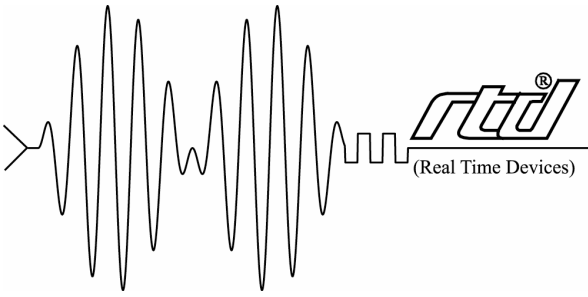
DM6420 Driver for Linux

User's Manual
Version 2.01.xx



RTD Embedded Technologies, Inc.
"Accessing the Analog World"®

SWM-640010014
rev C

ISO9001 and AS9100 Certified

RTD Embedded Technologies, INC.
103 Innovation Blvd.
State College, PA 16803-0906

Phone: +1-814-234-8087
FAX: +1-814-234-5218

E-mail
sales@rtd.com
techsupport@rtd.com

web site
http://www.rtd.com

Revision History

04/20/2004	Revision A issued Documented for ISO9000
07/14/2005	Revision B issued Removed duplicate word in “Using the API Functions” section
4/11/2008	Revision C issued Released new version of the driver that supports 2.6 kernels Updated the driver revision number and documented changes related to the 2.6 kernel Added dm6420-dual-dma-data-viewer example program

DM6420 Driver for Linux
Published by:

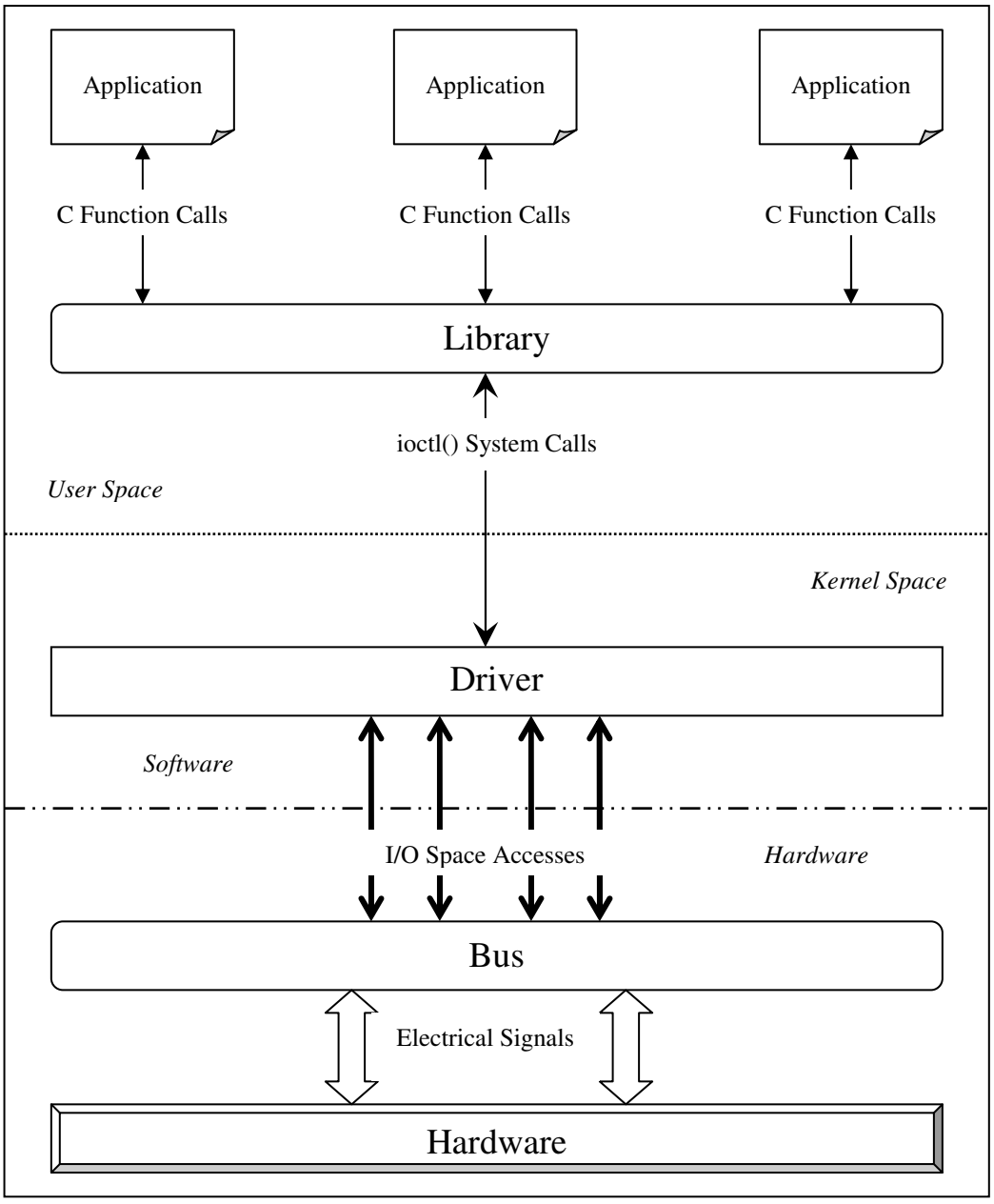
RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College, PA 16803-0906

Table of Contents

Introduction

This document targets anyone wishing to write Linux applications for an RTD DM6420 analog I/O dataModule. It provides information on building the software and about the Application Programming Interface used to communicate with the hardware and driver. Each high-level library function is described as well as any low-level ioctl() system call interface it may make use of.

The diagram below 1) provides a general overview of what hardware and software entities are involved in device access, 2) shows which units communicate with each other, and 3) illustrates the methods used to transfer data and control information.



Installation Instructions

Extracting the Software

All software comes packaged in a gzip'd tar file named dm6420_Linux_v2.01.xx.tar.gz. First, decide where you would like to place the software and make a copy of the tar file in this directory by issuing the command "cp <path to tar file>/ dm6420_Linux_v2.01.xx.tar.gz <installation path>" substituting the appropriate paths for <path to tar file> and <installation path>. Next, change your current directory to where you made a copy of the tar file by issuing the command "cd <installation path>". Once you are in this directory, extract the software by issuing the command "tar –xvzf dm6420_Linux_v2.01.xx.tar.gz"; this will create a directory dm6420_Linux_v2.01.xx.tar.gz / which contains all the files that are part of the software package.

Contents of Installation Directory

Once the tar file is extracted, you should see the following files and directories within dm6420_Linux_v2.01.xx/:

driver/
examples/
include/
lib/
CHANGES.TXT
LICENSE.TXT
README.TXT

The file CHANGES.TXT describes the changes made to the software for this release, as well as for previous releases. The file LICENSE.TXT provides details about the RTD end user license agreement which must be agreed to and accepted before using this software. The file README.TXT contains a general overview of the software and contact information should you experience problems, have questions, or need information. The directory driver/ contains the source code and Makefile for the drivers. The directory examples/ holds the source code and Makefile for the example programs. The directory include/ contains all header files used by the driver, example programs, library, and your application programs. Library source code and Makefile reside in the directory lib/.

Building the Driver

Driver source code uses files located in the kernel source tree. Therefore, you must have the full kernel source tree available in order to build the driver. The kernel source tree consumes a lot of disk space, on the order of 100 to 200 megabytes. Because production systems rarely contain this much disk space, you will probably use a development machine to compile the driver source code. The development system, which provides a full compilation environment, must be running the exact same version of the kernel as your production machine(s); otherwise the kernel module may not load or may load improperly. After the code is built, you can then move the resulting object files, libraries, and executables to the production system(s).

Building the driver consists of several steps: 1) compiling the source code, 2) loading the resulting kernel module into the kernel, and 3) creating hardware device files in the /dev directory. To perform any of the above steps, you must change your current directory to driver/. The file Makefile contains rules to assist you.

To compile the source code, issue the command “make”. This command will create the driver object file. The name of the driver object file depends on the version of the kernel being used. If compiling with a 2.4 kernel, the filename will be rtd-dm6420.o and if compiling with a 2.6 kernel the filename will be rtd-dm6420.ko. The GNU C compiler gcc is used to build the driver code.

Before the driver can be used, it must be loaded into the currently running kernel. Using the command “make insmod” will load the DM6420 driver into the kernel. This target assumes that:

* A single DM6420 is installed.
* The board’s base I/O address is set to the factory default of 0x300.
* The DM6420 will use IRQs 3 and 5.
* The board will use DMA channels 5 and 6.

If the previous assumptions do not match your hardware setup, you will need to edit the Makefile and change this rule to reflect your board configuration or manually issue an appropriate insmod command.

When you load the kernel driver, you might see the following message:
“Warning: loading ./rtd-dm6420.o will taint the kernel: no license”
You can safely ignore this message since it pertains to GNU General Public License (GPL) licensing issues rather than to driver operation.

The final step is to create /dev entries for the hardware. Previous versions of the driver always assumed a character device major number of 240 when registering the boards and creating the /dev entries. Instead, the driver now asks the kernel to dynamically assign a major number. Since this major number may change each time you load the driver, the best way to create the device files is to use the command “make devices”; this generates four files in /dev named rtd-dm6420-0 through rtd-dm6420-3.

If you ever need to unload the driver from the kernel, you can use the command “make rmmod”.

Building the Library

The example programs and your application use the DM6420 library, so it must be built before any of these can be compiled. To build the library, change your current directory to lib/ and issue the command “make”. The GNU C compiler gcc is used to compile the library source code.

The DM6420 library is statically linked and is created in the file librtd-dm6420.a.

