User Manual

DPIO2 Device Driver

Linux kernel 2.6.x

x86

Rev 2.4.13.1

Valid for release 2.4.13

VMETRO

CONTENTS

1 INTRODUCTION

This manual describes how to use the DPIO2 Device Driver on systems running Linux kernel version 2.6.x on a x86 platform. It is intended for application programmers who are familiar with the FPDP bus.

The reader is referred to *Front Panel Data Port Specifications* and *User's Manual DPIO2 Digital Parallel Input Output PMC Module* for information about hardware.

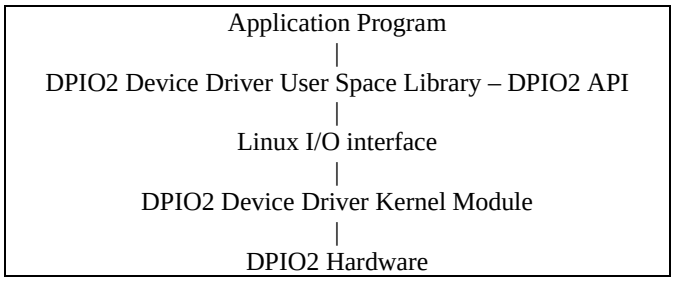
1.1 Overview of DPIO2 Device Driver

1.1.1 Application Programming Interfaces

The DPIO2 Device Driver includes a user space library that provides the Application Programming Interface (API) of the driver.

1.1.2 Software Structure

The DPIO2 Device Driver does not allow an application to access hardware registers directly. There are two software layers involved, as illustrated in the figure below:



1.2 Overview of the remaining chapters

Chapter 2 explains how the DPIO2 Device Driver distribution is installed from the Software Development Kit CD.

Chapter 3 shows how the DPIO2 Device Driver is initialised, and how an application can use it for receiving and transmitting data.

Chapter 4 describes the DPIO2 API of the DPIO2 Device Driver.

2 INSTALLATION

This section describes how to install the DPIO2 Device Driver distribution. The files are delivered on CD-ROM and are installed through a licence-key installation program (included on the CD-ROM).

2.1 Installing the DPIO2 Device Driver files

The DPIO2 Driver is distrubuted on a CD-ROM.
Mount the CD-ROM drive and copy the files from the CD into your hard disk.

2.2 Compile the DPIO2 Device Driver Kernel Module

Before the DPIO2 device driver can be used, the user must compile the DPIO2 driver kernel module for the specific Linux kernel version used on the target computer.

To compile DPIO2 Device Driver Kernel Module on the Target:

1. Go to: dpio2-drv-src/src/x86-linux-2.6.x/kernel
2. Type: make build

To cross compile DPIO2 Device Driver Kernel Module:

1. Go to: dpio2-drv-src/src/x86-linux-2.6.x/kernel
2. Edit the Makefile: Set LINUX_SRC = <path to kernel source>
3. Type: make build ARCH=i386 CROSS_COMPILE=<path to toolchain>

The output file is copied to lib/x86-linux-2.6.x

The DPIO2 Device Driver distribution includes precompiled kernel modules for the following Linux verison:

- Standard kernel version 2.6.9
- Fedora Core 2 version 2.6.8-1.521smp

Change the parameter module="" in the file dpio2_load if us of the one of the precompiled kernel modules.

2.3 Link DPIO2 Device Driver User Space Library with application program

The Makefile (dpio2-drv-src/examples/x86-linux-2.6.x/Makefile) shows how you can link the DPIO2 Devices Driver User Space Module (dpio2-drv.o) to your application program.

2.4 Loading the DPIO2 Device Driver Kernel Module

To load the DPIO2 Device Driver Kernel Module:

1. Login as root or su (super user)
2. Go to: dpio2-drv-src/lib/x86-linux-2.6.x
3. Type: ./dpio2_load

2.5 Removing the DPIO2 Device Driver Kernel Module

To remove the DPIO2 Device Driver Driver Kernel Module:

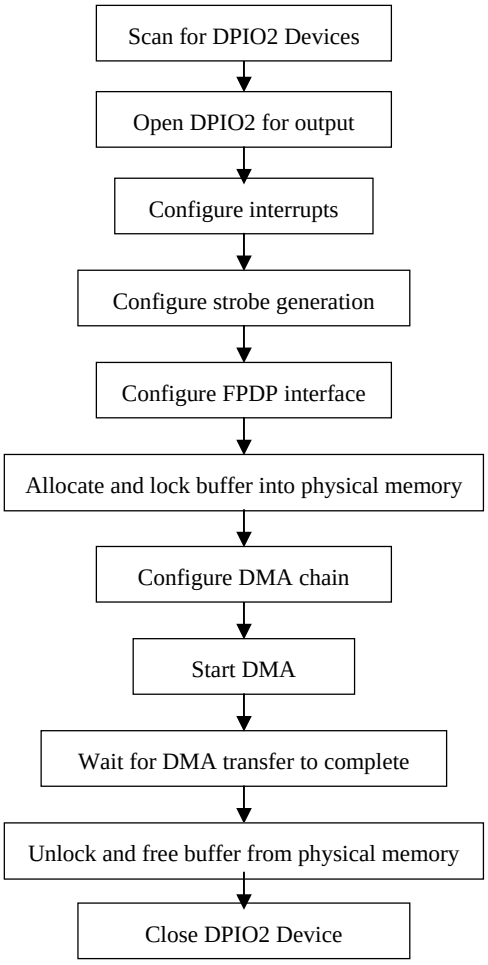
4. Login as root or su (super user)
5. Go to: dpio2-drv-src /lib/x86-linux-2.6.x
6. Type: ./dpio2_remove

3 USING THE DPIO2 DEVICE DRIVER

This chapter explains how an application can use the DPIO2 Device Driver to receive or transmit data.

3.1 Using a DPIO2 module for output

This section shows how an application typically uses a DPIO2 module for output. An overview of the steps involved is shown in the figure below.



3.1.1 Scan for DPIO2 Devices

An application must scan for DPIO2 Device in order to gain access to the DPIO2 modules controlled by the driver:

```
STATUS scanForDevices(int* pNumDpio2Modules)
{
  int           numOfDpio2Modules;
  DPIO2_INFO_T  info;

  numOfDpio2Modules = dpio2Scan(&info);
  if ( numOfDpio2Modules ==  ERROR ) {
    printf("Failed to scan for DPIO2 Devices\n");
    *pNumDpio2Modules = 0;
    return (ERROR);
  }

  if ( numOfDpio2Modules < 1 ) {
    printf("Failed find any DPIO2 Modules\n");
    *pNumDpio2Modules = numOfDpio2Modules;
    return (ERROR);
  }

  *pNumDpio2Modules = numOfDpio2Modules;

  return (OK);
}
```

3.1.2 Open DPIO2 Module for output

An application has to select whether to use a DPIO2 module as input or output.

The example below shows how a DPIO2 device is initialised for output:

```
STATUS openDpio2ForInput(int dpio2DeviceNumber)
{
  STATUS   status;

  status = dpio2Open(dpio2DeviceNumber, DPIO2_OUTPUT);
  if ( status != OK ) {
    printf("Failed to initialize the DPIO2 for input\n");
    return (ERROR);
  }

  return (OK);
}
```

3.1.3 Configuring interrupts

The DPIO2 modules have several interrupt sources. An application can attach one or more actions to each interrupt source before interrupts from the source are enabled.

The example below shows how a callback action is attached to the DMA Done Interrupt and how the interrupt is enabled afterwards:

3.1.4 Configuring strobe generation

DPIO2 modules can generate strobe from two clock sources, a programmable oscillator and a fixed oscillator. Before an application configures a DPIO2 module to generate strobe, it should select which oscillator the strobe should be based on.

The example below shows how to configure a DPIO2 module to generate strobe.

3.1.5 Configuring FPDP interface

Opening a DPIO2 device does not activate the FPDP interface of the corresponding DPIO2 module. This must be done explicitly as shown in the example below:

3.1.6 Allocating and locking buffer into physical memory

Before the DMA controller of the DPIO2 can access a buffer in virtual memory space, the buffer must be locked into physical memory. This is achieved by calling dpio2DMALock() (see 4.4).

The DPIO2_DMA structure used by dpio2DMALock() can only store information about a limited number of physical memory regions, as defined by the constant DPIO2_DMA_PAGES. So if a buffer maps to more physical memory pages than defined by DPIO2_DMA_PAGES, extra storage must be added to the DPIO2_DMA structure by allocating it dynamically as shown in the example below:

```
STATUS allocateAndLockBuffer(UINT32 lengthInBytes,
                             UINT32** ppBuffer,
                             DPIO2_DMA** ppPageInfo)
{
  STATUS   status;
  int      maxNumPages;


  *ppBuffer = (UINT32*) malloc(lengthInBytes);
  if ( *ppBuffer == NULL ) {
```

```
    printf("Failed to allocate %d bytes\n", (int) lengthInBytes);
    return (ERROR);
  }

  /* Determine the maximum number of pages the buffer can be mapped to.
   * One is added at the end of the computation to account for the fact
   * that the buffer may not be aligned to a page boundary. */
  maxNumPages = ((lengthInBytes + PAGE_SIZE - 1) / PAGE_SIZE) + 1;

  /* Allocate structure to store information about all the physical pages
   * the buffer maps to.
   */
  *ppPageInfo = malloc(sizeof(DPIO2_DMA)
                       + maxNumPages * sizeof(DPIO2_DMA_PAGE));
  if ( *ppPageInfo == NULL ) {
    printf("Failed to allocate Page Information structure\n");
    free(*ppBuffer);
    return (ERROR);
  }

  /* Lock the buffer into physical memory.
   */
  (*ppPageInfo)->pUserAddr = *ppBuffer;
  (*ppPageInfo)->dwBytes = lengthInBytes;
  (*ppPageInfo)->dwPages = maxNumPages;
  status = dpio2DMALock(*ppPageInfo);
  if ( status != OK ) {
    printf("Failed to lock buffer into physical memory\n");
    free(*ppPageInfo);
    free(*ppBuffer);
    return (ERROR);
  }

  return (OK);
}
```

3.1.7 Configuring DMA chain

On a DPIO2 module, DMA transfers are controlled by DMA descriptors, which can be chained together. Each DMA descriptor has a number of attributes whose values define how to transfer a block of data (see 4.6.6.1 for a description of these attributes).

