AN12358

LPC55xx/LPC55Sxx Dual-Core Debug in MCUXpresso

Rev. 0 — 5 March 2019 Application Note

Contents

1 Introductions

1.1 Overview

The LPC55xx/LPC55Sxx is an Arm® Cortex®-M33 based microcontroller for embedded applications. These devices include:

- an Arm Cortex-M33 coprocessor,
- CASPER Crypto/FFT engine,
- PowerQuad hardware accelerator for DSP functions,
- up to 320 KB of on-chip SRAM,
- up to 640 KB on-chip flash,
- PRINCE module for on-the-fly flash encryption/decryption,
- high-speed and full-speed USB host and device interface with crystal-less operation for full-speed,
- SDIO/MMC,
- five general-purpose timers,
- one SCTimer/PWM,
- one RTC/alarm timer,
- one 24-bit Multi-Rate Timer (MRT),
- one Windowed Watchdog Timer (WWDT),
- one high speed SPI (50 Mhz),
- nine flexible serial communication peripherals (which can be configured as a USART, SPI, I^{2}C, or I^{2}S interface),
- Programmable Logic Unit (PLU),
- one 16-bit 1.0 Msamples/sec ADC,
- one comparator,
- one temperature sensor.

The LPC55xx/LPC55Sxx offers two Arm Cortex-M33 cores which have the features as below:

- Arm Cortex-M33 core (CPU0, r0p3)
  - — Running at a frequency of up to 100 MHz.
  - — TrustZone, Floating Point Unit (FPU) and Memory Protection Unit (MPU).
  - — Arm Cortex M33 built-in Nested Vectored Interrupt Controller (NVIC).
  - — Non-Maskable Interrupt (NMI) input with a selection of sources.

- — Serial Wire Debug with eight breakpoints and four watch points, including Serial Wire Output for enhanced debug capabilities.
- — System tick timer.

- Arm Cortex-M33 co-processor (CPU1, r0p3)
  - — Running at a frequency of up to 100 MHz.
  - — The configuration of this instance does not include MPU, FPU, DSP, ETM, and Trustzone.
  - — System tick timer.

1.2 Dual core basic mechanism

The dual core in the LPC55xx/LPC55Sxx is in the asymmetric architecture, which means that one core (CPU0) is the master core and the other (CPU1) is the slave one. By default, CPU0 is set as the master and it can work normally, while CPU1 is set as the slave and it is on hold and its clock is disabled with the chip startup. Release the slave core and enable its clock via a register by the master core.

With dual core working, they need to communicate with each other, so the LPC55xx/LPC55Sxx provides the Inter-CPU Mailbox mechanism with the below features.

- Provides the Inter-Processor Communication, allowing multiple CPUs to share resources and communicate with each other in a simple manner.
- Each CPU can cause up to thirty-two user defined interrupts to its partner.
- Each CPU can claim a shared resource if it is available.
- Provides a mutual exclusion configuration for the communication handshake.

1.3 Related system resources

The Arm Cortex M33 includes three AHB-Lite buses, one system bus and the I-code and D-code buses. One bus is dedicated for instruction fetch (I-code), and one bus is dedicated for data access (D-code). The use of two core buses allows for simultaneous operations if concurrent operations target different devices.

Both CPUs share all resources (memories and peripherals) in the LPC55xx/LPC55Sxx. To get better performance on dual core usage.

The LPC55xx/LPC55Sxx support 320 KB SRAM with separate bus master access for higher throughput and individual power control for low-power operation. This makes it possible that the code and data of both CPUs can be separated to be stored and accessed.

LPC55xx/LPC55Sxx uses a multi-layer AHB matrix to connect the CPU buses and other bus masters to peripherals in a flexible manner that optimizes performance by allowing peripherals that are on different slave ports of the matrix to be accessed simultaneously by different bus masters.

1.4 Debug system

The debug system on the dual-core of LPC55xx/LPC55Sxx has the following features.

- It supports arm serial wire debug mode for **CPU0** and, if present, **CPU1**.

