AN12789

RT600 Dual-Core Communication and Debugging

Rev. 0 — March 20, 2020 Application Note

Contents

1 Introduction

1.1 Overview

The RT600 is a family of dual-core microcontrollers for embedded applications featuring an Arm® Cortex®-M33 CPU combined with a Cadence Xtensa HiFi4 advanced Audio Digital Signal Processor CPU. The Cortex-M33 includes two hardware coprocessors providing enhanced performance for an array of complex algorithms. The Arm Cortex-M33 is a next generation core, based on the Armv8-M architecture that offers system enhancements, such as Arm TrustZone® security, single-cycle digital signal processing, and a tightly-coupled coprocessor interface, combined with low power consumption, enhanced debug features, and a high level of support blockintegration.

The Cadence Xtensa HiFi4 Audio DSP engine is a highly optimized audio processor designed especially for efficient execution of audio and voice codecs and pre- and post-processing modules.

The RT600 provides up to 4.5 MB of on-chip SRAM (plus an additional 128 Kb of tightly-coupled HiFi4 ram) and several high-bandwidth interfaces to access off-chipflash.

The RT600 is designed to allow the Cortex-M33 to operate at frequencies of up to 300 MHz and the HiFi4 DSP to operate at frequencies of up to 600 MHz.

1.2 Dual-core basic mechanism

The Cortex-M33 (Processor A) is factory set to work normally. On the other hand, the HiFi4 DSP (Processor B) is not powered when the RT600 boots up to minimize power consumption. To run or debug DSP applications, execute some code on the Cortex-M33 (Processor A) to initialize HiFi4 DSP (Processor B).

With the dual-core running mode, the Cortex-M33 and the DSP need to communicate with each other. The RT600 provides two simple means, Message Unit (MU) and Semaphore Block, to achieve this task.

- **MU module**

  It enables two processors within the SoC to communicate and coordinate by passing messages (e.g. data, status and control) through the MU interface.

  The MU also provides the ability for one processor to signal the other processor usinginterrupts.

- **Semaphore Block**
  - The Semaphore module provides the hardware support needed in multi-core systems for implementing semaphores.
  - The Semaphore module also provides a simple mechanism to achieve the **lock/unlock** operations via a single write access.

NOTE

The application note's references regarding Processor A correspond to the Cortex-M33 and references regarding Processor B correspond to the HiFi4 DSP.

1.3 Related system resources

To get a better performance on dual-core usage while using the shared memories and peripherals, the following mechanisms are supported.

1.3.1 Shared system SRAM

The entire system SRAM space of up to 4.5 MB is divided into up to 30 separate partitions, which are accessible to both CPUs, both DMA engines and all other AHB bus masters. The HiFi4 CPU accesses the RAM via a dedicated 256-bit interface. All other masters, including the Cortex-M33 processor and the DMA engines, access RAM via the main 32-bit AHB bus. Hardware interface modules arbitrate access to each RAM partition between the HiFi4 and the AHB bus.

1.3.2 AHB multi-layer matrix

A multi-layer AHB matrix connects the CPU buses and other bus masters to peripherals in a flexible manner that optimizes performance by allowing peripherals on different slave ports of the matrix to be accessed simultaneously by different bus masters. Figure 1 shows the multi-layer AHB matrix block diagram.

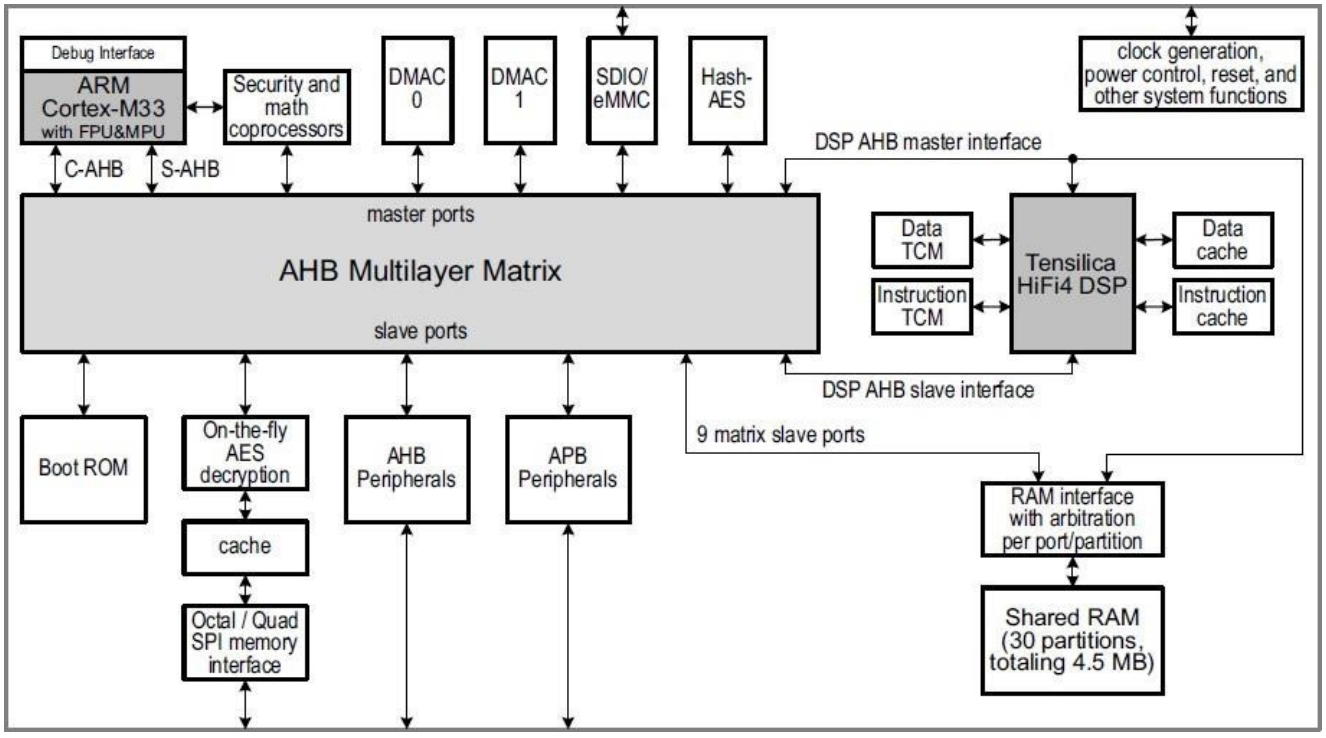

Figure 1. Multi-layer matrix block diagram

2 Message Unit

The MU is a shared peripheral with a 32-bit IP bus interface and interrupt request signals to each processor. The MU exposes a set of registers to each processor, which facilitates inter-processor communication via 32-bit words, interrupts and flags. Figure 2 shows the MU block diagram.

