PCI Device Communication in LabVIEW

This document describes the procedures required to communicate with the PCI device through LabVIEW using C library files. The C library files and the LabVIEW VI's were developed under Red Hat Linux 8.0 (kernel version 2.4.20-13.8) using astropci v1.7 device driver files. Sections I, II, and VII of this document are taken from Voodoo and Device Driver Programmer's Reference Manual by Scoot Streit which were modified for our programming environment.

Joseph Paul
japaul@umich.edu

August 8, 2003

I. Device Driver Installation

In order for the device driver to function properly, you must append the following line to the LILO:

mem=xxxM

where xxx is the amount of RAM you do not want to use for an image buffer. For example, if your computer has 128MB of RAM and you want to have a 28MB image buffer, you must append the following line in LILO:

mem=100M

Example:

1. Become a superuser

2. Edit /boot/grub/grub.conf and append the mem=100M line:

```
title Red Hat Linux (2.4.20-13.8)
root (hd1,5)
kernel /boot/vmlinuz-2.4.20-13.8 ro root=LABEL=/1 hdd=ide-scsi mem=100M
initrd /boot/initrd-2.4.20-13.78.img
```

3. Reboot the computer.

1. Create a directory where you want to store the driver files.

2. Copy the driver tar file to the new directory and unpack it:

    $ tar zxvf astropci_linux.tar.gz

3. Since the driver files are not compiled for the Red Hat 8.0 kernel, you must compile your own driver modules.

    First remove the all existing modules:

    $ rm *.o

    Then compile your own modules by the command:

    $ make

    Become a superuser and then run the installation script:

    $ ./astropci_load

    Please note that astropci_load script must be run after every system reboot, since the driver has no support for loading the driver at system startup.

4. Before the system shutdown, unload the driver:

    $ ./astropci_unload

II. PCI Device Communication in C

There are three device driver entry points (functions) for sending instructions to the PCI device. These functions are open(), close(), and ioctl().

1. To open up a connection to the PCI device, open() function is used. This function requires the system file fnctl.h. This function returns an integer (PCI file descriptor) which is used to establish further communications with the PCI device.

    open

    SYNOPSIS
    #include <fnctl.h>

    int open(const char **device node*, int *mode*)

ARGUMENTS

device node This is one of the nodes /dev/astropci0 or /dev/astropci1. These nodes are created during the driver installation process and correspond to the PCI board 1 and 2 depending on the number of boards you have.

mode This is the constant O_RDWR (Open for reading and writing) supplied by the system file fcntl.h.

DESCRIPTION

Open a connection with the PCI device.

RETURN VALUE

Some positive integer for success (usually 3 in a C program, 17 in LabVIEW), -1 for failure.

2. To close the connection with the PCI device, close() function is used. This must be done before the program is terminated to avoid possible program errors.

close

SYNOPSIS

int close(int *pci_fd*)

ARGUMENTS

pci_fd The integer returned from the open() function.

DESCRIPTION

Close the connection with the PCI device.

RETURN VALUE

0 for success, -1 for failure.

3. To send instructions to the PCI device, ioctl() function is used.

ioctl

SYNOPSIS

int ioctl(int *pci_fd*, int *command*, int **cmd_data*)

ARGUMENTS

pci_fd The integer returned from the open() function.

command This is one of the commands described below.

cmd_data This an array of six integers used to send parameters and receive values associated with the execution of the specified command.

DESCRIPTION
Send commands to the PCI device. Reply value associated with the commands is always stored in the first element of the cmd_data array, *cmd_data[0]*.

RETURN VALUE
0 for success, -1 for failure.

List of commands that were used in developing the LabVIEW program:

ASTROPCI_GET_PROGRESS (0x2)
Get the current pixel count.

ASTROPCI_COMMAND (0x15)
Sends one of the ASCII commands to either the timing board or the utility board. Please refer to the Voodoo manual for a complete list of ASCII commands.