The example below shows how a DMA chain can be built.

```
STATUS configureDmaChain(int dpio2DeviceNumber, DPIO2_DMA* pPageInfo)
{
  STATUS   status;
  UINT32   iPage;
  DPIO2_DMA_DESC   dmaDescriptor;


  if ( ( pPageInfo->dwBytes & 0x00000003 ) != 0 ) {
    printf("Length must be multiple of 4 bytes\n");
    return (ERROR);
  } else if ( ( pPageInfo->Page[0].pPhysicalAddr & 0x00000003) != 0 ) {
    printf("PCI Address must be aligned to 4 bytes boundary\n");
    return (ERROR);
  }

  for (iPage = 0; iPage < (pPageInfo->dwPages - 1); iPage++) {
    dmaDescriptor.descriptorId        = iPage;
    dmaDescriptor.nextDescriptorId    = iPage + 1;
    dmaDescriptor.pciAddress
      = (UINT32) pPageInfo->Page[iPage].pPhysicalAddr;
    dmaDescriptor.blockSizeInBytes
      = pPageInfo->Page[iPage].dwBytes;
```

```
    dmaDescriptor.lastBlockInChain    = FALSE;
    dmaDescriptor.useD64              = FALSE;
    dmaDescriptor.notEndOfFrame       = FALSE;
    dmaDescriptor.endOfBlockInterrupt = FALSE;

    status = dpio2Ioctl(dpio2DeviceNumber,
                        DPIO2_CMD_DMA_SET_DESC,
                        (int) &dmaDescriptor);
    if ( status != OK ) {
      printf("Failed to set DMA Descriptor\n");
      return (ERROR);
    }
  }

  dmaDescriptor.descriptorId        = iPage;
  dmaDescriptor.nextDescriptorId    = 0;
  dmaDescriptor.pciAddress
    = (UINT32) pPageInfo->Page[iPage].pPhysicalAddr;
  dmaDescriptor.blockSizeInBytes
    = pPageInfo->Page[iPage].dwBytes;
  dmaDescriptor.lastBlockInChain    = TRUE;
  dmaDescriptor.useD64              = FALSE;
  dmaDescriptor.notEndOfFrame       = FALSE;
  dmaDescriptor.endOfBlockInterrupt = TRUE;

  status = dpio2Ioctl(dpio2DeviceNumber,
                      DPIO2_CMD_DMA_SET_DESC,
                      (int) &dmaDescriptor);
  if ( status != OK ) {
    printf("Failed to set DMA Descriptor\n");
    return (ERROR);
  }

  return (OK);
}
```

3.1.8 Starting DMA transfer

Two steps are involved in starting a DMA transfer. First the application may specify which DMA descriptor the DMA chain should start with (see 4.6.6.1, 4.6.6.2, and 4.6.6.3 for details about DMA transfers, DMA chains, and DMA descriptors). Then the DMA controller must be instructed to start the DMA transfer.

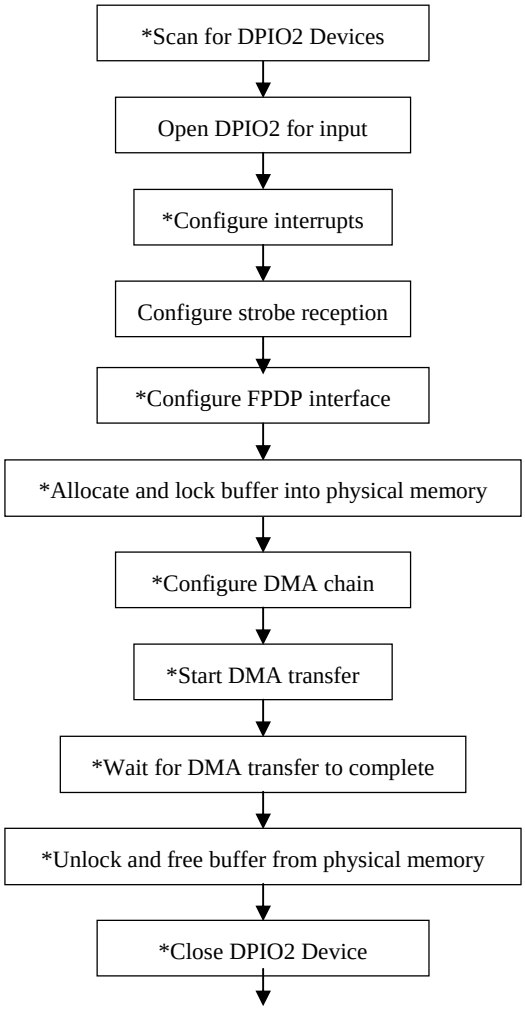
The example below shows how this is done:

3.1.9 Unlocking and freeing buffer

When the buffer is no longer needed, it must be unlocked and freed, as shown in the example below:

3.2 Using a DPIO2 module for input

This section shows how an application typically uses a DPIO2 module for input. An overview of the steps involved is shown in the figure below.



The steps marked by * in the figure above, are the same as those configured for Output. See the previous section for descriptions and examples of these steps.

The steps that are different are described in the subsections below.

3.2.1 Open DPIO2 for input

An application has to open a DPIO2 device to get exclusive access to it. In the same operation, the application selects whether to use the associated DPIO2 module as input or output.

The example below shows how a DPIO2 device is opened for input:

```
STATUS openDpio2ForInput(int dpio2DeviceNumber)
{
  STATUS   status;

  status = dpio2Open(dpio2DeviceNumber, DPIO2_INPUT);
  if ( status != OK ) {
    printf("Failed to initialize the DPIO2 for input\n");
    return (ERROR);
  }

  return (OK);
}
```

3.2.2 Configuring strobe reception

DPIO2 modules with the FPDP personality can receive either the TTL strobe signal or the PECL strobe signal. An application must specify the value `DPIO2_PRIMARY_STROBE` as an argument to the `DPIO2_CMD_STROBE_RECEPTION_ENABLE` command to select the TTL strobe, and `DPIO2_SECONDARY_STROBE` to select the PECL strobe.

DPIO2 modules with the LVDS, PECL, or RS422 personality can only receive one strobe signal, and must call `DPIO2_CMD_STROBE_RECEPTION_ENABLE` with `DPIO2_PRIMARY_STROBE` as argument.

The example below shows how to configure a DPIO2 module to receive the primary strobe signal.

```
STATUS enableStrobeReception(int dpio2DeviceNumber, UINT32
strobeFrequency)
{
  STATUS   status;
  int   range;

  status = dpio2Ioctl(dpio2DeviceNumber,
                      DPIO2_CMD_STROBE_RECEPTION_ENABLE,
                      DPIO2_PRIMARY_STROBE);
  if ( status != OK ) {
    printf("Failed to enable strobe reception\n");
    return (ERROR);
  }

  /* Determine which of the predefined ranges,
   * the specfied strobe frequency falls within.
   */
  if ( strobeFrequency <= (5*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_BELOW_OR_EQUAL_TO_5MHZ;
  } else  if ( strobeFrequency <= (10*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_5MHZ_BELOW_OR_EQUAL_TO_10MHZ;
  } else  if ( strobeFrequency <= (15*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_10MHZ_BELOW_OR_EQUAL_TO_15MHZ;
  } else  if ( strobeFrequency <= (20*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_15MHZ_BELOW_OR_EQUAL_TO_20MHZ;
  } else  if ( strobeFrequency <= (25*1000*1000) ) {
```

```
    range = DPIO2_STROBE_FREQUENCY_ABOVE_20MHZ_BELOW_OR_EQUAL_TO_25MHZ;
  } else  if ( strobeFrequency <= (30*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_25MHZ_BELOW_OR_EQUAL_TO_30MHZ;
  } else  if ( strobeFrequency <= (35*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_30MHZ_BELOW_OR_EQUAL_TO_35MHZ;
  } else  if ( strobeFrequency <= (40*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_35MHZ_BELOW_OR_EQUAL_TO_40MHZ;
  } else  if ( strobeFrequency <= (45*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_40MHZ_BELOW_OR_EQUAL_TO_45MHZ;
  } else  if ( strobeFrequency <= (50*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_45MHZ_BELOW_OR_EQUAL_TO_50MHZ;
  } else  if ( strobeFrequency <= (55*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_50MHZ_BELOW_OR_EQUAL_TO_55MHZ;
  } else  if ( strobeFrequency <= (60*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_55MHZ_BELOW_OR_EQUAL_TO_60MHZ;
  } else  if ( strobeFrequency <= (65*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_60MHZ_BELOW_OR_EQUAL_TO_65MHZ;
  } else  if ( strobeFrequency <= (70*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_65MHZ_BELOW_OR_EQUAL_TO_70MHZ;
  } else  if ( strobeFrequency <= (75*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_70MHZ_BELOW_OR_EQUAL_TO_75MHZ;
  } else  if ( strobeFrequency <= (80*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_75MHZ_BELOW_OR_EQUAL_TO_80MHZ;
  } else  if ( strobeFrequency <= (85*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_80MHZ_BELOW_OR_EQUAL_TO_85MHZ;
  } else  if ( strobeFrequency <= (90*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_85MHZ_BELOW_OR_EQUAL_TO_90MHZ;
  } else  if ( strobeFrequency <= (95*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_90MHZ_BELOW_OR_EQUAL_TO_95MHZ;
  } else  if ( strobeFrequency <= (100*1000*1000) ) {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_95MHZ_BELOW_OR_EQUAL_TO_100MHZ;
  } else {
    range = DPIO2_STROBE_FREQUENCY_ABOVE_100MHZ;
  }


  status = dpio2Ioctl(dpio2DeviceNumber,
                      DPIO2_CMD_STROBE_FREQUENCY_RANGE_SET,
                      range);
  if ( status != OK ) {
    printf("Failed to set %d as strobe frequency range\n", range);
    return (ERROR);
  }

  return (OK);
}
```

4 DPIO2 API

This section describes the functions that constitute the DPIO2 API, dpio2Scan(), dpio2Open(), dpio2Close(), and dpio2Ioctl().

4.1 dpio2Scan ()

Syntax: int `dpio2Scan(DPIO_INFO_T* pInfo)`

Description: Checks if the DPIO2 kernel module is properly installed and scans all PCI slots to find DPIO2 modules. It returns information about each DPIO2 modules found.

pInfo points to an array of DPIO_INFO structures, one for each DPIO2 found. The maximum number of DPIO2 Modules is 8. The DPIO_INFO structure has the following members:

devno	device number of DPIO2 module .
deviceId	device ID of DPIO2 module.
vendorId	vendor ID of DPIO2 module.
revisionId	revision ID of DPIO2 module.
moduleType	FPDP, LVDS, RS422 or PECL module. `DPIO2_FB_MODULE, DPIO2_LB_MODULE, DPIO2_EI_MODULE, DPIO2_EO_MODULE, DPIO2_DI_MODULE or DPIO2_DO_MODULE.`
pciBusNumber	bus number on the PCI bus.
pciDeviceNumber	device number on the PCI bus.
pciFpgaVersion	PCI FPGA version of the DPIO2 module.
forntEndFpgaVersion	front end FPGA version of the DPIO2 module.

Returns: Number of DPIO2 modules found or ERROR if an error occurred.