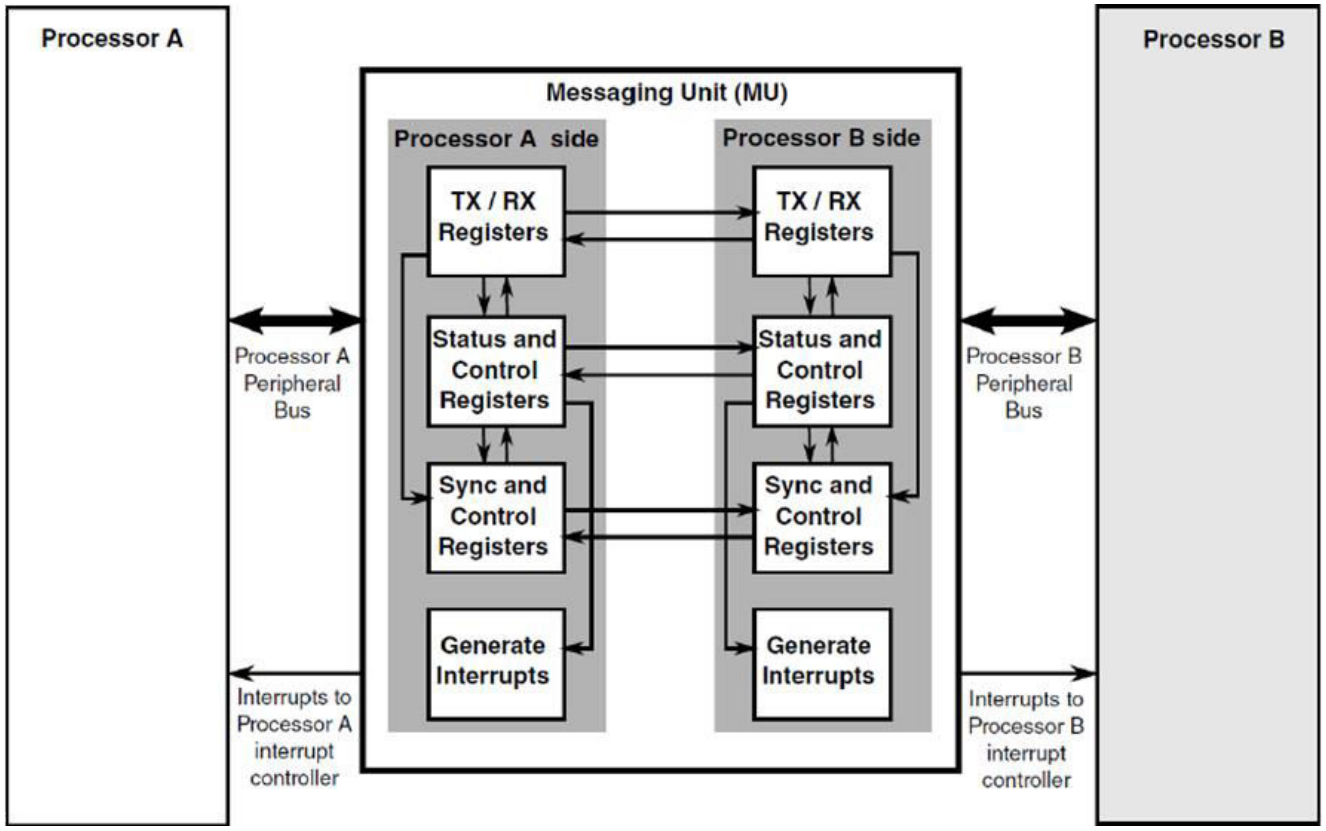

Figure 2. MU block diagram

2.1 Features

- Two ports: one for the Cortex-M33 and one for the HiFi4 DSP.
- Messaging control by interrupts or by polling.
- Symmetrical processor interfaces with each side supporting the following:
  - Three general-purpose flags reflected to the other side.
  - Four general-purpose interrupt requests reflected to the other side.
  - Four receive registers with maskable interrupt.
  - Four transmit registers with maskable interrupt.
- Cortex-M33 can take HiFi4 DSP core out of low-power modes by asserting one of the HiFi4 DSP interrupts and vice versa.

2.2 Basic configurations

Initial configuration of the MU can be accomplished as follows:

- Enable the clock of the Message Unit in the `CLKCTL1_PSCCTL1` register. This enables the register interface and the peripheral function clock.
- Clear the Message Unit peripheral reset in the `RSTCTL1_PRSTCTL1` register by writing to the `RSTCTL1_PRSTCTL1_CLR` register.

2.3 Messaging protocols in interrupt mode

The following steps and Figure 3 describe the messaging model using Transmit/Receive Registers (TR/RR) and interrupts. The example describes a messaging sequence sent from Processor A to Processor B with Transmit and Receive interrupts enabled.

1. **Processor A Data write.**

   Processor A writes the message to the Transmit Register (TRn). This message is immediately reflected in the Processor B Receive Register (RRn).

2. **Clear Tx Empty bit and Set Rx full bit.**

   The data write to the TRn register by Processor A clears the transmitter empty bit (TEn) in the Processor A Status Register. Additional to that, the Processor A writes to the TRn register causes the receiver full bit (RFn) in Processor B is set.

3. **Generate Receive Interrupt request.**

   The setting of the RFn bit generates a receive interrupt request to Processor B.

4. **Processor B Data read.**

   After the Receive interrupt request is triggered, Processor B performs a data read of its RRn register.

5. **Clear Rx Full bit and Set Tx Empty bit.**

   Reading the data out of the RRn register, clears the receiver full bit (RFn) in Processor B and sets the transmitter empty bit (TEn) in Processor A.

6. **Generate Transmit Interrupt request.**

   When the transmitter empty bit (TEn) in Processor A is set, it generates a Transmit interrupt request. At this point, if there are still messages that need to be sent, the process starts all over again.

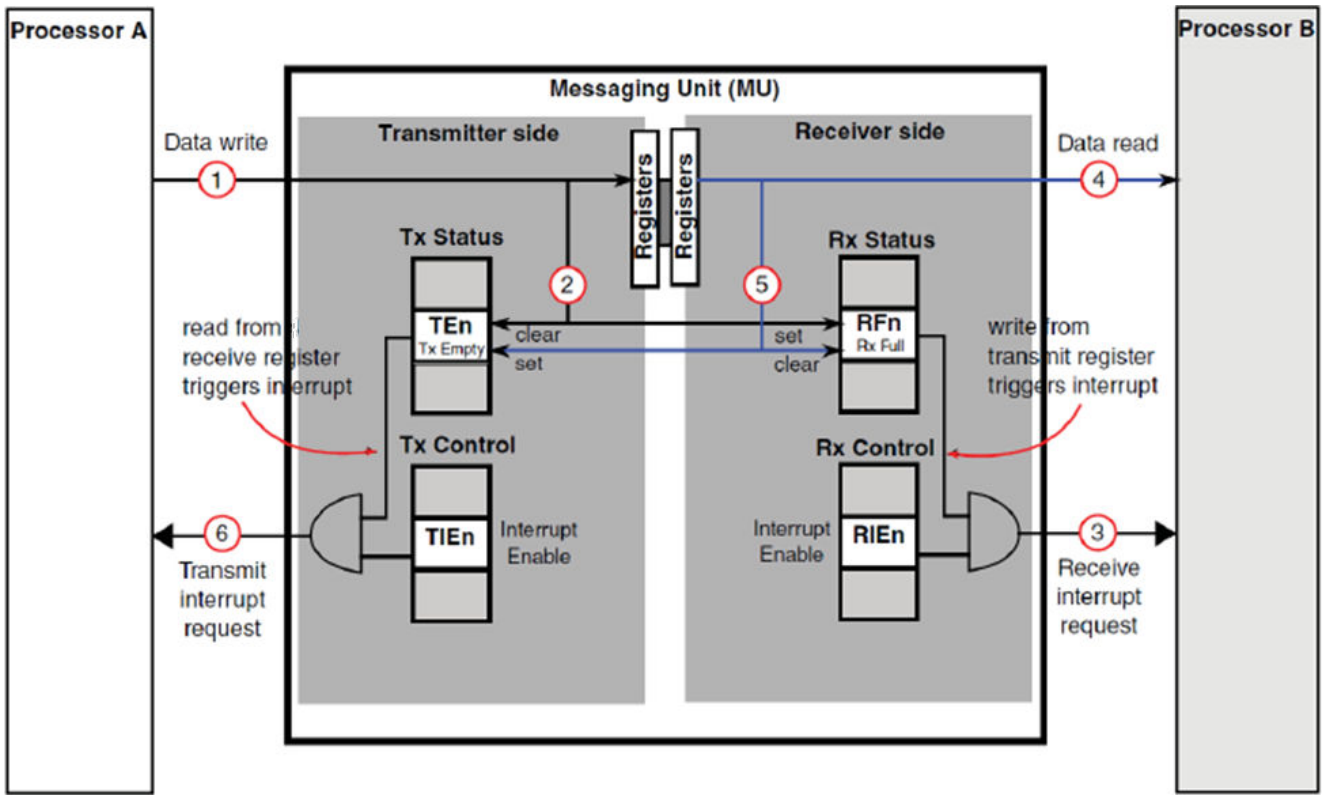

Figure 3. Messaging protocol in interrupt mode

2.4 Messaging protocols in interrupt mode

The following steps and Figure 4 show the messaging model using transmit/receive registers and polling messaging protocol. The example describes a messaging sequence sent from Processor A to Processor B.

1. **Wait TEn register to be empty.**

   Processor A needs to wait until the TEn bit on the Status Register (SR) is empty.

2. **Processor A data write.**

   Processor A writes to the TRn register the message, which is immediately reflected in Processor B Receive Register (RRn).

3. **Clear Tx Empty bit and Set Rx full bit.**

   The data write to the TRn register by Processor A clears the transmitter empty bit (TEn) in Processor A Status Register. Once Processor A writes the message into the TRn register, the receiver full bit (RFn) in Processor B is set.

4. **Wait until the RFn bit on the Status Register is set.**

   Processor B will be polling the RFn bit on the Status Register until it is set.

5. **Processor B data read.**

   After the RFn bit is set, Processor B performs a data read of the RRn register.

6. **Clear Rx Full bit and Set Tx Empty bit.**

   Reading the data out of the RRn register, clears the receiver full bit (RFn) in Processor B and sets the transmitter empty bit (TEn) in Processor A. At this point, if there are still messages that need to be sent, the process starts all over again.

