

# The DOM Hub Device Driver

Function and Design

Version 2.12

Describes driver release version 1.4

October 12, 2003

**John Jacobsen**

John Jacobsen IT Services  
for LBNL and the IceCube Collaboration

**John Jacobsen**  
Information Technology Services



1420 W. Edgewater Ave., Suite 3, Chicago, IL 60660  
Phone: +1 (773) 769 0522 FAX: +1 (775) 254 5992  
E-mail: [john@johnj.com](mailto:john@johnj.com)  
Web Site: [it.johnj.com](http://it.johnj.com)

This document can also be found at <ftp://it.johnj.com/pub/icecube/domhub/driver/doc>

<b>INTRODUCTION</b>	<b>1</b>
<b>THE ROLE OF THE DEVICE DRIVER</b>	<b>1</b>
<b>OUTLINE OF INTERFACES</b>	<b>1</b>
<b>FUNCTIONAL SPECIFICATION</b>	<b>2</b>
Building the Driver	2
Installing the Driver	2
Driver Startup and Initialization Parameters	3
Troubleshooting the Driver	4
The Driver System Interface in Detail	4
Communicating with DOMs	4
Communications in Simulation Mode	5
Control Functions	5
Updating the DOR Card Firmware	6
Additional Control Functions	7
The Interface to the Java Layer	12
<b>DRIVER DESIGN AND IMPLEMENTATION DETAILS</b>	<b>13</b>
General Remarks about Linux device drivers	13
Code Organization	14
PCI Interface to the DOR FPGA Firmware	14
Messages and Message Queues	15
Interrupt Handling	16
Direct Memory Access (DMA)	17
Time Calibration	17
Implementation of <code>/proc</code> Files	17
<b>REFERENCES</b>	<b>19</b>

## Introduction

The IceCube experiment, currently being commissioned at the South Pole, uses an array of about 5000 Digital Optical Modules (DOMs) to detect faint light signatures from extraterrestrial neutrinos. The DOM Hub is the primary testing and data acquisition interface to the Digital Optical Modules. A DOM Hub consists of a commercial, off-the-shelf PC with a CPU card attached to a passive PCI backplane. The backplane can accommodate up to eight DOM Readout (DOR) cards. These DOR cards are the hardware used to communicate with and control the DOMs. Each card is attached to up to four twisted pair cables, with two DOMs per cable, for a total of up to 60 DOMs per DOM Hub + 4 spare channels. There will be 80 DOM Hubs in the final detector; the full detector will consist of the  $80 \times 60 = 4800$  DOMs, the DOM Hub computers, plus additional data acquisition components for triggering and event processing. Several DOM Hubs will also be present at participating institutions for testing and development.

This document is meant to specify in detail the functionality and design of the DOR device driver, both to guide driver development and troubleshooting, as well as to help the users of the driver (DOM Hub application programmers).

Additional references (listed at the end of this document) may also be helpful. Basic device driver concepts are covered in much more detail in Reference 1. The device driver design and functionality depends very heavily on the DOR firmware, which is described in Reference 4. The DOMCOM device driver, a simpler predecessor to the DOM Hub device driver, is described in detail in Reference 2, Section 4.1.

## The Role of the Device Driver

The DOM Hub Device Driver (dh) provides a self-contained interface to the DOR cards, and, through that hardware, implements a low-level communications and control interface to the Digital Optical Modules.

The DOM Hub runs a commercial distribution of the Linux operating system with a kernel version of 2.4. (As of this writing, the most probable choice will be Red Hat 8.0, though that has not been fixed as of yet. The driver has yet to be tested against the brand-new 2.6 kernel.)

The driver runs as a kernel module. Application programs for testing and data acquisition use the driver to control and communicate with the DOR cards and, through these cards, the DOMs deployed deep in the ice. The driver must handle communications through and control of all the DOR cards concurrently. Messages received from and sent to the DOMs are buffered in both directions for maximal performance.

At the lowest level, the driver's operations consist of reading and writing of firmware (FPGA) registers on the cards via the PCI bus, as well as responding to interrupts generated by the DOR hardware. The complexities of buffering, register manipulation and interrupt handling are all hidden by the driver, which provides a straightforward interface for the application programmer via the Linux filesystem.

## Outline of Interfaces

For those who wish to understand role and function of the device driver, it is constructive to define the important interfaces which describe the layers of communication and control in the DOM Hub. These are, in order of least abstract to most:

*The Driver/Firmware Interface.* This is the "bottom level" of the device driver. The driver interacts with the DOM Hub card by reading and writing memory-mapped FPGA registers. The details of this interface are specified in Reference 4.

*The Driver System Interface.* This is the “top level” of the device driver. The system calls `open()`, `close()`, `read()`, `write()`, and `select()` are used to transmit and receive data. Furthermore, control functions (for example, time calibration, power on/off) are implemented through “virtual” files in the `/proc` filesystem. The Driver System Interface is described in detail below.

*Java Interfaces.* The functions of the driver are to be fully encapsulated by a few Java classes which will be used by the data-taking application. These classes will be described briefly below.

## Functional Specification

The following sections explain how the device driver is meant to be built, installed and used.

### Building the Driver

The device driver and supporting files are stored in the `dor-driver` project in the IceCube software repository on `glacier.lbl.gov`. The principle source files for the DOM Hub device driver are called `dh.c` and `dh.h` in the subdirectory `driver`. The other files in `driver` are used for testing the driver. DOR firmware versions are stored in `dor-driver/resources/dor-fpga`.

The build tools in the IceCube Software Development Environment (Ref. 6) are actually not the best fit for the driver, since that environment is optimized for the development of stand-alone, cross-platform software. The driver, being neither stand-alone or cross-platform, relies on a simple Makefile to guide compilation. The driver, when compiled, consists of a single object file `dh.o` which is to be loaded into the kernel. To compile the driver and associated test programs,

```
% make
```

*Please note!* You must have the kernel sources on your machine to build the driver correctly. To find out if you have these,

```
% ls /lib/modules/`uname -r`/build/include
```

If you see a “No such file or directory” error, then you’ll have to get your system administrator to put the kernel source tree on the machine, or do it yourself. Under Red Hat Linux, installing the kernel-source RPM appropriate to your processor and kernel release should do the trick.

