AN12335

LPC55xx/LPC55Sxx Dual Core Communication

Rev. 0 — 23 January 2019 Application Note

Contents

1 Introduction

1.1 Overview

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

- an Arm Cortex-M33 coprocessor,
- a CASPER Crypto/FFT engine,
- a 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^2C, or I^2S 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:

- 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. Includes 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 asymmetric architecture, which means one core (CPU0) is a master one and the other (CPU1) is a slave one. The CPU0 is factory set to the master which can work normally and the CPU1, slave core, is on hold and its clock is disabled with the chip startup. To let it work, the slave core needs to be released and its clock can be enabled through a register by the master core.

With the dual-core running mode, they need to communicate with each other. The LPC55xx/LPC55Sxx provides a simple means called Inter-CPU Mailbox mechanism which has the following features:

- Provides a means 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 dual-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 following mechanisms are supported.

1.3.1 Multi-bank SRAM

The LPC55xx/LPC55Sxx support 320 KB SRAM with separate bus master access for higher throughput and individual power control for low-power operation. The 320 KB SRAM consists of 32 KB SRAM on Code Bus, 272 KB SRAM on System Bus (272 KB is contiguous), and additional 16 KB USB SRAM. This makes it possible that the code and data of both CPUs can be separated to be stored and accessed.

1.3.2 AHB multi-layer matrix

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. The multilayer matrix block diagram in Figure 1. on page 3 shows details of the available matrix connections. The highlighted bus shows the memory access at best performance on dual core application.