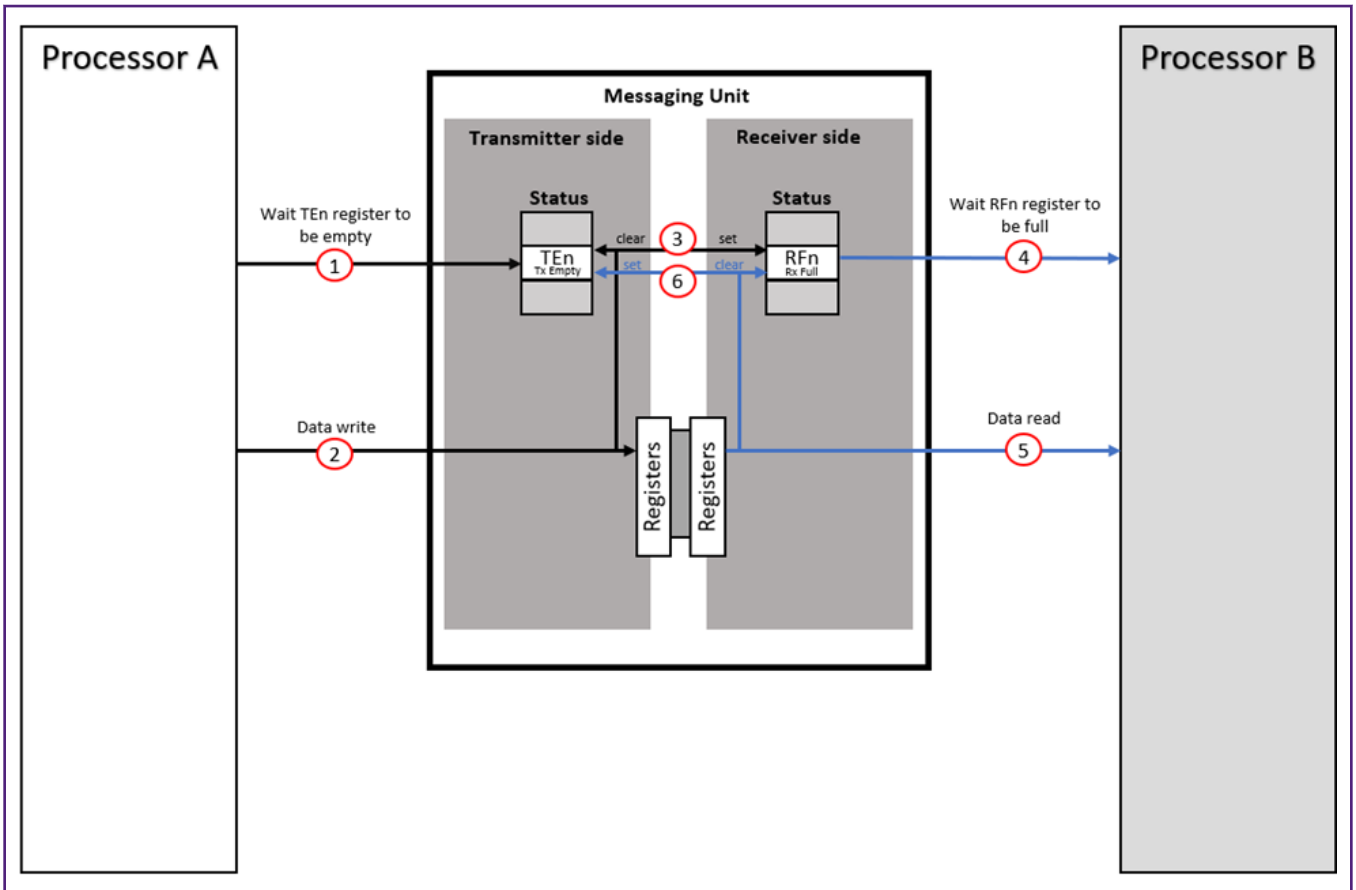

Figure 4. Messaging protocol in interrupt mode

3 Semaphore mode

Multi-processor systems, like the RT600, require a mean that can be used to provide a locking mechanism, which is used by system to control access to shared data, shared hardware resources and so on. These gating mechanisms are used to prevent race conditions and preserve memory coherency between different processes and processors. The Semaphore module provides the hardware support needed to achieve this in the RT600.

3.1 Features

- Support for 16 hardware-enforced gates in a multi-processor configuration. The term `cp0` is the Cortex-M33, `cp1` is the HiFi4. The term `cpX` represents core processor X.
- Gates appear as a 16-entry byte-size array with read and write accesses.
  - Processors lock gates by writing `processor_number+1` to the appropriate gate and must read back the gate value to verify the lock operation was successful.
  - After the gate is locked, it is unlocked by a write of zeros from the locking processor.
- Each hardware gate appears as a 16-state, 4-bit state machine.
  - 16-state implementation:
    - If gate = 0x0, then state = unlocked.
    - If gate = 0x1, then state = locked by processor (master) 0.
    - If gate = 0x2, then state = locked by processor (master) 1.
    - …
    - If gate = 0xF, then state = locked by processor (master) 14.
  - Uses the logical bus master number as a reference attribute and the specified data patterns to validate all write operation.
- Secure reset mechanisms are supported to clear the contents of individual gates, as well as a clear-all capability.

3.2 Basic configurations

Initial configuration of the Semaphore block can be accomplished as follows:

- Enable de clock to the Semaphore block in the `CLKCTL1_PSCCTL1` register. This enables the register interface and the peripheral function clock.
- Clear the Semaphore block peripheral reset in the `RSTCTL1_PRSTCTL1` register by writing to the `RSTCTL1_PRSTCTL1_CLR` register.

3.3 Functional descriptions

The following steps and Figure 5 describe the procedure that Processor A follows to block a gate before modifying shared data.

- **Wait until the gate is unlocked.**

  If the gate is locked by Processor B or a process, Processor A will wait until the gate is unlocked.
- **Lock the gate.**

  Processor A will lock the gate by writing `processor_number+1`.
- **Read back the gate value.**

  Processor A must read back the gate value to verify the lock operation was successful.
- **Notify Processor B that the gate is locked.**

  Processor A must notify Processor B that the gate is locked, this is done through the Message Unit.

- **Access shared data.**

  Processor A will modify the shared data.

- **Unlock the gate.**

  Finally, Processor A will unlock the gate.

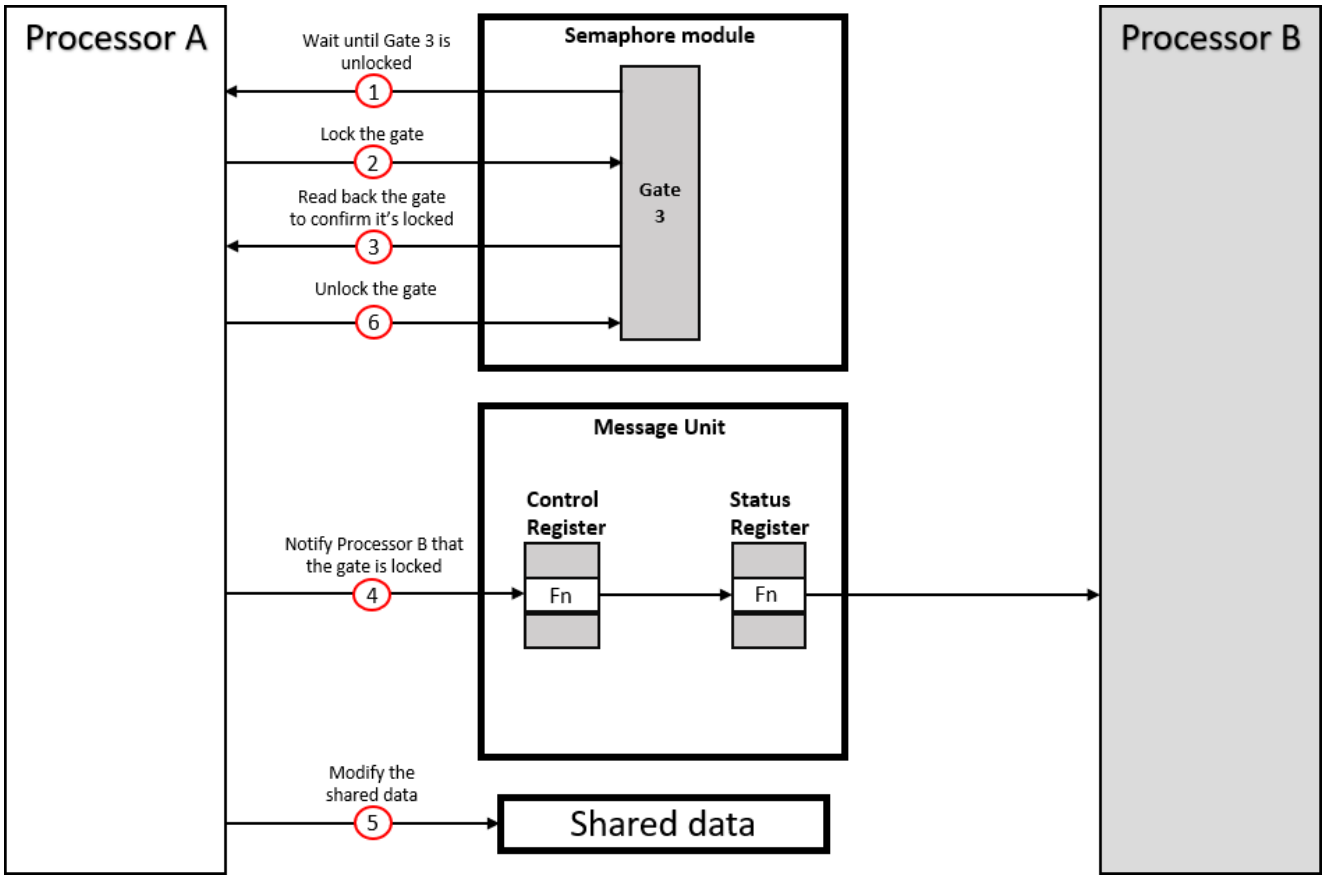

Figure 5. Semaphore block functional description

4 Software implementations

The SDK for the RT600-EVK provides examples of the Message Unit and the Semaphore Block. The SDK which can be downloaded from https://mcuxpresso.nxp.com.

To have a better understanding on how to properly configure the Semaphore block and the MU in polling and interrupt mode, this section will go through a high-level explanation of the examples `mu_interrupt`, `mu_polling`, and `sema42` from the SDK.

4.1 Pin settings

The Message Unit and the Semaphore functions are not associated with any device pins.

4.2 Messaging in interrupt mode implementation

The `dsp_mu_interrupt` demo application provided within the SDK demonstrates how to send messages in interrupt mode in both directions from Processor A to Processor B and vice versa.