### Installing the Driver

The installation of the driver has been simplified greatly since the last release (1.2) of the driver. To install the driver files into the appropriate directories, first build the driver as above, and then

```
% su
# make install
# make chkconfig
```

You should see output showing where the various driver files are installed. In addition to the actual driver file `dh.o`, you will see the latest firmware version, named (as a symbolic link) `dor-current.rbf`. There is also an “rc” script `dhrc` which enables the driver to be installed into the kernel when the DOM Hub boots. The “`make chkconfig`” step sets up the appropriate `/etc/init.d` links for this file.

The DOR hardware relies on an FPGA design stored in one of four “pages” in flash. Page 0 is loaded automatically when the hardware is powered on. When it is loaded into the kernel, the DOR driver can

load alternate FPGA designs from the DOR flash. This is useful when installing recent firmware releases on older DOR cards. Pages 1 and 2 are available for programming, and 3 will be available in the future. See the section on programming the flash below for instructions.

If, for example, you wish to use the FPGA design on flash page 1 when the system boots up,

```
% su
# echo "1" > /usr/local/dor/conf/default_fpga_page
```

The driver will then automatically load the design on page 1 of each card.

You must make sure that a working design is programmed into flash page 1 on EACH CARD before doing this; otherwise, the FPGA on the cards missing that page will crash, and the only way to recover will be to remove this configuration file, reboot, and then re-program the appropriate flash page using the fail-safe design (page 0) which is loaded by the hardware itself when the system is powered on.

### Driver Startup and Initialization Parameters

The driver runs in two modes – normal mode and simulation mode. In normal mode, the driver expects real hardware to work with. In simulation mode, no hardware need be present. Basic control and communications functions of the driver are simulated. Rather than communicating with actual hardware, the driver allows an application to connect to and communicate with a DOM simulation program, in a fashion similar to that of a named pipe.

The driver is loaded into the Linux kernel using the `insmod` command. If you did “make `chkconfig`” as described above, this will be done automatically when the system reboots. To load the driver by hand, become the superuser. Then use the `insmod` command:

```
# insmod dh.o
```

This starts the driver in the default, “normal” mode. To start the driver in simulation mode,

```
# insmod dh.o parm_mode="sim_X"
```

where `X` is one of “queue”, “copy”, or “nocopy”:

The `sim_queue` mode is a loopback simulation mode used for connecting a DOM simulation to DOM Hub communications software via the driver. This is explained further in the section “Communications in Simulation Mode”, below.

Most people will not need the `sim_copy` and `sim_nocopy` modes; they are used for benchmarking purposes only. They do not cause actual transmission or buffering of messages but allow one to measure the overhead of the kernel dispatch routines when doing repeated `read()` or `write()` to the driver. `sim_copy` involves the copying of data between user space and kernel space; `sim_nocopy` skips the copying step.

There is another switch to the driver which allows for one to set blocking or non-blocking I/O. (Traditionally this is done with an `fcntl()` call; however, see the discussion about the interface to the Java layer, below.) The `parm_block` option toggles blocking or non-blocking I/O:

```
# insmod dh.o parm_block=1
```

installs the driver in blocking mode. `parm_block=0` (non-blocking I/O) is the default.

A final parameter of note is `parm_fpga_page`. This can be set to the value 1, 2 or 3 to tell the driver to load alternate DOR flash pages, and is used by `dhrc` when the configuration file `default_fpga_page` is present, as described in the previous section.

Additional parameters which can be used to troubleshoot the driver are discussed in the section on Design and Implementation Details.

## Troubleshooting the Driver

Since the device driver is installed directly into the Linux kernel, it does not make use of the standard input and output streams that most user programs use. Instead, diagnostic messages are written to the system error log using the kernel's `printk()` function. In addition, any anomalous conditions are recorded in the driver and can be viewed by inspecting the `lasterr` proc file.

To see the kernel log messages as they appear, run the following command as superuser:

```
# tail -f /var/log/messages
```

You will then see messages from the device driver (generally prepended with “dh”) as well as other system messages.

Simply typing

```
# cat /proc/driver/domhub/lasterr
```

will show the last error condition (if, in fact, an error has occurred). Viewing the file resets the error message to “no error.”

The driver status file `/proc/driver/domhub/status` (only present when the driver is loaded) also shows driver statistics which can be helpful for diagnosing problems.

## The Driver System Interface in Detail

### Communicating with DOMs

Communicating with the DOMs through the DOM Hub cards is relatively straightforward. The basic unit of communication is the “message”, a string of characters up to 4096 bytes in length. One simply opens the appropriate device file, which looks like a regular file in the Unix filesystem, and then reads messages from and writes messages to that file.

A file name convention is used which identifies the card, channel (wire pair), and DOM label (A or B) for each DOM. For example,

<code>/dev/dhc3w1dB</code>	DOM Hub card 3, wire pair 1, B-DOM
<code>/dev/dhc0w3dA</code>	DOM Hub card 0, wire pair 3, A-DOM

These files are created when the driver is installed using “`make install`” (see above). The files can only be opened when the driver is loaded into the kernel.

Before reading and writing these files, use `open()` just like for a regular file. Of course, the DOM you’re interested in talking to has to be powered on. The section “Additional Control Functions,” below, explains how to make sure a pair of DOMs is powered on.

The `read()` and `write()` system calls are used to receive and transmit data to and from the DOMs. It is important to note that, whether reading or writing, entire messages (up to 4096 bytes) are transferred. That means that, when reading, the user program's buffer must be at least 4096 bytes in length, and the program must request 4096 bytes to be read, even if less bytes might be returned by `read()` (in the case of a smaller message length).

Both blocking and non-blocking I/O are supported in the DOM Hub driver. The default is non-blocking I/O; this can be changed using the `parm_block` parameter at `insmod` time (e.g., `insmod dh.o parm_block=1`). When the device is open for non-blocking I/O, you should first call `select()` to make sure there is data to be read, or `read()` will return immediately with no bytes read. When `select()` returns true, you can read one or more bytes from the device. The same idea applies when writing in non-blocking mode – call `select()` to be sure that the output buffer isn't full and waiting to be emptied.

