AN12351

LPC55(S)xx Dual-DMA Usage

Rev. 2 — 10/2020 Application Note

Contents

1 Introduction

The LPC55(S)xx is an Arm® Cortex®-M33 based micro-controller for embedded applications. These devices include:

- Up to 320 KB of on-chip SRAM
- Up to 640 KB on-chip flash
- High-speed and full-speed USB host and device interface with crystal-less operation for full-speed
- Five general-purpose timers
- One SCTimer/PWM
- One RTC/alarm timer
- One 24-bit Multi-Rate Timer (MRT)
- One Windowed Watchdog Timer (WWDT)
- Eight flexible serial communication peripherals (each of which can be a USART, SPI, I^2C, or I^2S interface)
- One 16-bit 1.0 Msp ADC
- Temperature sensor.

The Arm Cortex- M33 provides a security foundation, offering isolation to protect valuable IP and data with TrustZone® technology.

In addition to the information available in the LPC55(S)xx Data Sheet and User Manual. This application note provides the basic introduction of LPC55(S)xx Dual-DMA, the registers required to configure to use specified channel, the configuration steps based on SDK, hardware platform and program verification.

2 LPC55(S)xx DMA Introduction

2.1 DMA Introduction

DMA: Direct memory access (DMA) use provides high-speed data transfer between memory and memory or between peripherals and memory. DMA quickly moves the data is without any CPU action. This keeps CPU resources free for other operations.

NOTE

DMA only provides data transferring function and its transmit efficiency maybe not good as MCU Core.

DMA channel: Each DMA channel supports one DMA request line and one trigger input.

Example: Both USART0_RX and USART0_TX can be ‘request input’ and produce ‘request’ signal. The ‘request signal’ produced by a specific peripheral is connected to a DMA channel. User should follow ‘DMA requests Table’ to configure corresponding channel.

Possible DMA usage:

Table 1. DMA Transfer mode

DMA transfer mode	Source	Destination
Memory to Memory	AHB Memory Port	AHB Memory Port
Memory to Peripheral	AHB Memory Port	AHB Peripheral Port
Peripheral to Memory	AHB Peripheral Port	AHB Memory Port

2.2 Dual DMA block overview

- DMA controller: Two instances of SDMA IP that the user can decide which one is secure or not.
- DMA0: 22 channels, with multiplexers for 22 trigger sources. Each Flexcomm Interface provides a DMA RX and a DMA TX request to the DMA controller. The ADC is connected to 2 different DMA request channels. SCT and selected timers and pin interrupts are also used as DMA triggers. In addition, four DMA triggers are selected from among all of the DMA channel output triggers. SHA-2 and AES also provides DMA channel and trigger interface.
- DMA1: 10 channels with multiplexers for 15 trigger sources.
- Priority is user selectable for each channel (up to eight priority levels).
- Continuous priority arbitration.
- Supports single transfers up to 1,024 words.
- Address increment options allow packing and/or unpacking data.

2.3 Basic configuration of DMA block

This section descibes the DMA block diagram.

