Data Acquisition Card Driver

For:

General Standards Corporation

16AISS8AO4

16AISS16AO2

Linux Version

Dr. M. C. Nelson

Sensor Realtime, LLC

daqdrv

Rev. 070112

Notice and Disclaimer:

This document and the software described herein are provided with no claim or guarantee of safety, reliability or suitability for any purpose whatsoever.

Always test your software and systems exhaustively under safe, controlled circumstances before deployment.

Bug reports and feature requests are welcome.

Dr. M. C. Nelson
Sensor Realtime, LLC
12/26/2006

1 Introduction

Daqdrv is a device driver for real-time data acquisition. The driver supports event driven and streaming i/o, provides status and control functions, performs DMA transfers to and from user space buffers with asynchronous completion notifications, and also provides register level access to the data acquisition card and its PCI interface controller.

CPU requirements for the daqdrv device driver are minimal, even at maximum data rates.

The user software programming interface to the device driver is conventional and easy to understand. The companion software daqlib, provides an easy-to-use high level programming interface to the driver functions. This document describes the lowest level interface to the device driver. The important functions at this level are the standard unix open(), and ioctl().

2 Installation

The device driver kit is provided as part of the daqlib distribution tar-gz file. After unpacking the tar-gz file the daqdriver subdirectory should contain the following files.

daqdrv.ko	Loadable driver module (kernel 2.6.21).
daqdrvstart.sh	A shell script to load the driver and create the device nodes
Daqdrvstop.sh	A shell script to unload the driver and remove the device nodes
daqdrv.h	A C language include file for IOCTL function codes and data structures.
Makefile daqdrvmain.c daqdrvmain.h gsc16aiss8ao4.c gsc16aiss8ao4.h gsc16aiss8ao4regs.h gsc16aiss26ao2.c gsc16aiss16ao2.h gsc16aiss16ao2regs.h plx_regs.h	Source files to build the data acquisition device driver.

2.1 Building the driver

The device driver shipped with the kit is built for Linux kernel version 2.6.18. You may need to rebuild the driver for your specific kernel. For Linux installations that include the kernel source tree you can cd into the directory and enter the command “make”. If the tree is not present, it

should be obtained from your linux distribution source if possible, or it can be obtained from kernel.org. See the Kernel-Build-HOWTO for instructions for how to set up the kernel tree.

It is strongly recommended that the device driver source codes not be modified.

2.2 Starting and stopping the driver

The driver is stated by invokling the shell script file **daqdrvstart.sh**. The command to invoke the script can be added to the system startup, or it can be invoked by a user with root privileges.

The script is designed to reside in the same directory as the driver. If it is to be located elsewhere, the user can edit the script accordingly.

2.3 Verifying the driver

The driver can be tested by running the sample programs in the samples directory. The program exercise is a good place to start. The program source codes provide further information.

3 Programming Interface

3.1 Device Open

The daqdrv device is opened by the standard unix open() function, as in the following example.

```
fd = open( "/dev/daqdrv0", O_RDWR );
```

The unix open() function returns a file descriptor number to use in the file operations and ioctl( ) calls described below. The daqdrv device nodes are /dev/daqdrv[n], where [n] is 0, 1, 2, or 3.

3.2 Asynchronous Notifications

Asynchronous notifications alert the application program at DMA input done, DMA output done, input buffer overflow and output buffer over/underflow.

The low level mechanism for handling notifications is described here. The daqlib software handles notifications transparently thru functions such as **daqReadStart(*dev*)** and **daqReadWait(*dev*)**.

Notifications are sent by the driver as realtime signals with information about the source and type of event in the siginfo structure (see the **sigaction** man page). The notifications are received by installing a signal handler with the **siginfo** option. The siginfo field **si_fd** identifies which card is sending the signal, and **si_code** identifies the type of notification event. The si_code values supported in the driver are as follows.