### Communications in Simulation Mode

The files `/dev/dhcXwYdZ` provide communications to up to 64 DOMs. For the purposes of simulation, an additional 64 “sister” files `/dev/simcXwYdZ` are also available; when the driver is loaded into the kernel in “`sim_queue`” mode, then writing a message to a DOM file causes the message to be readable from the corresponding sister file, and vice versa. This allows one to run DOM simulations using the driver without the necessity of hardware being present.

### Control Functions

Control functions relate to specific actions with the hardware on the surface, as opposed to communicating with the DOMs. Examples of such actions are turning DOM power on or off, or querying the status of the FPGA on a DOR card. Control functions for this driver are implemented through the `/proc` interface. The most common method of passing control information to and from Unix/Linux device drivers is through the `ioctl()` system call. However, since the driver is envisioned to be used by a Java application, it is advantageous to use the `/proc` filesystem for control. This is because reading and writing a file in the filesystem is much more straightforward in Java than calling a system function such as `ioctl()`, which would require a native method interface to a C stub routine rather than the conventional Java input and output functions. Also, `/proc` files can be read or written without the presence of the Java control application, namely by simple shell commands or scripts, which can facilitate testing and debugging.

“Write” functions which cause side effects, such as powering on or off a pair of DOMs, are implemented by writing the appropriate `/proc` file. “Read” functions which get data from the device without side effects (for example, retrieving the local oscillator value for a particular DOM Hub card) are implemented by reading the appropriate file. These files are stored in the top level directory,

`/proc/driver/domhub`

which contain all of the control files described in detail below. The DOM Hub `proc` files are only available when the driver is loaded into the kernel. Once the driver is loaded, `proc` files for each card appear in the top level directory `/proc/driver/domhub`. Each card has subdirectories for its wire pairs, and further subdirectories for each DOM. Unless the driver is running in simulation mode, a DOM Hub missing DOR cards will not show `proc` files for the missing cards. When the driver is running in simulation mode, extra cards (numbers 8-15) show up in the top level directory; these correspond to “sister” device files and can be ignored.

Many of the read functions will return text messages indicating whether or not the requested operation has succeeded. However, `proc` writes don't have the same ability to communicate the success or failure of their operations. For all operations, especially write operations, check the `lasterr` `proc` file and make sure the operation succeeded.

A list of proc files and their associated functions follows, starting with functions to program the DOR flash and reload the FPGA from flash.

### Updating the DOR Card Firmware

Since all communications and control in the DOR card ultimately depends on the manipulation of FPGA registers, it is of crucial importance to be able to upgrade the FPGA firmware. The DOR card contains a 1 MB flash chip (an AMD 29LV800B) divided into four 256 KB “pages.” Each of these pages can contain one FPGA image (241 KB). Since access to the flash memory is typically done through the FPGA, it is important to have a write protected, “fail safe” FPGA design which can always be used to reprogram the flash. By convention, page 0 contains this design. Pages 1-3 can contain alternative “user designs.” The fail safe design in page 0 is loaded when the power is cycled; any of the alternative pages can be loaded at a later time. By convention, page 3 can be used as “scratch space” to verify that a new design can read and write to the flash. (Page 0 can only be modified if the board is appropriately jumpered.)

Installing a new FPGA design onto the DOR card consists of the following parts:

Send the design into flash page X (X is 1..3)

Tell the FPGA to reload itself from flash page X

Verify that the new FPGA design can itself be used to reprogram the flash

The proc interface which implements these steps is illustrated in the following steps, where we load `fpga.rbf` (a raw binary FPGA design file) into flash page 1 on card 0.

```
% cd /proc/driver/domhub/card0      # Change to proc dir for card 0

% echo "flash 1 program" > flash1    # Tell the flash to accept data
% cat /tmp/fpga.rbf > flash1         # Write the FPGA design to flash pg. 1
```

When one reads back the bytes, one gets back all 256 KB:

```
% cat flash1 | wc -c
262144
```

If one wishes, one can compare the first 246784 bytes (241 KB) with what was sent. This is left as an exercise for the reader. We now cause the new flash design to be loaded into the FPGA:

```
% echo "reload 1" > fpga              # Load flash page 1 into FPGA
% cat fpga                            # See if it worked
FPGA registers:
CTRL  0x10000000
.
.
.
FLASH 0xff000000
```

You should see a list of 32-bit registers. You can also examine the `pci` proc file to verify that the PCI configuration space has been correctly restored.

If all looks good, we can test the ability of the new design to read and write the flash, using page 3:

```
% echo "flash 3 program" > flash3    # Tell the flash to accept data
% echo "Testing 1 2 3" > flash3       # Test to see if writing works or not
% cat flash3 | head -c 14            # Read 14 bytes back
Testing 1 2 3
```



The “fpga” program, installed in `/usr/local/bin` after installation of the driver, automates these steps and also allows the manipulation of individual FPGA registers for testing purposes (experts only!). It must be run as root.

### Additional Control Functions

The other control functions are a bit more self explanatory. Their associated `/proc` files, and examples of use are as follows:

#### **System-Wide Functions** (`/proc/driver/domhub/*`)

##### *Get current driver status*

File: `status` (one file)

Read operation: reports the current status (messages sent and received, etc.).

Write operation: none.

##### *Get current driver revision number*

File: `revision` (one file)

Read operation: reports the current RCS revision number (NOT release version) of the driver code `dh.c`.

Write operation: none.

##### *Get last driver error number and message*

File: `lasterr` (one file)

Read operation: reports the most recent anomalous condition that occurred in the driver.

Write operation: none.

Example:

(Assumes that an FPGA reload operation was attempted while a DOM communications file was opened. This isn't allowed because the FPGA is in an undefined state during reload.)

```
% cat /proc/driver/domhub/lasterr
1: FPGA reload attempted while communications in progress
% cat /proc/driver/domhub/lasterr
0: no error occurred since last query
```

The second “cat” gave no error because the first one reset the error flag.

