3.9 直接内存存取控制器 DMAC

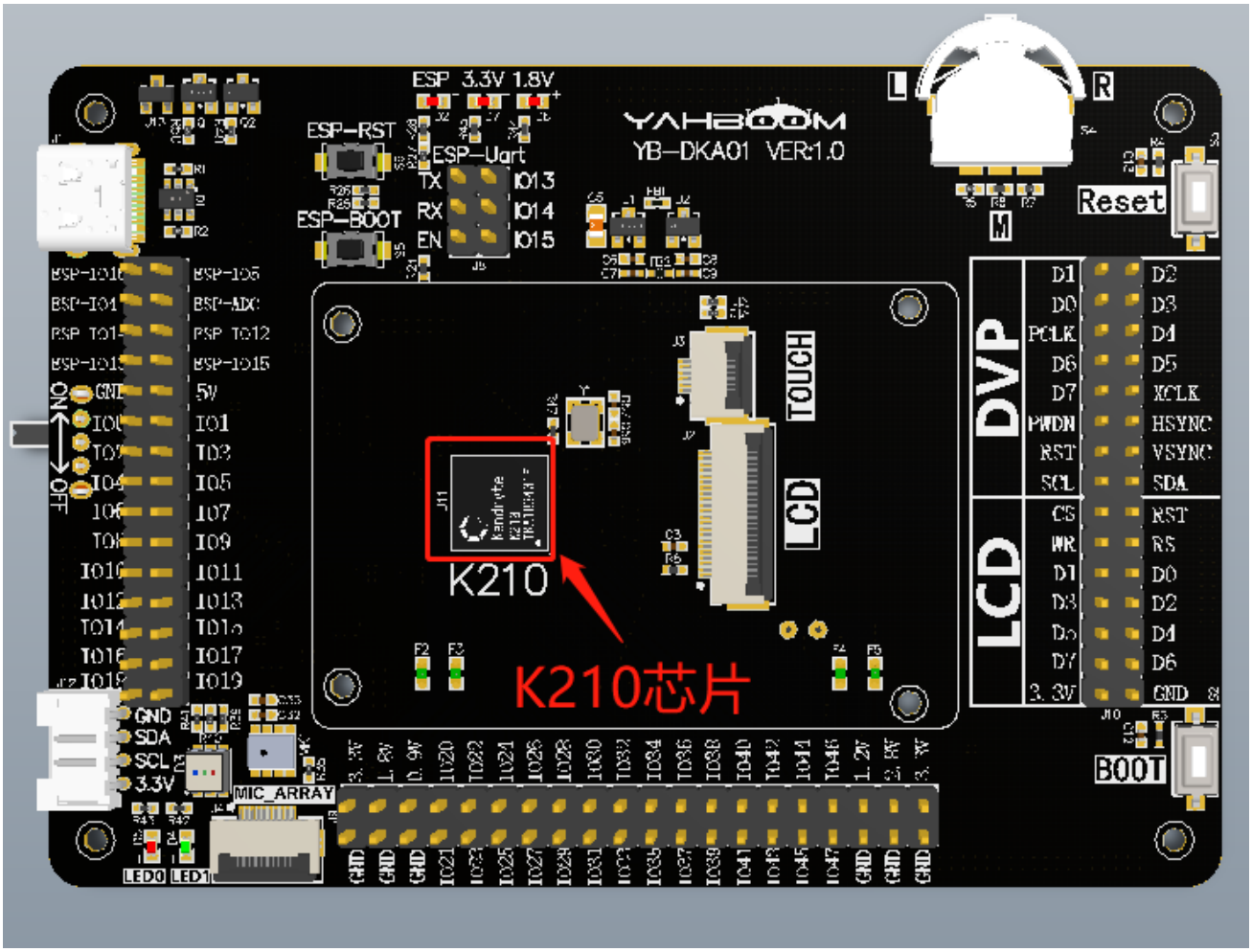
一、实验目的

本节课主要学习 K210 的直接内存存取控制器 DMAC 的功能。

二、实验准备

1.实验元件

K210 芯片的直接内存存取控制器 DMAC 功能



2.元件特性

直接存储访问（Direct Memory Access，DMA）用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。可以在无需任何 CPU 操作的情况下

