AN0123
入门指南

AT32F425 DMA使用指南

前言

AT32F425 拥有 1 个 DMA 控制器（DMA1），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列解析。

支持型号列表：

支持型号	AT32F425 系列

目录

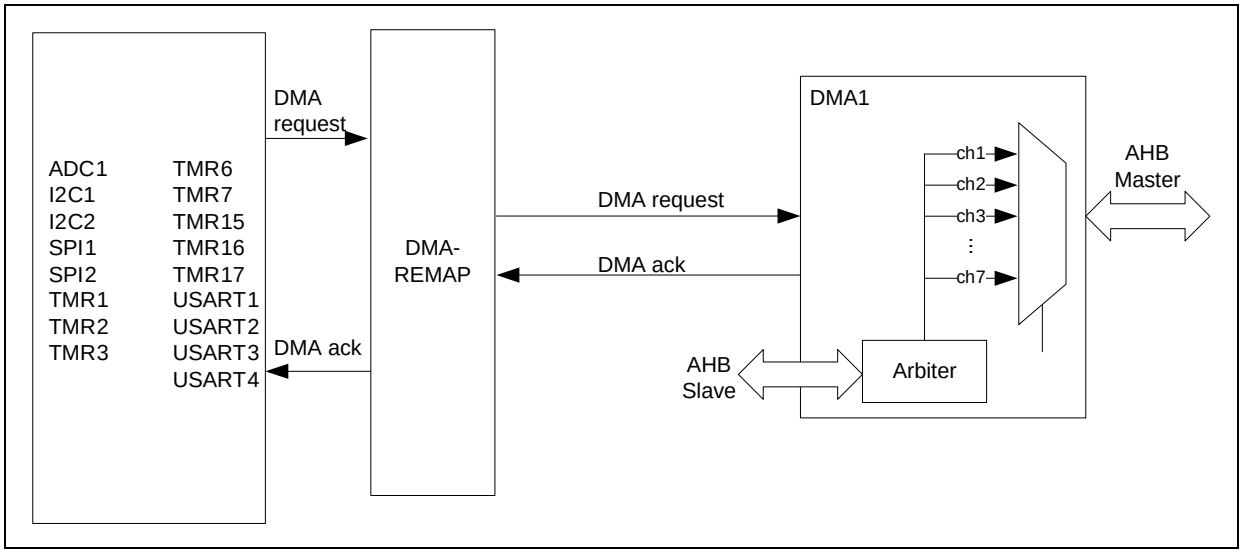
表目录

图目录

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 DMA 请求弹性映射简介

在使用 AT43F425xx 系列 DMA 时，必须配置 DMA 请求弹性映射功能，否则 DMA 不会运作。DMA 请求弹性映射可将任意一个外设产生的 DMA 请求映射到通道 1 到通道 7 中的任意一个通道。

当设定弹性模式时（DMA_FLEX_EN = 1），每个通道的请求来源由 CHx_SRC 来设定[x=1~7]。使用例子： 假如 DMA 通道 1 指定成 I2C1_TX，通道 3 要指定成 I2C1_RX，其他不使用，则设定上必须是 DMA_FLEX_EN=1，CH1_SRC=11，CH3_SRC=10，CH[2/4/5/6/7]_SRC=0。

CHx_SRC 设定值对应请求来源见下表：

各 IP 对应 ID 号如下表：

表 1. 各 IP 对应 ID 号列表

CHx_SRC	请求来源	CHx_SRC	DMA 来源	CHx_SRC	请求来源	CHx_SRC	请求来源
0	未选定	16	SPI1/I2S1_RX	32	TMR3_CH1	48	-
1	-	17	SPI1/I2S1_TX	33	TMR3_CH2	49	TMR17_OVERFLOW
2	-	18	SPI2/I2S2_RX	34	TMR3_CH3	50	USART1_RX
3	-	19	SPI2/I2S2_TX	35	TMR3_CH4	51	USART1_TX
4	-	20	TMR1_CH1	36	TMR3_TRIG	52	USART2_RX
5	ADC1	21	TMR1_CH2	37	TMR3_OVERFLOW	53	USART2_TX
6	-	22	TMR1_CH3	38	TMR6_OVERFLOW	54	USART3_RX
7	-	23	TMR1_CH4	39	TMR7_OVERFLOW	55	USART3_TX
8	-	24	TMR1_TRIG/ TMR1_HALL	40	TMR15_CH1	56	USART4_RX
9	-	25	TMR1_ OVERFLOW	41	TMR15_CH2	57	USART4_TX
10	I2C1_RX	26	TMR2_CH1	42	TMR15_TRIG/ TMR15_HALL	58	-
11	I2C1_TX	27	TMR2_CH2	43	TMR15_ OVERFLOW	59	-
12	I2C2_RX	28	TMR2_CH3	44	TMR16_CH1	60	SPI3/I2S3_RX
13	I2C2_TX	29	TMR2_CH4	45	-	61	SPI3/I2S3_TX
14	-	30	TMR2_TRIG	46	TMR16_ OVERFLOW		-
15	-	31	TMR2_ OVERFLOW	47	TMR17_CH1		-