##### *Power on/off all DOMs on the Hub*

File: `pwrall` (one file)

Read operation: none (use the individual, pair-wise `pwr` `proc` files).

Write operation: “on” or “off” to power on or off all the wire pairs plugged into the Hub. This is a clean way to power on all channels at once; if only one wire pair is needed, use the pair-wise `pwr` `proc` files.

### **Card-wise Functions** (/proc/driver/domhub/card\*/\*)

#### *Examining PCI configuration space for a DOR card*

File: cardN/pci (8 files)

Read operation: shows current PCI configuration space.

Write operation: restores the PCI configuration space, which is saved when the driver is loaded.

Example:

```
% cat /proc/driver/domhub/card0/pci
PCI configuration space for card 0:
      Vendor: 0x1234
      Device: 0x5678
      Command: 0x0087
      Status: 0x0480
      Revision: 0x00
      Class: 0x118000
      Cache line: 0x00
      Latency timer: 0xf8
      Header type: 0x00
      BIST: 0x00
      Base address register 0: 0x0000a001
      Base address register 1: 0x00000000
      Base address register 2: 0x00000000
      Base address register 3: 0x00000000
      CardBus CIS Pointer: 0x00000000
      Subsystem vendor ID: 0x0000
      Subsystem Device ID: 0x0000
      Expansion ROM Base Address: 0x00000000
      IRQ Line: 0x0a
      IRQ Pin: 0x01
      Min_Gnt: 0xf8
      Max_Lat: 0xf8
```

Your vendor and device numbers should agree with what is shown above.

#### *Show Snapshot of all FPGA Registers*

File: cardN/fpga (8 files)

Read operation: shows all the FPGA registers of a given card (except FIFOs).

Write operation: reload FPGA from DOR flash (see Updating the DOR Firmware, above).

Example:

```
% cat /proc/driver/domhub/card0/fpga
FPGA registers:
CTRL 0x80300001
GSTAT 0x000003c4
DSTAT 0x00000013
TTSIC 0x00000f0f
RTSIC 0x0002000f
INTEN 0x00000000
DOMS 0x00000c03
MRAR 0x00000000
MRTC 0x00000000
MWAR 0x00000000
MWTC 0x00000000
CKCT 0x00000000
DATE 0x00000000
CURL 0x00000000
DCUR 0x01f40000
FLASH 0xff000000
```

```
DOMC  0x00000000
DCREV 0x00000004
FREV  0x00000368
```

The final register value (FREV) can be used to determine the version of the DOR firmware currently running (368 corresponds to DC\_003h.rbf; 0x68 = ASCII('h')).

#### *Set or Read a single FPGA Register*

Note: Assumes 1 FPGA per card. Generalize appropriately if multiple FPGAs per card.

File: *cardN/fpga-regs* (8 files)

This function essentially bypasses the driver interface so that one can test and debug the FPGA firmware. To read a register, write “r N” to the file, where N is the register number to be read. Reading the file will then show the result. Reading the file without first writing the request will show “no data”.

To write a register, write “w N x”. Here N is the register number and x is the value to write. x is a 32-bit hexadecimal number (do *not* include “0x”).

Example, showing the reading and writing of register 0, which contains 255 as its beginning read value and is configured to store a value which can then be read back:

```
# echo "r 0" > /proc/driver/domhub/card1/fpga-regs
# cat /proc/driver/domhub/card1/fpga-regs
FPGA on card 1: register 0 = 268435456 (0x10000000)
# echo "w 0 2" > /proc/driver/domhub/card1/fpga-regs
# cat /proc/driver/domhub/card1/fpga-regs
no data
# echo "r 0" > /proc/driver/domhub/card1/fpga-regs
# cat /proc/driver/domhub/card1/fpga-regs
FPGA on card 1: register 0 = 2 (0x2)
#
```

The Perl script *fpga* installed in */usr/local/bin* streamlines the reading and writing of FPGA registers; this is helpful for driver or firmware debugging.

Please note that *fpga-regs* is writeable and readable only by root, because it is possible to do very bad things to a PC by writing certain values to firmware registers. For this reason, the *fpga* script can only be run by root.

#### **Pairwise Functions** (*/proc/driver/domhub/card\*/pair\*/\**)

##### *Power both DOMs on pair on or off (read/write)*

File: *cardN/pairM/pwr* (32 files)

Read operation: shows power status “on” or “off”

Write operation: powers DOM on or off (“on”, “off”, “reset”)

Example:

```
% cat /proc/driver/domhub/card2/pair0/pwr
Card 2 Pair 0 power status is on.
% echo "off" > /proc/driver/domhub/card2/pair0/pwr
% cat /proc/driver/domhub/card2/pair0/pwr
Card 2 Pair 0 power status is off.
%
```

The “on” and “off” scripts installed with the driver automate this process; e.g., “on 1 0” powers on the DOMs on wire pair 0, card 1.

*Determine whether a cable (DOM pair) is plugged in*

File: *cardN/pairM/is-plugged* (32 files)

Read operation: shows “yes” or “no”

Write operation: none

Example:

```
% cat /proc/driver/domhub/card2/pair0/is-plugged
Card 2 Pair 0 is not plugged in.
```

*Find the current draw of a DOM pair*

File: *cardN/pairM/current* (32 files)

Read operation: shows current value in milliamps

Write operation: none

Example:

```
% cat /proc/driver/domhub/card2/pair0/current
Card 2 pair 0 current is +24.444 mA.
```

**DOM-specific functions** (/proc/driver/domhub/card\*/pair\*/dom\*/\*)

*Get DOM ID*

File: *cardN/pairM/domX/id* (64 files)

Read operation: communicates with DOM to retrieve 12 byte, hexadecimal DOM ID

Write operation: none

Example:

```
% cat /proc/driver/domhub/card0/pair0/domA/id
Card 0 Pair 0 DOM A ID is 000135871AB2
```

*Show Communications State of DOM*

File: *cardN/pairM/domX/is-communicating* (64 files)

Read operation: shows if DOR-DOM communications firmware are in a state which allows packet transmission.

