# PDIUSBD12 DMA 模式应用指南

## 1. 基于协议的 DMA 操作简介

PDIUSBD12 有 6 个端点，其中 2 个控制端点，2 个普通端点和 2 个主端点。主端点支持 DMA 传输。

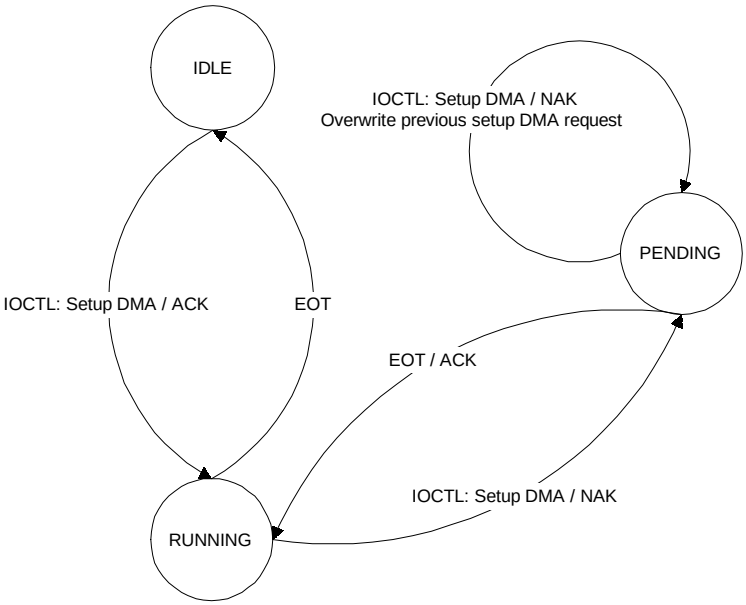
在基于协议的 DMA 操作中，主机应用程序询问器件的固件以通过由控制端点发送的厂商请求建立 DMA 传输。然后在主端点上执行实际的批量数据传输。在 DMA 控制器建立之后主机可向器件传输最多为 64k 字节的数据而不需要任何固件的干预。