POLL_IN	Input DMA completed
POLL_ERR	Input FIFO overflow
POLL_OUT	Output DMA competed
POLL_PRI	Output FIFO underflow

The following source code example shows how to connect a handler and enable asynchronous notifications.

```
#define _GNU_SOURCE
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>

void sighandler( int signum, siginfo_t *info, void *context )
{
    printf( “device %d sent %d\n”, into->si_fd, info->si_code);
}

int setasynch( int fd )
{
    struct sigaction sa = { { 0 } };
    int retv;

    sa.sa_sigaction = &ighandler;
    sa.sa_flags     = SA_SIGINFO | SA_RESTART;

    retv = sigaction( (SIGRTMIN+DAQRTSIG), &sa,  NULL );
    if ( retv == -1 ) return( retv );

    retv = fcntl( b->fd, F_SETOWN, getpid() );
    if ( retv == -1 ) return( retv );

    retv = fcntl( b->fd, F_SETSIG, (SIGRTMIN+DAQRTSIG) );
    if ( retv == -1 ) return( retv );

    retv = fcntl(  b->fd, F_GETFL );
    if ( retv == -1 ) return( retv );

    retv = fcntl(  b->fd, F_SETFL, (retv|FASYNC) );
    return( retv );
}
```

3.3 IOCTL

Low level functions are provided by the driver ioctl() calls as described below.

3.3.1 Get Board ID

The daq card model is identiified by the value stored as the pci subsystem id, retrieved from the device pci bus interface by the driver. Generally, this information is not needed in application programs.

The ioctl code **DAQDRV_IOC_SUBSYSTEMID** retrieves the PCI subsystem id, as shown in the following code excerpt.

```
int getSubsystemID( int fd )
{
    DAQDRV_RegReq regreq = { 0 };

    retv = ioctl( fd, DAQDRV_IOC_SUBSYSTEMID, &regreq );

    return( regreq.value );
}
```

The device model is identified by comparing to the subsytem id. The following macros are defined in gsc16aiss8ao4regs.h and gsc16aiss16ao2regs.h.

```
DAQDRV_SUBSYSTEM_ID_161AISS16AO2
DAQDRV_SUBSYSTEM_ID_161AISS8AO4
```

3.3.2 Get Driver Status

The ioctl code **DAQDRV_IOC_STATUS_READ** retrieves high level status information from the driver. The data includes a userdata field that the user may use to store a pointer to a structure or semaphore (see **`DAQDRV_IOC_SET_USERDATA`** below). The following code shows how to retrieve the status information from the driver.

```
#include <sys/ioctl.h>
#include “daqdrv.h”
DAQDRV_Status  daqdrvstatus;

retv = ioctl( fd, DAQDRV_IOC_STATUS_READ, &daqdrvstatus );
```

The DAQDRV_Status structure type is as follows:

```
typedef struct daqdrv_status {
    unsigned int status;
    unsigned int intstatus;
    unsigned int dma0counter;
    unsigned int dma1counter;
    unsigned int dma0overflow;
    unsigned int dma1overflow;
    unsigned int userdata;
    unsigned int useraddr0;
    unsigned int useraddr1;
    unsigned int usersize0;
    unsigned int usersize1;
} DAQDRV_Status;
```

The value codes for the **status** field are defined in daqdrv.h. The **intstatus** field is loaded from the data acquisition card primary status register (offset 0x30) by the driver interrupt handler. The fields **dma0counter**, **dma1counter**, **dma0overflow** and **dma1overflow** are initialized at the start of a DMA and updated at the completion or error. Dma0 reads data from the analog input, dma1 writes data to the analog output. The userdata field is set by the ioctl function code **DAQDRV_IOC_SET_USERDATA**. The **useraddr0** and **useraddr1** fields contain the user space address of the last buffer completed by DMA0 (analog input) or DMA1 (analog output), and usersize0 and usersize1, contain the size in bytes of the corresponding buffer.

3.3.3 Set User Data

The set user data ioctl code **DAQDRV_IOC_SET_USERDATA** sets the userdata field in an internal structure maintained by the driver for each device. This, in effect, provides a mechanism for associating a user supplied value or pointer with a file descriptor.

The following example stores a user supplied pointer in the **userdata** field.

```
int SetUserData( int fd, void *p )
{
    DAQDRV_Status  daqstatus;
    daqstatus.userdata = (unsigned int) p;
    return( ioctl( fd, DAQDRV_IOC_SET_USERDATA, &daqstatus ) )
}
```

The pointer would typically be retrieved by an event handler using the read status function described above.