Building the Example Programs

The example programs may be compiled by changing your current directory to examples/ and giving the command “make”, which builds all the example programs. If you wish to compile a subset of example programs, there are targets in Makefile to do so. For example, the command “make dm6420-auto-burst” will compile and link the source file dm6420-auto-burst.cc and the command “make dm6420-stream” will compile and link the source file dm6420-stream.cc. The GNU C compiler gcc is used to compile the example program code.

Using the API Functions

DM6420 hardware and the associated driver functionality can be accessed through the library API (Application Programming Interface) functions. Applications wishing to use library functions must include the include/dm6420lib.h header file and be statically linked with the lib/librtd-dm6420.a library file.

The following function reference provides for each library routine a prototype, description, explanation of parameters, and return value or error code. By looking at a function's entry, you should gain an idea of: 1) why it would be used, 2) what it does, 3) what information is passed into it, 4) what information it passes back, 5) how to interpret error conditions that may arise, and 6) the ioctl() system call interface if the function makes use of a single ioctl() call.

Note that errno codes other than the ones indicated in the following pages may be set by the library functions. Unless otherwise noted in the description of a function's return value, please see the ioctl(2) man page for more information

Function Reference

API Function Groups

Analog to Digital Conversion

ChannelGainDataStore6420
ClearADFIFO6420
ClearChannelGainTable6420
EnableTables6420
GetAutoincData6420
IsAboutTrigger6420
IsADConverting6420
IsADFIFOEmpty6420
IsADFIFOFull6420
IsADHalted6420
LoadADSampleCounter6420
LoadADTable6420
LoadDigitalTable6420
LoadTriggerRegister6420
ReadADData6420
ReadADDataMarker6420
ReadADDataWithMarker6420
ReadChannelGainDataStore6420
ResetChannelGainTable6420
SetBurstTrigger6420
SetChannelGain6420
SetConversionSelect6420
SetPauseEnable6420
SetStartTrigger6420
SetStopTrigger6420
SetTriggerPolarity6420
SetTriggerRepeat6420
StartConversion6420

Digital to Analog Conversion

LoadDAC6420

Digital I/O

ClearDINFIFO6420
ConfigDINClock6420
DIOClearChip6420
DINClockEnable6420
DIOClearIrq6420
DIOEnableIrq6420
DIOIsChipIrq6420
DIOIsChipIRQEnabled6420
DIOIsChipIRQEventMode6420
DIOIsChipPort1Output6420
DIOIsChipStrobe6420
DIOIsChipSystemClock6420
DIOLoadCompare6420
DIOLoadMask6420
DIORead6420
DIOReadCompareRegister6420
DIOReadStatus6420
DIOSelectClock6420
DIOSelectIrqMode6420
DIOSelectRegister6420
DIOSetPort0Direction6420
DIOSetPort1Direction6420
DIOWrite6420
IsDigitalIRQ6420
IsDINFIFOEmpty6420
IsDINFIFOFull6420
IsDINFIFOHalf6420
LoadDINConfigRegister6420
ReadDINFIFO6420

Direct Memory Access (DMA) Data Transfer

ClearADDMADone6420
DeInstallDMA6420
GetDmaData6420
InstallDMA6420
IsADDMADone6420
IsFirstADDMADone6420
StartDMA6420
StopDMA6420

General

ClearBoard6420
ClearRegister6420
CloseBoard6420
InitBoard6420
LoadControlRegister6420
OpenBoard6420
ReadStatus6420

Interrupt Control and Status

ClearIRQ06420
ClearIRQ16420
DisableIRQ6420
EnableIRQ6420
GetIRQCounter6420
InstallCallbackIRQHandler6420
LoadIRQRegister6420
RemoveIRQHandler6420
SetIRQ0Source6420
SetIRQ1Source6420

Timer/Counter Control and Status

ClockDivisor6420
ClockMode6420
DoneTimer6420
IsBurstClockOn6420
IsPacerClockOn6420
ReadTimerCounter6420
SelectTimerCounter6420
SetBurstClock6420
SetPacerClock6420
SetPacerClockSource6420
SetSampleCounterStop6420
SetUserClock6420

ChannelGainDataStore6420

int ChannelGainDataStore6420(int descriptor, int Enable);

Description:

Enable or disable a board's channel/gain data store.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Enable:	Flag to indicate whether the channel/gain data store should be enabled. A value of 0 means disable the data store. A nonzero value means enable the data store.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request ;

/*
 * Write to the Control Register at I/O base address + 2
*/

io_request.reg = r_CONTROL_6420;

/*
 * Only change bit 4 in the Control Register
 */

io_request.mask = 0xFFEF;

/*
 * Enable the channel/gain data store
 */

io_request.value = (1 << 4);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

ClearADDMADone6420

int ClearADDMADone6420(int descriptor);

Description:

Clear a board's A/D DMA done flag.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_AD_DMA_DONE);

ClearADFIFO6420

int ClearADFIFO6420(int descriptor);

Description:

Clear a board's A/D sample FIFO.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_AD_FIFO);

ClearBoard6420

int ClearBoard6420(int descriptor);

Description:

Reset a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_BOARD);

ClearChannelGainTable6420

int ClearChannelGainTable6420(int descriptor);

Description:

Clear the contents of a board's channel/gain table.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_CLEAR_GAIN);

ClearDINFIFO6420

int ClearDINFIFO6420(int descriptor);

Description:

Clear a board's digital input FIFO.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_DIO_FIFO);
```

ClearIRQ06420

int ClearIRQ06420(int descriptor);

Description:

Clear the first interrupt circuit on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_IRQ1);
```

ClearIRQ16420

int ClearIRQ16420(int descriptor);

Description:

Clear the second interrupt circuit on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_IRQ2);

ClearRegister6420

int ClearRegister6420(int descriptor, u_int16_t ClearValue);

Description:

Write a bit mask into a board's Program Clear Register and then read from the Clear Register to clear some part(s) of the board. Other library functions make use of this routine to perform their work.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

ClearValue: Bit mask to write into Program Clear Register before reading Clear Register.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;

/*
 * Clear the contents of board's channel/gain table.
 */

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_CLEAR_GAIN);
```

ClockDivisor6420

---

int ClockDivisor6420(int descriptor, enum DM6420HR_CLK Timer, u_int16_t Divisor);

Description:

Set the divisor for the specified counter on the 8254 chip. Before calling this function, the counter must have been set to receive the divisor least significant byte first then most significant byte.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Timer:	Indicates which counter to use. Valid values are DM6420HR_CLK0, DM6420HR_CLK1, and DM6420HR_CLK2.
Divisor:	Counter divisor value.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Timer is not valid.

IOCTL Interface:

This function makes use of several ioctl() requests.

ClockMode6420

```
int ClockMode6420(
   int descriptor,
   enum DM6420HR_CLK Timer,
   enum DM6420HR_CLK_MODE Mode
);
```

Description:

Set the mode of the specified counter on the 8254 chip. This function also sets the indicated counter to receive a subsequent divisor load least significant byte first then most significant byte.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Timer:	Indicates which counter to use. Valid values are DM6420HR_CLK0, DM6420HR_CLK1, and DM6420HR_CLK2.
Mode:	Indicates which clock mode to set. Valid values are DM6420HR_CLK_MODE0, DM6420HR_CLK_MODE1, DM6420HR_CLK_MODE2, DM6420HR_CLK_MODE3, DM6420HR_CLK_MODE4, and DM6420HR_CLK_MODE5.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	Timer is not valid.
	EINVAL	Mode is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Write to Timer/Counter Control Register at I/O base address + 22
 */

io_request.reg = r_TIMER_CTRL_6420;

/*
 * Operate on timer/counter 2
 */

io_request.value = ((DM6420HR_CLK2 & 0x3) << 6);

/*
 * Put timer/counter into Mode 2 (rate generator mode)
 */

io_request.value |= ((DM6420HR_CLK_MODE2 & 0x7) << 1);
```

```
/*
 * Set timer/counter value to be accessed least significant byte first then most significant byte
 */

io_request.value |= 0x30;

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTB, &io_request);
```

CloseBoard6420

int CloseBoard6420(int descriptor);

Description:

Close a DM6420 device file opened previously with OpenBoard6420().

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.

Return Value:

0:	Success.
-1:	Failure. Please see the close(2) or ioctl(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

ConfigDINClock6420

int ConfigDINClock6420(int descriptor, enum DM6420HR_DI_FIFO_CLK DIN_Clock);

Description:

Set the source for a board's digital input FIFO clock.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
DIN_Clock:	Source of digital input FIFO clock. Valid values are DM6420HR_DI_FIFO_CLK_USER_TC0, DM6420HR_DI_FIFO_CLK_USER_TC1, DM6420HR_DI_FIFO_CLK_AD_WRITE_FIFO, DM6420HR_DI_FIFO_CLK_EXTERNAL_PACER, and DM6420HR_DI_FIFO_CLK_EXTERNAL_TRIG.

Return Value:

0:	Success.
-1	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	DIN_Clock is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Digital Input FIFO Configuration Register at base I/O address + 10
 */

io_request.reg = r_DIN_CONFIG_6420;

/*
 * Only change bits 0 through 2 in Configuration Register
 */

io_request.mask = 0xFFF8;

/*
 * Set source for digital input FIFO clock to the output of User Timer/Counter 0
 */

io_request.value = DM6420HR_DI_FIFO_CLK_USER_TC0;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

DeInstallDMA6420

int DeInstallDMA6420(int descriptor, enum DM6420HR_DMA DMAChannel);

Description:

Configure the specified DMA circuit on a board so that DMA can no longer be performed on it.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
DMAChannel:	DMA circuit to configure. Valid values are DM6420HR_DMA1 and DM6420HR_DMA2.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	DMAChannel is not valid.
EINVAL	No DMA channel was ever allocated to DMAChannel.

