Philips Semiconductors
Interconnectivity

23 September 1998

Firmware Programming Guide for PDIUSBD12

Version 1.0

Philips Semiconductors - Asia Product Innovation Centre
Visit http://www.flexiusb.com

This is a legal agreement between you (either an individual or an entity) and Philips Semiconductors. By accepting this product, you indicate your agreement to the disclaimer specified as follows:

DISCLAIMER

PRODUCT IS DEEMED ACCEPTED BY RECIPIENT. THE PRODUCT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, PHILIPS SEMICONDUCTORS FURTHER DISCLAIMS ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANT ABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE PRODUCT AND DOCUMENTATION REMAINS WITH THE RECIPIENT. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL PHILIPS SEMICONDUCTORS OR ITS SUPPLIERS BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE, OR OTHER DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THIS AGREEMENT OR THE USE OF OR INABILITY TO USE THE PRODUCT, EVEN IF PHILIPS SEMICONDUCTORS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Contents

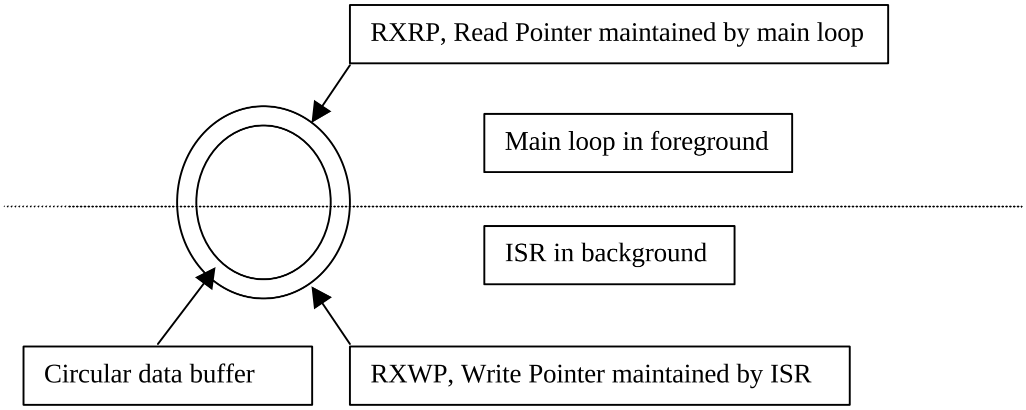
Firmware Programming Guide for PDIUSBD12

1. Introduction

PDIUSBD12 is a high-speed USB interface device with parallel bus and local DMA transfer capability. The objective of the recommended firmware design is to enable PDIUSBD12 to achieve the fastest transfer rate over USB.

Peripheral devices such as the printer, scanner, external mass storage, and digital camera can use PDIUSBD12 in transferring data over USB. The CPUs in these devices are very busy in handling many tasks like device control and data and image processing. The firmware of PDIUSBD12 is designed to be fully interrupt-driven. While CPU is doing its foreground task, the USB transfer is being handled in the background. This assures best transfer rate and better software structure and also simplifies programming and debugging.

The data exchange between the background ISR (Interrupt Service Routine) and the foreground Main Loop is achieved by event flags and data buffers. For example, the PDIUSBD12 Main bulk out endpoint can use a circular data buffer. When PDIUSBD12 receives a data packet from USB, an interrupt request is generated to the CPU and the CPU will service ISR immediately. Inside the ISR, the firmware moves the data packet to the circular buffer from PDIUSBD12's internal buffer and then clears the PDIUSBD12's internal buffer to enable PDIUSBD12 to receive new packet. The CPU can continue its current foreground task until completion, e.g. printing current page. Then it returns to the main loop, checks the circular buffers for new data, and starts another foreground task.


RXRP, Read Pointer maintained by main loop
Main loop in foreground
ISR in background
Circular data buffer
RXWP, Write Pointer maintained by ISR

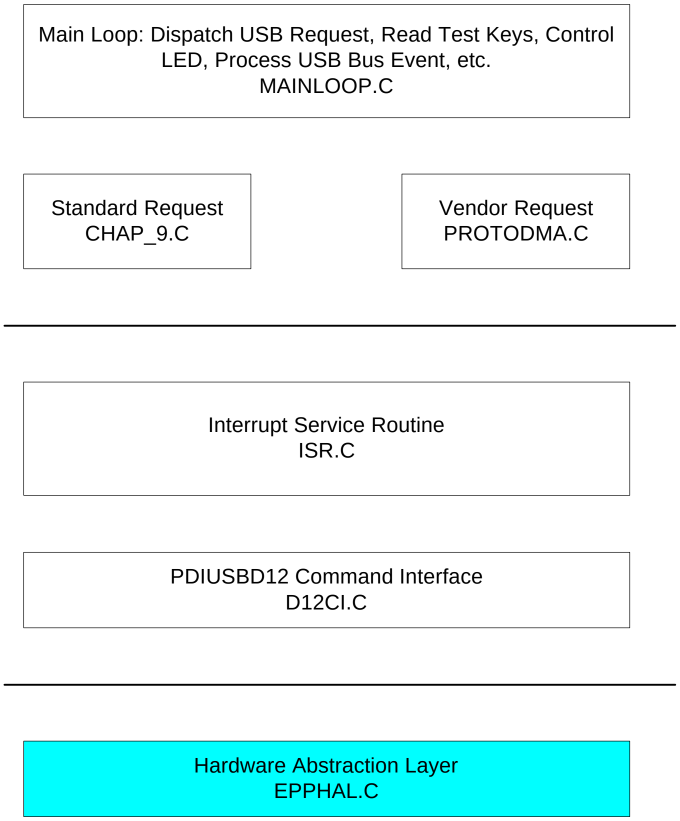
With this structure, the Main Loop does not care whether the data source is from USB, serial port, or parallel port. The Main Loop only checks the circular buffer for new data to be processed. This concept is very important. Thus, the Main Loop program can target on data processing and the ISR can do the job of data transfer at the fastest speed possible.

Similarly, the control endpoint uses the same concept in data packet handling. The ISR receives and stores control transfers in data buffers and sets the corresponding flag registers. The main loop will dispatch the request to protocol handling routines. Since all the standard device, class, and vendor requests are processed in protocol handling routines, ISR can maintain its efficiency. Also, when new request is added, only modification at the protocol level is needed.