Example codes:

```
/*************************************************************************
*     Name: TestPCI.c
*     Author: Joseph A. Paul
*     Date: 07/15/2003
*
*     Description: The following code will open and close the connection with the PCI
*                  device.
*
*************************************************************************/

#include <stdio.h>
#include <string.h>
#include <fcntl.h>

int main()
{
    int pci_fd;                                  /* PCI file descriptor   */
    char *pci_dev = "/dev/astropci0";            /* PCI device node       */

    /* Open a connection to the PCI and get the PCI file descriptor          */

    pci_fd = open(pci_dev, O_RDWR);

    printf("Opening connection with the PCI device... ");

    if (pci_fd == -1)
        printf("failed\n");
    else {
        printf("OK\n");
        printf("The PCI file descriptor is: pci_fd = %d\n", pci_fd);
    }

    printf("Closing connection with the PCI device... ");
    if (close(pci_fd) == -1)
        printf("failed\n");
    else
        printf("OK\n");

}
```

```
/*************************************************************************
*    Name: memRead.c
*    Author: Joseph A. Paul
*    Date: 07/15/2003
*
*    Description: This program will read the image dimensions on the controller
*                 by sending RDM to the timing board. The column dimensions are at
*                 Y:1 and the row dimensions are at Y:2.
*
*************************************************************************/

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <stropts.h>
#include <unistd.h>
#include <errno.h>
#include <sys/mman.h>

int main()
{

    int pci_fd;
    char *pci_dev = "/dev/astropci0";
    int cmd_data[6];
    int i;

    /* Open connection to the PCI device */

    if (-1 == open(pci_dev, O_RDWR))
        printf("\n\tOpening connection with the PCI device ... failed\n");
    else
        printf("\n\tOpening connection with the PCI device ... OK\n\n");

    /* Read the memory locations using the ioctl command */

    cmd_data[0] = ((0x2 << 8) | 3);                 /* 0x2 for timing board        */
    cmd_data[1] = 0x0052444D;                       /* RDM command                 */
    cmd_data[2] = (0x400000 | 0x0001);              /* 0x400000 for Y memory   */
    cmd_data[3] = -1;
    cmd_data[4] = -1;
    cmd_data[5] = -1;

    for(i=1; i<3; i++) {
```

```
        cmd_data[0] = ((0x2 << 8) | 3);
        cmd_data[2] = (0x400000 | i);

        if (ioctl_return == ioctl(pci_fd, 0x15, &cmd_data))
            printf("\n\tioctl call failed");
        else {
            printf("\tReply @ Address 0x000%d", i);
            printf(": 0x%X \n", cmd_data[0]);
        }

    }

    /* Close connection to the PCI device */

    if (-1 ==  close(pci_fd))
        printf("\n\tClosing connection with the PCI device ... failed\n\n");
    else
        printf("\n\tClosing connection with the PCI device ... OK\n\n");

}
```

Notes on compiling the C code:

To compile a C source file “TestPCI.c” as “TestPCI”, type:

```
$ gcc -o TestPCI TestPCI.c
```

To run the program, type:

```
$ ./TestPCI
```

III. PCI Device Communication in LabVIEW

Since the Linux version of LabVIEW does not support low level system device driver calls, LabVIEW cannot communicate directly with the PCI device. Instead, C functions in C library files are called using the C**all Library Function Node** in LabVIEW to establish a connection with the PCI device. A complete list of C library files currently used by the LabVIEW program is available in the section **IV. C Library Files**.

Call Library Function Node

For example, to open a connection with the PCI device and get the PCI file descriptor in LabVIEW, you must call the function **get_pci_fd()** which is contained in the **pci_setup.so** library file.

Example 1: Open and close connection with the PCI device in LabVIEW

Start LabVIEW and place a **Call Library Function Node** in the diagram. Double click on the **Call Library Function Node** icon. A new window should pop up.

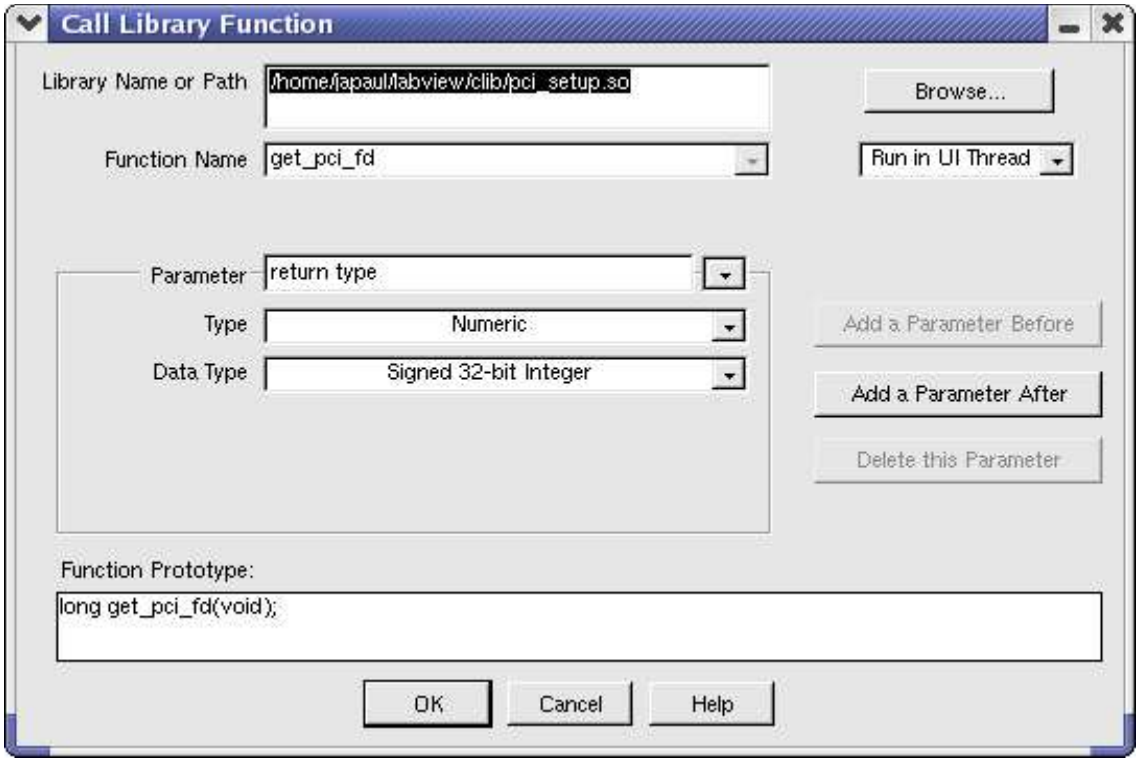


Figure 3.1. Call Library Function Node Window.