2.3.1 DMA block diagram description

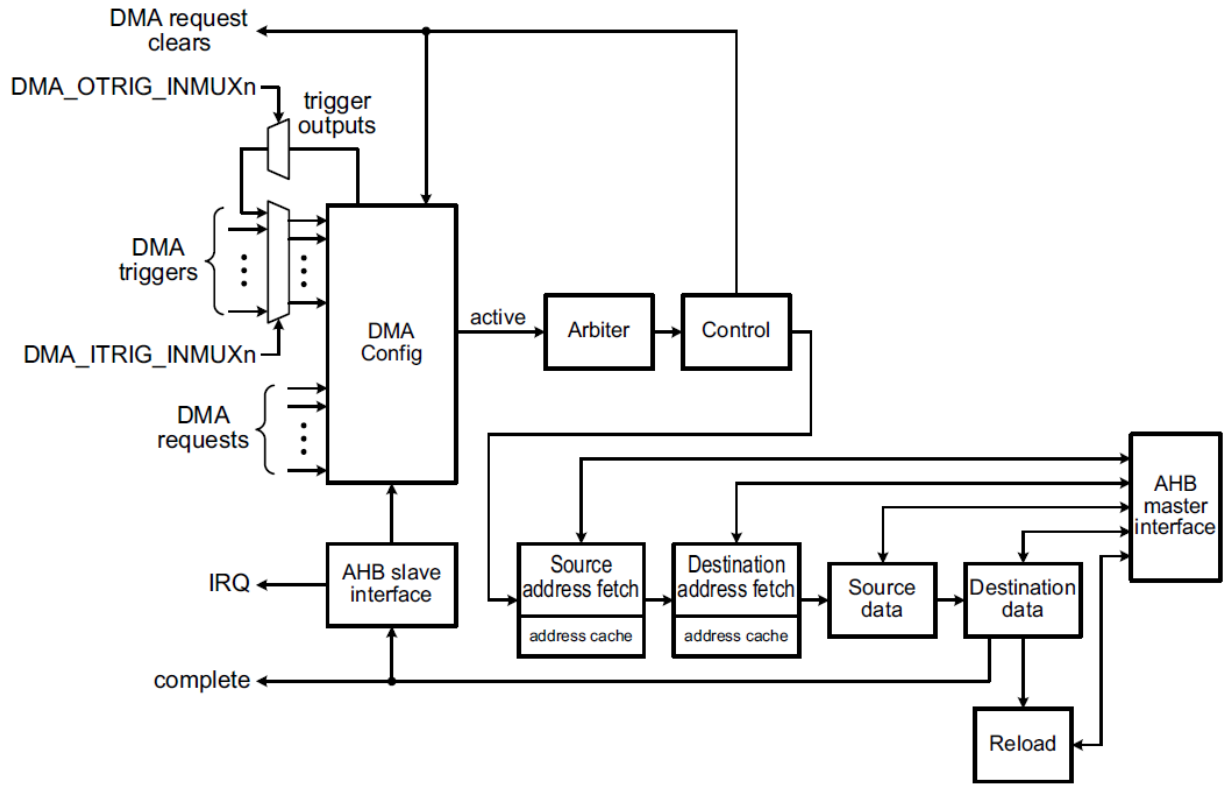

Figure 1. DMA block diagram

NOTE

DMA requests are directly connected to the peripherals. Each channel supports one DMA request line and one trigger input. Some DMA requests allow a selection of requests sources. DMA triggers are selected from many possible input sources.

2.3.2 DMA requests and trigger multiplexers

DMA requests are intended to pace transfer to match what the peripheral (including its FIFO if it has one) can do, then DMA triggers start the transfer, user can choose software trigger or hardware trigger depend on the design requirements.

The requests and trigger muxes for DMA0 and DMA1 are shown in the DMA0 requests and trigger multiplexers table.

Table 2. DMA0 requests and trigger multiplexers

DMA channel	Request input	DMA trigger MUX
0	Hash-Crypt DMA request	DMA0_ ITRIG_ INMUX0
1	Spare channel, no request connected	DMA0_ ITRIG_ INMUX1
2	High Speed SPI (Flexcomm 8) RX	DMA0_ ITRIG_ INMUX2
3	High Speed SPI (Flexcomm 8) TX	DMA0_ ITRIG_ INMUX3
4	Flexcomm Interface 0 RX / I2C Slave	DMA0_ ITRIG_ INMUX4

Table continues on the next page...

Table 2. DMA0 requests and trigger multiplexers (continued)

DMA channel	Request input	DMA trigger MUX
5	Flexcomm Interface 0 TX / I2C Master	DMA0_ ITRIG_ INMUX5
6	Flexcomm Interface 1 RX / I2C Slave	DMA0_ ITRIG_ INMUX6
7	Flexcomm Interface 1 TX / I2C Master	DMA0_ ITRIG_ INMUX7
8	Flexcomm Interface 2 RX / I2C Slave	DMA0_ ITRIG_ INMUX8
9	Flexcomm Interface 2 RTX / I2C Master	DMA0_ ITRIG_ INMUX9
10	Flexcomm Interface 3 RX / I2C Slave	DMA0_ ITRIG_ INMUX10
11	Flexcomm Interface 3 TX / I2C Master	DMA0_ ITRIG_ INMUX11
12	Flexcomm Interface 4 RX / I2C Slave	DMA0_ ITRIG_ INMUX12
13	Flexcomm Interface 4 TX / I2C Master	DMA0_ ITRIG_ INMUX13
14	Flexcomm Interface 5 RX / I2C Slave	DMA0_ ITRIG_ INMUX14
15	Flexcomm Interface 5 TX / I2C Master	DMA0_ ITRIG_ INMUX15
16	Flexcomm Interface 6 RX / I2C Slave	DMA0_ ITRIG_ INMUX16
17	Flexcomm Interface 6 TX / I2C Master	DMA0_ ITRIG_ INMUX17
18	Flexcomm Interface 7 RX / I2C Slave	DMA0_ ITRIG_ INMUX18
19	Flexcomm Interface 7 TX / I2C Master	DMA0_ ITRIG_ INMUX19
20	Spare channel, no request connected	DMA0_ ITRIG_ INMUX20
21	ADC0 FIFO 0	DMA0_ ITRIG_ INMUX21
22	ADC0 FIFO 1	DMA0_ ITRIG_ INMUX22

Table 3. DMA1 requests and trigger multiplexers

DMA channel	Request input	DMA trigger mux
0	Hash-Crypt DMA request	DMA1_ ITRIG_ INMUX0
1	Spare channel, no request connected	DMA1_ ITRIG_ INMUX1
2	High Speed SPI (Flexcomm 8) RX	DMA1_ ITRIG_ INMUX2
3	High Speed SPI (Flexcomm 8) TX	DMA1_ ITRIG_ INMUX3
4	Flexcomm Interface 0 RX / I2C Slave	DMA1_ ITRIG_ INMUX4
5	Flexcomm Interface 0 TX / I2C Master	DMA1_ ITRIG_ INMUX5
6	Flexcomm Interface 1 RX / I2C Slave	DMA1_ ITRIG_ INMUX6
7	Flexcomm Interface 1 TX / I2C Master	DMA1_ ITRIG_ INMUX7
8	Flexcomm Interface 2 RX / I2C Slave	DMA1_ ITRIG_ INMUX8
9	Flexcomm Interface 2 RTX / I2C Master	DMA1_ ITRIG_ INMUX9

