AN0090
入门指南

AT32F435/437 EDMA使用指南

前言

AT32 的 EDMA 控制器功能比较强大，支持 8 个数据流通道且外设的 DMA 请求可映射到任意数据流上、数据的打包与拆包、FIFO 开启与关闭、burst 数据传输模式、存储器端的双 buffer 模式、可配置数据链传输、二维传输。本文主要就 EDMA 的基本功能进行讲解和案列解析。

支持型号列表：

支持型号	AT32F435xx
	AT32F437xx

目录

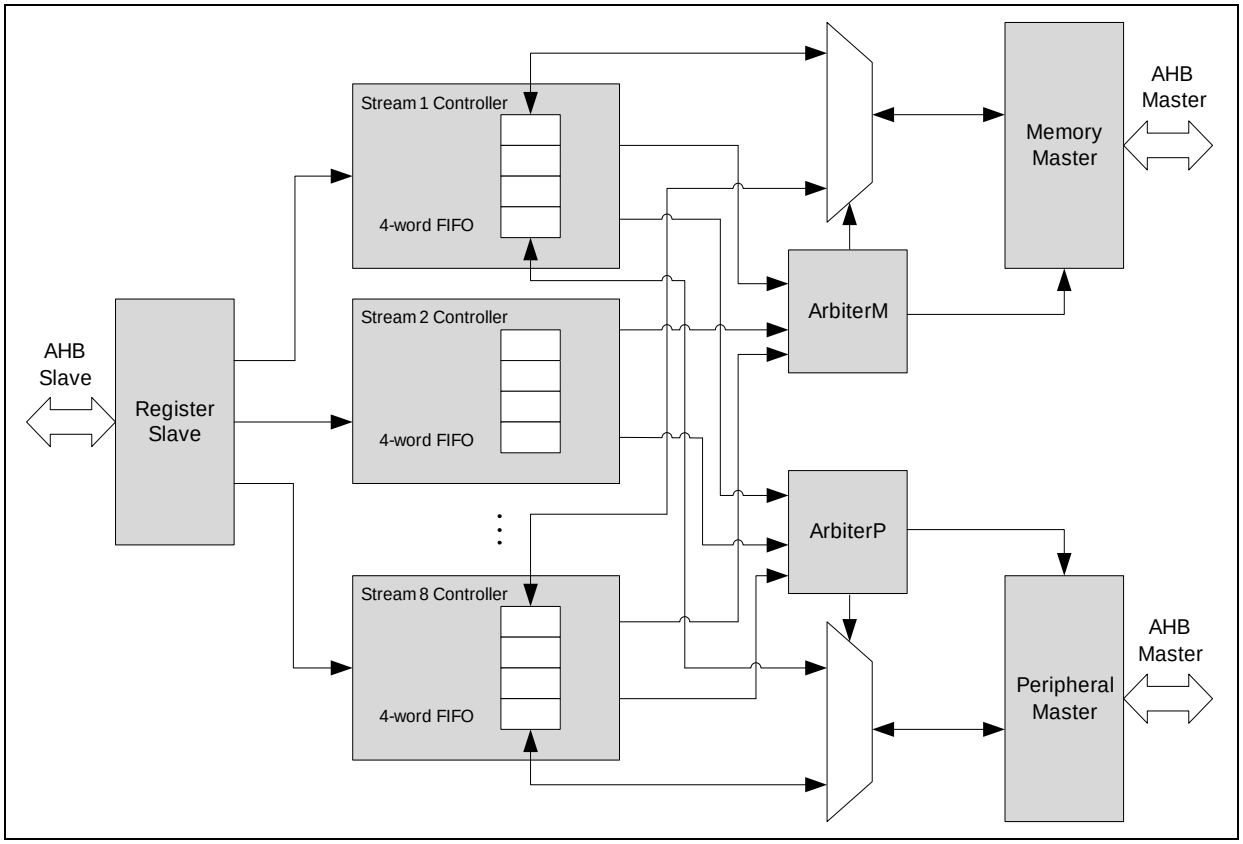
表目录

图目录

1 EDMA 简介

EDMA 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。EDMA 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八个数据流通道。每个通道都支持外设的 DMA 请求映射到任意数据流上、数据的打包与拆包、FIFO 开启与关闭、burst 数据传输模式、存储器端的双 buffer 模式、可配置数据链传输、二维传输功能。基于复杂的总线矩阵架构，将功能强大的双 AHB 总线架构与独立的 FIFO 结合在一起，优化了系统的带宽。

图 1. EDMA 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 EDMA 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 8 个数据流通道的使用情况，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

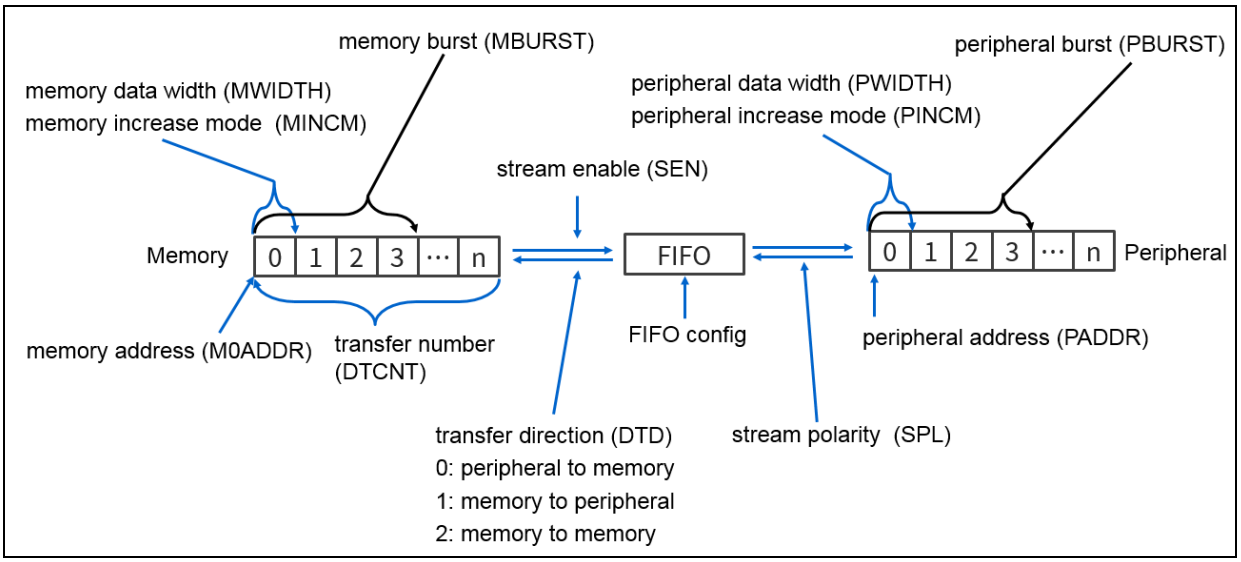
DMAMUX 请求	来源	DMAMUX 请求	来源	DMAMUX 请求	来源	DMAMUX 请求	来源
31	UART4_TX	63	TMR3_CH3	95	reserved	127	reserved
32	UART5_RX	64	TMR3_CH4	96	reserved		

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

3 EDMA 功能解析

3.1 基本配置

图 2. EDMA 基本参数配置



EDMA 传输所需要的基本参数配置如上图所示，列举如下

内存端口

- 地址（M0ADDR）
- 数据宽度（MWIDTH）：8 位、16 位、32 位
- 地址地址模式（MINCM）：固定、递增
- 突发传输（MBURST）：单次（1）、4、8、16 节拍

外设端口

- 地址（PADDR）
- 数据宽度（PWIDTH）：8 位、16 位、32 位
- 地址地址模式（PINCM）：固定、递增
- 突发传输（PBURST）：单次（1）、4、8、16 节拍

其他

- 传输方向（DTD）：外设到内存、内存到外设、内存到内存
- 通道优先级（SPL）：低、中、高、非常高
- 数据传输个数（DTCNT）：范围 1~65535
- FIFO 配置
- 流使能（SEN）

3.2 FIFO 功能

3.2.1 功能介绍

EDMA 控制器每个数据流都拥有独立的 4 word fifo，这使得 EDMA 传输变得更加灵活，所传输的数据在 FIFO 内可进行打包与拆包操作，不再限制源与目标的数据总线宽度必须相等。

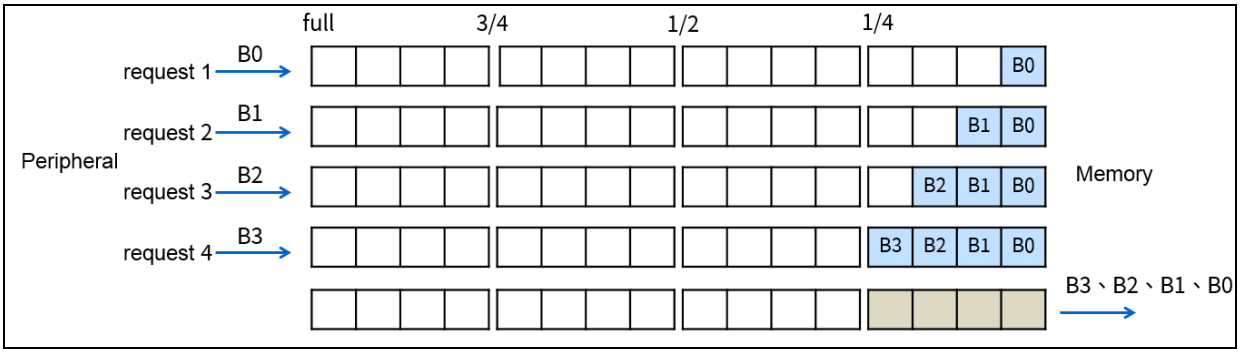
在 FIFO 模式下，源和目标数据宽度可以通过 SxCTRL 寄存器 PWIDTH 与 MWIDTH 位配置（byte，halfword，word），且允许 PWIDTH 与 MWIDTH 不同。