First we choose which library file to call. Click on the **Browse** button and choose the **pci_setup.so** file. Now we must specify which function to call from this library file. Type **get_pci_fd** in **Function Name** box.

Now we will set the return type of the function. Make sure that **return type** is selected for the **Parameter**. Since the **get_pci_fd()** function returns an integer, choose **Numeric** for **Type**, and choose **Signed 32-bit Integer** for **Data Type**. Now the **Function Prototype** box should look like:

long get_pci_fd(void);

The idea is to get the **Function Prototype** to match the declaration of the function in the library file. For example, **get_pci_fd()** is declared as:

int get_pci_fd(void);

in the **pci_setup.so** library file.

Now place another **Call Library Function Node** in the diagram, and set it up as shown in Figure 3.2.

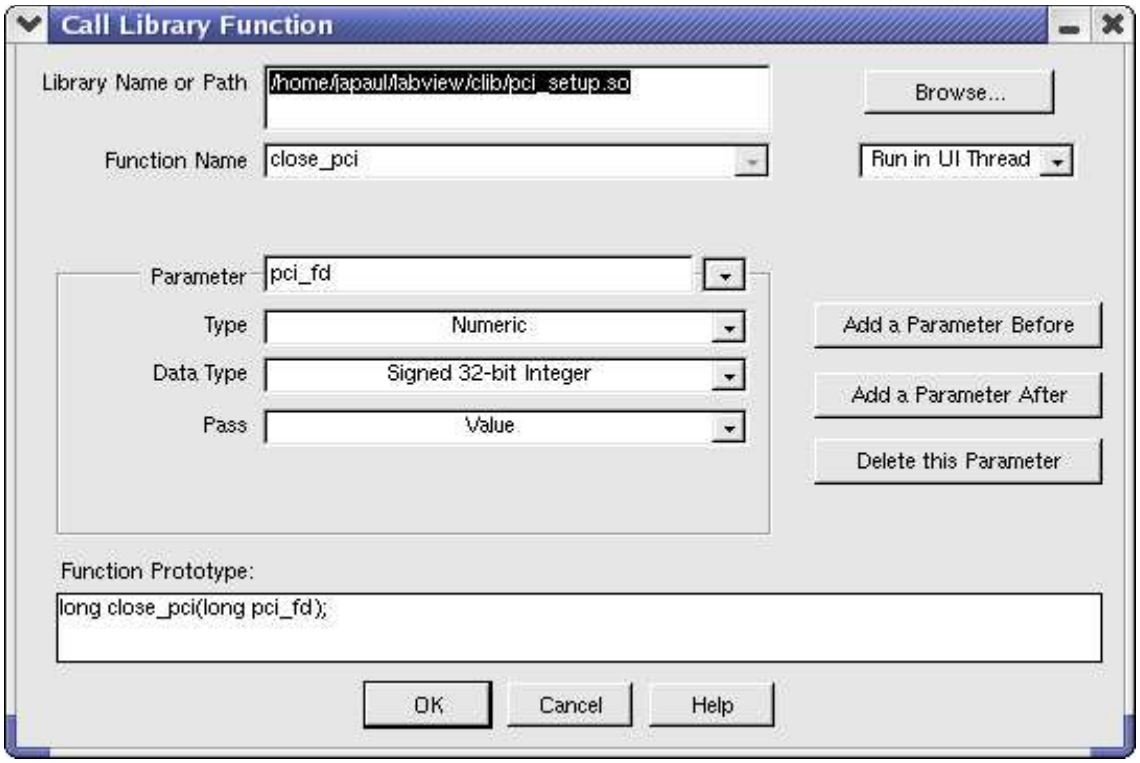


Figure 3.2. Call Library Function Node Window.

The **close_pci** requires that a pci_fd to be passed onto the function. You can set this by

clicking on the **Add a Parameter After** button. You will see **arg1** in the **Parameter** box. Rename it to **pci_fd**. Choose **Numeric** for **Type** and **Signed 32-bit Integer** for **Data Type**.

Now create two numeric indicators. Wire the numeric indicators and the function nodes as shown in Figure 3.3.