IOCTL Interface:

```
int rc;
struct DM6420HR_DI io_request;

/*
 * Operate on board's first DMA circuit
 */

io_request.dma = DM6420HR_DMA1;

/*
 * DMA can no longer be performed on the circuit
 */

io_request.action = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_INSTALL, &io_request);
```

DINClockEnable6420

```
int DINClockEnable6420(int descriptor, int Enable);
```

Description:

Enable or disable a board's digital input FIFO clock.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Enable:	Indicates whether or not to enable the digital input FIFO clock. A value of 0 means disable the clock. A nonzero value means enable the clock.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Digital Input FIFO Configuration Register at base I/O address + 10
 */

io_request.reg = r_DIN_CONFIG_6420;

/*
 * Only change bit 3 in Configuration Register
 */

io_request.mask = 0xFFF7;

/*
 * Disable digital input FIFO clock
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

DIOClearChip6420

---

int DIOClearChip6420(int descriptor);

Description:

Clear a board's digital I/O chip.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOClearIrq6420

---

int DIOClearIrq6420(int descriptor);

Description:

Clear a board's digital I/O IRQ status flag.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOEnableIrq6420

---

int DIOEnableIrq6420(int descriptor, int Enable);

Description:

Enable or disable a board's digital interrupts.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Enable: Flag indicating how digital interrupts should be set. A value of 0 means disable digital interrupts. A nonzero value means enable digital interrupts.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO8 io_request;

/*
 * Write to Digital Mode Register at base I/O address + 30
 */

io_request.reg = r_DIO_MODE_6420;

/*
 * Only change bit 4 in Mode Register
 */

io_request.mask = 0xEF;

/*
 * Enable digital interrupts
 */

io_request.value = (1 << 4);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTB, &io_request);
```

DIOIsChipIrq6420

---

int DIOIsChipIrq6420(int descriptor, int *interrupt_generated_p);

Description:

Determine whether or not a board has generated a digital I/O interrupt.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Interrupt_generated_p: Address where interrupt generated flag should be stored. If the board has generated a digital I/O interrupt, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from the Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
if (rc != -1) {

   /*
    * If bit 6 is set in Status Register, an interrupt occurred
    */

   if (io_request.value & 0x40) {
      fprintf(stdout, "Digital I/O interrupt generated.\n");
   }
}
```

DIOIsChipIRQEnabled6420

---

int DIOIsChipIRQEnabled6420(int descriptor, int *irq_enabled_p);

Description:

Determine whether or not digital interrupts are enabled for a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Irq_enabled_p:	Address where IRQ enabled flag should be stored. If digital interrupts are enabled, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from the Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
if (rc != -1) {

   /*
    * If bit 4 is cleared in Status Register, digital I/O interrupts are disabled
    */

   if ((io_request.value & 0x10) == 0) {
     fprintf(stdout, “Digital I/O interrupts disabled.\n”);
   }
}
```

DIOIsChipIRQEventMode6420

int DIOIsChipIRQEventMode6420(int descriptor, int *irq_event_p);

Description:

Determine whether or not a board’s digital interrupt mode is set to event.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Irq_event_p: Address where IRQ event mode flag should be stored. If digital interrupts are in event mode, a nonzero value will be stored here. Otherwise, 0 will be stored here (meaning digital interrupts are in match mode).

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from the Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
if (rc != -1) {

   /*
    * If bit 3 is cleared in Status Register
    * Then
    *     Interrupt mode is event
    * Else
    *     Interrupt mode is match
    */

   if ((io_request.value & 0x08) == 0) {
      fprintf(stdout, "Digital I/O interrupts in event mode.\n");
   } else {
      fprintf(stdout, "Digital I/O interrupts in match mode.\n");
   }
}
```

int DIOIsChipPort1Output6420(int descriptor, int *port1_output_p);

Description:

Determine whether or not a board's digital port 1 is set to output.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Port1_output_p:	Address where port 1 output flag should be stored. If port 1 is set to output, a nonzero value will be stored here. Otherwise, 0 will be stored here (meaning port 1 is set to input).

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from the Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
if (rc != -1) {

   /*
    * If bit 2 is set in Status Register
    * Then
    *     Port 1 is output
    * Else
    *     Port 1 is input
    */
```

```
   if (io_request.value & 0x04) {
      fprintf(stdout, "Port 1 set to output.\n");
   } else {
      fprintf(stdout, "Port 1 set to input.\n");
   }
}
```

DIOIsChipStrobe6420

int DIOIsChipStrobe6420(int descriptor, int *strobe_occurred_p);

Description:

Determine whether or not data has been strobed into digital I/O port 0.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Strobe_occurred_p:	Address where strobe event flag should be stored. If data was strobed into port 0, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from the Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
if (rc != -1) {
```

```
   /*
    * If bit 7 is set in Status Register
    * Then
    *     Data was strobed into port 0
    * Else
    *     No data strobed
    */

   if (io_request.value & 0x80) {
     fprintf(stdout, "Data strobed into digital I/O port 0.\n");
   } else {
     fprintf(stdout, "No data strobed into digital I/O port 0.\n");
   }
}
```

DIOIsChipSystemClock6420

---

int DIOIsChipSystemClock6420(int descriptor, int *system_clock_p);

Description:

Determine whether or not the 8 MHz system clock is driving digital I/O sampling.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
System_clock_p:	Address where system clock flag should be stored. If the 8 MHz system clock is driving digital sampling, a nonzero value will be stored here. Otherwise, 0 will be stored here (meaning that the User Timer/Counter is driving sampling).

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from the Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;
```

```
/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
if (rc != -1) {

   /*
    * If bit 5 is cleared in Status Register
    * Then
    *     8 MHz system clock in use
    * Else
    *     User Timer/Counter in use
    */

   if ((io_request.value & 0x20) == 0) {
      fprintf(stdout, "8 MHz system clock driving digital I/O sampling.\n");
   } else {
      fprintf(stdout, "User Timer/Counter driving digital I/O sampling.\n");
   }
}
```

DIOLoadCompare6420

int DIOLoadCompare6420(int descriptor, u_int8_t Compare);

Description:

Load the Compare Register for a board's digital I/O port 0.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Compare: The bit pattern to be matched.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOLoadMask6420

int DIOLoadMask6420(int descriptor, u_int8_t Mask);

Description:

Load the Mask Register for a board's digital I/O port 0.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Mask: The bit mask to be loaded.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIORead6420

int DIORead6420(int descriptor, enum DM6420HR_DIO Port, u_int8_t *digital_data_p);

Description:

Read 8 bits of data from board's specified digital I/O port.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Port: Indicates which digital I/O port to read. Valid values are DM6420HR_DIO0 and DM6420HR_DIO1.

Digital_data_p: Address where data read should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

EINVAL Port is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from Digital I/O Port 1 Register at base I/O address + 26
 */

io_request.reg = r_DIO_PORT_1_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
```

DIOReadCompareRegister6420

---

int DIOReadCompareRegister6420(int descriptor, u_int8_t *register_value_p);

Description:

Read the contents of a board's digital I/O Compare Register.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Register_value_p: Address where register contents should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOReadStatus6420

int DIOReadStatus6420(int descriptor, u_int8_t *status_p);

Description:

Read the contents of a board's Digital IRQ/Strobe Status register. Other library functions make use of this routine to perform their work.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Status_p: Address where digital I/O status register contents should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from Digital IRQ/Strobe Status Register at base I/O address + 30
 */

io_request.reg = r_DIO_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the byte read.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
```

DIOSelectClock6420

int DIOSelectClock6420(int descriptor, enum DM6420HR_CLK_SEL Clock);

Description:

Select the sample clock source for a board's digital I/O chip.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Clock: Indicates the clock source. Valid values are DM6420HR_CLOCK_TC and DM6420HR_USER_TC.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL Clock is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO8 io_request;

/*
 * Write to Digital Mode Register at base I/O address + 30
 */

io_request.reg = r_DIO_MODE_6420;

/*
 * Only change bit 5 in Mode Register
 */

io_request.mask = 0xDF;

/*
 * Set digital I/O sample clock source to User Timer/Counter 1
 */

io_request.value = (DM6420HR_USER_TC << 5);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTB, &io_request);
```

DIOSelectIrqMode6420

int DIOSelectIrqMode6420(int descriptor, enum DM6420HR_DIO_IRQ IrqMode);

Description:

Set the advanced digital interrupt mode for a board's digital I/O chip port 0.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

IrqMode: Indicates the digital interrupt mode. Valid values are DM6420HR_DIO_IRQ_EVENT and DM6420HR_DIO_IRQ_MATCH .