当 PWIDTH 与 MWIDTH 不同时：

- EDMA 要传输的数据宽度等于 PWIDTH，例如 PWIDTH = halfword，则需传输的数据量为 SxDTCNT*2。
- EDMA 控制器仅按照小字节序寻址源和目标。

FIFO 模式下外设到内存传输逻辑

图 3. FIFO 模式外设到内存传输

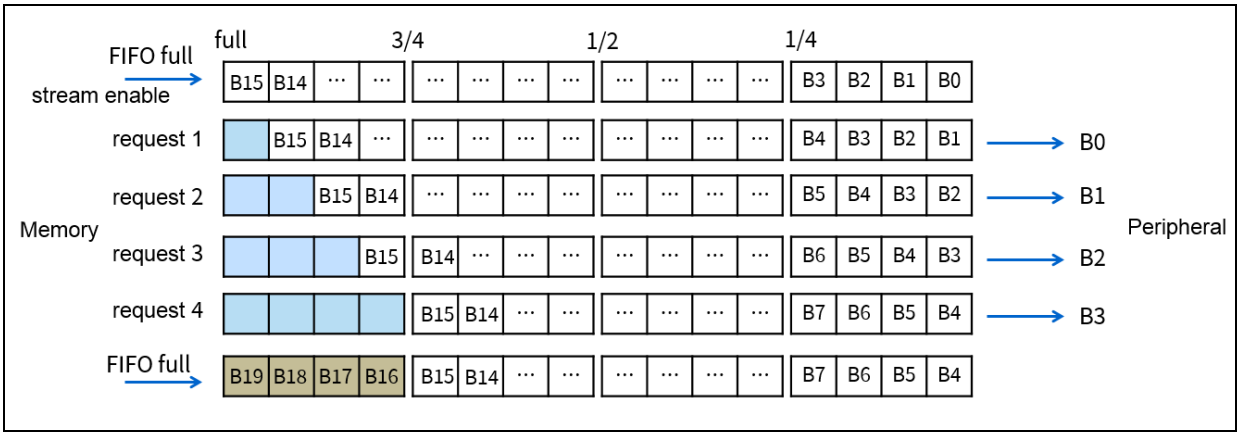


如上图所示，例如 FIFO 阈值设置成 1/4，外设和内存端数据都为字节。

刚开始 FIFO 状态为空，外设传输 4 笔数据后，FIFO 状态为 1/4，此时达到 FIFO 阈值，一次传输 4 个字节到内存。

FIFO 模式下内存到外设传输逻辑

图 4. FIFO 模式内存到外设传输



如上图所示，例如 FIFO 阈值设置成 1/4，外设和内存端数据都为字节。

刚开始 FIFO 状态为空，当使能数据流后，立即从内存传输 16 个字节到 FIFO，此时 FIFO 状态为满，然后外设传输 4 笔数据后，FIFO 状态为 3/4，此时达到 FIFO 阈值，一次从内存传输 4 个字节到 FIFO，再次填满 FIFO。

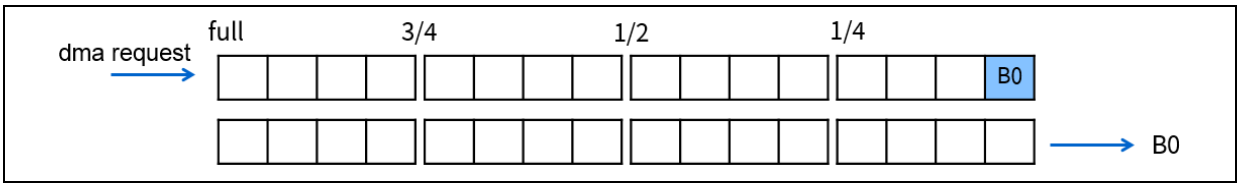
FIFO 模式下内存到内存传输逻辑

当使能数据流后，开始从源内存传输数据到 FIFO，当达到 FIFO 阈值时，FIFO 里的数据将被全部传输到目标内存，然后重复此步骤直到传输完成。

在直接模式下（SxFCTRL 内 FEN = 0），不可进行数据的打包与拆包。这种情况下，不允许源与目标的数据宽度不相等。数据宽度由 PWIDTH 定义，MWIDTH 的设定无效。

直接模式下外设到内存传输逻辑

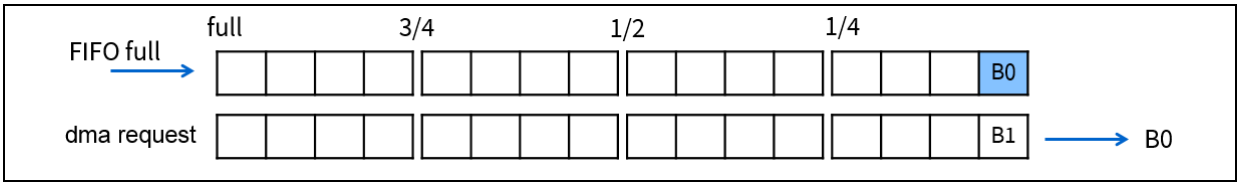
图 5. 直接模式外设到内存传输



当产生 DMA 请求后，数据从外设传输到 FIFO，然后 FIFO 内的数据将会立即传输到内存。

直接模式下内存到外设传输逻辑

图 6. 直接模式内存到外设传输



当使能数据流后，立即从内存传输一个数据到 FIFO（数据大小由 PWIDTH 决定），当产生 DMA 请求后，立即将数据从 FIFO 传输到外设，然后再次从内存传输一个数据到 FIFO。

数据打包与拆包示意图如下：

图 7. EDMA 数据打包示意图

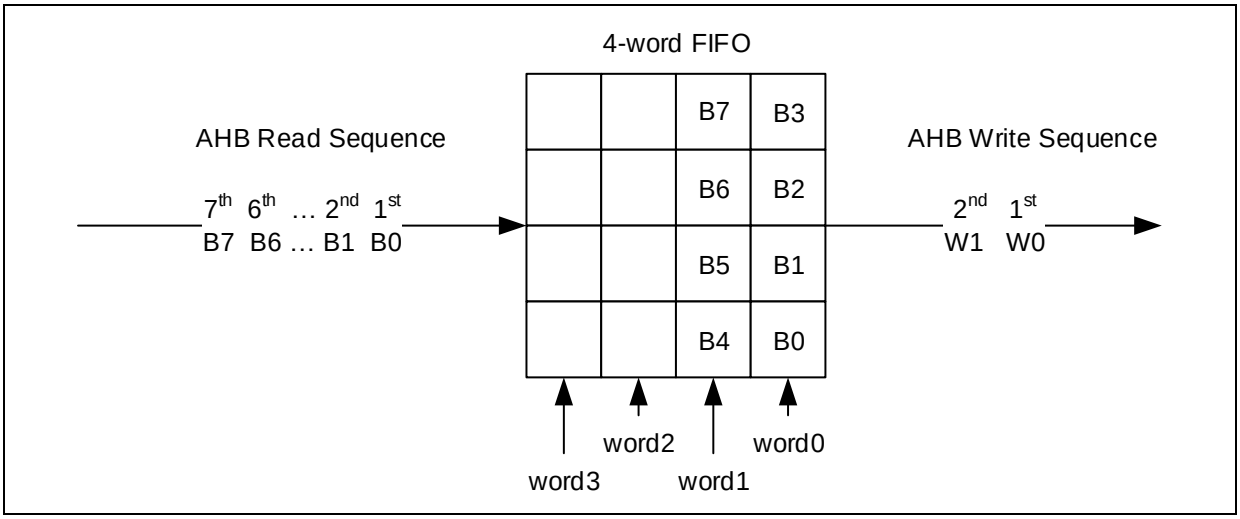
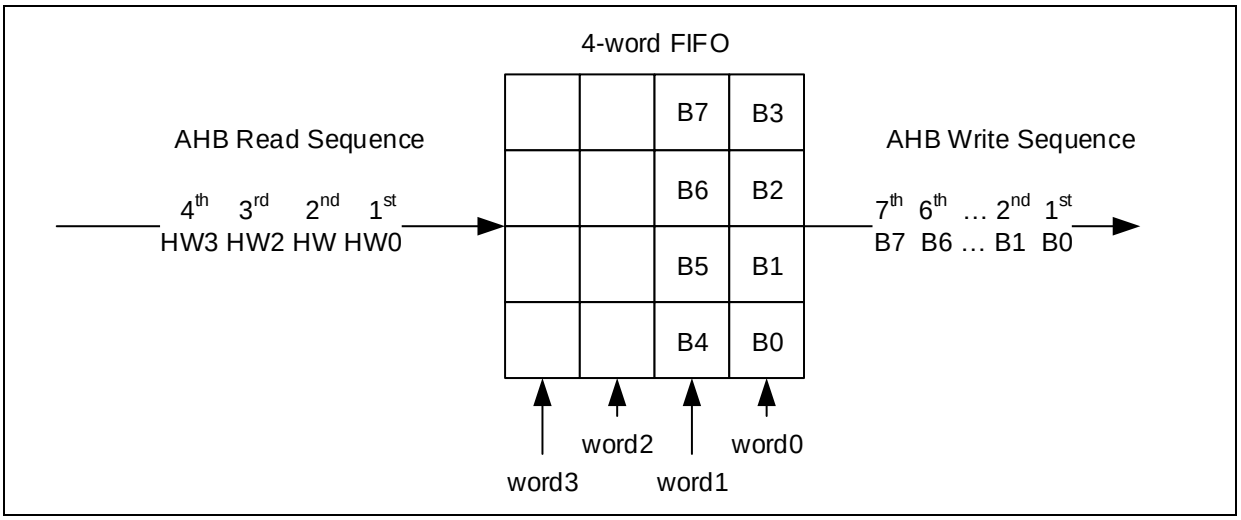


