

AT32F435/437 DMA使用指南

前言

AT32F435/437 拥有 2 个 DMA 控制器（DMA1/DMA2），每个 DMA 支持 7 个通道且外设的 DMA 请求可映射到任意通道上。本文主要就 DMA 的基本功能进行讲解和案例解析。

支持型号列表：

支持型号	AT32F435 系列
	AT32F437 系列

目录

1	DMA 简介	6
2	DMAMUX 简介	7
3	DMA 功能解析	9
3.1	可编程数据宽度	9
3.2	配置 DMAMUX	9
3.3	配置请求生成器模块	10
3.4	配置请求同步模块	10
4	DMA 配置解析	11
4.1	函数接口	11
4.2	数据流配置	11
4.3	配置流程	12
5	案例 数据从 FLASH 传输到 SRAM	13
5.1	功能简介	13
5.2	资源准备	13
5.3	软件设计	13
5.4	实验效果	14
6	案例 TMR 产生 DMA 请求将数据从 SRAM 传输到 GPIO	15
6.1	功能简介	15
6.2	资源准备	15
6.3	软件设计	15
6.4	实验效果	17
7	案例 DMA 请求生成器产生 DMA 请求	18

7.1	功能简介	18
7.2	资源准备	18
7.3	软件设计	18
7.4	实验效果	20
8	案例 DMA 传输数据需等待 DMA 请求同步信号	21
8.1	功能简介	21
8.2	资源准备	21
8.3	软件设计	21
8.4	实验效果	23
9	文档版本历史	24

表目录

表 1. 各 IP 对应 ID 号列表	7
表 2. 通道配置函数列表	11
表 3. 文档版本历史	24

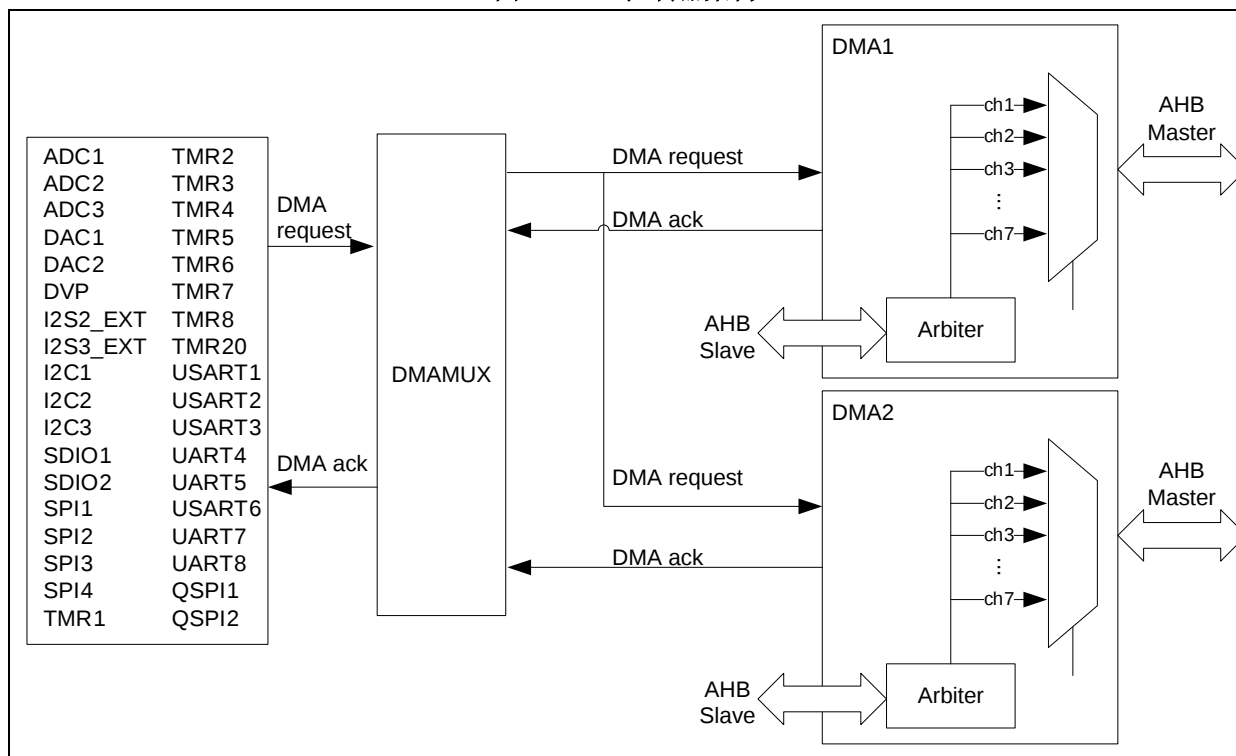
图目录

图 1. DMA 控制器架构	6
图 2. PWIDTH: byte, MWIDTH: half-word.....	9
图 3. PWIDTH: half-word, MWIDTH: word.....	9
图 4. Data to gpio 传输实验结果	17