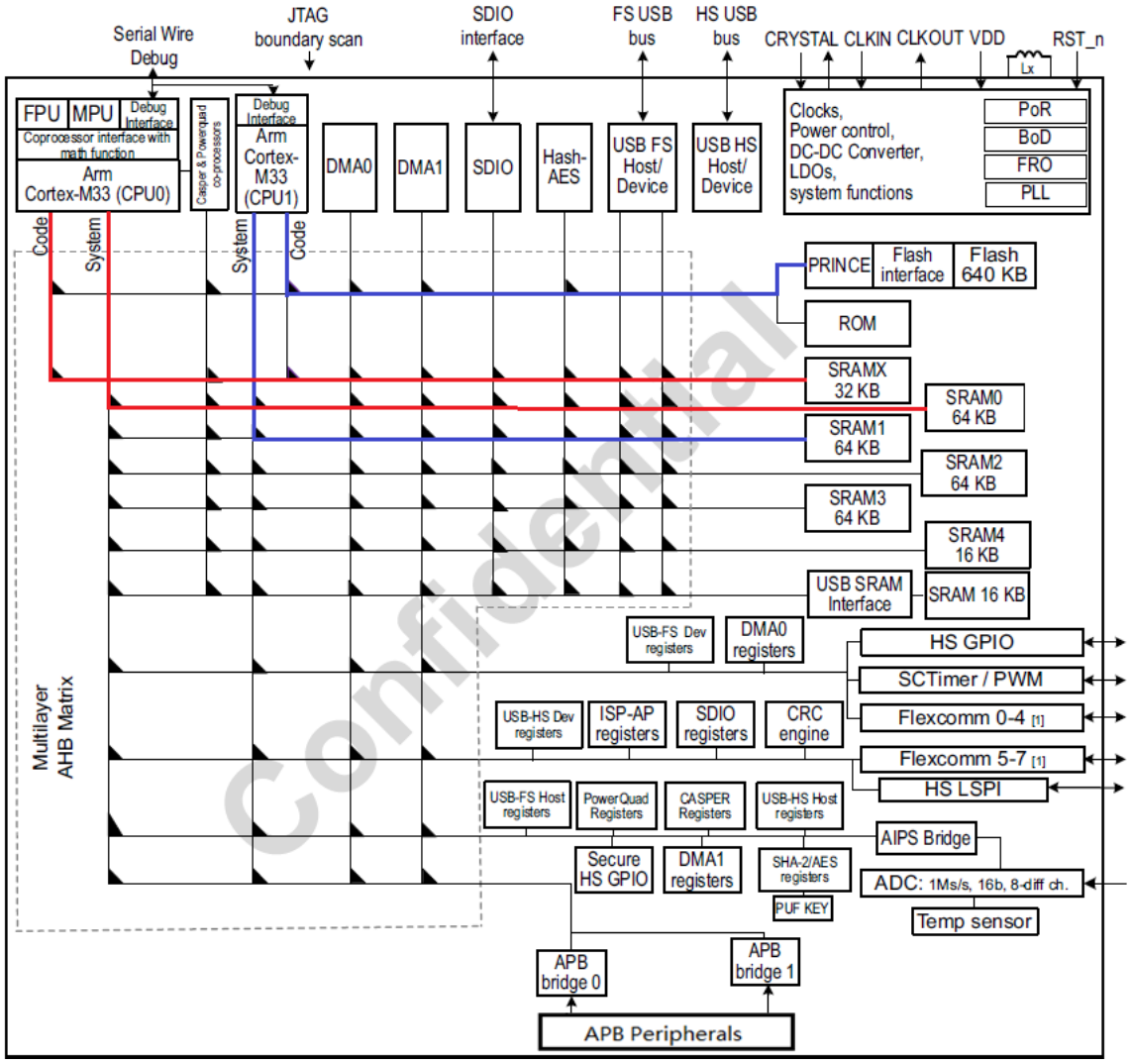

Figure 1. Multilayer matrix block diagram

2 Dual core implementation

The typical implementation of dual core contains the following processes:

- How to load slave core (CPU1) image on the master core (CPU0);
- How to start up slave core (CPU1) on the master core (CPU0);
- How to communicate between the slave core (CPU1) and the master core (CPU0).

These implementations will be introduced based on the driver example of mailbox_mutex in LPC55xx SDK.

2.1 Load slave core image

To get good performance, the slave core image should be assigned to run in a different bus matrix layer from the master core image. Generally, the slave core image is allocated in an independent bank of SRAM and the master core one in Flash. The binary image of slave core may be included in the code of master core to be built together into an integrated image which is programmed on flash. When powered up, the master core image will be executed while the slave core is on hold with the factory

settings. After doing the system initialization including the mailbox, the master core will load the slave image from flash to its execution space, SRAM. Figure 2. on page 4 shows the process of loading slave core binary image.

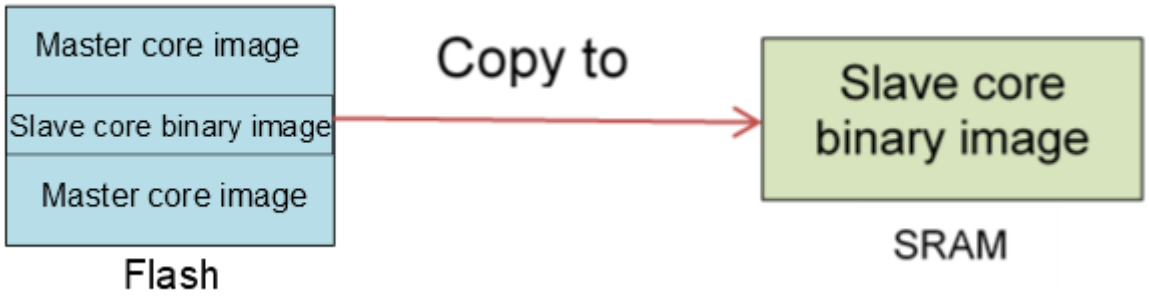

Figure 2. Load slave core binary image

2.2 Start-up slave core

The slave core binary image will be started up by the master core for running after it is loaded into a bank of SRAM. The clock to the slave core is required to be enabled and the slave core is required to be released to reset. The boot address of the slave core image needs to be set for the slave core to boot correctly. To do that, the LPC55xx/LPC55Sxx provides registers, as shown in Figure 3. on page 4.

CPU Control for multiple processors (CPUCTRL, offset = 0x800) bit description

Bit	Symbol	Access	Value	Description	Reset value
2:0	-	WO		Reserved. Read value is undefined, only zero should be written.	undefined
3	CPU1CLKEN	RW		CPU1 clock enable.	0x1
			0	The CPU1 clock is enabled.	
			1	The CPU1 clock is not enabled.	
4	-			Reserved. Read value is undefined, only zero should be written.	undefined
5	CPU1RSTEN	RW		CPU1 reset.	0x1
			0	The CPU1 is being reset.	
			1	The CPU1 is not being reset.	
15:6	-	WO	-	Reserved. Read value is undefined, only zero should be written.	-
31:16		RW		Must be written as 0xC0C4 for the write to have an effect.	-

Co-processor boot address (CPBOOT, offset = 0x804) bit description

Bit	Symbol	Value	Description	Reset value
31:0	CPBOOT		Co-processor boot address.	0x0

Figure 3. Registers for slave core startup to run

2.3 Dual core communication

After the slave core ia running as well, both cores need to communicate with each other for cooperation. As mentioned above, the LPC55xx/LPC55Sxx provides a simple hardware means called Inter-CPU Mailbox mechanism for communication. On the software, many methods can help to implement communications with both cores. The software methods could be simple or complex. The shared memory is the common base in the methods. Here, the hardware Inter-CPU Mailbox communication mechanism will be introduced.