4.2 dpio2Open ()

Syntax: `STATUS dpio2Open(int devno, int mode)`

Description: Opens a DPIO2 Module for either input or output.

The first parameter, `devno`, specifies the DPIO2 module to be opened.

The second parameter, `mode`, specifies the data direction the module should be for:

`DPIO2_INPUT` – the module is initialised to receive data

`DPIO2_OUTPUT` – the module is initialised to transmit data

dpio2Open() can be only be opened once. Call dpio2Close() and then dpio2Open to switch a module between input and output. Note that all previous configuration settings are lost when dpio2Close() is called, so the application must configure the module afterwards. Also note that all buffered data is lost.

The following actions are implicitly performed when dpio2Open():

- all interrupt sources on the device are disabled (see 4.6.7.2)
- all interrupt actions are removed (see 4.6.7.4 and 4.6.7.6)
- the front-end on the device is disabled (see 4.6.2.1)
- the device is configured to receive the primary strobe (see 4.6.1.7)
- the programmable oscillator is set to 20MHz (see 4.6.1.1)
- any active DMA transfer is aborted (see 4.6.6.4)
- the FIFO is flushed (see 4.6.8.1)
- the direction of the module is set according to the `mode` argument as described above

Returns: OK if device is opened successfully, ERROR otherwise.

4.3 dpio2Close ()

Syntax: `void dpio2Close(int devno)`

Description: Frees the resources allocated by dpio2Open().

It also performs the following actions for this DPIO2 modules:

- stops all transfers.
- disable route of interrupts from device.
- all interrupt sources on the device are disabled (see 3.9.7.2).
- all interrupt actions are removed (see 3.9.7.4
- disable strobe generation.

Returns: OK or ERROR.

4.4 dpio2DMALock()

Syntax: `STATUS dpio2DMALock(DPIO2_DMA* pDma)`

Description: dpio2DMALock() locks a virtual memory buffer into physical memory so that it can be accessed in DMA transfers. It generates a scatter list with PCI addresses to user DMA buffer.

The only parameter to this function is a pointer to a `DPIO2_DMA` structure. The

application must allocate this structure before calling dpio2DMALock(). Allocate the DPIO2_DMA structure as shown in 3.1.6).

Before dpio2DMALock() is called, the application must set the following fields in the DPIO2_DMA structure:

pUserAddr	- pointer to the virtual memory buffer to be locked.
dwBytes	- number of bytes in the virtual memory buffer to be locked.
dwPages	- number of pages the DPIO2_DMA structure can store information about

When dpio2DMALock() returns successfully, the following fields in the DPIO2_DMA structure are set:

dwPages	- number of locked pages the DPIO2_DMA structure has information about
Page[]	- array of DPIO2_DMA_PAGE structures holding information about the locked pages.

The DPIO2_DMA_PAGE structure has the following fields:

pPhysicalAddr	- physical address of page i.
dwBytes	- length in bytes of page i.

The rest of the fields in the DPIO2_DMA structure should not be used by the application.

Returns: OK if command is executed successfully, ERROR otherwise.

4.5 dpio2DMAUnlock()

Syntax: `void dpio2DMAUnlock(DPIO2_DMA* pDma)`

Description: dpio2DMAUnlock() unlocks a virtual memory buffer that was locked into physical memory by calling dpio2DMALock().

The only parameter to this function is a pointer to the same DPIO2_DMA that was used in the call to dpio2DMALock().

Returns: Nothing

4.6 dpio2Ioctl ()

Syntax: `int dpio2Ioctl(int devno, int command, int argument)`

Description: Issues commands to an open device.

The first argument is the device number returned for the module when dpio2Open() was called.

The second argument is an integer that specifies the command to be issued to the device. Commands applicable to a DPIO2 device are described below.

The third argument is an integer that is used to transfer further information if needed by a command.

Returns: OK if command is executed successfully, ERROR otherwise.

4.6.1 Strobe Configuration Commands

4.6.1.1 DPIO2_CMD_STROBE_FREQUENCY_SET

Description: Sets the frequency of the FPDP strobe.

Expects a pointer to a variable of type UINT32 as argument. When the command is issued, this variable must be set to the wanted frequency in Hz. Upon return the variable will contain the closest obtainable frequency, which the strobe frequency is actually set to:

```
UINT32   frequencyInHz = <wanted-frequency>;


status = dpio2Ioctl(fd, DPIO2_CMD_STROBE_FREQUENCY_SET,
                    (int) &frequencyInHz);
```

The DPIO2 module must have been set to generate strobe using the programmable oscillator, for this command to have any effect.

Related commands:

DPIO2_CMD_STROBE_GENERATION_ENABLE (see 4.6.1.6)

4.6.1.2 DPIO2_CMD_STROBE_FREQUENCY_RANGE_SET

Description: Informs the driver about the frequency range the FPDP strobe lies in.

Expects a integer value as argument, which identifies the frequency range being used:

```
status = dpio2Ioctl(fd, DPIO2_CMD_STROBE_FREQUENCY_RANGE_SET,
                    <frequency-range>);
```

One of the following values must be used for <frequency-range>:

```
DPIO2_STROBE_FREQUENCY_BELOW_OR_EQUAL_TO_5MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_5MHZ_BELOW_OR_EQUAL_TO_10MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_10MHZ_BELOW_OR_EQUAL_TO_15MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_15MHZ_BELOW_OR_EQUAL_TO_20MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_20MHZ_BELOW_OR_EQUAL_TO_25MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_25MHZ_BELOW_OR_EQUAL_TO_30MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_30MHZ_BELOW_OR_EQUAL_TO_35MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_35MHZ_BELOW_OR_EQUAL_TO_40MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_40MHZ_BELOW_OR_EQUAL_TO_45MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_45MHZ_BELOW_OR_EQUAL_TO_50MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_50MHZ_BELOW_OR_EQUAL_TO_55MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_55MHZ_BELOW_OR_EQUAL_TO_60MHZ
```

```
DPIO2_STROBE_FREQUENCY_ABOVE_60MHZ_BELOW_OR_EQUAL_TO_65MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_65MHZ_BELOW_OR_EQUAL_TO_70MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_70MHZ_BELOW_OR_EQUAL_TO_75MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_75MHZ_BELOW_OR_EQUAL_TO_80MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_80MHZ_BELOW_OR_EQUAL_TO_85MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_85MHZ_BELOW_OR_EQUAL_TO_90MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_90MHZ_BELOW_OR_EQUAL_TO_95MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_95MHZ_BELOW_OR_EQUAL_TO_100MHZ
DPIO2_STROBE_FREQUENCY_ABOVE_100MH
```

This command should be issued when the DPIO2 is receiving the FPDP strobe and when the DPIO2 is generating the strobe based on a fixed oscillator.

Related commands:

DPIO2_CMD_STROBE_GENERATION_ENABLE (see 4.6.1.6)
DPIO2_CMD_STROBE_RECEPTION_ENABLE (see 4.6.1.7)

4.6.1.3 DPIO2_CMD_CLOCKING_ON_BOTH_STROBE_EDGES_SELECT

Selects whether to enable clocking on both strobe edges. Expects a boolean value as argument and enables clocking on both edges if this value is TRUE:

```
status=dpio2Ioctl(fd,
                  DPIO2_CMD_CLOCKING_ON_BOTH_STROBE_EDGES_SELECT,
                  TRUE);
```

If the boolean argument is FALSE, clocking on both strobe edges is disabled:

```
status =dpio2Ioctl(fd,
                  DPIO2_CMD_CLOCKING_ON_BOTH_STROBE_EDGES_SELECT,
                  FALSE);
```

Related commands:

DPIO2_CMD_STROBE_GENERATION_ENABLE (see 4.6.1.6)
DPIO2_CMD_STROBE_RECEPTION_ENABLE (see 4.6.1.7)

4.6.1.4 DPIO2_CMD_STROBE_SKEW_SET

This command is used to specify the strobe skew when clocking data on both strobe edges (the command has no effect when data is clocked on only one of the strobe edges).

The amount of skew is specified as positive or negative multiples of a Time Unit (tu). The duration of one Time Unit is frequency dependent, as show in the table below:

Strobe Frequency (F_s) Range	**Time Unit**
≤ 25 MHz	1 / (64 * F_s)
25 – 50 MHz	1 / (32 * F_s)
> 50 MHz	1 / (16 * F_s)

The command takes an integer as argument:

```
status =dpio2Ioctl(fd,
                   DPIO2_CMD_STROBE_SKEW_SET,
                   <skew-in-time-units>);
```

Valid values for <skew-in-time-units> are –6, -4, -2, 0, 2, 4, and 6.

Related commands:

DPIO2_CMD_DEFAULT_STROBE_SKEW_SET
DPIO2_CMD_CLOCKING_ON_BOTH_STROBE_EDGES_SELECT

4.6.1.5 DPIO2_CMD_DEFAULT_STROBE_SKEW_SET

This command makes the driver use default strobe skew when clocking data on both strobe edges (the command has no effect when data is clocked on only one of the strobe edges).

The command takes no arguments:

```
status =dpio2Ioctl(fd, DPIO2_CMD_DEFAULT_STROBE_SKEW_SET, 0);
```

Related commands:

DPIO2_CMD_STROBE_SKEW_SET
DPIO2_CMD_CLOCKING_ON_BOTH_STROBE_EDGES_SELECT

4.6.1.6 DPIO2_CMD_STROBE_GENERATION_ENABLE

Description: Enables Strobe Generation and, consequently, disables Strobe Reception. This command also selects whether the fixed or the programmable oscillator should be used.

Expects an integer value as argument:

```
status = dpio2Ioctl(fd, DPIO2_CMD_STROBE_GENERATION_ENABLE,
                    <oscillator>);
```

One of the following values must be used for <oscillator>:

```
DPIO2_FIXED_OSCILLATOR
DPIO2_PROGRAMMABLE_OSCILLATOR
```

Related commands:

DPIO2_CMD_STROBE_RECEPTION_ENABLE

DPIO2_CMD_STROBE_FREQUENCY_SET

4.6.1.7 DPIO2_CMD_STROBE_RECEPTION_ENABLE

Description: Enables Strobe Reception and, consequently, disables Strobe Generation. This command also selects the strobe source to be used, through the integer value given as argument:

```
status = dpio2Ioctl(fd, DPIO2_CMD_STROBE_RECEPTION_ENABLE,
                    <strobe-signal>);
```

One of the following values must be used for <strobe-signal>:

```
DPIO2_PRIMARY_STROBE
```

DPIO2_SECONDARY_STROBE

For DPIO2 modules with FPDP personality, DPIO2_PRIMARY_STROBE selects the TTL strobe signal and DPIO2_SECONDARY_STROBE selects the PECL strobe signal.

For DPIO2 modules with LVDS, PECL, and RS422 personality, DPIO2_PRIMARY_STROBE is the only valid option – DPIO2_SECONDARY_STROBE cannot be used.

