AN0123
Application Note

AT32F425 DMA Application Note

Introduction

The AT32F425 series MCU features one DMA controller (DMA1). Each controller contains seven DMA channels, and DMA requests from peripherals can be mapped to any channel. This application note mainly introduces DMA functions and related application cases.

Applicable products:

Part number	AT32F425xx

Contents

List of Tables

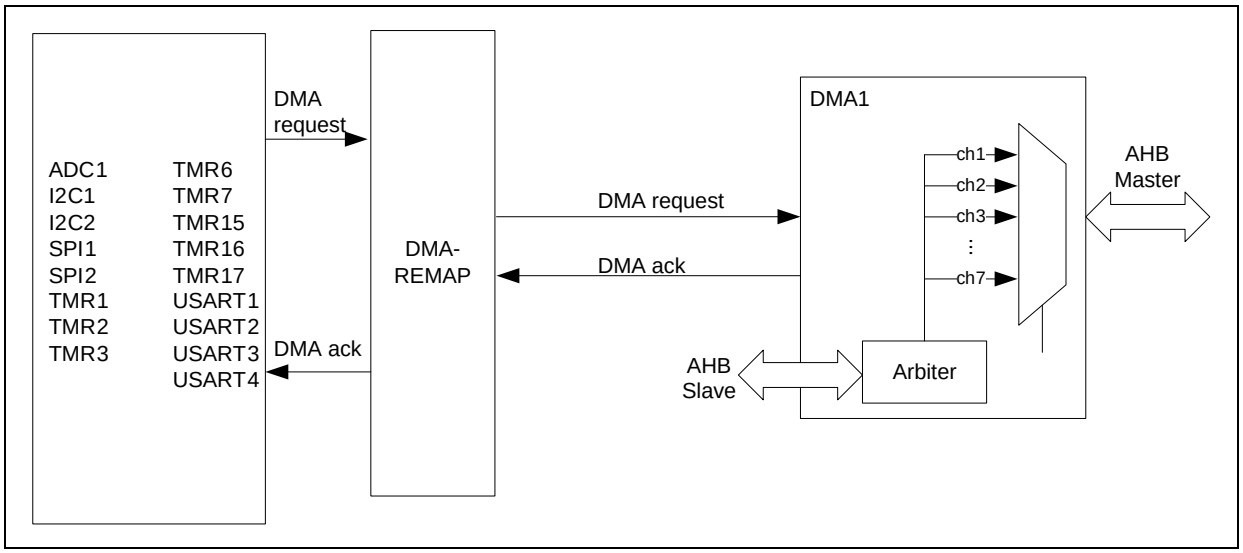
List of Figures

1 Introduction

Direct memory access (DMA) controller is designed for 32-bit MCU applications with the aim of enhancing system performance and reducing the generation of interrupts.

DMA controller provides seven channels for memory-to-memory, memory-to-peripheral and peripheral-to-memory transfers, and for flexible mapping of DMA requests from peripherals.

Figure 1. DMA block diagram



2 Flexible DMA request mapping

The DMA request flexible mapping feature must be configured before using the DMA controller in AT43F425xx series MCUs. This feature supports DMA requests from any peripheral to be mapped to any of the seven channels.

In flexible request mode (DMA_FLEX_EN = 1), the request source for each channel is selected through the CHx_SRC register (x=1~7). For example, to configure the DMA channel 1 as I2C1_TX, and channel 3 as I2C1_RX, others unused, then DMA_FLEX_EN=1, CH1_SRC=11, CH3_SRC=10, CH[2/4/5/6/7]_SRC=0 must be asserted.

The table below lists the DMA flexible request sources.

Table 1. DMA flexible request sources

CHx_SRC	Request source	CHx_SRC	DMA source	CHx_SRC	Request source	CHx_SRC	Request source
0	Unselected	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		-

Note: "CHx_SRC" is the ID number, and "request source" is the DMA request of each IP.

3 DMA function overview

3.1 Programmable data transfer width

Channels in DMA controller supports data transfer (byte/half-word/word). Transfer width of the source data and destination data is programmable through the PWIDTH and MWIDTH bits in the DMA_CxCTRL register. Generally, PWIDTH bit is set equal to MWIDTH bit. When PWIDTH is not equal to MWIDTH, it can be aligned according to the settings of PWIDTH/ MWIDTH.