Write operation: “reset” : Reset communications (can try before power on/off or softboot to restore communications if lost for some reason).

Example:

```
% cat /proc/driver/domhub/card0/pair0/domA/is-communicating
Card 0 Pair 0 DOM A is communicating
```

### *Perform Time Calibration*

File: cardN/pairM/domX/tcalib (64 files)

Write operation: Write “single” to the proc file to get a time calibration. If another channel on the card is being calibrated, write will return –EAGAIN.

Read operation: Reads the binary time calibration record back. You must first initiate the calibration using the write operation, then read exactly the number of bytes in the calibration record (164), or no data will be returned. If the time calibration isn’t ready yet, read will return –EAGAIN.

The format of the time calibration record is as follows. The format reflects the maximum number of waveform samples (64); to obtain the actual number of samples used, let  $P$  equal the low-order 16 bits of the header; the number of samples is then  $(P-32)/2$ . All numbers are in little-endian byte order.

1. 4 bytes header (0x10098)
  2. 8 bytes DOR\_t0 (time of pulse generation in DOR)
  3. 8 bytes DOR\_t3 (time of reply pulse trigger in DOR)
  4. 64 \* 2 bytes DOM waveform measured in DOR
  5. 8 bytes DOM\_t1 (time of pulse triggering in DOM)
  6. 8 bytes DOM\_t2 (time of reply generation in DOM)
  7. 64 \* 2 bytes DOR waveform measured in DOM
- 
- Total 292 bytes

Since the shell command “cat” won’t work on the time calibration proc file (because of the fixed length record), a program tcaltest.c is included in the driver distribution which performs the write operation, then reads and displays the correct number of bytes for inspection by eye. Generally, however, the data will be read by a user application which will decode the record and do something useful with it.

Example time calibration:

```
% tcaltest $proc/card1/pair3/domB/tcalib
cal(0) dor_tx(0x2fd7ffa90a) dor_rx(0x2fd7ffad2d) dom_rx(0x3530c8385)
dom_tx(0x3530c85e7)
dor_wf(518, 517, 517, 518, 519, 519, 518, 518, 518, 516, 516, 515, 516, 516, 518,
518, 520, 521, 522, 524, 533, 551, 577, 610, 646, 687, 726, 767, 806, 846, 883,
917)
dom_wf(530, 528, 529, 529, 531, 531, 531, 529, 528, 528, 528, 527, 528, 527, 528,
532, 532, 532, 532, 535, 539, 551, 572, 600, 633, 671, 712, 749, 791, 828, 868,
901)
%
```

### *Get Communications Statistics*

File: cardN/pairM/domX/comstat (64 files)

Write operation: none.

Read operation: shows number of messages and bytes exchanged with this DOM

Example:

```
$ cat /proc/driver/domub/card0/pair0/domA/comstat
/dev/simc0w0dA 9000 messages up, 0 messages down (9000 total); 29952000 bytes up,
0 bytes down (29952000 total).
```

Note: a device file must be open for comstat to report any information for it. E.g.,  
/proc/driver/domub/card0/pair0/domA/comstat reports useful information only when /dev/simc0w0dA is open.

In addition to these control functions, other files in `/proc` may be defined which show the status of the driver or help in debugging.

## **The Interface to the Java Layer**

The DOM Hub device driver will be used most commonly by one or more DOM Hub DAQ applications. Since all of the communications and control functions implemented by the driver can be carried out by reading and writing files in the filesystem (either in `/dev` or `/proc`), the normal I/O functions in Java can be used without resort to Java Native Interface (JNI) methods. This feature simplifies application development and allows for easier simulation and testing at the Java Layer level.

A full-fledged data acquisition application will need to communicate with multiple DOMs at the same time (synchronous I/O). Such an application can implement synchronous I/O either by multithreading and using blocking I/O, or by looping over open channels in non-blocking mode (the default). Since select behaviour is not (yet) supported in the Java NIO (Ref. 5), it remains an open question how to mock-up a selectable channel in the driver; most likely, the reading application will simply retry the read operation on channels with no data available, possibly tuning time delays so as to avoid unnecessary burden on the CPU. In anticipation of this situation, the driver implements non-blocking I/O as the default.

There are a few Java software layers which live above the driver. The communications functions of the driver are implemented in the `RealDOMDriver` class. Control operations (power on/off, etc.) are implemented in `RealCardDriver`. These communications and control objects are produced by a factory object call `RealDHDriverFactory`. Java programs can use these classes to interact with the driver without knowing the details of reading and writing the device files or the `/proc` files.

In addition to `RealDOMDriver` and `RealCardDriver`, there are simulation classes which implement the same functionality for network-based simulations of the DOM Hub and DOM components of IceCube. For example, `SimSocketDOMDriver` implements the analog of `RealDOMDriver` for the case where one is exchanging messages, not with the real hardware (nor with simulation programs talking through the `dh` running in `sim_queue` mode), but over a network socket to a DOM simulation program (`domapp`). The abstract class called `DOMDriver` can be used by higher-level DAQ and testing programs which function the same way, regardless of whether they are talking to the real hardware, to simulation programs using the device driver, or a pure simulation running in a distributed, networked computing environment.

These helper classes (`RealDOMDriver`, etc.) are part of the `domhub` project in the IceCube software archive.

## Driver Design and Implementation Details

The final sections of this document give details on how the DOR device driver is implemented. The intended audience is those people who wish to contribute to driver development, or use existing code to help with new projects, or just to understand better how the driver works.

### General Remarks about Linux device drivers

The text by Rubini and Corbet (Ref. 1) provides a much more thorough introduction to Linux device drivers than can be provided here. However, a few general principles can be outlined.

A device driver can be thought of as a mechanism within the operating system to connect a user's program to some physical hardware. The driver hides the complexity of the specific operations required to cause the hardware to perform its tasks. The writer of the driver has to understand how the operating system works with processes, memory, files, etc.; the hardware interface in detail, down to the level of individual bits and bytes; and what the user's needs and tools are likely to be.