图 8. EDMA 数据拆包示意图



外设端口可以是源或者目标（在 Memory to memory 模式下，也可以是存储器源），在使用 FIFO 时，一定要合理配置 PWIDTH、MWIDTH 和 SxDTCNT，已确保数据传输的完整性，在 PWIDTH 小于 MWIDTH 时，需要按照如下条件配置：

表 2. PWIDTH 和 MWIDTH 与 SxDTCNT 的限制条件

PWIDTH	MWIDTH	SxDTCNT
Byte	Halfword	必须是 2 的倍数
Byte	Word	必须是 4 的倍数
Halfword	Word	必须是 2 的倍数

FIFO 可配置阈值级别，软件可配置为 1/4、1/2、3/4、满这四种状态（通过 SxFCTRL 寄存器 FTHSEL 配置），并且可通过 SxFCTRL 寄存器的 FSTS 位观察 FIFO 当前状态。

3.2.2 软件接口

设置 FIFO 的软件接口包含在 EDMA 配置结构体内，以结构体成员的方式呈现，如下：

```
edma_init_struct.fifo_mode_enable = FALSE;
edma_init_struct.fifo_threshold = EDMA_FIFO_THRESHOLD_1QUARTER;
```

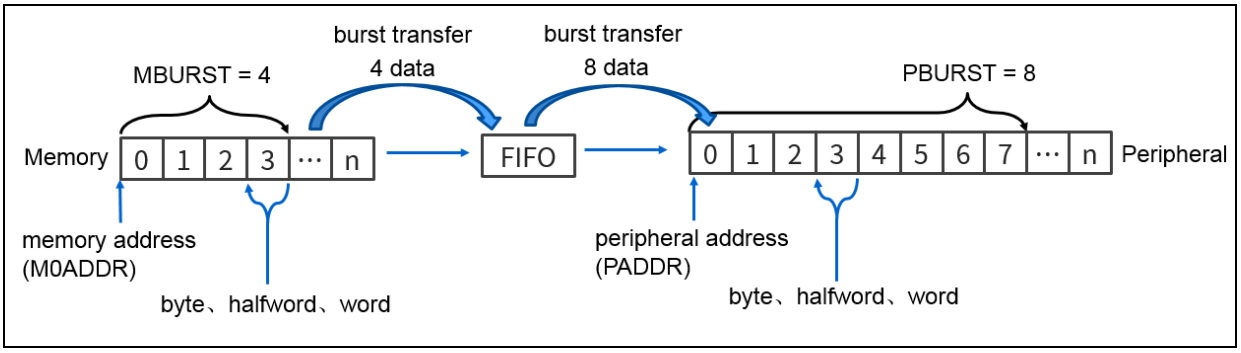
第一项为配置 FIFO 是否使能，当为 TURE 时，表示 FIFO 使能；当为 FALSE 时，表示 FIFO 禁止

第二项为 FIFO 阈值设定，其选项可以是 1/4、1/2、3/4 和满这四种状态。

3.3 突发传输功能

3.3.1 功能介绍

图 9. 突发传输示意图



EDMA 控制器可产生单次传输或 4 个、8 个、16 个节拍的突发传输。突发传输数量通过 SxCTRL 寄存器的 PBURST 与 MBURST 位设定。

如上图所示，传输方向为内存到外设，内存端 MBURST = 4，每一次数据传输，DMA 都会从内存连续传输 4 个数据到 FIFO（数据可以为字节、半字、字）。外设端 PBURST = 8，每一次数据传输，DMA 都会从 FIFO 连续传输 8 个数据到外设（数据可以为字节、半字、字）。

突发传输数量是按照节拍数来计算的，并非按照字节数计数，例如 PBURST=4，PWIDTH=halfword，则一次突发传输的数据量为 4*2 bytes。

突发传输只能在 FIFO 模式下才可使用；在直接模式下，数据流只能生成单次传输，而 MBURST 和 PBURST 位由硬件强制配置。

根据单次传输或突发传输的配置，每个 DMA 的请求在 AHB 外设端口上启动相应配置下的数据量传输：

- 当 AHB 外设端口配置为单次传输时，根据 PWIDTH 配置的值，每次 DMA 请求传输一个 byte、halfword 或者 word。
- 当 AHB 外设端口配置为突发传输时，根据 PWIDTH 配置的值，每次 DMA 请求传输 4 个、8 个、16 个 byte、halfword 或者 word。

注：源和目的地址自增模式下，才可使用突发传输。

EDMA 的 FIFO 阈值设定与存储器突发传输大小需要满足一定的关系才能正常启动数据流，否则在启动数据流时，就会产生 FIFO error，然后将禁止数据流。FIFO 阈值设定与存储器突发传输大小关系需满足如下：

表 3. MWIDTH 与 MBURST 配置关系

MWIDTH	FIFO 阈值	MBURST = INCR4	MBURST = INCR8	MBURST = INCR16
Byte	1/4	4 个节拍的 1 次突发	禁止	禁止
	1/2	4 个节拍的 2 次突发	8 个节拍的 1 次突发	禁止
	3/4	4 个节拍的 3 次突发	禁止	禁止
	满	4 个节拍的 4 次突发	8 个节拍的 2 次突发	16 个节拍的 1 次突发
Halfword	1/4	禁止	禁止	禁止

MWIDTH	FIFO 阈值	MBURST = INCR4	MBURST = INCR8	MBURST = INCR16
	1/2	4 个节拍的 1 次突发	禁止	禁止
	3/4	禁止	禁止	禁止
	满	4 个节拍的 2 次突发	8 个节拍的 1 次突发	禁止
Word	1/4	禁止	禁止	禁止
	1/2	禁止	禁止	禁止
	3/4	禁止	禁止	禁止
	满	4 个节拍的 1 次突发	禁止	禁止

从此表个可以得出结论：突发传输数据量与数据大小乘积不能超过 FIFO 阈值大小。

3.3.2 软件接口

设置突发传输软件接口与设置 FIFO 类似，其被包含在 EDMA 配置结构体内，以结构体成员的方式呈现：

```
edma_init_struct.peripheral_burst_mode = EDMA_PERIPHERAL_BURST_4;
edma_init_struct.memory_burst_mode = EDMA_PERIPHERAL_BURST_4;
```

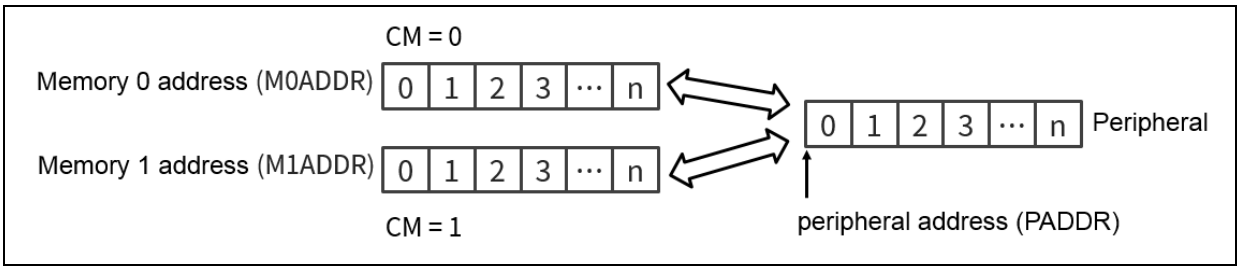
第一项为配置外设端口突发传输量；第二项为配置内存端口突发传输量。

注：在使用突发传输时，必须使能 FIFO 模式。

3.4 存储器端双缓存区功能

3.4.1 功能介绍

图 10. 双缓存区示意图



通过设置 SxCTRL 寄存器的 DMM 位，即可使能双缓存区功能。与常规（单缓存区）数据流工作模式相比，双缓存区功能拥有两个存储器指针；启动双缓存区功能时，EDMA 控制器的相应数据流通道会自动开启循环模式，在每次 SxDTCNT 减到 0 时自动交换存储区。

根据 SxCTRL 寄存器的 CM 位，可知道数据流当前使用的存储区是 memory0/1。当 CM=0，表示当前目标是 memory0；当 CM=1，表示当前目标为 memory1。

在使用双缓存区，需要注意以下几点：

- 当硬件正在使用某一块缓存时，当前缓存的基地址不可改变，即寄存器 SxM0ADDR 和

SxM1ADDR。例如数据流正在将数据填入 SxM0ADDR 时，那么软件只能改变 SxM1ADDR 的基地址。可根据 CM 位的状态，确定数据流当前操作的缓存区。

- 当使用双缓存区功能时，禁止使用存储器到存储器模式。

3.4.2 软件接口

对于双缓存区功能，软件实现了单独的函数接口，如下：

```
/* 配置双缓存区功能 */
void edma_double_buffer_mode_init(edma_stream_type *edma_streamx, uint32_t memory1_addr,
edma_memory_type current_memory);
/* 使能双缓存区功能 */
void edma_double_buffer_mode_enable(edma_stream_type *edma_streamx, confirm_state new_state);
```

对于双缓存区功能，第一个缓存区地址会在 EDMA 初始化结构体中配置，即 SxM0ADDR 寄存器的配置。

在配置双缓存区功能时，调用 edma_double_buffer_mode_init();函数，此函数第一个参数表示当前配置双缓存区的数据流，第二个参数为第二个缓存区的地址，第三个参数为启动后硬件使用的缓存区。

以上函数的调用，双缓存区功能就配置好了，现在只需要调用 edma_double_buffer_mode_enable();函数开启双缓存区功能即可。

3.5 链接列表传输功能

3.5.1 功能介绍

链接列表传输可实现将几块不连续的存储区的数据使用同一个数据流按照一定顺序发送出去或者同一数据流接收到数据按照一定顺序存储在几块不连续的存储区内。

EDMA 控制器如何知道数据存储规则的呢？此规则是由软件按照特定格式给出的，其格式如下图所示：

图 11. 链接描述符格式

		bit offset																													
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CTRL & CNT	DMM	PINCOS	SPL		MBURST		PBURST		MWIDTH		PWIDTH		MINCM	PINCM	DTD		DTCNT[15:0]															
+4	PADDR	PADDR[31:0]																													
+8	M0ADDR	M0ADDR[31:0]																													
+C	LLP	LLP[31:0]																													

