AN12351

LPC55(S)xx 双 DMA 用法

版本 2 — 2020 年 10 月 27 日

应用笔记

目录

1 介绍

LPC55(S)xx 是用于嵌入式开发的基于 Arm®Cortex®-M33 的微控制器。这些控制器包括：

- 多达 320 KB 的片上 SRAM
- 多达 640 KB 的片上 flash
- 高速和全速 USB 主机及设备接口，无晶体可实现全速 USB 操作
- 五个通用定时器
- 一个 SCTimer/PWM
- 一个 RTC/alarm timer
- 一个 24 位多速率计时器（MRT）
- 一个窗口化看门狗定时器（WWDT）
- 八个灵活的串行通信外设（每个外围设备可以是 USART，SPI，I2C 或 I2S 接口）
- 一个十六位的 1.0 Msp 数模转换器
- 温度传感器

Arm Cortex-M33 提供了安全基础，使用 TrustZone®技术提供隔离保护珍贵的 IP 和数据。除了 LPC55(S)xx 数据手册和用户手册中的可用信息外，本应用笔记提供了 LPC55(S)xx 双 DMA 的基本介绍，使用指定通道需要配置的寄存器，基于 SDK 的配置步骤，硬件平台和程序验证。

2 LPC55(S)xx DMA 介绍

2.1 DMA 介绍

DMA: 直接内存访问（DMA）在内存与内存之间或外设与内存之间提供了高速数据传输。 DMA 快速搬运数据而无需任何 CPU 操作。 这使 CPU 资源可用于其他操作。

注意

DMA 仅提供数据传输功能，而且其传输效率可能不如 MCU 核。

DMA 通道：每个 DMA 通道支持一个 DMA 请求线和一个触发输入。

示例：USART0_RX 和 USART0_TX 都可以是“请求输入”并产生“请求”信号。 特定外设产生的“请求信号”连接到 DMA 通道。 用户应遵循“ DMA 请求表”来配置相应的通道。

可能的 DMA 使用情况：

表 1. DMA 传输模式

DMA 传输模式	源头	目的地
内存到内存	AHB 内存端口	AHB 内存端口
内存到外设	AHB 内存端口	AHB 外设端口
外设到内存	AHB 外设端口	AHB 内存端口

2.2 双 DMA 模块概述

- DMA 控制器：SDMA IP 的两个实例，用户可以将一个设置在安全模式下运行，另一个设置在非安全模式下运行。
- DMA0：22 个通道，带有用于 22 个触发源的多路复用器。每个 Flexcomm 接口都向 DMA 控制器提供一个 DMA RX 和 DMA TX 请求。数模转换器连接到 2 个不同的 DMA 请求通道。SCT 和选定的定时器以及引脚中断也可用作 DMA 的触发。此外，从所有 DMA 通道输出触发器中选择了四个 DMA 触发器。SHA-2 和 AES 还提供 DMA 通道和触发接口。
- DMA1: 10 个带多路复用器的通道，用于 15 个触发源。
- 用户可以为每个通道选择优先级（最多八个优先级）。
- 连续优先权仲裁。
- 单次传输最大支持 1024 个字。
- 地址增量选项允许打包和/或拆包数据。