There are a few different classes of Linux device drivers. The DOR card is implemented as a character device. Character devices are meant to be written to and read from as streams of bytes. Block devices such as disks and network drivers behave somewhat differently. Consult Ref. 1 if you're curious to learn more about such things. (There is a subtlety here: actually the driver operates on chunks of bytes (packets or messages) rather than character streams, but the way the driver is used matches the traditional character driver fairly closely.)

The general behaviour of character device drivers under Linux (and Unix in general) is shaped by a core design philosophy of the Unix operating system, namely, to treat as many things as possible, including physical devices, as files in the hierarchical file system. The files can be opened, closed, read, and written using the same functions one uses to manipulate regular files.

The task of the driver writer, then, is largely to construct the appropriate "callback" functions which are called when the user calls `open()`, `close()`, `read()` and `write()`. In our case, the driver functions corresponding to these standard C library calls are `dh_open()`, `dh_close()`, `dh_read()`, and `dh_write()`. These functions arrange for the transfer of data from privileged "kernel space" (where the driver functions run) to the application's "user space" buffers. Generally speaking, additional control of the driver can be provided by an implementation of `ioctl()` and/or implementations of `/proc` files. As mentioned earlier, `/proc` files are favored for the DOR driver because of Java's platform-independence restrictions. The DOR driver therefore takes the Unix philosophy of "a file for everything" to the limit!

The callback functions are all packaged up into a compiled bit of C code (a "module") which can be either compiled statically into the kernel or loaded dynamically at run time using the `insmod` command.

The driver also consists of initialization code (`dh_init()`) and cleanup code (`dh_cleanup()`). The initialization code, invoked at `insmod` time, registers the `read/write/open/close` and `/proc` callbacks with the kernel, allocates software copies of write-only FPGA registers, and finds the PCI addresses of available DOR cards in the system. Further initialization such as creation of message queues is done when a user program opens a device file (`/dev/dhc*wd*` or `/dev/simc*wd*`) with `dh_open()` in order to communicate with a DOM. These message queues are deallocated when the device file is closed.

It is important to note that, although there are up to 8 DOR cards and up to 64 DOMs on a DOM Hub, there is only one copy of the driver code installed in the kernel. All of the devices share the same callback functions, though each has its own message queues and set of FPGA register copies. This makes sense because all the devices are identical (save for their DOR card IDs) and are handled the same way. The

situation reflects the fact that it is unlikely that substantially different DOR firmware designs will be running on the same Hub at the same time.

## Code Organization

Since the driver is basically a set of callbacks with supporting functions, the structure of the code is pretty straightforward. As mentioned earlier, all the code for the driver resides in a file called `dh.c` with a supporting header file `dh.h`, which defines the relevant numerical constants and FPGA register address locations (relative positions within the PCI resource space). Lower level functions are generally located closer to the top of `dh.c`, while the higher level functions that use them are towards the bottom. To avoid namespace pollution in the case that any symbols are exported, each function and global variable begins with the prefix `dh_`, and most functions and global variables are tagged with the `static` keyword.

## PCI Interface to the DOR FPGA Firmware

The DOR card firmware uses a commercially available firmware product to implement the 32-bit PCI specification. In addition to the PCI machinery, the FPGA also implements a set of registers used to control the DOR card and communicate with the DOM. The PCI functionality allows the device driver to use a set of standard kernel functions to discover the location of those registers in memory or I/O space.

The function `dh_get_pci_devices()` performs this discovery process by looping on the kernel function `get_pci_device()`, looking for the appropriate vendor (`DH_VENDOR`) and device (`DH_DEVICEID`) values. It then uses additional PCI functions to “enable” (in the sense of the PCI standard) each device and get its interrupt request line (IRQ) number. It also determines whether the FPGA registers are kept in memory-mapped or I/O space and finds out exactly where they live. The IRQ number and the register locations are stored in a struct `dh_pci_dev` for each device. The functions

```
dh_read_regi()
dh_write_regi()
dh_set_write_regi_bits()
dh_clear_write_regi_bits()
dh_test_read_regi_bit()
```

are then able to use the register locations to read or write the appropriate FPGA registers. The higher-level functions call these functions to accomplish the basic I/O and control tasks of the driver. (NOTE: memory-mode accessing of registers is temporarily disabled; I/O space should be used instead, and this is in fact the way it is done by all versions of the DOR firmware).

The loading and re-loading of the FPGA firmware is somewhat tricky but is explained in Ref. 4. One writes an FPGA image (`.rbf` file) into the DOR flash memory, then pulls on the correct register to initiate a re-load of the firmware from flash. Since the FPGA design implements the PCI functionality, reloading the design causes an “upset” in the PCI configuration registers, which have to be saved before reload and then restored afterwards. This is taken care of in `dh_reload_fpga_from_flash()`. Other functions used for firmware programming and loading are:

```
dh_fpga_write_proc(): implements the /proc user interface to dh_reload_fpga_from_flash();
dh_save_pci_config_space() and dh_restore_pci_config_space(): save and restore PCI
configuration during reload;
dh_flash_write_proc(): /proc callback which allows the user to program a new FPGA design
into flash
```



## Messages and Message Queues

Despite the fact that, from the kernel's point of view, the driver in question is a "character device," which suggests that bytes, rather than byte-aggregates, are the fundamental unit of data transfer, the DOR driver in fact transfers "messages" from 1 to 4096 bytes in length. This design choice allows one to move the DOM messaging protocol all the way down into the firmware, resulting in a significant savings in processor overhead for both the DOM and the DOM Hub. The decision has several implications for both the driver and the application writer. One nice feature, again, is that many details of the protocol are hidden. However, as stated above, a `read()` request to the driver must ask for the maximum message size, 4096 bytes. Less bytes may be returned in the case of a smaller message, but at least that many bytes must be allocated to the buffer in the calling application. If a smaller buffer were allocated, the driver would have nowhere to put a larger message arriving from the DOM.

The chunking of data into messages also affects how data is buffered in the driver. Buffering is crucial because it allows the DOR hardware to transfer data into memory when it is available (i.e. at interrupt time), regardless of whether a user application is reading it at that very moment; and, similarly, it allows a user application to do other things after writing data into the driver, which may take time to transmit the completed message to the DOM due to the restricted bandwidth of the twisted quad cable.