一个完整的 DMA 传输要求以下两个步骤：

1) 通过主管道发送一个请求，并允许器件用 DMA 传输方向、起始地址和传输规格对 DMAC 进行编程。
2) 在主端点上发送或接收数据包。

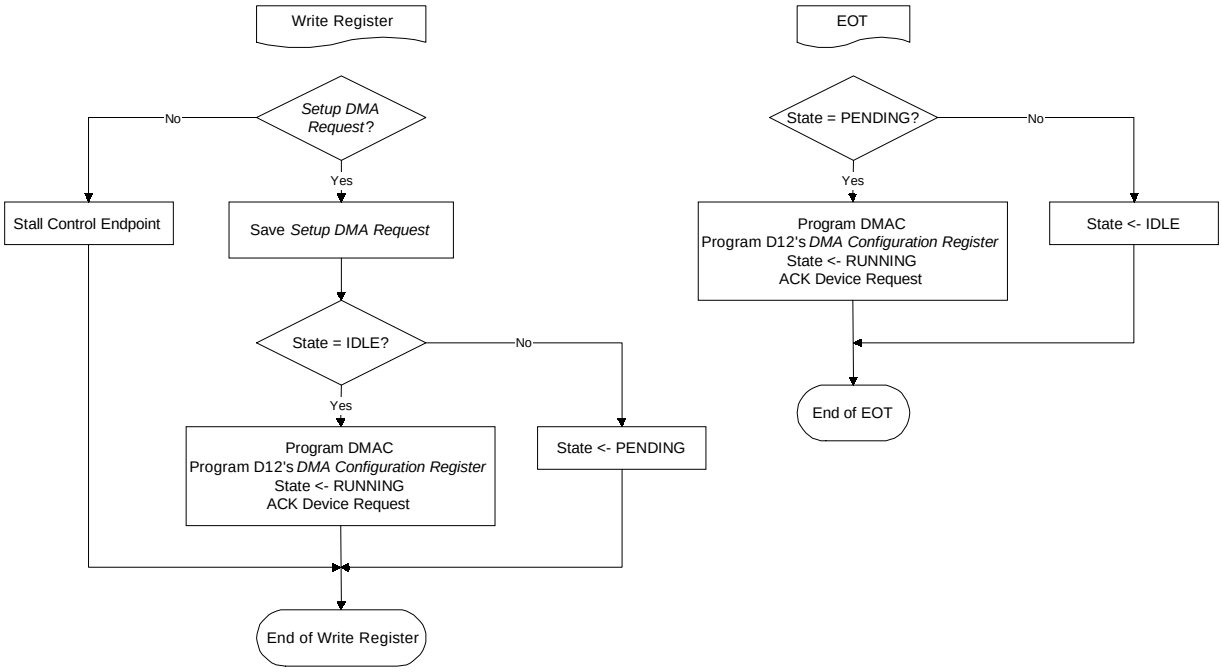
## 2. 器件 DMA 状态

建立 DMA 请求通过控制管道作为厂商请求从主机发出。器件的响应和动作依赖于 DMA 操作的状态。




上图所示为器件中 DMA 的 3 种状态：IDLE，RUNNING 和 PENDING。如果没有运行或推迟 DMA 操作，器件处于 IDLE 状态，那么建立 DMA 请求由 ACK 进行响应。如果器件在 DMA 传输的过程中就是 RUNNING 状态，建立 DMA 请求由 NAK 进行响应并导致器件进入 PENDING 状态。这表示有一个推迟的建立 DMA 请求。如果器件在 PENDING 状态中接收到另一个建立 DMA 请求，新的请求将覆盖旧的请求。

下图为固件处理建立 DMA 请求和 EOT 的程序流程图




**DMA 配置寄存器**

D12 的 DMA 操作由其 DMA 配置寄存器进行控制。该寄存器由“设置 DMA”命令设置。寄存器中不是所有的位都和 DMA 操作有关。位 4（中断脚模式）和位 7（时钟分频系数）控制 D12 的中断源。

下表所示为推荐的寄存器编程汇总：

| 位 | 名称 | DMA 模式 | 非 DMA 模式 |
|---|---|---|---|
| 0&1 | DMA 字符串 | `1`&`1` | 无关 |
| 2 | DMA 使能 | `1` | `0` |
| 3 | DMA 方向 | `1`用于 IN 标志<br>`0`用于 OUT 标志 | 无关 |
| 4 | 自动重装 | `0` | 无关 |
| 5 | 中断脚模式 | `0` | `0` |
| 6 | 端点 4 中断使能 | `0` | `1` |
| 7 | 端点 5 中断使能 | `0` | `1` |

默认情况下，D12 和 DMAC 都不处于自动重装模式。不想让器件的 DMA“自动重新启动”是因为这是一个协议层的操作，也就是说在主机的控制之下。在 EOT，D12 和 DMA 控制器的 DMA 都被禁止。固件需要重新使能以使它们在接收到主机的建立 DMA 请求时重新启动 DMA 传输。

需要注意的是，在 DMA 模式中来自端点 4 和 5 的中断被禁止。这些端点上的中断服务是不必要的，而且使 DMA 传输中有潜在的缺陷。DMA 可以看作是最高级的中断，可以在任意两条 CPU 指令之间执行，甚至在 ISR 中。由于在传输当中 DMA 的状态任何时候都有可能改变，因此任何想要用来检测 DMA 状态的程序都是不可靠的。

以下是一个 IN 标志 DMA 传输编程示例,dma_dir 和 dma_transfer_size 已通过建立 DMA 请求设置。

```
dma_start(dma_dir, MainDmaBuf, dma_transfer_size, 3);
dma.bits.dma_burst = 3;
dma.bits.dma_enable = 1;
dma.bits.dma_direction = dma_dir;
dma.bits.auto_reload = 0;
dma.bits.normal_plus_sof = 0;
dma.bits.endp_4_interrupt_enable = 0;
```

```
dma.bits.endp_5_interrupt_enable = 0;
D12_SetDMA(dma);
```