- Trace port provides Cortex-M33 CPU instruction trace capability on both **CPU0** and **CPU1**. Output via a serial wire viewer.
- Direct debug access to all memories, registers, and peripherals.
- No target resources are required for the debugging session.
- Breakpoints: **CPU0** and **CPU10** include eight instruction breakpoints.
- Watch-points: **CPU0** and **CPU10** M33 include four data watch-points that can also be used as triggers.
- Supports JTAG boundary scan.
- Instrumentation Trace Macrocell allows additional software controlled trace for **CPU0** and **CPU1**.

The debug system has the following modules.

- **JTAG-TAP**: Test access port is used by NXP Product & Test Engineering team.
- **DAP**: Debug access port which has Serial Wire port (**SWJ-DP**) which interprets the data coming in and routes to appropriate Access Port (AP).
- **CPU0 AP**: Debug access port for Cortex-M33 core instantiated as **CPU0**.
- **CPU1 AP**: Debug access port for Cortex-M33 core instantiated as **CPU1**. This instance of CM33 does not have security extension (TrustZone for Armv8-M).
- **DM-AP**: Debug Access port for debug mailbox.
  - — This port is always enabled, and external world can send and receive data to/from ROM.
  - — This port is used to implement NXP debug authentication protocol version 1.0.

Figure 1. on page 3 shows top-level debug ports and connections.

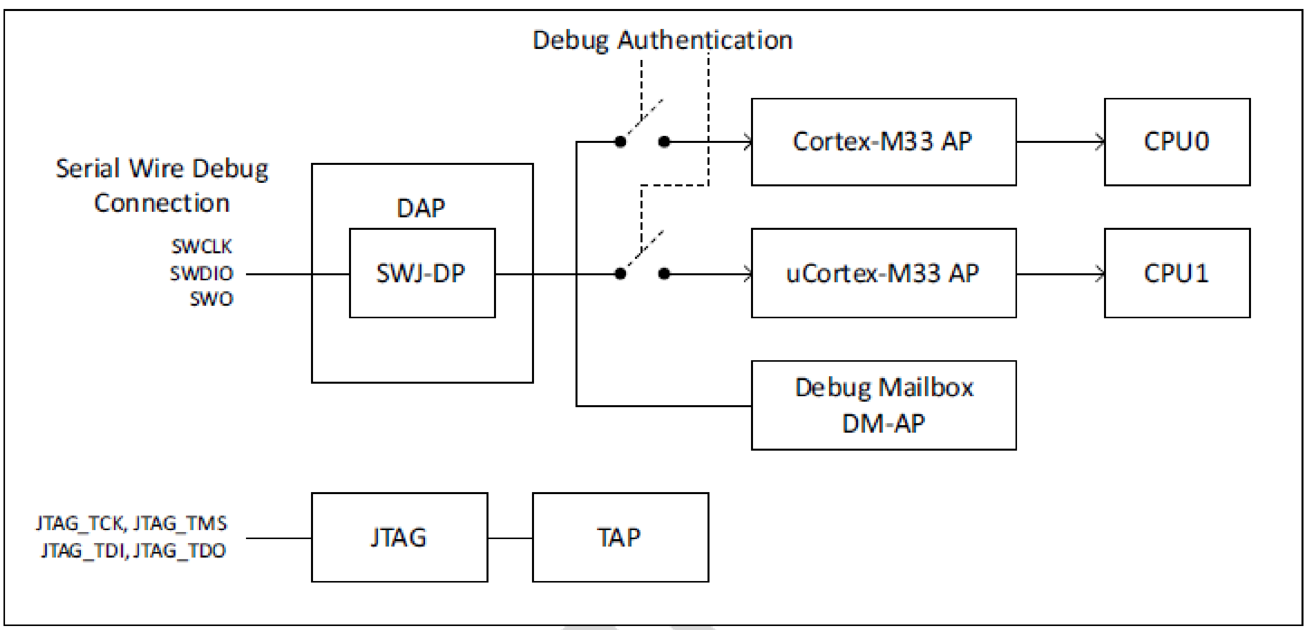

Figure 1. Serial wire debug internalconnections

2 Debug environments

- Hardware:
  - — Mini/Macro USB cable.
  - — LPCXpresso55s69 board.

— Personal computer.