Figure 2. PWIDTH: byte, MWIDTH: half-word

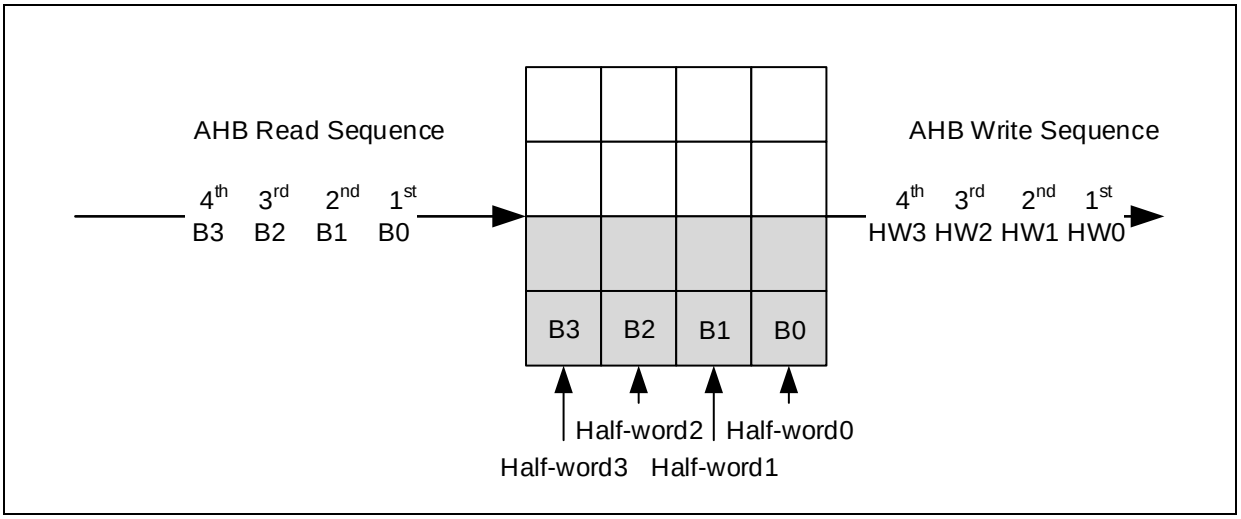
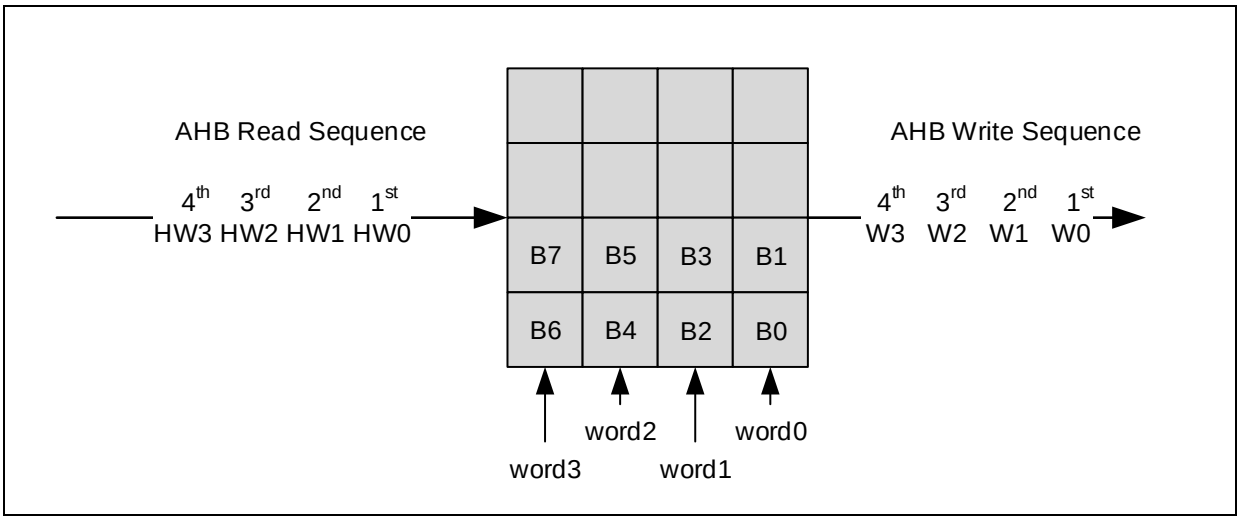