This example does the following actions:

- Processor A sends 32 messages one by one to Processor B in interrupt mode. Each sent message will trigger a Receive Full interrupt on Processor B.
- Processor B receives the messages in interrupt mode. Reading the message will trigger a Transmit Empty interrupt on Processor A.
- When Processor B finishes receiving all the messages, it will send the 32 received messages to Processor A in interrupt mode, each message will trigger a Receive Full interrupt on Processor A.
- Processor A receives the messages in interrupt mode. Reading the message will trigger a Transmit Empty interrupt on Processor B.
- When Processor A finishes receiving all the messages, it will compare the received messages with the ones it sent to see that the communication was successful.

Figure 6 shows a high-level description of how this SDK example works. For more details regarding this example, refer to the SDK project.

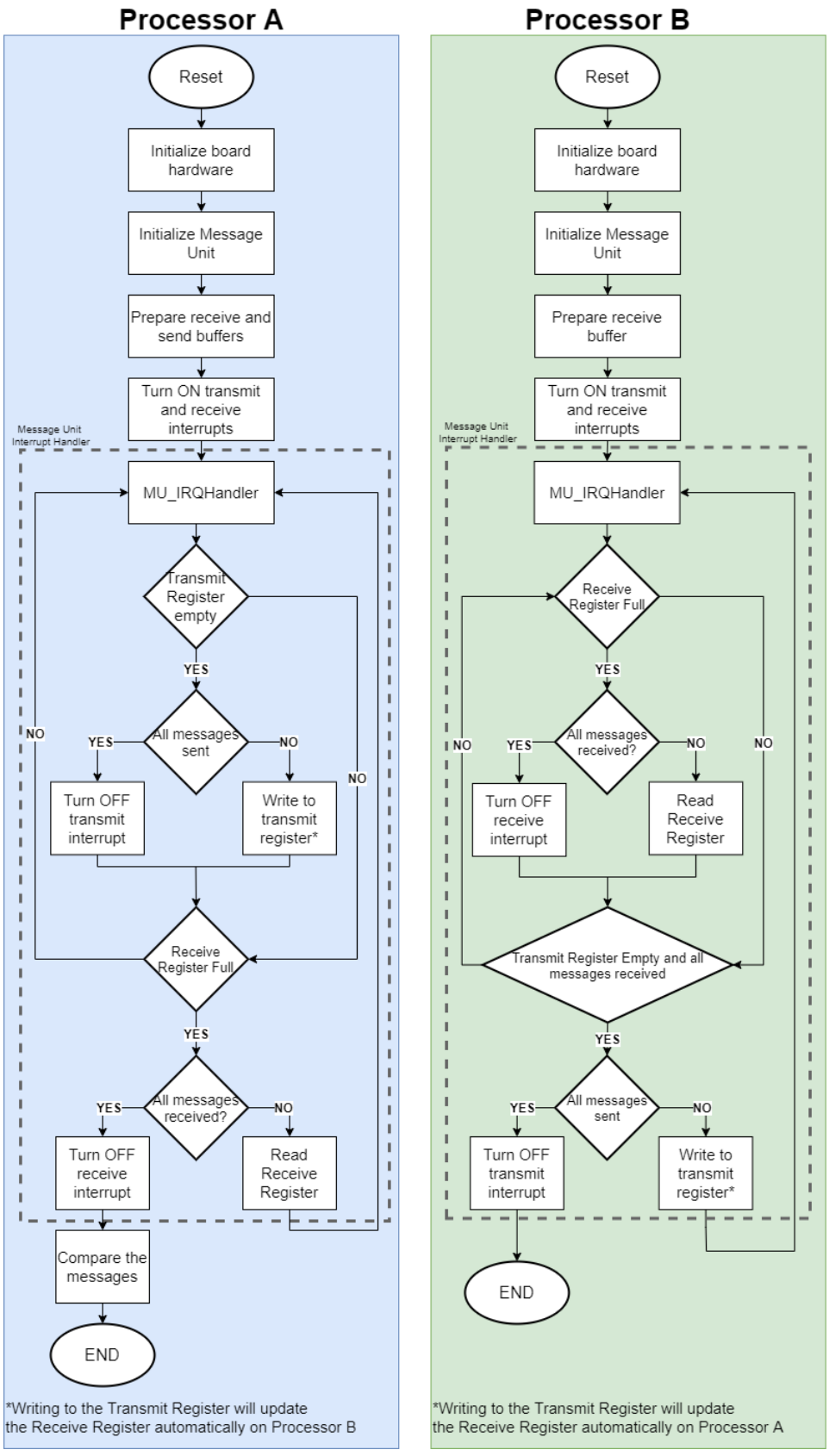

Figure 6. High-level description of the dsp_mu_interrupt SDK example

Figure 7 shows the result of running the project will be printed in the terminal.

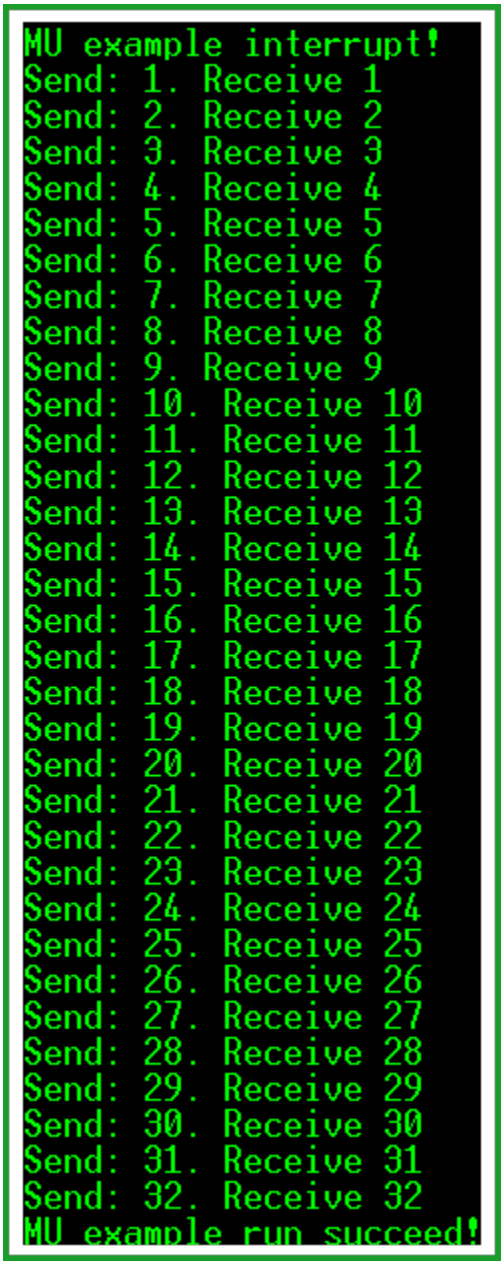

Figure 7. Message Unit interrupt messaging example output on a serial terminal

4.3 Messaging in polling mode implementation

The `dsp_mu_polling` demo application provided within the SDK sends and receives 32 messages between the two processors using polling instead of interrupts.

This example does the following actions:

- Processor A will be polling its Transmit Register. When it is empty, it will send the messages one by one to Processor B.
- Processor B will be polling its Receive Register. When it is full it will read the received message.

- When Processor B finishes receiving all the messages, it will be polling its Transmit Register. When it is empty, it will start sending the 32 received messages to Processor A.
- Processor A will be polling its Receive Register. When it is full, it will read thereceived message.
- When Processor A finishes receiving all the messages it will compare the received messages with the ones it sent to see that the communication was successful.

Figure 8 shows a high-level description of how this SDK example works. For more details regarding this example, refer to the SDK project.