2.3 DMA 模块的基本配置

本节介绍 DMA 模块框图。

2.3.1 DMA 模块框图描述

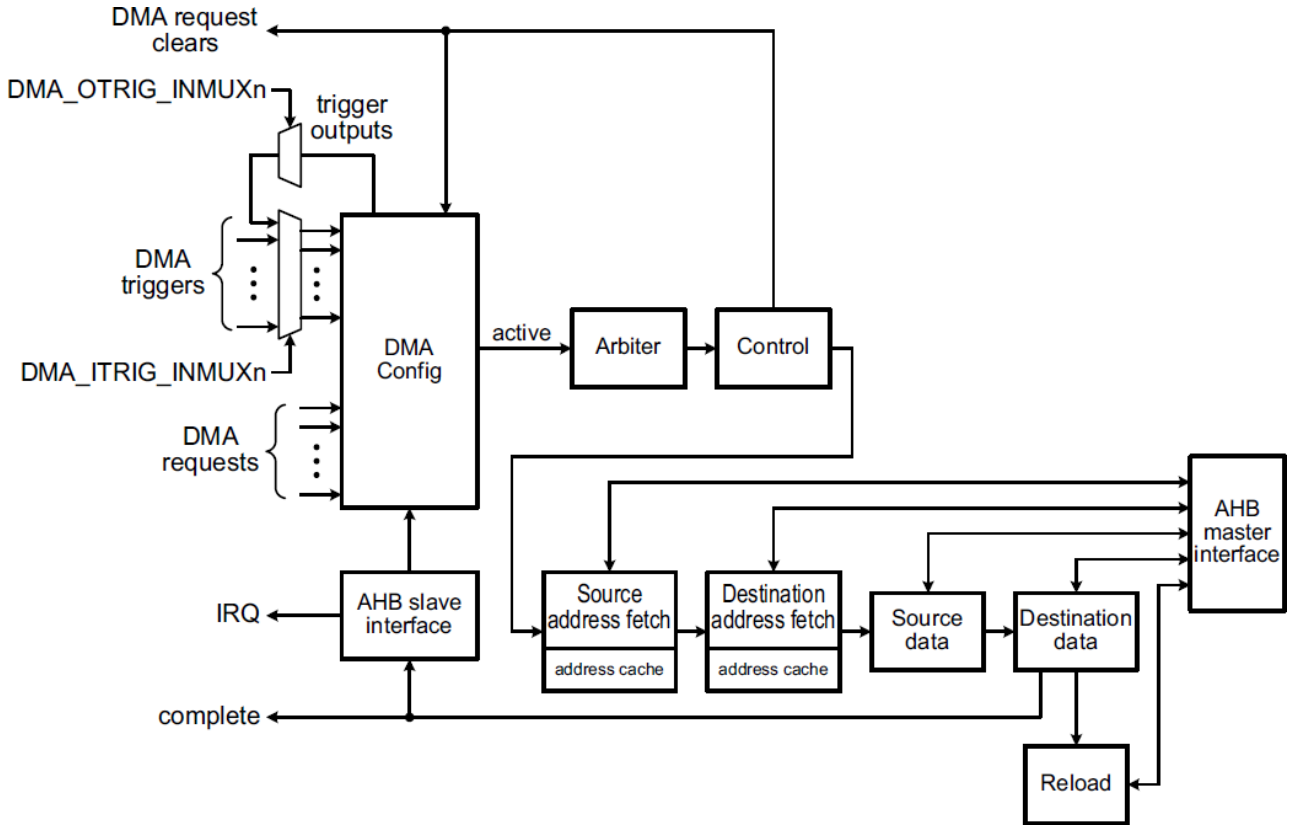

图 1. DMA 模块框图

注意

DMA 请求直接连接到外设。每个通道支持一个 DMA 请求线和一个触发输入。某些 DMA 请求允许选择请求源。DMA 触发器是从许多可能的输入源中选择的。

2.3.2 DMA 请求和触发多路复用器

DMA 请求旨在加快传输速度以匹配外设（如果有 FIFO 的话还包括 FIFO 寄存器）可以执行的操作，然后 DMA 触发启动传输，用户可以根据设计要求选择软件触发或硬件触发。

DMA0 请求和触发多路复用器在 表 2 中显示。

表 2. DMA0 请求和触发多路复用器

DMA 通道	请求输入	DMA 触发器 MUX
0	Hash-Crypt DMA 请求	DMA0_ ITRIG_ INMUX0
1	空闲通道, 无连接请求	DMA0_ ITRIG_ INMUX1
2	高速 SPI (Flexcomm 8) RX	DMA0_ ITRIG_ INMUX2
3	高速 SPI (Flexcomm 8) TX	DMA0_ ITRIG_ INMUX3

下页继续

表 2. DMA0 请求和触发多路复用器 (续上页)