Figure 3. PWIDTH: half-word, MWIDTH: word



3.2 Configure flexible DMA request mapping

In M2P and P2M mode, configure flexible mapping mode; otherwise, DMA controller will not response to DMA requests from peripherals. DMA flexible mapping is to multiplex channels for DMA requests from peripherals, that is, the DMA request of any peripheral can be mapped to any channel of DMA1, which increases the flexibility of DMA channel allocation.

Configure the DMA flexible mapping by calling a dedicated interface function:

```
/* Configure DMA channel flexible mapping */
void dma_flexible_config(dma_type* dma_x, uint8_t flex_channelx, dma_flexible_request_type flexible_request);
```

4 DMA configurations

DMA configuration interfaces and process are detailed below.

4.1 Function interface

Table 2. Channel configuration functions

/* reset channel */ void dma_reset(dma_channel_type *dmax_channely);
/* Initialize DMA structure parameter */ void dma_default_para_init(dma_init_type *dma_init_struct);
/* Initialize channel */ void dma_init(dma_channel_type *dmax_channely, dma_init_type *dma_init_struct);
/* Enable channel */ void dma_channel_enable(dma_channel_type *dmax_channely, confirm_state new_state);
/* Configure DMA channel flexible mapping */ void dma_flexible_config(dma_type* dma_x, uint8_t flex_channelx, dma_flexible_request_type flexible_request);

4.2 Channel settings configuration

- **Set the peripheral address in the CxPADDR register**

  The initial peripheral address for data transfer remains unchanged during transmission.

- **Set the memory address in the CxMADDR register**

  The initial memory address for data transfer remains unchanged during transmission.

- **Configure the amount of data to be transferred in the CxDTCNT register**

  Programmable data transfer size is up to 65535. This value is decremented after each data transfer.

- **Configure channel settings in the CxCTRL register**

  Include the channel priority, data transfer direction/width, address incremented mode, circular mode and interrupt mode.

  - **Channel priority (CHPL)**

    There are four levels, including very high priority, high priority, medium priority and low priority.

    If the two channels have the same priority level, then the channel with lower number will get priority over the one with higher number. For example, channel 1 has priority over channel 2

  - **Data transfer direction (DTD)**

    Include memory-to-peripheral (M2P), peripheral-to-memory (P2M) and memory-to-memory (M2M) transfer.

    Circular mode, double-buffer mode or direct mode are not allowed in the memory-to-memory transfer mode.

  - **Data transfer width (PWIDTH/ MWIDTH)**

    The data transfer width is configurable in terms of byte, half-word and word, as required.

- **Address incremented mode (PINCM/MINCM)**

  In incremented mode, the subsequent transfer address is the previous address plus transfer width (PWIDTH/MWIDTH).

- **Circular mode (LM)**

  In circular mode, the content in the CxDTCNT register is automatically reloaded with the initially programmed value after the completion of the last transfer.

- **Configure DMA flexible mapping through CHx_SRC in the DMA_SRC_SELx register**

  In non-M2M mode, write the DMA request number to enable channels to respond to DMA requests from peripherals.

- **Enable channels by setting the CHEN bit in the CxCTRL register**

4.3 Configuration process

- Enable DMA clock;
- Call the channel reset function to reset data flow;
- Call the structure initialization function to initialize the channel configuration structure;
- Call the initialization function to initialize channels;
- Call the DMA request mapping enable function to configure the flexible mapping function;
- Call the channel enable function to enable channels.

5 Case 1: Data transfer from FLASH to SRAM

5.1 Function overview

Implement data transfer from on-chip FLASH to internal SRAM by using DMA controller.