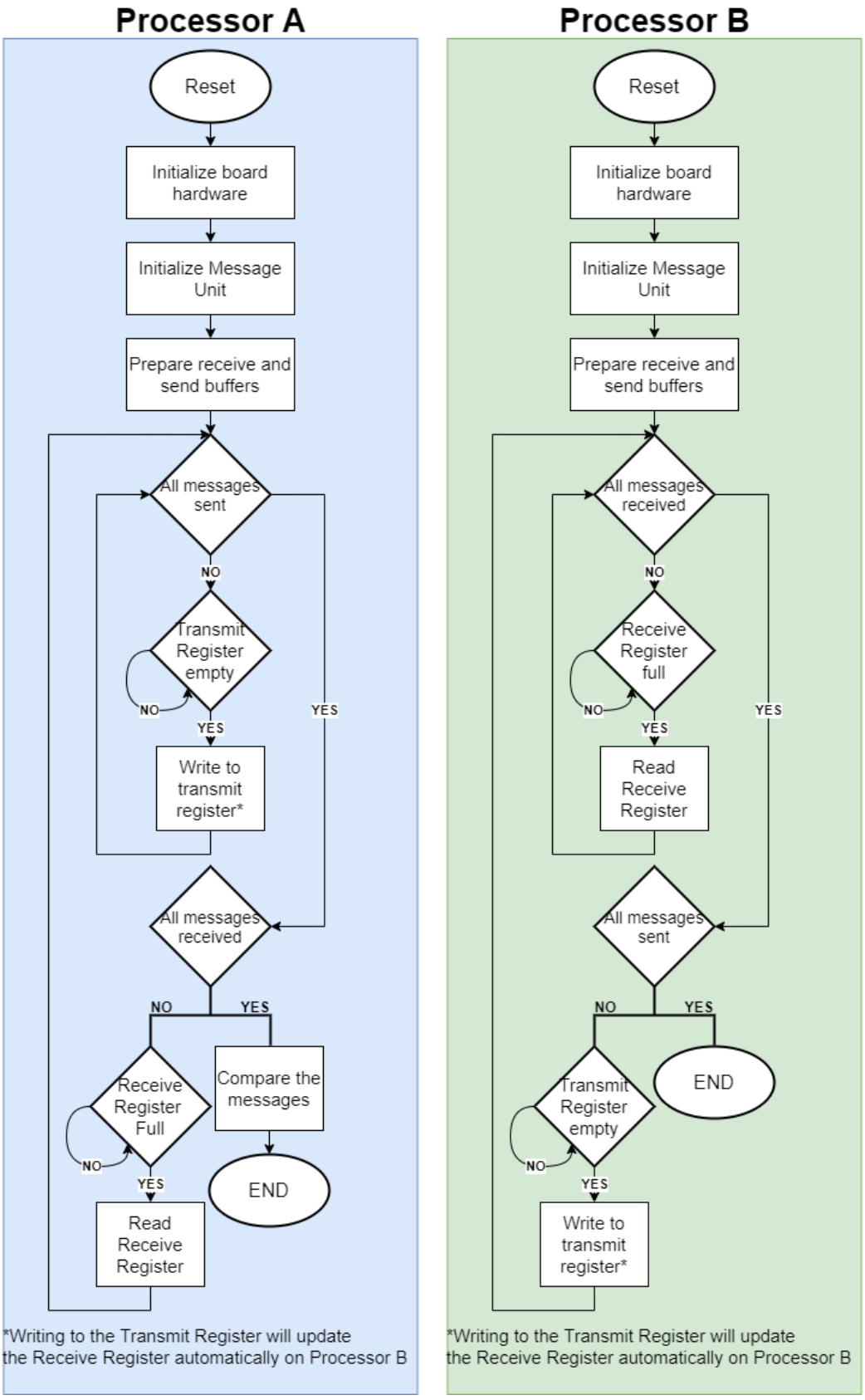

Figure 8. High-level description of the dsp_mu_polling SDK example

4.4 Semaphore block implementation

The `dsp_sema42` demo application provided within the SDK demonstrates how to use a semaphore gate to block an LED of the EVK.

This example does the following actions:

- Processor A core turns on LED D9 of the EVK and locks a semaphore gate.
- Processor B will wait until Processor A unlocks the semaphore gate.
- After user presses any key in a terminal window, the semaphore gate will be unlocked by the Processor A.
- Processor B will lock the semaphore gate and it will turn OFF the LED.

Figure 9 shows a high-level description of how this SDK example works. For more details regarding this example, refer to the SDK project.

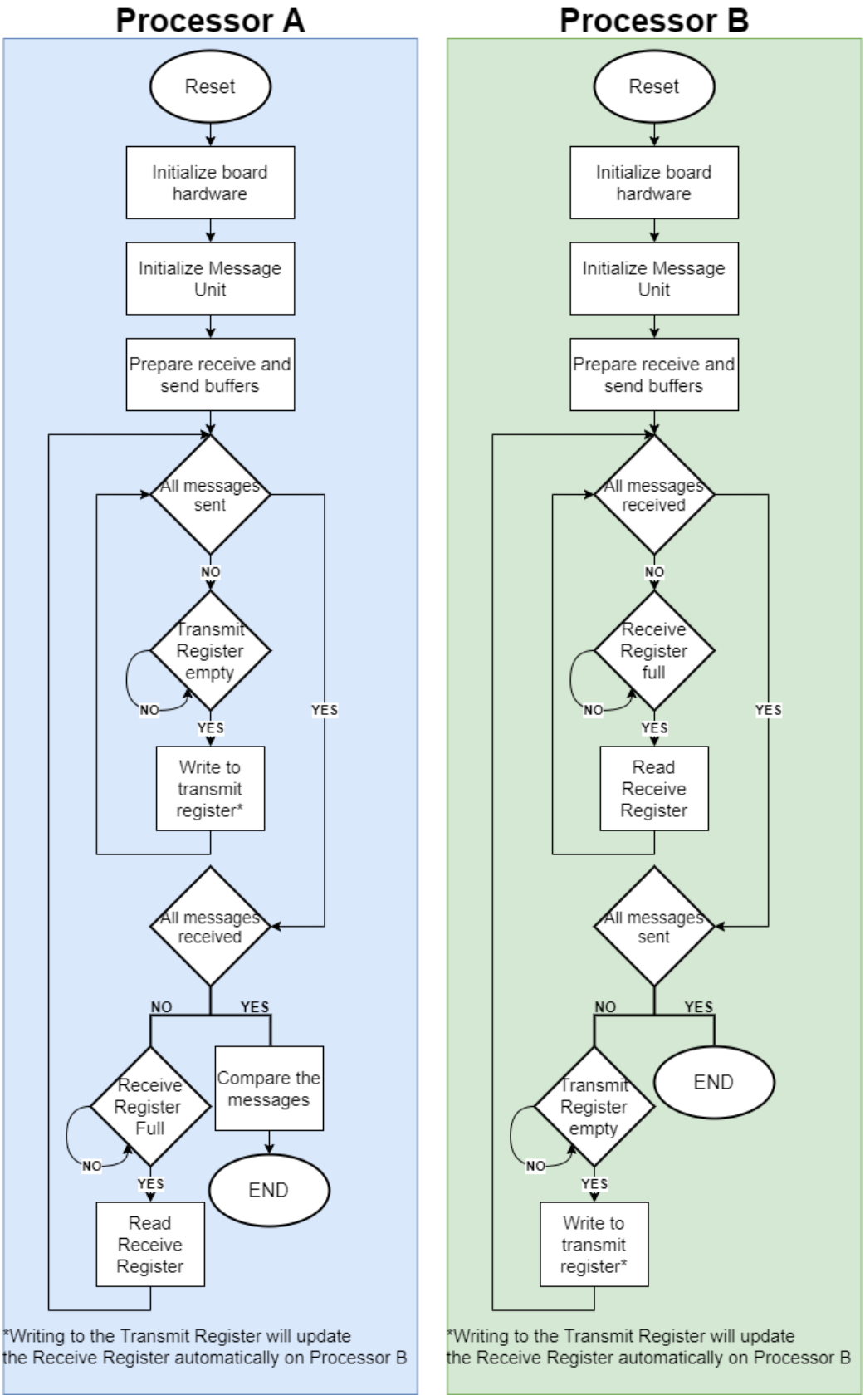

Figure 9. High-level description of the dsp_mu_sema42 SDK example

Figure 10 shows the result of running the project will be printed in the terminal.

```
Sema42 example!
Press any key to unlock semaphore and DSP core will turn off the LED
Now the LED should be turned off

Sema42 example succeed!
```

Figure 10. dsp_sema42 example output on a serial terminal

5 Debugging dual-core projects

5.1 Debugging system

The debug system on the dual-core of RT600 family has the following features:

- Supports Arm Serial Wire Debug mode for the Cortex-M33 and the HiFi4 DSP.
- Trace port provides Cortex-M33 CPU instruction trace capability. Output via a Serial Wire Viewer.
- Direct debug access to all memories, registers, and peripherals.
- No target resources are required for the debugging session.
- The Cortex-M33 includes instruction breakpoints that can also be used to remap instruction addresses for code patches.
- The Cortex-M33 includes data watchpoints that can also be used as triggers.
- Supports JTAG boundary scan.
- Instrumentation Trace Macrocell allows additional software controlled trace for the Cortex-M33.
- The HiFi4 DSP also includes address and data breakpoints and trace capability.

Figure 11 shows top-level debug ports and connections.

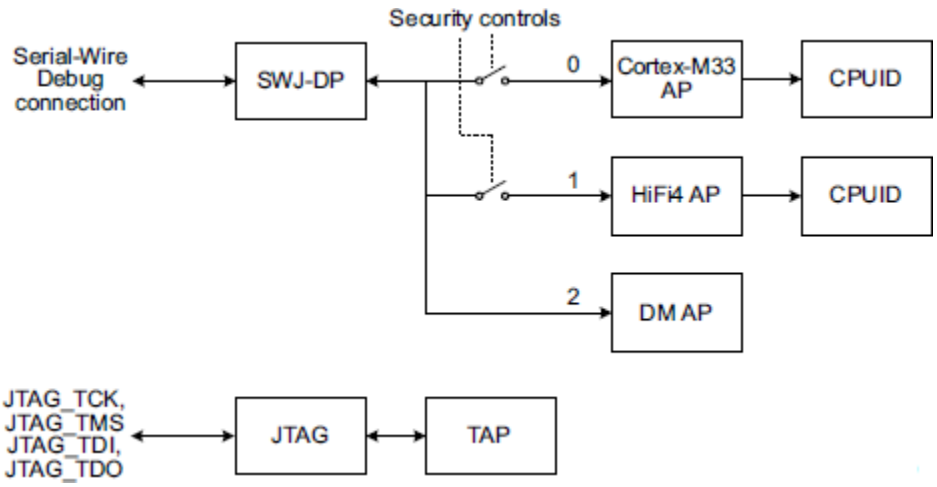

Figure 11. Serial wire internal connections

5.2 Basic configurations

Serial Wire Debug is the default function on pins PIO2_25 and PIO2_26 allowing for debug through reset.

