8 June 1998

Application Notes

Using PDIUSBD12 in DMA Mode

Application Notes: Using PDIUSBD12 in DMA Mode

1. Introduction to Protocol Based DMA Operation

PDIUSBD12 has 6 endpoints, 2 control endpoints, 2 Generic endpoints and 2 Main endpoints. The Main endpoints support DMA transfer.

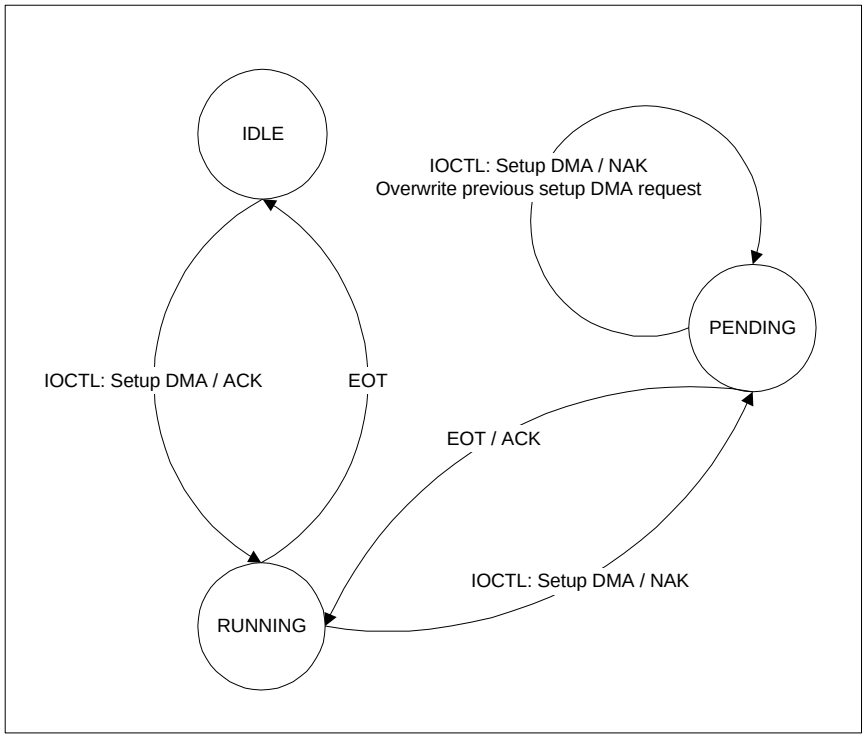
In the protocol based DMA operation, the host application first ask the device's firmware to setup DMA transfer using vendor request which is sent through control endpoint, then it performs actual bulk data transfer on the Main endpoints. After the setup of DMA controller, the host can transfer up to 64K bytes of data to the device without any firmware intervention.

A complete DMA transfer requires following two steps:

1) Send a *Setup DMA Request* through control pipe, let the device to program the DMAC with DMA transfer direction, start address and size of transfer;
2) Send or received data packets on Main endpoint.

2. Device's DMA States

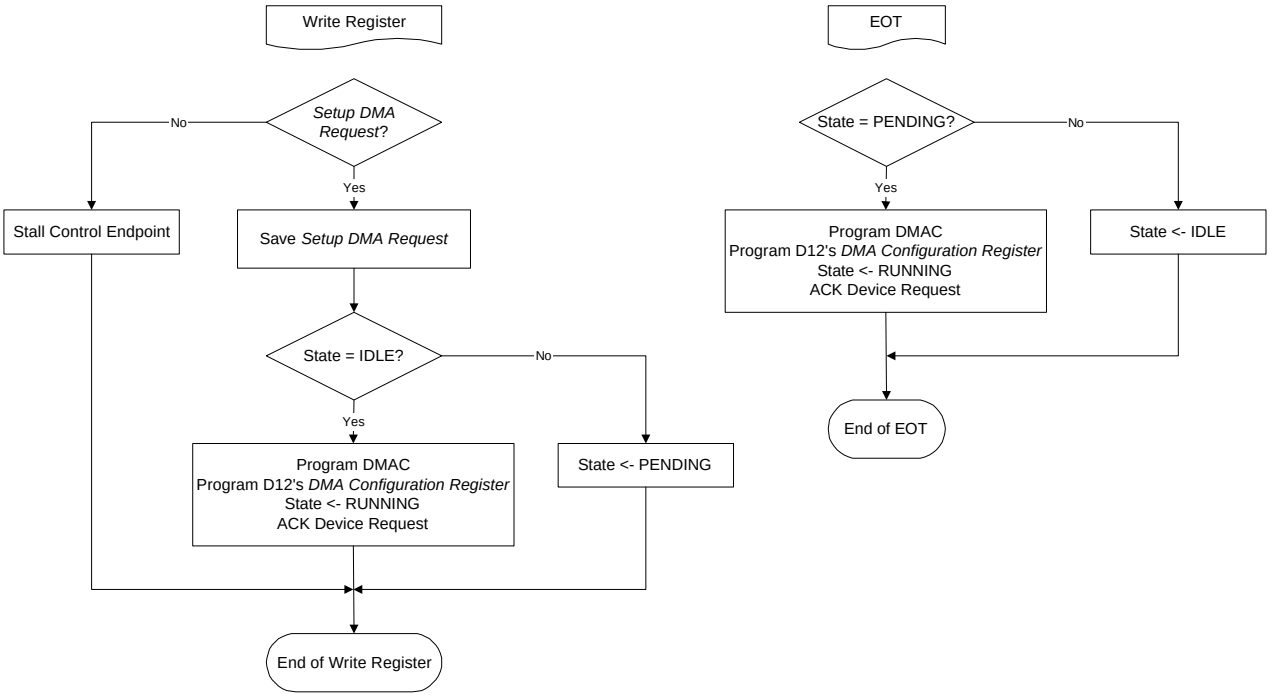
The *Setup DMA Request* is sent from the host as vendor request using control pipe. The device's response and action taken depend on its internal states of DMA operation.



