized for embedded systems with limited resources. It provides a comprehensive set of features for building modern graphical user interfaces, including widgets, animations, themes, and input device handling.

LVGL supports a wide range of display resolutions and color formats. It integrates efficiently with hardware abstraction layers and display drivers, making it suitable for microcontrollers like the KW47. The library uses a modular architecture that allows developers to enable only the required components, reducing memory footprint and improving performance.

In this proof-of-concept application, LVGL handles both display rendering and touch input. The library drives the LCD-PAR-S035 module using SPI for display control and I2C for capacitive touch input. LVGL internal task handler manages screen updates and animations, allowing smooth visual transitions even under moderate CPU load.

LVGL offers several advantages for embedded GUI development:

- **Low resource usage**: Designed for systems with constrained RAM and CPU.
- **Scalability**: Supports simple UIs as well as complex, multiscreen applications.
- **Portability**: Compatible with various platforms and display controllers.
- **Customization**: Allows full control over widget behavior, styling, and input handling.

These features make LVGL a suitable choice for integrating responsive graphical interfaces into embedded applications, especially when combined with real-time processing tasks.

11 Introduction to CMSIS DSP library and FFT q31 format

The CMSIS DSP library provides a collection of optimized signal-processing functions for Arm Cortex-M processors. It includes APIs for filtering, matrix operations, statistics, and transforms. Among its most widely used features are the FFT functions, which enable efficient frequency-domain analysis of real or complex signals.

This proof-of-concept application uses the complex FFT q31 format, which represents fixed-point data using 32-bit signed integers. In the q31 format, each value occupies four bytes and represents a number in the range of −1.0 to $((2^{31} - 1) / (2^{31}))$, which is effectively just under +1. The complex FFT input consists of interleaved real and imaginary components, meaning that each complex sample uses eight bytes (four bytes for real components, four bytes for imaginary components).

The memory footprint for the FFT input and output buffers scales with the FFT size, as shown in Table 3 illustrates.

Table 3. FFT input/output size

FFT input/output size	Size in bytes
32	256
64	512
128	1024
256	2048
512	4096

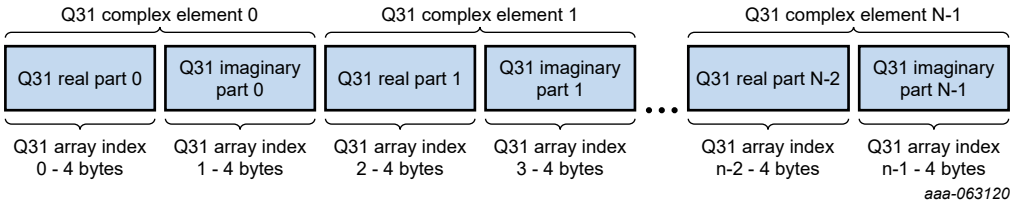

Figure 9. Example of interleaved complex q31 array of size n

These sizes apply to both input and output buffers, as the CMSIS DSP library performs in-place computation. Make sure to allocate enough of memory.

The FFT computation in this application uses the *arm_cfft_q31* API. This function performs a complex FFT on the q31 data and supports sizes that are powers of two. The library also provides helper functions for the magnitude and phase extraction, which can be used for further signal analysis.

12 Integrating the application intercore interface into Bluetooth LE firmware

The NBU Bluetooth LE firmware runs on top of the ThreadX RTOS. You can define user tasks and related RTOS objects inside the *tx_application_define* function. This is the recommended entry point for creating application-level logic within the NBU firmware.

To enable intercore communication, the application must initialize the intercore interface using the *AppIcIntf_InitLink* API. This API requires the RTOS scheduler to be active, as some RPMsg-Lite operations depend on RTOS synchronization primitives, such as mutexes. For this reason, *AppIcIntf_InitLink* must not be called from *tx_application_define*, since the scheduler is not yet running at that point.