5.3 Dual-core project debugging

5.3.1 Prerequisities

This section assumes that the following tools and software have already been installed on your personal computer:

- Xtensa Xplorer IDE
- RT600 DSP Build Configuration for Xtensa Xplorer IDE
- Xtensa On Chip Debugger Daemon
- MCUXpresso IDE V11.1.1 or newer
- SDK for the RT600-EVK v2.7.0 or newer
- JLink software V6.62c or newer

5.3.2 Programming LPC-Link2 with Segger-Jlink

Standalone debug probes could be used to debug HiFi4 DSP and Cortex-M33 over SWD. However, it's more convenient to use the onboard LPC-Link2 debug probe of the EVK. The RT600 EVKs has an LPC4300 MCU that is pre-programmed with CMSIS-DAP firmware. CMSIS-DAP is not compatible with HiFi4; due to this we need to flash J-Link firmware into the on-board debugger.

First, we need to install LPCScrypt. It can be downloaded from link. Make sure it is version 2.1.0 or later.

Program the LPC-Link2 debug probe with the following steps:

- Connect jumper JP1 of the EVK.
- Power up the EVK through USB J5.
- Run the script `program_JLINK.cmd` located at `LPCScrypt_Installation_Path\ LPCScrypt_2.1.1_15\scripts`. You will see the message shown in Figure 12.
- Press any key to begin the programming script for Segger-JLink firmware.
- If the script runs successfully, you will see what is shown in Figure 13.
- Disconnect the board from power.
- Remove JP1 jumper.

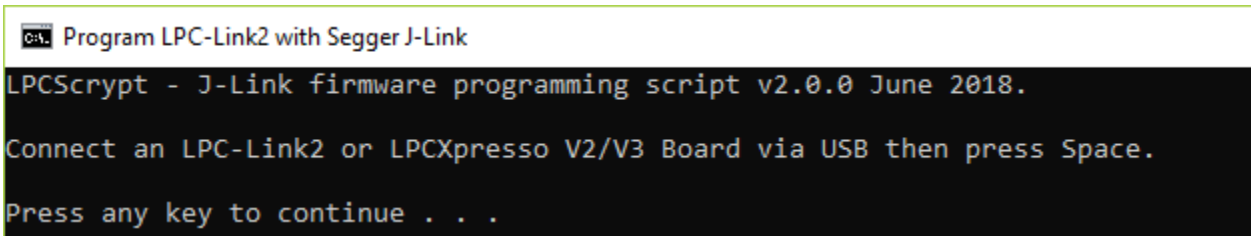

Figure 12. program_JLINK script

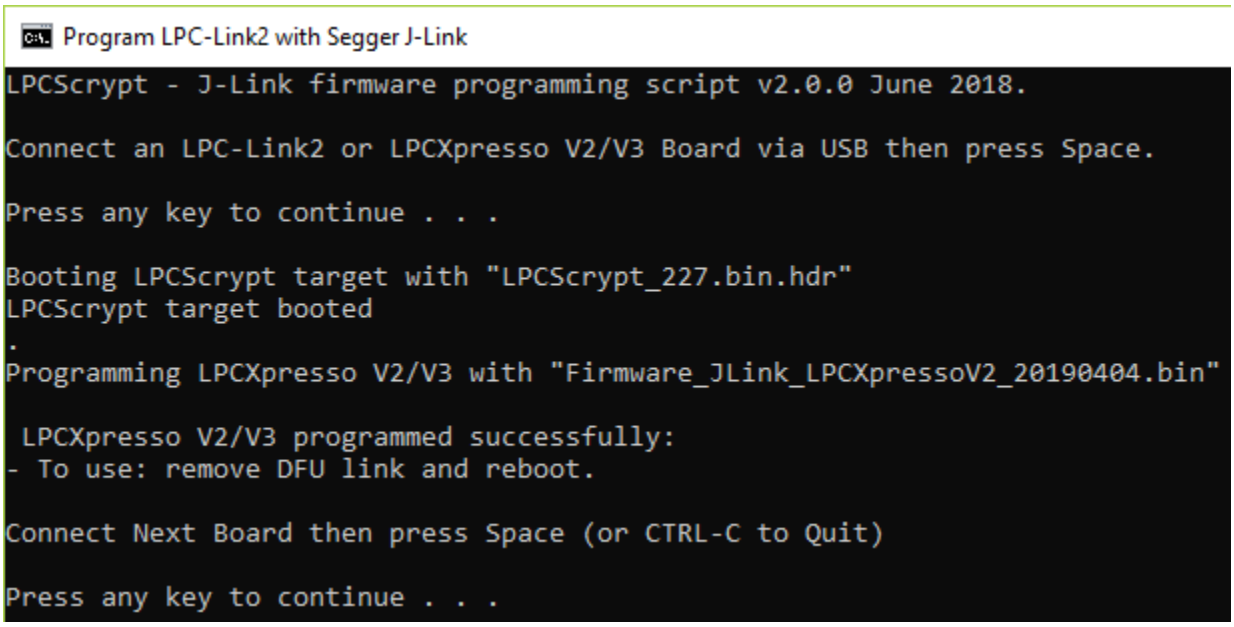

Figure 13. program_JLINK script ran succesfully

To check that the J-Link firmware is programmed succesfully, you can run the JLink commander. If everything goes well, you should be able to see the information of the debug probe as soon as you open JLink commander, as shown in Figure 14. Every LPC-Link2 has a different JLink serial number write down this number since it will be needed later.

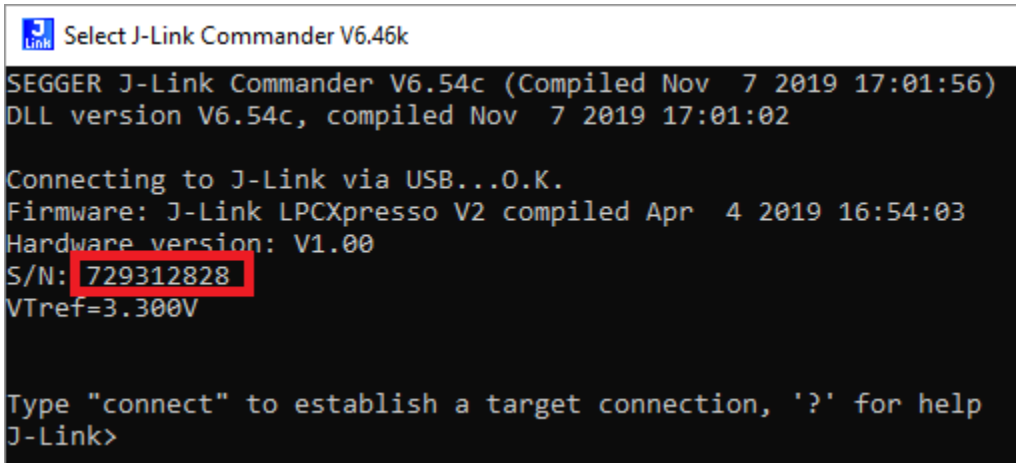

Figure 14. JLINK serial number

5.4 Debugging and running dual-core applications

5.4.1 Debugging and running MCUXpresso project

By default, the DSP core is not powered when the RT600 boots up. To run or debug DSP applications, you will first need to execute some code on the M33 core to initialize the DSP.

The DSP demos contained in the MCUXpresso SDK consist of two separate applications that run on the Arm core and DSP core.

For example, to debug the `dsp_mu_interrupt` project on the DSP you will first need to build and execute the M33 application using an environment of your choosing. MCUXpresso IDE is used in this document.

First, import the `dsp_mu_interrupt` example into your workspace. By default, the dsp examples on MCUXpresso IDE links the DSP pre-built application to RAM. When trying to debug multi-core applications this won't be the case because the DSP image will be loaded and debugged from Xtensa IDE. To modify this on MCUXpresso, open your project's properties and access the following path: **C/C++ Build > Settings > MCU C Compiler > Preprocessor**. In the **Preprocessor** section, set `DSP_IMAGE_COPY_TO_RAM=0` as shown in Figure 15.