2. Architecture

2.1 Firmware Structure

The firmware for the evaluation board consists of 6 building blocks. They are as follows:


Main Loop: Dispatch USB Request, Read Test Keys, Control LED, Process USB Bus Event, etc.
MAINLOOP.C
Standard Request
CHAP_9.C
Vendor Request
PROTODMA.C
Interrupt Service Routine
ISR.C
PDIUSBD12 Command Interface
D12CI.C
Hardware Abstraction Layer
EPPHAL.C

2.1.1 Hardware Abstraction Layer - EPPHAL.C

This is the lowest layer code in the firmware, which performs hardware dependent I/O access to PDIUSBD12, as well as Evaluation Board hardware. When porting the firmware to other CPU platforms, this part of code always needs modifications or additions.

2.1.2 PDIUSBD12 Command Interface - D12CI.C

To further simplify programming with PDIUSBD12, the firmware defines a set of command interfaces, which encapsulate all the functions used to access PDIUSBD12.

2.1.3 Interrupt Service Routine - ISR.C

This part of the code handles interrupt generated by PDIUSBD12. It retrieves data from PDIUSBD12's internal FIFO to CPU memory, and set up proper event flags to inform Main Loop program for processing.

2.1.4 Main Loop - MAINLOOP.C

The Main Loop checks the event flags and passes to appropriate subroutine for further processing. It also contains the code for human interface, such as LED and key scan.

2.1.5 Protocol Layer - CHAP_9.C, PROTODMA.C

The Protocol layer handles standard USB device requests, as well as specific vendor requests such as DMA and TWAIN.

2.2 Porting the Firmware to Other CPU Platform

Below table shows the modifications on building blocks need to be done. There are two levels of porting. First is Chapter 9 only, which is only make the firmware to pass enumeration by supporting standard USB requests. The second level is full product development, this will involve product specific firmware code.

File Name	Chapter 9 Only	Product Level
EPPHAL.C	Port to hardware specific.	Port to hardware specific.
D12CI.C	No change.	No change.
CHAP_9.C	No change.	Product specific USB descriptors.
PROTODMA.C	No change.	Add vendor request supports if necessary.
ISR.C	No change.	Add product specific processing on Generic and Main endpoints.
MAINLOOP.C	Depends on the CPU and system, ports, timer and interrupt initialization need to be rewritten.	Add product specific Main Loop processing.

There are CPU and compiler dependent pre-definitions in MAINLOOP.H:

```
#ifdef __C51__
        #define SWAP(x)   ((((x) & 0xFF) << 8) | (((x) >> 8) & 0xFF))
#else
        #define SWAP(x)   (x)
        #define code
        #define idata
#endif
```

Note the "SWAP" is necessary for micro-controllers, which are big endian format, such as 8031. The "code" and "idata" is only necessary when using 8031 and Keil C compiler.

2.3 Using the Firmware in Polling Mode

It's very easy to use the firmware in polling mode. Inside the Main Loop, add following code:

```
if(interrupt_pin_low)
        fn_usb_isr();
```

Normally ISR is initiated by hardware. In polling mode, the Main Loop detects interrupt pin's state, and invokes ISR if necessary.

3. Hardware Abstraction Layer

This layer contains the lowest layer functions that need to be changed on different CPU platforms.

```
void outportb(unsigned char port, unsigned char val);
void inportb(unsigned char port);
void dma_start(PIO_REQUEST pio)
```

All I/O access to PDIUSBD12 should be implemented by the first two functions above (outportb and inportb). As for the last function, it is meant for implementing the "EPP DMA" function. The latter is for setting the EPP page address and CPLD counter. This type of implementation can allow the system to be platform independent, which means that the application's architecture can be used for platform other than 8051 or the PC.

For USB_EPP evaluation board, the dma_start() functions calls the 2 function below, which are not necessary to be implemented in target application.

```
void eppAwrite(unsigned char A_data);
void program_cpld(unsigned short uSize, unsigned char bCommand);
```