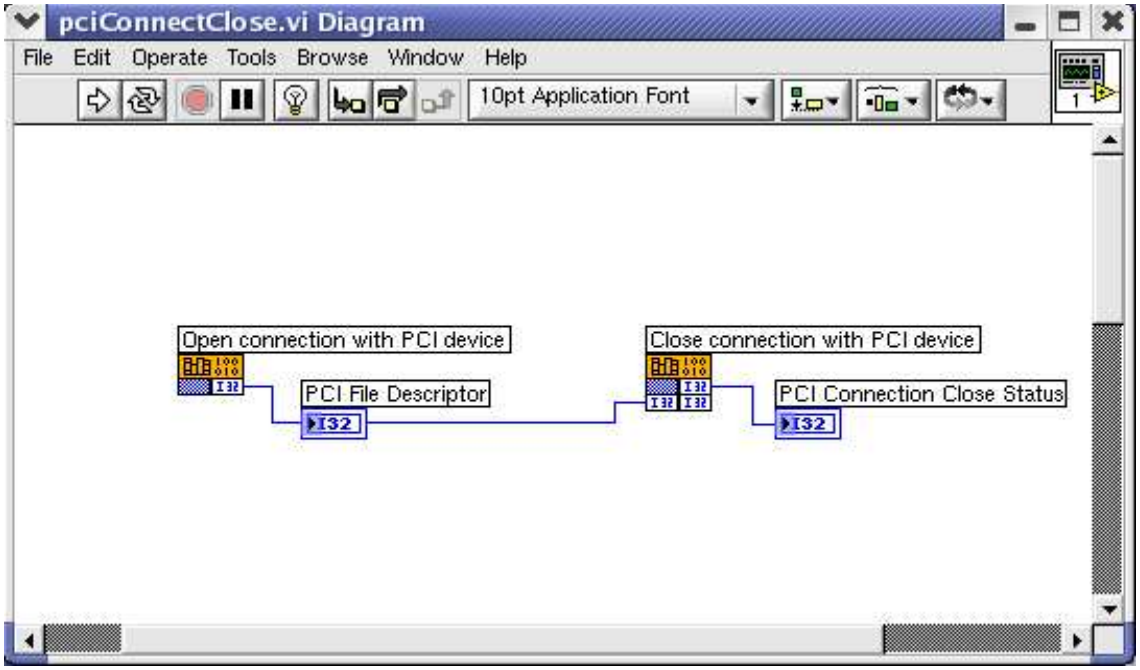


Figure 3.3.

Now run the VI and you should get some positive integer (usually 17) for the PCI File Descriptor and a 0 indicating a success for the PCI Connection Close Status.

Example 2: Reading number of columns in the image

To read the number of image columns, we must read the Y:1 memory (Y:2 is the image rows) on the timing board. To do this we need to send the **RDM** (Read Memory) command to the PCI device using **ioctl().** However, **ioctl()** requires an array to be passed into the function, which requires the use of a pointer.

To avoid complications using pointers in LabVIEW, **DSPCommand.so** library file is used. **DSPCommand.so** requires only integer parameters to be passed into the function. It handles construction of an array and passes its pointer to **ioctl()** inside the functions themselves in the **DSPCommand.so** library file.

For each of the ASCII Commands, there is a fixed number of arguments which is

required to be passed into the ioctl(). For example, RDM requires one argument. The function we want to use is then **doCommand1()**. The code for **doCommand1()** is shown below.

```
int cmd_data[6];
#define ASTROPCI_COMMAND  0x15
#define UNDEFINED -1

int doCommand1(int pci_fd, int board_id, int command, int arg1)
{
    cmd_data[0] = ((board_id << 8) | 3);
    cmd_data[1] = command;
    cmd_data[2] = arg1;
    cmd_data[3] = UNDEFINED;
    cmd_data[4] = UNDEFINED;
    cmd_data[5] = UNDEFINED;

    ioctl(pci_fd, ASTROPCI_COMMAND, &cmd_data);
    return cmd_data[0];
}
```

To call the **doCommand1()** in LabVIEW, open the VI created in **Example 1** and add an another **Call Library Function Node** in the diagram.