3.3.4 Set and Read Data Acquisition Controls

Data acquisition controls can be set and read as a group. The ioctl() function codes are

DAQDRV_IOC_CONTROLS_READ

DAQDRV_IOC_CONTROLS_WRITE

The following example sets the analog input and analog output controls and updates the structure from the actual resulting values. The input FIFO and output FIFO are cleared as a byproduct of the write function.

```
int SetControls( int fd, DAQDRV_Ctl *ctl )
{
    return( ioctl( fd, DAQDRV_IOC_CONTROLS_WRITE, ctl ) );
}
```

The data structure passed to/from the driver is

```
typedef struct daq_ctl {
    DaqBoardConfig    cfg;          // board assembly (read-only)
    DaqAioCtl         ictl;         // analog input controls
    DaqAioCtl         octl;         // analog output controls
    int               trigmaster;   //  internal (SW) trigger (1), extern (0)
} DAQDRV_Ctl;
```

The sub-structure **DaqBoardConfig** contains the assembly configuration for the board.

```
typedef struct daqboardconfig {
    int   version;
    int   inchans;
    int   outchans;
    float clock;
} DaqBoardConfig;
```

The firmnware version is returned in version, the numbers of input and output channels are returned in inchans and outchans, and the master clock speed is returned in clock.

The input and output control settings, ictl and octl, are defined by the structure type **DaqAioCtl**.

```
typedef struct daqaioctl {
    float vfs;
    float clockrate;
    int   ndiv;          // clock divider
    int   clkmaster;     // internal clock (1), external clock (0)
    int   nmode;         //  for input only, single ended, differential, etc.
    int   nrange;        //  input range setting
    int   nchans;        //  number of active channels (0 thru n-1)
    int   nburst;        //  burst length
    int   nthreshold;    //  fifo threshold setting
    int   ndata;         // data requirement ( nchans * nburst )
} DaqAioCtl;
```

The driver does not use the floating point values **vfs** and **clockrate.** They are managed by the daqlib software.

The field **trigmaster,** in DAQDRV_Ctl, is set to 1 for software trigger, and 0 for external trigger.

3.3.5 Hardware Burst Trigger Functions

The following functions initiate bursts in the analog input, the analog output, or both, by setting the burst start bit (or bits) in the control register of the device.

DAQDRV_IOC_TRIGGER_INPUT
DAQDRV_IOC_TRIGGER_OUTPUT
DAQDRV_IOC_TRIGGER_INPUTOUTPUT

Before using a trigger function, the controls should be set with trigmaster=1 (see above). If DMA is to be used to read or write the data, then before using these functions, the buffers should be added to the DMA engine and the DMA engine should be started.

The parameter argument is ignored for these fucntions.

```
DAQDRV_RegReq regreq = { 0 };
retv = ioctl( fd, DAQDRV_IOC_TRIGGER_INPUT, &regreq );
```

The combined input and output start function, for recent firmware versions produces a simultaneous start. For earlier versions (c.f. 16aiss8ao4), offset of 4 to 8 microseconds are observed. For the older devices, the recommended method to simultaneously start the inputs and outputs is to use a digital i/o line to drive the trigger input.

3.3.6 Data Transfers using Programmed IO (PIO)

PIO is a simple interface that is suitable for modest transfers to or from the analog output or input. Performance will depend on the data rate, the size of the transfer and the processor speed and the presence of other processing loads. The interrupt driven DMA interface is described later.
The following source code example is excerpted from the daqlib software.

```
int daqReadPio( int fd, unsigned int *data, int ndata )
{
    int retv;
    DAQDRV_Fifo fiforeq = { 0 };

    fiforeq.data  = data;
    fiforeq.ndata = ndata;

    retv = ioctl( fd, DAQDRV_IOC_FIFO_READPIO, &fiforeq );
    return( (retv ? retv : fiforeq.ndata) );  // -1 if error
}
```

The structure used with the PIO functions is

```
typedef struct daqdrv_fiforeq {
    void *data;
    int ndata;
} DAQDRV_Fifo;
```

The field **data** is a pointer to the user space buffer and **ndata** is the number of words to transfer. The transfer size can be larger than the capacity of the FIFO, the function will run and reschedule as needed until the data has been transferred. The name DAQDRV_Fifo is historical.