A suitable location to call *AppIcIntf_InitLink* is within the *NBU_Init* function. This ensures that the system is ready for intercore communication.

When the intercore link is active, the application must handle incoming IPC messages efficiently. Processing messages directly in the ISR introduces latency and risks blocking time-critical operations. To avoid this, the application should defer message handling to a dedicated task. This is achieved by signaling an RTOS event object, such as a ThreadX event flag, from the ISR. The application task waits on this event and processes messages when signaled.

This approach improves system responsiveness and maintains Bluetooth LE protocol timing. Assign lower priorities to user-defined tasks not to Bluetooth LE radio-related tasks. This priority scheme prevents interference with Bluetooth LE stack operations and ensures reliable wireless communication.

13 High-level software architecture overview

This section presents a high-level software architecture for the FFT offload application. The implementation supports the complex q31 format with FFT sizes ranging from 32 to 512 points.

The architecture places the user-defined application, FFT processing in this proof of concept, above the intercore interface software component. This separation allows the application to remain agnostic to the underlying communication mechanism while using it for task offload.

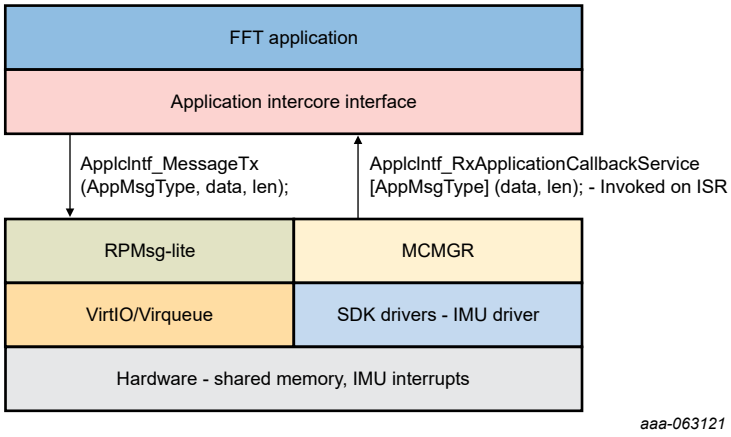

Figure 10. FFT intercore application software architecture

The intercore protocol defines four frame types: request, input, Continue to Send (CTS), and output. The main core initiates the offload process by sending a request frame. This frame includes metadata, such as the data size and the number of FFT iterations.

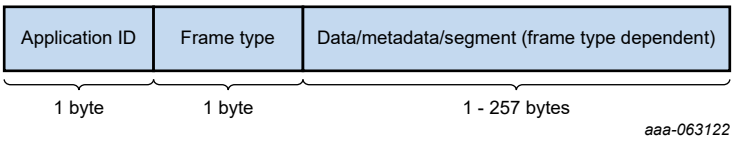

Figure 11. FFT intercore application protocol frame format

If the data size exceeds 256 bytes, segmentation is required. The NBU uses the size field in the request frame to determine the number of input and output frames needed. Each segmented frame includes a frame number to ensure correct reassembly.

The CTS frame serves as a handshake mechanism. It is sent in response to the request frame and to each segment of input and output data. This control flow mechanism ensures synchronization and robustness in intercore communication.

Figure 12 and Figure 13 show the software architecture and protocol flow between the main core and the NBU:

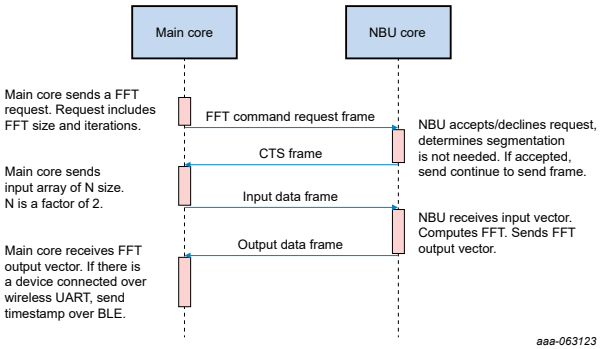

Figure 12. FFT intercore application flow diagram - single-frame case

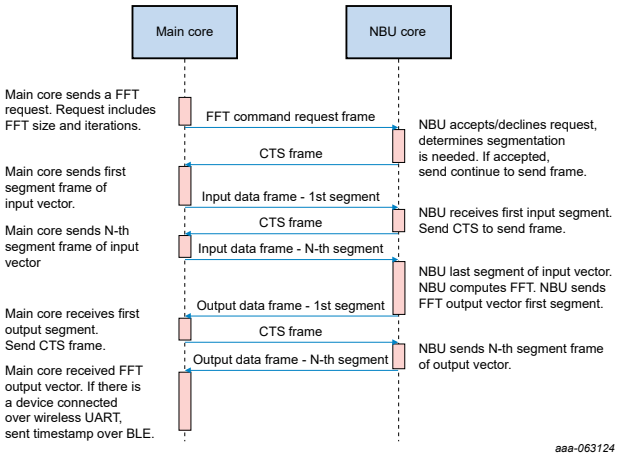

Figure 13. FFT intercore application flow diagram - multiframe case

13.1 Application setup and programming procedure

This section describes the steps required to configure the software and hardware components for the PoC intercore application. It includes instructions for preparing the development environment and configuring the required hardware.

13.1.1 Software setup and flashing procedure

The FFT intercore application includes two projects for MCUXpresso for VS Code:

- **kw47_wuart_fft_freertos_display** (main core)
- **kw47_fft_nbu_ble** (NBU core)

Each project contains the necessary source files, configuration scripts, and build settings required for execution on KW47-EVK boards.

To modify the FFT input or output size, update the FFT_SIZE definition in the *app_nbu_fft.h* source file. Supported values are 32, 64, 128, 256, and 512, as listed in Table 3. You may change this value before building the project.

Set up and flash the application as follows:

1. Extract the software package to a local workspace directory.
2. Launch MCUXpresso for VS Code and open the workspace.
3. Import both projects using the "Import Project" option.
4. Select the KW47-EVK board during import.
5. Build both projects using the "Build Project" option.
6. Connect the target board via USB and ensure that the debugger is recognized.
7. Flash the firmware to each core using the "Flash the Selected Target" option in the IDE.

13.1.2 Hardware configuration

This section lists the hardware required for the FFT intercore application and describes the necessary configuration steps. The setup uses a KW47-EVK evaluation board, a KW47-001-M10, and an LCD-PAR-S035 TFT LCD module. The module and evaluation board connect through the Pmod connector. The LCD display uses the SPI interface for data and the I2C interface for touch-panel input. The following subchapters provide detailed instructions for setup configuration.

13.1.2.1 KW47-EVK settings

Prepare the KW47-EVK board for the application as follows:

1. Set the board to the default jumper configuration. See the KW47-EVK schematics for the default jumper layout.
2. Remove jumpers JP33 and JP30. Leave these connections open to prevent interference from the onboard devices that use I2C or SPI as an interface. This ensures clean communication with the external TFT module. Figure 14 shows the locations of the mentioned jumpers on the evaluation board.