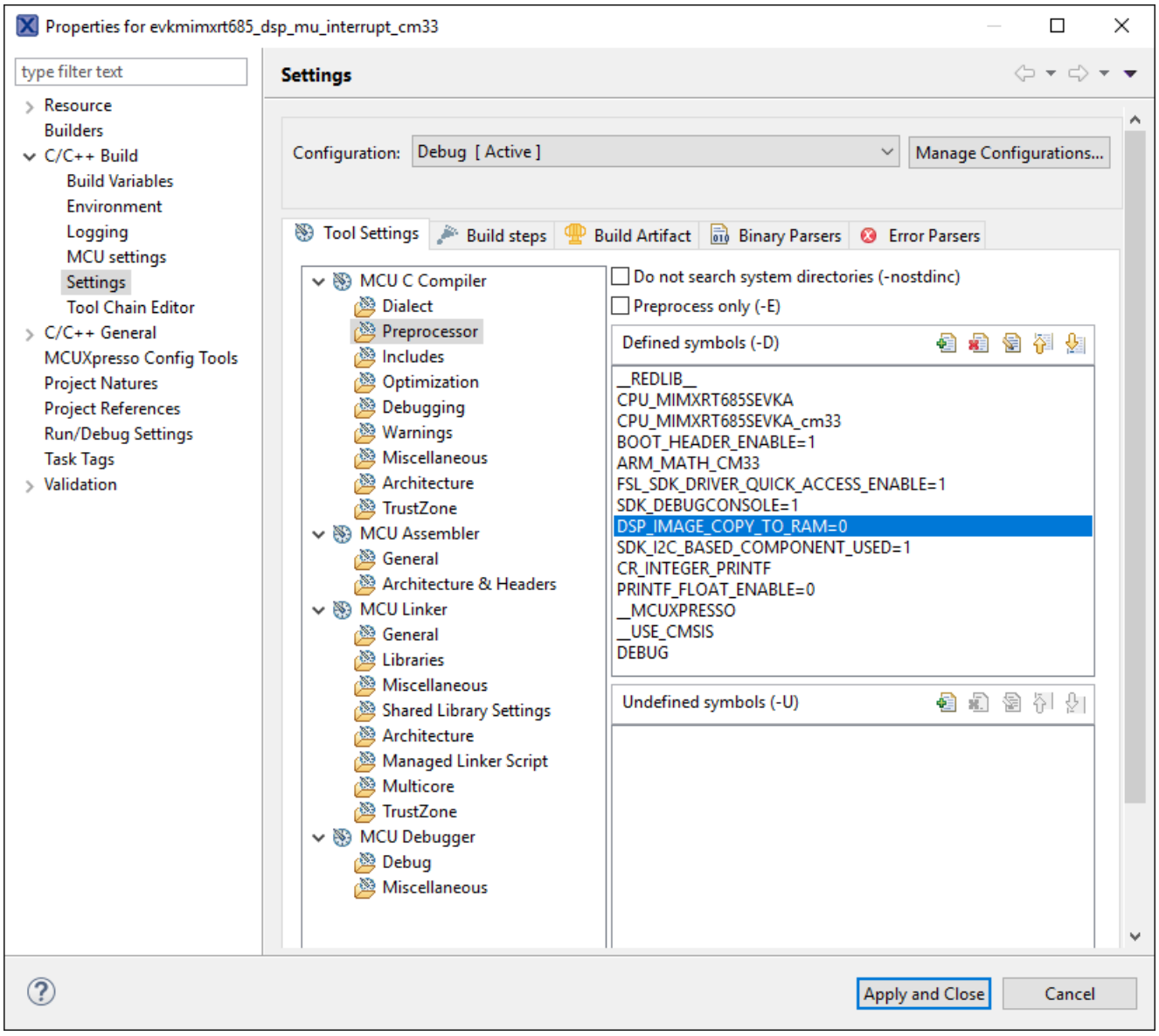

Figure 15. Properties of the MCUXpresso project

With this modification the M33 core will still initialize the DSP but it won't load the pre-built binaries to RAM. Having the DSP running is needed to be able to start the Xtensa Daemon debugger.

After making this modification you can build and launch the debug session normally. The debug session will be stuck in a while loop until the DSP boots-up as shown in Figure 16.

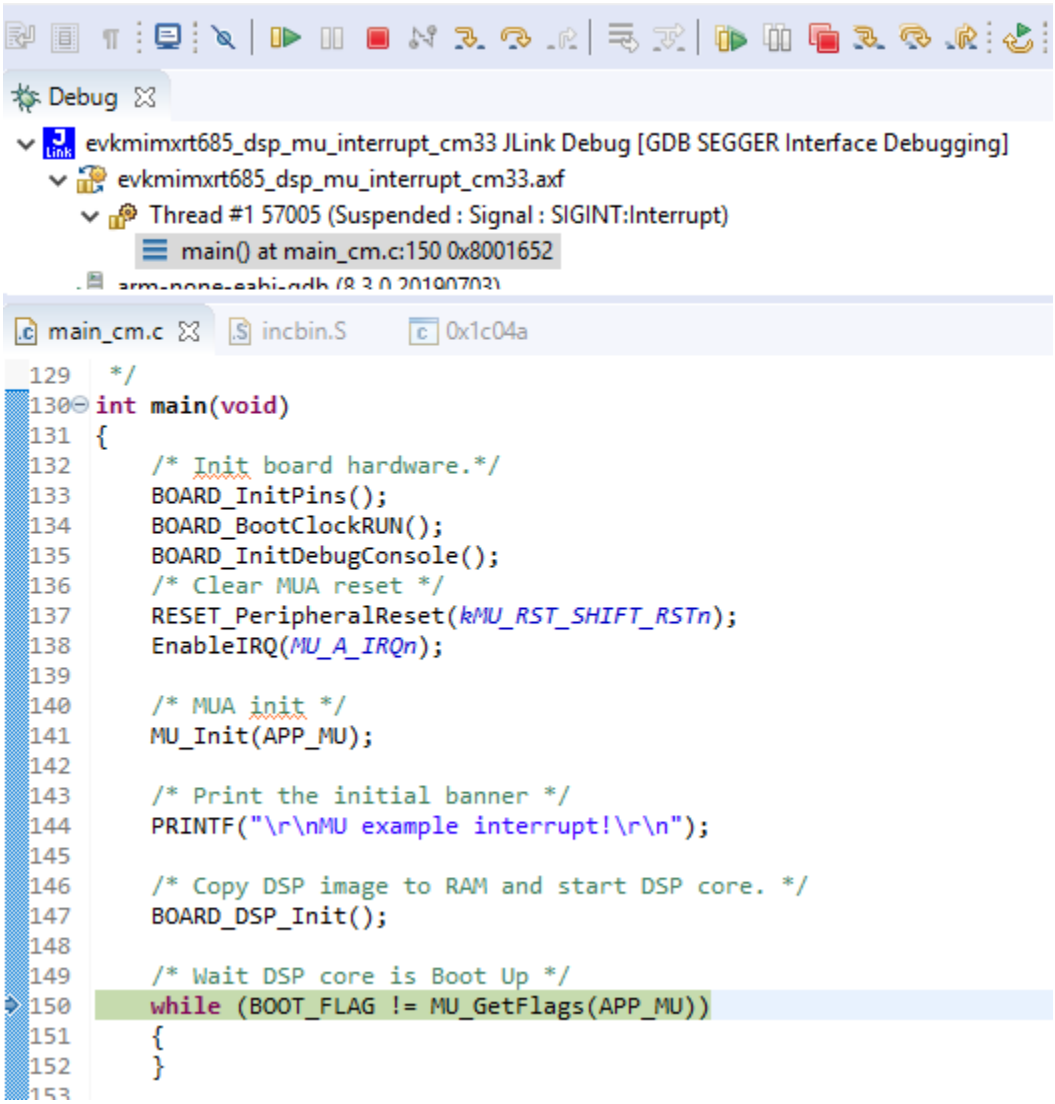

Figure 16. Debug session active on MCUXpresso IDE

SEGGER J-Link software version 6.62c or newer is required for compatibility with RT600. MCUXpresso IDE may ship with an older version, which can be changed within the MCUXpresso IDE preferences. Go to **Window > Preferences > MCUXpresso IDE > Debug Options > J-Link Options**, as shown in Figure 17.

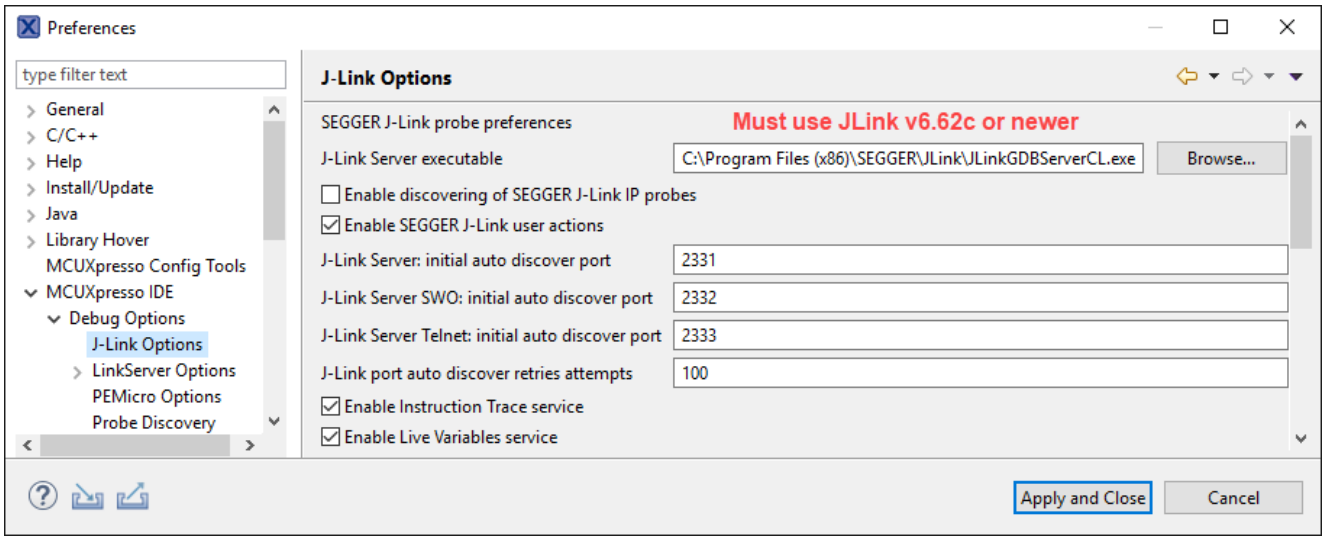

Figure 17. Changing JLink software version on MCUXpresso IDE

5.4.2 Starting Xtensa debugger Daemon

Explaining this procedure is outside of the scope of this document. Refer to *Getting Started with Xplorer for MIMXRT600.pdf* located under `<SDK_ROOT>/docs/` for information on how to start Xtensa Debugger Daemon.