2.3.1 Request to communicate

The mailbox module needs to be initialized before used. It enables the clock to the module and resets the module to work. To trigger a communication, one CPU will send interrupt requests to the other. The module provides registers to do it and meanwhile, it can allow to send a 32 bit non-zero data to each other. What the communication data represents could be status/event or user defined data. The use of the feature is entirely up to the user. Figure 4. on page 5 shows the Inter-CPU Mailbox communication.

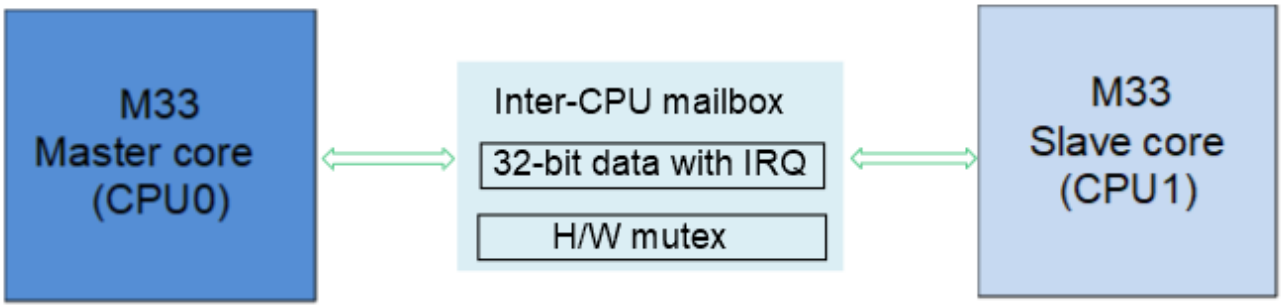

Figure 4. Inter-CPU mailbox communication

2.3.2 Communication synchronization

The mechanism also supports mutual exclusion control via a register called MUTEX for the communication synchronization. When read for any reason, the current value in the register will be returned and the bit will be cleared. The bit will be set again following any write. This can be used as a resource allocation handshake between two cores. Whenever a core wishes to access a shared resource (e.g. a shared variable in the driver example of mailbox_mutex), it reads the MUTEX register. If it sees a 1, it has control over the shared resource allocation. When it has made any needed changes, it writes to the register, causing the EX bit it to become set again, and making control of shared resource allocation available to another core. If a core reads a 0, it must wait for the bit to read as a 1 before accessing the shared resource allocation information. The mutual exclusion control flow is shown as Figure 5. on page 6.