通过 DMA 快速移动数据，从而提高了 CPU 的效率。

DMA 模块具有以下功能：

自动选择一路空闲的 DMA 通道用于传输，

根据源地址和目标地址自动选择软件或硬件握手协议，

支持 1、2、4、8 字节的元素大小，源和目标大小不必一致，

异步或同步传输功能，

循环传输功能，常用于刷新屏幕或音频录放等场景。

3.SDK 中对应 API 功能

对应的头文件 dmac.h

- dmac_init：初始化 DMA
- dmac_set_single_mode：设置单路 DMA 参数
- dmac_is_done：用于 DMAC 启动后判断是否完成传输。用于 DMAC 启动传输后，如果在启动前判断会不准确。
- dmac_wait_done：等待 DMA 完成工作
- dmac_set_irq：设置 DMAC 中断的回调函数
- dmac_set_src_dest_length：设置 DMAC 的源地址、目的地址和长度，然后启动 DMAC 传输。如果 src 为 NULL 则不设置源地址，dest 为 NULL 则不设置目的地址，len<=0 则不设置长度。

该函数一般用于 DMAC 中断中，使 DMA 继续传输数据，而不必再次设置 DMAC 的所有参数以节省时间。

- dmac_is_idle：判断 DMAC 当前通道是否空闲，该函数在传输前和传输后都可以用来判断 DMAC 状态。
- dmac_wait_idle：等待 DMAC 进入空闲状态。

三、实验原理

DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器来实行和完成。典型的例子就是移动一个外部内存的区块到芯片内部更快的内存区。像是这样的操作并没有让处理

器工作拖延，反而可以被重新排程去处理其他的工作。

在实现 DMA 传输时，是由 DMA 控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即 DMA 传输前，CPU 要把总线控制权交给 DMA 控制器，而在结束 DMA 传输后，DMA 控制器应立即把总线控制权再交回给 CPU。一个完整的 DMA 传输过程必须经过 DMA 请求、DMA 响应、DMA 传输、DMA 结束 4 个步骤。

请求：CPU 对 DMA 控制器初始化，并向 I/O 接口发出操作命令，I/O 接口提出 DMA 请求。

响应：DMA 控制器对 DMA 请求判别优先级及屏蔽，向总线裁决逻辑提出总线请求。当 CPU 执行完当前总线周期即可释放总线控制权。此时，总线裁决逻辑输出总线应答，表示 DMA 已经响应，通过 DMA 控制器通知 I/O 接口开始 DMA 传输。

传输：DMA 控制器获得总线控制权后，CPU 即刻挂起或只执行内部操作，由 DMA 控制器输出读写命令，直接控制 RAM 与 I/O 接口进行 DMA 传输。

在 DMA 控制器的控制下，在存储器和外部设备之间直接进行数据传送，在传送过程中不需要中央处理器的参与。开始时需提供要传送的数据的起始位置和数据长度。

结束：当完成规定的成批数据传送后，DMA 控制器即释放总线控制权，并向 I/O 接口发出结束信号。当 I/O 接口收到结束信号后，一方面停止 I/O 设备的工作，另一方面向 CPU 提出中断请求，使 CPU 从不介入的状态解脱，并执行一段检查本次 DMA 传输操作正确性的代码。最后，带着本次操作结果及状态继续执行原来的程序。