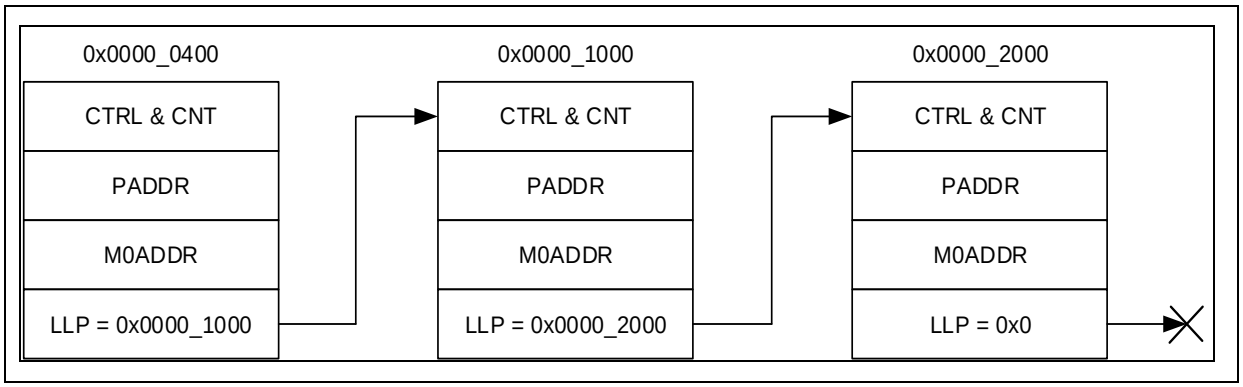
将此格式成为链接描述符，此描述符存储在片上存储器中，当打开数据流时，EDMA 控制器从片上存储器加载此描述符配置数据流，然后开启数据的传输。

描述符分为 4 项：

- 数据流基础配置和数据传输数量配置
- 外设基地址
- 存储器基地址
- 下一个描述符的首地址

通过描述符的具体实现，最终实现一个单向链表结构，如下：

图 12. 链接描述符链表结构



注：描述符在片上存储地址必须要 4 word 对齐，否则不能正常启动数据流。

3.5.2 软件接口

对于链接列表传输功能，软件单独实现了函数接口，如下：

```
/* 链接列表传输功能配置 */
void edma_link_list_init(edma_stream_link_list_type *edma_streamx_ll, uint32_t pointer);
/* 链接列表传输功能开启 */
void edma_link_list_enable(edma_stream_link_list_type *edma_streamx_ll, confirm_state new_state);
```

第一个函数是对链接列表传输功能的一些配置。参数有两个，第一个参数表示所配置的数据流号，第二个参数表示链表描述符的首地址。

经过此函数的配置后，只需调用第二个函数即可开启链表传输功能。

3.6 二维处理功能

3.6.1 功能介绍

二维数据传输（2D）功能，使用者可以较为方便的对数据块进行一些操作，提高对数据的处理效率，在图像处理方面效果较为明显。

EDMA 为二维数据传输（2D）提供了 4 个可配置的参数值：

- DMA_Sx2DCNT 中的 XCOUNT：跳转到下一个跨步之前要传输的数据计数；
- DMA_Sx2DCNT 中的 YCOUTNT：迭代计数；
- DMA_Sx2DSTRIDE 中的 SRCSTD：源跨步值，该值应在源端迭代之前添加或减去；
- DMA_Sx2DSTRIDE 中的 DSTSTD：目的跨步值，该值应在目标端迭代之前添加或减去。

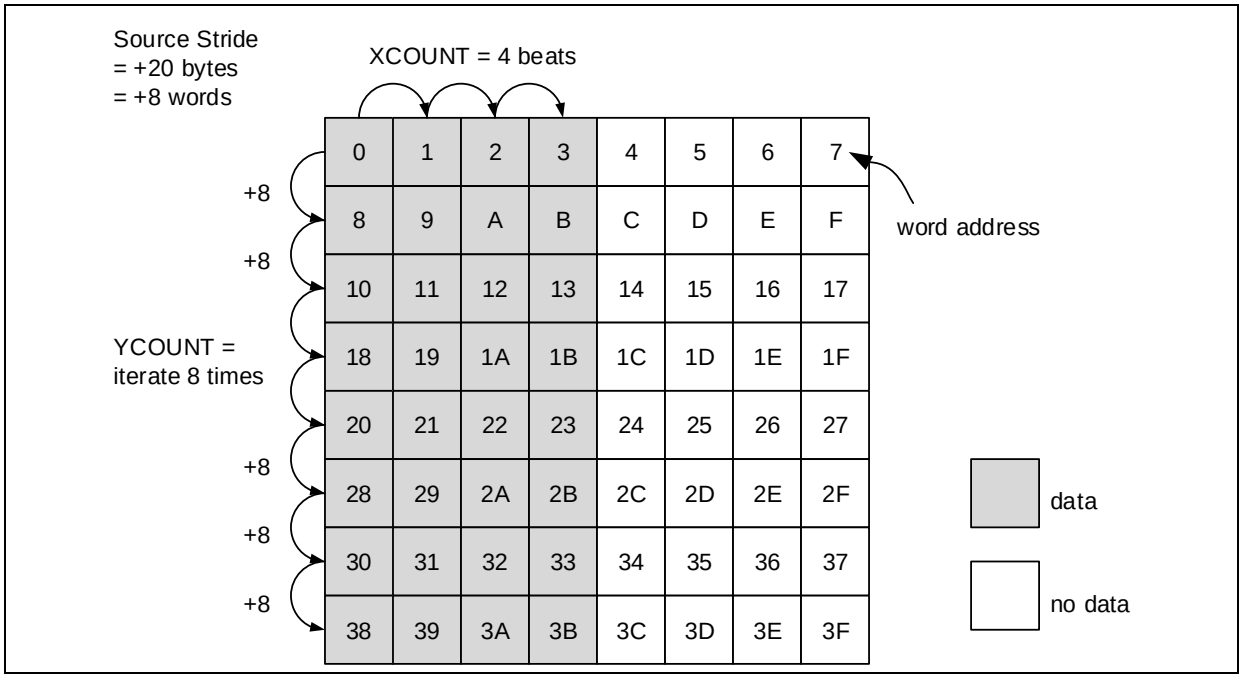
二维数据传输（2D）举例：

表 4. 2D 传输举例

源跨步（字节地址）	目的跨步（字节地址）	XCNT	YCNT
0x20	0x10	0x4	0x8

传输前数据在源存储器中：

图 13. 2D 传输前数据



传输后数据在目的存储器中：

图 14. 2D 传输后数据

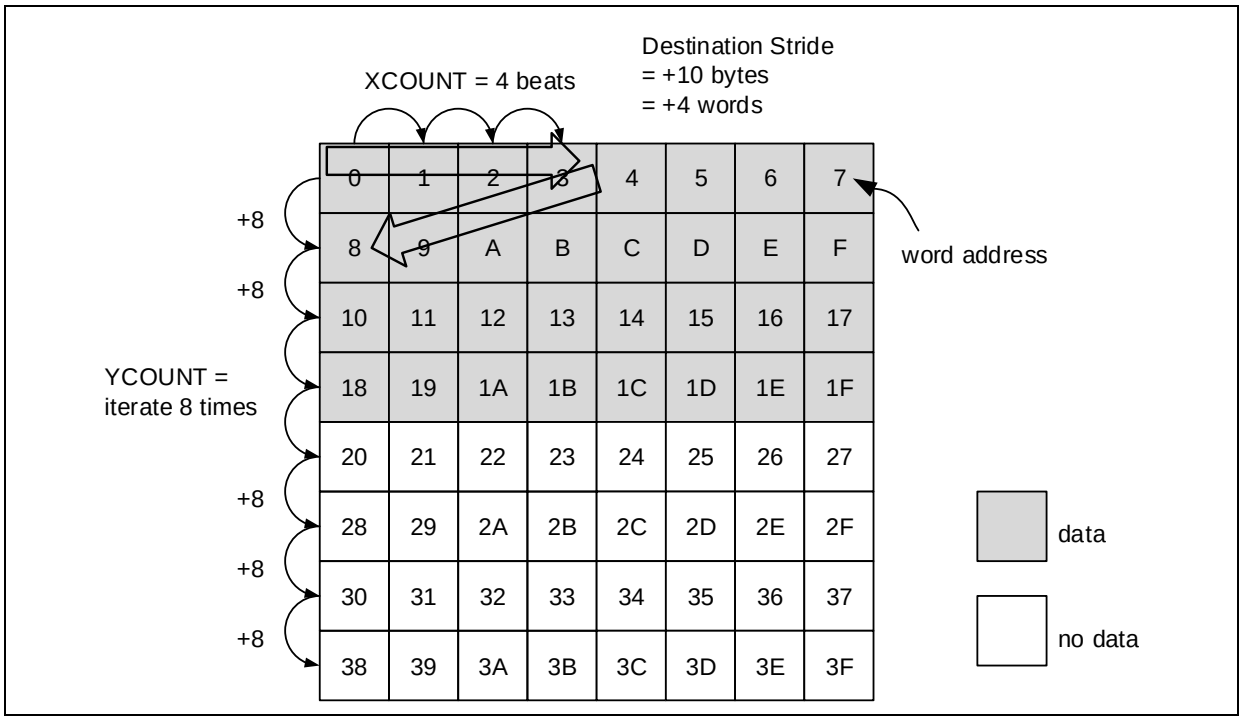


图 4 和图 5 的阴影部分为真实数据。根据图中可以清晰看到将数据看成二维的存储在存储器中，2D 传输将此数据块的左边 1/2 的数据取走并存储在新二维数据块的上 1/2 处。