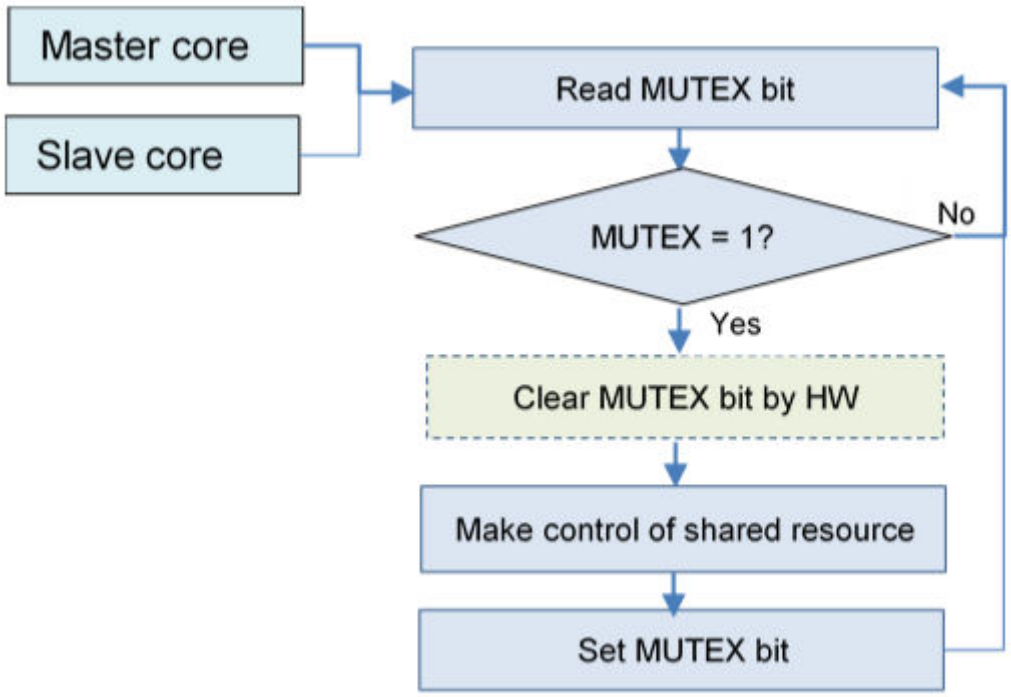

Figure 5. Mutual exclusion control flow

3 Example

The Inter-CPU Mailbox mechanism with mutual exclusion control is shown in the driver example of mailbox_mutex in LPC55xx SDK. It is a simple example

(The path is: boards\lpcxpresso55s69\driver_examples\mailbox\mutex).

The master core (core0) and slave core (core1) have its own project individually. In the example, the core 0 sends address of shared variable g_shared to core 1 by mailbox (see Figure 6. on page 6) and core1 gets it from mailbox in ISR (see Figure 6. on page 6). Both cores trying to get mutex in while loop. After updates shared variable, sets mutex to allow the other core to access shared variable.

The main loop process in master core (core0) and slave core (core1) are separately shown as Figure 8. on page 7 and Figure 9. on page 7.

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

static inline void MAILBOX_SetValue(MAILBOX_Type *base, mailbox_cpu_id_t cpu_id, uint32_t mboxData)
{
    base->MBOXIRQ[cpu_id].IRQ = mboxData;
}
```

Figure 6. Codes to send address of shared variable

```
void MAILBOX_IRQHandler()
{
    g_shared = (uint32_t *)MAILBOX_GetValue(MAILBOX, kMAILBOX_CM33_Core1);
    MAILBOX_ClearValueBits(MAILBOX, kMAILBOX_CM33_Core1, 0xffffffff);
}
```

Figure 7. Codes to get address of shared variable

```
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 8. Main loop process on slave core (core1)

```
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 9. Main loop process on master core (core0)

The result can be showed in a serial terminal window with the following settings: 115200 baud rate + 8 data bits + No parity + One stop bit + No flow control.

Prepare and run the demo as the below steps:

1. Build the slave core (core1) project to generate binary image.
2. Build the master core (core0) project with core1 binary image.
3. Open the serial terminal with the above settings.
4. Download the core0 image to the LPCXpresso55s69 board.
5. Press the reset button on the board.

The result will be printed in the terminal as shown in Figure 10. on page 8.