DMA 通道	请求输入	DMA 触发器 MUX
4	Flexcomm 接口 0 RX / I2C 从机	DMA0_ ITRIG_ INMUX4
5	Flexcomm 接口 0 TX / I2C 主机	DMA0_ ITRIG_ INMUX5
6	Flexcomm 接口 1 RX / I2C 从机	DMA0_ ITRIG_ INMUX6
7	Flexcomm 接口 1 TX / I2C 主机	DMA0_ ITRIG_ INMUX7
8	Flexcomm 接口 2 RX / I2C 从机	DMA0_ ITRIG_ INMUX8
9	Flexcomm 接口 2 RTX / I2C 主机	DMA0_ ITRIG_ INMUX9
10	Flexcomm 接口 3 RX / I2C 从机	DMA0_ ITRIG_ INMUX10
11	Flexcomm 接口 3 TX / I2C 主机	DMA0_ ITRIG_ INMUX11
12	Flexcomm 接口 4 RX / I2C 从机	DMA0_ ITRIG_ INMUX12
13	Flexcomm 接口 4 TX / I2C 主机	DMA0_ ITRIG_ INMUX13
14	Flexcomm 接口 5 RX / I2C 从机	DMA0_ ITRIG_ INMUX14
15	Flexcomm 接口 5 TX / I2C 主机	DMA0_ ITRIG_ INMUX15
16	Flexcomm 接口 6 RX / I2C 从机	DMA0_ ITRIG_ INMUX16
17	Flexcomm 接口 6 TX / I2C 主机	DMA0_ ITRIG_ INMUX17
18	Flexcomm 接口 7 RX / I2C 从机	DMA0_ ITRIG_ INMUX18
19	Flexcomm 接口 7 TX / I2C 主机	DMA0_ ITRIG_ INMUX19
20	空闲通道, 无连接请求	DMA0_ ITRIG_ INMUX20
21	ADC0 FIFO 0	DMA0_ ITRIG_ INMUX21
22	ADC0 FIFO 1	DMA0_ ITRIG_ INMUX22

表 3. DMA1 请求和触发多路复用器

DMA 通道	请求输入	DMA 触发器 mux
0	Hash-Crypt DMA 请求	DMA1_ ITRIG_ INMUX0
1	空闲通道, 无连接请求	DMA1_ ITRIG_ INMUX1
2	高速 SPI (Flexcomm 8) RX	DMA1_ ITRIG_ INMUX2
3	高速 SPI (Flexcomm 8) TX	DMA1_ ITRIG_ INMUX3

下页继续

表 3. DMA1 请求和触发多路复用器 (续上页)

DMA 通道	请求输入	DMA 触发器 mux
4	Flexcomm 接口 0 RX / I2C 从机	DMA1_ ITRIG_ INMUX4
5	Flexcomm 接口 0 TX / I2C 主机	DMA1_ ITRIG_ INMUX5
6	Flexcomm 接口 1 RX / I2C 从机	DMA1_ ITRIG_ INMUX6
7	Flexcomm 接口 1 TX / I2C 主机	DMA1_ ITRIG_ INMUX7
8	Flexcomm 接口 2 RX / I2C 从机	DMA1_ ITRIG_ INMUX8
9	Flexcomm 接口 2 RTX / I2C 主机	DMA1_ ITRIG_ INMUX9

表 4. DMA 触发源

DMA 触发器	DMA0 触发器输入	DMA1 触发器输入
0	引脚中断 0	引脚中断 0
1	引脚中断 1	引脚中断 1
2	引脚中断 2	引脚中断 2
3	引脚中断 3	引脚中断 3
4	定时器 CTIMER0 匹配 0	定时器 CTIMER0 匹配 0
5	定时器 CTIMER0 匹配 1	定时器 CTIMER0 匹配 1
6	定时器 CTIMER1 匹配 0	定时器 CTIMER2 匹配 0
7	定时器 CTIMER1 匹配 1	定时器 CTIMER4 匹配 0
8	定时器 CTMER2 匹配 0	DMA 输出触发 0
9	定时器 CTMER2 匹配 1	DMA 输出触发 1
10	定时器 CTIMER3 匹配 0	DMA 输出触发 2
11	定时器 CTIMER3 匹配 1	DMA 输出触发 3
12	定时器 CTIMER4 匹配 0	SCT0 DMA 请求 0
13	定时器 CTIMER4 匹配 1	SCT0 DMA 请求 1
14	比较器 0 输出	Hash-Crypt 输出 DMA
15	DMA 输出触发 0	N/A
16	DMA 输出触发 1	N/A