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL IrqMode is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO8 io_request;

/*
 * Write to Digital Mode Register at base I/O address + 30
 */

io_request.reg = r_DIO_MODE_6420;

/*
 * Only change bit 3 in Mode Register
 */

io_request.mask = 0xF7;

/*
 * Set digital I/O interrupt mode to Event Mode
 */

io_request.value = (DM6420HR_DIO_IRQ_EVENT << 3);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTB, &io_request);
```

DIOSelectRegister6420

int DIOSelectRegister6420(int descriptor, enum DM6420HR_REG_SEL Select);

Description:

Set the mode of a board's digital I/O port 0 Direction/Mask/Compare Register located at base I/O address + 28. Other library functions make use of this routine to perform their work.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Select: Which mode to configure. Valid values are DM6420HR_REG_CLEAR, DM6420HR_REG_DIR, DM6420HR_REG_MASK, and DM6420HR_REG_CMP.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL Select is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO8 io_request;

/*
 * Write to Digital Mode Register at base I/O address + 30
 */

io_request.reg = r_DIO_MODE_6420;

/*
 * Only change bits 0 and 1 in Mode Register
 */

io_request.mask = 0xFC;

/*
 * Put digital I/O port 0 Direction/Mask/Compare Register into Clear Mode.  A subsequent read
 * from this register will clear the digital IRQ status flag whereas a subsequent write to this
 * register will clear the digital I/O chip.
 */

io_request.value = DM6420HR_REG_CLEAR;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTB, &io_request);
```

DIOSetPort0Direction6420

int DIOSetPort0Direction6420(int descriptor, u_int8_t Direction);

Description:

Set the digital I/O port 0 bit direction (input or output) on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Direction: Bit mask indicating bit direction for port 0 bits. A 1 in the mask means set that bit to output. A 0 in the mask means set that bit to input.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSetPort1Direction6420

---

int DIOSetPort1Direction6420(int descriptor, int Direction);

Description:

Set the digital I/O port 1 bit direction (input or output) on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Direction: Flag indicating bit direction. A value of 0 means set port 1 bits to input. A nonzero value means set port 1 bits to output.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO8 io_request;

/*
 * Write to Digital Mode Register at base I/O address + 30
 */

io_request.reg = r_DIO_MODE_6420;
```

```
/*
 * Only change bit 2 in Mode Register
 */

io_request.mask = 0xFB;

/*
 * Set digital I/O port 1 bit direction to output.  All port bits are set to this direction.
 */

io_request.value = (1 << 2);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTB, &io_request);
```

DIOWrite6420

---

int DIOWrite6420(int descriptor, enum DM6420HR_DIO Port, u_int8_t Data);

Description:

Write data to a board's selected digital I/O port.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Port:	The port to write to. Valid values are DM6420HR_DIO0 and DM6420HR_DIO1.
Data:	The data to write.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Port is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Write to digital I/O port 0
 */

io_request.reg = r_DIO_PORT_0_6420;
```

```
/*
 * Write the value 0x44 (68 decimal)
 */

io_request.value = 0x44;

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTB, &io_request);
```

DisableIRQ6420

---

int DisableIRQ6420(int descriptor, enum DM6420HR_INT IRQChannel);

Description:

Disable the specified interrupt circuit on a board.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
IRQChannel:	Interrupt circuit to disable. Valid values are DM6420HR_INT1 and DM6420HR_INT2.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	IRQChannel is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IE io_request;

/*
 * Target of operation is second interrupt circuit
 */

io_request.intr = DM6420HR_INT2;

/*
 * Disable the interrupt circuit
 */

io_request.action = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_IRQ_ENABLE, &io_request);
```

DoneTimer6420

int DoneTimer6420(int descriptor);

Description:

Initialize a board's user timer/counter counter 0 and counter 1 for high speed to ensure immediate load. This function puts these two counters into mode 2.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure. Please see the descriptions of the ClockDivisor6420(), ClockMode6420(), and SelectTimerCounter6420() functions or the ioctl(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

EnableIRQ6420

int EnableIRQ6420(int descriptor, enum DM6420HR_INT IRQChannel);

Description:

Enable the specified interrupt circuit on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

IRQChannel: Interrupt circuit to enable. Valid values are DM6420HR_INT1 and DM6420HR_INT2.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL IRQChannel is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IE io_request;

/*
 * Target of operation is first interrupt circuit
 */

io_request.intr = DM6420HR_INT1;

/*
 * Enable the interrupt circuit
 */

io_request.action = 1;

rc = ioctl(descriptor, DM6420HR_IOCTL_IRQ_ENABLE, &io_request);
```

EnableTables6420

---

int EnableTables6420(int descriptor, int Enable_AD_Table, int Enable_Digital_Table);

Description:

Enable or disable the A/D and digital tables in the channel/gain scan memory on a board.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Enable_AD_Table:	Flag to indicate whether the A/D table should be enabled. A value of 0 means disable the A/D table. A nonzero value means enable the A/D table.
Enable_Digital_Table:	Flag to indicate whether the digital table should be enabled. A value of 0 means disable the digital table. A nonzero value means enable the digital table.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for write access.
EOPNOTSUPP	The digital table is to be enabled but the A/D table is to be disabled.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Control Register at base I/O address + 2
 */

io_request.reg = r_CONTROL_6420;

/*
 * Only change bits 2 and 3 in Control Register
 */

io_request.mask = 0xFFF3;

/*
 * Enable just the A/D table
 *
 * To disable both tables, set io_request.value to 0.
 * To enable both tables, set io_request.value to (0x3 << 2)
 */

io_request.value = (1 << 2);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

GetAutoincData6420

```
int GetAutoincData6420(
   int descriptor,
   enum DM6420HR_STR_Regs from_register,
   enum DM6420HR_STR_TYPE type,
   void *buffer_p,
   size_t element_num
);
```

Description:

Initiate a streaming read from a board. Once can specify the board register to read from, what size data is to be transferred, and how many data elements to transfer.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
From_register:	Register from which the data should be read. Valid values are rSTR_AD_6420 and rSTR_DIN_FIFO_6420.

Type:	Type/size of element to be transferred. Valid values are DM6420HR_STR_TYPE_BYTE and DM6420HR_STR_TYPE_WORD.
Buffer_p:	Address of buffer in which to place the data read.
Element_num:	How many data elements of type “type” should be read.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for read access.
EFAULT	buffer_p is not a valid user address.
EINVAL	from_register is not valid.
EINVAL	type is not valid.
EOPNOTSUPP	from_register is rSTR_AD_6420 and type is DM6420HR_STR_TYPE_BYTE.
EOPNOTSUPP	from_register is rSTR_DIN_FIFO_6420 and type is DM6420HR_STR_TYPE_WORD.

IOCTL Interface:

```
int rc;
struct DM6420HR_GID io_request;
int16_t ad_buffer[600];
uint8_t digital_buffer[512];

/*
 * Read from A/D input FIFO
 */

io_request.port = rSTR_AD_6420;

/*
 * Transfer 16-bit values
 */

io_request.type = DM6420HR_STR_TYPE_WORD;

/*
 * Transfer data into ad_buffer[] array
 */

io_request.buf = (void *) &(ad_buffer[0]);
```

```
/*
 * Transfer 600 16-bit values
 */

io_request.times = 600;

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_GETINC, &io_request);

/*
 * Read from digital port 0 input FIFO
 */

io_request.port = rSTR_DIN_FIFO_6420;

/*
 * Transfer 8-bit values
 */

io_request.type = DM6420HR_STR_TYPE_BYTE;

/*
 * Transfer data into digital_buffer[] array
 */

io_request.buf = (void *) &(digital_buffer[0]);

/*
 * Transfer 512 8-bit values
 */

io_request.times = 512;

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_GETINC, &io_request);
```

GetDmaData6420

```
int GetDmaData6420(
    int descriptor,
    void *dma_buffer_p,
    enum DM6420HR_DMA DMAChannel,
    size_t length,
    size_t offset,
    size_t *bytes_transferred_p
);
```

Description:

Copy data from specified DMA circuit's DMA buffer into user buffer.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Dma_buffer_p:	Address of user buffer.
DMAChannel:	DMA circuit to operate on. Valid values are DM6420HR_DMA1 and DM6420HR_DMA2.
Length:	Number of bytes to transfer.
Offset:	Offset in bytes from beginning of driver's DMA buffer where read should begin.
Bytes_transferred_p:	Address where actual number of bytes transferred will be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for read access.
EFAULT	dma_buffer_p is not a valid user address.
EFAULT	dma_buffer_p is not large enough to hold the data.
EINVAL	DMAChannel is not valid.
EINVAL	No DMA channel was ever allocated to DMAChannel.
EINVAL	No DMA buffer was ever allocated to DMAChannel.
EINVAL	(length + offset) lies beyond the end of the driver's DMA buffer.

IOCTL Interface:

```
int rc;
struct DM6420HR_GDD io_request;
int16_t user_buffer[8192];

/*
 * Read from DMA buffer on second DMA circuit
 */

io_request.dma = DM6420HR_DMA2;

/*
 * Transfer data into user_buffer[] array
 */

io_request.buf = (void *) &(user_buffer[0]);
```

```
/*
 * Transfer 8192 16-bit values (16384 8-bit values)
 */

io_request.length = 16384;

/*
 * Read from beginning of driver's DMA buffer
 */

io_request.offset = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_GETDATA, &io_request);
```

GetIRQCounter6420

```
int GetIRQCounter6420(
   int descriptor,
   enum DM6420HR_INT IRQChannel,
   unsigned long *counter_value_p
);
```

Description:

Get the number of interrupts that have occurred on a board's specified interrupt circuit.

Data returned from this function is interpreted differently depending upon usage of the force argument on the insmod command. If you specify a nonzero value for the force argument, the value returned represents the number of interrupts that occurred since the driver module was loaded. If you specify a zero value for the force argument or do not use force at all, the value returned represents the number of interrupts that occurred since the device file was opened.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
IRQChannel:	Interrupt circuit to read counter value from. Valid values are DM6420HR_INT1 and DM6420HR_INT2.
Counter_value_p:	Address where counter value should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for read access.
EINVAL	IRQChannel is not valid.
EINVAL	No IRQ was ever allocated to IRQChannel.

IOCTL Interface:

```
int rc;
struct DM6420HR_GIC io_request;

/*
 * Target of operation is first interrupt circuit
 */

io_request.intr = DM6420HR_INT1;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable counter contains the interrupt count.
 */

io_request.counter = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_GET_IRQ_COUNTER, &io_request);
if (rc != -1) {
   fprintf(stdout, “Interrupt count: %ld.\n”, io_request.counter);
}
```

InitBoard6420

---

int InitBoard6420(int descriptor);

Description:

Initialize a board. This will 1) clear the board, 2) clear the A/D DMA done flag, 3) clear the channel/gain table, and 4) clear the A/D input FIFO.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

This function makes use of several ioctl() requests.

