Overview of the DOM Surface Software

John Jacobsen

Lawrence Berkeley National Laboratory

jacobsen@rust.lbl.gov

Version 2.0
10/18/01

Abstract

In 2000, a set of forty Digital Optical Modules (DOMs) were deployed deep in the ice underneath the South Pole. The purpose of these devices is to sense faint light signals from muon and neutrino radiation in deep Antarctic ice. From an engineering standpoint, the DOMs are somewhat autonomous embedded systems, with which one communicates over long electrical cables. At the other end of these cables sit hardware and software for data acquisition (DAQ). A DAQ for testing purposes exists which routes communications from a Control PC on a network to the DOMs. A set of software applications was created for this Control PC which provide the ability to communicate with and control the DOMs. The applications were developed using the Perl language. They rely on a multi-layered, object-oriented communications Application Program Interface (API), also written in Perl. These programs are supplemented by a network-enabled application running in an embedded PC-104 system which allows for direct readout and transmission of timing and communications signals.

This document provides an overview of the software for both the Control PC and the PC-104 CPU. It includes a cursory description of the underlying hardware, instructions on how to install and use the software, as well as some information about the internal structure of the programs.

Please note: This document is a work-in-progress. Some sections are still missing or incomplete. Updated versions of this document will be available on the author's Web page,
`http://rust.lbl.gov/~jacobsen/docs/dom_surface_software.pdf`(PDF format) or
`http://rust.lbl.gov/~jacobsen/docs/dom_surface_software.doc`(Word format).

Table of Contents

Introduction

This document describes software for communicating with and controlling the string of Digital Optical Modules (DOMs) deployed at the South Pole in January of 2000. The DOMs are custom-built optical sensors with fast digitizing electronics (ATWD & FPGA), memory, and a controlling CPU, whose purpose is the detection of faint light signals from charged particles travelling through transparent materials (e.g., water or ice).

The DOMs are buried in the ice at a depth of about 2000 m and are connected to the surface by long twisted-quad electrical cables. The cables deliver power to the DOMs and allow for the exchange of messages between the DOM and the surface hardware. This document describes the software on the surface in its current form. Software running in the DOMs is currently undocumented.

The surface software provides the interface which most people will use in order to operate, diagnose and collect data from the DOMs. Most of the software runs on a "vanilla" Linux control PC which communicates over an Ethernet with specialized communications hardware (Test DAQ) attached directly to the electrical cables which go into the ice. This software is written entirely in Perl. In addition, two small C-language applications which handle specialized tasks run in an embedded PC-104 Linux processor in the Test DAQ. The Perl code in the control PC communicates with the C applications in the PC-104 CPU over network sockets. This document includes descriptions of the protocols by which the Perl software in the control PC communicates with the DOMs, as well as with the PC-104 CPU in the Test DAQ.

The subject of time calibration and other functions controlled by the Test DAQ was covered in *Control of the LBNL Digital Optical Module Test Boards Using a PC-104 Single-Board Computer*, by J. Jacobsen. This document supercedes that one.

The primary application programs to be documented here are:

domtest	The main program for testing and configuring the DOMs.
domtalk	A program for controlling DOMs in "boot mode."
domlogger	A program for ongoing data collection from the DOMs
domhv	A program for monitoring, setting and deactivating high voltage on the DOMs.

The API used by these programs to access and control the DOMs will be described. In addition, the programs **syncserver** and **gpstime** which run in on the PC-104 CPU in the Test DAQ are described in detail.

Overview of Digital String Components

Before launching into the use, care and feeding of the applications making up the DOM surface software, it is instructive to describe in more depth the various hardware elements at play. The setup has several variants - these include different configurations for testing and development at LBNL and the actual setup at the South Pole. As an example, we will focus on the setup currently at the South Pole which consists of four DOMs connected to the 2001 LBNL Test DAQ. (Many other channels at the Pole are instrumented through a different prototype DAQ produced by DESY/Zeuthen in 2000).

The following hardware elements are involved in this example (see Figure 1):

1) The DOM, which collects data from a photomultiplier tube, sends and receives messages over 2 km of twisted quad (TQ) electrical cable.
2) The Test DAQ which is accessible over a local area network and which sends and receives messages down the 2 km cables. The Test DAQ consists of a PC-104 CPU and four "Test Boards" with analog front-end electronics and a controlling FPGA. The Test Boards route serial character data between the TQ cables and the terminal server (see #3, below), and also send and receive bipolar electrical pulses on the TQ cables for the purposes of time synchronization. The PC-104 CPU "talks" to the test boards over a PC-104 (ISA) bus. Each test board talks to a single DOM. The application on the PC-104 CPU which provides the interface from the test boards to the network is called **syncserver**.
3) A network-to-serial-port terminal server. This makes the serial data stream to and from the test boards accessible by the network. It is connected to the Test Boards via RS-232 serial cables, and to the local network via an Ethernet cable. Four ports are dedicated on the terminal server, one to each Test Board/DOM.
4) A Linux "DOM Control PC" (typically rust.lbl.gov or fireball.spole.gov) which runs the applications which access and control the DOMs. The applications communicate to the terminal server using network sockets. An additional network socket is used to communicate with syncserver on the PC-104 CPU.
5) A GPS clock which sends UTC time signals to the PC-104 CPU. The data is sent as a string of bytes, over a serial cable, once per second. The current time is made available to applications running on the DOM Control PC, for the purposes of time calibration and synchronization. The program **gpstime** running on the PC-104 CPU is responsible for reading out the GPS time and handing it to syncserver.

The diagram in Figure 1 illustrates these relationships:

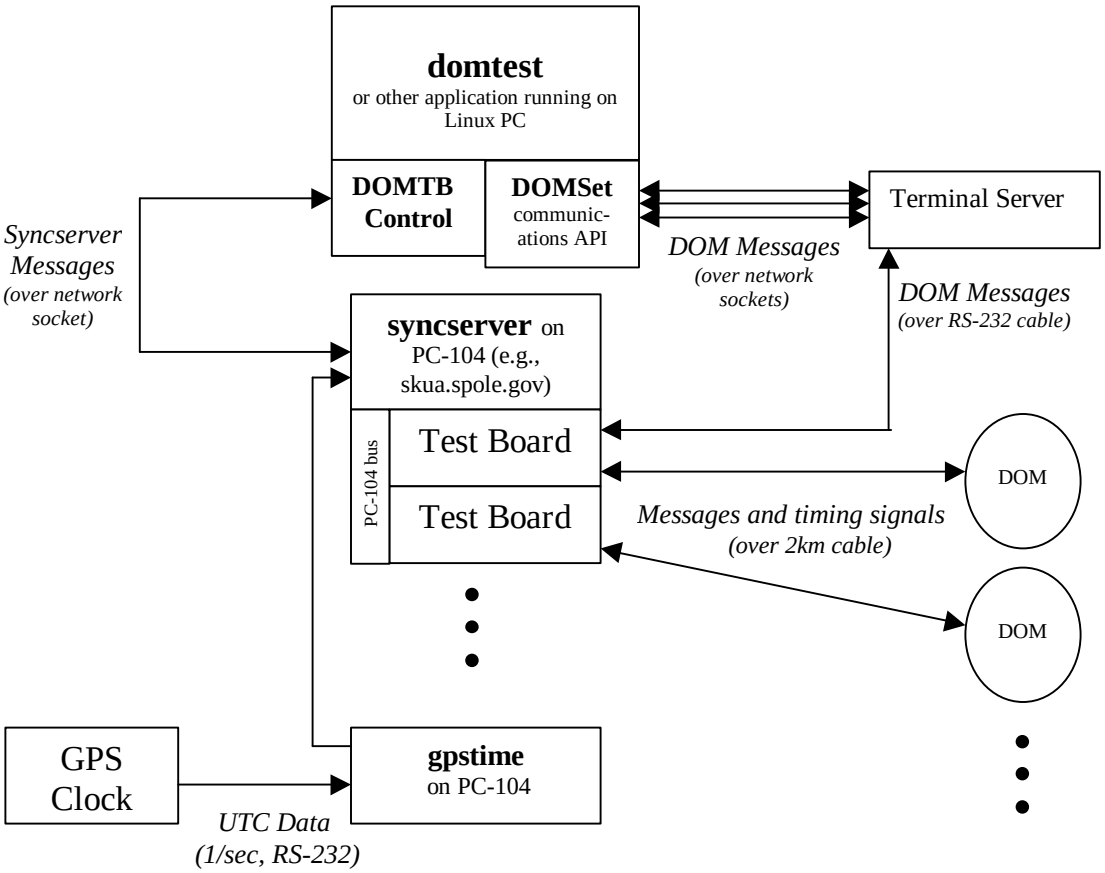


Figure 1 - Digital String Components with Communications Channels Illustrated

DOM Surface Software: History, Motivation and Philosophy

The basic task of the software described here is to enable communication with the DOMs. At the beginning of the DOM project, we simply needed for a user to be able to sit at a Windows or Linux workstation hooked directly to a DOM circuit board, to give commands to the DOM boot program and to see the result. Free, publicly-available software such as TeraTerm was adequate at first, but we soon needed extra capability such as the ability to upload files with cyclic-redundancy checks (CRC) and the ability to communicate with the DOMs over a network connection. The program **domtalk** was written to address these requirements. Meanwhile, a more sophisticated application (**domtest**) was also needed which would communicate with the application embedded in the DOM using a packet-based messaging protocol. Finally, it was anticipated that additional programs would be needed to automatically control the DOMs and collect data. The program **domlogger** does this. Since it was recognized from the beginning that multiple programs would need to communicate with and control the DOMs in well-defined ways, communications and control tasks were grouped together in an API. The programs and the API grew as hardware and firmware functionality improved. One important aspect of the API is the built-in networking functionality. This allowed for a lot of flexibility in the experimental setup which was helpful when developing and testing hardware in different locations. The philosophy readily accommodated the introduction of the Test Boards and the `syncserver` program.