Above DMA state diagram shows 3 DMA states in the device: IDLE, RUNNING and PENDING. When there is no running or pending DMA operation, the device is in IDLE and the *Setup DMA Request* will be responded with ACK. If the device is in the process of a DMA transfer, it is in RUNNING. *Setup DMA Request* received at RUNNING will be responded with NAK and causes the device to enter PENDING, which indicates there is a pending *Setup DMA Request*. If the device receives another *Setup DMA Request* in PENDING, the new request will overwrite the old one.

Application Notes: Using PDIUSBD12 in DMA Mode

Below is the flow chart of the firmware, which handles *Setup DMA Request* and EOT.



3) DMA Configuration Register

The D12's DMA operation is controlled by its *DMA Configuration Register,* which is set by command *Set DMA*. Not all the bits inside the register are related to DMA operation. The bit 4, *Interrupt Pin Mode,* controls D12 sources of interrupt together with bit 7 of *Clock Division Factor*, *SOF-ONLY*.

Below is a summary of recommended register programming:

Bit	Name	DMA Mode	Non-DMA Mode
0 & 1	DMA Burst	'1' & '1'	Don't care
2	DMA Enable	'1'	'0'
3	DMA Direction	'1' for IN token; '0' for OUT token	Don't care
4	Auto Reload	'0'	Don't care
5	Interrupt Pin Mode	'0'	'0'
6	Endpoint 4 Interrupt Enable	'0'	'1'
7	Endpoint 5 Interrupt Enable	'0'	'1'

By default, both of D12 and DMAC are not in auto-reload mode. We do not want the device's DMA "auto-restart" because this is a protocol based operation, that is under host's control. At EOT, both of D12 and DMA controller's DMA will be disabled. The firmware needs to re-enable them to restart DMA transfer upon receiving *Setup DMA Request* from the host.

Please also note that interrupt from endpoints 4 and 5 are disabled in DMA mode. Servicing interrupt on these endpoints is unnecessary and has a potential flaw during DMA transfer. DMA can be treated as highest "interrupt" that happens between any CPU instructions, even inside

Application Notes: Using PDIUSBD12 in DMA Mode

ISR. Any routines, that may want to be used to check DMA status, are not reliable because the DMA status during transfer may change at any time.

Below is an example of programming IN token DMA transfer, the *dma_dir* and *dma_transfer_size* have been set through *Setup DMA Request*:

```
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);
```

4) Setup DMA Request

Setup DMA request is a vendor request that is sent through control pipe. In PDIUSBD12 sample firmware and test applet, this is done by IOCTL_WRITE_REGISTER, which is defined by Microsoft Still Image USB Interface in Windows 98 DDK. Below is the device request description:

Offset	Field	Size	Value	Comments
0	bmRequestType	1	**0x40**	Vendor request, device to host
1	bRequest	1	**0x0C**	Fixed value for IOCTL_WRITE_REGISTER
2	wValue	2	**0**	Offset, set to zero
4	wIndex	2	**0x0471**	Fixed value of *Setup DMA Request*
6	wLength	2	**6**	Data length of *Setup DMA Request*

The details of requested DMA operation are sent in the data phase after the device request. The sample firmware and test applet use a proprietary definition which is shown below:

Offset	Field	Comments
0	Address [7:0]	The start address of requested DMA transfer.
1	Address [15:8]	
2	Address [23:16]	
3	Size [7:0]	The size of transfer.
4	Size [15:8]	
5	Command	Bit 7: ‘1’ start DMA transfer Bit 0: ‘1’ IN token; ‘0’ OUT token

5) Host Side Programming Considerations

The USB device is not the only criteria, which decides the transfer rate. The performance of host side application plays a more important role in overall system performance because host always controls USB transactions.

The DMA transfer is a sequential operation that involves both control endpoint and Main endpoint. Co-operation is important because next step of operation is determined by the result of

Application Notes: Using PDIUSBD12 in DMA Mode

last operation. While multithreads can be used to access different pipes to increase system performance, it makes programming much easier to process *Setup DMA Request* (IOCTL) and data transfer (WriteFile/ReadFile) operation on Main endpoints with a single thread.

IOCTL_WRITE_REGISTER and IOCTL_READ_REGISTER use structure IO_BLOCK to exchange data with the device driver. Below structure definition is part of Microsoft Still Image USB Interface.

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

IO_REQUEST structure is a proprietary definition that contains details of the *Setup DMA Request.*

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

See the sample code below:

```
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);
```