InstallCallbackIRQHandler6420

```
int InstallCallbackIRQHandler6420(
   int descriptor,
   void (*callback)(void),
   enum DM6420HR_INT IRQChannel
);
```

Description:

Install a function which will be called whenever an interrupt occurs and the driver sends a signal to the process to indicate that the interrupt happened.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Callback:	Address of callback function.
IRQChannel:	Board interrupt circuit that signal should be attached to. Valid values are DM6420HR_INT1 and DM6420HR_INT2.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	callback is NULL.
	EINVAL	IRQChannel is not valid.
	ENOMEM	Library callback descriptor memory could not be allocated.
	Please see the sigaction(2) man page, the sigprocmask(2) man page, or the ioctl(2) man page for information on other possible values errno may have in this case.	

IOCTL Interface:

This function makes use of several ioctl() requests.

InstallDMA6420

```
int InstallDMA6420(int descriptor, enum DM6420HR_DMA DMAChannel);
```

Description:

Configure the specified DMA circuit on a board to be able to perform DMA.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
DMAChannel:	DMA circuit to configure. Valid values are DM6420HR_DMA1 and DM6420HR_DMA2.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	DMAChannel is not valid.
	EINVAL	No DMA channel was ever allocated to DMAChannel.
	ENOMEM	DMA buffer memory allocation failed.

IOCTL Interface:

```
int rc;
struct DM6420HR_DI io_request;

/*
 * Target of operation is second DMA circuit
 */

io_request.dma = DM6420HR_DMA2

/*
 * Enable DMA on the circuit
 */

io_request.action = 1;

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_INSTALL, &io_request);
```

IsAboutTrigger6420

---

int IsAboutTrigger6420(int descriptor, int *ad_about_trigger_p);

Description:

Determine whether or not a board's A/D about trigger has occurred.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Ad_about_trigger_p:	Address where A/D about trigger flag should be stored. If A/D about trigger has occurred, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 8 is set in Status Register, then about trigger has occurred
    *
    * If bit 8 is cleared, then about trigger has not occurred
    */

   if (io_request.value & 0x0100) {
     fprintf(stdout, “About trigger occurred.\n”);
   }
}
```

int IsADConverting6420(int descriptor, int *ad_converting_p);

Description:

Determine whether or not a board's A/D converter is converting.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Ad_converting_p: Address where A/D converting flag should be stored. If A/D conversion is occurring, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 3 is cleared in Status Register, then A/D converter is converting
    *
    * If bit 3 is set, then A/D converter is not converting
    */
```

```
   if ((io_request.value & 0x0008) == 0) {
      fprintf(stdout, "A/D converter is still converting.\n");
   }
}
```

IsADDMADone6420

int IsADDMADone6420(int descriptor, int *ad_dma_done_p);

Description:

Determine whether or not a board's A/D DMA transfer is complete.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Ad_dma_done_p	Address where A/D DMA done flag should be stored. If A/D DMA is finished, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {
```

```
   /*
    * If bit 4 is set in Status Register, then A/D DMA is done
    *
    * If bit 4 is cleared, then A/D DMA is not done
    */

   if (io_request.value & 0x0010) {
     fprintf(stdout, "A/D DMA completed.\n");
   }
}
```

IsADFIFOEmpty6420

---

int IsADFIFOEmpty6420(int descriptor, int *ad_fifo_empty_p);

Description:

Determine whether or not a board's A/D FIFO is empty.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Ad_fifo_empty_p:	Address where FIFO empty flag should be stored. If the A/D FIFO is empty, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;
```

```
rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 0 is cleared in Status Register, then A/D FIFO is empty
    *
    * If bit 0 is set, then A/D FIFO is not empty
    */

   if ((io_request.value & 0x0001) == 0) {
      fprintf(stdout, "A/D FIFO empty.\n");
   }
}
```

IsADFIFOFull6420

int IsADFIFOFull6420(int descriptor, int *ad_fifo_full_p);

Description:

Determine whether or not a board's A/D FIFO is full.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Ad_fifo_empty_p: Address where FIFO full flag should be stored. If the A/D FIFO is full, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;
```

```
/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 1 is set in Status Register, then A/D FIFO is not full
    *
    * If bit 1 is cleared, then A/D FIFO is full
    */

   if (io_request.value & 0x0002) {
     fprintf(stdout, "A/D FIFO not full.\n");
   }
}
```

IsADHalted6420

---

int IsADHalted6420(int descriptor, int *ad_halted_p);

Description:

Determine whether or not a board's A/D conversion has been stopped because the sample buffer is
full.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Ad_halted_p: Address where A/D halted flag should be stored. If A/D conversion
has been stopped, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;
```

```
/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 2 is cleared in Status Register, then A/D conversion has not been stopped
    *
    * If bit 2 is set, then A/D conversion has been stopped
    */

   if ((io_request.value & 0x0004) == 0) {
      fprintf(stdout, "A/D conversion not halted due to sample buffer being full.\n");
   }
}
```

IsBurstClockOn6420

```
int IsBurstClockOn6420(int descriptor, int *ad_burst_clock_on_p);
```

Description:

Determine whether or not a board's A/D burst clock gate is on.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Ad_burst_clock_on_p: Address where A/D burst clock flag should be stored. If A/D burst clock gate is on, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 6 is set in Status Register, then burst gate is on
    *
    * If bit 6 is cleared, then burst gate is off
    */

   if (io_request.value & 0x0040) {
     fprintf(stdout, “Burst gate is on.\n”);
   }
}
```

IsDigitalIRQ6420

int IsDigitalIRQ6420(int descriptor, int *digital_interrupt_p);

Description:

Determine whether or not an advanced digital mode interrupt has occurred on a board.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Digital_interrupt_p:	Address where digital interrupt flag should be stored. If an advanced digital interrupt has occurred, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 9 is cleared in Status Register, then no digital interrupt occurred
    *
    * If bit 9 is set, then digital interrupt occurred
    */

   if ((io_request.value & 0x0200) == 0) {
      fprintf(stdout, "Digital interrupt has not occurred.\n");
   }
}
```

IsDINFIFOEmpty6420

int IsDINFIFOEmpty6420(int descriptor, int *digital_fifo_empty_p);

Description:

Determine whether or not a board's digital input FIFO is empty.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Digital_fifo_empty_p:	Address where digital FIFO empty flag should be stored. If the digital FIFO is empty, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 10 is set in Status Register, then digital input FIFO is not empty
    *
    * If bit 10 is cleared, then digital input FIFO is empty
    */

   if (io_request.value & 0x0400) {
      fprintf(stdout, “Digital input FIFO is not empty.\n”);
   }
}
```

int IsDINFIFOFull6420(int descriptor, int *digital_fifo_full_p);

Description:

Determine whether or not a board's digital input FIFO is full.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Digital_fifo_full_p:	Address where digital FIFO full flag should be stored. If the digital FIFO is full, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 12 is cleared in Status Register, then digital input FIFO is full
    *
    * If bit 12 is set, then digital input FIFO is not full
    */
```

```
    if ((io_request.value & 0x1000) == 0) {
      fprintf(stdout, "Digital input FIFO is full.\n");
    }
}
```

IsDINFIFOHalf6420

---

int IsDINFIFOHalf6420(int descriptor, int *digital_fifo_half_full_p);

Description:

Determine whether or not a board's digital input FIFO is half full.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Digital_fifo_half_full_p:	Address where digital FIFO half full flag should be stored. If the digital FIFO is half full, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {
```

```
   /*
    * If bit 11 is set in Status Register, then digital input FIFO is not half full
    *
    * If bit 11 is cleared, then digital input FIFO is half full
    */

   if (io_request.value & 0x0800) {
     fprintf(stdout, "Digital input FIFO is not half full.\n");
   }
}
```

IsFirstADDMADone6420

int IsFirstADDMADone6420(int descriptor, int *ad_first_dma_done_p);

Description:

Determine whether or not a board's A/D first DMA transfer is complete when in dual channel DMA mode.

NOTE: The description of the ad_first_dma_done_p parameter below is based upon the hardware manual's description of the First DMA Flag in the Status Register. However, the manual erroneously provides an inverted interpretation of this flag. Since this function examines the First DMA Flag, the value stored in *ad_first_dma_done_p is inverted also. In reality, the value stored in this address will be zero if A/D first DMA transfer is finished and nonzero otherwise.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Ad_first_dma_done_p:	Address where A/D first DMA done flag should be stored. If A/D first DMA is finished, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */
```

```
io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 5 is cleared in Status Register, then first DMA is done
    *
    * If bit 5 is set, then first DMA is not done
    */

   if ((io_request.value & 0x0020) == 0) {
     fprintf(stdout, "First DMA done.\n");
   }
}
```

IsPacerClockOn6420

```
int IsPacerClockOn6420(int descriptor, int *ad_pacer_clock_on_p);
```

Description:

Determine whether or not a board's A/D pacer clock gate is on.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Ad_pacer_clock_on_p:	Address where A/D pacer clock flag should be stored. If A/D pacer clock gate is on, a nonzero value will be stored here. Otherwise, 0 will be stored here.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;
```

```
/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {

   /*
    * If bit 7 is set in Status Register, then A/D pacer clock gate is on
    *
    * If bit 7 is cleared, then pacer clock gate is off
    */

   if (io_request.value & 0x0080) {
      fprintf(stdout, "A/D pacer clock gate is on.\n");
   }
}
```

LoadADSampleCounter6420

int LoadADSampleCounter6420(int descriptor, u_int16_t NumOfSamples);

Description:

Load a board's analog to digital sample counter.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
NumOfSamples:	Number of samples to take.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

This function makes use of several ioctl() requests.

LoadADTable6420

int LoadADTable6420(int descriptor, u_int16_t ADEntries, ADTableRow *ADTable_p);

Description:

Load a board's A/D table with the given number of entries.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

ADEntries: Number of entries in A/D table.

ADTable_p: Address of memory containing A/D table to send to board.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	ADEntries is 0.
EINVAL	ADEntries is greater than 1024.
EINVAL	An entry in the table pointed to by ADTable_p has an invalid Channel, Gain, ADRange, or Se_Diff member variable value.

IOCTL Interface:

This function makes use of several ioctl() requests.

LoadControlRegister6420

int LoadControlRegister6420(int descriptor, u_int16_t value);

Description:

Load a value into a board's Control Register.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Value: Data to write into Control Register.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Write to Control Register at base I/O address + 2
 */

io_request.reg = r_CONTROL_6420;

/*
 * Set first DMA circuit to use DMA channel 5
 */

io_request.value = (1 << 12);

/*
 * Set second DMA circuit to use DMA channel 7
 */

io_request.value |= (3 << 14);

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTW, &io_request);
```

LoadDAC6420

int LoadDAC6420(int descriptor, enum DM6420HR_DAC dac, u_int16_t Data);

Description:

Load a 12-bit binary value into one of the digital to analog converters on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Dac: D/A converter to load. Valid values are DM6420HR_DAC1 and DM6420HR_DAC2.