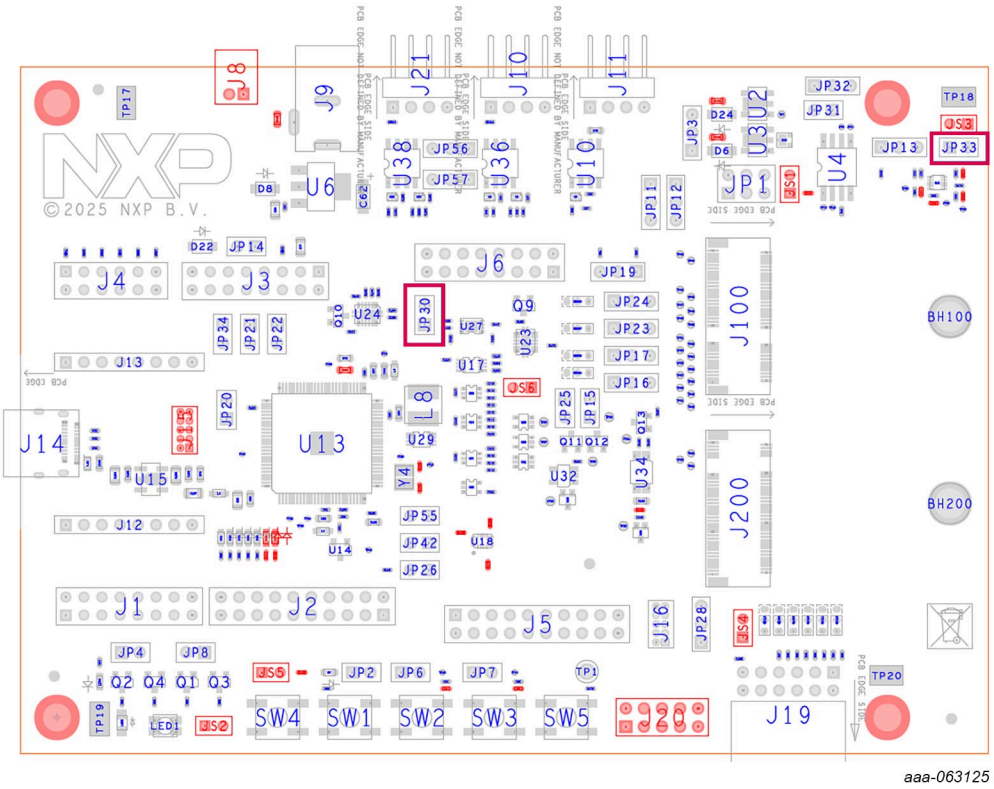

Figure 14. KW47-EVK board view

3. Connect the KW47-001-M10 to the M.2 MAIN connector on the evaluation board.

This configuration enables proper operation of the SPI and I2C interfaces used by the external display and touch panel.

13.1.2.2 External hardware

The FFT intercore application uses the LCD-PAR-S035 module for graphical output. This module features a 3.5-inch 480 × 320 IPS TFT display with wide viewing angles and 5-point capacitive touch input. The display supports both SPI and parallel (8/16-bit 8080/6800) interfaces and is compatible with evaluation boards through either a Pmod connector or a parallel LCD interface.

Figure 15. LCD-PAR-S035 TFT LCD module

To configure the display for SPI operation:

1. Set the interface mode to 4-wire SPI. Adjust the DIP switches on the back of the TFT module to binary 111. See Figure 16 for switch locations.