Related commands:

DPIO2_CMD_STROBE_GENERATION_ENABLE

DPIO2_CMD_STROBE_FREQUENCY_RANGE_SET

4.6.2 Frontend Commands

4.6.2.1 DPIO2_CMD_FPDP_ACTIVATION_SELECT

Description: Selects whether the DPIO2 module shall activate its FPDP interface or not.

Expects a value of type BOOL as argument, and activates the interface if the argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_FPDP_ACTIVATION_SELECT,TRUE);
```

If the argument is FALSE the FPDP interface is deactivated:

```
status=dpio2Ioctl(fd,DPIO2_CMD_FPDP_ACTIVATION_SELECT,FALSE);
```

4.6.2.2 DPIO2_CMD_SYNC_GENERATION_COUNTER_SET

Description: Sets the value of the counter which is used to control SYNC generation if the DPIO2_SYNC_GENERATION_ON_COUNT mode is selected by the DPIO2_CMD_SYNC_GENERATION_SELECT command.

If the counter is set to N, SYNC will be asserted for every N'th word transferred. The minimum valid counter value is 2.

This command expects an unsigned long integer as argument:

```
unsigned long   counterValue;

/* Set 'counterValue' to the desired value.*/
status = dpio2Ioctl(fd,
                    DPIO2_CMD_SYNC_GENERATION_COUNTER_SET,
                    (int) counterValue);
```

Related commands:

DPIO2_CMD_SYNC_GENERATION_SELECT

4.6.2.3 DPIO2_CMD_SYNC_GENERATION_SELECT

Description: Selects how a DPIO2 configured as output shall generate SYNC.

Expects a integer as argument, and configures the SYNC generation according to the value of this function:

```
status = dpio2Ioctl(fd, DPIO2_CMD_SYNC_GENERATION_SELECT,
```

```
                                <syncGenerationMode>);
```

The argument <syncGenerationMode> must have one of the following values:

```
DPIO2_SYNC_GENERATION_DISABLED

DPIO2_SYNC_GENERATION_BEFORE_DATA

DPIO2_SYNC_GENERATION_AT_END_OF_FRAME

DPIO2_SYNC_GENERATION_AT_START_OF_FRAME

DPIO2_SYNC_GENERATION_ON_ODD_FRAME

DPIO2_SYNC_GENERATION_ON_COUNT
```

If the DPIO2_SYNC_GENERATION_ON_COUNT mode is selected, the DPIO2_CMD_SYNC_GENERATION_COUNTER_SET command must called to specify the counter value to be used.

In order to have effect, this command must be issued before the FPDP interface is activated (see command DPIO2_CMD_FPDP_ACTIVATION_SELECT).

Related commands:

```
DPIO2_CMD_D0_TO_BE_USED_FOR_SYNC_SELECT
DPIO2_CMD_SYNC_GENERATION_COUNTER_SET
DPIO2_CMD_SYNC_RECEPTION_SELECT
```

4.6.2.4 DPIO2_CMD_SYNC_RECEPTION_SELECT

Description: Selects how a DPIO2 configured as input shall handle received SYNC.

Expects a integer as argument, and configures the SYNC reception according to the value of this function:

```
status = dpio2Ioctl(fd, DPIO2_CMD_SYNC_RECEPTION_SELECT,
                        <syncReceptionMode>);
```

The argument <syncReceptionMode> must have one of the following values:

```
DPIO2_SYNC_RECEPTION_DISABLED

DPIO2_SYNC_RECEPTION_STARTS_DATA_RECEPTION
```

In order to have effect, this command must be issued before the FPDP interface is activated (see command DPIO2_CMD_FPDP_ACTIVATION_SELECT).

Related commands:

```
DPIO2_CMD_D0_TO_BE_USED_FOR_SYNC_SELECT
DPIO2_CMD_SYNC_GENERATION_SELECT
```

4.6.2.5 DPIO2_CMD_D0_TO_BE_USED_FOR_SYNC_SELECT

Description: Selects whether data bit 0 (D0) shall be used in handling SYNC or not.

If D0 is selected to handle SYNC on a DPIO2 module configured for output, D0 is routed to the SYNC signal. The opposite will happen on modules configured for input; the SYNC signal is routed to D0.

The command expects a value of type BOOL as argument, and enables the routing if this parameter is TRUE:

```
status = dpio2Ioctl(fd,
                        DPIO2_CMD_D0_TO_BE_USED_FOR_SYNC_SELECT,
```

```
                                TRUE);
```

If the parameter is FALSE, D0 will not be used to handle SYNC:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_D0_TO_BE_USED_FOR_SYNC_SELECT,
                    FALSE);
```

Related commands:

```
DPIO2_CMD_SYNC_GENERATION_SELECT
DPIO2_CMD_SYNC_RECEPTION_SELECT
```

4.6.2.6 DPIO2_CMD_VIDEO_MODE_SELECT

Description: Selects whether Video Mode shall be enabled or not.

Expects a value of type BOOL as argument, and enables Video Mode if this argument is TRUE:

```
status = dpio2Ioctl(fd, DPIO2_CMD_VIDEO_MODE_SELECT, TRUE);
```

If the argument is FALSE, Video Mode is disabled:

```
status = dpio2Ioctl(fd, DPIO2_CMD_VIDEO_MODE_SELECT, FALSE);
```

To have effect this command must be issued before the FPDP interface is activated (see DPIO2_CMD_FPDP_ACTIVATION_SELECT).

This command is only applicable if the DPIO2 is configured as input.

4.6.2.7 DPIO2_CMD_COUNTER_ADDRESSING_ENABLE

Description: Enables Counter Addressing on a DPIO2 module configured as input.

Expects a pointer to a `DPIO2_COUNTER_ADDRESSING_INFO` structure as argument:

```
DPIO2_COUNTER_ADDRESSING_INFO   addressingInfo;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_COUNTER_ADDRESSING_ENABLE,
                    (int) &addressingInfo);
```

The `DPIO2_COUNTER_ADDRESSING_INFO` structure has the following fields:

- initialSkipCount - unsigned integer specifying how many words the DPIO2 module shall ignore immediately after the FPDP interface is activated
- skipCount - unsigned integer specifying how many words the DPIO2 shall ignore after it has received a specified amount of data
- receiveCount - unsigned integer specifying how many words the DPIO2 module shall receive after skipping the number of words specified by ‘skipCount’ or ‘initialSkipCount’

NOTE: In order to have effect this command must be called before the FPDP interface is activated (`DPIO2_CMD_FPDP_ACTIVATION_SELECT`).

Related Commands:

```
DPIO2_CMD_COUNTER_ADDRESSING_DISABLE
```

4.6.2.8 DPIO2_CMD_COUNTER_ADDRESSING_DISABLE

Description: Disables Counter Addressing on a DPIO2 module configured as input.

This command requires no arguments:

```
status=dpio2Ioctl(fd,DPIO2_CMD_COUNTER_ADDRESSING_DISABLE,0);
```

NOTE: In order to have effect this command must be called before the FPDP interface is activated (`DPIO2_CMD_FPDP_ACTIVATION_SELECT`).

Related Commands:

```
DPIO2_CMD_COUNTER_ADDRESSING_ENABLE
```

4.6.2.9 DPIO2_CMD_TEST_PATTERN_GENERATION_ENABLE

Description: On a DPIO2 configured for input, this command causes test patterns to be written into the FIFO instead of data from the FPDP bus. And on a DPIO2 configured for output, this command causes test patterns to be clocked onto the FPDP bus instead of data from the FIFO.

The command expects an integer as argument:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_TEST_PATTERN_GENERATION_ENABLE,
                    <patternId>);
```

The integer given as argument, <patternId>, must have one of the following values:

```
DPIO2_WALKING_ONE_PATTERN
DPIO2_WALKING_ZERO_PATTERN
DPIO2_COUNTER_PATTERN
DPIO2_COUNTER_PATTERN_WITH_PROGRAMMABLE_START
```

Note: To have effect, this command must be issued before the FPDP interface is activated (DPIO2_CMD_FPDP_ACTIVATION_SELECT).

Related commands:

```
DPIO2_CMD_FPDP_ACTIVATION_SELECT
DPIO2_CMD_TEST_PATTERN_GENERATION_DISABLE
```

4.6.2.10 DPIO2_CMD_TEST_PATTERN_GENERATION_DISABLE

Description: This command disables test pattern generation.

The command takes no arguments:

```
status=dpio2Ioctl(fd,
                  DPIO2_CMD_TEST_PATTERN_GENERATION_DISABLE,
                  0);
```

Note: To have effect, this command must be issued before the FPDP interface is activated (DPIO2_CMD_FPDP_ACTIVATION_SELECT).

Related commands:

```
DPIO2_CMD_FPDP_ACTIVATION_SELECT
DPIO2_CMD_TEST_PATTERN_GENERATION_ENABLE
```

4.6.2.11 DPIO2_CMD_TEST_PATTERN_START_VALUE_SET

This command specifies the start value that will be used when the test pattern generator is set to produce the counter pattern with programmable start (DPIO2_COUNTER_PATTERN_WITH_PROGRAMMABLE_START).

The command takes a 16 bits unsigned integer as argument:

```
UINT16   startValue;

status=dpio2Ioctl(fd,
                  DPIO2_CMD_TEST_PATTERN_START_VALUE_SET,
                  (int) startValue);
```

Related commands:

```
DPIO2_CMD_TEST_PATTERN_GENERATION_ENABLE
DPIO2_CMD_TEST_PATTERN_GENERATION_DISABLE
```

4.6.3 Data Formatting Commands

4.6.3.1 DPIO2_CMD_DATA_SWAP_MODE_SELECT

Description: Selects how a DPIO2 shall swap 8 bit, 16 bit, and 32 bit data.

Expects an integer as argument, and configures the data swap mode according to this value:

```
status = dpio2Ioctl(fd, DPIO2_CMD_DATA_SWAP_MODE_SELECT,
                    <dataSwapMode>);
```

The argument <dataSwapMode> must have one of the values listed in the first column in the table below. The second column shows how two consecutive 32 bits words are transformed by the various swap modes.

Data Swap Mode Identifier	0x11223344 0x55667788 is transformed to
DPIO2_NO_SWAP	0x11223344 0x55667788
DPIO2_8BIT_SWAP	0x44332211 0x88776655
DPIO2_16BIT_SWAP	0x33441122 0x77885566
DPIO2_8BIT_16BIT_SWAP	0x22114433 0x66558877
DPIO2_32BIT_SWAP	0x55667788 0x11223344
DPIO2_8BIT_32BIT_SWAP	0x88776655 0x44332211
DPIO2_16BIT_32BIT_SWAP	0x77885566 0x33441122
DPIO2_8BIT_16BIT_32BIT_SWAP	0x66558877 0x22114433

NOTE: 32 bit swapping will only have effect for DMA transfers using D64 accesses.

4.6.3.2 DPIO2_CMD_DATA_PACKING_CAPABILITY_GET

Tells which data packing modes are available on a DPIO2 module.

The command expects pointer to an integer variable as argument, and returns a value in this variable which represents the available packing modes:

```
int   packingCapability;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_DATA_PACKING_CAPABILITY_GET,
                    (int) &packingCapability);
```

The returned value will have one of the following values depending on your DPIO2 module. Consult VMTERO if this does not return the expected value:

```
DPIO2_8BIT_4BIT_PACKING_AVAILABLE

DPIO2_16BIT_10BIT_PACKING_AVAILABLE
```

Related commands:

```
DPIO2_CMD_DATA_PACKING_ENABLE

DPIO2_CMD_DATA_PACKING_DISABLE
```

4.6.3.3 DPIO2_CMD_DATA_PACKING_ENABLE

Enables unpacking from 32 bits words on DPIO2 modules configured for output, and packing into 32 bits on modules configured for input.

The command expects a value that specifies how data should be packed/unpacked:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_DATA_PACKING_ENABLE,
                    <packingMode>);
```

The argument `<packingMode>` must have one of the following values:

```
DPIO2_PACK_16_LSB_ON_FPDP

DPIO2_PACK_16_MSB_ON_FPDP

DPIO2_PACK_10_LSB_ON_FPDP

DPIO2_PACK_8_LSB_ON_FPDP

DPIO2_PACK_4_LSB_ON_FPDP
```

```
Note: 4 Bit and 8 Bit packing are not supported in the current
firmware release.
```

Related commands:

```
DPIO2_CMD_DATA_PACKING_CAPABILITY_GET

DPIO2_CMD_DATA_PACKING_DISABLE

DPIO2_CMD_DATA_PACKING_PIPELINE_CHECK

DPIO2_CMD_DATA_PACKING_PIPELINE_FLUSH
```

4.6.3.4 DPIO2_CMD_DATA_PACKING_DISABLE

Disables unpacking from 32 bits words on DPIO2 modules configured for output, or packing into 32 bits on modules configured for input.

The command takes no arguments:

```
status = dpio2Ioctl(fd, DPIO2_CMD_DATA_PACKING_DISABLE, 0);
```

Related commands:

DPIO2_CMD_DATA_PACKING_ENABLE

DPIO2_CMD_DATA_PACKING_PIPELINE_CHECK

DPIO2_CMD_DATA_PACKING_PIPELINE_FLUSH

4.6.3.5 DPIO2_CMD_DATA_PACKING_PIPELINE_CHECK

Checks whether there are data in the packing pipeline on a DPIO2 module configured as input. Expects a pointer to a BOOL as argument:

```
BOOL   pipelineNotEmpty;
status = dpio2Ioctl(fd,
                    DPIO2_CMD_DATA_PACKING_PIPELINE_CHECK,
                    (int) &pipelineNotEmpty);
```

If there are data in the pipeline, the boolean variable referenced by the pointer argument, is set to TRUE. If the pipeline is empty, the boolean variable is set to FALSE.

Related commands:

DPIO2_CMD_DATA_PACKING_ENABLE

DPIO2_CMD_DATA_PACKING_DISABLE

DPIO2_CMD_DATA_PACKING_PIPELINE_FLUSH

4.6.3.6 DPIO2_CMD_DATA_PACKING_PIPELINE_FLUSH

Flushes data in the packing pipeline on a DPIO2 module configured for input. Garbage data is added for the lacking bits in the resulting 32 bits word.

This commands takes no arguments:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_DATA_PACKING_PIPELINE_FLUSH,
                    0);
```

Related commands:

DPIO2_CMD_DATA_PACKING_ENABLE

DPIO2_CMD_DATA_PACKING_DISABLE

DPIO2_CMD_DATA_PACKING_PIPELINE_CHECK

4.6.4 Flow Control Commands

4.6.4.1 DPIO2_CMD_SUSPEND_FLOW_CONTROL_SELECT

Description: Selects whether flow control by the SUSPEND signal shall be enabled or not. Expects a value of type BOOL as argument, and enables SUSPEND flow control if this argument is TRUE:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_SUSPEND_FLOW_CONTROL_SELECT,
                    TRUE);
```

If the argument is FALSE, SUSPEND flow control is disabled:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_SUSPEND_FLOW_CONTROL_SELECT,
                    FALSE);
```

By default SUSPEND flow control is enabled.

Related commands:

`DPIO2_CMD_NRDY_FLOW_CONTROL_SELECT`

4.6.4.2 DPIO2_CMD_SUSPEND_ASSERTION_FORCE

Description: This command controls the SUSPEND signal directly on DPIO2 modules configured for input. When the command is issued with the Boolean value TRUE as argument, the SUSPEND signal is asserted regardless of the FIFO state:

```
status=dpio2Ioctl(fd,
                  DPIO2_CMD_SUSPEND_ASSERTION_FORCE,
                  TRUE);
```

To restore the default SUSPEND flow control, the command is issued with the Boolean value FALSE as argument:

```
status=dpio2Ioctl(fd,
                  DPIO2_CMD_SUSPEND_ASSERTION_FORCE,
                  FALSE);
```

Related commands:

`DPIO2_CMD_SUSPEND_FLOW_CONTROL_SELECT`

4.6.4.3 DPIO2_CMD_NRDY_FLOW_CONTROL_SELECT

Description: Selects whether flow control by the NRDY signal shall be enabled or not. Expects a value of type BOOL as argument, and enables NRDY flow control if this argument is TRUE:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_NRDY_FLOW_CONTROL_SELECT,
                    TRUE);
```

If the argument is FALSE, NRDY flow control is disabled:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_NRDY_FLOW_CONTROL_SELECT,
                    FALSE);
```

By default NRDY flow control is enabled.

Related commands:

`DPIO2_CMD_SUSPEND_FLOW_CONTROL_SELECT`

4.6.5 IO Signalling Commands

4.6.5.1 DPIO2_CMD_RES1_DIRECTION_SELECT

Description: Configures the direction of the RES1 signal. Expects a value of type BOOL as argument, and configures RES1 as output if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES1_DIRECTION_SELECT,TRUE);
```

If the argument is FALSE, RES1 is configured as input:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES1_DIRECTION_SELECT,FALSE);
```

By default RES1 is configured as input.

Related commands:

`DPIO2_CMD_RES1_OUTPUT_VALUE_SET`

`DPIO2_CMD_RES1_VALUE_GET`

4.6.5.2 DPIO2_CMD_RES1_OUTPUT_VALUE_SET

Description: Sets the output value of the RES1 signal. Expects a value of type BOOL as argument, and sets RES1 high if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES1_OUTPUT_VALUE_SET,TRUE);
```

If the argument is FALSE, RES1 is set low:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES1_OUTPUT_VALUE_SET,FALSE);
```

Note: This command has no effect, unless the RES1 signal has been configured as output by the DPIO2_CMD_RES1_DIRECTION_SELECT command.

Related commands:

`DPIO2_CMD_RES1_DIRECTION_SELECT`

`DPIO2_CMD_RES1_VALUE_GET`

4.6.5.3 DPIO2_CMD_RES1_VALUE_GET

Description: Returns the value of the RES1 signal. Expects a pointer to a variable of type BOOL as argument, and sets this variable to TRUE if RES1 is high and FALSE if RES1 is low:

```
BOOL   res1IsHigh;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_RES1_VALUE_GET,
                    (int) &res1IsHigh);
```

Related commands:

`DPIO2_CMD_RES1_DIRECTION_SELECT`

`DPIO2_CMD_RES1_OUTPUT_VALUE_SET`

4.6.5.4 DPIO2_CMD_RES2_DIRECTION_SELECT

Description: Configures the direction of the RES2 signal. Expects a value of type BOOL as argument, and configures RES2 as output if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES2_DIRECTION_SELECT,TRUE);
```

If the argument is FALSE, RES2 is configured as input:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES2_DIRECTION_SELECT,FALSE);
```

By default RES2 is configured as input.

Related commands:

DPIO2_CMD_RES2_OUTPUT_VALUE_SET

DPIO2_CMD_RES2_VALUE_GET

4.6.5.5 DPIO2_CMD_RES2_OUTPUT_VALUE_SET

Description: Sets the output value of the RES2 signal. Expects a value of type BOOL as argument, and sets RES2 high if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES2_OUTPUT_VALUE_SET,TRUE);
```

If the argument is FALSE, RES2 is set low:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES2_OUTPUT_VALUE_SET,FALSE);
```

Note: This command has no effect, unless the RES2 signal has been configured as output by the DPIO2_CMD_RES2_DIRECTION_SELECT command.

Related commands:

DPIO2_CMD_RES2_DIRECTION_SELECT

DPIO2_CMD_RES2_VALUE_GET

4.6.5.6 DPIO2_CMD_RES2_VALUE_GET

Description: Returns the value of the RES2 signal. Expects a pointer to a variable of type BOOL as argument, and sets this variable to TRUE if RES2 is high and FALSE if RES2 is low:

```
BOOL   res2IsHigh;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_RES2_VALUE_GET,
                    (int)&res2IsHigh);
```

Related commands:

DPIO2_CMD_RES2_DIRECTION_SELECT

DPIO2_CMD_RES2_OUTPUT_VALUE_SET

4.6.5.7 DPIO2_CMD_RES3_DIRECTION_SELECT

Description: Configures the direction of the RES3 signal. Expects a value of type BOOL as argument, and configures RES3 as output if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES3_DIRECTION_SELECT,TRUE);
```

If the argument is FALSE, RES3 is configured as input:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES3_DIRECTION_SELECT,FALSE);
```

By default RES3 is configured as input.

Related commands:

DPIO2_CMD_RES3_OUTPUT_VALUE_SET

DPIO2_CMD_RES3_VALUE_GET

4.6.5.8 DPIO2_CMD_RES3_OUTPUT_VALUE_SET

Description: Sets the output value of the RES3 signal. Expects a value of type BOOL as argument, and sets RES3 high if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES3_OUTPUT_VALUE_SET,TRUE);
```

If the argument is FALSE, RES3 is set low:

```
status=dpio2Ioctl(fd,DPIO2_CMD_RES3_OUTPUT_VALUE_SET,FALSE);
```

Note: This command has no effect, unless the RES3 signal has been configured as output by the DPIO2_CMD_RES3_DIRECTION_SELECT command.