Data: Value to write to converter.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL dac is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Write to DAC1 Output Register at base I/O address + 12
 */

io_request.reg = r_DAC1_6420;

/*
 * Write 1984 (0x07C0 in two's complement) to D/A converter
 */

io_request.value = 0x07C0;

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTW, &io_request);
```

LoadDigitalTable6420

---

int LoadDigitalTable6420(int descriptor, u_int16_t entries, u_int8_t *table_p);

Description:

Load a board's digital table with the given number of entries.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Entries: Number of entries in digital table.

Table_p: Address of memory containing digital table to send to board. This memory is an array of unsigned bytes.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL entries is 0.

EINVAL entries is greater than 1024.

IOCTL Interface:

This function makes use of several ioctl() requests.

LoadDINConfigRegister6420

---

int LoadDINConfigRegister6420(int descriptor, u_int16_t value);

Description:

Load a 16-bit value into a board's Digital Input FIFO Configuration Register.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Value: Value to write into register.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Write to Digital Input FIFO Configuration Register at base I/O address + 10
 */

io_request.reg = r_DIN_CONFIG_6420;

/*
 * Set digital input FIFO clock source to output of user timer/counter 1
 */

io_request.value = 0x0001;
```

```
/*
 * Enable digital input FIFO clock
 */

io_request.value |= (1 << 3);

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTW, &io_request);
```

LoadIRQRegister6420

---

int LoadIRQRegister6420(int descriptor, u_int16_t value);

Description:

Load a 16-bit value into a board's Interrupt Register.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Value: Value to load into Interrupt Register.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Write to Interrupt Register at base I/O address + 8
 */

io_request.reg = r_IRQ_6420

/*
 * Set source of interrupts on first circuit to be write to digital input FIFO
 */

io_request.value = 0x000F;

/*
 * Use IRQ5 for first interrupt circuit
 */

io_request.value |= (1 << 6);
```

```
rc = ioctl(descriptor, DM6420HR_IOCTL_OUTW, &io_request);
```

LoadTriggerRegister6420

---

int LoadTriggerRegister6420(int descriptor, u_int16_t value);

Description:

Load a 16-bit value into a board's Trigger Mode Register.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Value: Value to write into Trigger Mode Register.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Pacer Clock controls A/D conversion
 */

io_request.value = 0x0001;

/*
 * Start Pacer Clock when digital interrupt received
 */

io_request.value |= (1 << 3);

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTW, &io_request);
```

OpenBoard6420

int OpenBoard6420(int DeviceNumber);

Description:

Open a DM6420 device file.

Parameters:

DeviceNumber: Minor number of board device file.

Return Value:

>=0: Success. The integer returned is the file descriptor from open() system call.

-1: Failure. Please see the open(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

None.

ReadADData6420

int ReadADData6420(int descriptor, int16_t *ad_data_p);

Description:

Read the 12-bit A/D sample from a board's analog to digital FIFO.

NOTE: This function discards marker data from the A/D FIFO data. Calling ReadADData6420() and ReadADDataMarker6420() separately with the intention of getting data from a single sample will likely result in data being returned from two distinct samples.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Ad_data_p: Address where A/D sample data read should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
int16_t ad_data;
struct DM6420HR_IO16 io_request;

/*
 * Read from A/D Data FIFO Register at base I/O address + 4
 */

io_request.reg = r_AD_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the A/D data.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {
   ad_data = ((int16_t) io_request.value >> 3);
}
```

ReadADDataMarker6420

int ReadADDataMarker6420(int descriptor, u_int8_t *data_marker_p);

Description:

Read the 3-bit data marker from a board's analog to digital FIFO .

NOTE: This function discards A/D sample data from the A/D FIFO data. Calling ReadADDataMarker6420() and ReadADData6420() separately with the intention of getting data from a single sample will likely result in data being returned from two distinct samples.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Data_marker_p: Address where data marker should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;
u_int8_t data_marker;

/*
 * Read from A/D Data FIFO Register at base I/O address + 4
 */

io_request.reg = r_AD_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the A/D data.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {
   data_marker = (u_int8_t) (io_request.value & 0x0007);
}
```

ReadADDataWithMarker6420

---

int ReadADDataWithMarker6420(int descriptor, int16_t *ad_fifo_p);

Description:

Read the entire 16-bit contents of a board's analog to digital FIFO . The most significant bit is the
sign bit. The next 12 bits are the converted A/D data. The least significant 3 bits are the data marker.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Ad_fifo_p: Address where A/D FIFO contents should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
int16_t ad_data;
struct DM6420HR_IO16 io_request;
u_int8_t data_marker;

/*
 * Read from A/D Data FIFO Register at base I/O address + 4
 */

io_request.reg = r_AD_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the A/D data.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
if (rc != -1) {
   ad_data = ((int16_t) io_request.value >> 3);
   data_marker = (u_int8_t) (io_request.value & 0x0007);
}
```

ReadChannelGainDataStore6420

int ReadChannelGainDataStore6420(int descriptor, u_int16_t *cgds_data_p);

Description:

Read a board's channel/gain data word. This function assumes the caller knows whether or not the channel/gain data store has been enabled since there is no way to query the board for this status.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Cgds_data_p: Address where data read should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from A/D Data FIFO Register at base I/O address + 4
 */

io_request.reg = r_AD_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the channel/gain data.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
```

ReadDINFIFO6420

---

int ReadDINFIFO6420(int descriptor, u_int8_t *digital_data_p);

Description:

Read 8 bits of data from the port 0 digital input FIFO.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Digital_data_p:	Address where data read should be stored.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO8 io_request;

/*
 * Read from Digital Input FIFO Register at base I/O address + 10
 */

io_request.reg = r_DIN_FIFO_6420;
```

```
/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the digital data.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INB, &io_request);
```

ReadStatus6420

int ReadStatus6420(int descriptor, u_int16_t *status_p);

Description:

Read a board's Status Register. Other library functions make use of this routine to perform their work.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Status_p: Address where status should be stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Status Register at base I/O address + 2
 */

io_request.reg = r_STATUS_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the register contents.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
```

```
int ReadTimerCounter6420(
   int descriptor,
   enum DM6420HR_CLK_SEL Timer,
   enum DM6420HR_CLK Clock,
   u_int16_t *counter_value_p
);
```

Description:

Read the 16 bit contents of the desired timer/counter. The read is done as two 8-bit reads: least significant byte then most significant byte.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Timer:	Indicates which timer/counter to use. Valid values are DM6420HR_CLOCK_TC and DM6420HR_USER_TC.
Clock:	Indicates which counter to use. Valid values are DM6420HR_CLK0, DM6420HR_CLK1, and DM6420HR_CLK2.
Counter_value_p:	Address where timer contents should be stored.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for read access.
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	Timer is not valid.
	EINVAL	Clock is not valid.

IOCTL Interface:

This function makes use of several ioctl() requests.

RemoveIRQHandler6420

int RemoveIRQHandler6420(int descriptor, enum DM6420HR_INT IRQChannel);

Description:

Uninstall the function which was previously registered as an interrupt callback by InstallCallbackIRQHandler6420().

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
IRQChannel:	Board interrupt circuit that signal should be detached from. Valid values are DM6420HR_INT1 and DM6420HR_INT2.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	IRQChannel is not valid.
EINVAL	No IRQ was ever allocated to IRQChannel.

IOCTL Interface:

This function makes use of a single ioctl() call. However in addition to this ioctl() call, several other system calls are used to reset process signal handling state; a description is beyond the scope of this document.

ResetChannelGainTable6420

int ResetChannelGainTable6420(int descriptor);

Description:

Reset a board's channel/gain table starting point to the beginning of the table.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

IOCTL Interface:

int rc;

rc = ioctl(descriptor, DM6420HR_IOCTL_CLEAR, DM6420_CL_RESET_GAIN);

SelectTimerCounter6420

---

int SelectTimerCounter6420(int descriptor, enum DM6420HR_CLK_SEL Select);

Description:

Select which timer/counter on a board will be accessed when a subsequent operation is performed on the registers located at base I/O address + 16 through base I/O address + 22. Other library functions make use of this routine to perform their work.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Select: Indicate which timer/counter will be accessed. Valid values are DM6420HR_CLOCK_TC and DM6420HR_USER_TC.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL Select is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Control Register at base I/O address + 2
 */

io_request.reg = r_CONTROL_6420;

/*
 * Only change bits 5 and 6 in Control Register
 */
```

```
io_request.mask = 0xFF9F;

/*
 * Select User Timer/Counter
 */

io_request.value = (1 << 5);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetBurstClock6420

int SetBurstClock6420(int descriptor, double BurstRate, double *actual_p);

Description:

Set a board's burst clock rate.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
BurstRate:	Burst clock rate desired.
Actual_p:	Address where the actual programmed frequency should be stored. If this function fails, the actual frequency is not updated.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.