下页继续

表 4. DMA 触发源 (续上页)

DMA 触发器	DMA0 触发器输入	DMA1 触发器输入
17	DMA 输出触发 2	N/A
18	DMA 输出触发 3	N/A
19	SCT0 DMA 请求 0	N/A
20	SCT0 DMA 请求 1	N/A
21	Hash-Crypt 输出 DMA	N/A

3 DMA 配置 API 函数

本节列出了 DMA 配置 API 函数并提供了数据 API 示例。

3.1 DMA 配置 API 列表

DMA 寄存器分为 DMA 控制，中断和状态寄存器以及 DMA 通道寄存器。每个 DMA 传输通道由一组寄存器控制，这些寄存器包括 CFG[0:29], CTRLSTAT[0:29] and XFERCFG[0:29]. 存在两个 DMA 控制器：DMA0 和 DMA1（安全）。

本章介绍如何在不更改 SDK 驱动程序的情况下使用 API 配置相关寄存器，并让用户轻松使用 DMA 函数，主要函数原型显示在表 5 中。

表 5. DMA API 调用列表

函数原型	API 描述
*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.

下页继续

表 5. DMA API 调用列表 (续上页)

函数原型	API 描述
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 数据 API 示例

本节列出了以下数据 API 示例：

- 内存到内存
- 内存到外设
- 外设到内存

3.2.1 内存到内存

1. DMA_Init:

   初始步骤包括启用 DMA 时钟，设置描述符表和启用 DMA 外设。

```
/** 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);
```

注意

用户可以使用 DMA1.比如, 'DMA_CreateHandle(&g_DMA_Handle, DMA1, 0)' 表示选择 DMA1_Channel0 作为通道传输数据。

3. DMA_EnableChannel:

   该函数使能 DMA 通道。 要使能所选通道，将使用在步骤 2 中选择的 DMA0_Channel0 来调用 API。

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

注意

值得注意的是 LPC55(S)xx 具有两个 DMA (DMA0 & DMA1)。如果 DMA0 是当前选择的 DMA，‘结构体 COMMON[x]’ 应为‘COMMON[0]’ 。如果 DMA1 是当前选择的 DMA，则结构体为 ‘COMMON[1]’。寄存器类型定义 Typedef ‘DMA_Type’ ，包含所有相关的寄存器。

```
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;
```

图 2. DMA_Type 结构

4. DMA_SetCallback:

   在 DMA IRQ 处理程序中调用此回调函数。 使用以下 API ,用户可以获取中断 A 或 B 的标志，并检测其是否由软件设置。

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

5. DMA_PrepareTransfer:

   此函数根据用户输入准备传输配置结构。 以下代码表示用户可以根据数据传输需求配置 DMA 传输源地址，目标地址，DMA 传输类型和自定义描述符。

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

注意

第六个参数决定是否应该增加源地址和目标地址。 例如，源地址和目的地址都应增加以按顺序在内存之间复制数据。

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

6. DMA_SubmitTransfer:

   此函数根据传输配置结构提交 DMA 传输请求。以下代码表示用户可以通过配置结构'dma_xfercfg_t' "来完成 DMA 传输配置，其中包括重载通道配置，执行软件触发和其它信息。

```
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;
```

图 3. DMA_xfercfg_t 结构

然后，用户可以使用以下 API 创建在传输链中使用的特定 DMA 描述符。

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

注意

CFG 寄存器位 1" HWTRIGEN"的默认值为" 0"。 这意味着未使用硬件触发器。 要使用硬件触发 DMA 传输（如引脚中断 0，定时器 CTIMER0 匹配 0 等），请按照以下步骤操作完成额外的配置。

a. CFG 寄存器位 1 'HWTRIGEN' 应该使用以下伪代码设置。

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

b. XFERCFGn 寄存器的第 2 位软件触发应被禁用，并可以使用以下伪代码等待硬件触发信号。

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

c. 用户应通过配置寄存器" DMA0_ITRIG_INMUX [0:22]"或" DMA1_ITRIG_INMUX [0：9]"来选择" DMA 触发源"。 以下伪代码意味着我们可以选择硬件触发器 2" 引脚中断 2"做为 DMA0_Channlx 的请求输入。

```
Base->DMA0_ITRIG_INMUX[x] = 0x2;
config->xfercfg.srcInc = 0;
config->xfercfg.dstInc = 1;
config->isPeriph = true;
```