1 DMA 简介

DMA 控制器的作用不仅在增强系统性能并减少处理器的中断生成，而且还针对 32 位 MCU 应用程序专门优化设计。DMA 控制器为存储器到存储器，存储器到外设和外设到存储器的传输提供了 7 个通道。每个通道都支持外设的 DMA 请求映射到任意通道上。

图 1. DMA 控制器架构



2 DMAMUX 简介

对于如何将外设的 DMA 请求映射到任意的数据流通道上，就需要使用到 DMAMUX。DMAMUX 针对每个外设都设计了独有的 ID 号，使用者只需要将此 ID 号写入对应的寄存器中并打开 DMAMUX 功能即可。DMAMUX 的引入，使得 DMA 相较于传统 DMA 控制器变得更加灵活，使用者可以随意的分配 7 个通道的使用情况，不必再纠结与某个 IP 的 DMA 请求只能固定使用在某个或某几个通道上。

各 IP 对应 ID 号如下表：

表 1. 各 IP 对应 ID 号列表

DMAMUX 请求	来源	DMAMUX 请求	来源	DMAMUX 请求	来源	DMAMUX 请求	来源
1	DMA_MUXREQG1	33	UART5_TX	65	TMR3_OVERFLOW	97	reserved
2	DMA_MUXREQG2	34	reserved	66	TMR3_TRIG	98	reserved
3	DMA_MUXREQG3	35	reserved	67	TMR4_CH1	99	reserved
4	DMA_MUXREQG4	36	ADC2	68	TMR4_CH2	100	reserved
5	ADC1	37	ADC3	69	TMR4_CH3	101	reserved
6	DAC1	38	reserved	70	TMR4_CH4	102	reserved
7	reserved	39	SDIO1	71	TMR4_UP	103	SDIO2
8	TMR6_OVERFLOW	40	QSPI1	72	TMR5_CH1	104	QSPI2
9	TMR7_OVERFLOW	41	DAC2	73	TMR5_CH2	105	DVP
10	SPI1_RX	42	TMR1_CH1	74	TMR5_CH3	106	SPI4_RX
11	SPI1_TX	43	TMR1_CH2	75	TMR5_CH4	107	SPI4_TX
12	SPI2_RX	44	TMR1_CH3	76	TMR5_OVERFLOW	108	reserved
13	SPI2_TX	45	TMR1_CH4	77	TMR5_TRIG	109	reserved
14	SPI3_RX	46	TMR1_OVERFLOW	78	reserved	110	I2S2_EXT_RX
15	SPI3_TX	47	TMR1_TRIG	79	reserved	111	I2S2_EXT_TX
16	I2C1_RX	48	TMR1_HALL	80	reserved	112	I2S3_EXT_RX
17	I2C1_TX	49	TMR8_CH1	81	reserved	113	I2S3_EXT_TX
18	I2C2_RX	50	TMR8_CH2	82	reserved	114	USART6_RX
19	I2C2_TX	51	TMR8_CH3	83	reserved	115	USART6_TX
20	I2C3_RX	52	TMR8_CH4	84	reserved	116	UART7_RX
21	I2C3_TX	53	TMR8_OVERFLOW	85	reserved	117	UART7_TX
22	reserved	54	TMR8_TRIG	86	TMR20_CH1	118	UART8_RX
23	reserved	55	TMR8_HALL	87	TMR20_CH2	119	UART8_TX
24	USART1_RX	56	TMR2_CH1	88	TMR20_CH3	120	reserved
25	USART1_TX	57	TMR2_CH2	89	TMR20_CH4	121	reserved
26	USART2_RX	58	TMR2_CH3	90	TMR20_OVERFLOW	122	reserved
27	USART2_TX	59	TMR2_CH4	91	reserved	123	reserved
28	USART3_RX	60	TMR2_OVERFLOW	92	reserved	124	reserved
29	USART3_TX	61	TMR3_CH1	93	TMR20_TRIG	125	reserved
30	UART4_RX	62	TMR3_CH2	94	TMR20_HALL	126	TMR2_TRIG