All the programs described in this document are text-based rather than GUI-based. This was a conscious choice based on the realities of low bandwidth communications to and from the South Pole. It has allowed for extensive remote operation of the DOMs at the Pole, enabling us to demonstrate features and performance of the digital system during the Winter seasons.

The software was developed and maintained in a Concurrent Versions System (CVS) archive at LBNL, with mirrors at the South Pole. This facilitated software installation at the remote site and reduced version conflicts between different copies of the software.

The choice of Perl, a scripting language with associations to Web site development and system administration, might seem a curious one. However, Perl can be a very powerful tool for the quick construction of prototype systems. It is fully object-oriented and benefits from a huge user community, as well an extensive library of modules for networking and myriad other tasks. Development time for projects carried out in Perl tends to be significantly shorter than for other OO languages like C++. While somewhat "high-level" when compared to languages like C, it has functions for directly accessing data structures down to the level of individual bits (a crucial capability for the DOM software). Perl code executes surprisingly fast in most circumstances, and can incorporate separately compiled, C-language modules. In our case, speed was always constrained by network and communications bottlenecks, not by CPU time.

In the case of the DOM software, the original motivation for using Perl was to get something working as quickly as possible, but the language proved to be a flexible and robust enough basis on which to build a complete system. The development of the surface software took place in parallel with the hardware production, testing and debugging, with only minimal programming effort. It was possible to make additions to the software very quickly to test new hardware functions as soon as they became available. The fast development time afforded by Perl helped us to meet extremely tight deadlines arising from the operational realities of working at the Pole.

On the Test DAQ PC-104 system, the programs syncserver and readgps are written in C. In this case, C was chosen rather than Perl because we wanted the ability to run using the PC-104's flash disk only, which has very limited storage space, not enough to put Perl on (although there is a backup hard disk with a complete implementation of Perl on it, the system at the Pole is configured by default to run off of its flash disk).

Organization of the Software

The DOM surface software consists of four applications - `domtalk`, `domtest`, `domlogger` and `domhv`. These all use the communications API, which encapsulate the various functions of the DOMs and the means of accessing them through the terminal servers. The layers of the API are as follows:

1) **DOMSet - The Functional Layer.** A DOMSet object represents the functions of a collection of DOMs. Applications use the methods of DOMSet to instruct the DOM to perform high level tasks, and report any resulting error conditions. For example, the function SingleADCRead causes a message to be sent to the DOM requesting the value of a particular ADC channel on a particular DOM.
2) **DOMMsg - The Messaging Layer.** A DOMMsg object implements the messaging protocol using the exchange of packets between the surface and the DOM.
3) **DOMPacket - The Packet Layer.** Implements the sending and receiving of serial data in packet form.
4) **GenericDOMSerial - The Serial Communications Layer.** Implements the sending and receiving of raw data over a serial connection. Has support for Linux (LinuxDOMSerial.pm) and Windows NT (Win32DOMSerial.pm, now deprecated).

Each of these layers is implemented by means of a Perl package which describes an object (e.g., a DOMPacket object) with several methods (e.g., send() and receive()).

The specifics of each layer and the communications protocol will be described in more detail in further sections.

In addition to the communications API, there is a second API called **DOMTBControl** which allows one to communicate to a remote PC (the PC-104 CPU) running syncserver. This is further described in the sections on syncserver, below.

Getting and Installing the Software

The DOM software resides in a CVS repository on Rust.lbl.gov. Currently, one needs a login shell account on Rust in order to get the software. If you have such an account, and are logged into Rust, the command "cvs -d /usr/local/cvsroot checkout domsoft" will copy the repository into your current directory.

On a remote machine (at the South Pole, for example), you'll need:

```
setenv CVS_RSH ssh
cvs -d username@rust.lbl.gov:/usr/local/cvsroot checkout domsoft
```

Substitute your username on Rust for `username`. You'll have to give your Rust password to get the files.

Once you have the repository copied to your directory, you can begin to look at the Perl code. All the necessary code is in the directories `domsoft/src/domio` and `domsoft/src/portio`. Portio contains code used to interface with the PC-104 computer in the Test DAQ; domio contains all the other software. The CVS repository also contains some assorted documentation, as well as source code for the DOM Boot software. It does *not* contain the source code to the DOM application.

The DOM surface software has a "release" location on both Rust and on Fireball.spole.gov (the machine currently used to control the DOMs at the Pole). The location of the files is `/usr/local/dom/bin` and `/usr/local/dom/lib`. The former must be in your PATH variable in order to use the release version of the code (edit `$HOME/.login` or `$HOME/.cshrc`). If you just want to use the "release" code and your path is set up correctly, then typing `domtest`, `domlogger`, `domtalk` or `domhv` at the command prompt will start the appropriate program.

The script `domsoft/src/domio/install_domtest.pl` will copy all the files in your domio directory into the release directories. DO NOT DO THIS unless you know what you're doing. This means you if you're just looking at this manual for the first time!

Once you are comfortable making changes to the software (and, more importantly, other people working on the DOMs are comfortable with you making changes…), you can use normal CVS commands to bring the repository up to date with your modifications. See the CVS manual (http://www.cvshome.org/docs/manual/) for instructions about how to do this.

The software is already installed on Rust and on Fireball. See John Jacobsen (jacobsen@rust.lbl.gov) or Azriel Goldschmidt (AGoldschmidt@lbl.gov) if you have additional questions.

Using the Applications

As mentioned above, the DOM control applications are currently all text-based programs that can be used from any real or virtual terminal connected to the Linux control PC (e.g., rust or fireball). You need to have the directory /usr/local/dom/bin in your path. Starting one of the applications is then simply a matter

of typing its name at the shell prompt. You can also run the application directly out of your copy of the domsoft CVS repository.

Interacting with the DOM Boot program using `domtalk`

We start with an explanation of the two modes of DOM operation - "boot mode" and "application mode." In boot mode, plain text commands are give to the DOM Boot program ("domboot"). One can, for example, list the current files in the flash file system, read out ADC values, or tell the DOM to boot an application, at which time the DOM will be in "application mode." In application mode, one can no longer communicate using plain text. Instead, the messaging protocol (explained in detail below) is used. The messaging protocol was developed to allow for reliable communications at high speed using a well-defined data structure that could be "spoken" by either the DOM's FPGA (in the case of trigger data) or the CPU (in the case of slow control information).

When a DOM is in application mode, one can get back to the DOM's boot mode by power-cycling the DOM, or by issuing the "reboot" command to the DOM's embedded application using domtest.

Domtalk was the first of the four surface applications to be written, and is still the simplest. It allows you to communicate with the DOM in boot mode. Using domtalk, you interact with the DOM by sending it simple, single character commands, and seeing the result. If you're old enough to remember manually dialing a modem using a terminal program, the process is similar.

To run domtalk, type "domtalk" at the command line. You will be asked for the name of a terminal server and a port number. The default terminal server is at the South Pole; the default port is 3001, which corresponds to port 1 on the terminal server. As a shortcut, you can type "domtalk terminal_server port" on the command line, with terminal_server a valid terminal server and port a number between 3001 and 3032.