Messages are buffered in circular queues of fixed-length, 4100-byte-wide arrays (4096 bytes + 32 bit byte count). Each device file has an input and an output queue which are allocated in their entirety when the device file is opened. The queues accommodate 100 output and 100 input messages per device file. This may not seem like much but should be more than adequate since the typical mode of operation in the DOM is "call-and-response" (DOM Hub initiates request with one message, then waits for the DOM's reply)<sup>1</sup>. Although fixed-length queues might seem strange, it is anticipated that the largest message lengths will be the most common (waveform transmission). The choice of pre-allocated queue buffers allows one to avoid the overhead and complexity of repeated memory allocation and de-allocation.

In the case of the driver running in the "sim\_queue" mode, a DOM simulation program opens one device file (e.g., `/dev/simc0w0dA`), and a DAQ or test program opens the corresponding sister file (`/dev/dhc0w0dA`). Each device file has its own input queue and an output queue; the input queue of one is connected to the output queue of the other within the `dh_write()` callback function. In other words, a write to `/dev/simc0w0dA` causes the message to be queued in and readable from `/dev/dhc0w0dA`.

Message queue manipulations are provided by the following functions, some of which have self-explanatory names:

```
dh_init_message_queue()  
dh_queue_isempty()  
dh_queue_isfull()  
dh_num_in_queue()  
dh_release_message_queue()
```

`dh_queue_get()` – read a message from a queue

`dh_queue_add()` – add a message to a queue

`dh_move_message()` – move a message from one queue to another (as described for `dh_write()`, above)

The queue functions make use of `dh_copy_message()`, which moves whole messages around within kernel space, or from kernel space to user space (see Ref. 1 for a detailed explanation of the difference between user space and kernel space).

---

<sup>1</sup> In the case where the DOM is running in one of its boot modes (Iceboot or Configboot), many messages in a single direction are possible. A flow control mechanism in the driver, DOR firmware and DOM Boot code ensures that no packets are lost if the user application is too slow to read messages out of the driver's buffers and the buffers become completely filled.

## Interrupt Handling

Almost every device driver makes use of interrupts, which allow the hardware to signal the driver that an event has occurred. The most typical example is when the device has received data that should be read out and buffered by the driver. The DOR firmware can provide a variety of interrupts, but the most important are interrupts that signal the presence of data in the RX (receive) FIFO, or the availability of enough space in the TX (transmit) FIFO to receive a message to be sent to the DOM.

A Linux driver handles interrupts by registering an “interrupt handler” with the kernel. The interrupt handler is a subroutine which is called immediately when an interrupt occurs. This handler must determine if the interrupt in fact belongs to it (interrupts can be shared between multiple handlers), and then handle whatever condition occurred to cause the interrupt. This has to happen quickly because the processor will generally not do anything else until the handler exits. Operations that can’t be accomplished immediately are typically scheduled to be handled at a later time by a so-called “bottom-half” routine which will be invoked at a time that is more convenient for the kernel, as explained in Ref. 1. In the case of the DOR driver, unfortunately, bottom halves are not scheduled quickly enough to fill or drain the communications FIFOs without overflowing (at least given the current FIFO depths in the DOR firmware), so this is currently done immediately in “bh-like” subroutines called by the interrupt handler.

In the DOR driver, the same handler `dh_intr_hdlr()` gets registered for each DOR card. Since interrupts are shared in the PCI specification, the handler might get registered multiple times for a single IRQ. When a signal occurs on that interrupt line, each handler runs and looks at the firmware registers to see if the interrupt belongs to its card. If it does, it then sees what sort of interrupt it is (using the firmware registers), temporarily disables interrupts of that kind on that card, and executes the appropriate “bh-like” drain or fill routine. If an unknown interrupt occurs, then all interrupts on that card are disabled as a last resort.

The “bh-like” functions for data transmission called by the interrupt handler are `dh_do_send_message()` (for TX) and `dh_do_recv_message()` (for RX). The functions check to see which FIFOs have room to receive data (TX), or have data to read out (RX); perform the appropriate fill/drain operations; and then reenables the appropriate type of interrupt for that card.

All this somewhat low-level stuff connects with the actual reading or writing user program in the following way. When the user program calls `write()`, the message is queued by `dh_write()` and TX interrupts are enabled (they are disabled by default). RX interrupts are enabled by default; when data arrives in the DOR card from the DOM, the interrupt handler uses `dh_do_recv_message()` to queue the message and then, as a last step, awakens any reading process which may have blocked in `read()` / `dh_read()`.

Most drivers which handle interrupts are susceptible to race conditions. These are avoided in the DOM Hub driver by keeping separate data structures for each device file, and by using spinlocks to guarantee mutually exclusive access to shared resources. More on spinlocks and race conditions can be found in Ref. 1.

Additional types of interrupts, such as interrupting when a DOM pair cable is plugged into or unplugged from the DOM Hub, are described in the DOR firmware interface, but they are not (yet) implemented in the driver.

For testing purposes, interrupts can be disabled in the driver using the `parm_rx_poll` and/or `parm_tx_poll` options at insmod time. E.g.,

```
# insmod dh.o parm_rx_poll=1 parm_tx_poll=1
```

disables interrupts and causes the driver to poll the firmware directly when sending and receiving data.

## Direct Memory Access (DMA)

DMA is a way of copying blocks of data to and from a peripheral such as the DOR card without having to issue repeated single instructions on the PCI bus. The driver provides the hardware with a direct pointer to a memory location and a count, and the hardware reads from or writes to this memory space without the driver having to do multiple reads or writes on a FIFO. As a result, using DMA can result in significantly better performance. In the case of the DOR driver, DMA is advantageous only for reading out RX messages because they are significantly longer than TX messages (4096 bytes vs. roughly 8-16 bytes).

In August of 2003 I implemented DMA in the receive portion of the DOR driver interrupt handler, and used fine-grained benchmarking tools provided by Andrew Morton (ref. 10) along with a rebuilt Linux kernel to calculate that the DMA version was about three times faster than the non-DMA version for receive. Preliminary numbers from these tests indicate that the DOR driver will provide adequate throughput even for the fully-populated DOM Hub with 60 DOMs each running at a data rate of 2 Mbps or less. DMA for TX was added for testing purposes in October of 2003.