- Toolchain:
  — MCUXpresso IDE V10.3.0.
- Software:
  — SDK_2.5.0_LPCXpresso55s69_EAR_3.

3 Dual core project configurations in MCUXpresso

Dual core project can be designed in many ways, however within MCUXpresso IDE there is an underlying expectation that one core (Master) will control the execution (or at least the startup) of code running on other core (Slave). The dual core projects in MCUXpresso IDE consist of two linked projects – one project containing the Master code and the other containing the Slave code.

After a power-on or Reset, the Master core boots and is then responsible for booting the Slave core. However, this relationship only applies to the booting process; after boot, the application may treat either of the cores as the Master or the Slave.

MCUXpresso IDE allows for the easy creation of **linked** projects that support the targeting of Multicore MCUs. This application note only mentions the project configurations for dual-core based on the dual core mutex example in the LPC5500 SDK (The path is boards\lpcxpresso55s69\driver_examples\mailbox\mutex.) and does not introduce how to create them (for the information, can read the MCUXpresso IDE user guide under the folder of MCUXpresso IDE). Concerning the dual-core mode, the Master and Slave projects contain the following configurations:

1. Memory configurations;
2. Architecture configurations;
3. Multicore configurations.

First of all, launch the MCUXpresso IDE workspace and install the LPC55S69 SDK and import the mailbox mutex examples on both master and slave core. Then, SDK and examples can be shown in MCUXpresso, as shown in Figure 2. on page 4.

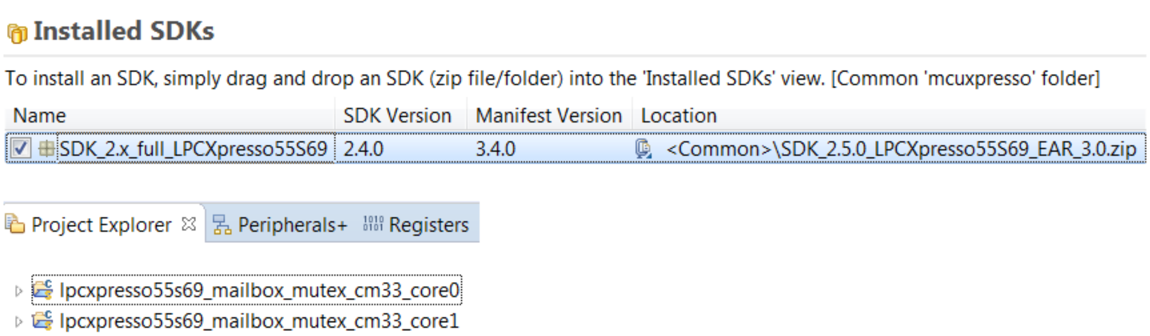

Figure 2. Installed SDK and imported dual-core projects in MCUXpresso

3.1 Memory configurations

The memory configurations must be managed to avoid unintended overlap. In the projects, the Slave application is executed entirely from a RAM location unused by the Master. Table 1. Address assignment for dual-core images on page 4 shows the address assignment for dual-core images.

Table 1. Address assignment for dual-core images

CPU	Storage address	Loading address
Master core	0x0	0x0
Slave core	up to compiler (in flash)	0x20033000

Correspondingly, Figure 3. on page 5 shows the memory configurations on both projects in MCUXpresso. The highlighted named as **Ram1** (Alias is **RAM2** in Master project and **RAM** in Slave project) is the execution location of the Slave application. The alias **RAM2** is referred in multicore configurations (see Figure 5. on page 6).

Memory details (LPC55S69)* -- Master

Default LinkServer Flash Driver

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0xa0000	LPC55xx.cfx
RAM	Ram0	RAM	0x20000000	0x31800	
RAM	Ram1	RAM2	0x20033000	0x11000	

Memory details (LPC55S69)* -- Slave

Default LinkServer Flash Driver

Type	Name	Alias	Location	Size	Driver
RAM	Ram1	RAM	0x20033000	0x11000	

Figure 3. Memory configurations in Master and Slave projects

3.2 Architecture configurations

As introduced above, the Master and Slave cores are both based on Arm Cortex-M33 though there are some differences. The architecture is configured correspondingly in the Master and Slave projects, as shown in Figure 4. on page 5.