Related commands:

`DPIO2_CMD_RES3_DIRECTION_SELECT`

`DPIO2_CMD_RES3_VALUE_GET`

4.6.5.9 DPIO2_CMD_RES3_VALUE_GET

Description: Returns the value of the RES3 signal. Expects a pointer to a variable of type BOOL as argument, and sets this variable to TRUE if RES3 is high and FALSE if RES3 is low:

```
BOOL   res3IsHigh;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_RES3_VALUE_GET,
                    (int)&res3IsHigh);
```

Related commands:

`DPIO2_CMD_RES3_DIRECTION_SELECT`

`DPIO2_CMD_RES3_OUTPUT_VALUE_SET`

4.6.5.10 DPIO2_CMD_PIO1_DIRECTION_SELECT

Description: Configures the direction of the PIO1 signal. Expects a value of type BOOL as argument, and configures PIO1 as output if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO1_DIRECTION_SELECT,TRUE);
```

If the argument is FALSE, PIO1 is configured as input:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO1_DIRECTION_SELECT,FALSE);
```

By default PIO1 is configured as input.

Related commands:

`DPIO2_CMD_PIO1_OUTPUT_VALUE_SET`

`DPIO2_CMD_PIO1_VALUE_GET`

4.6.5.11 DPIO2_CMD_PIO1_OUTPUT_VALUE_SET

Description: Sets the output value of the PIO1 signal. Expects a value of type BOOL as argument, and sets PIO1 high if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO1_OUTPUT_VALUE_SET,TRUE);
```

If the argument is FALSE, PIO1 is set low:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO1_OUTPUT_VALUE_SET,FALSE);
```

Note: This command has no effect, unless the PIO1 signal has been configured as output by the DPIO2_CMD_PIO1_DIRECTION_SELECT command.

Related commands:

DPIO2_CMD_PIO1_DIRECTION_SELECT

DPIO2_CMD_PIO1_VALUE_GET

4.6.5.12 DPIO2_CMD_PIO1_VALUE_GET

Description: Returns the value of the PIO1 signal. Expects a pointer to a variable of type BOOL as argument, and sets this variable to TRUE if PIO1 is high and FALSE if PIO1 is low:

```
BOOL   pio1IsHigh;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_PIO1_VALUE_GET,
                     (int)&pio1IsHigh);
```

Related commands:

DPIO2_CMD_PIO1_DIRECTION_SELECT

DPIO2_CMD_PIO1_OUTPUT_VALUE_SET

4.6.5.13 DPIO2_CMD_PIO2_DIRECTION_SELECT

Description: Configures the direction of the PIO2 signal. Expects a value of type BOOL as argument, and configures PIO2 as output if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO2_DIRECTION_SELECT,TRUE);
```

If the argument is FALSE, PIO2 is configured as input:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO2_DIRECTION_SELECT,FALSE);
```

By default PIO2 is configured as input.

Related commands:

DPIO2_CMD_PIO2_OUTPUT_VALUE_SET

DPIO2_CMD_PIO2_VALUE_GET

4.6.5.14 DPIO2_CMD_PIO2_OUTPUT_VALUE_SET

Description: Sets the output value of the PIO2 signal. Expects a value of type BOOL as argument, and sets PIO2 high if this argument is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO2_OUTPUT_VALUE_SET,TRUE);
```

If the argument is FALSE, PIO2 is set low:

```
status=dpio2Ioctl(fd,DPIO2_CMD_PIO2_OUTPUT_VALUE_SET,FALSE);
```

Note: This command has no effect, unless the PIO2 signal has been configured as output by the DPIO2_CMD_PIO2_DIRECTION_SELECT command.

Related commands:

DPIO2_CMD_PIO2_DIRECTION_SELECT

DPIO2_CMD_PIO2_VALUE_GET

4.6.5.15 DPIO2_CMD_PIO2_VALUE_GET

Description: Returns the value of the PIO2 signal. Expects a pointer to a variable of type

BOOL as argument, and sets this variable to TRUE if PIO2 is high and FALSE if PIO2 is low:

```
BOOL   pio2IsHigh;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_PIO2_VALUE_GET,
                    (int)&pio2IsHigh);
```

Related commands:

`DPIO2_CMD_PIO2_DIRECTION_SELECT`

`DPIO2_CMD_PIO2_OUTPUT_VALUE_SET`

4.6.6 DMA Commands

4.6.6.1 DPIO2_CMD_DMA_SET_DESC

Description: Writes a DMA descriptor to the SRAM on the DPIO2 module.

Expects a pointer to a DPIO2_DMA_DESCRIPTOR structure as argument:

```
DPIO2_DMA_DESCRIPTOR   descriptor;

dpio2Ioctl(fd, DPIO2_CMD_DMA_SET_DESC, (int) &descriptor);
```

The DPIO2_DMA_DESCRIPTOR structure have the following fields:

descriptorId	- value that uniquely identifies the descriptor to be written.
nextDescriptorId	- value that uniquely identifies the next descriptor the DMA Controller will load when the transfer represented by this descriptor is completed.
pciAddress	- base address in PCI Memory Space where the data represented by this descriptor is written to or read from. This address must be aligned to a 4 bytes (8 bytes) boundary when 32 bits (64 bits) accesses are to be used.
blockSizeInBytes	- number of bytes to transfer. The number of bytes must be a multiple of 4 bytes (8 bytes) when 32 bits (64 bits) accesses are to be used, and its minimum value is 16.
lastBlockInChain	- boolean value which must be set to TRUE if this descriptor shall be the last in a DMA chain.
endOfBlockInterrupt	- boolean value which must be set to TRUE if the DMA Controller shall generate an interrupt when the transfer represented by this descriptor is completed.
useD64	- boolean value which must be set to TRUE if the DMA controller shall try to use D64 accesses to transfer the data represented by this descriptor. If this value is FALSE the DMA controller will use D32 accesses to transfer.

notEndOfFrame - boolean value which must be set to TRUE if the data represented by this descriptor shall not be handled as the last part of a frame.

4.6.6.2 DPIO2_CMD_DMA_SET_START_DESCRIPTOR

Description: Sets ID of the DMA descriptor that the DMA controller shall load first. Expects the ID of the start descriptor as argument:

```
UINT32   startDmaId;
status = dpio2Ioctl(fd, DPIO2_CMD_DMA_SET_START_DESCRIPTOR,
                    startDmaId);
```

4.6.6.3 DPIO2_CMD_DMA_START

Description: Starts the DMA controller:

```
status = dpio2Ioctl(fd, DPIO2_CMD_DMA_START, 0);
```

The ID of the DMA descriptor the DMA controller shall load and use for the first transfer, must be set before this command is issued.

Related commands:

`DPIO2_CMD_DMA_SET_START_DESCRIPTOR`

`DPIO2_CMD_DMA_ABORT`

`DPIO2_CMD_DMA_GET_DONE`

4.6.6.4 DPIO2_CMD_DMA_ABORT

Description: Stops the DMA controller if it is running. In order to restart afterwards, the DMA controller needs load a new DMA descriptor. This command can return the ID of the next descriptor the DMA controller would have loaded, if a pointer to a UINT32 is given as argument:

```
UINT32   nextDescriptorId;
status = dpio2Ioctl(fd,
                    DPIO2_CMD_DMA_ABORT,
                    (int) &nextDescriptorId);
```

If the ID of the next descriptor is not required, set the argument to zero:

```
status = dpio2Ioctl(fd, DPIO2_CMD_DMA_ABORT, 0);
```

Related commands:

`DPIO2_CMD_DMA_SET_START_DESCRIPTOR`

`DPIO2_CMD_DMA_START`

`DPIO2_CMD_DMA_SUSPEND`

4.6.6.5 DPIO2_CMD_FLUSH_ON_DMA_ABORT_SELECT

Selects whether data queued in DMA Controller should be flushed when a DMA transfer is aborted. Data stored in the FIFO will never be flushed when a DMA transfer is aborted.

This command expects a boolean value as parameter and enables flushing if this value is TRUE:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_FLUSH_ON_DMA_ABORT_SELECT,
                    TRUE);
```

Flushing is disabled if the boolean value is FALSE:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_FLUSH_ON_DMA_ABORT_SELECT,
                    FALSE);
```

Related commands:

`DPIO2_CMD_DMA_STOP`

4.6.6.6 DPIO2_CMD_DMA_SUSPEND

Description: Pauses the DMA controller if it is running. In order to restart, the DMA controller does not need to load a new DMA descriptor. This command can return the ID of the next descriptor the DMA controller would have loaded, if a pointer to a UINT32 is given as argument:

```
UINT32   nextDescriptorId;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_DMA_SUSPEND,
                    (int) &nextDescriptorId);
```

If the ID of the next descriptor is not required, set the argument to zero:

```
status = dpio2Ioctl(fd, DPIO2_CMD_DMA_SUSPEND, 0);
```

Related commands:

`DPIO2_CMD_DMA_RESUME`

`DPIO2_CMD_DMA_ABORT`

4.6.6.7 DPIO2_CMD_DMA_RESUME

Description: Restarts the DMA controller after it has been paused by the DPIO2_DMA_SUSPEND command.

```
status = dpio2Ioctl(fd, DPIO2_CMD_DMA_RESUME, 0);
```

The DMA controller does not load a new DMA descriptor, but continues the transfer that was paused.

Related commands:

`DPIO2_CMD_DMA_SUSPEND`

`DPIO2_CMD_DMA_START`

`DPIO2_CMD_DMA_GET_DONE`

4.6.6.8 DPIO2_CMD_DMA_GET_DONE

Description: Checks whether the DMA controller has completed a data transfer.

Expects a pointer to a BOOL variable <dmaTransferIsCompleted> as argument, and sets this variable to TRUE if the DMA controller has completed the transfer or FALSE if it has not:

```
BOOL   dmaTransferIsCompleted;

status = dpio2Ioctl(fd, DPIO2_CMD_DMA_GET_DONE,
```

```
                                    (int) &dmaTransferIsCompleted);
```

Related commands:

```
DPIO2_CMD_DMA_START
DPIO2_CMD_DMA_RESUME
```

4.6.6.9 DPIO2_CMD_REG_GET_DEMAND_MD

Description: Expects a pointer to a BOOL variable <demandModeIsEnabled> as argument, and sets this variable to TRUE if DMA Demand Mode is enabled or FALSE if not:

```
BOOL   demandModeIsEnabled;
status = dpio2Ioctl(fd, DPIO2_CMD_REG_GET_DEMAND_MD,
                    (int) &demandModeIsEnabled);
```

4.6.6.10 DPIO2_CMD_REG_SET_DEMAND_MD

Description: Enables DMA Demand Mode:

```
status = dpio2Ioctl(fd, DPIO2_CMD_REG_SET_DEMAND_MD, 0);
```

4.6.6.11 DPIO2_CMD_REG_CLR_DEMAND_MD

Description: Disables DMA Demand Mode:

```
status = dpio2Ioctl(fd, DPIO2_CMD_REG_CLR_DEMAND_MD, 0);
```

4.6.6.12 DPIO2_CMD_CONTINUE_ON_EOT_SELECT

Selects whether the DMA controller should stop the transfer on EOT, or continue on the next descriptor in the DMA chain.

This command expects a boolean value as parameter, and configures the DMA controller to continue on EOT if this value is TRUE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_CONTINUE_ON_EOT_SELECT,TRUE);
```

The DMA is configured to stop on EOT if the boolean value is FALSE:

```
status=dpio2Ioctl(fd,DPIO2_CMD_CONTINUE_ON_EOT_SELECT,FALSE);
```

Related Commands:

```
DPIO2_CMD_EOT_ENABLE
DPIO2_CMD_EOT_DISABLE
```

4.6.6.13 DPIO2_CMD_EOT_ENABLE

Description: Enables the End-Of-Transfer mechanism for a DPIO2 configured as input.