由此可见，DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据

的通路，使 CPU 的效率大为提高。

四、实验过程

1. 首先初始化 K210 的硬件引脚和软件功能使用的是 FPIOA 映射关系。

```
/*****************************HARDWARE-PIN*************************
// 硬件IO口，与原理图对应
#define PIN_RGB_R               (6)
#define PIN_RGB_G               (7)
#define PIN_RGB_B               (8)

#define PIN_UART_USB_RX         (4)
#define PIN_UART_USB_TX         (5)

/*****************************SOFTWARE-GPIO************************
// 软件GPIO口，与程序对应
#define RGB_R_GPIONUM           (0)
#define RGB_G_GPIONUM           (1)
#define RGB_B_GPIONUM           (2)

#define UART_USB_NUM            UART_DEVICE_3

/*****************************FUNC-GPIO****************************
// GPIO口的功能，绑定到硬件IO口
#define FUNC_RGB_R              (FUNC_GPIOHS0 + RGB_R_GPIONUM)
#define FUNC_RGB_G              (FUNC_GPIOHS0 + RGB_G_GPIONUM)
#define FUNC_RGB_B              (FUNC_GPIOHS0 + RGB_B_GPIONUM)

#define FUNC_UART_USB_RX        (FUNC_UART1_RX + UART_USB_NUM * 2)
#define FUNC_UART_USB_TX        (FUNC_UART1_TX + UART_USB_NUM * 2)
```

```
void hardware_init(void)
{
    /* fpioa映射 */
    fpioa_set_function(PIN_RGB_R, FUNC_RGB_R);
    fpioa_set_function(PIN_RGB_G, FUNC_RGB_G);
    fpioa_set_function(PIN_RGB_B, FUNC_RGB_B);

    fpioa_set_function(PIN_UART_USB_RX, FUNC_UART_USB_RX);
    fpioa_set_function(PIN_UART_USB_TX, FUNC_UART_USB_TX);
}
```

2. 在使用 RGB 灯前需要初始化，也就是把 RGB 灯的软件 GPIO 设置为输出模

式。

```
void init_rgb(void)
{
    /* 设置RGB灯的GPIO模式为输出 */
    gpiohs_set_drive_mode(RGB_R_GPIONUM, GPIO_DM_OUTPUT);
    gpiohs_set_drive_mode(RGB_G_GPIONUM, GPIO_DM_OUTPUT);
    gpiohs_set_drive_mode(RGB_B_GPIONUM, GPIO_DM_OUTPUT);

    /* 关闭RGB灯 */
    rgb_all_off();
}
```

3.然后关闭 RGB 灯，同样是设置 RGB 灯的 GPIO 为高电平则可以让 RGB 灯熄灭。

```
void rgb_all_off(void)
{
    gpiohs_set_pin(RGB_R_GPIONUM, GPIO_PV_HIGH);
    gpiohs_set_pin(RGB_G_GPIONUM, GPIO_PV_HIGH);
    gpiohs_set_pin(RGB_B_GPIONUM, GPIO_PV_HIGH);
}
```

4.初始化串口 3，并且设置串口串口波特率为 115200，然后通过 dma 通道 0 发送“hello yahboom!”的欢迎语。

```
/* 初始化串口，设置波特率为115200 */
uart_init(UART_USB_NUM);
uart_configure(UART_USB_NUM, 115200, UART_BITWIDTH_8BIT, UART_STOP_1, UART_PARITY_NONE);

/* 开机发送hello yahboom!欢迎语 */
char *hel = {"hello yahboom!\n"};
uart_send_data_dma(UART_USB_NUM, DMAC_CHANNEL0, (uint8_t *)hel, strlen(hel));
```