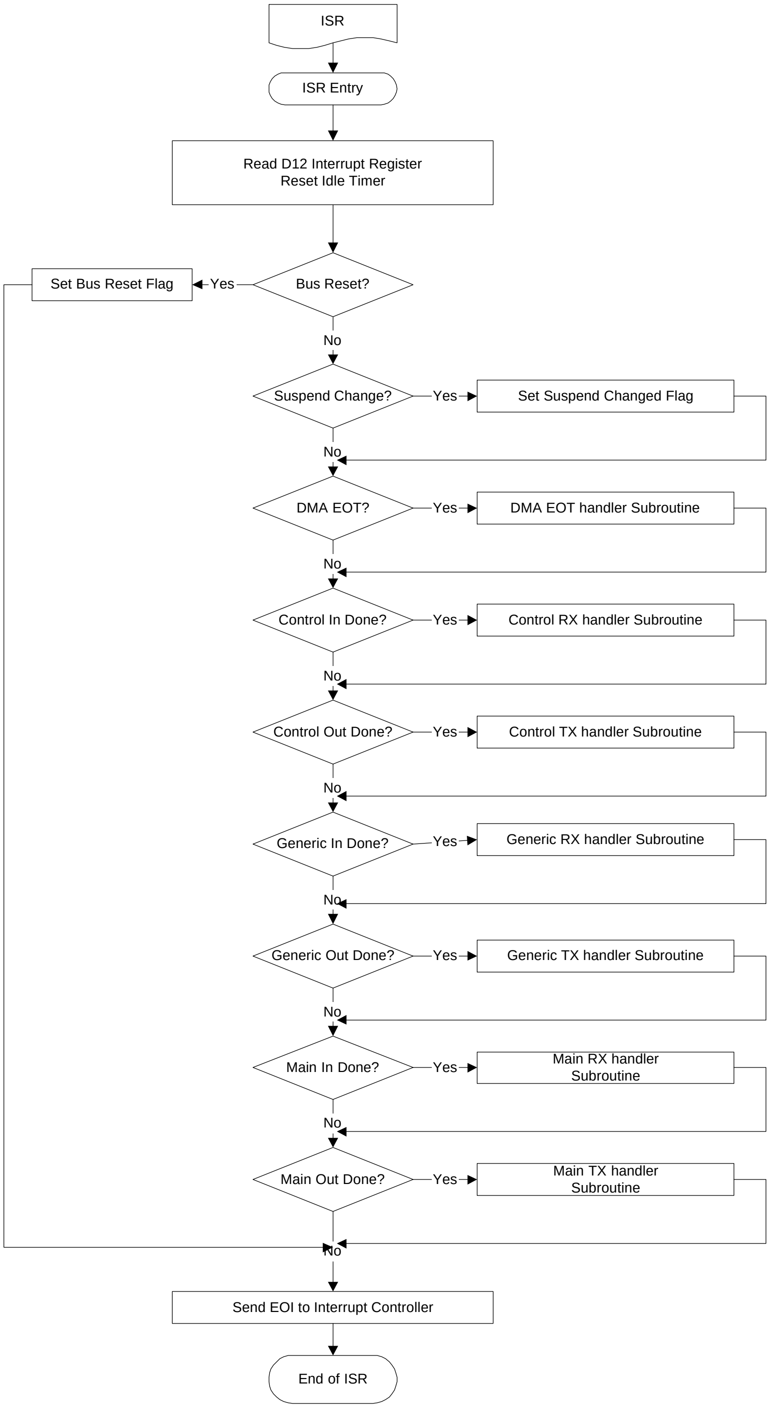
4. PDIUSBD12 Command Interface

The following functions are defined as PDIUSBD12 command interface to simplify device programming. They are simple implementations of the PDIUSBD12 command set, which is defined in the data sheet.

```
void D12_SetAddressEnable(unsigned char bAddress, unsigned char bEnable);
void D12_SetEndpointEnable(unsigned char bEnable);
void D12_SetMode(unsigned char bConfig, unsigned char bClkDiv);
void D12_SetDMA(unsigned char bMode);
unsigned short D12_ReadInterruptRegister(void);
unsigned char D12_SelectEndpoint(unsigned char bEndp);
unsigned char D12_ReadLastTransactionStatus(unsigned char bEndp);
void D12_SetEndpointStatus(unsigned char bEndp, unsigned char bStalled);
void D12_SendResume(void);
unsigned short D12_ReadCurrentFrameNumber(void);

unsigned char D12_ReadEndpoint(unsigned char endp, unsigned char * buf, unsigned char len);
unsigned char D12_WriteEndpoint(unsigned char endp, unsigned char * buf, unsigned char len);
void D12_AcknowledgeEndpoint(unsigned char endp);
```

5. Interrupt Service Routine

The PDIUSBD12 firmware is fully interrupt driven. The flow of ISR is straightforward and this is shown below.


ISR
ISR Entry
Read D12 Interrupt Register
Reset Idle Timer
Bus Reset?
Yes
Set Bus Reset Flag
No
Suspend Change?
Yes
Set Suspend Changed Flag
No
DMA EOT?
Yes
DMA EOT handler Subroutine
No
Control In Done?
Yes
Control RX handler Subroutine
No
Control Out Done?
Yes
Control TX handler Subroutine
No
Generic In Done?
Yes
Generic RX handler Subroutine
No
Generic Out Done?
Yes
Generic TX handler Subroutine
No
Main In Done?
Yes
Main RX handler Subroutine
No
Main Out Done?
Yes
Main TX handler Subroutine
No
Send EOI to Interrupt Controller
End of ISR

At the entrance of ISR, the firmware uses D12_ReadInterruptRegister() to decide the source of interrupt and then to dispatch it to the appropriate subroutines for processing.

The ISR communicates with the foreground Main Loop through event flags "EPPFLAGS" and data buffers "CONTROL_XFER".

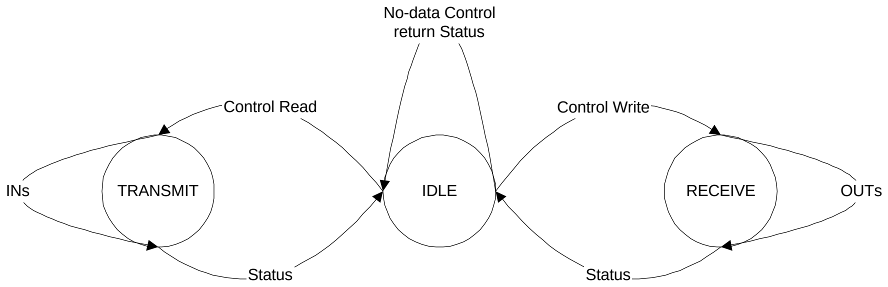
```
typedef union _epp_flags
{
        struct _flags
        {
                unsigned char timer             : 1;
                unsigned char bus_reset          : 1;
                unsigned char suspend            : 1;
                unsigned char setup_packet     : 1;
                unsigned char remote_wakeup    : 1;
                unsigned char in_isr             : 1;
                unsigned char control_state     : 2;

                unsigned char configuration     : 1;
                unsigned char verbose           : 1;
                unsigned char ep1_rxdone        : 1;
                unsigned char setup_dma         : 1;
                unsigned char dma_state         : 2;

        } bits;
        unsigned short value;
} EPPFLAGS;

typedef struct _device_request
{
        unsigned char bmRequestType;
        unsigned char bRequest;
        unsigned short wValue;
        unsigned short wIndex;
        unsigned short wLength;
} DEVICE_REQUEST;

typedef struct _control_xfer
{
        DEVICE_REQUEST DeviceRequest;
        unsigned short wLength;
        unsigned short wCount;
        unsigned char * pData;
        unsigned char dataBuffer[MAX_CONTROLDATA_SIZE];
} CONTROL_XFER;
```

The task splitting between Main Loop and ISR is that ISR collects data from D12 and Main Loop will process the data. The ISR will only inform Main Loop that data is ready for processing when it has collected enough data. For example, in the case of setup packet with OUT data phase, the ISR will store both setup packet and OUT data to buffer "CONTROL_XFER" before it signals "setup_packet" flag to Main Loop. This will reduce unnecessary Main Loop servicing time and also simply Main Loop programming.

5.1 Bus Reset and Suspend Change

Bus reset and suspend does not require special processing within ISR. ISR either sets the bus_reset flag or suspends the bit in EPPFLAGS and exit.