通过以上主要步骤的配置，DMA 已准备好传输数据。 现在，用户可以通过检测在" DMA_Callback"函数中更改的" g_Transfer_Done"值来等待 DMA 传输完成。

```
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 内存到外设

用户可以将数据从内存传输到外围设备。 例如，将数据从 RAM 传输到 USART0_FIFOWR。 除了步骤 5 以外，主要步骤与内存到内存部分中的说明相似。

```
/** 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);
```

注意

考虑到外围设备（USART0_FIFOWR）具有固定的基地址：0x4008 6000h + 0xE20，因此该地址不会递增。 如果用作“目标数据地址”，而将内存（RAM）用作“源数据地址”，则其地址应递增。

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

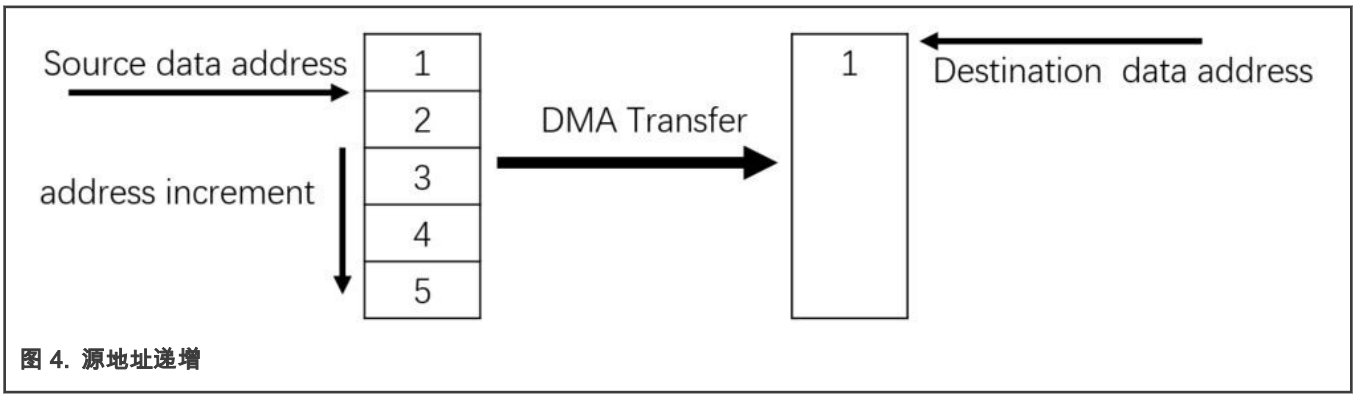

图 4. 源地址递增

3.2.3 外设到内存

用户可以将数据从外围设备传输到内存。 例如，如果传感器和微控制器单元正在通过 USART0 通信，则用户可以通过将数据从 USART0_FIFORD 传输到 RAM 来将传感器收集的数据存储到内存中。 除了步骤 5 以外，主要步骤与内存到内存部分中的说明相似。

1. 寄存器 FIFORD 位 0-8 可用于存储从 FIFO 接收的数据。 使用的位数取决于 DATALEN 和 PARITYSEL 设置。
2. 例如，用户可以使用以下 API 配置“ DMA_PrepareTransfer”函将一个字符从 USART0-> FIFORD 传输到 RAM。

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

注意

考虑到外围设备（USART0_NS-> FIFORD）具有固定的基地址：0x4008 6000h + 0xE30，因此其地址不会递增。 如果用户将其用作“源数据地址”，而将内存（RAM）用作“目标数据地址”，则该地址应递增。