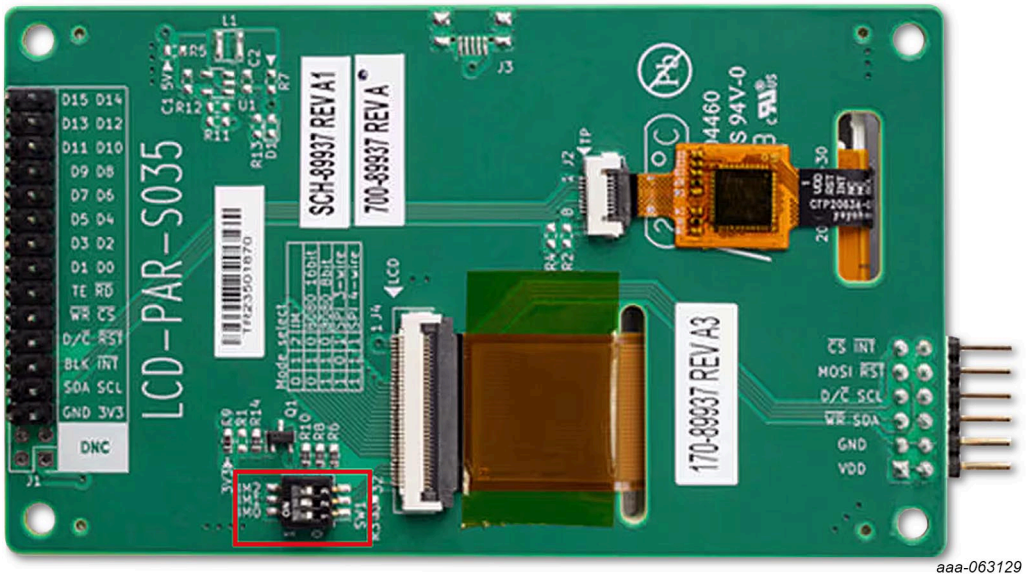

Figure 16. LCD-PAR-S035 TFT LCD module - interface configuration switches

2. Connect the display to the Pmod connector (J19) on the KW47-EVK board.
3. Press the RESET_PB (SW1) button on the evaluation board to start the application.

14 Application benchmarking

To evaluate the performance of FFT computation on KW47, a set of representative input signals was processed using both the main core and the NBU core. Each input was hardcoded and executed across varying FFT sizes. The number of instruction cycles required for each computation was measured and plotted. The results are shown in the following figures, which illustrate the instruction cycles distribution between cores and highlight the efficiency gains achieved through offloading.