5.接下来是 dma 通道 1 读取串口的数据，并且经过协议过滤的过程，通过设置 rec_flag 的值，从而让数据必须是 16 进制的 FFAA 开头才可以进行解析，解析后，串口把接收到的真正数据通过 dma 通道 0 发出。

```c
uint8_t recv = 0;
int rec_flag = 0;

while (1)
{
    /* 通过DMA通道1接收串口数据，保存到recv中 */
    uart_receive_data_dma(UART_USB_NUM, DMAC_CHANNEL1, &recv, 1);
    /* 以下是判断协议，必须是FFAA开头的数据才可以 */
    switch(rec_flag)
    {
    case 0:
        if(recv == 0xFF)
            rec_flag = 1;
        break;
    case 1:
        if(recv == 0xAA)
            rec_flag = 2;
        else if(recv != 0xFF)
            rec_flag = 0;
        break;
    case 2:
        /* 解析真正的数据 */
        parse_cmd(&recv);
        /* 通过dma通道0发送串口数据 */
        uart_send_data_dma(UART_USB_NUM, DMAC_CHANNEL0, &recv, 1);
        rec_flag = 0;
        break;
    }
}
```

6.以下是串口通过 DMA 发送和接收数据的函数的源码，SDK 已经包含了这两个函数，直接调用即可。

```c
void uart_receive_data_dma(uart_device_number_t uart_channel, dmac_channel_number_t dmac_channel, uint8_t *buffer, size_t buf_len)
{
#if FIX_CACHE
    uint32_t *v_recv_buf = (uint32_t *)iomem_malloc(buf_len * sizeof(uint32_t));
#else
    uint32_t *v_recv_buf = (uint32_t *)malloc(buf_len * sizeof(uint32_t));
#endif
    configASSERT(v_recv_buf != NULL);

    sysctl_dma_select((sysctl_dma_channel_t)dmac_channel, SYSCTL_DMA_SELECT_UART1_RX_REQ + uart_channel * 2);

    dmac_set_single_mode(dmac_channel, (void *)(&uart[uart_channel]->RBR), v_recv_buf, DMAC_ADDR_NOCHANGE, DMAC_ADDR_INCREMENT,
                         DMAC_MSIZE_1, DMAC_TRANS_WIDTH_32, buf_len);

    dmac_wait_done(dmac_channel);
    for(uint32_t i = 0; i < buf_len; i++)
    {
        buffer[i] = (uint8_t)(v_recv_buf[i] & 0xff);
    }
#if FIX_CACHE
    iomem_free(v_recv_buf);
#else
    free(v_recv_buf);
#endif
}
```

```c
void uart_send_data_dma(uart_device_number_t uart_channel, dmac_channel_number_t dmac_channel, const uint8_t *buffer, size_t buf_len)
{
#if FIX_CACHE
    uint32_t *v_send_buf = iomem_malloc(buf_len * sizeof(uint32_t));
#else
    uint32_t *v_send_buf = malloc(buf_len * sizeof(uint32_t));
#endif
    configASSERT(v_send_buf != NULL);

    for(uint32_t i = 0; i < buf_len; i++)
        v_send_buf[i] = buffer[i];

    sysctl_dma_select((sysctl_dma_channel_t)dmac_channel, SYSCTL_DMA_SELECT_UART1_TX_REQ + uart_channel * 2);
    dmac_set_single_mode(dmac_channel, v_send_buf, (void *)(&uart[uart_channel]->THR), DMAC_ADDR_INCREMENT, DMAC_ADDR_NOCHANGE,
                         DMAC_MSIZE_1, DMAC_TRANS_WIDTH_32, buf_len);

    dmac_wait_done(dmac_channel);
#if FIX_CACHE
    iomem_free((void *)v_send_buf);
#else
    free((void *)v_send_buf);
#endif
}
```