Table 4. DMA trigger sources

DMA trigger	DMA0 trigger input	DMA1 trigger input
0	Pin interrupt 0	Pin interrupt 0
1	Pin interrupt 1	Pin interrupt 1
2	Pin interrupt 2	Pin interrupt 2
3	Pin interrupt 3	Pin interrupt 3
4	Timer CTIMER0 Match 0	Timer CTIMER0 Match 0
5	Timer CTIMER0 Match 1	Timer CTIMER0 Match 1
6	Timer CTIMER1 Match 0	Timer CTIMER2 Match 0
7	Timer CTIMER1 Match 1	Timer CTIMER4 Match 0
8	Timer CTIMER2 Match 0	OMA output trigger 0
9	Timer CTIMER2 Match 1	OMA output trigger 1
10	Timer CTIMER3 Match 0	OMA output trigger 2
11	Timer CTIMER3 Match 1	OMA output trigger 3
12	Timer CTIMER4 Match 0	SCTO OMA request 0
13	Timer CTIMER4 Match 1	SCTO OMA request 1
14	Comparator 0 output	Hash-Crypt output DMA
15	DMA output trigger 0	N/A
16	DMA output trigger 1	N/A
17	DMA output trigger 2	N/A
18	DMA output trigger 3	N/A
19	SCT0 DMA request 0	N/A
20	SCT0 DMA request 1	N/A
21	Hash-Cryptoutput DMA	N/A

3 DMA configuration API

This section lists the DMA configuration APIs and provides Data API examples.

3.1 DMA configuration API lists

DMA registers are grouped into DMA control, interrupt, and status registers and DMA channel registers. Each DMA transfer channel is controlled by a set of registers including CFG[0:29], CTRLSTAT[0:29] and XFERCFG[0:29]. Two DMA controllers are present: DMA0 and DMA1 (secure).

This chapter describes how to configure related registers using API without changing the SDK driver and let users use DMA function easily, the main function prototypes are shown in the DMA API calls table.

Table 5. DMA API calls

Function prototype	API description
void DMA_Init *(DMA_Type *base)*	This function enable the DMA clock, set descriptor table and enable DMA peripheral. * param base DMA peripheral base address.
void DMA_CreateHandle *(dma_handle_t *handle,* *DMA_Type *base,* *uint32_t channel)*	This function is called if using transaction API for DMA. This function initializes the internal state of DMA handle. * param handle DMA handle pointer. The DMA handle stores callback function and parameters. * param base DMA peripheral base address. * param channel DMA channel number.
Void DMA_EnableChannel *(DMA_Type *base,* *uint32_t channel)*	This function Enable DMA channel. * param base DMA peripheral base address. * param channel DMA channel number.
void DMA_SetCallback *(dma_handle_t *handle, dma_callback callback,* *void *userData)*	This callback is called in DMA IRQ handler. Use the callback to do something after the current major loop transfer completes. * param handle DMA handle pointer. * param callback DMA callback function pointer. * param userData Parameter for callback function.
void DMA_PrepareTransfer *(dma_transfer_config_t *config, void *srcAddr, void *dstAddr,* *uint32_t byteWidth, uint32_t transferBytes,* *dma_transfer_type_t type, void *nextDesc)*	This function prepares the transfer configuration structure according to the user input. * param config The user configuration structure of type dma_transfer_t. * param srcAddr DMA transfer source address. * param dstAddr DMA transfer destination address. * param byteWidth DMA transfer destination address width(bytes). * param transferBytes DMA transfer bytes to be transferred. * param type DMA transfer type. * param nextDesc Chain custom descriptor to transfer.
status_t DMA_SubmitTransfer *(dma_handle_t *handle, dma_transfer_config_t *config)*	This function submits the DMA transfer request according to the transfer configuration structure. * param handle DMA handle pointer. * param config Pointer to DMA transfer configuration structure.
void DMA_StartTransfer *(dma_handle_t *handle)*	This function enables the channel request. * param handle DMA handle pointer.

3.2 Data API example

This section lists Data API example for:

- Memory to memory
- Memory to peripheral
- Peripheral to memory

3.2.1 Memory to memory

1. DMA_Init:

   The initial step includes enable the DMA clock, set descriptor table, and enable DMA peripheral.

```
/** Peripheral DMA0 base address */
    #define DMA0_BASE_NS (0x40082000u)
    /** Peripheral DMA0 base pointer */
    #define DMA0_NS                  ((DMA_Type *)DMA0_BASE_NS)
    volatile DMA_Type *DMA0_NS_Base = (DMA_Type *)DMA0_NS;
    DMA_Init(DMA0);
```

2. DMA_CreateHandle

```
DMA_CreateHandle(&g_DMA_Handle, DMA0, 0);
```

> **NOTE**
>
> User can use DMA1. For example, ‘DMA_CreateHandle(&g_DMA_Handle, DMA1, 0)’ means DMA1_Channel0 is selected as channel transferring data.