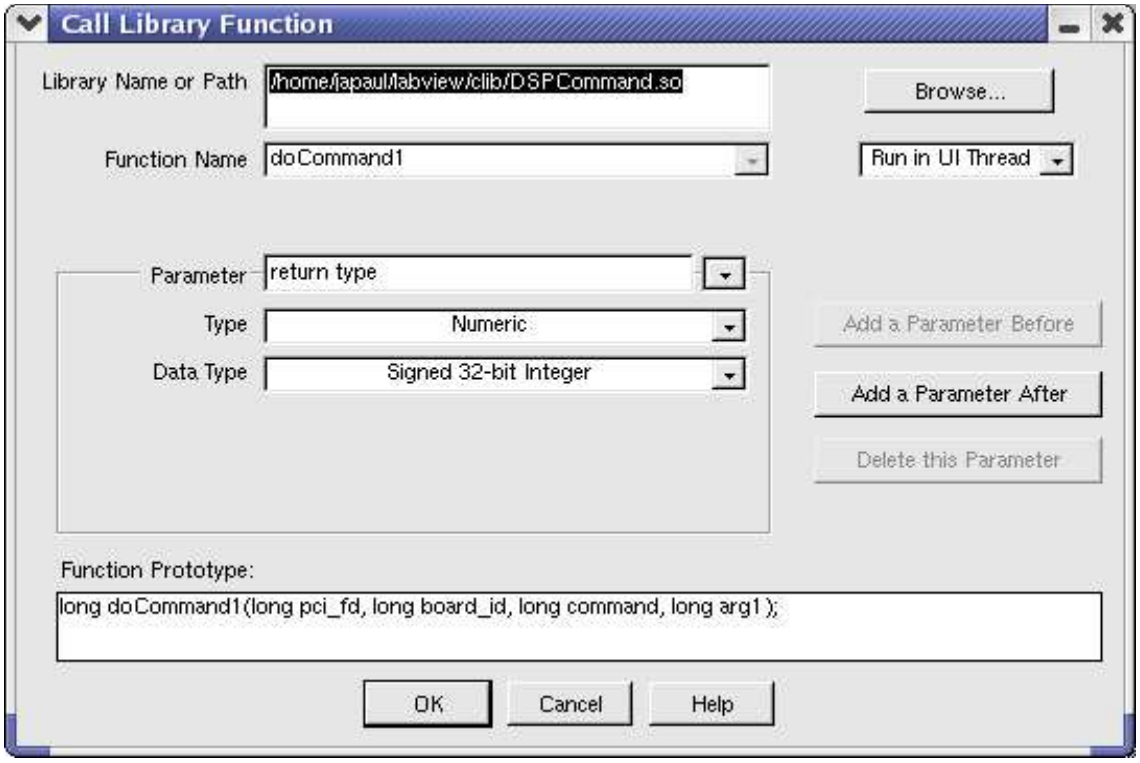


Figure 3.4.

Double click on the **Call Library Function Node** icon and set it up as show in Figure 3.4.

Now we must send the *pci_fd*, *board_id*, *command*, and a*rg1* to this function. For reading a memory location on the timing board, we need to send:

pci_fd	An integer value obtained by calling the **get_pci_fd()** function.
board_id	**0x2** for timing board.
command	**0x0052444D** for RDM
arg1	**0x400001** for Y:1. 0x400000 | 0x0001

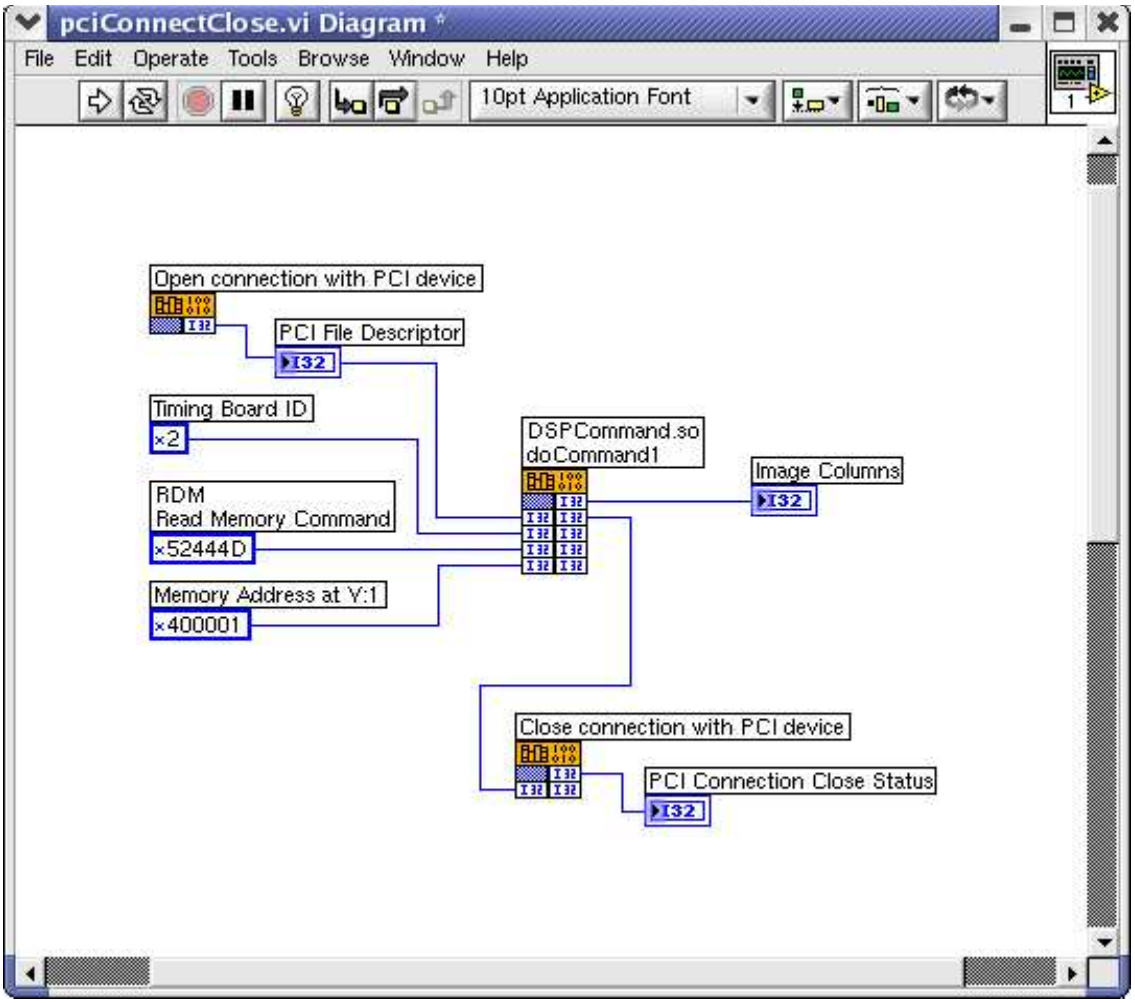


Figure 3.5.

Now create a new numeric indicator to see display the reply (number of image columns) from the **doCommand1()**. Also create constants necessary for inputs into the **doCommand1()** function and wire them as shown in Figure 3.5

Now run the VI and you should see the number of image columns.