You may have to type return to see the boot prompt from the DOM. If no prompt occurs after hitting the return key, then the application may be running in the DOM - try power cycling the DOM. Also, if you're using a Test Board to communicate with a DOM, and have just powered on the Test Board DAQ, you'll need to load the Test Board's FPGA firmware using domtest. See the section "Test Board / PC-104 Functions," below, for more information. If the Test Board FPGA is loaded, all the hardware is hooked up correctly, and you are talking to the correct terminal server and port, then you should see the following appear on the screen:

```
Welcome to DOMBOOT release (CRC Enabled) of 15-Nov-1999

CRC table initialized...

Timeout value set to 30 seconds...
```

After the return key is pressed, the domboot prompt should appear:

```
Domboot 1.16 DOM 16  Enter command (? for menu):
```

With one exception, the keystrokes you type while running domtalk are sent to the DOM boot program. The exception is the escape sequence "Control-]". The escape sequence gives you a series of options:

```
c: Continue  s: Send file  b: Change baud rate  q: Quit
B: Boot DOM Application
```

With the next keystroke, you can tell domtalk what you want to do. The "Send file" option is used when sending a flash file to the DOM - do not do this unless you have instructed the DOM Boot program to accept a file! Otherwise you run the risk of having the DOM interpret the file you send as a sequence of commands which can have bad side effects such as the loss of data in the DOM flash file system. When

you select "send file," you can use tab-completion to list and complete file names, much like in newer Unix shells like `tcsh`. Flash files must be prepared in the correct format using the makeflash.pl or the makeflashcrc.pl script in the directory domsoft/src/boot.

The other options are more or less self-explanatory.

This is not a manual for the DOM Boot program, and there currently is none, but most of the commands you will encounter while interacting with the DOMs using domtalk are self-explanatory. Also, see the example, below.

Troubleshooting: if you start domtalk and can't seem to get a prompt from the DOM Boot program, make sure that:

a) the DOM is powered on (80 V), and is drawing current (~ 29 mA). You may have to reboot the DOM if it is already booted fully into the application (domtalk won't work with a DOM running its application);
b) you're talking to the right terminal server and port address
c) if using a test board to communicate with the DOM, the test board FPGA is loaded successfully (see the section on syncserver (Test Board Control with Syncserver), below.

`Domtalk` Example: Uploading a New FPGA Design on a DOM

In this example, we load an FPGA design file into the flash file system of the DOM. We assume we have prepared an FPGA design file prepared with a .fl extension (flash header bytes added using makeflash.pl). We further assume the DOM is on the first terminal server port of a terminal server termserv.lbl.gov.

First we connect with the DOM by firing up domtalk on `rust.lbl.gov`. In all the examples, text in bold is typed by the user; non-bold text is output from the program. <CR> means return is pressed:

```
$$ domtalk domtest.lbl.gov 3001
Welcome to /usr/bin/domtalk V0.2, by John Jacobsen / LBNL.
Trying to create socket ...OK.
Trying to talk to board domtest.lbl.gov:3001... press <return> to wake up
DOM.
<CR>
Domboot 1.16 DOM 16  Enter command (? for menu):
```

It is assumed that the DOM is already in boot mode or power-cycled just before return is pressed, above. We now list the contents of the flash file system using the "l" command:

```
Domboot 1.16 DOM 16  Enter command (? for menu): l

Filename:      dom_hv_intg_application
File ID:       4
Major Version: 0
Minor Version: 0
File Type:     Application
Numb. Sectors: 2
Start Offset:  0x00020080
File Size:     240684 (0x0003ac2c)

Filename:      dom_test6
File ID:       4
Major Version: 0
Minor Version: 0
File Type:     FPGA design
Numb. Sectors: 1
Start Offset:  0x00060080
File Size:     45705 (0x0000b289)
```

Now we upload the FPGA file:

```
Domboot 1.16 DOM 16  Enter command (? for menu): u
Enter name of file you will upload:
dom_fpga_test
Enter the file ID, major version, minor version:
0 0 0
Enter the file type (0 = unknown, 1 = FPGA design, 2 = routine, 3 =
application):
1
Send the flash file...
<Control-]>
c: Continue  s: Send file  b: Change baud rate  q: Quit
B: Boot DOM Application

s
Type the name of the file you want.  (Use the TAB key
for file completion / directory listings) : dom_fpga_test<TAB>
Unique match: ./dom_fpga_test.fl

./dom_fpga_test.fl<CR>
Sending file ./dom_fpga_test.fl now (61735 bytes)...
Done sending file!

c: Continue  s: Send file  b: Change baud rate  q: Quit
B: Boot DOM Application

c
Continuing... press RETURN to get DOM prompt....
Sent = 0x5c50dd64 -- Calculated = 0x5c50dd64
Sent = 0xebfd2ab6 -- Calculated = 0xebfd2ab6
Sent = 0x571bbd85 -- Calculated = 0x571bbd85
…. (some checksums omitted for brevity) ….
Sent = 0x361010a6 -- Calculated = 0x361010a6
Sent = 0x3873a835 -- Calculated = 0x3873a835
Done.  Wrote 61483 bytes, reached memory location 1018f0ab.
61483 bytes were loaded.
```

Now the FPGA design file must be loaded from the DOMs RAM into the flash file system. The “f” command does this:

```
Domboot 1.16 DOM 16  Enter command (? for menu): f
Blowing RAM image into flash...
Sector mask = 0xffffc80f...
First sector used for the new file will be sector 11
Flash ID = 00010001 22492249 00000000
Done programming RAM buffer into flash.
```

The “l” command can now be used again to verify that the image has been added to the flash file system. This is left as an exercise to the reader. To quit domtalk, type the escape combination “Control-]”, followed by “q.”

Please note that uploading an FPGA file to the flash file system is *not* the same as configuring the actual FPGA hardware using that file. Typically, the configuration of an FPGA is done by domtest (see “DOM FPGA Operations,” below), or automatically on startup by changing preferences in domboot.

Using `domtalk` to prepare the DOM for communication via the messaging protocol

To use domhv, domtest or domlogger, the DOM application must be started. This is important! The aforementioned programs use the DOM Message Protocol to talk to the applications, and if the application in the DOM is not started, then domboot will see the messages as garbage and may even cause trouble by erasing settings or deleting files from the flash file system. This design "feature" will be corrected in future designs of the DOM boot program.

To start the DOM application from domtalk, type "B" at the boot prompt, followed by the name of the application as it appears in the flash file system. To list the files in the flash file system, type "l" at the boot prompt.

Using domtalk/domboot, one can configure the DOM so that an application boots in the DOM automatically. Please remember, however, that using the Test Board setup, an FPGA must be loaded in the appropriate Test Board before any communication with the DOMs will be possible.

Controlling DOM High Voltages with `domhv`

The program domhv was written to provide a simple, easy-to-use interface to the DOMs which allows a user to turn on or off the high voltage without being confronted with a long and confusing list of other DOM functions. It is meant for users who want to operate the digital string in the mode where data is read out via the analog fiber optic cables into the standard AMANDA DAQ. It is a limited version of domtest (described in the next section), and does not allow one to exercise most of the digital functions of the DOM. It should be noted that some of the material in this section which applies to domhv also applies to domtest, as will be noted.

To use domhv, a database with DOM addresses and parameters such as high voltage must be prepared in advance by domtest. The default location for this database is in the directory /usr/local/dom/db, although the location of this can be changed with the command-line argument "-db" to domhv (e.g., "domhv -db /tmp" tells domhv to look for the database in the directory /tmp).

Both domhv and domtest produce HTML log files which record their activities. The location of these output logs can be specified when either of these programs starts up. The log file location can also be specfied with the "-dl" argument to either program. The log files record results of all commands given to the program, which are also printed to the screen in plain text format. Each time you run domhv or domtest you will get a new log file for each DOM that you talk to.

In general, one uses domhv and domtest by selecting choices from menus with a single keystroke. One can exit each menu using the escape key. The programs also have some protection against simultaneous use of the DOMs, although the protection is not foolproof (it can be circumvented by using a non-standard path for the DOM database). The protection is established using a lock file in the database directory. If a program crashes for some reason and the lock file must be deleted, one can do that by giving the "-unlock" argument to domtest or domhv.

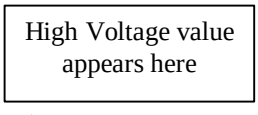
Please note! You should have some idea of what you're doing before turning high voltage on or off. In general, please observe the following:

a) Don't turn on or off high voltage on DOMs in the ice while the AMANDA detector is taking data.
b) Don't turn high voltage on a DOM in a lab unless it's in a sealed dark box.
c) Don't exceed the properly calibrated voltage setting for the DOM.