3. DMA_EnableChannel:

   This function enables the DMA channel. To enable the chosen channel, DMA0_Channel0 selected at Step 2 is used to call the API.

```
DMA_EnableChannel(DMA0, 0);
```

> **NOTE**
>
> It is worth noting that LPC55(S)xx has two DMA (DMA0 & DMA1), the ‘ struct COMMON[x] ’ should be ‘ COMMON[0] ’ if DMA0 is current chosen DMA. If DMA1 is chosen DMA, the struct is ‘ COMMON[1] ’. The Register Layout Typedef ‘ DMA_Type ’ has full definition of related registers.

```
typedef struct {
  __IO uint32_t CTRL;                          /**< DMA control., offset: 0x0 */
  __I  uint32_t INTSTAT;                       /**< Interrupt status., offset: 0x4 */
  __IO uint32_t SRAMBASE;                      /**< SRAM address of the channel configuration table., offset: 0x8 */
       uint8_t RESERVED_0[20];
  struct {                                     /* offset: 0x20, array step: 0x5C */
    __IO uint32_t ENABLESET;                     /**< Channel Enable read and Set for all DMA channels., array offset: 0x20, array step: 0x5C */
         uint8_t RESERVED_0[4];
    __O  uint32_t ENABLECLR;                     /**< Channel Enable Clear for all DMA channels., array offset: 0x28, array step: 0x5C */
         uint8_t RESERVED_1[4];
    __I  uint32_t ACTIVE;                        /**< Channel Active status for all DMA channels., array offset: 0x30, array step: 0x5C */
         uint8_t RESERVED_2[4];
    __I  uint32_t BUSY;                          /**< Channel Busy status for all DMA channels., array offset: 0x38, array step: 0x5C */
         uint8_t RESERVED_3[4];
    __IO uint32_t ERRINT;                        /**< Error Interrupt status for all DMA channels., array offset: 0x40, array step: 0x5C */
         uint8_t RESERVED_4[4];
    __IO uint32_t INTENSET;                      /**< Interrupt Enable read and Set for all DMA channels., array offset: 0x48, array step: 0x5C */
         uint8_t RESERVED_5[4];
    __O  uint32_t INTENCLR;                      /**< Interrupt Enable Clear for all DMA channels., array offset: 0x50, array step: 0x5C */
         uint8_t RESERVED_6[4];
    __IO uint32_t INTA;                          /**< Interrupt A status for all DMA channels., array offset: 0x58, array step: 0x5C */
         uint8_t RESERVED_7[4];
    __IO uint32_t INTB;                          /**< Interrupt B status for all DMA channels., array offset: 0x60, array step: 0x5C */
         uint8_t RESERVED_8[4];
    __O  uint32_t SETVALID;                      /**< Set ValidPending control bits for all DMA channels., array offset: 0x68, array step: 0x5C */
         uint8_t RESERVED_9[4];
    __O  uint32_t SETTRIG;                       /**< Set Trigger control bits for all DMA channels., array offset: 0x70, array step: 0x5C */
         uint8_t RESERVED_10[4];
    __O  uint32_t ABORT;                         /**< Channel Abort control for all DMA channels., array offset: 0x78, array step: 0x5C */
  } COMMON[1];
       uint8_t RESERVED_1[900];
  struct {                                     /* offset: 0x400, array step: 0x10 */
    __IO uint32_t CFG;                           /**< Configuration register for DMA channel ., array offset: 0x400, array step: 0x10 */
    __I  uint32_t CTLSTAT;                       /**< Control and status register for DMA channel ., array offset: 0x404, array step: 0x10 */
    __IO uint32_t XFERCFG;                       /**< Transfer configuration register for DMA channel ., array offset: 0x408, array step: 0x10 */
         uint8_t RESERVED_0[4];
  } CHANNEL[30];
} DMA_Type;
```

Figure 2. DMA_Type introduction

4. DMA_EnableChannel:

   This callback is called in DMA IRQ handler. User can get the interrupt A or B flags and monitor whether it is set by the software using the following API.

```
DMA_SetCallback(&g_DMA_Handle, DMA_Callback, NULL);
```

5. DMA_PrepareTransfer:

   This function prepares the transfer configuration structure according to the user input. The following code represents that the user can configure DMA transfer source address, destination address, DMA transfer type, and custom descriptor according to data transferring requirements.

```
DMA_PrepareTransfer(&transferConfig,srcAddr,destAddr,sizeof(srcAddr[0]),sizeof(srcAddr),
kDMA_MemoryToMemory,NULL);
```

NOTE

The sixth parameter determines whether the source address and destination address should increment or not. For example, both source address and destination address should increment to copy data between the memory in order.

```
config->xfercfg.srcInc = 1;
config->xfercfg.dstInc = 1;
config->isPeriph = false;
```