IV. C Library Files

Since the Linux version of LabVIEW does not support low level system device driver calls, communication with the PCI device must be done through C functions.

Here is the list of C library files currently used by LabVIEW. Some of these files are modified versions of the C library files from the API Test code.

pci_setup.so

get_pci_fd

SYNOPSIS
int get_pci_fd(void)

ARGUMENTS
none

DESCRIPTION
Open a connection with the PCI device and get the PCI file descriptor.

RETURN VALUE
If successful, PCI file descriptor is returned. (usually 3 in C program, 7 in LabVIEW) Otherwise, -1 is returned.

close_pci

SYNOPSIS
int close_pci(int *pci_fd*)

ARGUMENTS
pci_fd PCI file descriptor

DESCRIPTION
Close with connection with the PCI device.

RETURN VALUE
If successful, 0 is returned. Otherwise, -1 is returned.

LoadDspFile.so

loadFile

SYNOPSIS
int loadFile(int *pci_fd*, const char **filename,* const char **expected_file_type*)

ARGUMENTS
pci_fd	PCI file descriptor
**filename*	Name of the file to load. This is usually tim.lod.
**expected_file_type*	The controller board to load the file into. May be "timing" or "utility".

DESCRIPTION
Used to load the timing file onto the timing board.

RETURN VALUE
If successful, 0 is returned. Otherwise, -1 is returned.

DSPCommand.so

pciCommand

SYNOPSIS
int pciCommand(int *pci_fd*, int *command*)

ARGUMENTS
pci_fd	PCI file descriptor
command	One of the commands such as ASTROPCI_GET_PROGRESS, ASTROPCI_GET_HSTR, etc.

DESCRIPTION
Used to send one of the commands such as ASTROPCI_GET_PROGRESS, ASTROPCI_GET_HSTR, etc. to the PCI device.

RETURN VALUE
If successful, expected data type is returned. For example, if ASTROPCI_GET_PROGRESS command was sent, current pixel count would be returned. Otherwise, -1 is returned.

doCommand

SYNOPSIS
int doCommand(int *pci_fd*, int *board_id*, int *command*)

ARGUMENTS

pci_fd	PCI file descriptor
board_id	0x2 for the timing board, 0x3 for the utility board.
command	One of the ASCII commands.

DESCRIPTION

Used to send an ASCII command which requires no arguments to the PCI device.

RETURN VALUE

If successful, expected data type is returned. Usually this is the reply DON (0x00444F4E). Otherwise, -1 is returned.

doCommand1

SYNOPSIS

int doCommand1(int *pci_fd*, int *board_id*, int *command*, int *arg1*)

ARGUMENTS

pci_fd	PCI file descriptor
board_id	0x2 for the timing board, 0x3 for the utility board.
command	One of the ASCII commands.
arg1	First argument required by the ASCII command.

DESCRIPTION

Used to send an ASCII command which requires one argument to the PCI device.

RETURN VALUE

If successful, expected data type is returned. Usually this is the reply DON (0x00444F4E). Otherwise, -1 is returned.

doCommand2

SYNOPSIS

int doCommand1(int *pci_fd*, int *board_id*, int *command*, int *arg1*, int *arg2*)

ARGUMENTS

pci_fd	PCI file descriptor
board_id	0x2 for the timing board, 0x3 for the utility board.
command	One of the ASCII commands.
arg1	First argument required by the ASCII command.
arg2	Second argument required by the ASCII command.

DESCRIPTION
Used to send an ASCII command which requires two arguments to the PCI device.

RETURN VALUE
If successful, expected data type is returned. Usually this is the reply DON (0x00444F4E). Otherwise, -1 is returned.

doCommand3

SYNOPSIS
int doCommand1(int *pci_fd*, int *board_id*, int *command*, int *arg1,* int *arg2,* int *arg3*)

ARGUMENTS

pci_fd	PCI file descriptor
board_id	0x2 for the timing board, 0x3 for the utility board.
command	One of the ASCII commands.
arg1	First argument required by the ASCII command.
arg2	Second argument required by the ASCII command.
arg3	Third argument required by the ASCII command.

DESCRIPTION
Used to send an ASCII command which requires three arguments to the PCI device.

RETURN VALUE
If successful, expected data type is returned. Usually this is the reply DON (0x00444F4E). Otherwise, -1 is returned.

doCommand4

SYNOPSIS
int doCommand1(int *pci_fd*, int *board_id*, int *command*, int *arg1*, int *arg2*, int *arg3*, int *arg4*)

ARGUMENTS

pci_fd	PCI file descriptor
board_id	0x2 for the timing board, 0x3 for the utility board.
command	One of the ASCII commands.
arg1	First argument required by the ASCII command.
arg2	Second argument required by the ASCII command.
arg3	Third argument required by the ASCII command.

arg4 Fourth argument required by the ASCII command.

DESCRIPTION
Used to send an ASCII command which requires four arguments to the PCI device.

RETURN VALUE
If successful, expected data type is returned. Usually this is the reply DON (0x00444F4E). Otherwise, -1 is returned.