DMAMUX 请求	来源	DMAMUX 请求	来源	DMAMUX 请求	来源	DMAMUX 请求	来源
31	UART4_TX	63	TMR3_CH3	95	reserved	127	reserved
32	UART5_RX	64	TMR3_CH4	96	reserved		

注：表格中“DMAMUX 请求”为ID 号；“来源”为各IP 的DMA 请求。

3 DMA 功能解析

3.1 可编程数据宽度

DMA 控制器的通道可支持传输不同数据宽度,byte/halfword/word。通过 DMA_CxCTRL 中的 PWIDTH 和 MWIDTH 位可以对源数据和目标数据的数据宽度进行编程,通常情况下需要设置 PWIDTH 和 MWIDTH 位相等,当 PWIDTH 不等于 MWIDTH 时,会依据 PWIDTH/ MWIDTH 设定将资料对齐。

图 2. PWIDTH: byte, MWIDTH: half-word

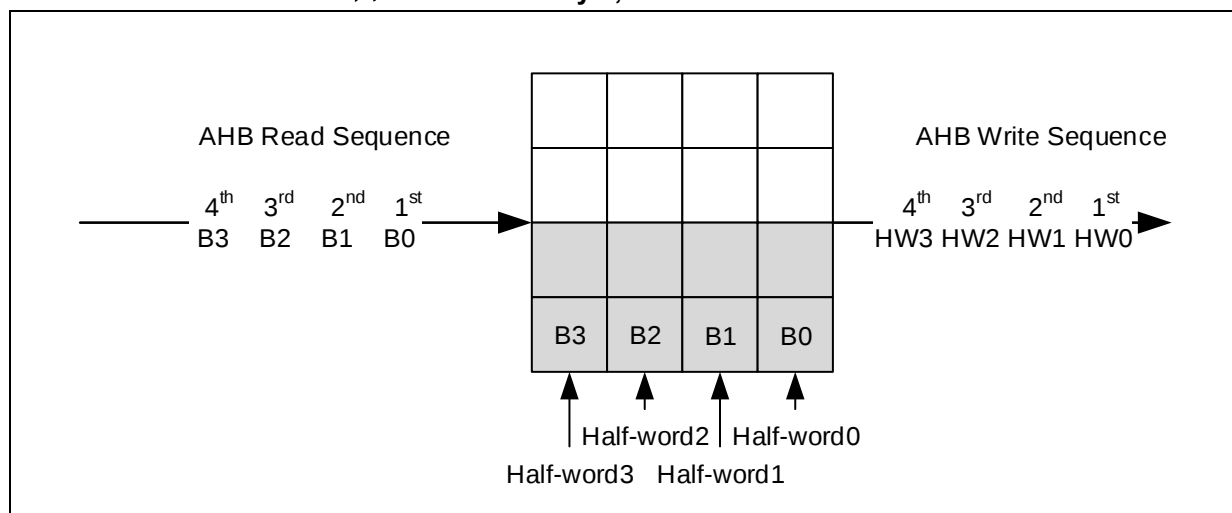
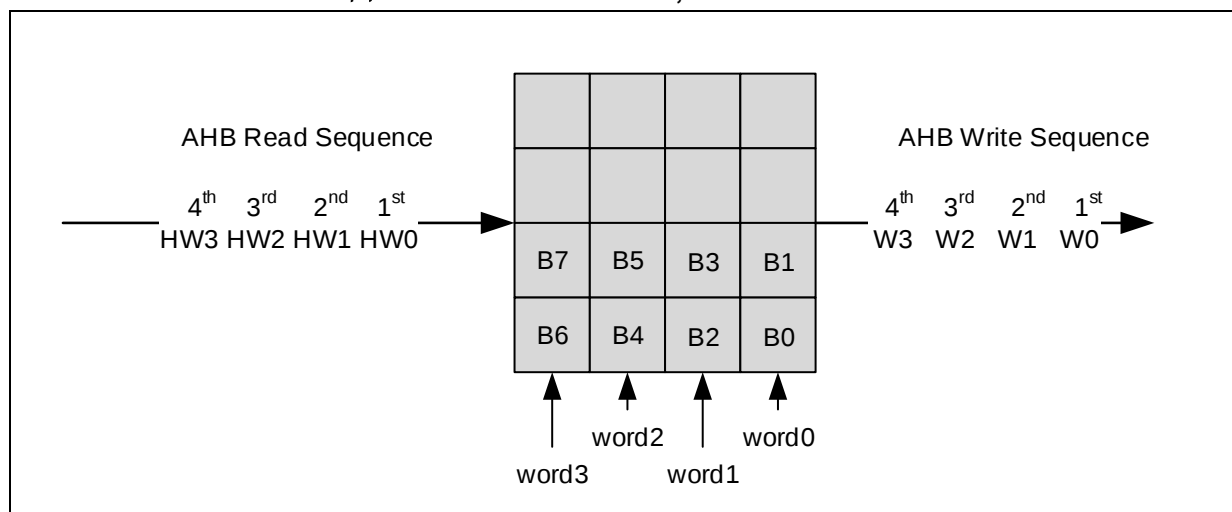