In the following example of the use of domhv, we have one DOM configured in the database. We look at its "high" voltage setting, see that it is a modest 5 Volts (for testing purposes - typical voltages are about 1000 V), turn on the voltage, see that it's on, then turn it off again. Domhv and domtest do not echo

keystrokes used to select choices from their menus, so in each case the menu item is highlighted in bold to show which choice is selected.

```
$$ domhv
Welcome to /usr/local/dom/bin/domhv (Author: J. Jacobsen, LBNL)
Run by jacobsen on Fri Apr 20 18:12:49 2001.
_____________________________________________________________________________

This program ties up whichever DOMs you access, so please don't leave it running.
Please enter a directory where domhv log files are kept [/usr/local/dom/logs]: /tmp
domdb: Found DOM with ID 1016 in DOM database.
Adding DOM 1016 at domtest.lbl.gov:3001 to DOMSet...
Opening domhv_log_ID1016trial002.html

AVAILABLE DOM FUNCTIONS         ESC->back
_____________________________________________________________________________
        S: List the currently available DOMs
        H: High Voltage Functions
        Q: Exit /usr/local/dom/bin/domhv


List the currently available DOMs

Please choose one of :

    (L)ist DOMs in database
    (V)erify that active DOMs are online
    (I)nteract with the DOMs
    (Q)uit /usr/local/dom/bin/domhv

List of available DOMs:
 CHAN  DOMID               ADDRESS     DELAY   DEF.VOLTAGE DEF.RATIO    THRESH     STATE
 ====  ====  =============          ====          ====       ====    ======    ======
 9998  1016     domtest.lbl.gov:3001     0.000            5       0.73     120.00    ACTIVE

Please choose one of :

    (L)ist DOMs in database
    (V)erify that active DOMs are online
    (I)nteract with the DOMs
    (Q)uit /usr/local/dom/bin/domhv


AVAILABLE DOM FUNCTIONS         ESC->back
_____________________________________________________________________________
        S: List the currently available DOMs
        H: High Voltage Functions
        Q: Exit /usr/local/dom/bin/domhv

High Voltage functions:                 ESC->back
_____________________________________________________________________________
        V: Show high voltage for currently selected DOMs
        S: Enable and set HV on one or more DOMs
        D: Turn off HV on one or more DOMs (set to 0 and disable)


Enable and set HV on one or more DOMs
Do you really want to turn on the high voltage [no]? yes
Setting High Voltage for board 1016
Setting LASER_BIAS DAC to 128.
OK.

Target voltage ANODE: 3.65 VDC (5.98016 DAC), DYNODE: 1.35 (2.21184 DAC).

Getting current HV limits...
Limits are DYNODE(400 DAC units), ANODE(600 DAC units).
Initiating HV request...
Got challenge value of 2264530.
Verifying challenge value...
```



```
HV enabled.
Successfully set voltage
DOM 1016 dynode DAC (8) : 4
DOM 1016 anode DAC (9) : 7
Measuring Voltages using DOM ADCs...
Voltage for DOM 1016:
Dynode Voltage: 2 VDC.
 Anode Voltage: 3 VDC.
                _____
 Total Voltage: 5 VDC.


High Voltage functions:                ESC->back
_____________________________________________________________________________
        V: Show high voltage for currently selected DOMs
        S: Enable and set HV on one or more DOMs
        D: Turn off HV on one or more DOMs (set to 0 and disable)


Show high voltage for currently selected DOMs
Measuring Voltages using DOM ADCs...
Voltage for DOM 1016:
Dynode Voltage: 2 VDC.
 Anode Voltage: 3 VDC.
                _____
 Total Voltage: 5 VDC.


High Voltage functions:                ESC->back
_____________________________________________________________________________
        V: Show high voltage for currently selected DOMs
        S: Enable and set HV on one or more DOMs
        D: Turn off HV on one or more DOMs (set to 0 and disable)


Turn off HV on one or more DOMs (set to 0 and disable)
Zeroing and disabling High Voltage for board 1016
Successfully zeroed voltage on board 1016.
Disabling voltage...
Disabled HV.


High Voltage functions:                ESC->back
_____________________________________________________________________________
        V: Show high voltage for currently selected DOMs
        S: Enable and set HV on one or more DOMs
        D: Turn off HV on one or more DOMs (set to 0 and disable)


AVAILABLE DOM FUNCTIONS         ESC->back
_____________________________________________________________________________
        S: List the currently available DOMs
        H: High Voltage Functions
        Q: Exit /usr/local/dom/bin/domhv


Exit /usr/local/dom/bin/domhv
Testing sequence ended on Fri Apr 20 18:15:57 2001.
$$
```

Interacting with the DOM Application using `domtest`

Domtest is currently the largest and most important program for controlling the DOMs. It allows users to exercise most of the functions of the DOM hardware in an interactive fashion. As with domhv, one interacts with domtest by choosing selections from text menus from single keystrokes. Domtest converts these menu selections to messages which are issued to the application running in the DOM. For each message, a response is received and the results presented to the user.

Domtest also is used to set up the DOM configuration database, associating DOM IDs with high voltage values, terminal server addresses, and test board IDs. This configuration database is used by domhv and domlogger.

Domtest is started by typing "domtest" at the Unix shell prompt. You will then be asked which directory you want log files to appear, and where your DOM database lives. You will then see the Top Level Menu (TLM):

```
AVAILABLE DOM FUNCTIONS         ESC->back
_____________________________________________________________________
        S: Select, list, update or configure the current list of DOMs
        D: DAC Functions
        A: ADC Functions
        H: High Voltage Functions
        F: FPGA Functions
        T: Test Board Functions
        L: Flasher Functions
        O: Other Functions
        Q: Exit /usr/bin/domtest
```

Configuring the DOM Database

Domtest, domhv and domlogger all use a common database to store information on the DOMs. This information consists of DOM IDs, their terminal server addresses and ports, default high voltage settings and other data for each DOM. Only domtest can change this database information.

Selecting "`S: Select, list, update or configure the current list of DOMs`" from the Top Level Menu will allow you to change this database, or to connect with new DOMs and store the relevant information in the database. The following choices will appear:

```
Please choose one of :

    (L)ist current DOM configuration database
    (V)erify that active DOMs are online
    (P)robe for more DOMs on terminal servers
    (C)hange downgoing delay constant for a DOM
    (T) Change SPE trigger DAC threshold
    (D)elete one or more DOMs from the database
    (A)ctivate a DOM
    (N) Deactivate one or more DOMs (remove from communications list)
    (S)ubstitute an existing DOM for another one
    (E) Associate a test board ID with a DOM channel
        (for DOMs connected to test boards only!)
    (Q) Change test DAQ address in database
    (I)nteract with the DOMs
```

Watch this space for more on this topic.

ADC, DAC and High Voltage Operations

Watch this space for more on this topic.

Time Calibration Functions

Watch this space for more on this topic.

Test Board / PC-104 Functions

Watch this space for more on this topic.

DOM FPGA Operations

Watch this space for more on this topic.

DOM Flasher Functions

Watch this space for more on this topic.

Automated Data Taking with `domlogger`

Domlogger allows one to repeatedly collect various types of DOM data. The program can be run interactively (see the "-interactive" switch, below), but is designed primarily to be run in the background ("batch mode"), generating data files in a target directory. In the absence of a completely functioning DAQ for the digital string, domlogger allowed for the automated collection of large amounts of DOM data over an eight month period. This helped to establish the performance and stability of the digital system.

Using domlogger

In normal operation mode at the South Pole during the year 2000, seven simultaneous versions of domlogger ran simultaneously, each collecting data from two to four DOMs. The resulting data was sent to LBNL in batches by the automated data transfer scripts polechomper and sableblanc. Reliability was ensured by a watchdog script "watch_domlogger" which was executed by the cron task scheduler every 15 minutes [check interval]. Watch_domlogger restarted any domlogger process that quit for any reason. This system required virtually no operator intervention during the entire 2000 season.

Domlogger can also be run by hand at the Unix shell prompt. Just type "domlogger" followed by a list (by DOM ID) of the DOMs you want to collect data from. For example, "domlogger 1016 1045". One can follow the list of DOMs by one or more optional switches, below.

The DOMs specified as arguments to domlogger must exist in a DOM database created by domtest. The default location of this database is /usr/local/dom/db, but can be changed using the "-d" switch, e.g., "domlogger 1016 -d /tmp/my_database_dir".

Before collecting data, domlogger attempts to bring up the high voltage on each DOM, based on the current high voltage setting in the DOM database. This feature can be suppressed using the -nohv switch. If all you want to do is set the high voltage, use the domhv program described above, or use the -hvonly switch to domlogger, and the program will quit after setting the high voltage.

By default, the following types of data are repeatedly collected for the DOMs, in this order:

hv	Current high voltage settings
fpga_status	Status of the DOM's FPGA (shows whether or not the FPGA is configured)
adc	Current ADC values (averaged over 1024 samples for all 8 channels)
dac	Current DAC settings for all 20 DAC channels
disc_rates	Shows the single- and multi-photoelectron discriminator firing rates as a function of threshold setting

waveforms	Reads out 10 sample waveforms for each of the two ATWDs
bg_tests	Runs the integrating background tests in the DOM (gets average waveforms and integrated waveforms for each ATWD and the fast and communications ADCs)
time_calib	Runs the reciprocal active pulsing ("RAP") time calibration between the DOM and the test board
sleep	Not really a type of data to be collected, the sleep option inserts a random delay between 0.1 and 1 second before continuing with the next cycle of data collection (at the top of this list)

To select only a subset of these data types, use the "-collect" switch to domlogger. For example, 'domlogger -collect "adc, hv" 1016' will read out only the high voltage settings and the ADC values for DOM 1016. Please note that the "-collect" switch does not change the order in which the data is collected.