在使用二维传输时，有一些限制条件需要注意：

- 仅支持 memory to memory 传输
- PWIDTH 和 MISZE 需要设置为 0x2（字）
- 源和目标跨步值需与字对齐
- 源和目标跨步值均为二补数
- PINCM 和 MINCM 均需设置为 1（增量模式）
- PFCTRL 需设置为 0
- PINCOS 需设置为 0
- 不支持循环模式、双缓冲模式
- 不支持链接列表传输

3.6.2 软件接口

对于二维数据传输功能，软件单独实现了接口函数，如下：

```
/* 二维传输初始化函数 */
void edma_2d_init(edma_stream_2d_type *edma_streamx_2d, int16_t src_stride, int16_t dst_stride, uint16_t
xcnt, uint16_t ycnt);
/* 二维传输使能 */
void edma_2d_enable(edma_stream_2d_type *edma_streamx_2d, confirm_state new_state);
```

通过函数 edma_2d_init();可以配置二维传输的源跨步、目的跨步、XCNT 和 YCNT，这四个参数的

在使能 2D 传输前就需要配置好；然后调用函数 edma_2d_enable();即可开启二维传输功能。

4 EDMA 配置解析

以下对 EDMA 的配置接口及流程进行说明。

4.1 函数接口

表 5. 数据流配置函数列表

/* 复位数据流 */ void edma_reset(edma_stream_type *edma_streamx);
/* 初始化 EDMA 结构体参数 */ void edma_default_para_init(edma_init_type *edma_init_struct);
/* 初始化数据流 */ void edma_init(edma_stream_type *edma_streamx, edma_init_type *edma_init_struct);
/* 使能数据流 */ void edma_stream_enable(edma_stream_type *edma_streamx, confirm_state new_state);
/* 使能 DMAMUX */ void edmamux_enable(confirm_state new_state);
/* 写入 DMA 请求 ID 号 */ void edmamux_init(edmamux_channel_type *edmamux_channelx, edmamux_requst_id_sel_type edmamux_req_id);

4.2 数据流配置

- **设置外设地址（SxPADDR 寄存器）**

  数据传输的初始外设地址，在传输过程中（SEN = 1）不可被改变。

- **设置存储器地址（SxM0ADDR 寄存器）**

  数据传输的初始内存地址，在传输过程中（SEN = 1）不可被改变。

- **数据流配置（SxCTRL 寄存器）**

  包含优先级，数据传输方向、模式和宽度，地址增量模式、循环模式、双缓冲模式和传输/半传输/传输错误中断使能位。

  - **优先级（SPL）**

    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；EDMA 内部的打包、拆包机制允许 PWIDTH 与 MWIDTH 宽度不一致。

  - **地址增量模式（PINCM/MINCM）**

    当流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）。

  - **数据传输模式（PBURST/MBURST）**

分为单次传输（SINGLE），或突发传输（BURST）。

突发传输，在每个 DMA 请求到来后，根据配置传输 4、8、16 拍。

在非增量模式下，设置突发传输会将 4、8、16 拍突发传输转换为 4、8 或 16 个单次传输。

在直接模式下，PBURST 和 MBURST 位被硬件强制为 0（单次传输）。

- **外设流控选择（PFCTRL）**

  若选择外设做为流控制（PFCTRL = 1），硬件会将 DMA_SxDTCNT 的值强制为 0xFFFF，并由外设的 dma_ch_lt 信号指示数据传输结束。外设流控制不支持 M2M 模式和循环模式。

- **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 SxDTCNT 寄存器的内容会恢复成初始值。

- **双缓存模式（DMM）**

  当流配置设定为双缓冲模式时，硬件会自动启用循环模式。双缓冲区流有两个存储器指针 SxM0ADR/ SxM1ADR，由 CT 位標示當前使用之存储器指针，允许软件在 DMA 填充/使用第二个存储区的同时处理另一个存储区。

- **直接模式与 FIFO 模式设定（SxFCTRL 寄存器）**

  包含 FIFO 阈值选择，直接模式禁用和 FIFO 错误中断使能位。

  - **FIFO 阈值选择（FTHSEL）**

    分为 1/4、2/4、3/4 和全 FIFO 阈值；该设定仅在非直接模式下有效。

  - **FIFO 模式使能（FEN）**

    直接模式（FEN = 0）仅可在 P2M 或 M2P 模式下使用，且外设/存储器传输宽度需相等（MWIDTH = PWIDTH）并设定为单次传输（PBURST = MBURST = 0）。如果选择 M2M 模式，该位（FEN）被硬件强制为 1。

- **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（MUXSxCTRL 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（SxCTRL 寄存器的 SEN 位）**

4.3 配置流程

- 打开 EDMA 时钟；
- 调用数据流复位函数复位数据流；
- 调用结构体初始化函数初始化数据流结构体；
- 调用初始化函数初始化数据流；
- 调用 DMAMUX 使能函数以及 ID 号写入函数配置 DMAMUX 相关内容；
- 调用数据流使能函数开启数据流。

5 案例 数据从 FLASH 传输到 SRAM

5.1 功能简介

实现了使用 EDMA 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\edma\flash_to_sram

5.3 软件设计

1) 配置流程

- 开启 EDMA 外设时钟
- 配置 EDMA 数据流
- 开启传输完成中断
- 开启数据流
- 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 EDMA 时钟 */
  crm_periph_clock_enable(CRM_EDMA_PERIPH_CLOCK, TRUE);
  /* 初始化 EDMA 数据流 */
  /* edma stream1 configuration */
  edma_default_para_init(&edma_init_struct);
  edma_init_struct.direction = EDMA_DIR_MEMORY_TO_MEMORY;
  edma_init_struct.buffer_size = (uint32_t)BUFFER_SIZE;
  edma_init_struct.peripheral_data_width = EDMA_PERIPHERAL_DATA_WIDTH_WORD;
  edma_init_struct.peripheral_base_addr = (uint32_t)src_const_buffer;
  edma_init_struct.peripheral_inc_enable = TRUE;
  edma_init_struct.memory_data_width = EDMA_MEMORY_DATA_WIDTH_WORD;
  edma_init_struct.memory0_base_addr = (uint32_t)dst_buffer;
  edma_init_struct.memory_inc_enable = TRUE;
  edma_init_struct.peripheral_burst_mode = EDMA_PERIPHERAL_SINGLE;
```

```
  edma_init_struct.memory_burst_mode = EDMA_MEMORY_SINGLE;
  edma_init_struct.fifo_mode_enable = FALSE;
  edma_init_struct.fifo_threshold = EDMA_FIFO_THRESHOLD_1QUARTER;
  edma_init_struct.priority = EDMA_PRIORITY_HIGH;
  edma_init_struct.loop_mode_enable = FALSE;
  edma_init(EDMA_STREAM1, &edma_init_struct);
  /* 开启传输完成中断 */
  /* enable transfer full data intterrupt */
  edma_interrupt_enable(EDMA_STREAM1, EDMA_FDT_INT, TRUE);

  /* edma stream1 interrupt nvic init */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  nvic_irq_enable(EDMA_Stream1_IRQn, 1, 0);
  /* 开启数据流 */
  edma_stream_enable(EDMA_STREAM1, TRUE);
  /* 等待数据传输完成 */
  /* wait the end of transmission */
  while(data_counter_end != 0)
  {
  }
  /* 比较数据是否正确 */
  /* check if the transmitted and received data are equal */
  transfer_status = buffer_compare(src_const_buffer, dst_buffer, BUFFER_SIZE);

  /* transfer_status = success, if the transmitted and received data are the same
  transfer_status = error, if the transmitted and received data are different */
  if(transfer_status == SUCCESS)
  {
    /* 数据传输无误，开启 LED */
    /* turn led2/led3/led4 on */
    at32_led_on(LED2);
    at32_led_on(LED3);
    at32_led_on(LED4);
  }

  while(1)
  {
  }
}
```

■ EDMA_Stream1_IRQHandler 中断函数代码描述

```
void EDMA_Stream1_IRQHandler(void)
{
```

```
  if(edma_flag_get(EDMA_FDT1_FLAG) != RESET)
  {
    data_counter_end = 0;
    edma_flag_clear(EDMA_FDT1_FLAG);
  }
}
```

5.4 实验效果

- 如若数据传输无误，LED2/3/4 会点亮。

6 案例 EDMA 突发传输

6.1 功能简介

EDMA 突发传输为 EDMA 控制器的特有功能。

使用 EDMA 的突发传输将数据从 SRAM 传输到 TMR1 的 c1dt 至 c4dt，实现同时改变 TMR1 四个通道占空比。

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\edma\burst_mode

6.3 软件设计

1) 配置流程

- 开启 EDMA/TMR1/GPIOA 外设时钟
- 配置 EDMA 数据流
- 配置 TMR1 的通道 1 到 4 输出 PWM 波形
- 开启传输完成中断
- 开启数据流
- 开启 TMR1,使其溢出中断产生 DMA 请求

2) 代码介绍

- main 函数代码描述

```
Int main(void)
{
  /* 系统时钟初始化 */
  system_clock_config();
  /* 板级初始化 */
  at32_board_init();

  /* turn led2/led3/led4 on */
  at32_led_on(LED2);
  at32_led_on(LED3);
  at32_led_on(LED4);
  /* 打开 TMR1/GPIOA 时钟 */
  /* enable tmr1/gpioa clock */
  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
  /* 打开 EDMA 时钟 */
  /* enable dma1 clock */
  crm_periph_clock_enable(CRM_EDMA_PERIPH_CLOCK, TRUE);
```