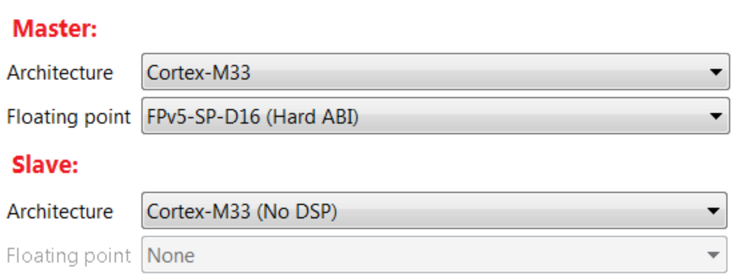

Figure 4. Architecture configurations in Master and Slave projects

3.3 Multicore configurations

The multicore configurations show the link between the Master and Slave projects, as shown in Figure 5. on page 6. It indicates the Slave (named as **M33SLAVE**) application image is integrated in Master memory region. The alias of the Master memory region for the Slave image is called as **RAM2**. This is consistent with the memory configurations (see Figure 3. on page 5).

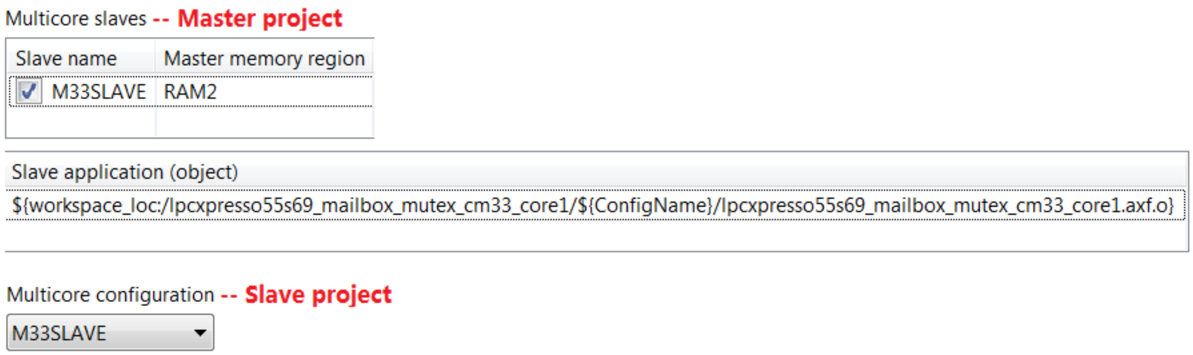

Figure 5. Multicore configurations in Master and Slave projects

4 Dual core project debug in MCUpresso

4.1 Launch debug session

After building both projects successfully, launch the debug session of the Master project first. To do it, click the project name in the workspace to select the Master project, and then click the **Debug** in the **Quick Panel**.

NOTE

The USB cable should be connected to the USB jack marked with **Debug Link** on the LPCXpresso55S69 board for debugging.

For the first launch, select the SWD device – CPU core. The Device **0** should be selected for the Master core/project and the Device **1** for the Slave one. Figure 6. on page 6 shows the SWD configuration for the Master debug session.

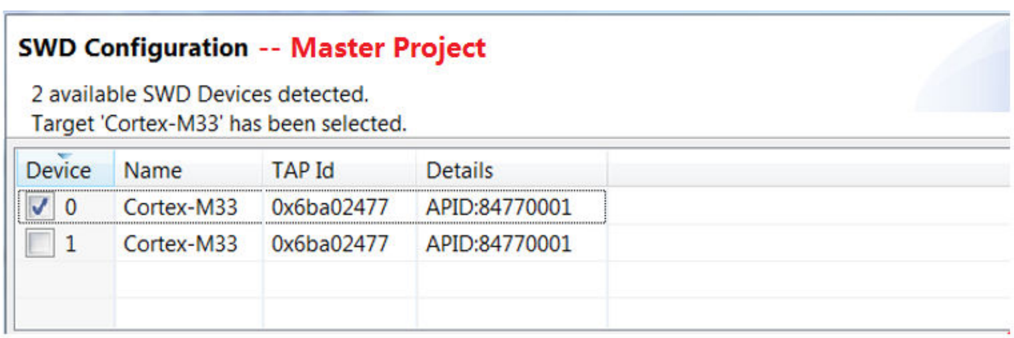

Figure 6. SWD configuration when first launching debug session

After both debug sessions launching is completed, the Master debug thread is suspended (the Master PC pointer stops at main() in the code based on the breakpoint setup in this project) and the Slave debug thread can not stop for debugging since the Slave core is on hold and required to be released by the Master. Figure 7. on page 7 shows the states of both debug threads in the **Debug** window.