When waveforms are read out, the "-sync" option can be used between separate domlogger processes to cause them to read out (nearly) simultaneously. This uses a semaphore construction provided by the operating system as described in the implementation notes, below. To use this option, each domlogger process must be started using "-sync <m>", where <m> is the number of simultaneous processes to be synchronized. One of the processes must be designated as the "gang leader" using "-sync <m> -boss". Only one set of synchronous processes is allowed. The domlogger process acting as the "boss" must be started first.

Note that the collection of time calibration ("time_calib") data requires that domlogger be able to connect to the Test DAQ over the network. The default location is currently dom-tbdaq.spole.gov, and can be changed with the "-tbaddr" switch. Please give a suitable test board address even if you're not collecting "time_calib" data. You don't want to run domlogger at LBNL or somewhere else and try to connect to the Test DAQ at the South Pole! (Note that this is currently not idiot-proof - it may be possible to cause adverse side-effects at Pole when running domlogger somewhere else, if satellite connectivity is available. This is on the list of things to fix.)

Domlogger normally writes all data for the specified DOMs to a file in the default data directory (/usr/local/dom/data/polechomper). If desired, data can instead be sent to the terminal using the "-interactive" switch. Or, the data directory can be changed to directory "mydir" using the "-datadir mydir". If running in the default, non-interactive mode, domlogger will start a new data file every four hours.

The following additional switches modify the behaviour of domlogger.

-h	help - show all available switches
-kill	Kill all domlogger processes currently running
-interactive	Run interactively, sending DOM data to the terminal rather than to files in the data directory.
-events <N>	Rather than collect indefinitely, only collect <N> instances of each of the data types.

Notes on the Implementation of domlogger

A look at the Perl code for DOMLogger will show things to be pretty straightforward. Generally, the bulk of the work is done by DOMSet.pm for the control of the DOMs (see the discussion of the communications API, below). DOMTBControl.pm provides the interface to the Test DAQ.

One interesting item is the implementation of the "-sync" option for simultaneous waveform readout. The synchronization is implemented using a Perl package called SyncProc.pm, written by Jacobsen for this project. SyncProc is essentially a wrapper for the System V IPC Semaphore functions, made available using Graham Barr's IPC::Semaphore, which is itself a wrapper for the semget and semop system calls. Semaphores are shared variables that can be changed by only one process at a time. Processes can be made

to wait ("block") until the semaphore reaches some target value. Let's assume there are four domlogger processes, three "subordinates" and one "boss." In this implementation, the function SyncProc::waitForGo allows each of the three subordinates to increment the semaphore by one, and then wait until the semaphore resets to zero. The boss uses SyncProc::goSignal to wait until the semaphore reaches three, then resets the semaphore to zero. When the semaphore resets, all three subordinates and the boss "simultaneously" issue waveform requests to each of their DOMs. "Simultaneously" is in quotes here because (a) all the domloggers are time-sharing the same CPU, and (b) the messaging requests to the DOMs are subject to network latencies. A test program measured the variation of the timing to be of the order 2 msec or less, not including the network latencies (b). Network latencies are not known but, for the 10 BaseT networking, should be much less than the 100 msec time scale required by this application.

Test Board Control with Syncserver

We now begin a series of sections on the internal workings of the DOM surface software. This section describes the control of the Test Boards in the Test DAQ using the syncserver program. The Test Boards were introduced in the '00/'01 season at South Pole to supplement the existing communications functionality provided by the DESY data acquisition system (DAQ) and terminal servers. The most important new features provided by the Test Boards were the ability to carry out time calibration and fast digital communications (work on the second item is still in progress as of this writing. The target bandwidth is about 60 kb/sec).

The Test Boards are controlled by a Linux PC-104 system. PC-104 is an embedded computing standard based on IBM-PC compatible, single board computers with a compact form factor and a shared, compact ISA bus (also known as the PC-104 bus). Up to four Test Boards sharing a PC-104 bus and a partitioned address space can be controlled by one PC-104 unit via that bus. A program called "syncserver" running on the PC-104 accepts commands over a network connection and translates these commands into memory-mapped I/O on one or more Test Boards. The Test Boards, in turn, control communications to the DOMs over the long cables. The Test Board API is described in some detail by Jerry Przybylski at `http://rust.lbl.gov/~gtp/local/testboard_API.htm`

Notes on the setup and installation of the PC-104 CPUs can be found in *Appendix A - Notes on Configuration of the PC-104/Test Board Systems.*

In the Test DAQ, most of the interaction between an application on the surface and the DOMs currently occurs via an RS-232 serial connection which the Test Board translates into tri-state differential encoding for transmission down the long cable. This process is largely transparent to syncserver - for normal communication, a process opens a socket to a terminal server which in turn connects via the RS-232 interface to the Test Board. However, for the transmission of time calibration pulses, normal RS-232 communications is suspended, and the Test Board must be told to generate a calibration pulse, and also can provide a digitization of the return pulse. This is the key functionality which syncserver must provide.

You may want to refer back to the system diagram (Figure 1), which illustrates how syncserver and the test boards relate to the other components in the system.

The Test DAQ software itself can be divided into the following components:

1. The **syncserver** PC-104 control software (written in C), which talks to the FPGA in the test board using memory-mapped registers in port I/O space.
2. A low-level C-language interface (**portio.c / portio.h**) which syncserver uses to talk to the test boards. It also includes functionality to talk to a PC-104 relay switch system, used to turn on or off the power to the test boards and other hardware.
3. Another C program for the PC-104 CPU, **gpstime**, which reads data from the GPS clock over a serial link. Gpstime makes this data available to syncserver for use in time calibration by writing to a file in

/tmp, which syncserver reads. The software setup for this serial link is described in the section, *Appendix A - Notes on Configuration of the* PC-104/Test Board Systems, below.

4. A **Jamplayer** package, from Altera, Inc., ported to the PC-104 system. This package allows syncserver to program and deprogram the FPGA.
5. A Perl package, called **DOMTBControl.pm**, which provides the interface whereby a Perl application on the surface can issue commands to syncserver.
6. One of the DOM Perl applications: **domlogger**, **domtest** or the skeleton test program **client_test.pl** which uses DOMTBControl.pm

All these software components are in the CVS[1] tree on rust.lbl.gov in the directories domsoft/src/domio and domsoft/src/portio. The former directory is the location of the Perl test and data logging programs (domtest, domtalk, domlogger). The latter is the location of the PC-104-specific software (syncserver, portio, Jamplayer) and the DOMTBControl.pm Perl interface to syncserver.

The following example will highlight the most important features of this design. Assume a Perl application (e.g. domtest) on the surface wants to send a timing pulse down the cable. The application must first create a DOMTBControl object, which opens a socket to syncserver running on the PC-104. When a timing pulse is desired, the application calls the method DOMTBControl::sendTimePulse which sends a single byte command on the socket. Syncserver reads this byte, interprets it as an instruction to send a timing signal, and uses functions implemented in portio.c to write the appropriate registers in I/O space. The FPGA in the Test Board generates the timing pulse, and provides a 16-byte integer timestamp of the timing pulse, which is then read out by syncserver. If all goes well, syncserver then sends a confirmation response byte on the socket, followed by the time stamp data, and the DOMTBControl method will return a "status ok" value so that the application knows that everything's fine; the method also returns the time stamp data.

The `syncserver` Program

As explained above, syncserver runs on the PC-104 and listens for network commands from a client application. When a command is given, syncserver reads and writes the appropriate FPGA registers on the test board, and sends response data back on the socket to the client.

The Test Board FPGA registers are accessed in memory-mapped I/O space using the inb() and outb() functions. A layer of routines, defined in portio.c, encapsulate this functionality. The details of which bits in which registers have which functions are specified in Jerry's Test Board API document.

Syncserver runs setuid root (its file permissions should be 04755). The source code is located in the portio directory. For compilation and installation instructions, see *How to Compile, Install and Run the Syncserver Software*, below. On the PC-104 CPUs in use at LBNL (skua.lbl.gov) or at Pole (domtb-daq.spole.gov), startup of syncserver is automatic at boot time.

The Perl interface to syncserver

The Perl package DOMTBControl.pm provides all the functionality one needs to talk to syncserver, and therefore to the Test Board.