6. DMA_SubmitTransfer:

   This function submits the DMA transfer request according to the transfer configuration structure. The following code represents that user can finish DMA transfer configuration by configure structure 'dma_xfercfg_t' that include reload channel configuration, perform software trigger and other information.

```
typedef struct _dma_xfercfg
{
    bool valid;             /*!< Descriptor is ready to transfer */
    bool reload;            /*!< Reload channel configuration register after
                                 current descriptor is exhausted */
    bool swtrig;            /*!< Perform software trigger. Transfer if fired
                                 when 'valid' is set */
    bool clrtrig;           /*!< Clear trigger */
    bool intA;              /*!< Raises IRQ when transfer is done and set IRQA status register flag */
    bool intB;              /*!< Raises IRQ when transfer is done and set IRQB status register flag */
    uint8_t byteWidth;      /*!< Byte width of data to transfer */
    uint8_t srcInc;         /*!< Increment source address by 'srcInc' x 'byteWidth' */
    uint8_t dstInc;         /*!< Increment destination address by 'dstInc' x 'byteWidth' */
    uint16_t transferCount; /*!< Number of transfers */
} dma_xfercfg_t;
```

Figure 3. DMA_xfercfg_t instruction

Then user can create specific DMA descriptor to be used in a chain in transfer by using following API.

```
DMA_StartTransfer(&g_DMA_Handle);
```

NOTE

The default value of CFG register bit 1 ‘HWTRIGEN’ is ‘0’. This means Hardware trigger is unused. To trigger a DMA transfer like Pin interrupt 0, Timer CTIMER0 Match 0 and others using hardware, follow these steps and finish the extra configuration.

a. CFG register bit 1 ‘HWTRIGEN’ should be set by using the following pseudo code.

```
Base->CHANNEL[x]. CFG  |= 0x2U;
```

b. XFERCFGn register bit 2 software trigger should be disabled and could wait for hardware trigger signal by using the following pseudo code.

```
Base->CHANNEL[x]. XFERCFG |=~ 0x4U
```

c. User should choose ‘DMA trigger sources’ by configuring register ‘DMA0_ITRIG_INMUX[0:22]’ or ‘DMA1_ITRIG_INMUX[0:9]’. The following pseudo code means we can choose hardware trigger 2 ‘Pin interrupt 2’ for DMA0_Channlx’s request input.

```
Base->DMA0_ITRIG_INMUX[x] = 0x2;
```

With the above configuration of seven main steps, DMA is ready to transfer data. Now user can wait for DMA transfer finish by monitoring the value of ‘g_Transfer_Done’ which is changed in ‘DMA_Callback’ function.

```
void DMA_Callback(dma_handle_t *handle, void *param, bool transferDone, uint32_t tcds)
{
          if (transferDone){
             g_Transfer_Done = true;
          }
}
while (g_Transfer_Done != true)
          {}
```

3.2.2 Memory to peripheral

User can transfer data from memory to peripheral. For example, transfer data from RAM to USART0_FIFOWR. The main steps are similar to instructions in the section Memory to memory except Step 5.

```
/** Peripheral USART0 base address  */
#define USART0_BASE_NS (0x40086000u)
/** Peripheral USART0 base pointer */
#define USART0_NS  ((USART_Type *)USART0_BASE_NS)
volatile USART_Type *USART0_NS_Base = (USART_Type *)USART0_NS;
DMA_PrepareTransfer(&transferConfig,srcAddr,
                    USART0_NS_Base ->FIFOWR, sizeof(srcAddr[0]),sizeof(srcAddr),
                    kDMA_MemoryToPeripheral,NULL);
```

NOTE

Considering the Peripheral(USART0_FIFOWR) has its fixed base address: 0x4008 6000h+ 0xE20, so the address does not increment. If used as 'Destination data address', while use Memory(RAM) as 'Source data address' and its address should increment.

```
/* Peripheral register - destination doesn't increment */
    config->xfercfg.srcInc = 1;
    config->xfercfg.dstInc = 0;
    config->isPeriph = true;
```

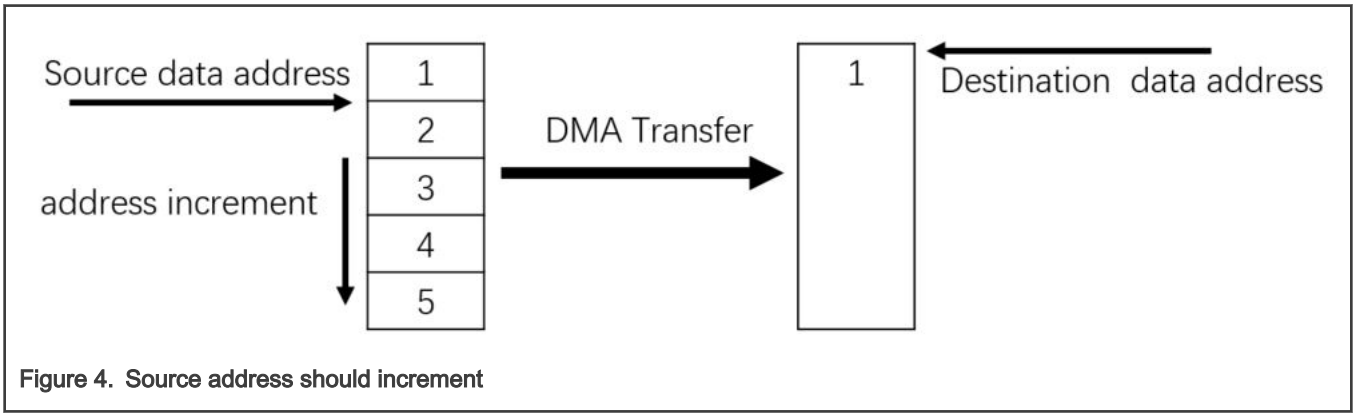

Figure 4. Source address should increment

3.2.3 Peripheral to memory

User can transfer data from peripheral to memory. For example, if the sensor and microcontroller unit are communicating through USART0 then user can store the data collected by the sensor into Memory by transferring data from USART0_FIFORD to RAM. The main steps are similar to instructions in the section Memory to memory except Step 5.