IOCTL Interface:

This function makes use of several ioctl() requests.

SetBurstTrigger6420

int SetBurstTrigger6420(int descriptor, enum DM6420HR_BURST_TRIG Burst_Trigger);

Description:

Select a board's burst mode trigger.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Burst_Trigger:	What triggers a burst. Valid values are DM6420HR_BURST_TRIG_SOFTWARE, DM6420HR_BURST_TRIG_PACER, DM6420HR_BURST_TRIG_EXTERNAL, and DM6420HR_BURST_TRIG_DIGITAL.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Burst_Trigger is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bits 10 and 11 in Trigger Mode Register
 */

io_request.mask = 0xF3FF;

/*
 * Bursts triggered by external trigger
 */

io_request.value = (1 << 11);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

```
int SetChannelGain6420(
   int descriptor,
   enum DM6420HR_AIN Channel,
   enum DM6420HR_GAIN Gain,
   enum DM6420HR_RANGE Range,
   enum DM6420HR_SE Se_Diff
);
```

Description:

Load a board's channel/gain latch.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Channel:	A/D channel. Valid values are DM6420HR_AIN1, DM6420HR_AIN2, DM6420HR_AIN3, DM6420HR_AIN4, DM6420HR_AIN5, DM6420HR_AIN6, DM6420HR_AIN7, DM6420HR_AIN8, DM6420HR_AIN9, DM6420HR_AIN10, DM6420HR_AIN11, DM6420HR_AIN12, DM6420HR_AIN13, DM6420HR_AIN14, DM6420HR_AIN15, and DM6420HR_AIN16.
Gain:	A/D gain. Valid values are DM6420HR_GAINx1, DM6420HR_GAINx2, DM6420HR_GAINx4, and DM6420HR_GAINx8.
Range:	A/D input voltage range and polarity. Valid values are DM6420HR_RANGE_SIGNED5, DM6420HR_RANGE_SIGNED10, and DM6420HR_RANGE_UNSIGNED10.
Se_Diff:	Select single-ended or differential mode. Valid values are DM6420HR_SE_SE and DM6420HR_SE_DIFF.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	Channel is not valid.
	EINVAL	Gain is not valid.
	EINVAL	Range is not valid.
	EINVAL	Se_Diff is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Before loading channel/gain latch, bits 1 and 0 in Control Register at base
 * I/O address + 2 must be set to zeroes.  This is not shown here.
 */

/*
 * Write to Load Channel/Gain Latch Register at base I/O address + 4
 */

io_request.reg = r_CHANNEL_GAIN_6420;

/*
 * Target of operation is analog input channel 7
 */

io_request.value = 0x0006;

/*
 * Set x2 gain
 */

io_request.value |= (1 << 4);

/*
 * Set differential mode
 */

io_request.value |= (1 << 9);

rc = ioctl(descriptor, DM6420HR_IOCTL_OUTW, &io_request);
```

SetConversionSelect6420

---

int SetConversionSelect6420(int descriptor, enum DM6420HR_CONV Select);

Description:

Configure how a board's A/D conversion is done.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Select: Indicates how A/D conversions are controlled. Valid values are DM6420HR_CONV_SOFT_TRIGGER, DM6420HR_CONV_PACER_CLOCK, DM6420HR_CONV_BURST_CLOCK, and DM6420HR_CONV_DIGITAL_INT.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL Select is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bits 0 and 1 in Trigger Mode Register
 */

io_request.mask = 0xFFFC;

/*
 * Pacer clock controls A/D conversion
 */

io_request.value = 0x0001;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetIRQ0Source6420

int SetIRQ0Source6420(int descriptor, enum DM6420HR_INTSRC IRQSource);

Description:

Select the interrupt source of the first interrupt circuit on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

IRQSource: Source of interrupt on first circuit. Valid values are
IRQS_AD_SAMPLE_CNT_6420,
IRQS_AD_START_CONVERT_6420,
IRQS_AD_END_CONVERT_6420,
IRQS_AD_WRITE_FIFO_6420,
IRQS_AD_FIFO_HALF_6420,
IRQS_AD_DMA_DONE_6420,
IRQS_RESET_GAIN_TABLE_6420,
IRQS_PAUSE_GAIN_TABLE_6420,
IRQS_EXT_PACER_CLOCK_6420,
IRQS_EXT_TRIGGER_6420,
IRQS_DIGITAL_6420,
IRQS_TC_COUNTER0_6420,
IRQS_TC_COUNTER0_INVERTED_6420,
IRQS_TC_COUNTER1_6420,
IRQS_DIO_FIFO_HALF_6420, and
IRQS_DIO_WRITE_FIFO_6420.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL IRQSource is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Interrupt Register at base I/O address + 8
 */

io_request.reg = r_IRQ_6420;

/*
 * Only change bits 0 through 4 in register
 */

io_request.mask = 0xFFE0;

/*
 * Set interrupt source to be A/D FIFO half full
 */

io_request.value = 0x0004;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

int SetIRQ1Source6420(int descriptor, enum DM6420HR_INTSRC IRQSource);

Description:

Select the interrupt source of the second interrupt circuit on a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

IRQSource: Source of interrupt on first circuit. Valid values are
IRQS_AD_SAMPLE_CNT_6420,
IRQS_AD_START_CONVERT_6420,
IRQS_AD_END_CONVERT_6420,
IRQS_AD_WRITE_FIFO_6420,
IRQS_AD_FIFO_HALF_6420,
IRQS_AD_DMA_DONE_6420,
IRQS_RESET_GAIN_TABLE_6420,
IRQS_PAUSE_GAIN_TABLE_6420,
IRQS_EXT_PACER_CLOCK_6420,
IRQS_EXT_TRIGGER_6420,
IRQS_DIGITAL_6420,
IRQS_TC_COUNTER0_6420,
IRQS_TC_COUNTER0_INVERTED_6420,
IRQS_TC_COUNTER1_6420,
IRQS_DIO_FIFO_HALF_6420, and
IRQS_DIO_WRITE_FIFO_6420.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for write access.

EINVAL IRQSource is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Interrupt Register at base I/O address + 8
 */

io_request.reg = r_IRQ_6420;

/*
 * Only change bits 8 through 12 in register
 */
```

```
io_request.mask = 0xE0FF;

/*
 * Set interrupt source to be User Timer/Counter 0 countdown to zero
 */

io_request.value = (0xB << 8);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetPacerClock6420

---

int SetPacerClock6420(int descriptor, double clock, double *actual_p);

Description:

Set a board's pacer clock rate. This function decides whether to use a 16-bit or 32-bit clock depending upon the clock rate.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Clock:	Clock rate desired.
Actual_p:	Address where the actual programmed frequency should be stored. If this function fails, the actual frequency is not updated.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	clock is greater than 8 MHz.
EINVAL	clock is 0.0.
EINVAL	clock is less than 0.0.

IOCTL Interface:

This function makes use of several ioctl() requests.

int SetPacerClockSource6420(int descriptor, enum DM6420HR_PACER_CLK Source);

Description:

Select the source of a board's pacer clock.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Source:	Pacer clock source. Valid values are DM6420HR_PACER_CLK_INTERNAL and DM6420HR_PACER_CLK_EXTERNAL.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	Source is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bit 9 in register
 */

io_request.mask = 0xFDFF;

/*
 * Set internal pacer clock
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

int SetPauseEnable6420(int descriptor, int Enable);

Description:

Enable or disable a board's A/D table pause bit.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Enable:	Flag to indicate whether the A/D table pause bit should set. A value of 0 means enable the pause bit. A nonzero value means disable the pause bit.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Control Register at base I/O address + 2
 */

io_request.reg = r_CONTROL_6420;

/*
 * Only change bit 8 in register
 */
io_request.mask = 0xFEFF;

/*
 * Activate pause bit in A/D table channel/gain scan memory
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

int SetSampleCounterStop6420(int descriptor, int Disable);

Description:

Enable or disable a board's A/D Sample Counter Stop.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Disable:	Flag to indicate whether the A/D sample counter should stop the pacer clock. A value of 0 means enable sample counter stop. A nonzero value means disable sample counter stop.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Control Register at base I/O address + 2
 */

io_request.reg = r_CONTROL_6420;

/*
 * Only change bit 7 in register
 */

io_request.mask = 0xFF7F;

/*
 * Disable A/D sample counter stop
 */

io_request.value = (1 << 7);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

int SetStartTrigger6420(int descriptor, enum DM6420HR_START_TRIG Start_Trigger);

Description:

Configure how a board's pacer clock is started during A/D conversion.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Start_Trigger:	What starts the pacer clock. Valid values are DM6420HR_START_TRIG_SOFTWARE, DM6420HR_START_TRIG_EXTERNAL, DM6420HR_START_TRIG_DIGITAL_INT, DM6420HR_START_TRIG_USER_TC1, and DM6420HR_START_TRIG_GATE.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Start_Trigger is not valid.
EOPNOTSUPP	Start_Trigger is one of the three reserved bit patterns as defined in the hardware manual.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bits 2 through 4 in register
 */

io_request.mask = 0xFFE3;

/*
 * Start pacer clock when User Timer/Counter 1 counts down to zero
 */

io_request.value = (0x3 << 2);
```

```
rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetStopTrigger6420