```
/* timer1 output pin Configuration */
gpio_init_struct.gpio_pins = GPIO_PINS_8 | GPIO_PINS_9 | GPIO_PINS_10 |
GPIO_PINS_11;
gpio_init_struct.gpio_mode = GPIO_MODE_MUX;
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);
/* GPIO IOMUX 配置 */
gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE8, GPIO_MUX_1);
gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE9, GPIO_MUX_1);
gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE10, GPIO_MUX_1);
gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE11, GPIO_MUX_1);
/* EDMA 数据流 1 配置 */
/* edma stream1 configuration */
edma_default_para_init(&edma_init_struct);
edma_init_struct.direction = EDMA_DIR_MEMORY_TO_PERIPHERAL;
edma_init_struct.buffer_size = (uint32_t)4;
edma_init_struct.peripheral_data_width = EDMA_PERIPHERAL_DATA_WIDTH_WORD;
edma_init_struct.peripheral_base_addr = (uint32_t)&(TMR1->c1dt);
edma_init_struct.peripheral_inc_enable = TRUE;
edma_init_struct.memory_data_width = EDMA_MEMORY_DATA_WIDTH_WORD;
edma_init_struct.memory0_base_addr = (uint32_t)duty_buffer;
edma_init_struct.memory_inc_enable = TRUE;
/* burst 传输配置为一次 DMA 请求传输 4 拍数据 */
edma_init_struct.peripheral_burst_mode = EDMA_PERIPHERAL_BURST_4;
edma_init_struct.memory_burst_mode = EDMA_MEMORY_BURST_4;
edma_init_struct.fifo_mode_enable = TRUE;
edma_init_struct.fifo_threshold = EDMA_FIFO_THRESHOLD_FULL;
edma_init_struct.priority = EDMA_PRIORITY_HIGH;
edma_init_struct.loop_mode_enable = TRUE;
edma_init(EDMA_STREAM1, &edma_init_struct);
/* DMAMUX 初始化 */
/* edmamux init */
edmamux_enable(TRUE);
edmamux_init(EDMAMUX_CHANNEL1, EDMAMUX_DMAREQ_ID_TMR1_OVERFLOW);
/* 打开数据流 */
/* enable stream */
edma_stream_enable(EDMA_STREAM1, TRUE);
/* 配置 TMR1 每 500 的 clk 产生一次 DMA 请求 */
/* tmr1 configuration */
tmr_base_init(TMR1, 500, 0);
tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);
/* 配置 TMR1 的通道 1 到 4 输出 PWM 波 */
/* channel 1, 2, 3 and 4 configuration in output mode */
```

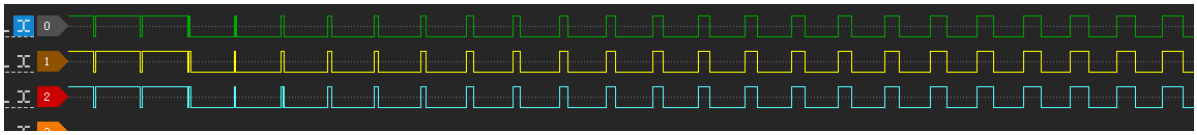
```
  tmr_output_default_para_init(&tmr_output_struct);
  tmr_output_struct.oc_mode = TMR_OUTPUT_CONTROL_PWM_MODE_B;
  tmr_output_struct.oc_output_state = TRUE;
  tmr_output_struct.oc_polarity = TMR_OUTPUT_ACTIVE_LOW;
  tmr_output_struct.oc_idle_state = TRUE;
  tmr_output_struct.occ_output_state = FALSE;
  tmr_output_struct.occ_polarity = TMR_OUTPUT_ACTIVE_HIGH;
  tmr_output_struct.occ_idle_state = FALSE;

  /* channel 1 */
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_1, &tmr_output_struct);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_1, duty_buffer[0]);

  /* channel 2 */
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_2, &tmr_output_struct);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_2, duty_buffer[1]);

  /* channel 3 */
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_3, &tmr_output_struct);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_3, duty_buffer[2]);

  /* channel 4 */
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_4, &tmr_output_struct);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_4, duty_buffer[3]);

  /* tmr1 output enable */
  tmr_output_enable(TMR1, TRUE);

  /* enable tmr1 overflow edam request */
  tmr_dma_request_enable(TMR1, TMR_OVERFLOW_DMA_REQUEST, TRUE);
  /*  打开 TMR1 溢出中断  */
  /* overflow interrupt enable */
  tmr_interrupt_enable(TMR1, TMR_OVF_INT, TRUE);

  /* tmr1 overflow interrupt nvic init */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  nvic_irq_enable(TMR1_OVF_TMR10_IRQn, 1, 0);
  /*  打开 TMR1 */
  /* enable tmr1 */
  tmr_counter_enable(TMR1, TRUE);

  while(1)
  {
  }
}
```

6.4 实验效果

- TMR1 的通道 1 到 4 每个周期会改变一次占空比。

如下为使用逻辑分析仪抓取的几个通道的输出波形，可以看到 PWM 波形的占空比每个周期都在改变。

图 15. Burst 传输实验结果



7 案例 EDMA 双缓冲区

7.1 功能简介

EDMA 存储器端双缓存区为 EDMA 控制器的特有功能。

本案例使用 EDMA 的双缓存区功能将 SRAM 内部的两块数据通过 IIS3 传输给 IIS2 并存储在 SRAM 内的另外两个数据块中。i2s3_buffer1_tx[32]和 i2s3_buffer2_tx[32]为 IIS3 传输的两块数据，i2s2_buffer1_rx[32]和 i2s2_buffer2_rx[32]为 IIS2 接收到的数据存储区域。

7.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

使用杜邦线将 IIS3 与 IIS2 连接起来，连接关系如下：

PD0 < ------- > PA4 (ws)

PD1 < ------- > PC10 (sck)

PD4 < ------- > PC12 (sd)

2) 软件环境

project\at_start_f4xx\examples\edma\i2s_halfduplex_edma_doublebuffer

7.3 软件设计

1) 配置流程

- 开启 EDMA/IIS3/IIS2/GPIO 外设时钟
- 配置 EDMA 数据流，使能双缓存区功能
- 配置 IIS3/2
- 开启传输完成中断
- 开启数据流

2) 代码介绍

- 代码描述

```
/* 数据流 1 传输完成中断函数入口 */
void EDMA_Stream1_IRQHandler(void)
{
  if(edma_flag_get(EDMA_FDT1_FLAG) != RESET)
  {
    if(transfer_count == 0)
    {
      if(buffer_compare(i2s3_buffer1_tx, i2s2_buffer1_rx, 32) == SUCCESS)
      {
        transfer_count = 1;
      }
      else
      {
```

```
            /* 发生错误 */
            at32_led_off(LED2);
            at32_led_off(LED3);
            at32_led_off(LED4);
          }
        }
        if(transfer_count == 1)
        {
          if(buffer_compare(i2s3_buffer2_tx, i2s2_buffer2_rx, 32) == SUCCESS)
          {
            /* 传输正确 */
            at32_led_on(LED2);
            at32_led_on(LED3);
            at32_led_on(LED4);
            /* end of transfer,close edma stream */
            edma_stream_enable(EDMA_STREAM1, FALSE);
            edma_stream_enable(EDMA_STREAM2, FALSE);
          }
          else
          {
            /* 发生错误 */
            at32_led_off(LED2);
            at32_led_off(LED3);
            at32_led_off(LED4);
          }
        }
        edma_flag_clear(EDMA_FDT1_FLAG);
      }
    }

    /* IIS 初始化函数 */
    static void i2s_config(void)
    {
      edma_init_type edma_init_struct;
      i2s_init_type i2s_init_struct;

      /* enable dma1 clock */
      crm_periph_clock_enable(CRM_EDMA_PERIPH_CLOCK, TRUE);

      /* i2s2_rx:edma stream1 configuration */
      edma_default_para_init(&edma_init_struct);
      edma_init_struct.direction = EDMA_DIR_PERIPHERAL_TO_MEMORY;
      edma_init_struct.buffer_size = (uint32_t)32;
      edma_init_struct.peripheral_data_width = EDMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
      edma_init_struct.peripheral_base_addr = (uint32_t)&(SPI2->dt);
      edma_init_struct.peripheral_inc_enable = FALSE;
```

```
edma_init_struct.memory_data_width = EDMA_MEMORY_DATA_WIDTH_HALFWORD;
edma_init_struct.memory0_base_addr = (uint32_t)i2s2_buffer1_rx;
edma_init_struct.memory_inc_enable = TRUE;
edma_init_struct.peripheral_burst_mode = EDMA_PERIPHERAL_SINGLE;
edma_init_struct.memory_burst_mode = EDMA_MEMORY_SINGLE;
edma_init_struct.fifo_mode_enable = FALSE;
edma_init_struct.fifo_threshold = EDMA_FIFO_THRESHOLD_1QUARTER;
edma_init_struct.priority = EDMA_PRIORITY_HIGH;
edma_init_struct.loop_mode_enable = TRUE;
edma_init(EDMA_STREAM1, &edma_init_struct);

/* edma stream1 double buffer mode init */
edma_double_buffer_mode_init(EDMA_STREAM1, (uint32_t)i2s2_buffer2_rx,
EDMA_MEMORY_0);
edma_double_buffer_mode_enable(EDMA_STREAM1, TRUE);

/* i2s3_tx:edma stream2 configuration */
edma_init_struct.direction = EDMA_DIR_MEMORY_TO_PERIPHERAL;
edma_init_struct.peripheral_base_addr = (uint32_t)&(SPI3->dt);
edma_init_struct.memory0_base_addr = (uint32_t)i2s3_buffer1_tx;
edma_init(EDMA_STREAM2, &edma_init_struct);

/* edma stream2 double buffer mode init */
edma_double_buffer_mode_init(EDMA_STREAM2, (uint32_t)i2s3_buffer2_tx,
EDMA_MEMORY_0);
edma_double_buffer_mode_enable(EDMA_STREAM2, TRUE);

/* enable transfer full data intterrupt */
edma_interrupt_enable(EDMA_STREAM1, EDMA_FDT_INT, TRUE);