图 3. PWIDTH: half-word, MWIDTH: word



3.2 配置 DMAMUX

在 M2P 与 P2M 模式下,必须配置 DMAMUX,否则 DMA 不会响应外设 DMA 请求。DMAMUX 的作用是为外设的 DMA 请求复用通道,即任何一个外设的 DMA 请求可以映射到 DMA1/DMA2 的任意通道,这大大增加了 DMA 通道分配的灵活性。

配置 DMAMUX 比较减到,只需调用专门提供的两个接口函数即可:

```
/* 使能 DMAMUX 功能 */
void dmamux_enable(dma_type *dma_x, confirm_state new_state);
/* 配置 DMAMUX 通道 */
```

```
void dmamux_init(dmamux_channel_type *dmamux_channelx, dmamux_request_id_sel_type dmamux_req_sel);
```

3.3 配置请求生成器模块

在配置了 DMAMUX 时，可选择配置 DMA 请求生成器模块，模块一共有 4 个请求生成器通道。此模块无需任何传统外设（如 TIMER、SPI 等）提供 DMA 请求，可通过外部 EXINT 输入作为 DMA 请求源输入。

配置请求生成器模块较为简单，只需调用专门提供的接口函数即可：

```
/* 初始化请求生成器结构体参数 */
dmamux_generator_default_para_init(&dmamux_gen_init_struct);
/* 配置请求生成器结构体 */
dmamux_gen_init_struct.gen_polarity = DMAMUX_GEN_POLARITY_RISING;
dmamux_gen_init_struct.gen_request_number = 4;
dmamux_gen_init_struct.gen_signal_sel = DMAMUX_GEN_ID_EXINT0;
dmamux_gen_init_struct.gen_enable = TRUE;
/* 配置请求生成器 */
dmamux_generator_config(DMA2MUX_GENERATOR1, &dmamux_gen_init_struct);
```

3.4 配置请求同步模块

在配置了 DMAMUX 时，可选择配置 DMA 同步模块，模块一共有 7 个同步通道。使能此功能后，当外设产生 DMA 请求时，DMA 不会马上响应并传输数据，而是要等待同步信号的到来，当接收到同步信号后，DMA 才会根据配置传输数据；同步信号可由外部 EXINT 输入提供

配置同步模块较为简单，只需调用专门提供的接口函数即可：

```
/* 初始化同步模块结构体参数 */
dmamux_sync_default_para_init(&dmamux_sync_init_struct);
/* 配置同步模块结构体参数 */
dmamux_sync_init_struct.sync_request_number = 4;
dmamux_sync_init_struct.sync_signal_sel = DMAMUX_SYNC_ID_EXINT0;
dmamux_sync_init_struct.sync_polarity = DMAMUX_SYNC_POLARITY_RISING;
dmamux_sync_init_struct.sync_event_enable = FALSE;
dmamux_sync_init_struct.sync_enable = TRUE;
/* 配置同步模块 */
dmamux_sync_config(DMA2MUX_CHANNEL4, &dmamux_sync_init_struct);
```

4 DMA 配置解析

以下对 DMA 的配置接口及流程进行说明。

4.1 函数接口

表 2. 通道配置函数列表

<pre>/* 复位通道 */ void dma_reset(dma_channel_type *dma_channel);</pre>
<pre>/* 初始化 DMA 结构体参数 */ void dma_default_para_init(dma_init_type *dma_init_struct);</pre>
<pre>/* 初始化通道 */ void dma_init(dma_channel_type *dma_channel, dma_init_type *dma_init_struct);</pre>
<pre>/* 使能通道 */ void dma_channel_enable(dma_channel_type *dma_channel, confirm_state new_state);</pre>
<pre>/* 使能 DMAMUX */ void dmamux_enable(dma_type *dma_x, confirm_state new_state);</pre>
<pre>/* 写入 DMA 请求 ID 号 */ void dmamux_init(dmamux_channel_type *dmamux_channel, dmamux_request_id_sel_type dmamux_req_sel);</pre>