5.2 Control Endpoint Handler

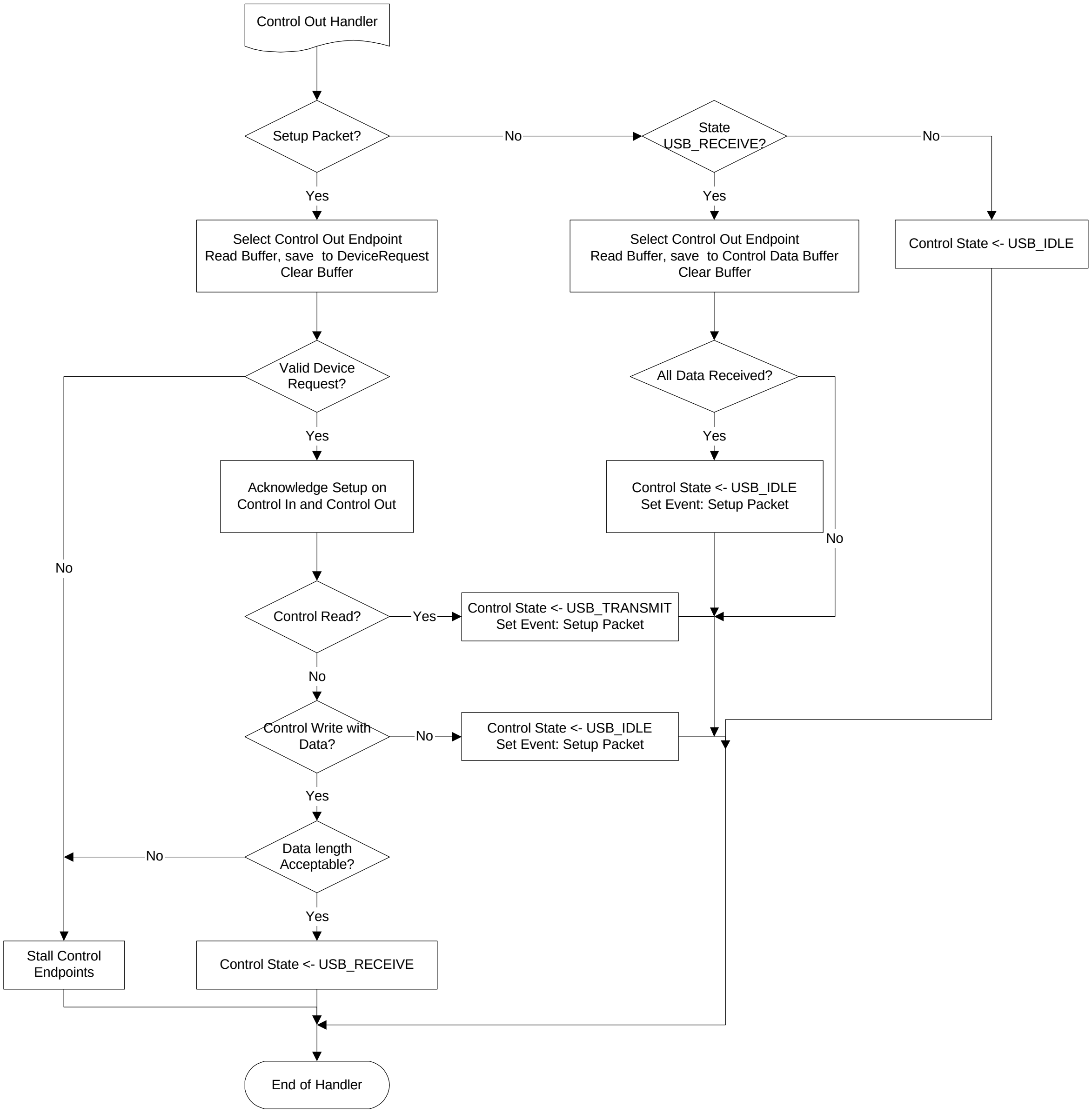


Control transfer always begins with the SETUP stage and then followed later by an optional DATA stage. It then ends with the STATUS stage. The diagram below shows the various states of transitions on the Control endpoints. The firmware uses these 3 states to handle Control transfer correctly.

The flowchart below shows the Control Out handler. To illustrate, an example of the Host requesting a standard device request say "Get Descriptor()".

When the setup packet is received by the USB device D12, this device will generate an interrupt to the MCU. The microcontroller will need to service this interrupt by reading D12 interrupt register to determine whether the packet is send to Control Endpoint or the Generic Endpoint. If the packet is for the Control Endpoint, MCU will need to further determine whether the data is a setup packet through reading the D12 Read Last Transaction Status Register. For the Get_Descriptor device request, the first packet will have to be the setup packet.

From the flowchart above, MCU will need to extract the content of the setup packet through Select Control Out Endpoint to determine whether this endpoint is full or empty. If the control endpoint is full, MCU will then read the content from the buffer and save the content to its memory. After that, it will validate the host device's request from the memory. If it is a valid request, MCU must send the Acknowledge Setup command to the Control Out endpoint to re-enable the next setup phase.


Control Out Handler
Setup Packet?
No
State USB_RECEIVE?
No
Yes
Yes
Select Control Out Endpoint Read Buffer, save to DeviceRequest Clear Buffer
Select Control Out Endpoint Read Buffer, save to Control Data Buffer Clear Buffer
Control State <- USB_IDLE
Valid Device Request?
All Data Received?
Yes
Yes
Acknowledge Setup on Control In and Control Out
Control State <- USB_IDLE Set Event: Setup Packet
No
No
Control Read?
Yes
Control State <- USB_TRANSMIT Set Event: Setup Packet
No
Control Write with Data?
No
Control State <- USB_IDLE Set Event: Setup Packet
Yes
Data length Acceptable?
No
Yes
Stall Control Endpoints
Control State <- USB_RECEIVE
End of Handler

Firmware Programming Guide for PDIUSBD12

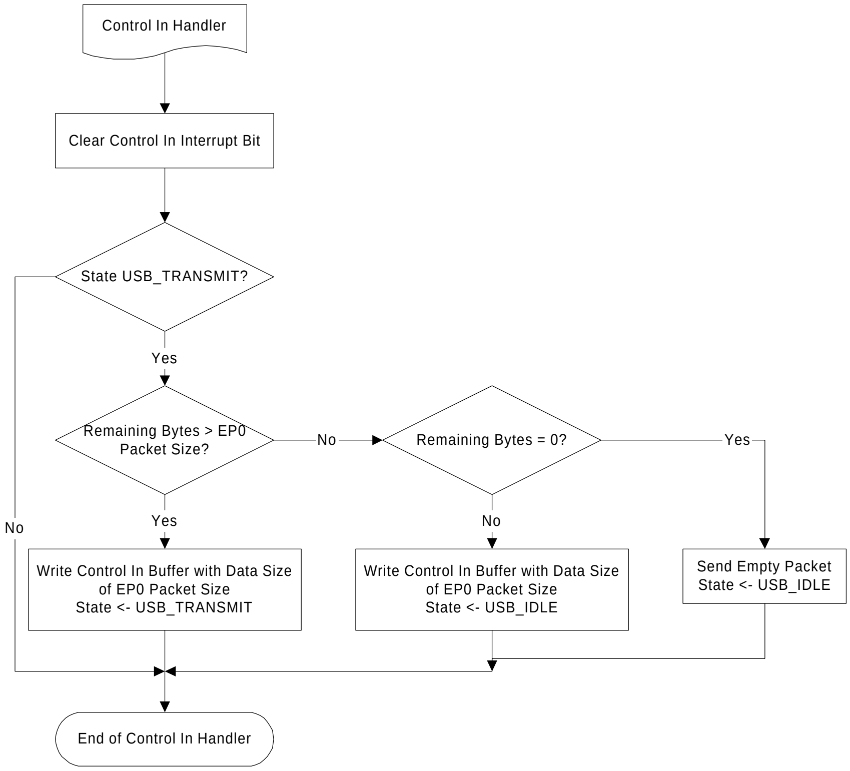
Next, MCU will need to verify whether the control transfer is a Control Read/Write. This can be achieved through reading the 8th bit of the bmRequestType from the setup packet. In our case, the control transfer is a Control Read type, that is, the device will need to send data packet back to the host in the coming data phase. MCU will need to set a flag to indicate that the USB device is now in the transmit mode, that is, ready to send its data upon the host's request.