Memory.so

create_memory

SYNOPSIS
int create_memory(int *pci_fd*, int *rows*, int *cols*, int *bufferSize*)

ARGUMENTS
pci_fd PCI file descriptor.
rows Number of row pixels in the image.
cols Number of columns pixels in the image.
BufferSize Size of the image buffer. This is 2200*2200*2 by default.

DESCRIPTION
Create an image buffer.

RETURN VALUE
If successful, integer value of the image buffer location is returned. This value is passed onto other functions and casted as a pointer. Otherwise, -1 is returned.

free_memory

SYNOPSIS
int free_memory(int *mem_fd*, int *bufferSize*)

ARGUMENTS
mem_fd *Integer* value of the image buffer location. This is casted as a pointer in the function.
BufferSize Size of the image buffer. This is 2200*2200*2 by default.

DESCRIPTION

Create an image buffer.

RETURN VALUE
If successful, 0 is returned. Otherwise, -1 is returned.

Deinterlace.so

deinterlace

SYNOPSIS
int deinterlace(int *cols*, int *rows*, int *image_fd*, int *algorithm*)

ARGUMENTS

cols	Number of columns pixels in the image.
rows	Number of row pixels in the image.
image_fd	Integer value of the image buffer location. This is casted as a pointer in the function.
algorithm	Deinterlacing algorithm. 1 split-parallel 2 split-serial 3 quad CCD 4 quad IR 5 CDS quad IR

DESCRIPTION
Deinterlace the image stored in the image buffer.

RETURN VALUE
If successful, 0 is returned. Otherwise, -1 is returned.

FiltsFile.so

writeFitsFile

SYNOPSIS
void writeFitsFile(int *rows* int *cols*, int *exptime*, const char **image_name*, int *mem_fd*)

ARGUMENTS

cols	Number of columns pixels in the image.
rows	Number of row pixels in the image.

Exptime Exposure time in milliseconds.
mem_fd Integer value of the image buffer location. This is casted as a pointer inside the function. Same as the *image_fd*.
**image_name* Name of the fits file to be created.

DESCRIPTION
Write a FITS file of the image stored in the image buffer to the hard drive.

RETURN VALUE
none

Some notes on library files:
There are two types of library files. Static library and dynamically linked library (DLL). A library file contains functions which can be loaded into a program. Loading of DLL is done at run time rather than at compile time (this is static library). This means that when a DLL is modified, the program need not be recompiled. DLL files are called SO (shared object) under Linux.

To compile a shared object file DSPCommand.so using the DSPCommand.c source file, type:

```
$ gcc -shared -o DSPCommand.so DSPCommand.c
```

-shared tells the compiler that DSPCommand.c should be compiled as a shared object, and *-o DSPCommand.so* tells the compiler to create a file named DSPCommand.so.

V. DSP Command VI's

Since the ASCII commands such as **RDM** and **WRM** are used many times in the LabVIEW program, sub-VI's were created. For example, the structure of **RDM .vi** is shown in Figure 5.1.