/* edma stream1 interrupt nvic init */
nvic_irq_enable(EDMA_Stream1_IRQn, 1, 0);

/* edmamux init */
edmamux_enable(TRUE);
edmamux_init(EDMAMUX_CHANNEL1, EDMAMUX_DMAREQ_ID_SPI2_RX);
edmamux_init(EDMAMUX_CHANNEL2, EDMAMUX_DMAREQ_ID_SPI3_TX);

crm_periph_clock_enable(CRM_SPI3_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_SPI2_PERIPH_CLOCK, TRUE);
i2s_default_para_init(&i2s_init_struct);
i2s_init_struct.audio_protocol = I2S_AUDIO_PROTOCOL_PHILLIPS;
i2s_init_struct.data_channel_format = I2S_DATA_16BIT_CHANNEL_32BIT;
i2s_init_struct.mclk_output_enable = TRUE;
i2s_init_struct.audio_sampling_freq = I2S_AUDIO_FREQUENCY_48K;
i2s_init_struct.clock_polarity = I2S_CLOCK_POLARITY_LOW;
```

```
  i2s_init_struct.operation_mode = I2S_MODE_MASTER_TX;
  i2s_init(SPI3, &i2s_init_struct);

  i2s_init_struct.operation_mode =I2S_MODE_SLAVE_RX;
  i2s_init(SPI2, &i2s_init_struct);

  edma_stream_enable(EDMA_STREAM1, TRUE);
  edma_stream_enable(EDMA_STREAM2, TRUE);
  spi_i2s_dma_receiver_enable(SPI2, TRUE);

  spi_i2s_dma_transmitter_enable(SPI3, TRUE);

  i2s_enable(SPI2, TRUE);

  i2s_enable(SPI3, TRUE);
}

/* GPIO 口初始化 */
static void gpio_config(void)
{
  gpio_init_type gpio_initstructure;
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOD_PERIPH_CLOCK, TRUE);

  /* master ws pin */
  gpio_initstructure.gpio_out_type       = GPIO_OUTPUT_PUSH_PULL;
  gpio_initstructure.gpio_pull           = GPIO_PULL_UP;
  gpio_initstructure.gpio_mode           = GPIO_MODE_MUX;
  gpio_initstructure.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_initstructure.gpio_pins           = GPIO_PINS_4;
  gpio_init(GPIOA, &gpio_initstructure);
  gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE4, GPIO_MUX_6);

  /* master ck pin */
  gpio_initstructure.gpio_pull           = GPIO_PULL_DOWN;
  gpio_initstructure.gpio_pins           = GPIO_PINS_10;
  gpio_init(GPIOC, &gpio_initstructure);
  gpio_pin_mux_config(GPIOC, GPIO_PINS_SOURCE10, GPIO_MUX_6);

  /* master sd pin */
  gpio_initstructure.gpio_pull           = GPIO_PULL_UP;
  gpio_initstructure.gpio_pins           = GPIO_PINS_12;
  gpio_init(GPIOC, &gpio_initstructure);
  gpio_pin_mux_config(GPIOC, GPIO_PINS_SOURCE12, GPIO_MUX_6);
```

```
  /* master mck pin */
  gpio_initstructure.gpio_pull              = GPIO_PULL_UP;
  gpio_initstructure.gpio_pins              = GPIO_PINS_7;
  gpio_init(GPIOC, &gpio_initstructure);
  gpio_pin_mux_config(GPIOC, GPIO_PINS_SOURCE7, GPIO_MUX_6);

  /* slave ws pin */
  gpio_initstructure.gpio_pull              = GPIO_PULL_UP;
  gpio_initstructure.gpio_pins              = GPIO_PINS_0;
  gpio_init(GPIOD, &gpio_initstructure);
  gpio_pin_mux_config(GPIOD, GPIO_PINS_SOURCE0, GPIO_MUX_7);

  /* slave ck pin */
  gpio_initstructure.gpio_pull              = GPIO_PULL_DOWN;
  gpio_initstructure.gpio_pins              = GPIO_PINS_1;
  gpio_init(GPIOD, &gpio_initstructure);
  gpio_pin_mux_config(GPIOD, GPIO_PINS_SOURCE1, GPIO_MUX_6);

  /* slave sd pin */
  gpio_initstructure.gpio_pull              = GPIO_PULL_UP;
  gpio_initstructure.gpio_pins              = GPIO_PINS_4;
  gpio_init(GPIOD, &gpio_initstructure);
  gpio_pin_mux_config(GPIOD, GPIO_PINS_SOURCE4, GPIO_MUX_6);
}

/* 主函数入口 */
int main(void)
{
  /* 设置中断优先级分组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  /* 系统时钟初始化 */
  system_clock_config();
  /* 板载初始化 */
  at32_board_init();
  /* pin 脚初始化 */
  gpio_config();
  /* IIS 和 EDMA 初始化 */
  i2s_config();
  while(1)
  {
  }
}
```

7.4 实验效果

- LED2/3/4 都点亮则表示数据传输正确。

8 案例 EDMA 链接列表传输

8.1 功能简介

EDMA 链接列表传输为 EDMA 控制器的特有功能。

本案例使用 EDMA 的链接列表传输功能，将片上存储器的 3 块区域的数据通过 EDMA 发送给 USART1,USART1 在通过自身的 TX 引脚将数据发送出去。

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\edma\link_list_mode

8.3 软件设计

1) 配置流程

- 开启 EDMA/USART1/GPIO 外设时钟
- 配置 EDMA 数据流，使能链接列表传输功能
- 开启传输完成中断
- 开启数据流

2) 代码介绍

- 代码描述

```
#define LIST_COUNT                      3
#define BUFFER_SIZE                     35

/* 链接列表描述符定义 */
typedef struct
{
  uint32_t ctrl_dtcnt;
  uint32_t paddr;
  uint32_t m0addr;
  uint32_t llp;
} ll_descriptors_type;

/* 发送数据 buffer */
const int8_t tx_buffer[LIST_COUNT][BUFFER_SIZE] ={
  {'T','h','i','s',' ','i','s',' ','E','D','M','A',' '},
  {'L','i','n','k',' ','L','i','s','t',' ','M','o','d','e',' ','t','e','s','t','!'},
  {'\r','\n','T','h','i','s',' ','t','e','s','t',' ','c','a','s','e',' ','u','s','e',' ','U','s','a','r','t','1','_','T','X','!','\r','\n'}
};
/* 使用链接列表传输时，描述符必须 4 字对齐 */
/* link list descriptor array, start address must be aligned by 16 bytes */
```

```
__ALIGNED(16) ll_descriptors_type edma_ll_descriptors_tx[LIST_COUNT];

/* declared private functions */
void uart_init(uint32_t bound);
void edma_ll_descriptors_init(void);

/* 对描述符进行初始化 */
void edma_ll_descriptors_init(void)
{
  uint32_t index = 0;

  for(index = 0; index < LIST_COUNT; index ++)
  {
    edma_ll_descriptors_tx[index].ctrl_dtcnt = 0x20090000 | sizeof(tx_buffer[index]);
    edma_ll_descriptors_tx[index].paddr = (uint32_t)(&USART1->dt);
    edma_ll_descriptors_tx[index].m0addr = (uint32_t)(&tx_buffer[index][0]);
    if(index == (LIST_COUNT - 1))
    {
      edma_ll_descriptors_tx[index].llp = 0;
    }
    else
    {
      edma_ll_descriptors_tx[index].llp = (uint32_t)(&edma_ll_descriptors_tx[index + 1]);
    }
  }
}

/* 主函数 */
int main(void)
{
  /* 系统时钟初始化 */
  system_clock_config();
  /* 板载设备初始化 */
  at32_board_init();
  /* 串口初始化 */
  uart_init(115200);
  /* 打开 dma1 时钟 */
  crm_periph_clock_enable(CRM_EDMA_PERIPH_CLOCK, TRUE);

  /* edma link list descriptor init */
  edma_ll_descriptors_init();

  /* 链接列表传输初始化 */
  edma_link_list_enable(EDMA_STREAM1_LL, TRUE);
  edma_link_list_init(EDMA_STREAM1_LL, (uint32_t)edma_ll_descriptors_tx);
```

```
  /* edmamux configuration */
  edmamux_enable(TRUE);
  edmamux_init(EDMAMUX_CHANNEL1, EDMAMUX_DMAREQ_ID_USART1_TX);

  /* enable transfer full data intterrupt */
  edma_interrupt_enable(EDMA_STREAM1, EDMA_FDT_INT, TRUE);

  /* edma stream1 interrupt nvic init */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  nvic_irq_enable(EDMA_Stream1_IRQn, 1, 0);

  /* enable edma stream1 */
  edma_stream_enable(EDMA_STREAM1, TRUE);

  while(1)
  {
  }
}

/* 串口初始化函数 */
void uart_init(uint32_t bound)
{
  gpio_init_type gpio_init_struct;

  /* enable the uart clock */
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_USART1_PERIPH_CLOCK, TRUE);

  /* set default parameter */
  gpio_default_para_init(&gpio_init_struct);