After the Setup stage is finished, the host will execute the data phase. D12 will expect to receive the Control_In packet. The process is shown in the next flowchart, "Control_In Handler". Again, MCU will first need to clear the Control_In interrupt Bit on the D12 by reading its Read Last Transaction Register. Then, MCU will proceed to send the data packet after verifying that D12 is in the appropriate mode, that is, the Transmit mode.

As D12 control endpoint has only 16 bytes FIFO, MCU will have to control the amount of data during the transmitting phase if the requested length is more than 16 bytes. As indicated on the flowchart, MCU will have to check its current and remaining size of the data to be sent to the host. If the remaining bytes are greater than 16 bytes, MCU will send the first 16 bytes and then subtract the reference length (requested length) by 16.

When the next Control_In token comes, MCU will determine whether the remaining bytes is zero. If there is no more data to send, MCU will need to send an empty packet to inform the host that there will be no more data to be sent over.

If the setup packet is Set_Descriptor() request, then the control transfer in the setup packet will indicate that it is a Control Write type. After executing the procedure for the Set_Descriptor request, MCU will wait for the data phase. The host will send a Control_Out token and MCU will have to extract the data from the D12 buffer. The flow will now be on the right side of the Control_Out Handler flow sequence. MCU will have to first verify whether D12 is in the USB_Receive mode. Then, MCU will have to verify whether the buffer is full by checking the Select Control_Out Endpoint and read the data out from the buffer.

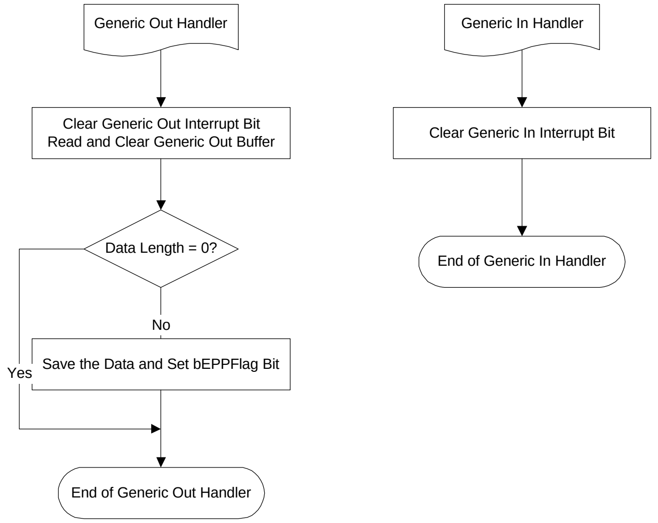

Control In Handler
Clear Control In Interrupt Bit
State USB_TRANSMIT?
Yes
No
Remaining Bytes > EP0 Packet Size?
No
Remaining Bytes = 0?
Yes
Yes
No
Write Control In Buffer with Data Size of EP0 Packet Size State <- USB_TRANSMIT
Write Control In Buffer with Data Size of EP0 Packet Size State <- USB_IDLE
Send Empty Packet State <- USB_IDLE
End of Control In Handler

The flowchart above shows the Control In handler.

5.3 Generic Endpoint Handler

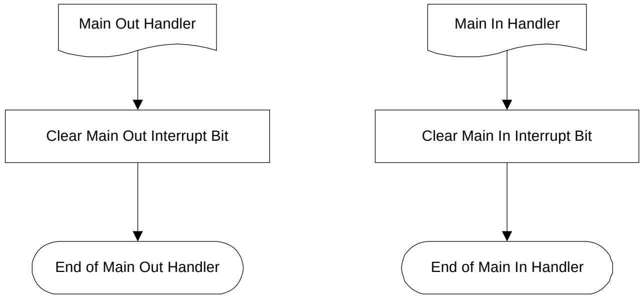
The Generic Out Endpoint has been configured to receive data packet from the host, which interprets it as LED Control data. When MCU receives the Generic_Out token from the host (identified through the Read Interrupt Register), the D12 interrupt bit has to be cleared. The Select Endpoint will clear the Generic_Out buffer and then the buffer sequence. Next, MCU will need to verify the data length and interpret the data.

For Generic_In Endpoint, this endpoint has been configured to send the information of the button activity as a byte to the host.


Generic Out Handler
Clear Generic Out Interrupt Bit
Read and Clear Generic Out Buffer
Data Length = 0?
No
Yes
Save the Data and Set bEPPFlag Bit
End of Generic Out Handler
Generic In Handler
Clear Generic In Interrupt Bit
End of Generic In Handler

5.4 Main Endpoint Handler

Since the Main In/Out Endpoints are configured to the DMA mode and the interrupts are disabled for these endpoints, no ISR services are normally required for these endpoints. However, for safer code implementation, we have put in place clear interrupt routine here.