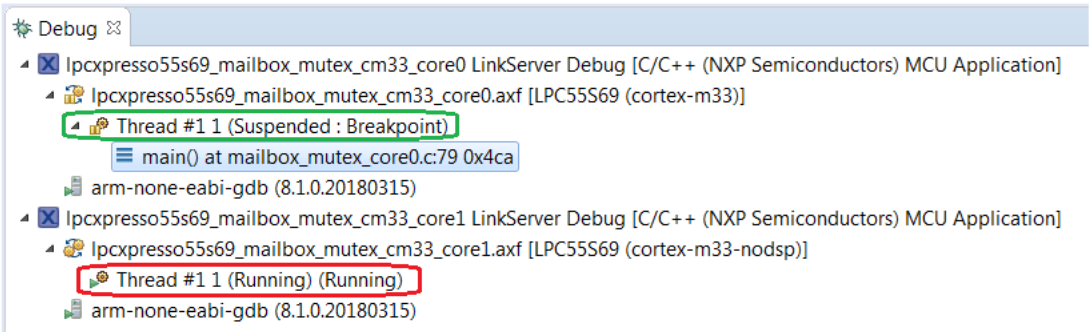

Figure 7. Debug threads for both projects

4.2 Slave debug setup

At this time, the debug operations, such as, step over, run, and so on, can be performed on the Master project.

In Figure 7. on page 7, click the green or red block in the **Debug** window to select the Master or Slave thread for debugging. After step over the code line of starting up the slave core in the Master project (see the coded highlighted in green in Figure 8. on page 7), the Slave core will be released to work. The current debug thread will be switched to the Slave and the Slave PC pointer will be stopped at main() in the Slave code (See Figure 9. on page 8). It means that the debugging control (e.g. step over) can be performed on the Slave project as well.