1. Register FIFORD bit 0-8 can be used to store received data from FIFO. The number of bits used depends on the DATALEN and PARITYSEL settings.
2. For example, user can transfer a character in USART0->FIFORD to RAM by using following API to configure 'DMA_PrepareTransfer' function.

```
DMA_PrepareTransfer(&transferConfig, (void *)(&USART0_NS_Base->FIFORD),desAddr, 4, 4,
kDMA_PeripheralToMemory, NULL);
```

NOTE

Considering the Peripheral(USART0_NS->FIFORD) has its fixed base address: 0x4008 6000h+ 0xE30, so its address does not increment. If user uses it as 'Source data address', while use Memory (RAM) as 'Destination data address the address should increment.

```
/*Peripheral register - source doesn't increment */
    config->xfercfg.srcInc = 0;
    config->xfercfg.dstInc = 1;
    config->isPeriph = true;
```

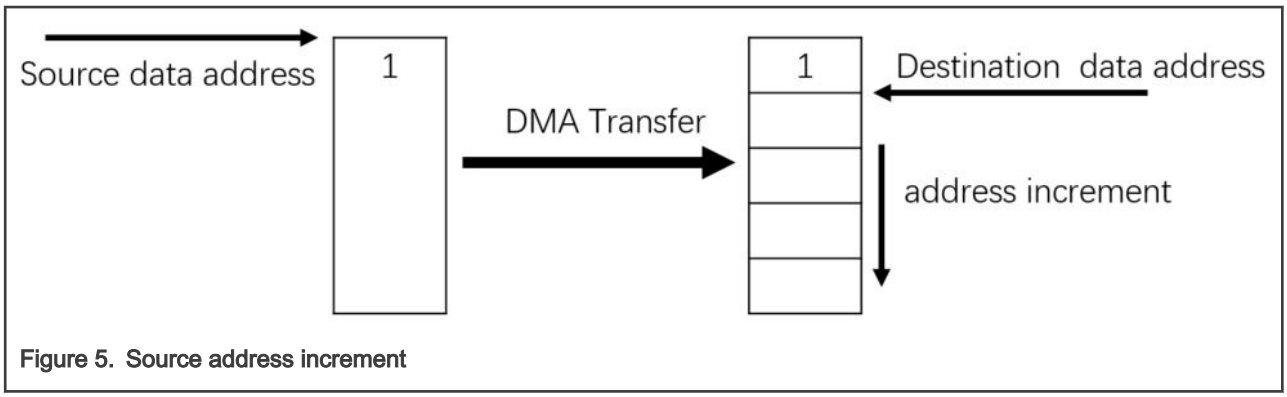

Figure 5. Source address increment

4 DMA verification

This application note provides examples to finish DMA transfer work including Memory to memory, Memory to peripheral, and Peripheral to memory mode. This application note takes LPC55S6x as an example to show the function of Dual DMA.

The following demos are based on the IAR, MCUXpresso IDE platform of NXP is also support. User can download related project on the LPC55S6x to verify Dual-DMA function. This demo uses the USB Virtual COM (VCOM) to display log information.