When you create a new DOMTBControl object, the "new" method opens a socket to the PC-104 system (syncserver listens on port 3666). Calls to that object's methods cause data to be written to syncserver over the socket. This data consists of a control byte along with zero or more data bytes, with the number of data bytes fixed by the value of the control byte. Response data consists of a status byte followed by zero or

[1] CVS - Concurrent Versions System. This software allows for version control for code development by multiple developers running on separate development machines. All the DOM software except for the DOM Application and FPGA designs are in the domsoft CVS archive on rust.lbl.gov. For more information, see **http://rust.lbl.gov/~jacobsen/cvs**.

more message data bytes. The details of the interface are described below, in **The Network Interface to syncserver**.

If a DOMTBControl method returns the string "STATUS_OK" in the first argument, that indicates that the command completed successfully. Otherwise, the error string will contain information about what went wrong.

As an example, the following code sends a time tick down the cable to the DOM:

```
use DOMTBControl;

my $tb = DOMTBControl::new("skua.spole.gov"); # Open connection
die "Can't connect!\n" unless defined $tb;

# Send the time pulse:
my ($err, $timestamp0, $timestamp1) = $tb->sendTimePulse(0);

if($err eq "STATUS_OK") {
   print "Time pulse was acknowledged by syncserver.\n";
   print "Timestamp: $timestamp0 $timestamp1.\n";
} else {
   print "Time pulse request generated an error : $err.\n";
}

$tb = undef;
# When the test board object goes out of scope
# or is no longer referred to by any
# scalar, the socket connection to the PC-104 is closed.
```

For a real time calibration, the example would have to include DOMSet function calls to the DOM which tell it to "go quiet" (stop communications using the RS-232 interface), digitize the time tick, send a time tick in response, and DOMTBControl functions to read out the received time tick in the test board. DOMSet functions are described in the section entitled “The Communications API,” below.

Current methods supported by DOMTBControl.pm:

- new() -- Creates a DOMTBControl object and opens a connection to syncserver running on the PC-104.
- echoTest() -- just a test to make sure things are working, by sending a byte to syncserver and making sure syncserver sends the same byte back.
- FPGARegisterTest() -- syncserver writes byte patterns to FPGA Control Register 0 & makes sure they can be read back ok
- sendTimePulse() -- causes syncserver to tell the FPGA on the Test Board to send a timing pulse down the cable.
- readTBTimetickData() -- get timestamp and digitized waveform of DOM timetick received by test board. (Returns error string, time stamp, and reference to an array of waveform values).
- FPGAUnload() -- unload the current program in the FPGA on the PC/104 system
- FPGAListFiles() -- list currently available FPGA files on the PC/104 system
- ForceDOMCommunicationsEnable() -- force the test board FPGA to reenable communications with the DOM over the RS-232/terminal server connection.
- ReadFPGARegister() -- read a test board FPGA register directly
- WriteFPGARegister() -- write a test board FPGA register directly
- DoTBClockSync() -- cause test board clock counters to reset; get current GPS time string

The Network Interface to syncserver

This section consists of messages to syncserver and their response values. In other words, this is the API which DOMTBControl.pm uses to communicate with syncserver. Messages are all call-and-response, with the call consisting of a command byte with zero or more data bytes, and the response consisting of a status byte with zero or more data bytes. Please note - all multi-byte integer values are sent big-endian (network byte order).

Messages to syncserver

Message: **Echoback**
Byte value: 0x02
Associated DOMTBControl method: echoTest
Number of arguments: 1
Return value: the argument
Code status: implemented

Message: **FPGA Test**
Byte value: 0x09
Associated DOMTBControl method: FPGARegisterTest
Number of arguments: 1 byte (test board ID)
Return value: status byte
Code status: implemented

Message: **Issue Timing Pulse**
Byte value: 0x03
Associated DOMTBControl method: sendTimePulse
Number of arguments: 1 byte (test board ID)
Return values:
Status byte
If status is STATUS_OK, two four-byte integer timestamps of timing pulse.
Code status: implemented

Message: **Read Received Timetick**
Byte value: 0x08
Associated DOMTBControl method: readTBTimetickData
Number of arguments: 1 byte (test board ID)
Return values:
Status byte
If status is STATUS_OK, two four-byte integer timestamps of the received pulse,
as well as 256 bytes containing the digitized waveform.
Code status: implemented

Message: **FPGA load command**
Byte value: 0x04
Associated DOMTBControl method: loadFPGAFile
Number of arguments: 1 byte (test board ID) followed by variable number of bytes (name of jam file, terminated with \n)
Return value: status byte (see syncserver.h).
Code status: implemented

Message: **FPGA list available designs**
Byte value: 0x05
Associated DOMTBControl method: listFPGAFiles
Return message:

Status byte
If status byte is STATUS_OK, rest of the bytes are a string containing file names, separated by commas, terminated by a newline.
Code status: implemented

Message: **FPGA unload command**
Byte value: 0x06
Associated DOMTBControl method: unloadFPGA
Number of arguments: 1 byte (test board ID)
Return message: Command status byte
Code status: not implemented at lowest level (needs API info from Jerry); otherwise implemented

Message: **FPGA status command**
Byte value: 0x07
Associated DOMTBControl method: getFPGAStatus
Number of arguments: 1 byte (test board ID)
Return message:
Command status byte
FPGA status : 1 (loaded) or 0 (unloaded)
Code status: not implemented (needs API info)

Message: **Force DOM Communications Enable**
Byte value: 0x0A
Associated DOMTBControl method: ForceDOMCommunicationsEnable
Number of arguments: 1 byte (test board ID)
Return message: command status byte
Code status: implemented.

Message: **Read FPGA Register**
Byte value: 0x0B
Associated DOMTBControl method: ReadFPGARegister
Arguments: 1 byte test board ID, 1 byte register number
Return message: status byte, register contents byte
Code status: implemented

Message: **Write FPGA Register**
Byte value: 0x0C
Associated DOMTBControl method: WriteFPGARegister
Arguments: 1 byte test board ID, 1 byte register number, 1 byte value to write
Return message: status byte
Code status: implemented

Message: **Turn Power Relay On**
Byte value: 0x0D
Associated DOMTBControl method: PowerRelayOn
Arguments: 1 byte relay number
Return message: status byte
Code status: implemented

Message: **Turn Power Relay Off**
Byte value: 0x0E
Associated DOMTBControl method: PowerRelayOff
Arguments: 1 byte relay number
Return message: status byte
Code status: implemented

Message: **Synchronize Test Board Clocks**
Byte value: 0x0F
Associated DOMTBControl method: DoTBClockSync
Arguments: 1 byte test board ID (any valid test board)
Return message:
    Status byte
    If status byte is STATUS_OK, rest of the bytes are a string containing the time
    as reported by the GPS clock, terminated by a newline
Code status: implemented

Message: **Report Test Board Latched Clock Bytes**
Byte value: 0x10
Associated DOMTBControl method: TBGetLatchedClockBytes
Arguments: none
Return message:
    Status byte
    If status byte is STATUS_OK:
        8 x 2 = 16 bytes, two for each clock
        GPS time string, terminated by a newline
Code status: implemented

Message: **Turn DOMCOM Power On**
Byte value: 0x11
Associated DOMTBControl method: DOMCOMPowerOn
Arguments: 1 byte DOMCOM board ID
Return message: status byte
Code status: implemented

Message: **Turn DOMCOM Power On**
Byte value: 0x12
Associated DOMTBControl method: DOMCOMPowerOn
Arguments: 1 byte DOMCOM board ID
Return message: status byte
Code status: implemented

A response of UNKNOWN_COMMAND (byte value 2) is given if the command byte is not one of the above commands.

Message Status Response Byte Definitions

Status: **STATUS_OK**
Byte value: 0x01
Meaning: It worked.

Status: **UNKOWN_COMMAND**
Byte value: 0x02
Meaning: I don't know what you're talking about

Status: **FUNC_NOT_IMPL**
Byte value: 0x04
Meaning: I know what you're talking about, but I can't do it yet

Status: **FPGA_FILE_NOT_FOUND**
Byte value: 0x03
Meaning: You wanted me to load an FPGA file that I can't find in the usual place on the PC-104.

Status: **FILENAME_TOO_LONG**
Byte value: 0x05
Meaning: The name of the FPGA file you asked me to load is too long.

Status: **JAM_LOAD_FAILED**
Byte value: 0x06
Meaning: The Jamplayer software couldn't load the FPGA file for some reason.

Status: **SHORT_WF_READ**
Byte value: 0x07
Meaning: I didn't get the expected number of bytes from the digitized waveform.

Status: **CANT_READ_WF**
Byte value: 0x08
Meaning: I can't read the waveform data from the COM ADC.

Status: **FPGA_TEST_FAILED**
Byte value: 0x09
Meaning: Read/write test failed

Status: **REGI_OUT_OF_RANGE**
Byte value: 0x0A
Meaning: The FPGA register number was too large

How to Compile, Install and Run the Syncserver Software

It should first be noted that syncserver and gpstime are currently configured to start automatically on skua.lbl.gov (at LBNL) and on dom-tbdaq.spole.gov (at Pole). This is accomplished by rc run level commands (see /etc/init.d and /etc/rc2.d on skua).