The EOT mode to be used is specified by an integer, <eotMode>:

```
status = dpio2Ioctl(fd,DPIO2_CMD_EOT_ENABLE, <eotMode>);
```

The value <eotMode> must have one of the following values:

```
DPIO2_SYNC_MARKS_END_OF_TRANSFER
DPIO2_PIO1_MARKS_END_OF_TRANSFER
DPIO2_PIO1_MARKS_END_OF_TRANSFER
DPIO2_RES1_MARKS_END_OF_TRANSFER
```

In order to have effect, this command must be issued before the FPDP interface is activated (see command DPIO2_CMD_FPDP_ACTIVATION_SELECT).

4.6.6.14 DPIO2_CMD_EOT_DISABLE

Description: Disables the End-Of-Transfer mechanism for a DPIO2 configured as input:

```
status = dpio2Ioctl(fd, DPIO2_CMD_EOT_DISABLE, 0);
```

4.6.6.15 DPIO2_CMD_EOT_COUNT_ENABLE

This command enables the EOT count mechanism, which makes the DPIO2 ait for a specific number of EOT marks before it terminates a DMA transfer. The number of EOT marks is specified by a 16 bits unsigned integer given as argument to the command:

```
UINT16   numEotMarks;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_EOT_COUNT_ENABLE,
                    (int) numEotMarks);
```

Note: The EOT count mechanism is only applicable when using the SYNC signal as EOT mark.

Related Commands:

```
DPIO2_CMD_EOT_ENABLE
DPIO2_CMD_EOT_DISABLE
DPIO2_CMD_EOT_COUNT_DISABLE
```

4.6.6.16 DPIO2_CMD_EOT_COUNT_DISABLE

This command disables the EOT count mechanism, making the DPIO2 terminate a DMA transfer on the first EOT mark it detects. The command takes no arguments:

```
status = dpio2Ioctl(fd, DPIO2_CMD_EOT_COUNT_ENABLE, 0);
```

Note: The EOT count mechanism is only applicable when using the SYNC signal as EOT mark.

Related Commands:

```
DPIO2_CMD_EOT_ENABLE
DPIO2_CMD_EOT_DISABLE
DPIO2_CMD_EOT_COUNT_ENABLE
```

4.6.6.17 DPIO2_CMD_REMAINING_BYTE_COUNT_GET

This command returns the number of bytes that were specified for transfer by the last DMA descriptor but not transferred when the DMA controller terminated the command.

The command expects a pointer to a DPIO2_REMAINING_BYTE_COUNT_INFO structure as argument:

```
DPIO2_REMAINING_BYTE_COUNT_INFO   byteCountInfo;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_REMAINING_BYTE_COUNT_GET,
```

```
                                (int) &byteCountInfo);
```

The DPIO2_REMAINING_BYTE_COUNT_INFO contains two fields:

byteCount - the number of bytes that were specified for transfer by the last DMA descriptor but not transferred

overflowFlag - boolean value which signals whether a previous value of the byte count was overwritten before it was read

Related Commands:

DPIO2_CMD_TRANSFERRED_BYTE_COUNT_GET

4.6.6.18 DPIO2_CMD_TRANSFERRED_BYTE_COUNT_GET

This command returns the number of bytes that were already transferred by the last DMA descriptor, when the DMA controller terminated the transfer.

The command expects a pointer to an unsigned 32 bits integer as argument:

```
UINT32   byteCount;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_TRANSFERRED_BYTE_COUNT_GET,
                    (int) &byteCount);
```

Related Commands:

DPIO2_CMD_REMAINING_BYTE_COUNT_GET

4.6.7 Interrupt Commands

4.6.7.1 DPIO2_CMD_INTERRUPT_ENABLE

Description: Enables an interrupt.

Expects an integer as argument which specifies what interrupt to enable:

```
status=dpio2Ioctl(fd,DPIO2_CMD_INTERRUPT_ENABLE,<interrupt>);
```

The following vales can be used for <interrupt>:

DPIO2_INT_COND_DMA_DONE_CHAIN

DPIO2_INT_COND_DMA_DONE_BLOCK

DPIO2_INT_COND_FIFO_NOT_EMPTY

DPIO2_INT_COND_FIFO_EMPTY

DPIO2_INT_COND_FIFO_ALMOST_EMPTY

DPIO2_INT_COND_FIFO_HALF_FULL

DPIO2_INT_COND_FIFO_ALMOST_FULL

DPIO2_INT_COND_FIFO_FULL

DPIO2_INT_COND_FIFO_OVERFLOW

DPIO2_INT_COND_PIO1

DPIO2_INT_COND_PIO2

DPIO2_INT_COND_SYNC

```
DPIO2_INT_COND_SUSPEND
DPIO2_INT_COND_TARGET_ABORT
DPIO2_INT_COND_MASTER_ABORT
```

4.6.7.2 DPIO2_CMD_INTERRUPT_DISABLE

Description: Disables an interrupt.

Expects an integer as argument which specifies what interrupt to disable:

```
status = dpio2Ioctl(fd,
                    DPIO2_CMD_INTERRUPT_DISABLE,
                    <interrupt>);
```

The following vales can be used for <interrupt>:

```
DPIO2_INT_COND_DMA_DONE_CHAIN
DPIO2_INT_COND_DMA_DONE_BLOCK
DPIO2_INT_COND_FIFO_NOT_EMPTY
DPIO2_INT_COND_FIFO_EMPTY
DPIO2_INT_COND_FIFO_ALMOST_EMPTY
DPIO2_INT_COND_FIFO_HALF_FULL
DPIO2_INT_COND_FIFO_ALMOST_FULL
DPIO2_INT_COND_FIFO_FULL
DPIO2_INT_COND_FIFO_OVERFLOW
DPIO2_INT_COND_PIO1
DPIO2_INT_COND_PIO2
DPIO2_INT_COND_SYNC
DPIO2_INT_COND_SUSPEND
DPIO2_INT_COND_TARGET_ABORT
DPIO2_INT_COND_MASTER_ABORT
```

4.6.7.3 DPIO2_CMD_INTERRUPT_CALLBACK_ATTACH

Description: Attaches a callback function to a specified interrupt condition.

Note: It is not recommended to perform any DPIO2 API calls inside the interrupt callback routine. The callback routine should instead signal (ie semaphore) the waiting thread.

Expects a pointer to a DPIO2_INTERRUPT_CALLBACK structure as argument:

```
DPIO2_INTERRUPT_CALLBACK   callback;
status = dpio2Ioctl(fd, DPIO2_CMD_INTERRUPT_CALLBACK_ATTACH,
                    (int) &callback);
```

The DPIO2_INTERRUPT_CALLBACK structure has the following fields:

condition - value identifying the interrupt condition which the callback function shall signal

pCallbackFunction - pointer to the callback function. The callback

function must take an integer (int) as argument and return an integer (int).

argument - argument to be passed to the callback function each time it is called.

The condition field must have one of the following values:

```
DPIO2_INT_COND_DMA_DONE_CHAIN
DPIO2_INT_COND_DMA_DONE_BLOCK
DPIO2_INT_COND_FIFO_NOT_EMPTY
DPIO2_INT_COND_FIFO_EMPTY
DPIO2_INT_COND_FIFO_ALMOST_EMPTY
DPIO2_INT_COND_FIFO_HALF_FULL
DPIO2_INT_COND_FIFO_ALMOST_FULL
DPIO2_INT_COND_FIFO_FULL
DPIO2_INT_COND_FIFO_OVERFLOW
DPIO2_INT_COND_PIO1
DPIO2_INT_COND_PIO2
DPIO2_INT_COND_SYNC
DPIO2_INT_COND_SUSPEND
DPIO2_INT_COND_TARGET_ABORT
DPIO2_INT_COND_MASTER_ABORT
```

4.6.7.4 DPIO2_CMD_INTERRUPT_CALLBACK_DETACH

Description: Detaches a callback function from a specified interrupt condition.

Expects an integer as argument which specifies the interrupt condition to detach the callback function from:

```
status = dpio2Ioctl(fd, DPIO2_CMD_INTERRUPT_CALLBACK_DETACH,
                    <interrupt>);
```

The following vales can be used for <interrupt>:

```
DPIO2_INT_COND_DMA_DONE_CHAIN
DPIO2_INT_COND_DMA_DONE_BLOCK
DPIO2_INT_COND_FIFO_NOT_EMPTY
DPIO2_INT_COND_FIFO_EMPTY
DPIO2_INT_COND_FIFO_ALMOST_EMPTY
DPIO2_INT_COND_FIFO_HALF_FULL
DPIO2_INT_COND_FIFO_ALMOST_FULL
DPIO2_INT_COND_FIFO_FULL
DPIO2_INT_COND_FIFO_OVERFLOW
DPIO2_INT_COND_PIO1
DPIO2_INT_COND_PIO2
DPIO2_INT_COND_SYNC
```

```
DPIO2_INT_COND_SUSPEND
DPIO2_INT_COND_TARGET_ABORT
DPIO2_INT_COND_MASTER_ABORT
```

4.6.7.5 DPI02_CMD_AUTO_DISABLE_INTERRUPT_ENABLE

Description: Enables automatical disabling of interrupt source when the first interrupt from it is serviced. Expects an integer as argument which specifies the interrupt:

```
status = dpio2Ioctl(fd,
                    DPI02_CMD_AUTO_DISABLE_INTERRUPT_ENABLE,
                    <interrupt>);
```

The following vales can be used for <interrupt>:

```
DPIO2_INT_COND_DMA_DONE_CHAIN
DPIO2_INT_COND_DMA_DONE_BLOCK
DPIO2_INT_COND_FIFO_NOT_EMPTY
DPIO2_INT_COND_FIFO_EMPTY
DPIO2_INT_COND_FIFO_ALMOST_EMPTY
DPIO2_INT_COND_FIFO_HALF_FULL
DPIO2_INT_COND_FIFO_ALMOST_FULL
DPIO2_INT_COND_FIFO_FULL
DPIO2_INT_COND_FIFO_OVERFLOW
DPIO2_INT_COND_PIO1
DPIO2_INT_COND_PIO2
DPIO2_INT_COND_SYNC
DPIO2_INT_COND_SUSPEND
DPIO2_INT_COND_TARGET_ABORT
DPIO2_INT_COND_MASTER_ABORT
```