```
int SetStopTrigger6420(int descriptor, enum DM6420HR_STOP_TRIG Stop_Trigger);
```

Description:

Configure how a board's pacer clock is stopped during A/D conversion.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Stop_Trigger:	What stops the pacer clock. Valid values are DM6420HR_STOP_TRIG_SOFTWARE, DM6420HR_STOP_TRIG_EXTERNAL, DM6420HR_STOP_TRIG_DIGITAL_INT, DM6420HR_STOP_TRIG_SAMPLE_CNT, DM6420HR_STOP_TRIG_ABOUT_SOFTWARE, DM6420HR_STOP_TRIG_ABOUT_EXTERNAL, DM6420HR_STOP_TRIG_ABOUT_DIGITAL, and DM6420HR_STOP_TRIG_ABOUT_USER_TC1.

Return Value:

0:	Success.	
-1:	Failure with errno set as follows:	
	EACCES	descriptor refers to a file that is open but not for write access.
	EINVAL	Stop_Trigger is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bits 5 through 7 in register
 */

io_request.mask = 0xFF1F;
```

```
/*
 * Stop pacer clock by software trigger
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetTriggerPolarity6420

---

int SetTriggerPolarity6420(int descriptor, enum DM6420HR_POLAR Polarity);

Description:

Select which edge of an external pacer clock triggers a board's burst mode.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Polarity:	Which external pacer clock edge triggers burst mode. Valid values are DM6420HR_POLAR_POSITIVE and DM6420HR_POLAR_NEGATIVE.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Polarity is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bit 12 in register
 */

io_request.mask = 0xEFFF;
```

```
/*
 * Set external trigger to occur on negative edge
 */

io_request.value = (1 << 12);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetTriggerRepeat6420

int SetTriggerRepeat6420(int descriptor, enum DM6420HR_REPEAT Repeat);

Description:

Select whether or not a trigger initiates multiple A/D conversion cycles.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Repeat:	Indicates whether or not A/D conversion cycles repeat. Valid values are DM6420HR_REPEAT_SINGLE and DM6420HR_REPEAT_REPEAT.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Repeat is not valid.

IOCTL Interface:

```
int rc;
struct DM6420HR_MIO16 io_request;

/*
 * Write to Trigger Mode Register at base I/O address + 6
 */

io_request.reg = r_TRIGGER_6420;

/*
 * Only change bit 13 in register
 */

io_request.mask = 0xDFFF;
```

```
/*
 * Enable trigger repeat
 */

io_request.value = (1 << 13);

rc = ioctl(descriptor, DM6420HR_IOCTL_MOUTW, &io_request);
```

SetUserClock6420

```
int SetUserClock6420(
   int descriptor,
   enum DM6420HR_CLK Timer,
   double InputRate,
   double OutputRate,
   double *actual_p
);
```

Description:

Set up a board's user clock.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
Timer:	Indicates which counter to use. Valid values are DM6420HR_CLK0, DM6420HR_CLK1, and DM6420HR_CLK2.
InputRate	Input frequency to specified counter.
OutputRate:	Desired output rate from specified counter.
Actual_p:	Address where the actual programmed frequency should be stored. If this function fails, the actual frequency is not updated.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	Timer is not valid.

IOCTL Interface:

This function makes use of several ioctl() requests.

StartConversion6420

int StartConversion6420(int descriptor);

Description:

Issue a Start Convert (software trigger) command to a board.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EACCES descriptor refers to a file that is open but not for read access.

IOCTL Interface:

```
int rc;
struct DM6420HR_IO16 io_request;

/*
 * Read from Start Convert Register at base I/O address + 6
 */

io_request.reg = r_START_CONVERSION_6420;

/*
 * Any value works here because it is ignored making this request.  However, on return from
 * ioctl(), the member variable value contains the data read but in this case we're not interested
 * in the value at all … just the fact that the read was performed.
 */

io_request.value = 0;

rc = ioctl(descriptor, DM6420HR_IOCTL_INW, &io_request);
```

StartDMA6420

int StartDMA6420(int descriptor, enum DM6420HR_DMA DMAChannel, size_t TransferBytes);

Description:

Start DMA on a board's specified DMA circuit.

Parameters:

descriptor: File descriptor from OpenBoard6420() call.

DMAChannel:	DMA circuit to operate on. Valid values are DM6420HR_DMA1 and DM6420HR_DMA2.
TransferBytes:	Number of bytes to transfer in a single DMA. This value must be even because the entities transferred are 16-bit signed converted A/D values.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:

EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	DMAChannel is not valid.
EINVAL	No DMA channel was ever allocated to DMAChannel.
EINVAL	No DMA buffer was ever allocated to DMAChannel.
EINVAL	TransferBytes is odd.
EINVAL	TransferBytes is greater than the DMA buffer size.

IOCTL Interface:

```
int rc;
struct DM6420HR_DST io_request;

/*
 * Target of operation is first DMA circuit
 */

io_request.dma = DM6420HR_DMA1;

/*
 * DMA transfer size is 4096 bytes (2048 words)
 */

io_request.length = 4096;

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_START, &io_request);
```

StopDMA6420

int StopDMA6420(int descriptor, enum DM6420HR_DMA DMAChannel);

Description:

Stop DMA on a board's specified DMA circuit.

Parameters:

descriptor:	File descriptor from OpenBoard6420() call.
DMAChannel:	DMA circuit to operate on. Valid values are DM6420HR_DMA1 and DM6420HR_DMA2.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:
EACCES	descriptor refers to a file that is open but not for write access.
EINVAL	DMAChannel is not valid.
EINVAL	No DMA channel was ever allocated to DMAChannel.

IOCTL Interface:

```
int rc;

/*
 * Halt DMA on second DMA circuit
 */

rc = ioctl(descriptor, DM6420HR_IOCTL_DMA_STOP, DM6420HR_DMA2);
```

Example Programs Reference

Name	Remarks
dm6420-auto-burst	Demonstrates how to use the pacer clock, burst clock, sample counter, and channel/gain table to scan groups of channels multiple times.
dm6420-auto-scan	Demonstrates using the sample counter and pacer clock to do multi-scan sampling.
dm6420-dac	Demonstrates how to use the digital to analog converter.
dm6420-digital-interrupt	Demonstrates how to use advanced digital interrupts in event mode.
dm6420-digital-io	Demonstrates how to read and write the digital I/O ports. Uses port 0 for input and port 1 for output.
dm6420-dma	Demonstrates using DMA transfers to acquire data.
dm6420-dual-dma	Demonstrates using both DMA circuits on a board to transfer data.
dm6420-dual-dma-data-viewer	Demonstrates how to read the data from the dual_dma.dat file generated by the dm6420-dual-dma example program.
dm6420-multi-burst	Demonstrates how to perform an analog to digital conversion on multiple channels using the channel/gain table.
dm6420-sample-counter	Demonstrates how to use the sample counter to generate interrupts.
dm6420-soft-trigger	Demonstrates how to initiate an analog to digital conversion using a software trigger.
dm6420-speed-test	Demonstrates three different IRQ handling and data read methods: 1) using a callback routine invoked via the driver interrupt handler sending a signal to the process; data is read using streaming input, 2) polling the IRQ counter for an interrupt circuit; data is read one sample at a time, and 3) polling the IRQ counter for an interrupt circuit; data is read using streaming input.
dm6420-stream	Demonstrates using streaming input to read blocks of data from the digital input FIFO.
dm6420-test-lib-errors	Test program to validate library error checking. This program exercises the error checking code added to the library. All exported functions are verified. The following types of tests are performed: 1) a function fails when expected, 2) if a function fails as expected. errno is set as expected depending upon the cause of failure, 3) a function succeeds when expected, and 4) if a function can accept multiple valid inputs, it succeeds on all combinations of such inputs.
dm6420-throughput	Allows one to see how the driver works when a board is converting A/D data at a particular throughput rate and converted data is being read using a specified method. Data can be retrieved as follows: 1) reading one sample at a time, 2) using streaming input, 3) using single channel DMA, and 4) using dual channel DMA. In all cases, interrupts are used to signal availability of data. Every five seconds, the program prints status information.
dm6420-timers	Demonstrates how to program the 8254 programmable interval timers.
dm6420-user-timer	Demonstrates how to set up the User Timer to generate interrupts at a specified rate, poll the appropriate interrupt counter, and perform an A/D conversion when the counter changes.

Limited Warranty

RTD Embedded Technologies, Inc. warrants the hardware and software products it manufactures and produces to be free from defects in materials and workmanship for one year following the date of shipment from RTD Embedded Technologies, INC. This warranty is limited to the original purchaser of product and is not transferable.

During the one year warranty period, RTD Embedded Technologies will repair or replace, at its option, any defective products or parts at no additional charge, provided that the product is returned, shipping prepaid, to RTD Embedded Technologies. All replaced parts and products become the property of RTD Embedded Technologies. Before returning any product for repair, customers are required to contact the factory for an RMA number.

THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY PRODUCTS WHICH HAVE BEEN DAMAGED AS A RESULT OF ACCIDENT, MISUSE, ABUSE (such as: use of incorrect input voltages, improper or insufficient ventilation, failure to follow the operating instructions that are provided by RTD Embedded Technologies, "acts of God" or other contingencies beyond the control of RTD Embedded Technologies), OR AS A RESULT OF SERVICE OR MODIFICATION BY ANYONE OTHER THAN RTD Embedded Technologies. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES ARE EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND RTD Embedded Technologies EXPRESSLY DISCLAIMS ALL WARRANTIES NOT STATED HEREIN. ALL IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES FOR MECHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED TO THE DURATION OF THIS WARRANTY. IN THE EVENT THE PRODUCT IS NOT FREE FROM DEFECTS AS WARRANTED ABOVE, THE PURCHASER'S SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. UNDER NO CIRCUMSTANCES WILL RTD Embedded Technologies BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, AND SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College PA 16803-0906
USA
Our website: www.rtd.com