To build syncserver, gpstime and to install DOMTBControl.pm:

1. change to domsoft/src/portio
2. cvs update your code
3. make
4. make install (this copies DOMTBControl.pm to /usr/local/dom/lib)

To install syncserver on the PC-104 system:
1. in the portio directory, run update_syncserver. This copies syncserver to the PC-104 system in the directory /tmp and gives it the correct file permissions

To run syncserver:
1. Telnet to the PC-104 system
2. Verify, using "ps ax | grep syncserver" that syncserver isn't running
3. If it is, kill it with "kill -9 <pid>" where <pid> is the process ID of the currently running syncserver.
4. Start syncserver with "/tmp/syncserver".
5. On the client machine (e.g., fireball.spole.gov or rust.lbl.gov) run client_test.pl or domtest. If running domtest, choose "Test Board Functions" followed by "connect to test board/PC-104 system." If it connects successfully, syncserver is running and accepting connections.

Similar instructions obtain for the **gpstime** program.

To install an FPGA design on the PC-104 system:
1. FTP to the PC-104 system with username "dom"

2. cd /tmp
3. put the FPGA design file
4. telnet to the PC-104 system
5. su to root
6. make the root filesystem writable (see General Notes in **Appendix A - Notes on Configuration of the** PC-104/Test Board Systems)
7. move the FPGA design file from /tmp to /home/dom/fpga
8. make the root filesystem read-only

If syncserver and domtest are running, you can test that the FPGA design has made it by selecting "Load a new FPGA file on PC-104 system" from the Test Board Functions menu, and loading the new design.

The DOM Communications Protocol

Section on the communications protocol developed by Chuck and Karl-Heinz. … More information to appear here soon….

The Communications API

The Perl applications for communicating with and controlling the DOMs make use of a set of communications layers. Generally speaking, each layer is implemented by an object (Perl package). Each object makes use of the layer directly below it. For example, the messaging object (DOMMsg) makes use of a packet object (DOMPacket) to send and receive chunks of a message.

All the source code lives in the domsoft/src/domio directory on CVS. The code is written to work on a host PC running either Windows NT or Linux. All platform dependencies are taken care of at the low-level, generic serial layer.

Heirarchy of communications layers/object classes:

<table>
<tr><td colspan="2">Functional Layer (DOMSet)</td></tr>
<tr><td colspan="2">Messaging Layer (DOMMsg)</td></tr>
<tr><td colspan="2">Packet Layer (DOMPacket)</td></tr>
<tr><td>GenericDOMSerial layer - Linux or</td><td>Windows NT</td></tr>
<tr><td colspan="2">Physical Hardware talking to the DOM: Serial ports, terminal servers</td></tr>
</table>

Detailed Descriptions of the Layers of the Communications API

The following sections include technical details on the layers of the communications API, with list of class methods and variables for each object/layer.

The Functional Layer (DOMSet)

The functional layer implements all functions for interacting with a set of DOMs. Uses DOMMsg objects to talk to the DOMs.

Class Methods

Function Name	Arguments (scalar, unless otherwise noted)	Return Values (scalar, unless otherwise noted)	Purpose
new	1) Mode: "tty" or "termserv" 2) a reference to a hash of the form given for R_PORT_HASH (see Fields, below).	1) new object reference 2) error string, if object ref. undefined	Constructor of a DOMSet object
incrementMessageID	the DOM's ID number	Incremented message ID	Increments the ID of DOM
getMessageID	the DOM's ID number	the current message ID	Gets current message ID
SingleDACRead	1) The DOM's ID number 2) the channel of the DAC to read	1) DAC value 2) the error string, if DAC val. Undefined	Reads the last value written to the DAC channel
SingleDACWrite	1) the DOM's ID number 2) the channel to write to 3) the value in DAC units to write	1) Boolean success value 2) Error string, if failure	Write a value to a single DAC channel
SingleADCStream	1) DOM ID 2) Channel ID 3) Time delay in msec between samples	1) Number of samples 2) Error string, if num. samples undefined 3) Array of ADC values	Read an ADC channel multiple times
SingleADCRead	1) DOM ID 2) Channel ID	1) ADC value 2) Error string, if ADC value undefined	Read an ADC value once
SetHVLimit	1) DOM ID 2) Max anode value (DAC units) 3) Max dynode value (DAC units)	Error string; "ok" if success	Set the maximum allowed anode and dynode voltages
GetHVLimit	DOM ID	1) Current max anode value (DAC units) 2) Current max dynode value (DAC units)	Fetches the max. allowed anode and dynode voltages
InitiateEnableHVRequest	DOM ID	1) Challenge value 2) Error string, if challenge value undefined	Enables the setting of HV (requires subsequent call to VerifyEnableHV). USE WITH CARE.
VerifyEnableHV	1) DOM ID 2) Challenge value plus offset	Error string; "ok" if success	Verifies the request to enable HV USE WITH CARE.
InitiateSetHVRequest	1) DOM ID 2) Anode HV (DAC units) 3) Dynode HV (DAC units)	1) Challenge value 2) Error string, if challenge value undefined	Initiates a request to set the high voltage on a DOM (must be successfully enabled using InitiateEnableHVRequest and VerifyEnableHV; requires subsequent call to VerifySetHV). USE WITH CARE.
VerifySetHV	1) DOM ID 2) Anode HV (DAC units) 3) Dynode HV (DAC units) 4) Challenge Value plus offset	Error string; "ok" if success	Verify challenge value and completes the request to set high voltage. USE WITH CARE.
DisableHV	DOM ID	Error string; "ok" if success	Disable high-voltage on the DOM.
TrigDiscriminatorTest	1) DOM ID 2) Direction ("up" or "down") 3) Channel ("spe", "mpe", "spebar", "mpebar")	1) Error string; "ok" if success 2) Array of values from the test	Test reads the output of the discriminator for either single or multi photo electron channel.
TrigDiscriminatorRateTest	1) DOM ID 2) Channel ("spe", "mpe", "spebar", "mpebar")	1) Number of discriminator firings 2) Error string; "ok" if success	Read out the result of an FPGA test of the trigger discriminator. Must set the appropriate SPE or MPE discriminator DAC first.
ComDiscriminatorTest	1) DOM ID 2) Direction ("up" or "down") 3) Channel (0 .. 3)	1) Error string; "ok" if success 2) Array of values from the test	Causes the application to histogram the Com. discriminator output as a function of varying input voltage.
LocalCoinTest	DOM ID	1) Result from test: single byte pattern from FPGA test program, or undef if test fails 2) Error string; "ok" if success	Method causes FPGA test program to test local coincidence circuitry
StartWaveformTest	1) DOM ID 2) Test Type : pick numeric test type from SymbolTrans.pm (currently, ranges from 0 to 30) 3) Poll Delay (msec) 4) Repetition count	Error String: "ok" if success	Begin a test of the ATWD or slow ADC, compiling statistics bin-by-bin for a number of iterations or triggers
StopWaveformTest	DOM ID		NOT YET IMPLEMENTED
GetWaveformTestStatus	DOM ID	1) Error string : "ok" if success 2) Numerical test id (corresponds to "test type" in StartWaveformTest) 3) Current state ("idle", "testing", "done" or "error") 4) Repetition count: how many tests have been performed so far	Check on the status of an already-running test
GetWaveformTestData	DOM ID	1) Error string: "ok" if success 2) Numeric test id	Fetches waveform test data in the default format (E.g. tests such as "TT_ATWD0_CH0_HIST")

		3) Poll Delay (msec) given at start 4) Repetition count 5) Number of Bins 6) Four references to arrays of the length given by "Number of Bins": Minimum values, maximum values, sum of values, sum of squares of values	
GetSpectrumTestData	DOM ID	1) Error string: "ok" if success 2) Numeric test id 3) Poll Delay (msec) given at start 4) Repetition count 5) Number of Bins 6) Reference to array of the length given by "Number of Bins" containing spectrum for the desired channel	Fetches channel spectrum data for tests such as "TT_ATWD0_CH0_INTEGRAL_HIST"
PerformEchoBackTest	1) DOM ID 2) Scalar data to send and receive	Error string : "ok" if success	Sends data down the hole and sees if it comes back intact
GetFPGAStatus	DOM ID	1) Error string: "ok" if success 2) "loaded" or "unloaded" 3) Major version ID 4) Minor version ID 5) Board ID 6) FPGA ID 7) FPGA filename	Gets status of FPGA. If "unloaded", version IDs and filename are undefined
ListFPGAFiles	DOM ID	1) Error string: "ok" if success 2) Array with names of files	Sends a sequence of DSC_FPGA_LIST_FILES messages to generate the list of current FPGA files which are programmed into the flash
LoadFPGAFile	1) DOM ID 2) File name	Error string: "ok" if success	Load the named file into the FPGA
UnloadFPGAFile	DOM ID	Error string: "ok" if success	Unload whatever is in the FPGA
SetTimeTickParams	1) DOM ID 2) Value of Time Tick Byte	Error string: "ok" if success	Set the bits of the FPGA register responsible for configuring time ticks (see Jerry's DOM FPGA API document)
GetTimeTickInfo	DOM ID	1) Error string: "ok" if success 2) lower 4-byte longword of time in DOM when time pulse was captured 3) upper longword of same 4) longword PMT trigger capture time 5) longword time of transmission to surface 6) pretrigger depth read from DOM FPGA 7) value of time tick parameter register at last write via slow control	Get various pieces of information about time ticks sent from and received by DOM

Class Data for DOMSet objects:

R_PORT_HASH : Reference to table of IDs / Port names in the form

{ 1 = "domtest.lbl.gov:80",
2 = "termserv32.spole.gov:81" }
where the first scalar is the OM ID and the second is either IP address:IP port or serial device file name.