4.6.7.6 DPI02_CMD_AUTO_DISABLE_INTERRUPT_DISABLE

Description: Disables automatical disabling of interrupt source when the first interrupt from it is serviced. Expects an integer as argument which specifies the interrupt:

```
status = dpio2Ioctl(fd,
                    DPI02_CMD_AUTO_DISABLE_INTERRUPT_DISABLE,
                    <interrupt>);
```

The following vales can be used for <interrupt>:

```
DPIO2_INT_COND_DMA_DONE_CHAIN
DPIO2_INT_COND_DMA_DONE_BLOCK
DPIO2_INT_COND_FIFO_NOT_EMPTY
DPIO2_INT_COND_FIFO_EMPTY
DPIO2_INT_COND_FIFO_ALMOST_EMPTY
DPIO2_INT_COND_FIFO_HALF_FULL
DPIO2_INT_COND_FIFO_ALMOST_FULL
```

```
DPIO2_INT_COND_FIFO_FULL
DPIO2_INT_COND_FIFO_OVERFLOW
DPIO2_INT_COND_PIO1
DPIO2_INT_COND_PIO2
DPIO2_INT_COND_SYNC
DPIO2_INT_COND_SUSPEND
DPIO2_INT_COND_TARGET_ABORT
DPIO2_INT_COND_MASTER_ABORT
```

4.6.8 FIFO Commands

4.6.8.1 DPIO2_CMD_FIFO_FLUSH

Description: Flushes any data currently stored in the FIFO:

```
status = dpio2Ioctl(fd, DPIO2_CMD_FIFO_FLUSH, 0);
```

4.6.8.2 DPIO2_CMD_GET_CURRENT_FIFO_STATUS

Description: Gets the current state of the FIFO.

Expects a pointer to a variable of type int as argument, and sets this variable to a constant representing the current state:

```
int   fifoState;
status = dpio2Ioctl(fd, DPIO2_CMD_GET_CURRENT_FIFO_STATUS,
                    (int) &fifoState);
```

The constant returned in <fifoState> will be one of the following:

```
DPIO2_FIFO_EMPTY
DPIO2_FIFO_ALMOST_EMPTY
DPIO2_FIFO_LESS_THAN_HALF_FULL
DPIO2_FIFO_GREATER_THAN_HALF_FULL
DPIO2_FIFO_ALMOST_FULL
DPIO2_FIFO_FULL
```

4.6.8.3 DPIO2_CMD_RESET_OCCURRED_FLAGS

Description: Resets history flags which tell what states the FIFO has been in:

```
status = dpio2Ioctl(fd, DPIO2_CMD_RESET_OCCURRED_FLAGS, 0);
```

Note: In order to clear the history flag for FIFO Overflow, the FPDP interface must first be deactivated and then reactivated before the DPIO2_CMD_RESET_OCCURRED_FLAGS command is issued:

```
dpio2Ioctl(fd, DPIO2_CMD_FPDP_ACTIVATION_SELECT, FALSE);
dpio2Ioctl(fd, DPIO2_CMD_FPDP_ACTIVATION_SELECT, TRUE);
dpio2Ioctl(fd, DPIO2_CMD_RESET_OCCURRED_FLAGS, 0);
```

4.6.8.4 DPIO2_CMD_GET_FIFO_OVERFLOW_OCCURRED_FLAG

Description: Gets the flag which tells whether the FIFO has overflowed since the last time the occurred flags were reset (see DPIO2_CMD_RESET_OCCURRED_FLAGS).

Expects a pointer to a BOOL variable as argument, and sets this variable to TRUE if the FIFO has overflowed or FALSE if not:

```
BOOL   fifoHasOverflowed;

status=dpio2Ioctl(fd,
                  DPIO2_CMD_GET_FIFO_OVERFLOW_OCCURRED_FLAG,
                  (int) &fifoHasOverflowed);
```

4.6.8.5 DPIO2_CMD_GET_FIFO_FULL_OCCURRED_FLAG

Description: Gets the flag which tells whether the FIFO has been full since the last time the occurred flags were reset (see DPIO2_CMD_RESET_OCCURRED_FLAGS).

Expects a pointer to a BOOL variable as argument, and sets this variable to TRUE if the FIFO has been full or FALSE if not:

```
BOOL   fifoHasBeenFull;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_GET_FIFO_FULL_OCCURRED_FLAG,
                    (int) &fifoHasBeenFull);
```

4.6.8.6 DPIO2_CMD_GET_FIFO_ALMOST_FULL_OCCURRED_FLAG

Description: Gets the flag which tells whether the FIFO has been almost full since the last time the occurred flags were reset (see DPIO2_CMD_RESET_OCCURRED_FLAGS).

Expects a pointer to a BOOL variable as argument, and sets this variable to TRUE if the FIFO has been almost full or FALSE if not:

```
BOOL   fifoHasBeenAlmostFull;

status = dpio2Ioctl(fd,
                 DPIO2_CMD_GET_FIFO_ALMOST_FULL_OCCURRED_FLAG,
                 (int) &fifoHasBeenAlmostFull);
```

4.6.8.7 DPIO2_CMD_GET_FIFO_HALF_FULL_OCCURRED_FLAG

Description: Gets the flag which tells whether the FIFO has been more than half full since the last time the occurred flags were reset (see DPIO2_CMD_RESET_OCCURRED_FLAGS).

Expects a pointer to a BOOL variable as argument, and sets this variable to TRUE if the FIFO has been half full or FALSE if not:

```
BOOL   fifoHasBeenHalfFull;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_GET_FIFO_HALF_FULL_OCCURRED_FLAG,
                    (int) &fifoHasBeenHalfFull);
```

4.6.8.8 DPIO2_CMD_GET_FIFO_ALMOST_EMPTY_OCCURRED_FLAG

Description: Gets the flag which tells whether the FIFO has been almost empty since the last

time the occurred flags were reset (see DPIO2_CMD_RESET_OCCURRED_FLAGS).

Expects a pointer to a BOOL variable as argument, and sets this variable to TRUE if the FIFO has been almost empty or FALSE if not:

```
BOOL   fifoHasBeenAlmostEmpty;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_GET_FIFO_ALMOST_EMPTY_OCCURRED_FLAG,
                    (int) &fifoHasBeenAlmostEmpty);
```

4.6.8.9 DPIO2_CMD_GET_FIFO_EMPTY_OCCURRED_FLAG

Description: Gets the flag which tells whether the FIFO has been empty since the last time the occurred flags were reset (see DPIO2_CMD_RESET_OCCURRED_FLAGS).

Expects a pointer to a BOOL variable as argument, and sets this variable to TRUE if the FIFO has been empty or FALSE if not:

```
BOOL   fifoHasBeenEmpty;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_GET_FIFO_EMPTY_OCCURRED_FLAG,
                    (int) &fifoHasBeenEmpty);
```

4.6.8.10 DPIO2_CMD_GET_PTR_FIFO

Description: Gets a pointer to the FIFO of the DPIO2 module.

Expects a pointer to a UINT32 pointer as argument, and sets this pointer variable equal to the local CPU address where the FIFO is mapped:

```
UINT32*   pFifo;

status = dpio2Ioctl(fd,
                    DPIO2_CMD_GET_PTR_FIFO,
                    (int) &pFifo);
```

4.6.8.11 DPIO2_CMD_GET_DEVICE_FSIZE

Description: Gets the size of the FIFO in words.

Expects a pointer to a variable of type UINT32 as argument, and sets this variable to the size of the FIFO.

```
UINT32   fifoSizeInWords;

status = dpio2Ioctl(fd, DPIO2_CMD_GET_DEVICE_FSIZE,
                    (int) &fifoSizeInWords);
```

4.6.9 Miscellaneous Commands

4.6.9.1 DPIO2_CMD_LATENCY_TIMER_GET

Description: Gets the current value of the PCI Latency Timer.

Expects a pointer to a UINT8 variable as argument, and sets this variable to the value of the PCI Latency Timer:

```
UINT8   latencyTimer;
status = dpio2Ioctl(fd,
                    DPIO2_CMD_LATENCY_TIMER_GET,
                    (int) &latencyTimer);
```

4.6.9.2 DPIO2_CMD_LATENCY_TIMER_SET

Description: Sets a new value for the PCI Latency Timer.

Expects UINT8 value as argument, and the Latency Timer equal to this:

```
UINT8   latencyTimer;
status = dpio2Ioctl(fd,
                    DPIO2_CMD_LATENCY_TIMER_SET,
                    latencyTimer);
```

4.6.9.3 DPIO2_CMD_GET_DEVICE_BUSNUM

Description: Gets the PCI bus number of the PMC slot where the DPIO2 module is fitted.

Expects a pointer to a variable of type int as argument, and stores the PCI bus number in this variable:

```
int   pciBusNumber;
status = dpio2Ioctl(fd,
                    DPIO2_CMD_GET_DEVICE_BUSNUM,
                    (int) &pciBusNumber);
```

4.6.9.4 DPIO2_CMD_GET_DEVICE_DEVNUM

Description: Gets the PCI device number of the PMC slot where the DPIO2 module is fitted.

Expects a pointer to a variable of type int as argument, and stores the PCI device number in this variable:

```
int   pciDeviceNumber;
status = dpio2Ioctl(fd, DPIO2_CMD_GET_DEVICE_DEVNUM,
                    (int) &pciDeviceNumber);
```

APPENDIX A. DOWNLOADING FPGA CODE

This appendix describes the procedure for downloading new FPGA code to the FLASH on a DPIO2 module:

1. Set jumper JP8 to its UP-position if the FLASH of the DPIO2 is empty. If the FLASH of the DPIO2 contains previously downloaded FPGA images, jumper JP8 may be set to its DOWN-position during the whole update procedure.
2. If a DPIO2 66MHz module is used, make sure that SW1-1 (switch 1 on DIP switch SW1) is ON. Otherwise the FLASH will be write-protected.
3. Go to the directory where the setup program placed the DPIO2 Device Driver. (dpio2-drv-src/lib/x86-linux-2.6.x)
4. Update the FPGA-images in the DPIO2 FLASH by running the FpgaFlashLoad program:

   ```
   FpgaFlashLoad <dpio2-device-number>
   ```

   The download program needs to get the name of the FPGA image file (include the directory name if necessary):

   ```
   Name of EXO file: <exo-file-name>
   ```

5. Make sure that jumper JP8 is in the DOWN-position. This makes the DPIO2 load the newly downloaded FPGA images after power-up.
6. Turn power off and on again. This is necessary because the FPGAs on a DPIO2 module are only initialised after power-up.