注：表格中“CHx_SRC”为 ID 号；“请求来源”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

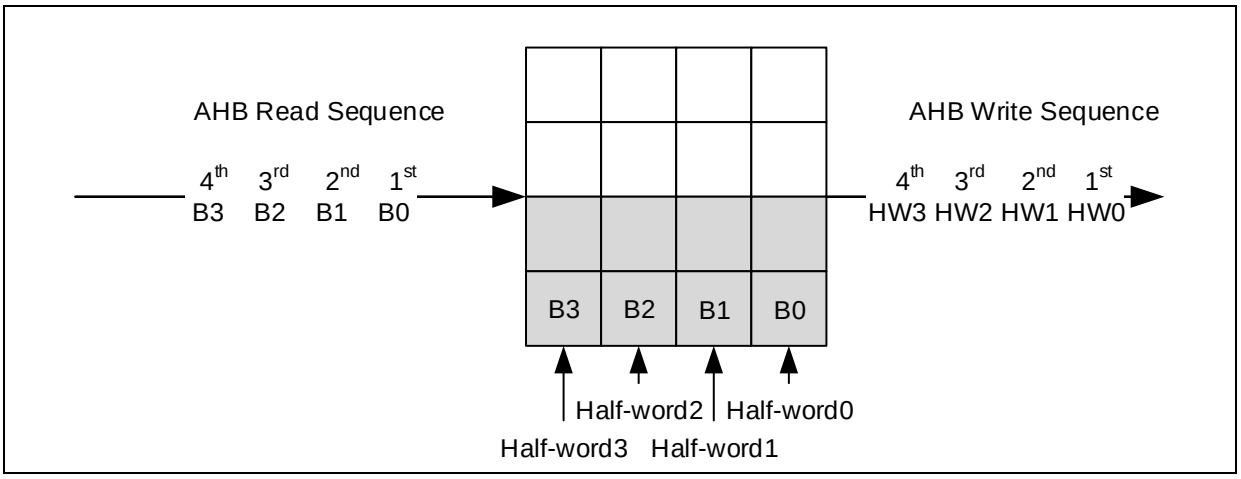
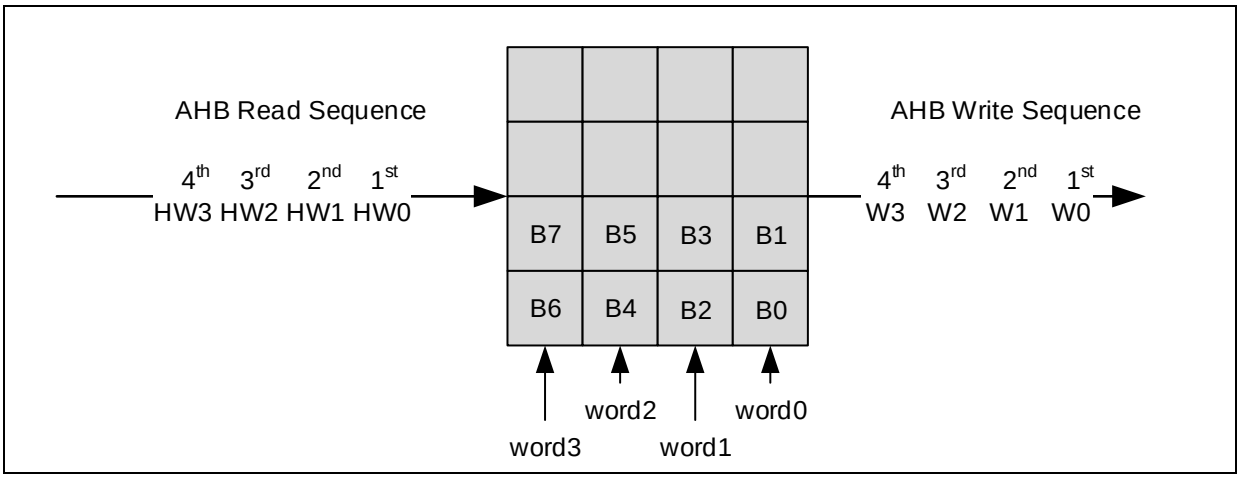