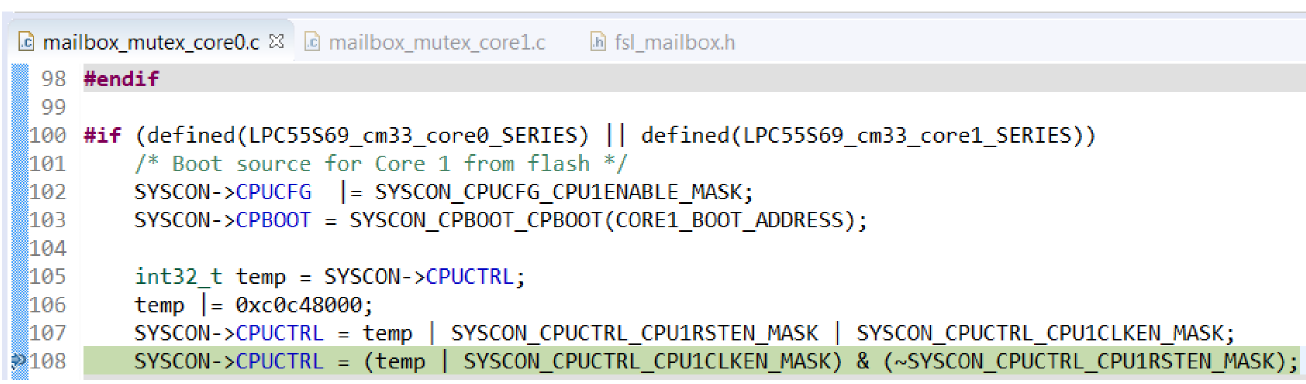

Figure 8. Codes of starting up the slave core

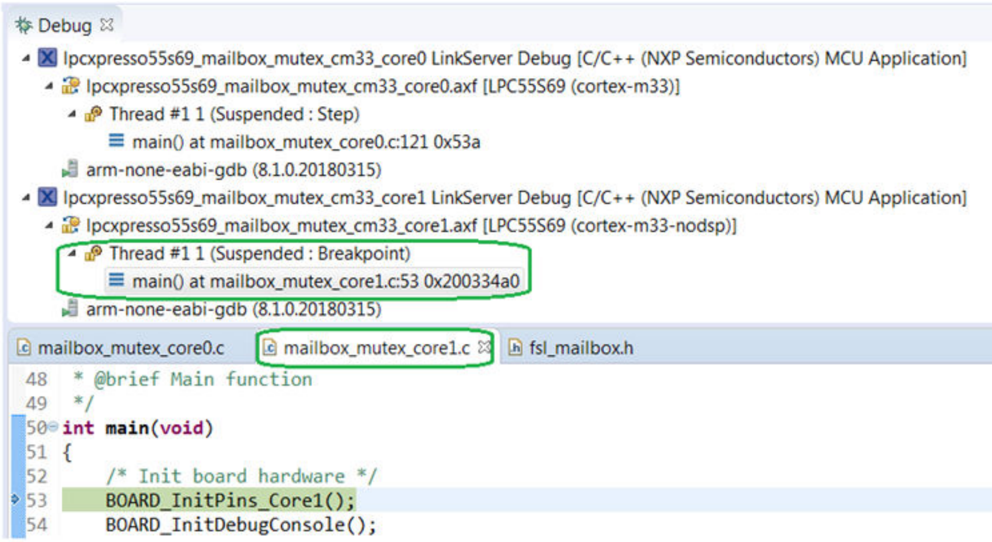

Figure 9. Slave core released for debugging

4.3 Dual-core communication debug

With switching to the Slave debug thread, before performing the debugging control on the dual core communication process, add the shared variable - *g_shared which is the shared data to be communicated between dual core via the Mailbox to Global Variables window in Slave debug session (g_shared is the pointer to the address of *g_shared). As an initial state, the current value of the variable is random, and the current address is 0 as defined in the Slave code. It is shown as Figure 10. on page 8.

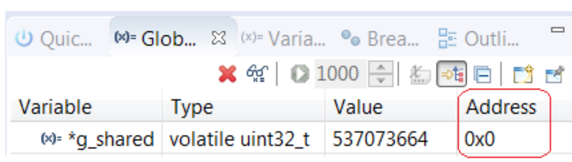

Figure 10. Initial shared variable in slave debug session

Begin and keep stepping over the Slave codes into the while(1) loop where the implemented functions are to get Mailbox Mutex control, update the global variable and set Mailbox Mutex control. Figure 11. on page 9 shows the loop codes.

```
while (1)
{
    /* Get Mailbox mutex */
    while (MAILBOX_GetMutex(MAILBOX) == 0)
        ;

    /* The core1 has mutex, can change shared variable g_shared */
    if (g_shared != NULL)
    {
        (*g_shared)++;
        PRINTF("Core1 has mailbox mutex, update shared variable to: %d\r\n", *g_shared);
    }

    /* Set mutex to allow access other core to shared variable */
    MAILBOX_SetMutex(MAILBOX);
}
```

Figure 11. Main loop process on slave core

And it can be observed that the code of getting Mailbox Mutex is stepped over. However, the global variable - *g_shared will not be updated since the initial address is zero (NULL) and still not transmitted from the Master via Mailbox. The address value of *g_shared keeps as zero in the **Global Variable** window, as shown in Figure 10. on page 8.

Click the green block in the **Debug** window (as shown in Figure 7. on page 7) to go to the Master debug thread. Send the address of the shared variable - g_shared to the Slave core via Mailbox. Click the **mailbox_mutex_core0.c**, and it switches to the Master code window. Before performing the debugging in the Master debug session, add the global variable – g_shared in the Global Variable window for observation. The initial value is zero and the address is 0x20000024. It is as shown in Figure 12. on page 9 .