R_MSG_HASH : Reference to table of DOMMsg objects

The Messaging Layer (DOMMsg)

The DOMMsg layer implements the DOM Message Passing protocol, using a DOMPacket object.

Fields:
- PORT : DOMPacket object
- MSG_ERROR : String

Methods:
- new ($mode, $port)
- msgError : query last message error type
- type
- subtype
- msgID
- msgStatus
- msgStatusSeverity
- msgErrorLogging
- dataRef
- dataLen
- DOMid
- sync
- send
- recv

The Packet Layer (DOMPacket)

Implements the 8+1 byte DOM packet protocol, using a GenericDOMSerial object.

Fields
- serialObject : GenericDomSerial object
- PACKET_ERROR : Packet error type
- timeoutSeconds

Methods
- new
- packetError : query last packet error type
- outputPkt($pkt) : output packet $pkt over the serialObject
- inputPkt : reads (and modifies) packet from the serialObject; returns undef if troubles ensue.

The Serial Communications Layer (GenericDOMSerial)

This is the lowest level object handles input and output over a serial connection. Functionality includes local serial port access or access to the serial lines of a terminal server, via TCP/IP (currently Linux only). To handle the connection in a system-independent way, the methods of the class are implemented with either Win32DOMSerial (Windows NT code) or LinuxDOMSerial (Linux/Unix code).

Although some functionality has been built in for direct serial I/O (as opposed to the more standard TCP/IP I/O using terminal servers), and for Windows NT, these options are not fully tested and debugged as of this writing.

Fields
- mode ("tty", "termserv" or "network") - the “tty” method is not fully supported
- deviceName (e.g. "COM1", "/dev/ttyS0", "128.104.238.212:80")

Methods
- new
- initSerial
- readNBytes

- readSerialUntilTimeout
- readSerial
- writeSerial

Appendix A - Notes on Configuration of the PC-104/Test Board Systems

As of this writing, there is one working PC/104 system (skua), with two others which have been received and are being configured (petrel and fulmar)[2]. The systems are made by Emac Inc. (www.emacinc.com) and are configured as follows:

PCM-3346 Single board PC-104 computer
64MB 144p EDO SODIMM (RAM)
3.5" HD Floppy Drive (currently on back-order)
3.5" IDE Hard Disk
48MB flash disk
Linux installation on IDE drive and Flash disk
Perl installation on IDE disk
Small SMTP client for sending mail installed on both disks
Realtime extensions installed on both disks
Apache Web server installed on both disks

Emac has fairly extensive hardware and configuration documentation for this system:
The PCM-3364 User's Manual, http://rust.lbl.gov/~jacobsen/docs/dom/3346_fnt.pdf
is a description of the hardware and low-level configuration and
The Servier in a Box Manual, http://rust.lbl.gov/~jacobsen/docs/dom/sib_20_manual.pdf
is a description of the Linux system and software configuration.

The following notes should fill in some of the missing details on what extra steps had to be taken to get the PC-104 systems to run at LBNL and the Pole.

A few general notes:

- Our systems are set up to run off the flash disks, with the hard disks as backups in case problems arise with the flash disks.
- The flash disks are configured read-only to help ensure longevity for the flash filesystems, with directories such as /tmp implemented by RAM disks**. To make the root filesystem writable**, issue the following as root:
  mount -o remount,rw /dev/tffs1 /
  To make it read-only again, substitute "ro" for "rw".

The installation / configuration sequence for the PC-104 systems is as follows:

1) If not already done, cable the PC-104 unit. The cables on petrel and fulmar are labeled, but you may need to refer to the connector diagram on the PCM-3364 User's Manual, pp. 9-10. If connector orientation is ambiguous, match the triangle on the connector with the arrow on the PCB.
   The crucial things to cable here are the power supply, network cable, and, if interactivity is desired, keyboard and VGA monitor.
2) See if the unit powers on ok and passes the self-tests. Examine the BIOS settings before it boots the operating system. Unit is configured to boot from the flash disk (Disk on a Chip or DOC) by default.
3) Make sure Linux boots.

[2] The names refer to Antarctic birds

4) Log in as root, with emac_inc as the password.
5) Quit the configuration menu. Mount / with read/write access: "mount -o remount,rw /dev/tffs1 /". Use the "ae" editor to comment out the line in /root/.profile invoking the SIBconfig.sh script. The script can be invoked by hand later with "/root/SIBconfig.sh". Getting rid of this line avoids the automatic running of the script when root logs in.
6) Change the password with the "passwd" command. Create a "dom" account by editing /etc/passwd (again, use the "ae" editor, and set the UID to 400) and create a home directory /home/dom.
7) Run the configuration script /root/SIBconfig.sh
8) Change the host name (e.g., skua.spole.gov)
9) Change the ethernet settings. Use ethernet device zero. Enter IP address, gateway, netmask, network address, broadcast address.
10) Change the default gateway by selecting "3) Routing tables" followed by "3) Set default gateway". Use the same number here as the network address in the previous step.
11) Change the DNS nameserver addresses (menu item 5) - remove old nameservers and static host address, add new nameservers and static host address (use the same name as the hostname you entered in item 8). Also change the domain and search names by editing /etc/resolv.conf with the "ae" editor. You may have to add the nameserver entries by hand when editing this file.
12) Create FPGA JAM files directory /home/dom/fpga (see instructions, above)
13) Reboot the machine using "shutdown -r now", make sure it boots and can be talked to on the network. Install and run syncserver (see instructions, above).

    Further instructions for configuring the PC-104 system to boot from the hard disk:

14) Cable the hard disk. The IDE cable goes from the PC-104 system to the disk. There should also be a red, yellow and black four-wire power cable from the power supply to the disk.
15) Power up and hit the "DEL" key as soon as you see the first BIOS screen. Select "Advanced BIOS Features." Set the primary boot disk to HDD0 (hard disk), secondary boot disk to HDD1 (flash disk), third boot disk to floppy. Escape to main menu and then select "Save and Exit Setup."
16) At the LILO prompt type Linux, or just hit return, or just wait.
17) Wait for the system to boot.
18) Log in as root, typing default password.
19) Change root password w/ passwd command.
20) If not already present, add a user "dom" with the following command: "useradd -u 400 -s /bin/tcsh dom"
21) Change the password on the dom account with the passwd command.
22) Log in as dom using "su - dom". Create the "fpga" directory with "mkdir fpga". Log out of the dom account with "logout"
23) Edit the following files:
    /etc/HOSTNAME -- change host name
    /etc/hosts -- change host name and IP address. Leave localhost line alone.
    /etc/resolv.conf -- change domain name and add nameserver info.
    /etc/sysconfig/network -- change info for HOSTNAME, DOMAINNAME, GATEWAY.
    /etc/sysconfig/network-scripts/ifcfg-eth0 -- change info for IPADDR, NETMASK, NETWORK (same as GATEWAY, above), BROADCAST.
24) Finally, reboot the machine ("shutdown -r now"), verify that you can log in and use the ping command locally and remotely to test the new network settings.

In this configuration, with both hard disks and flash disks ready to boot, unplugging the power to the hard disk will result in automatically booting from the flash disk. As of this writing, this is the default behaviour.

Configuring Serial Ports for GPS Readout

If you want to read out the GPS clock using the PC-104 system in the DOM Test DAQ or similar system, you'll have to set up the serial ports with the proper settings. This is best done using the configuration script which ships with the PC-104 computers (/root/SIBconfig.sh). Select "Serial Settings"… "Configure

standard serial port". Select the device number (0 for /dev/ttyS0, or COM1; 1 for /dev/ttyS1 or COM2). Use the following parameters:

Line speed: 9600
Data bits: 7
Stop bits: 1
Parity: E
Flow control: N