4.1 Hardware design

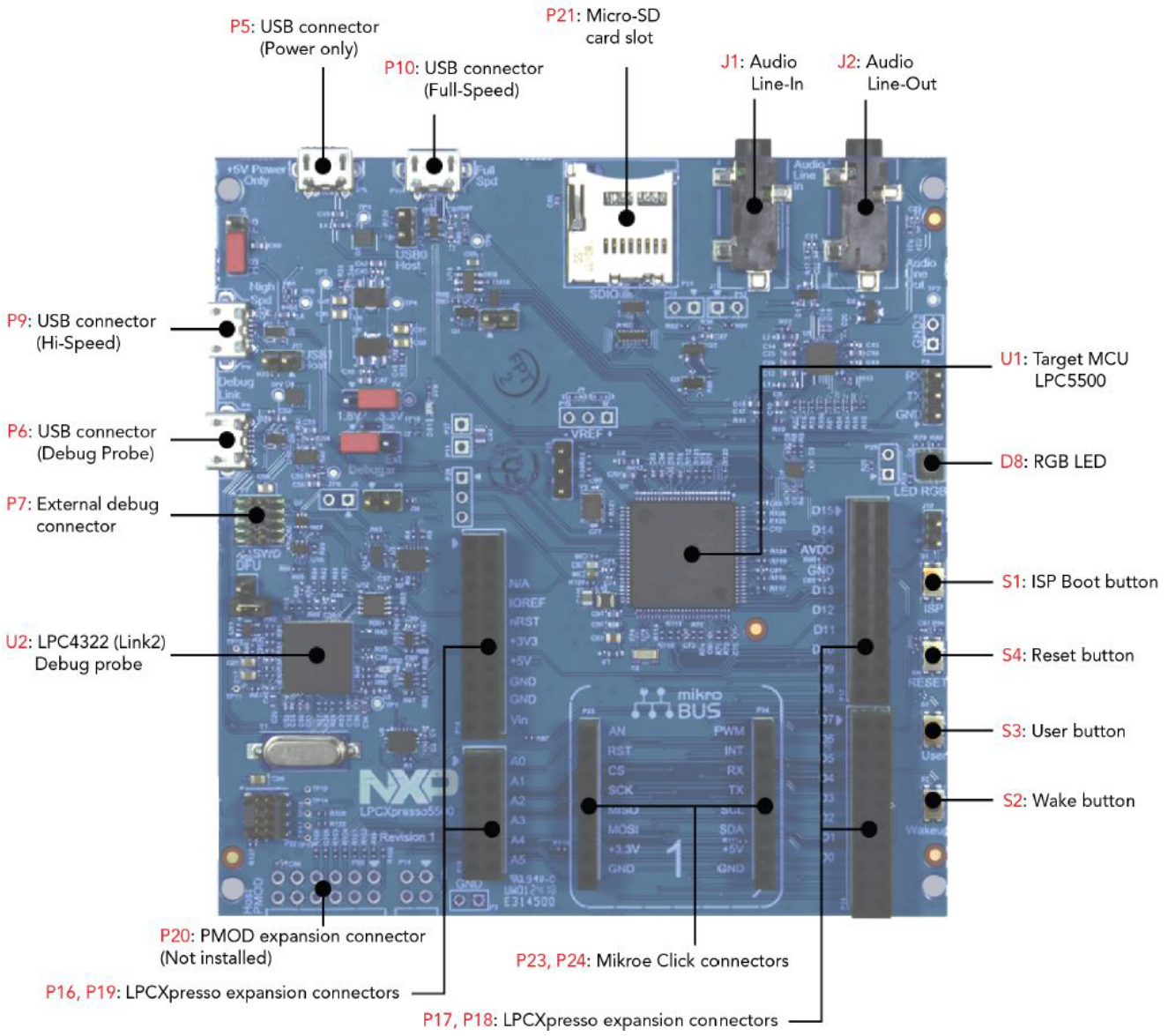

Figure 6. LPCXpresso55S69 EVK

To download and debug the project and view serial port data, user can connect USB port P6 with PC by using USB interface.