7.解析真正的数据，根据数据的不同数值，修改 RGB 灯的颜色和关闭状态。如果是 0x11 则亮红灯，0x22 则红灯灭；0x33 则亮绿灯，0x44 则绿灯灭；0x55 则亮蓝灯，0x66 则蓝灯灭；

```c
#define CMD_RGB_R_ON            0x11
#define CMD_RGB_R_OFF           0x22
#define CMD_RGB_G_ON            0x33
#define CMD_RGB_G_OFF           0x44
#define CMD_RGB_B_ON            0x55
#define CMD_RGB_B_OFF           0x66
```

```c
int parse_cmd(uint8_t *cmd)
{
    switch(*cmd)
    {
    case CMD_RGB_R_ON:
        /* RGB亮红灯*/
        gpiohs_set_pin(RGB_R_GPIONUM, GPIO_PV_LOW);
        break;
    case CMD_RGB_R_OFF:
        /* RGB红灯灭*/
        gpiohs_set_pin(RGB_R_GPIONUM, GPIO_PV_HIGH);
        break;
    case CMD_RGB_G_ON:
        /* RGB亮绿灯*/
        gpiohs_set_pin(RGB_G_GPIONUM, GPIO_PV_LOW);
        break;
    case CMD_RGB_G_OFF:
        /* RGB绿灯灭*/
        gpiohs_set_pin(RGB_G_GPIONUM, GPIO_PV_HIGH);
        break;
    case CMD_RGB_B_ON:
        /* RGB亮蓝灯*/
        gpiohs_set_pin(RGB_B_GPIONUM, GPIO_PV_LOW);
        break;
    case CMD_RGB_B_OFF:
        /* RGB蓝灯灭*/
        gpiohs_set_pin(RGB_B_GPIONUM, GPIO_PV_HIGH);
        break;
    }
    return 0;
}
```

8.编译调试，烧录运行

把本课程资料中的 dmac 复制到 SDK 中的 src 目录下,然后进入 build 目录，运行以下命令编译。

```
cmake .. -DPROJ=dmac -G "MinGW Makefiles"
make
```

```
Scanning dependencies of target dmac
[ 97%] Building C object CMakeFiles/dmac.dir/src/dmac/main.c.obj
[100%] Linking C executable dmac
Generating .bin file ...
[100%] Built target dmac
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

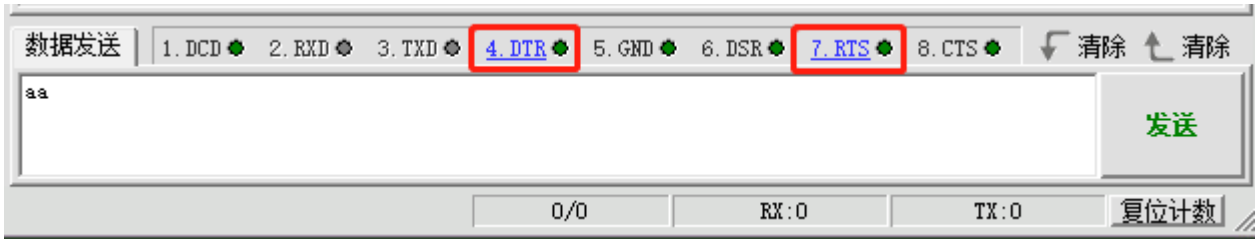
编译完成后，在 build 文件夹下会生成 dmac.bin 文件。

使用 type-C 数据线连接电脑与 K210 开发板，打开 kflash，选择对应的设备，再将程序固件烧录到 K210 开发板上。

五、实验现象

烧录完成固件后，系统会弹出一个终端界面，如果没有弹出终端界面的可以打开串口助手显示调试内容。

打开电脑的串口助手，选择对应的 K210 开发板对应的串口号，波特率设置为 115200，然后点击打开串口助手。注意还需要设置一下串口助手的 DTR 和 RTS。在串口助手底部此时的 4.DTR 和 7.RTS 默认是红色的，点击 4.DTR 和 7.RTS，都设置为绿色，然后按一下 K210 开发板的复位键。



由于通讯里使用的是 uint8_t 类型数据，所以我们需要把串口助手的发送和接收设置改为 HEX，如果没有选择 HEX 类型的话，K210 芯片会认为数据不正确而不进行解析数据。

在串口助手输入 FF AA 11，红灯亮，FF AA 22，红灯熄灭；输入 FF AA 33，绿灯亮，FF AA 44，绿灯熄灭；输入 FF AA 55，蓝灯亮，FF AA 66，蓝灯熄灭。

然后点击发送就可以看到效果了。

六、实验总结

1.直接内存存取控制器 DMAC 需要搭配其他的设备，如串口、I2C 或者 I2S 通讯来使用。

2.DMAC 是可以提高 CPU 效率，直接通过 DMA 在设备和内存之间传输数据，而 CPU 只需要启动 dma 传输就可以，等待完成即可。

附：API 对应的头文件 dmac.h

dmac_init

描述

初始化 DMA。

函数原型