Figure 12. Initial shared variable in master debug session

After stepping over the code line to send the address of shared variable – g_shared to the Slave core by Mailbox (see Figure 13. on page 9), go back to the Slave debug thread by clicking the red block in the **Debug** window (as shown in Figure 7. on page 7) and return to the main() of the Slave code.

mailbox_mutex_core0.c | mailbox_mutex_core1.c

```
/* Send address of shared variable to CM33 core1 by Mailbox*/
MAILBOX_SetValue(MAILBOX, kMAILBOX_CM33_Core1, (uint32_t)&g_shared);
```

Figure 13. Master code line to send address of shared variable

It can be observed that the shared variable - *g_shared in the **Global Variable** window has the same value (0) and address (0x20000024) (see Figure 14. on page 9) synchronized with the Master once the window display is refreshed, e.g. perform the **step over** operation. It indicates that the Slave core has received the address of the shared data successfully via Mailbox.

Figure 14. Synchronized shared variable in slave debug session

After stepping over the code (*g_shared)++;, it can be observed that the value of the shared variable *g_shared changes to 1 (see Figure 15. on page 10).

Figure 15. Updated shared variable in slave debug session

After stepping over the code of setting Mutex (MAILBOX_SetMutex(MAILBOX);), it means the access to shared resource is available to either of the cores. Currently, switch to the Master thread for debugging the Master while(1) loop which has the similar functions as the Slave: get Mailbox Mutex, update the shared variable g_shared and set Mailbox Mutex (see Figure 16. on page 10). The code of getting Mutex will be stepped over when performing the **step over**. Meanwhile, the value of the shared variable g_shared in the **Global Variable** window will be refreshed to 1 (see Figure 17. on page 10), as it has been changed by the Slave core.

```
    while (1)
    {
        /* Get Mailbox mutex */
        while (MAILBOX_GetMutex(MAILBOX) == 0)
            ;

        /* The core0 has mutex, can change shared variable g_shared */
        g_shared++;

        PRINTF("Core0 has mailbox mutex, update shared variable to: %d\r\n", g_shared);

        /* Set mutex to allow access other core to shared variable */
        MAILBOX_SetMutex(MAILBOX);
    }
```

Figure 16. Main loop process on master

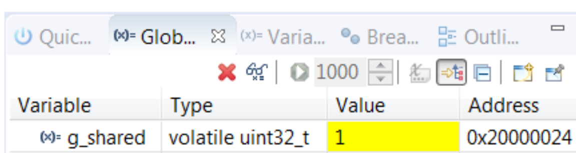

Figure 17. Synchronized shared variable in master debug session

After stepping over the code g_shared++, the value of the shared variable g_shared in the **Global Variable** window can be observed to change to 2. Before stepping over the code of setting Mutex (MAILBOX_SetMutex(MAILBOX);) in the Master code, switch to the Slave debug thread and perform the **step over**, it will keep stuck at the code of getting Mutex, as the access is still not released by the Master core. Switch to the Master debug thread and step over the code of setting Mutex for releasing the Mutex control.

Afterwards, go back to the Slave debug thread and perform the **step over**. At present, it can be observed that the code is stepped over as the Mutex control can be obtained. Similarly, the Master debug thread will be stuck at the code of getting Mutex if the Slave code of setting Mutex is not executed after the code of getting Mutex has been done.

Next, if repeat switching between the Master and Slave debug thread to doing debugging control as the above operations, it will be observed that the shared variable is being increased in turn by the Master and Slave core. Only the core that gets the Mutex control successfully can make a change to the shared variable. It means, if keep running one core's codes during debugging, the shared variable will be increased by the core only.

5 Conclusion

This application note has a quick look at the dual-core mechanism and debug system in LPC55xx/LPC55Sxx. Then it briefly introduces some configurations related to dual-core projects and elaborates how to debug dual-core projects in MCUXresso IDE based on the driver example of mailbox_mutex in LPC5500 SDK.

It is easy and convenient for users to debug dual-core projects in MCUXpresso. The debug sessions of both cores can be launched, and the debug process can be completed in one workspace. Just clicking the Master thread or Slave thread in the Debug window will switch either of dual core projects for debug. The debug operations, e.g. set breakpoint, variable view, for each core are same to a single core.

NOTE

Run the Master debug thread before the Slave core is started up by the Master.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

Date of release: 5 March 2019

Document identifier: AN12358

arm