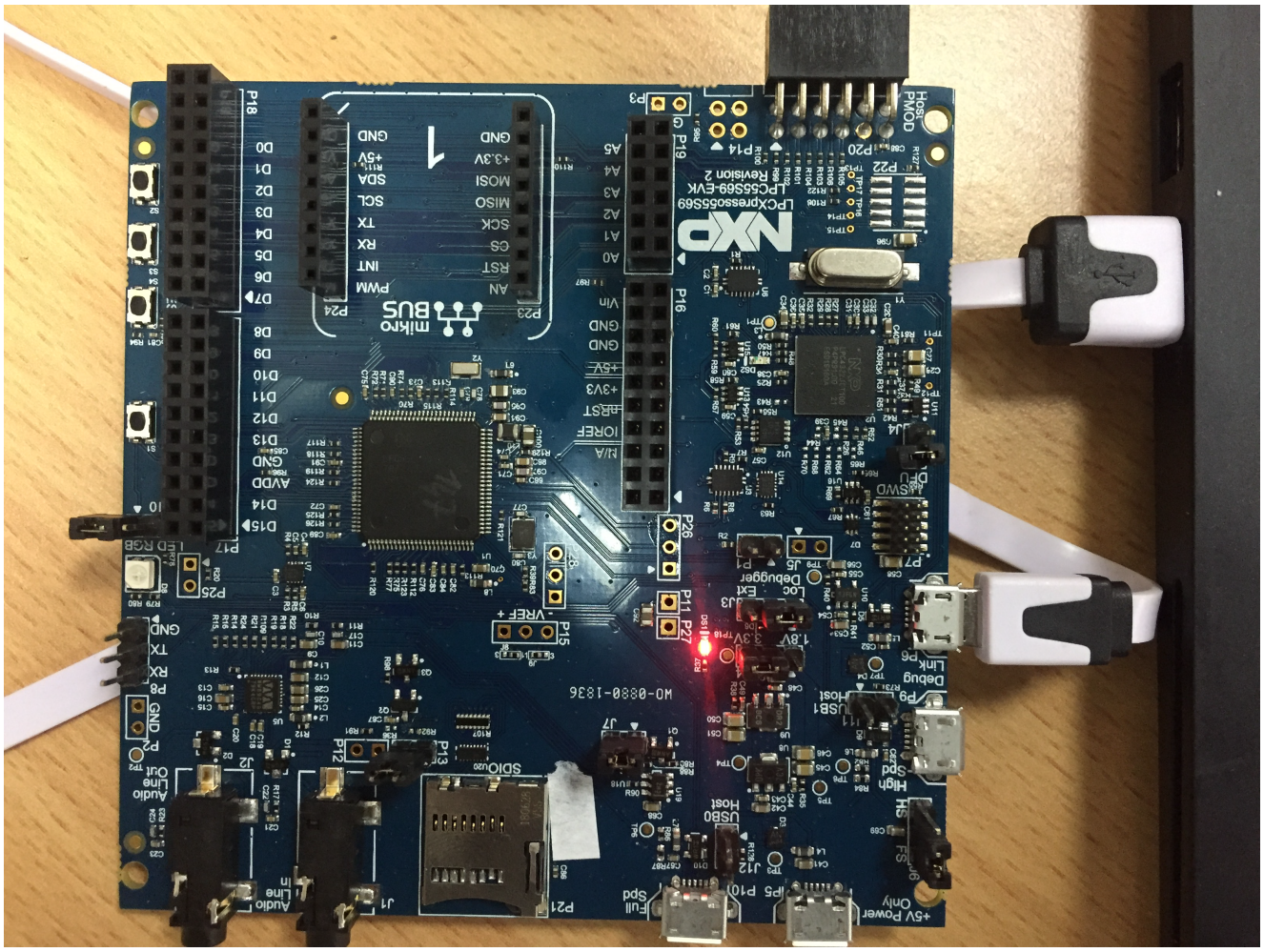

Figure 7. Connect LPCXpresso55S69 with PC

4.2 Software set-up

Two IDEs were used to verify the sample DMA projects:

- IAR Embedded Workbench v8.32.1.
- MCUXpresso IDE v10.3.0 (can download from www.nxp.com).

To compile and load the software example using IAR IDE, follow these steps.

- Connect USB port P6 of an LPCXpresso55S69 board to the PC (J3 set to Loc, P4 set to 3.3 V, J6 set to FS)
- Unzip the project folder "…\LPC55S6x Dual-DMA.zip" for testing.
- Compile all projects by clicking 'Compile (Ctrl+F7)' or 'Make (F7)' from the quickstart menu.

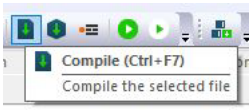

Figure 8. Compile the project in IAR

- Starting the debugger by clicking 'Download and Debug (Ctrl+D)'. This step compiles and flashes the code and starts the debugger.

Figure 9. Download and Debug the project in IAR

- Terminal software, like tera-term.

4.3 Program verification

To run the code after downloading the DMA project, press the 'RESET(S4)' button. Choose the DMA transfer mode according to the log information from the USB Virtual COM (VCOM) port.

```
Select a DMA transfer mode
        1. Memory to Memory(Software Trigger)
        2. Memory to Peripheral(Hardware Trigger)
        3. Memory to Peripheral(Software Trigger)
        4. Peripheral to Memory(Software Trigger)
```

Figure 10. DMA transfer mode selection menu

- Input '1' to trigger DMA memory to memory transfer by software.

```
1
Entering 1.M2M (Note: Do not need other operation) ...
desAddr[] Buffer:
0       0       0       0
desAddr[] Buffer:
1       2       3       4
DMA Memory to Memory(Software Trigger) example finish, pressing 'S4_RESET' to verify other DMA transfer.
```

Figure 11. Memory to memory software trigger transfer success result

- Input '2' to trigger DMA memory to peripheral transfer by hardware.

```
2
Entering 2.M2P (Note: User need press Button_'S3_USER' to Trigger DMA) ...
1234
DMA Memory to Peripheral(Hardware Trigger) example finish, pressing 'S4_RESET' to verify other DMA transfer.
```

Figure 12. Memory to peripheral hardware trigger transfer success result

- Input '3' to trigger DMA memory to peripheral transfer by software.

```
3
Entering 3.M2P (Note: Do not need other operation) ...
1234
DMA Memory to Peripheral(Software Trigger) example finish, pressing 'S4_RESET' to verify other DMA transfer.
```

Figure 13. Memory to peripheral software trigger transfer success result

- Input '4' to trigger DMA peripheral to memory by software.

```
4
Entering 4.P2M (Note: User need Input a character) ...
Input a character for P2M and then read this character from (USART0->FIFOWR(RO)):4
 desAddr[] Buffer: = 4
DMA Peripheral to Memory(Software Trigger) example finish, pressing 'S4_RESET' to verify other DMA transfer.
```

Figure 14. Peripheral to memory transfer software trigger success result

5 Conclusion

The software example provided with this application note shows the basic configuration for LPC55S6x DMA block. Both DMA0 and DMA1 channels can be selected to transfer data according to the actual needs.

6 Revision history

Table 6 summarizes the changes since the initial release.

Table 6. Revision history

Rev.	Date	Substantive changes
0	14 February 2019	Initial release
1	26 February 2020	Figure 1 updated
2	27 October 2020	Updated LPC55(S)xx for LPC55S6x

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: 10/2020

Document identifier: AN12351

arm