```
void dmac_init(void)
```

参数

无。

返回值

无。

dmac_set_single_mode

描述

设置单路 DMA 参数。

函数原型

```
void dmac_set_single_mode(dmac_channel_number_t channel_num, const void *src,
void *dest, dmac_address_increment_t src_inc, dmac_address_increment_t
dest_inc, dmac_burst_trans_length_t dmac_burst_size, dmac_transfer_width_t
dmac_trans_width, size_t block_size)
```

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
src	源地址	输入
dest	目标地址	输出
src_inc	源地址是否自增	输入
dest_inc	目标地址是否自增	输入
dmac_burst_size	突发传输数量	输入
dmac_trans_width	单次传输数据位宽	输入
block_size	传输数据的个数	输入

返回值

无。

dmac_is_done

描述

用于 DMAC 启动后判断是否完成传输。用于 DMAC 启动传输后，如果在启动前判断会不准确。

函数原型

```
int dmac_is_done(dmac_channel_number_t channel_num)
```

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

返回值

返回值	描述
0	未完成
1	已完成

dmac_wait_done

描述

等待 DMA 完成工作。

函数原型

```
void dmac_wait_done(dmac_channel_number_t channel_num)
```

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

返回值

无。

dmac_set_irq

描述

设置 DMAC 中断的回调函数

函数原型

```
void dmac_set_irq(dmac_channel_number_t channel_num , plic_irq_callback_t
dmac_callback, void *ctx, uint32_t priority)
```

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
dmac_callback	中断回调函数	输入
ctx	回调函数的参数	输入
priority	中断优先级	输入

返回值

无。

dmac_set_src_dest_length

描述

设置 DMAC 的源地址、目的地址和长度，然后启动 DMAC 传输。如果 src 为 NULL 则不设置源地址，dest 为 NULL 则不设置目的地址，len<=0 则不设置长度。

该函数一般用于 DMAC 中断中，使 DMA 继续传输数据，而不必再次设置 DMAC 的所有参数以节省时间。

函数原型

```
void dmac_set_src_dest_length(dmac_channel_number_t channel_num, const void
*src, void *dest, size_t len)
```

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
src	中断回调函数	输入
dest	回调函数的参数	输入
len	传输长度	输入

返回值

无。

dmac_is_idle

描述

判断 DMAC 当前通道是否空闲，该函数在传输前和传输后都可以用来判断 DMAC 状态。

函数原型

```
int dmac_is_idle(dmac_channel_number_t channel_num)
```

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

返回值

返回值	描述
0	忙
1	空闲

dmac_wait_idle

描述

等待 DMAC 进入空闲状态。

参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

返回值

无。

举例