  /* configure uart tx gpio */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX;
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
  gpio_init_struct.gpio_pins = GPIO_PINS_9;
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
  gpio_init(GPIOA, &gpio_init_struct);
  /* configure uart rx gpio */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX;
  gpio_init_struct.gpio_pins = GPIO_PINS_10;
  gpio_init(GPIOA, &gpio_init_struct);

  gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE9, GPIO_MUX_7);
  gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE10, GPIO_MUX_7);
```

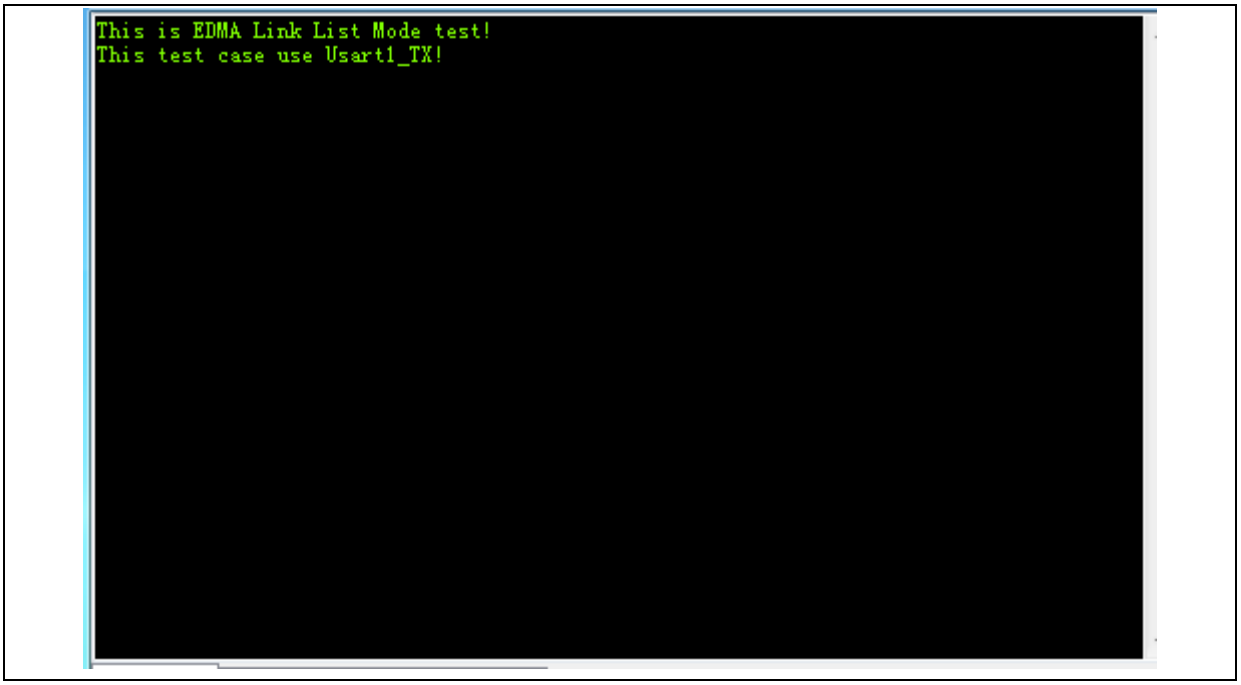
```
  /* configure usart */
  usart_init(USART1, bound, USART_DATA_8BITS, USART_STOP_1_BIT);
  usart_parity_selection_config(USART1, USART_PARITY_NONE);
  usart_transmitter_enable(USART1, TRUE);
  usart_receiver_enable(USART1, TRUE);
  usart_dma_transmitter_enable(USART1, TRUE);
  usart_enable(USART1, TRUE);
}

/*  数据流 1 传输完成中断函数入口  */
void EDMA_Stream1_IRQHandler(void)
{
  if(edma_flag_get(EDMA_FDT1_FLAG) != RESET)
  {
    /* led2/3/4 on    */
    at32_led_on(LED2);
    at32_led_on(LED3);
    at32_led_on(LED4);
    edma_flag_clear(EDMA_FDT1_FLAG);
  }
}
```

8.4 实验效果

■ LED2/3/4 都点亮则表示数据传输完成，并可通过串口工具查看发送出来的数据。

串口工具接收的数据如下图所示：

图 16. 链接列表传输实验结果



9 案例 EDMA 二维传输

9.1 功能简介

EDMA 二维传输为 EDMA 控制器的特有功能。

本案例使用 EDMA 的二维传输功能，将片上存储器的 1 块区域的数据通过 EDMA 二维传输功能剪裁出其左边 1/2，并通过串口打印出来。

9.2 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\edma\ two_dimension_mode

9.3 软件设计

1) 配置流程

- 开启 EDMA/USART1/GPIO 外设时钟
- 配置 EDMA 数据流，使能二维传输功能
- 开启传输完成中断
- 开启数据流

2) 代码介绍

- 代码描述

```
int main(void)
{
  int index;
  system_clock_config();
  /* 板载设备初始化 */
  at32_board_init();
  /* 串口初始化 */
  uart_init(115200);
  /* 打开 dma1 时钟 */
  crm_periph_clock_enable(CRM_EDMA_PERIPH_CLOCK, TRUE);

  /* 初始化 EDMA 数据流 1 */
  edma_default_para_init(&edma_init_struct);
  edma_init_struct.direction = EDMA_DIR_MEMORY_TO_MEMORY;
  edma_init_struct.buffer_size = (uint32_t)BUFFER_SIZE;
  edma_init_struct.peripheral_data_width = EDMA_PERIPHERAL_DATA_WIDTH_WORD;
  edma_init_struct.peripheral_base_addr = (uint32_t)src_const_buffer;
  edma_init_struct.peripheral_inc_enable = TRUE;
  edma_init_struct.memory_data_width = EDMA_MEMORY_DATA_WIDTH_WORD;
  edma_init_struct.memory0_base_addr = (uint32_t)dst_buffer;
```

```
edma_init_struct.memory_inc_enable = TRUE;
edma_init_struct.peripheral_burst_mode = EDMA_PERIPHERAL_SINGLE;
edma_init_struct.memory_burst_mode = EDMA_MEMORY_SINGLE;
edma_init_struct.fifo_mode_enable = TRUE;
edma_init_struct.fifo_threshold = EDMA_FIFO_THRESHOLD_FULL;
edma_init_struct.priority = EDMA_PRIORITY_HIGH;
edma_init_struct.loop_mode_enable = FALSE;
edma_init(EDMA_STREAM1, &edma_init_struct);

/* 打开数据流传输完成中断 */
edma_interrupt_enable(EDMA_STREAM1, EDMA_FDT_INT, TRUE);

/* edma stream1 interrupt nvic init */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
nvic_irq_enable(EDMA_Stream1_IRQn, 1, 0);

/* 配置二维传输 */
/* source stride = 0x10, destination stride = 0x8, xcnt = 0x2, ycnt = 0x8 */
edma_2d_init(EDMA_STREAM1_2D, 0x10, 0x8, 0x2, 0x8);
edma_2d_enable(EDMA_STREAM1_2D, TRUE);

edma_stream_enable(EDMA_STREAM1, TRUE);

/* wait the end of transmission */
while(data_counter_end != 0)
{
}
/* 打印出数据 */
printf("source buffer:\r\n");
for(counter = 0; counter < BUFFER_SIZE; counter ++)
{
  printf("%x    ",src_const_buffer[counter]);
  index ++;
  if(index == 4)
  {
    printf("\r\n");
    index = 0;
  }
}

printf("destination buffer:\r\n");
for(counter = 0; counter < BUFFER_SIZE/2; counter ++)
{
  printf("%x    ",dst_buffer[counter]);
  index ++;
  if(index == 2)
```

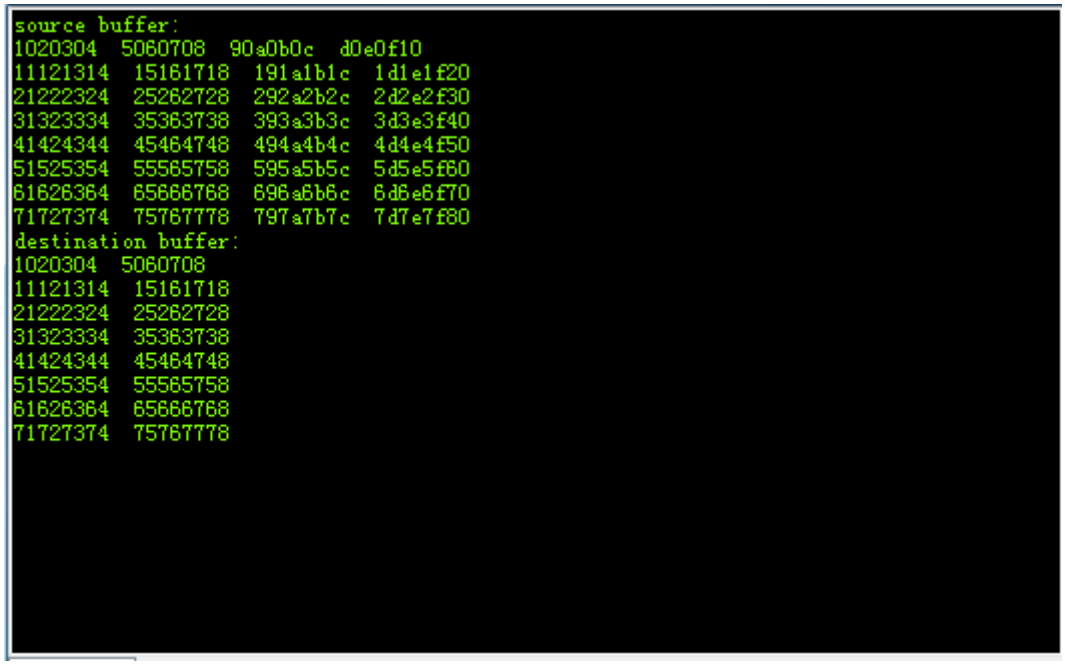
```
    {
      printf("\r\n");
      index = 0;
    }
  }

  while(1)
  {
  }
}
```

9.4 实验效果

- 通过串口工具查看发送出来的数据。

串口工具接收的数据如下图所示：

图 17. 二维传输实验结果



10 文档版本历史

表 6. 文档版本历史

日期	版本	变更
2021.2.29	2.0.0	最初版本
2022.6.13	2.0.1	对一些功能的描述增加图示

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，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) 汽车应用或汽车环境；(D) 航天应用或航天环境，且/或(E) 武器。因雅特力产品不是为前述应用设计的，而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，风险由购买者单独承担，并且独力负责在此类相关使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。