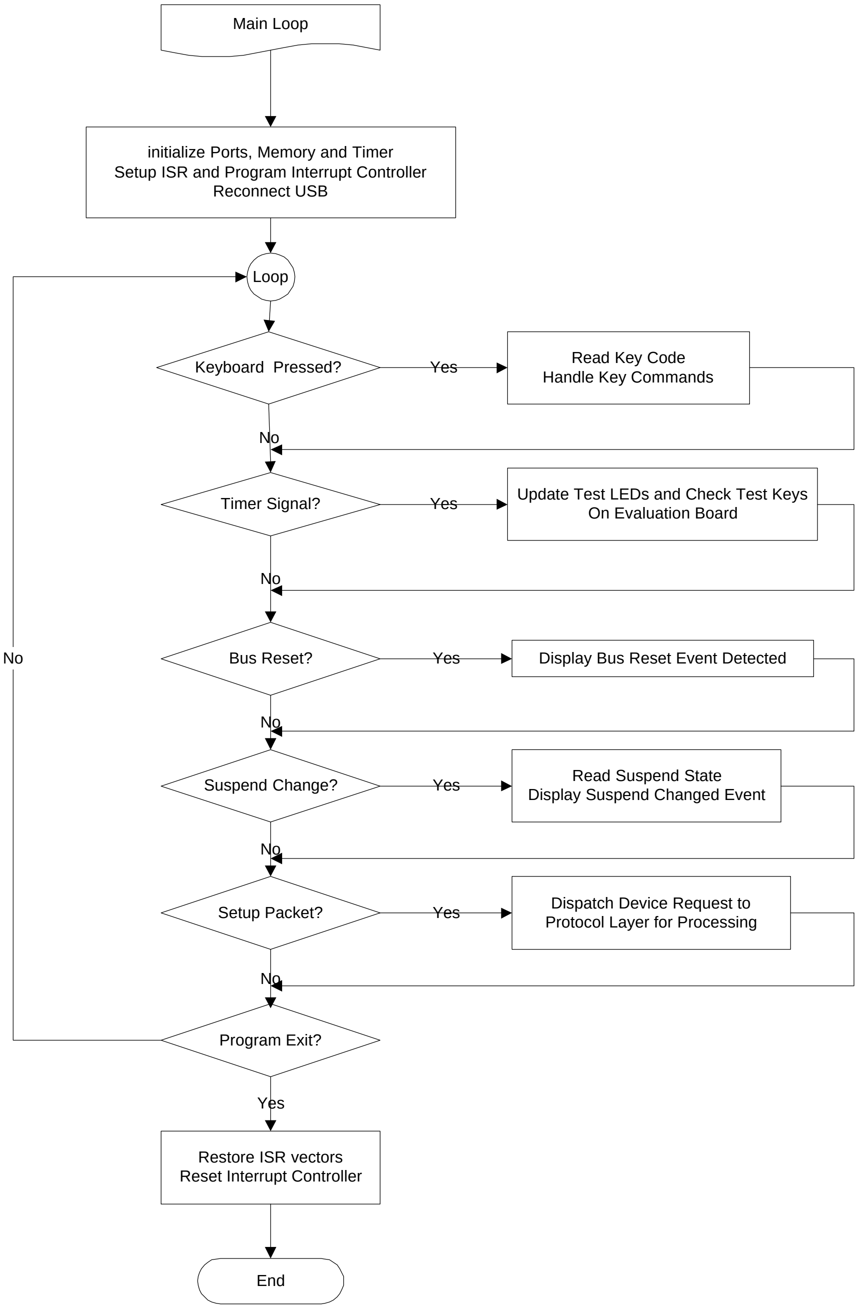
5.5 EOT Handler

For more information on EOT handler, please refer to the section "DMA Support".

6. Main Loop

Once powered up, MCU will need to initialize all its ports, memory, timer, and interrupt service routine handler. After that, MCU will reconnect USB, which involves setting the Soft_Connect register to ON. This procedure is important because it ensures that D12 will not operate before MCU is ready to serve D12.
In the main loop routine, MCU will poll for any activity on the keyboard. If any one of the specific keys is pressed, the handle key commands will execute the routine and then return to the main loop. This routine is added for debugging purposes only. A 1mSec timer is programmed to activate the routine to check for any button pressed on the evaluation board.

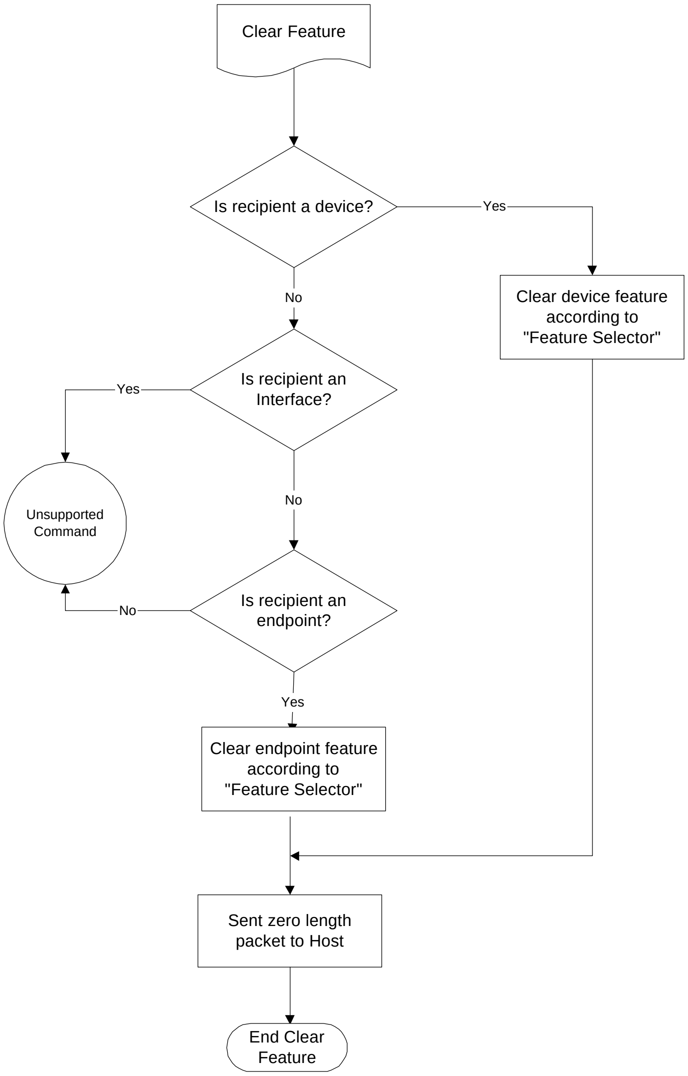
When the polling reaches the check Setup packet, it verifies the setup flag set previously by the interrupt service routine. If the setup flag is set, it will dispatch a device request to the protocol layer for processing. The flowchart above shows the main program executing on the foreground.