Enabling and disabling DMA can be done using the `parm_tx_dma` and `parm_rx_dma` parameters at `ismod` time; e.g.,

```
# insmod dh.o parm_tx_dma=1 parm_rx_dma=0
```

will load the driver configured to use DMA for TX but not for RX. Similar parameters `parm_rx_poll` and `parm_tx_poll` can be used to enable polling I/O instead of interrupt-based I/O; this can be helpful for debugging race conditions between the read/write methods and the interrupt handler.

## Time Calibration

A crucial aspect of the IceCube experiment is the ability to calibrate the free-running DOM clocks against a global time standard with an accuracy of ~5 ns. The problem is solved by sending short time calibration pulses in both directions on the cable and digitizing each at the opposite end, and then collecting the local and remote time-stamps and digitized waveforms of each pulse at the surface. The resulting data can be analyzed to give a very precise measurement of the offset between local (DOM) and global time. The bulk of this work is done in firmware in the DOM and in the DOR card; the driver merely formats the information from the firmware and makes it available to a user application. Currently the time calibration must be initiated “by hand” by writing to the `tcalib` proc file. Eventually, time calibration will occur synchronously during a brief shutdown of communications, and this will be controlled completely by the hardware.

Some care is taken in the driver to guarantee that the time calibration and communications don’t interfere with each other: a global spinlock protects the firmware on each card so that time calibration initiation and readout can occur completely asynchronously along with communications, without races.

## Implementation of `/proc` Files

As mentioned before, it is most common to use `ioctl()` to perform device control tasks not associated with I/O, but the `/proc` interface allows one to do these tasks more easily with Java programs. Because of the somewhat complicated heirarchical structure of the DOM Hub functions (8 cards supporting 4 wire pairs each; each wire pair supporting 2 DOMs), some extra code was written to help package the `/proc` interface in such a way that adding or removing additional control files or subdirectories would be more straightforward.

The `/proc` filesystem is discussed in detail in Refs. 1, 8 and 9. Each file or subdirectory in `/proc` is associated with a `struct proc_dir_entry`. In the DOR driver, this structure is wrapped up along with a few other variables in a `struct fancy_pde_entry`. These extra variables contain the file or directory's name and a flag indicating whether the file was successfully "registered" with the kernel. This allows the function `dh_cleanup_proc_entries()` to be called at any point if the module is removed from the kernel or if initialization fails for some reason; only proc files which have been successfully registered are then unregistered. `dh_initialize_proc_files()` is called at module init time to register all the proc files; it uses the helper functions `dh_make_fpde_as_file()` and `dh_make_fpde_as_dir()` to cleanly register the files/directories. As stated earlier, only cards physically present in the DOM Hub (or detectable using the PCI interface) show up in the proc filesystem. The exception to this rule is, of course, if the device driver is running in simulation mode, in which case proc files are not only shown for cards 0..7, but also 8..15 (the files `/dev/simcXwYdZ` correspond to the higher-numbered cards).

Each proc file has optional read and write callback functions which are supplied as arguments to `dh_make_fpde_as_file()`. The function also allocates a data structure of arbitrary size (specified by the `data_size` argument) to be used to pass data to the callback functions. This allows the same callback to be used for multiple proc files. For example, each card has an `fpga` proc file – each of these files uses the callback `dh_fpga_read_proc()` to return status information on the card's FPGA firmware; the callback looks at the data structure passed in through the `data` argument (`data_size` bytes) to determine which card to report data for.

The callbacks report data (`dh_*_read_proc()`) or get requests (`dh_*_write_proc()`) by transferring data to and from the user program through the `buffer` argument to the callback. `snprintf()` is generally used to report data in the read callbacks and `sscanf()` or `strcmp()` are used to parse the arguments to write requests. Ultimately these result in FPGA registers being read or written to accomplish the requested task, unless the driver is running in simulation mode. In that case, most write functions do nothing, and the read functions attempt to report something reasonable.

## References

1. Rubini & J. Corbet, *Linux Device Drivers*, 2nd Ed., 2001. O'Reilly & Associates.
2. J. Jacobsen, *Data Acquisition and Control Software for AMANDA's String Eighteen*, May 24, 2002. [http://rust.lbl.gov/~jacobsen/docs/string18\\_software.pdf](http://rust.lbl.gov/~jacobsen/docs/string18_software.pdf).
3. K. Sulanke, *The DOMCOM FPGA/PLD 8-bit register map*, 2001.  
[http://www-zeuthen.desy.de/~sulanke/Projects/DOMCOM/Doc/Domcom\\_API.doc](http://www-zeuthen.desy.de/~sulanke/Projects/DOMCOM/Doc/Domcom_API.doc).
4. K. Sulanke, *DOR-API Description*, 2003. Description of the FPGA registers used to control the DOR hardware. Contact [sulanke@ifh.de](mailto:sulanke@ifh.de) for details.
5. R. Hitchens, *Java NIO*, 1st Ed., 2002. O'Reilly & Associates.
6. S. Patton, *An Introduction to the IceCube Software Development Environment*, 2002. IceCube internal memo.
7. D. van Heesch, *Doxygen project*. <http://www.stack.nl/~dimitri/doxygen/>
8. E. Mouw, *Linux Kernel Procfs Guide*, Rev. 1.1, 2001. Contact [J.A.K.Mouw@its.tudelft.nl](mailto:J.A.K.Mouw@its.tudelft.nl) for details.
9. Nicholas McGuire, *The Linux Proc File System for Embedded Systems – Concepts and Programming*, [ftp://ftp.opentech.at/pub/rtlinux/contrib/hofrat/embedded\\_proc.pdf](ftp://ftp.opentech.at/pub/rtlinux/contrib/hofrat/embedded_proc.pdf), 2003.
10. Andrew Morton's Timepeg code, <http://www.zipworld.com.au/~akpm/linux/#timepegs>