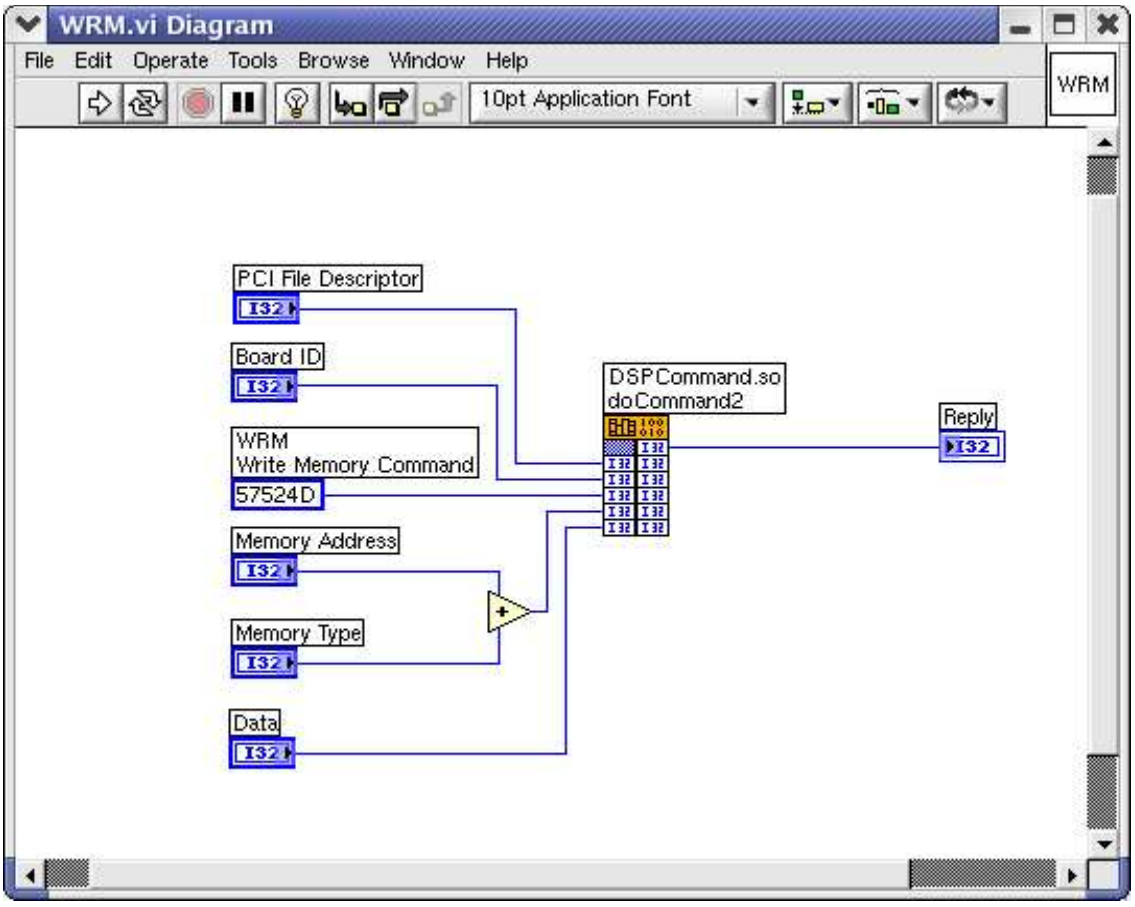


Figure 5.1. WRM sub VI.

So whenever a **WRM** command is required, **WRM.vi** would be called from the main VI and *pci_fd*, *board_id*, *memory address*, *memory type*, and the *data* would be passed onto the **WRM.vi.**

Here is the complete list of the VI currently used for some of the ASCII commands. These files are located in the DSPCommand directory.

WRM (Write Memory)

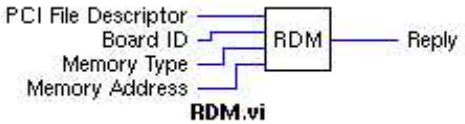
Inputs

PCI File Descriptor	PCI File Descriptor obtained by **open()** function.
Board ID	May be one of the following: 0x2 Timing board 0x3 Utility board
Memory Type	May be one of the following: P (0x100000) Program X (0x200000) Y (0x400000) R (0x800000) ROM
Memory Address	The memory location which you want to read.
Data	Data that you want to write to that memory location.

Output

Reply	0x00444F4E if successful.

RDM (Read Memory)



Inputs

PCI File Descriptor	PCI File Descriptor obtained by **open()** function.
Board ID	May be one of the following: 0x2 Timing board 0x3 Utility board

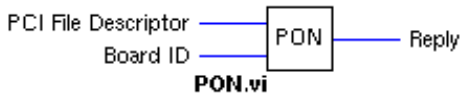
Memory Type — May be one of the following:
P (0x100000) Program
X (0x200000)
Y (0x400000)
R (0x800000) ROM

Memory Address — The memory location which you want to read.

Output

Reply — Value at the specified memory location.

PON (Power On)



Inputs

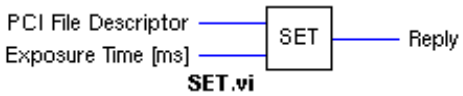
PCI File Descriptor — PCI File Descriptor obtained by **open()** function.

Board ID — May be one of the following:
0x2 Timing board
0x3 Utility board

Output

Reply — 0x00444F4E if successful.

SET (Set Exposure Time)



Inputs

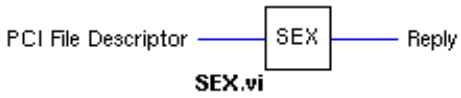
PCI File Descriptor — PCI File Descriptor obtained by **open()** function.

Exposure Time — Exposure time in milliseconds.

Output

Reply 0x00444F4E if successful.

SEX (Start Exposure)



Inputs

PCI File Descriptor PCI File Descriptor obtained by **open()** function.

Output

Reply 0x00444F4E if successful.

VI. Sequence of DSP Commands

The general sequence of DSP commands for capturing an image can be outlined as follows:

1. Open connection with the PCI device and get the pci_fd using **get_pci_fd()** in **pci_setup.so**.

2. Perform the setup commands sequence. (**controllerSetup.vi**)

    1. Set the image dimensions
        WRM at Y:1 and Y:2

    2. Load the timing file to the timing board.
        LoadFile() in **LoadDspFile.so**

    3. Power On
        PON

3. Perform the exposure commands sequence. (**exposeDeinterlaceWrite.vi**)

    1. Create an image buffer to store the image.
        create_memory() in **Memory.so**

2. Set exposure time in milliseconds.
   SET

3. Start Exposure.
   SEX

4. While reading out the image data, get the current pixel count and update the progress bar.
   Send **ASTRO_GET_PROGRESS** using **pciCommand()** in **DSPCommand.so**

5. Deinterlace the image data stored in the image buffer.
   deinterlace() in **Deinterlace.so**

6. Clear the image buffer.
   free_memory() in **Memory.so**

4. Close the device driver connection using **close_pci()** in **pci_setup.so**.

Currently, the entire DSP commands sequence can be performed by first running **controllerSetup.vi** and then running **exposeDeinterlaceWrite.vi**.

VII. DSP and Driver Reply

The full list of DSP and driver replies:

ASCII Reply	**Hex Equivalent**	**Description**
DON	0x00444F4E	Done
ERR	0x00455252	Error
SYR	0x00535952	System Reset
TOUT	0x544F5554	Timeout
NO REPLY	0xFFFFFFFF	Reply Buffer Empty