NOTE

For this step it's necessary to have the JLink serial number on hand. For example, see Figure 14 for the probe used in this example.

5.4.3 Debugging and running Xtensa project

Once the debugger daemon is up and running, you can launch the debug configuration on Xtensa IDE.

1. Import the `dsp_mu_interrupt` into Xtensa IDE. Refer to *Getting Started with Xplorer for MIMXRT600.pdf* located under `<SDK_ROOT>/docs/` for information on how to import an SDK project into Xtensa IDE.
2. You will see `dsp_hello_world_hifi4` in the Project Explorer. Use the drop-down buttons on the menu bar to make a build selection for the project and hardware target configuration, as shown in Figure 18.

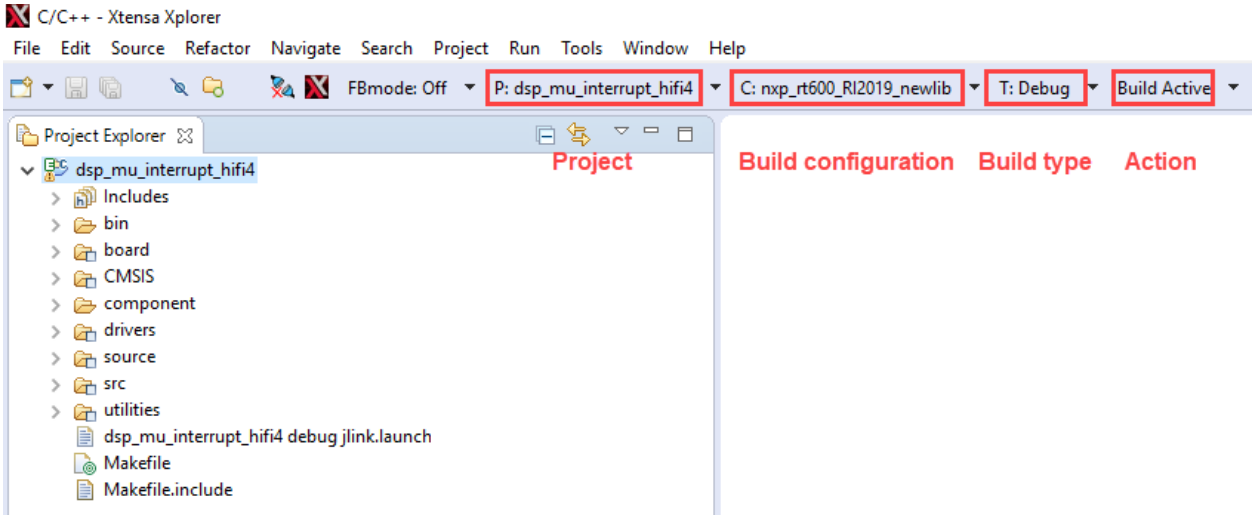

Figure 18. Build selection and hardware target configuration

3. Click on the action button (Build Active) to build the project.
4. Use the Debug action button on the right side of the menu bar to start the debug session. A default debug configuration is provided by the SDK project which will utilize the on-chip debugger.

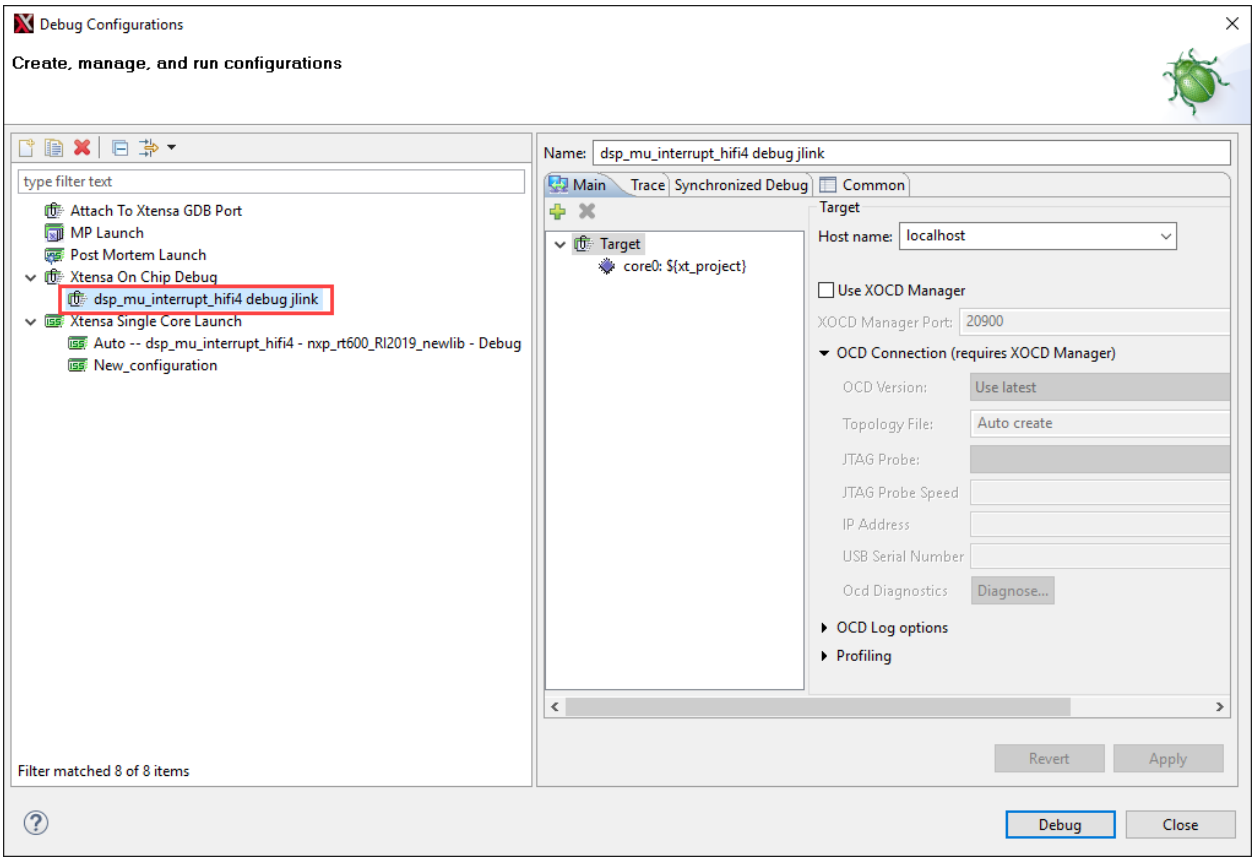

Figure 19. Launch debug session on Xtensa

5. Once the Debug button is selected, the actual debug on-chip will be started. Xplorer will ask you if you would like to download binaries to the hardware. Select **Yes**. Xplorer IDE will transit to the debug perspective after the binary download is finished. The debug perspective is as shown in Figure 20.

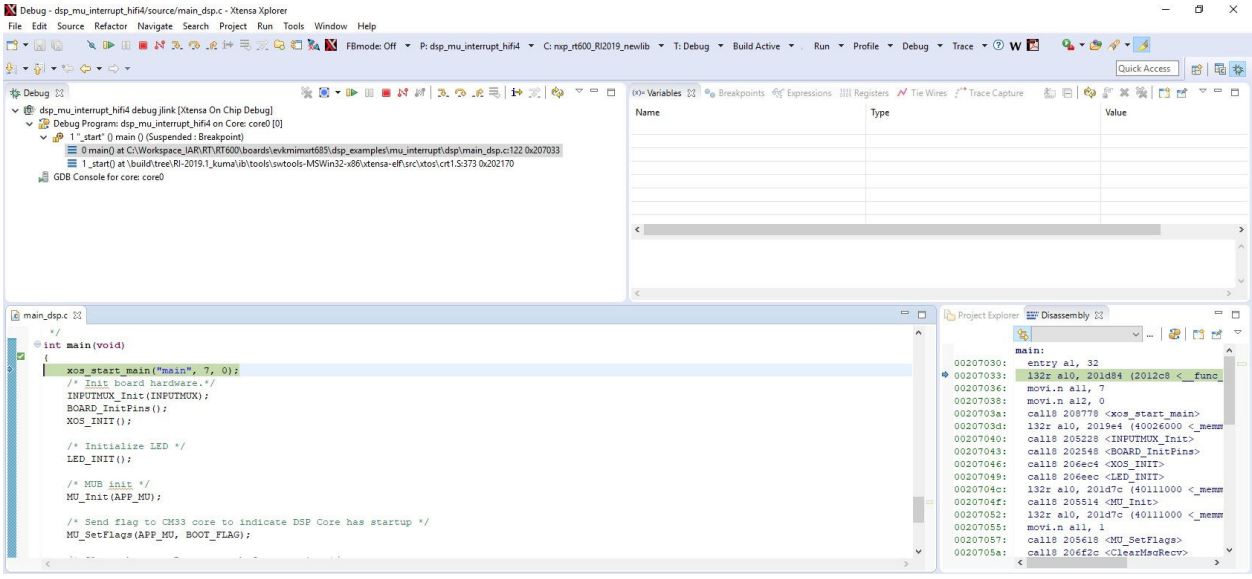

Figure 20. Debug perspective on Xtensa

Now you have both debug sessions up and running.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: March 20, 2020

Document identifier: AN12789

arm