4.2 数据流配置

■ 设置外设地址（CxPADDR 寄存器）

数据传输的初始外设地址，在传输过程中不可被改变。

■ 设置存储器地址（CxMADDR 寄存器）

数据传输的初始内存地址，在传输过程中不可被改变。

■ 配置数据传输量（CxDTCNT 寄存器）

可编程的传输数据长度最大为 65535。在传输过程中，该传输数据量的值会逐渐递减。

■ 数据流配置（CxCTRL 寄存器）

包含通道优先级，数据传输的方向、宽度、地址增量模式、循环模式和中断方式。

➤ 优先级（CHPL）

分为 4 个等级，最高优先级、高优先级、中等优先级和低优先级。

若有 2 个流优先级设定相同，则较低编号的流有较高的优先权。举例，流 1 优先于流 2。

➤ 数据传输方向（DTD）

分为存储器到外设（M2P），外设到存储器（P2M）或存储器到存储器（M2M）传输。

在存储器到存储器传输模式下不允许使用循环模式、双缓冲模式和直接模式。

➤ 数据传输宽度（PWIDTH/ MWIDTH）

根据实际使用情景，可配置宽度为 byte、halfword、word。

➤ 地址增量模式（PINC/MINCM）

当通道配置设定为增量模式时，下一笔传输的地址将是前一笔传输地址加上传输宽度

(PWIDTH/MWIDTH)。

➤ 循环模式 (LM)

当流配置设定为循环模式时，在最后一次传输后 CxDTCNT 寄存器的内容会恢复成初始值。

■ 使能 DMAMUX (MUXSEL 寄存器的 TBL_SEL 位)

在非存储器到存储器 (M2M) 模式下时，需要使能 DMAMUX 功能，才能启动数据流响应外设的 DMA 请求。

■ 写入外设 ID 号 (MUXCxCTRL 寄存器的 REQSEL)

在非存储器到存储器 (M2M) 模式下时，需要将外设的 DMA 请求 ID 号写入，才能启动数据流响应外设的 DMA 请求。

■ 打开数据流 (CxCTRL 寄存器的 CHEN 位)

4.3 配置流程

- 打开 DMA 时钟；
- 调用通道复位函数复位数据流；
- 调用结构体初始化函数初始化通道配置结构体；
- 调用初始化函数初始化通道；
- 调用 DMAMUX 使能函数以及 ID 号写入函数配置 DMAMUX 相关内容；
- 调用通道使能函数开启通道。

5 案例 数据从 FLASH 传输到 SRAM

5.1 功能简介

实现了使用 DMA 将数据从片上 FLASH 搬运到内部 SRAM 中。

5.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

2) 软件环境

project\at_start_f4xx\examples\dma\flash_to_sram

5.3 软件设计

1) 配置流程

- 开启 DMA 外设时钟
- 配置 DMA 通道
- 开启传输完成中断
- 开启通道
- 等待数据传输完成
- 比较数据传输是否正确

2) 代码介绍

■ main 函数代码描述

```
int main(void)
{
    /* 初始化系统时钟 */
    /* initial system clock */
    system_clock_config();
    /* 板载初始化 */
    /* at board initial */
    at32_board_init();
    /* 打开 DMA1 时钟 */
    crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);

    /* dma1 通道初始化 */
    dma_reset(DMA1_CHANNEL1);
    dma_init_struct.buffer_size = BUFFER_SIZE;
    dma_init_struct.direction = DMA_DIR_MEMORY_TO_MEMORY;
    dma_init_struct.memory_base_addr = (uint32_t)dst_buffer;
    dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_WORD;
    dma_init_struct.memory_inc_enable = TRUE;
    dma_init_struct.peripheral_base_addr = (uint32_t)src_const_buffer;
    dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_WORD;
```

```
dma_init_struct.peripheral_inc_enable = TRUE;
dma_init_struct.priority = DMA_PRIORITY_MEDIUM;
dma_init_struct.loop_mode_enable = FALSE;
dma_init(DMA1_CHANNEL1, &dma_init_struct);

/* 打开传输完成中断 */
dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, TRUE);

/* dma1 channel1 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(DMA1_Channel1_IRQn, 1, 0);

dma_channel_enable(DMA1_CHANNEL1, TRUE);

/* wait the end of transmission */
while(data_counter_end != 0)
{
}

/* check if the transmitted and received data are equal */
transfer_status = buffer_compare(src_const_buffer, dst_buffer, BUFFER_SIZE);

/* 判断数据是否正确，正确则点亮 LED 灯 */
if(transfer_status == SUCCESS)
{
    /* turn led2/led3/led4 on */
    at32_led_on(LED2);
    at32_led_on(LED3);
    at32_led_on(LED4);
}

while(1)
{
}
```