**建立 DMA 请求**

建立 DMA 请求是一个通过控制管道发送的厂商请求。在 PDIUSBD12 采样固件和小应用程序中，这些是由 IOCTL_WRITE_REGISTER（由微软 Windows 98 DDK 中的静态图像 USB 接口定义）完成的。器件请求描述如下：

| 偏移 | 字段 | 规格 | 值 | 备注 |
|---|---|---|---|---|
| 0 | BmRequestType | 1 | 0x40 | 厂商请求，器件到主机 |
| 1 | Brequest | 1 | 0x0C | IOCTL_WRITE_REGISTERd 固定值 |
| 2 | Wvalue | 2 | 0 | 偏移，设置为零 |
| 4 | Windex | 2 | 0x0471 | 建立 DMA 请求的固定值 |
| 6 | Wlength | 2 | 6 | 建立 DMA 请求的数据长度 |

在器件请求之后，DMA 操作的具体请求在数据阶段发送。采样固件和小应用程序使用专有的定义，如下所示：

| 偏移 | 字段 | 备注 |
|---|---|---|
| 0 | 地址［7：0］ | 已请求 DMA 传输的地址 |
| 1 | 地址［15：8］ | |
| 2 | 地址［23：16］ | |
| 3 | 类型［7：0］ | 传输的规格 |
| 4 | 类型［15：8］ | |
| 5 | 命令 | 位 7：`1`启动 DMA 传输<br>位 0：`1`IN 标志；`0`OUT 标志 |

**主机端编程注意事项**

USB 器件不是决定传输速率的唯一标准。主机端的应用程序在整个系统的性能占有更重要的地位，因为主机一直在控制 USB 的处理。

DMA 传输是一个包括了控制端点和主端点的连续操作。由于下一步操作由前一操作的结果决定，所以协作非常重要。虽然多线程可用于对不同管道进行访问以增加系统的性能，但使用单线程却使处理建立 DMA 请求（IOCTL）和数据传输（写文件/读文件）操作的编程容易许多。

IOCTL_WRITE_REGISTER 和 IOCTL_READ_REGISTER 使用结构 IO_BLOCK 与器件驱动器进行数据交换。下面所示的结构定义是微软静态图像 USB 接口的一部分。

```
typedef struct _IO_BLOCK {
    IN     unsigned uOffset;
    IN     unsigned uLength;
    IN OUT PUCHAR pbyData;
    IN     unsigned uIndex;
} IO_BLOCK, *PIO_BLOCK;
```

IO_REQUEST 结构是一个专有的定义，包含了建立DMA请求的细节。

```
typedef struct _IO_REQUEST {
    unsigned short    uAddressL;
    unsigned char     bAddressH;
    unsigned short    uSize;
    unsigned char     bCommand;
} IO_REQUEST, *PIO_REQUEST;
```

举例程序如下：

```
ioRequest.uAddressL = 0;
```

```
ioRequest.bAddressH = 0;
ioRequest.uSize = transfer_size;
ioRequest.bCommand = 0x80; //start, write
ioBlock.uOffset = 0;
ioBlock.uLength = sizeof(IO_REQUEST);
ioBlock.pbyData = (PUCHAR)&ioRequest;
ioBlock.uIndex = 0x471;
bResult = DeviceIoControl(hDevice,
        IOCTL_WRITE_REGISTERS,
        (PVOID)&ioBlock,
        sizeof(IO_BLOCK),
        NULL,
        0,
        &nBytes,
        NULL);
if (bResult != TRUE) {
        testDlg->MessageBox("Setup DMA request failed!", "Test Error");
        return;
        }
bResult = WriteFile(hFile,
        pcIoBuffer,
        transfer_size,
        &nBytes,
        NULL);
```