Main Loop
initialize Ports, Memory and Timer
Setup ISR and Program Interrupt Controller
Reconnect USB
Loop
Keyboard Pressed?
Yes
Read Key Code
Handle Key Commands
No
Timer Signal?
Yes
Update Test LEDs and Check Test Keys
On Evaluation Board
No
Bus Reset?
Yes
Display Bus Reset Event Detected
No
Suspend Change?
Yes
Read Suspend State
Display Suspend Changed Event
No
Setup Packet?
Yes
Dispatch Device Request to
Protocol Layer for Processing
No
Program Exit?
No
Yes
Restore ISR vectors
Reset Interrupt Controller
End

7. Chapter 9 Protocol

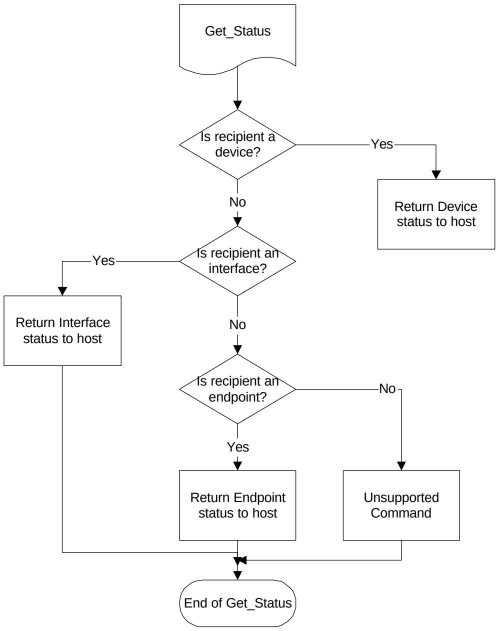
7.1 Clear Feature Request

In the Clear feature request, MCU will need to clear or disable a specific feature of the device. In our case, MCU will determine whether the request is meant for the device, interface, or endpoints. Note that there will not be any support if the recipient is an interface. Feature selectors are used when enabling or setting features specific to the device or endpoint such as remote wakeup. When the recipient is a device, MCU will need to disable the remote wakeup function if this function has been enabled. If the recipient is the endpoint, the MCU will have to unstall the specific endpoint through the Set Endpoint Status command.


Clear Feature
Is recipient a device?
Yes
Clear device feature according to "Feature Selector"
No
Is recipient an Interface?
Yes
Unsupported Command
No
Is recipient an endpoint?
No
Yes
Clear endpoint feature according to "Feature Selector"
Sent zero length packet to Host
End Clear Feature

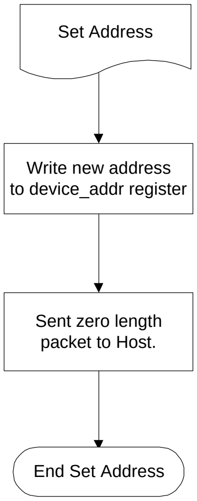
7.2 Get Status Request

In the Get_Status request, MCU will have to return the status for the specific recipient. In our case, MCU will need to determine again the recipient of the request. If the request is to the device, then MCU will have to return the status of the device to the host. For system having remote wakeup and self-power capabilities, the returning data will be 0x0003. If the recipient is an interface, then MCU should return 0x0000 to the host.


Get_Status
Is recipient a device?
Yes
Return Device status to host
No
Is recipient an interface?
Yes
Return Interface status to host
No
Is recipient an endpoint?
No
Yes
Return Endpoint status to host
Unsupported Command
End of Get_Status

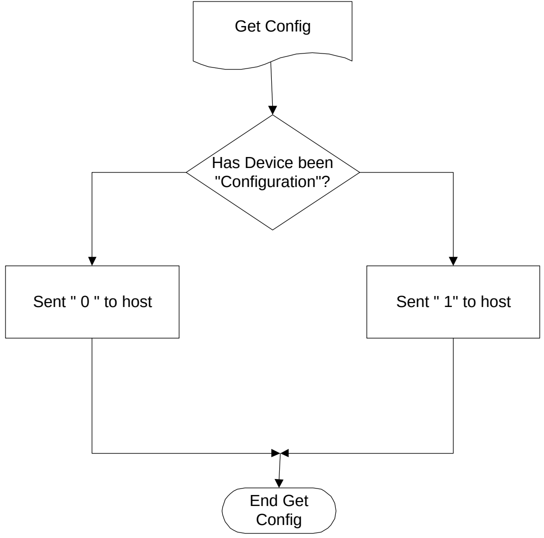
7.3 Set Address Request

In the Set address request, the device will get the new address from the content of the setup packet. Note that this set address request does not have a data phase. Therefore, MCU will need to write a zero length data packet to the host as the acknowledgment phase.


Set Address
Write new address to device_addr register
Sent zero length packet to Host.
End Set Address

7.4 Get Config Request

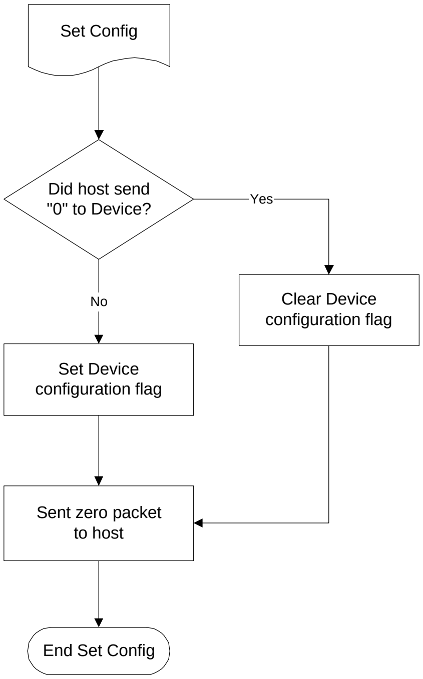
In the Get_config request, the MCU will have to return the current configuration value. Firstly, the MCU will determine whether the device has been configured or not. If the device is not configured, it returns a zero to the host, or else it returns a one if the device is configured.



7.5 Get Descriptor Request

For the Get Descriptor request, MCU must return the specific descriptor if the descriptor exists. Firstly, MCU will determine whether the descriptor type request is for the device or the configuration. It will then send the first 16 bytes of device descriptor if the descriptor type is for device. The reason for controlling the size of returning bytes is due to the fact that the control buffer has only 16 bytes of memory. MCU will need to set a register to indicate the location of the transmitted size.