```
/*Peripheral register - source doesn't increment */
```

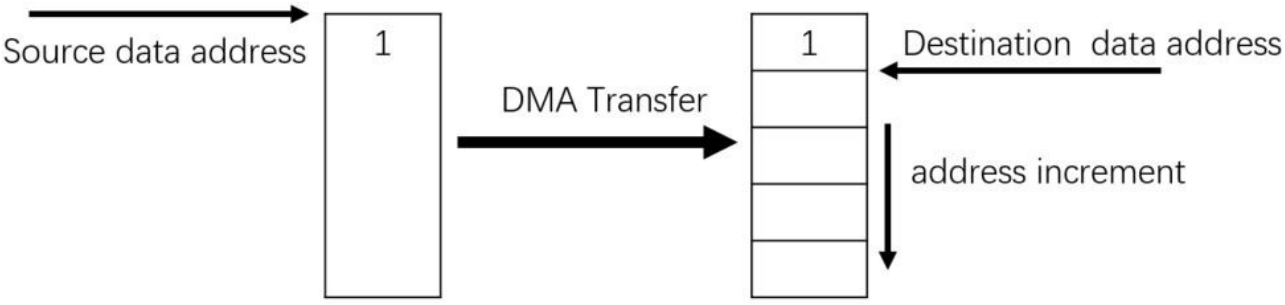

图 5. 目标地址递增

4 DMA 验证

本应用笔记提供了一些示例来完成 DMA 传输工作，包括内存到内存，内存到外设以及外设到内存模式。本应用笔记以 LPC55S6x 为例来说明双 DMA 的功能。

以下演示基于 IAR，也支持 NXP 的 MCUXpresso IDE 平台。 用户可以在 LPC55S6x 上下载相关项目以验证 Dual-DMA 功能。本演示使用 USB 虚拟 COM（VCOM）显示日志信息。

4.1 硬件设计

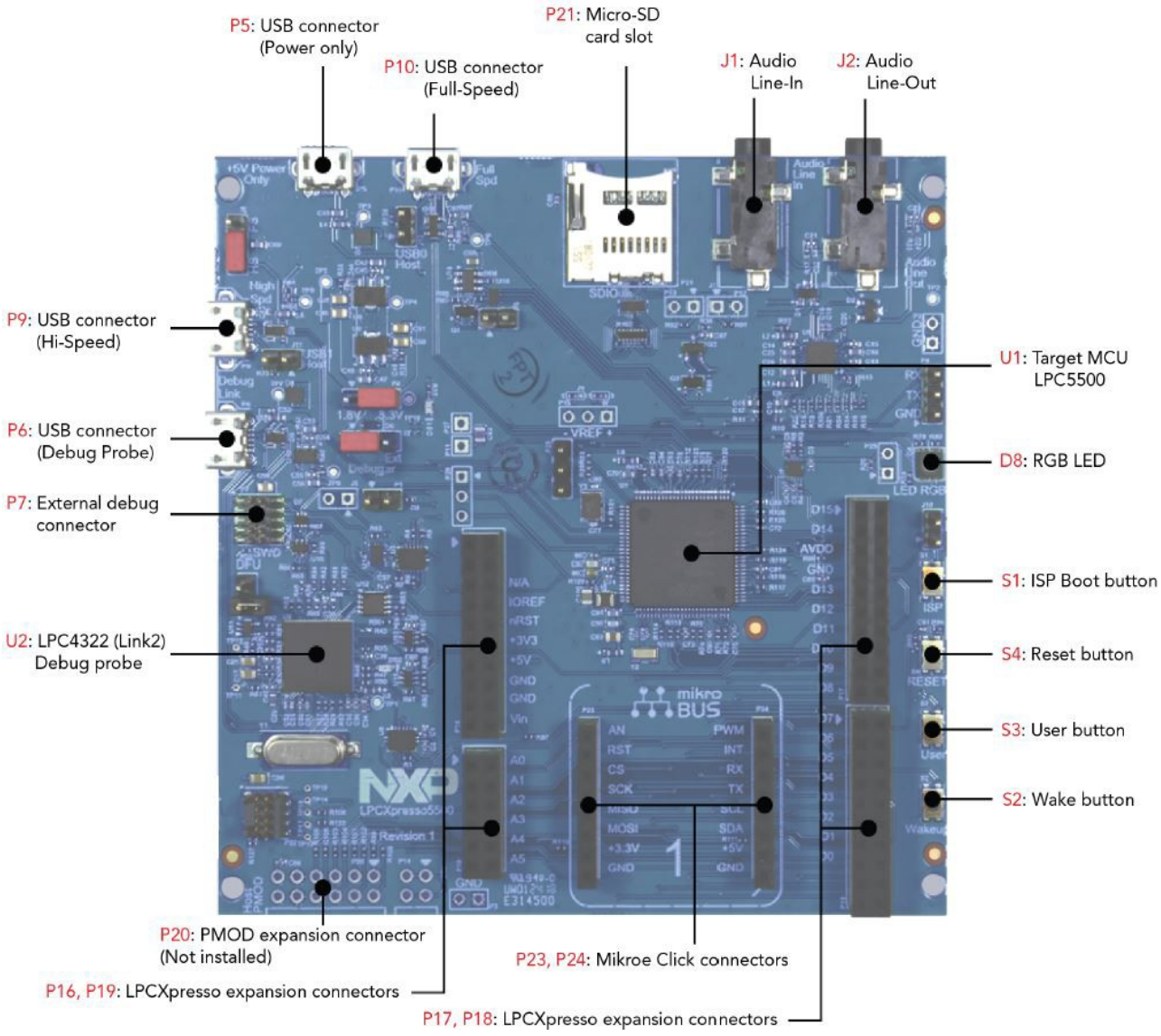

图 6. LPCXpresso55S69 EVK

要下载和调试项目并查看串行端口数据，用户可以使用 USB 接口将 USB 端口 P6 与 PC 连接。

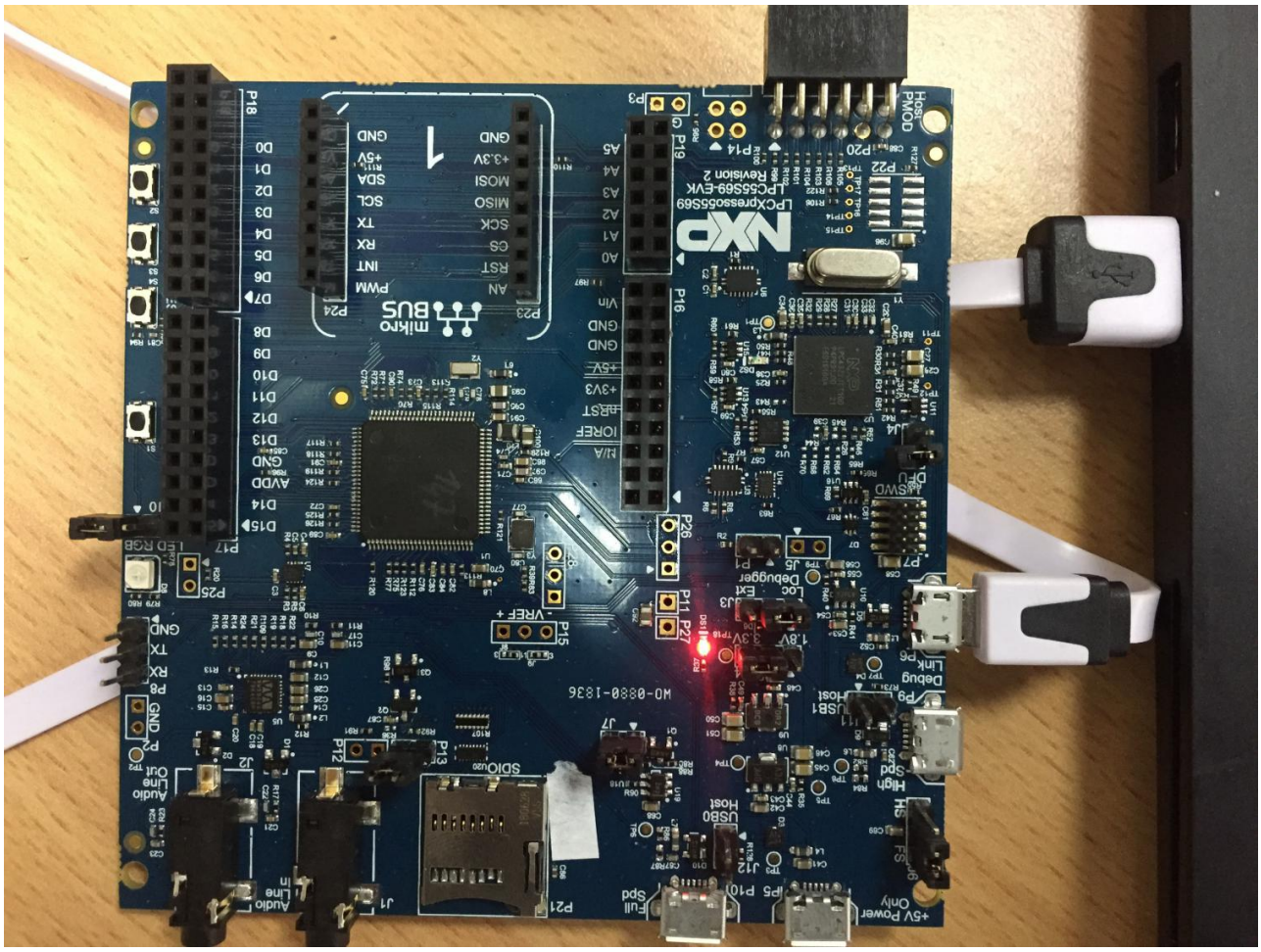

图 7. 连接 LPCXpresso55S69 到 PC

4.2 软件设置

使用两个 IDE 来验证 DMA 示例项目：

- IAR Embedded Workbench v8.32.1
- MCUXpresso IDE v10.3.0 (可以从 www.nxp.com 下载)

如要使用 IAR IDE 编译和加载软件示例，请遵循以下步骤。

1. 解压缩项目文件夹“…\LPC55S6x Dual-DMA.zip” 以进行测试。
2. 将 LPCXpresso55S69 板的 USB 端口 P6 连接到 PC（J3 设置为 Loc，P4 设置为 3.3 V，J6 设置为 FS）。