5.4 实验效果

- 如若数据传输无误，LED2/3/4 会点亮。

6 案例 TMR 产生 DMA 请求将数据从 SRAM 传输到 GPIO

6.1 功能简介

本案例介绍 TMR 产生 DMA 请求将数据从 SRAM 传输到 GPIOC，可通过逻辑分析仪等仪器查看波形。

6.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

2) 软件环境

project\at_start_f4xx\examples\dma\dmamux_data_to_gpio

6.3 软件设计

1) 配置流程

- 开启 DMA/TMR2/GPIOC 外设时钟
- 配置 DMA 通道
- 配置 TMR2 开启溢出 DMA 请求
- 开启传输完成中断
- 开启 DMAMUX 功能
- 开启通道
- 开启 TMR2,使其溢出中断产生 DMA 请求

2) 代码介绍

■ main 函数代码描述

```
#define BUFFER_SIZE    16
uint16_t src_buffer[BUFFER_SIZE] = {0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008,
                                      0x0009, 0x000a, 0x000b, 0x000c, 0x000d, 0x000e, 0x000f, 0x0010};

Int main(void)
{
    /* 系统时钟初始化 */
    system_clock_config();
    /* 板载初始化 */
    at32_board_init();
    /* 开启 dma2/gpioc/tmr2 时钟 */
    crm_periph_clock_enable(CRM_DMA2_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_TMR2_PERIPH_CLOCK, TRUE);

    /* config gpioc pin for output mode */
    gpio_init_struct.gpio_pins = GPIO_PINS_ALL;
    gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
```

```
gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
gpio_init(GPIOC, &gpio_init_struct);
/* 初始化 TMR2 */
tmr_base_init(TMR2, 0xFF, 0);
tmr_cnt_dir_set(TMR2, TMR_COUNT_UP);

/* 开启 TMR2 溢出 DMA 请求 */
tmr_dma_request_enable(TMR2, TMR_OVERFLOW_DMA_REQUEST, TRUE);

/* dma2 通道 1 初始化 */
dma_reset(DMA2_CHANNEL1);
dma_init_struct.buffer_size = BUFFER_SIZE;
dma_init_struct.direction = DMA_DIR_MEMORY_TO_PERIPHERAL;
dma_init_struct.memory_base_addr = (uint32_t)src_buffer;
dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
dma_init_struct.memory_inc_enable = TRUE;
dma_init_struct.peripheral_base_addr = (uint32_t)&GPIOC->odr;
dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
dma_init_struct.peripheral_inc_enable = FALSE;
dma_init_struct.priority = DMA_PRIORITY_MEDIUM;
dma_init_struct.loop_mode_enable = FALSE;
dma_init(DMA2_CHANNEL1, &dma_init_struct);
/* 开启传输完成中断 */
dma_interrupt_enable(DMA2_CHANNEL1, DMA_FDT_INT, TRUE);

/* dma2 channel1 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(DMA2_Channel1_IRQn, 1, 0);

/*开启 TMR2 的 DMAMUX 功能 */
dmamux_enable(DMA2, TRUE);
dmamux_init(DMA2MUX_CHANNEL1, DMAMUX_DMAREQ_ID_TMR2_OVERFLOW);

/* enable dma channel */
dma_channel_enable(DMA2_CHANNEL1, TRUE);

/* enable tmr2 */
tmr_counter_enable(TMR2, TRUE);

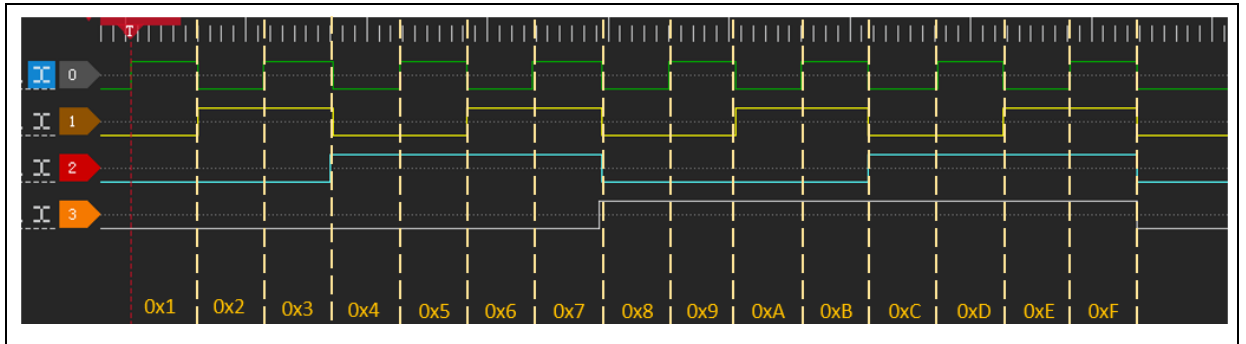
while(1)
{
}
}
```