7.6 Set Config Request


Set Config
Did host send "0" to Device?
Yes
No
Clear Device configuration flag
Set Device configuration flag
Sent zero packet to host
End Set Config

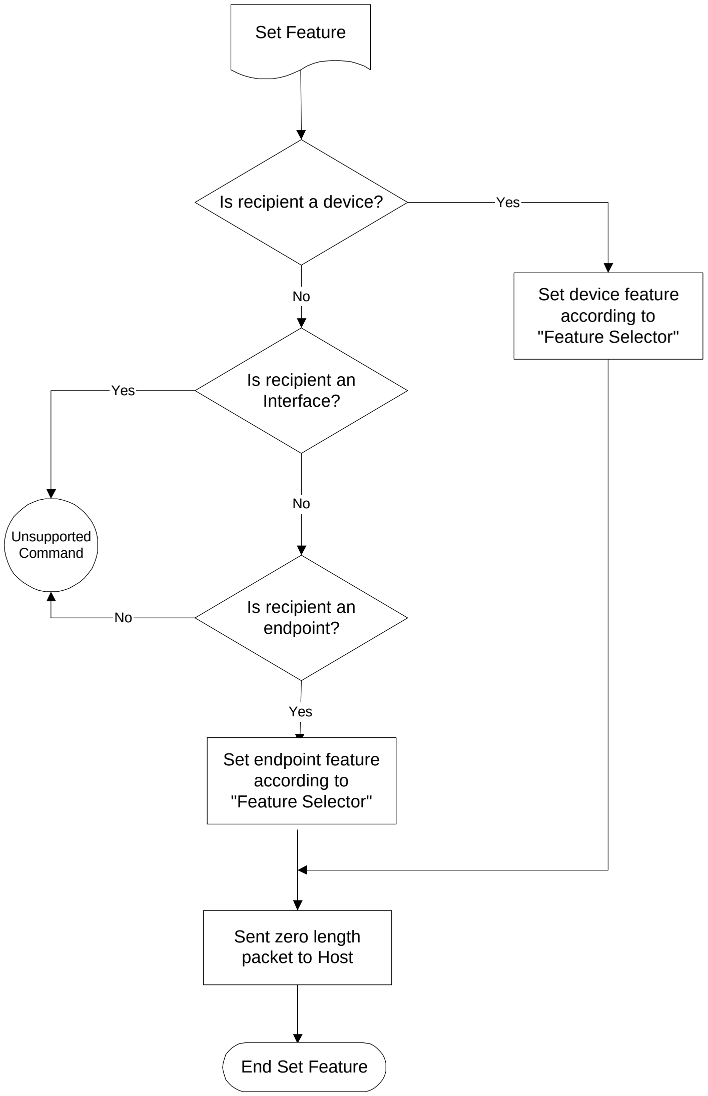
For Set Config request, MCU will determine the configuration value from the setup packet. If the value is zero, then MCU will have to clear the configuration flag on its memory and disable the endpoint. If the value is one, then MCU will need to set the configuration flag. Once the flag is set, MCU will also need to send the zero data packet to the host for the acknowledgment phase.

7.7 Get/Set Interface Request

or Get Interface request, MCU will just need to send one zero data packet to the host as our evaluation board only supports one type of interface. For Set Interface request on our evaluation board, MCU need not do anything except to send one zero data packet to the host as an acknowledgment phase.

7.8 Set Feature Request

Set Feature request is just the opposite of Clear Feature request. If the recipient is a device, then MCU will need to set the device's feature according to the feature selector on the setup packet. Again, there is no support for the Interface recipient. For example, if the feature selector is 0 (which means enabling endpoint), the D12 specific endpoint will have to be stalled through "D12_SetEndpoint status" command.


Set Feature
Is recipient a device?
Yes
No
Set device feature according to "Feature Selector"
Is recipient an Interface?
Yes
No
Unsupported Command
Is recipient an endpoint?
No
Yes
Set endpoint feature according to "Feature Selector"
Sent zero length packet to Host
End Set Feature

8. DMA Support

8.1 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.

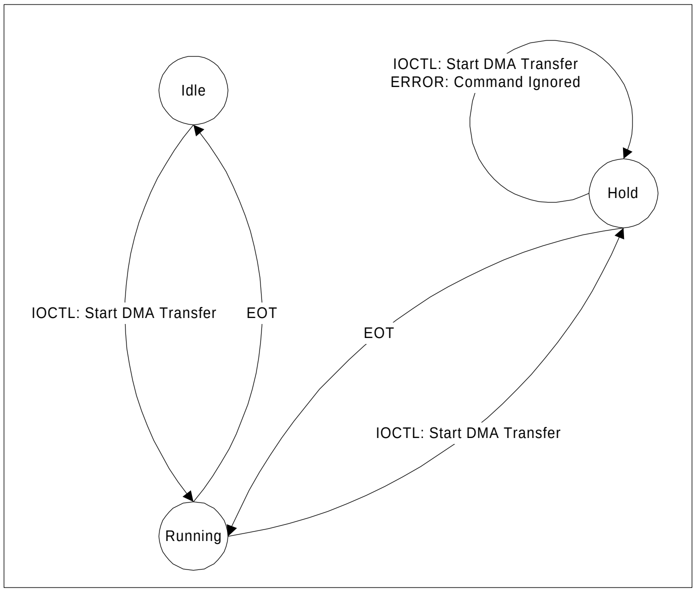
In the protocol based DMA operation, the host application first asks the device's firmware to setup DMA transfer using vendor request, which is sent through the control endpoint. It then performs actual bulk data transfer on the main endpoints. After the setup of the DMA controller, the host can transfer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

A complete DMA transfer requires the following two steps:

1. Send a Setup DMA Request through the control pipe, and allow the device to program the DMAC with DMA transfer direction, start address, and size of transfer.
2. Send or receive data packets on the main endpoint.

8.2 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 depend on its internal states of DMA operation.


Idle
IOCTL: Start DMA Transfer
ERROR: Command Ignored
Hold
IOCTL: Start DMA Transfer
EOT
EOT
IOCTL: Start DMA Transfer
Running

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

8.3 DMA Configuration Register

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

The table below is a summary of the 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 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 the interrupt from endpoints 4 and 5 are disabled in DMA mode. Servicing interrupt on these endpoints is unnecessary and has a potential flaw during the DMA transfer. DMA can be treated as the highest "interrupt" that happens between any CPU instructions, even inside ISR. Any routines that may want to be used to check DMA status are not reliable because the DMA status during the transfer may change at any time.

8.4 Setup DMA Request

Setup DMA request is a vendor request that is sent through control pipe. In PDIUSBD12 sample firmware and Applet, this is done by IOCTL_WRITE_REGISTER, which is defined by Microsoft Still Image USB Interface in Windows 98 DDK. The device's request is described below.

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 requested by the DMA operation are sent in the data phase after the device's request. The sample firmware and 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

8.5 Host Side Programming Considerations

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

The DMA transfer is a sequential operation that involves both control endpoint and main endpoint. Cooperation is important because the next step of the operation is determined by the result of the 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) operations on the 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);
```