3. 通过在快速开始菜单中单击“Compile（Ctrl + F7）”或 “Make（F7）”来编译所有项目。

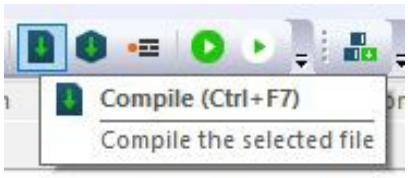

图 8. 编译 IAR 工程项目

4. 点击‘Download and Debug (Ctrl+D)’以启动调试器。 此步骤将编译并刷新代码，然后启动调试器。

图 9. 在 IAR 中下载和调试项目

5. 终端软件, 比如 tera-term。

4.3 程序验证

要在下载 DMA 项目后运行代码，请按“RESET（S4）”按钮。 根据 USB 虚拟 COM（VCOM）端口的日志信息选择 DMA 传输模式。

```
                        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)
```

图 10. DMA 传输模式选择菜单

- 输入“1”通过软件触发 DMA 内存到内存的传输。

```
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.
```

图 11. 软件触发内存到内存传输

- 输入“2”通过硬件触发 DMA 内存到外设的传输。

```
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.
```

图 12. 硬件触发内存到外设传输

- 输入“3”通过软件触发 DMA 内存到外设的传输。

```
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.
```

图 13. 软件触发内存到外设传输

- 输入“4”通过软件触发 DMA 外设到内存的传输。

```
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.
```

图 14. 软件触发外设到内存传输

5 总结

本应用笔记随附的软件示例展示了 LPC55S6x DMA 模块的基本配置。可以根据实际需要选择 DMA0 和 DMA1 通道来传输数据。

6 修订记录

表 6. 修订记录

版本号	日期	说明
0	2019 年 2 月 14 日	初始版本
1	2020 年 2 月 26 日	更新了图 1
2	2020 年 10 月 27 日	“LPC55S6x”替换成“LPC55(S)xx”

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — 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.

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.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. 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.

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

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

Date of release: 2020 年 10 月 27 日

Document identifier: AN12351

arm