5.2 Resources preparation

1) Hardware environment

   AT-START BOARD of the corresponding product model

2) Software environment

   project\at_start_f4xx\examples\dma\flash_to_sram

5.3 Software design

1) Configuration process

   - Enable DMA peripheral clock
   - Configure DMA channels
   - Enable transfer complete interrupt
   - Enable channels
   - Wait for the completion of data transfer
   - Compare and verity that the data transfer is correct

2) Code

   - Main function code

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

  /* Board initialization */
  at32_board_init();
  /* Enable DMA1 clock */
  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 Test result

- If data transfer is correct, LED2/3/4 will be ON.

6 Case 2: TMR generates a DMA request to transfer data from SRAM to GPIO

6.1 Function overview

This application case introduces that TMR generates a DMA request to transfer data from SRAM to GPIOB. Users can use logic analyzer to check the waveform.

6.2 Resources preparation

1) Hardware environment

   AT-START BOARD of the corresponding product model

2) Software environment

   project\at_start_f4xx\examples\dma\data_to_gpio

6.3 Software design

1) Configuration process

   - Enable DMA/TMR2/GPIOB peripheral clock
   - Configure DMA channels
   - Configure TMR2 to enable overflow DMA request
   - Enable transfer complete interrupt
   - Enable DMA mapping
   - Enable channels
   - Enable TMR2, so that the overflow interrupt generates a DMA request

2) Code

   - main function code

```
#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)
{
  /* Initialize system clock */
  system_clock_config();
  /* Board initialization */
  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;
```

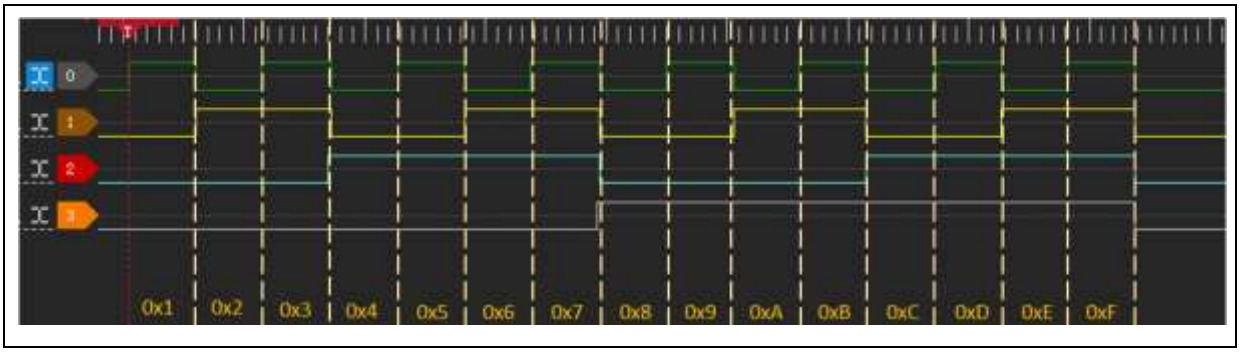
```
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init(GPIOB, &gpio_init_struct);

  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 Test result

- Capture the GPIOC waveform to view related data. For example, capture the data of GPIO0-GPIO3.

Use a logic analyzer to capture the output waveform of GPIO0-GPIO3, and view the data in buffers that are defined for the program implementation, as shown below.

Figure 4. Data to GPIO transfer test result



7 Revision history

Table 3. Document revision history

Date	Version	Revision note
2022.1.29	2.0.0	Initial release

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services; ARTERY assumes no liability for purchasers' selection or use of the products and the relevant services.

No license, express or implied, to any intellectual property right is granted by ARTERY herein regardless of the existence of any previous representation in any forms. If any part of this document involves third party's products or services, it does NOT imply that ARTERY authorizes the use of the third party's products or services, or permits any of the intellectual property, or guarantees any uses of the third party's products or services or intellectual property in any way.

Except as provided in ARTERY's terms and conditions of sale for such products, ARTERY disclaims any express or implied warranty, relating to use and/or sale of the products, including but not restricted to liability or warranties relating to merchantability, fitness for a particular purpose (based on the corresponding legal situation in any unjudicial districts), or infringement of any patent, copyright, or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Auto-motive application or environment; (D) Aerospace applications or environment, and/or (E) weapons. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.