6.4 实验效果

- 通过抓取 GPIOC 的波形，可查看数据，下面抓取 GPIO0-GPIO3 的数据。

如下为使用逻辑分析仪抓取的 GPIO0-GPIO3 的输出波形，依次为程序实现定义 buffer 内的数据。

图 4. Data to gpio 传输实验结果



7 案例 DMA 请求生成器产生 DMA 请求

7.1 功能简介

实现了通过板载按键，使 DMAMUX 产生 DMA 请求将数据从源地址传输到目的地址。

7.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

2) 软件环境

project\at_start_f4xx\examples\dma\dmamux_genertor_exint

7.3 软件设计

1) 配置流程

- 开启 DMA 外设时钟
- 配置 DMA 通道，配置 DMAMUX 请求生成器模块
- 开启传输完成中断
- 开启通道
- 等待数据传输完成

2) 代码介绍

■ main 函数代码描述

```
int main(void)
{
    /* initial system clock */
    system_clock_config();

    /* at board initial */
    at32_board_init();

    /* enable dma2/gpioa clock */
    crm_periph_clock_enable(CRM_DMA2_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);

    /* config pa1 for input mode */
    gpio_init_struct.gpio_pins = GPIO_PINS_0;
    gpio_init_struct.gpio_mode = GPIO_MODE_INPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
    gpio_init(GPIOA, &gpio_init_struct);

    scfg_exint_line_config(SCFG_PORT_SOURCE_GPIOA, SCFG_PINS_SOURCE0);
```

```
exint_default_para_init(&exint_init_struct);
exint_init_struct.line_enable = TRUE;
exint_init_struct.line_mode = EXINT_LINE_INTERRUPT;
exint_init_struct.line_select = EXINT_LINE_0;
exint_init_struct.line_polarity = EXINT_TRIGGER_RISING_EDGE;
exint_init(&exint_init_struct);

/* exint line1 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(EXINT0_IRQn, 1, 0);

/* dma2 channel4 configuration */
dma_reset(DMA2_CHANNEL4);
dma_init_struct.buffer_size = BUFFER_SIZE;
dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
dma_init_struct.memory_base_addr = (uint32_t)dst_buffer;
dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
dma_init_struct.memory_inc_enable = TRUE;
dma_init_struct.peripheral_base_addr = (uint32_t)src_buffer;
dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
dma_init_struct.peripheral_inc_enable = TRUE;
dma_init_struct.priority = DMA_PRIORITY_MEDIUM;
dma_init_struct.loop_mode_enable = FALSE;
dma_init(DMA2_CHANNEL4, &dma_init_struct);

/* genertor1 configuration */
dmamux_generator_default_para_init(&dmamux_gen_init_struct);
dmamux_gen_init_struct.gen_polarity = DMAMUX_GEN_POLARITY_RISING;
dmamux_gen_init_struct.gen_request_number = 4;
dmamux_gen_init_struct.gen_signal_sel = DMAMUX_GEN_ID_EXINT0;
dmamux_gen_init_struct.gen_enable = TRUE;
dmamux_generator_config(DMA2MUX_GENERATOR1, &dmamux_gen_init_struct);

/* enable transfer full data interrupt */
dma_interrupt_enable(DMA2_CHANNEL4, DMA_FDT_INT, TRUE);

/* dma2 channel4 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(DMA2_Channel4_IRQn, 1, 0);

/* dmamux function enable */
dmamux_enable(DMA2, TRUE);
dmamux_init(DMA2MUX_CHANNEL4, DMAMUX_DMAREQ_ID_REQ_G1);

/* enable dma channe4 */
```

```
dma_channel_enable(DMA2_CHANNEL4, TRUE);

while(1)
{
}
}
```