```
Mailbox mutex example
Copy CORE1 image to address: 0x20033000, size: 6348
Core0 has mailbox mutex, update shared variable to: 1
Core1 has mailbox mutex, update shared variable to: 2
Core0 has mailbox mutex, update shared variable to: 3
Core1 has mailbox mutex, update shared variable to: 4
Core0 has mailbox mutex, update shared variable to: 5
Core1 has mailbox mutex, update shared variable to: 6
Core0 has mailbox mutex, update shared variable to: 7
Core1 has mailbox mutex, update shared variable to: 8
Core0 has mailbox mutex, update shared variable to: 9
Core1 has mailbox mutex, update shared variable to: 10
```

Figure 10. Mailbox mutex example output on a serial terminal

4 Multicore SDK

The Inter-CPU mailbox with Mutex module is a simple means for dual core communication as mentioned the above. There are many ways to implement dual core communication in practice. Based on dual core and multi core architecture, NXP provides a more complete means of multicore development kit called multicore SDK with plenty of examples in LPC55xx/LPC55Sxx SDK package. The examples can be found in the path: boards\lpcxpresso55s69\multicore_examples. The multicore SDK architecture is shown as Figure 11. on page 9.

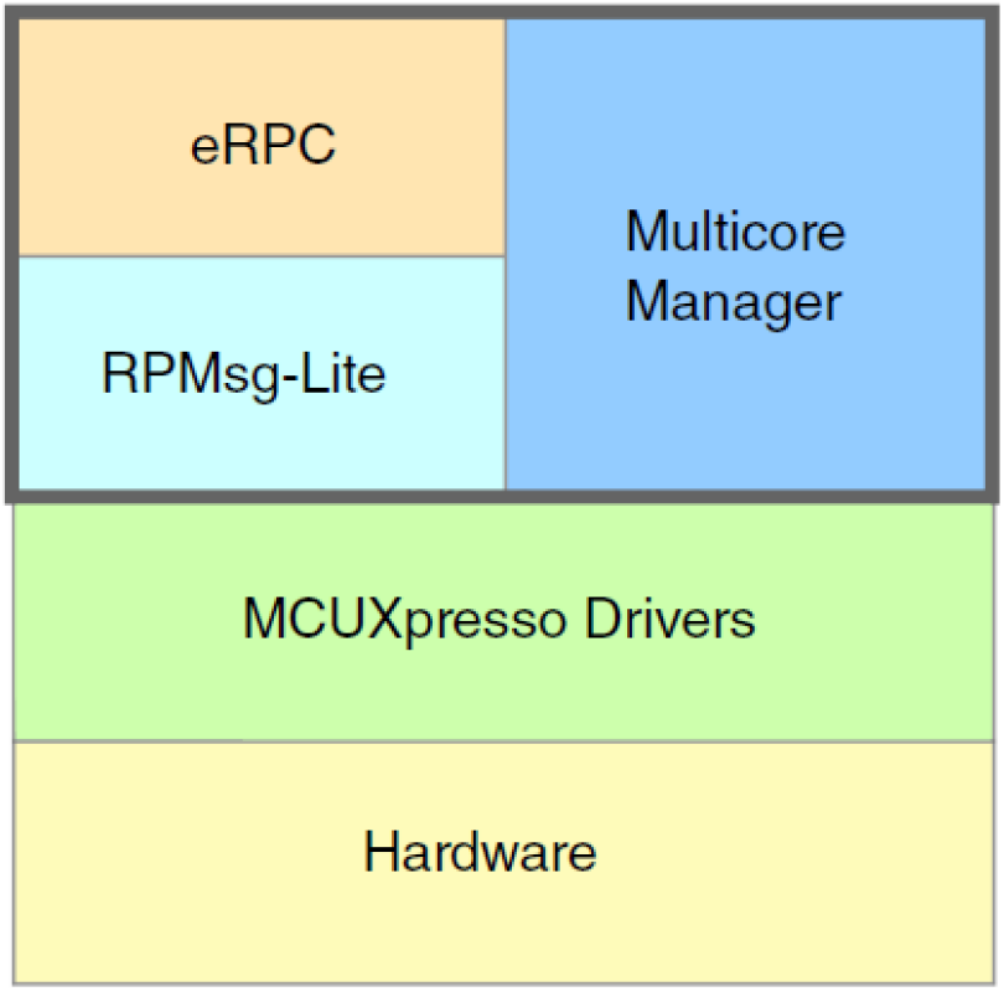

Figure 11. Multicore SDK architecture

- **eRPC**: Embedded Remote Procedure Call. The set of APIs provides to call remote services on another core.
- **RPMsg-Lite**: Remote Processor Messaging – Lite. The function library contains all processor communications.
- **Multicore Manager**: The APIs contain all CPU cores information and slave core startup.

With the Multicore SDK, the users can use directly the APIs to implement multicore communication for multicore application without focusing on the low-level drivers.

5 Conclusion

This application note introduces comprehensively the dual core mechanism supported on LPC55xx/LPC55Sxx and emphatically how to communicate with both cores based on the driver example of mailbox_mutex in LPC5500 SDK.

The dual core in the LPC55xx/LPC55Sxx is asymmetric architecture. One core is factory set to master core called CPU0/core0 and the other is slave called CPU1/core1. The master core can work while the slave is hold with the chip startup. The slave core needs to be started to work by the master. Before it, the slave core image typically should be loaded into the SRAM what is assigned as its execution space.

The LPC55xx/LPC55Sxx provides a simple means called Inter-CPU Mailbox module for dual core communication. A non-zero 32-bit data can be sent to each other with IRQ trigger via registers. The use of the feature is flexible and is entirely up to the user. In this module, a mutual exclusion control is supported for synchronization on a shared resource access.

There is a simple and intelligible example in LPC5500 SDK to demonstrate dual core usage with mailbox and mutex features. It is introduced briefly in this application note. It is very useful for a reader to understand the dual core usage.

In addition to this simple dual core communication mechanism, there is also multicore SDK packaged in LPC55xx/55Sxx SDK which has rich examples, can also be used for dual core communication development. For more information on this multicore SDK, please refer to <SDK installation directory>\middleware\multicore.

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: 23 January 2019
Document identifier: AN12335

arm