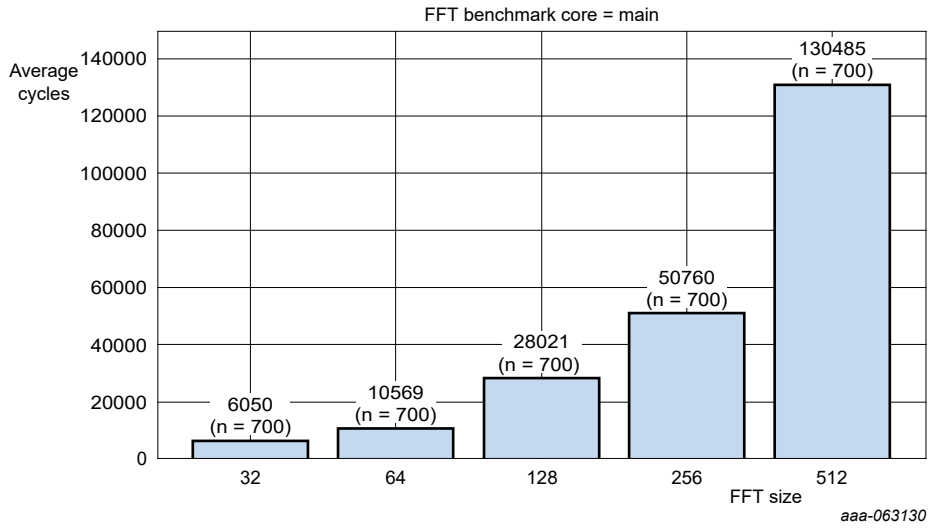

Figure 17. FFT benchmark - main core

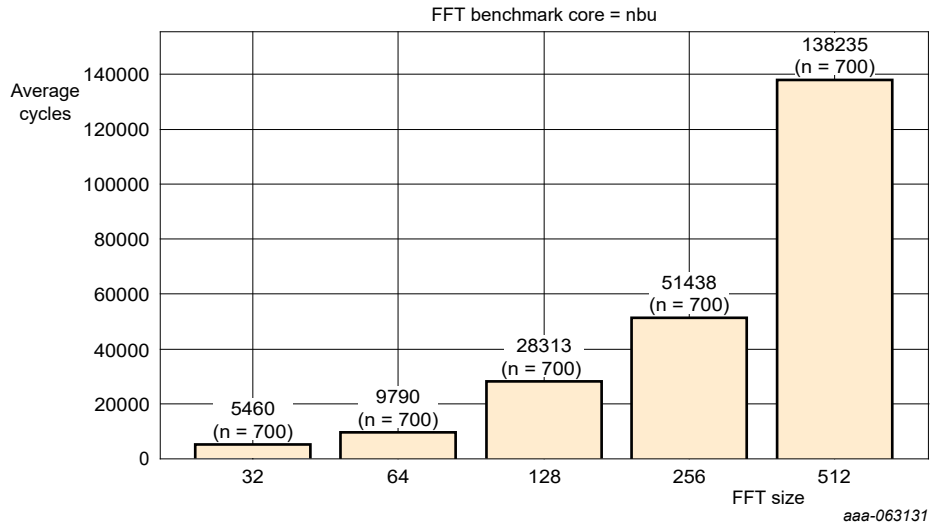

Figure 18. FFT benchmark - NBU core

Table 4. FFT benchmark results

FFT size	Main core average instruction cycles	NBU core average instruction cycles
32	6050	5460
64	10569	9790
128	28021	28313

Table 4. FFT benchmark results*...continued*

FFT size	Main core average instruction cycles	NBU core average instruction cycles
256	50760	51438
512	130485	138235

15 Conclusion

The PoC application demonstrates the effectiveness of KW47 dual-core architecture in handling compute-intensive workloads. The system maintains responsiveness while executing graphics rendering, wireless link layer operations, and FFT computations. Benchmark results show that FFT performance is comparable across cores. The NBU core performs better for small FFT sizes due to lower overhead, while the main core outperforms the NBU as the FFT size increases, using its DSP-extension instruction set. Despite lacking DSP extensions, the NBU core delivers strong performance and remains a viable target for custom computing tasks. You can extend its use to offload applications and optimize the overall system behavior.

16 Abbreviations

Table 5. Abbreviations

Abbreviation	Description
NBU	Narrowband Unit
SoC	System-on-Chip
Bluetooth LE	Bluetooth Low Energy
DSP	Digital Signal Processor
FPU	Floating-Point Unit
FFT	Fast Fourier Transform
CPU	Central Processing Unit
MCSDK	Multicore Software Development Kit
IPC	Inter Processor Communication
MU	Message Unit
OpenAMP	Open Asymmetric Multi Processing
DMEM	Data Memory
IMEM	Instruction Memory
PoC	Proof of Concept

17 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted if the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABluetooth LE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

18 Revision history

Table 6. Revision history

Document ID	Release date	Description
AN14846 v.1.0	10 December 2025	• Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

1 Introduction 2
2 KW47 SoC architecture 2
2.1 Using the NBU as a secondary compute core 3
3 Practical use case: Fast Fourier Transform (FFT) computation 4
4 Main core and NBU interface components 5
5 Dedicated IPC hardware 5
5.1 MU 5
5.1.1 Functional description 6
5.1.2 IMU registers description 6
5.2 SMU2 7
5.2.1 NBU SMU/DMEM multiplexing 7
5.2.2 SMU linker considerations 8
5.3 IPC using low-level drivers 9
5.4 Multicore SDK 9
5.4.1 MCMGR 10
5.4.2 Remote Processor Messaging Lite (RPMsg-Lite) 10
5.4.2.1 RPMsg-Lite linker considerations 11
6 Integrating a user-defined application with Bluetooth LE firmware 13
7 Custom application intercore interface 13
8 How to register a user-defined application to use the intercore interface 14
9 Proof-of-concept application demonstrating NBU-assisted performance enhancement 15
10 Introduction to LVGL graphics library 16
11 Introduction to CMSIS DSP library and FFT q31 format 17
12 Integrating the application intercore interface into Bluetooth LE firmware 18
13 High-level software architecture overview 18
13.1 Application setup and programming procedure 19
13.1.1 Software setup and flashing procedure 19
13.1.2 Hardware configuration 20
13.1.2.1 KW47-EVK settings 20
13.1.2.2 External hardware 21
14 Application benchmarking 23
15 Conclusion 24
16 Abbreviations 24
17 Note about the source code in the document 24
18 Revision history 25
Legal information 26