```
/* I2C 通过 DMA 发送 128 个 int 数据 */
uint32_t buf[128];
dmac_wait_idle(SYSCTL_DMA_CHANNEL_0);
sysctl_dma_select(SYSCTL_DMA_CHANNEL_0, SYSCTL_DMA_SELECT_I2C0_TX_REQ);
dmac_set_single_mode(SYSCTL_DMA_CHANNEL_0, buf,
(void*)(&i2c_adapter->data_cmd), DMAC_ADDR_INCREMENT, DMAC_ADDR_NOCHANGE,
DMAC_MSIZE_4, DMAC_TRANS_WIDTH_32, 128);
dmac_wait_done(SYSCTL_DMA_CHANNEL_0);
```

数据类型

相关数据类型、数据结构定义如下：

- dmac_channel_number_t：DMA 通道编号。
- dmac_address_increment_t：地址增长方式。
- dmac_burst_trans_length_t：突发传输数量。
- dmac_transfer_width_t：单次传输数据位数。

dmac_channel_number_t

描述

DMA 通道编号。

定义

```
typedef enum _dmac_channel_number
{
    DMAC_CHANNEL0 = 0,
    DMAC_CHANNEL1 = 1,
    DMAC_CHANNEL2 = 2,
    DMAC_CHANNEL3 = 3,
    DMAC_CHANNEL4 = 4,
    DMAC_CHANNEL5 = 5,
```

```
    DMAC_CHANNEL_MAX
} dmac_channel_number_t;
```

成员

成员名称	描述
DMAC_CHANNEL0	DMA 通道 0
DMAC_CHANNEL1	DMA 通道 1
DMAC_CHANNEL2	DMA 通道 2
DMAC_CHANNEL3	DMA 通道 3
DMAC_CHANNEL4	DMA 通道 4
DMAC_CHANNEL5	DMA 通道 5

dmac_address_increment_t

描述

地址增长方式。

定义

```
typedef enum _dmac_address_increment
{
    DMAC_ADDR_INCREMENT = 0x0,
    DMAC_ADDR_NOCHANGE  = 0x1
} dmac_address_increment_t;
```

成员

成员名称	描述
DMAC_ADDR_INCREMENT	地址自动增长
DMAC_ADDR_NOCHANGE	地址不变

dmac_burst_trans_length_t

描述

突发传输数量。

定义

```
typedef enum _dmac_burst_trans_length
{
    DMAC_MSIZE_1   = 0x0,
```

```
    DMAC_MSIZE_4   = 0x1,
    DMAC_MSIZE_8   = 0x2,
    DMAC_MSIZE_16  = 0x3,
    DMAC_MSIZE_32  = 0x4,
    DMAC_MSIZE_64  = 0x5,
    DMAC_MSIZE_128 = 0x6,
    DMAC_MSIZE_256 = 0x7
} dmac_burst_trans_length_t;
```

成员

成员名称	描述
DMAC_MSIZE_1	单次传输数量乘 1
DMAC_MSIZE_4	单次传输数量乘 4
DMAC_MSIZE_8	单次传输数量乘 8
DMAC_MSIZE_16	单次传输数量乘 16
DMAC_MSIZE_32	单次传输数量乘 32
DMAC_MSIZE_64	单次传输数量乘 64
DMAC_MSIZE_128	单次传输数量乘 128
DMAC_MSIZE_256	单次传输数量乘 256

dmac_transfer_width_t

描述

单次传输数据位数。

定义

```
typedef enum _dmac_transfer_width
{
    DMAC_TRANS_WIDTH_8   = 0x0,
    DMAC_TRANS_WIDTH_16  = 0x1,
    DMAC_TRANS_WIDTH_32  = 0x2,
    DMAC_TRANS_WIDTH_64  = 0x3,
    DMAC_TRANS_WIDTH_128 = 0x4,
    DMAC_TRANS_WIDTH_256 = 0x5
} dmac_transfer_width_t;
```

成员

成员名称	描述
DMAC_TRANS_WIDTH_8	单次传输 8 位

成员名称	描述
DMAC_TRANS_WIDTH_16	单次传输 16 位
DMAC_TRANS_WIDTH_32	单次传输 32 位
DMAC_TRANS_WIDTH_64	单次传输 64 位
DMAC_TRANS_WIDTH_128	单次传输 128 位
DMAC_TRANS_WIDTH_256	单次传输 256 位