7.4 实验效果

通过板载 user 按键，每按一次，DMA 会传输 4 个数据到 dst_buffer 数组，当数据传输 16 笔时会产生 DMA 完成中断。

8 案例 DMA 传输数据需等待 DMA 请求同步信号

8.1 功能简介

实现了当 TMR 产生 DMA 请求后，需等待通过板载按键输入的同步信号，数据才会从源地址传输到目的地址。

8.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

2) 软件环境

project\at_start_f4xx\examples\dma\dmamux_synchronization_exint

8.3 软件设计

1) 配置流程

- 开启 DMA 外设时钟
- 配置 DMA 通道，配置 DMAMUX 同步模块
- 配置 TMR
- 开启传输完成中断
- 开启通道
- 等待数据传输完成

2) 代码介绍

■ main 函数代码描述

```
int main(void)
{
    /* initial system clock */
    system_clock_config();

    /* at board initial */
    at32_board_init();

    /* enable dma2/gpioa clock */
    crm_periph_clock_enable(CRM_DMA2_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

    /* config pa1 for input mode */
    gpio_init_struct.gpio_pins = GPIO_PINS_0;
    gpio_init_struct.gpio_mode = GPIO_MODE_INPUT;
    gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
    gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
    gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
```

```
gpio_init(GPIOA, &gpio_init_struct);

scfg_exint_line_config(SCFG_PORT_SOURCE_GPIOA, SCFG_PINS_SOURCE0);

exint_default_para_init(&exint_init_struct);
exint_init_struct.line_enable = TRUE;
exint_init_struct.line_mode = EXINT_LINE_INTERRUPT;
exint_init_struct.line_select = EXINT_LINE_0;
exint_init_struct.line_polarity = EXINT_TRIGGER_RISING_EDGE;
exint_init(&exint_init_struct);

/* exint line1 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(EXINT0_IRQn, 1, 0);

/* dma2 channel4 configuration */
dma_reset(DMA2_CHANNEL4);
dma_init_struct.buffer_size = BUFFER_SIZE;
dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
dma_init_struct.memory_base_addr = (uint32_t)dst_buffer;
dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
dma_init_struct.memory_inc_enable = TRUE;
dma_init_struct.peripheral_base_addr = (uint32_t)src_buffer;
dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
dma_init_struct.peripheral_inc_enable = TRUE;
dma_init_struct.priority = DMA_PRIORITY_MEDIUM;
dma_init_struct.loop_mode_enable = FALSE;
dma_init(DMA2_CHANNEL4, &dma_init_struct);

dmamux_sync_default_para_init(&dmamux_sync_init_struct);
dmamux_sync_init_struct.sync_request_number = 4;
dmamux_sync_init_struct.sync_signal_sel = DMAMUX_SYNC_ID_EXINT0;
dmamux_sync_init_struct.sync_polarity = DMAMUX_SYNC_POLARITY_RISING;
dmamux_sync_init_struct.sync_event_enable = FALSE;
dmamux_sync_init_struct.sync_enable = TRUE;
dmamux_sync_config(DMA2MUX_CHANNEL4, &dmamux_sync_init_struct);

/* enable transfer full data interrupt */
dma_interrupt_enable(DMA2_CHANNEL4, DMA_FDT_INT, TRUE);

/* dma2 channel4 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(DMA2_Channel4_IRQn, 1, 0);

/* dmamux function enable */
dmamux_enable(DMA2, TRUE);
```

```
dmamux_init(DMA2MUX_CHANNEL4, DMAMUX_DMAREQ_ID_TMR1_OVERFLOW);

/* enable dma channe4 */
dma_channel_enable(DMA2_CHANNEL4, TRUE);

/* tmr1 configuration */
tmr_base_init(TMR1, 5000, 5000);
tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);

/* enable tmr1 overflow edam request */
tmr_dma_request_enable(TMR1, TMR_OVERFLOW_DMA_REQUEST, TRUE);

/* enable tmr1 */
tmr_counter_enable(TMR1, TRUE);
while(1)
{
}
}
```

8.4 实验效果

当 TMR 产生 overflow 事件并产生 DMA 请求，但此时 DMA 请求不会被响应，只有通过板载 user 按键，每按一次，会同步 4 个 TMR 的 DMA 请求并将数据传输到 dst_buffer 数组，当数据传输 16 笔时会产生 DMA 完成中断。

9 文档版本历史

表 3. 文档版本历史

日期	版本	变更
2021.10.29	2.0.0	最初版本
2023.03.27	2.0.1	增加案例

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损失的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独立负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。

© 2023 雅特力科技 保留所有权利