图 3. PWIDTH： half-word, MWIDTH： word



3.2 配置 DMA 弹性映射

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

配置 DMA 弹性映射比较简单，只需调用专门提供的接口函数即可：

```
/* 配置 DMA 通道弹性映射 */
void dma_flexible_config(dma_type* dma_x, uint8_t flex_channelx, dma_flexible_request_type flexible_request);
```

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

/* 复位通道 */ void dma_reset(dma_channel_type *dmax_channely);
/* 初始化 DMA 结构体参数 */ void dma_default_para_init(dma_init_type *dma_init_struct);
/* 初始化通道 */ void dma_init(dma_channel_type *dmax_channely, dma_init_type *dma_init_struct);
/* 使能通道 */ void dma_channel_enable(dma_channel_type *dmax_channely, confirm_state new_state);
/* 配置 DMA 通道弹性映射 */ void dma_flexible_config(dma_type* dma_x, uint8_t flex_channelx, dma_flexible_request_type flexible_request);

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，通道 1 优先于通道 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（PINCM/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。

- **配置 DMA 弹性映射（DMA_SRC_SELx 寄存器的 CHx_SRC）**

  在非存储器到存储器（M2M）模式下时，需要将外设的 DMA 请求 DMA 请求号写入，才能启动通道响应外设的 DMA 请求。

- **打开通道（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 DMA 请求映射使能函数配置弹性映射功能；
- 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 */
  system_clock_config();

  /* 板载初始化 */
  at32_board_init();
  /* 打开 DMA1 时钟 */
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);

  /* dma1 channel1 configuration */
  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);

/* enable transfer full data intterrupt */
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);

/* transfer_status = passed, if the transmitted and received data are the same
     transfer_status = failed, if the transmitted and received data are different */
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 GPIOB，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\data_to_gpio

6.3 软件设计

1) 配置流程

- 开启 DMA/TMR2/GPIOB 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 DMA 映射功能
- 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();

  /* enable dma1/gpiob/tmr1 clock */
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

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

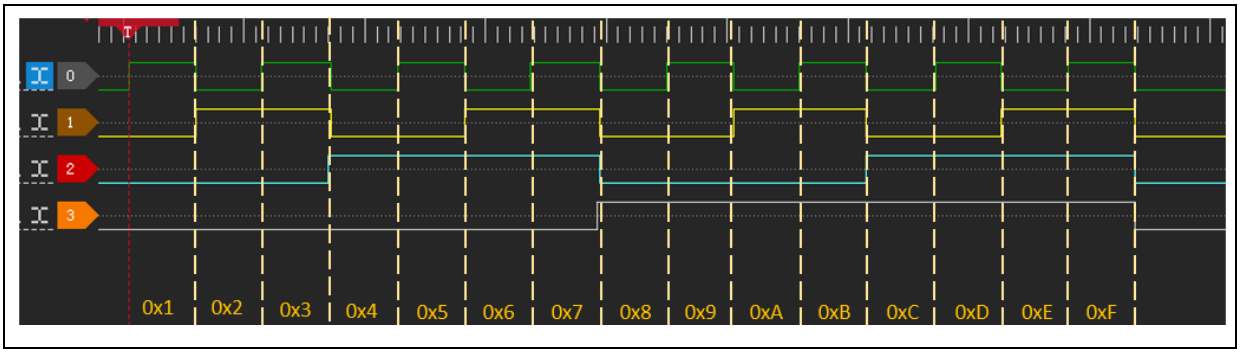
```
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init(GPIOB, &gpio_init_struct);

  tmr_base_init(TMR1, 0xFF, 0);
  tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);

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

  /* dma1 channel1 configuration */
  dma_reset(DMA1_CHANNEL5);
  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)&GPIOB->odt;
  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(DMA1_CHANNEL5, &dma_init_struct);

  /* enable transfer full data intterrupt */
  dma_interrupt_enable(DMA1_CHANNEL5, DMA_FDT_INT, TRUE);

  /* config dma1 flexible mode */
  dma_flexible_config(DMA1, FLEX_CHANNEL5, DMA_FLEXIBLE_TMR1_OVERFLOW);

  /* dma1 channel1 interrupt nvic init */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  nvic_irq_enable(DMA1_Channel7_4_IRQn, 1, 0);

  /* enable dma channel */
  dma_channel_enable(DMA1_CHANNEL5, TRUE);

  /* enable tmr1 */
  tmr_counter_enable(TMR1, 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 文档版本历史

表 3. 文档版本历史

日期	版本	变更
2022.1.29	2.0.0	最初版本

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、死亡及财产损害的应用。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，且采购商应独力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。