3.3.7 Data Transfers using Direct Memory Access (DMA)

The DMA functions provided by the driver, directly map user space buffers. Once started, DMA runs asynchronously, managed by the driver interrupt handler. At completion a signal is delivered to the user process. CPU usage is minimal to negligible. The transfer size is limited only by physical memory and not by the size of the hardware FIFO.
The DMA transfer should usually be started before triggering or starting the data acquisition or analog output. For input, the transfer will begin when the first portion of the requested data is available. For output, the first portion of data is transferred as soon as space is available.

3.3.7.1 Adding Buffers

User space buffers are mapped and locked into memory by the following ioctl functions.

DAQDRV_IOC_ADD_INPUT_BUFFER
DAQDRV_IOC_ADD_OUTPUT_BUFFER

The buffer is specified by the DAQDRV_Fifo structure.

```
typedef struct daqdrv_fiforeq {
    void *data;
    int ndata;
} DAQDRV_Fifo;
```

For example,

```
int AddInputBuffer( int fd, unsigned int *data, int ndata )
{
    DAQDRV_Fifo fiforeq = { 0 };
    fiforeq.data  = data;
    fiforeq.ndata = ndata;
    return( ioctl( fd, DAQDRV_IOC_ADD_INPUT_BUFFER, &fiforeq ) );
}
```

Multiple buffers can be added to each of the input, and/or output DMA engines. If one buffer is added, the engine will operate in single buffer mode completing one transfer for each start. If two or more buffers are added, theDMA engine circles through the buffers as data or space is avaialble, until stopped.

NOTE: Remember to set the end-of-frame bit in the last data word in the output buffer if appropriate for your application.

3.3.7.2 Starting the DMA

DMA is started by the following IOCTL codes.

DAQDRV_IOC_START_INPUT_DMA
DAQDRV_IOC_START_OUTPUT_DMA

The FIFO request data structure (DAQDRV_Fifo) is defined for the function but ignored.

```
int StartInputDma( int fd )
{
    DAQDRV_Fifo fiforeq = { 0 };
    return( ioctl( fd, DAQDRV_IOC_START_INPUT_DMA, &fiforeq ) );
}
```

Once the DMA engine is started, the ioctl() returns to the user process, and the transfer waits or runs as data or space is available, until the transfer is complete. Data acquisition or waveform output is normally started or triggered after starting the DMA engine.

For single buffer transfers, the DMA can be restarted after each completion. For multi buffer transfers (after adding two or more buffers, see above), the transfer continues with the next buffer. A trigger may be needed to restart the data acquisition or waveform output after for each buffer, depending on the control settings.

3.3.7.3 DMA Completion

DMA completion notifications can be connected to a semaphore by using the userdata field in DAQDRV_Status to store the address of a semaphore or that of a structure containing a semaphore.

For example, the signal handler might be constructed as in the following.

```
void sighandler( int signum, siginfo_t *info, void *context )
{
    DAQDRV_Status  daqdrvstatus;
    retv = ioctl(into->si_fd, DAQDRV_IOC_STATUS_READ, &daqdrvstatus );
    if ( !retv ) sem_post( (sem_t *) daqdrvstats.userdata );
}
```

The semaphore is then prepared and connected to the event notification by the following, using the software functions from the examples described earlier in this document.

```
sem_t semdone;

sem_init( &semdone, 0, 0 );
int SetUserData( fd, &semdone);
int setasynch( fd );
```

The user process can then wait for a DMA to complete by simply calling sem_wait().

```
sem_wait( &semdone );
```

3.3.7.4 Stopping the DMA

DMA is canceled by the following IOCTL codes.

```
DAQDRV_IOC_CANCEL_INPUT_DMA
DAQDRV_IOC_CANCEL_OUTPUT_DMA
```

The fifo request data structure (DAQDRV_Fifo) is defined for the function but ignored.

```
int CancelInputDma( int fd )
{
    DAQDRV_Fifo fiforeq = { 0 };
    return( ioctl( fd, DAQDRV_IOC_CANCEL_INPUT_DMA, &fiforeq ) );
}
```

It is recommended that DMA be cancelled after it is complete, although this is not strictly necessary.

3.3.7.5 Releasing Buffers

Buffers are unmapped and released from the DMA engine by the following ioctl function codes.

```
DAQDRV_IOC_RELEASE_INPUT_BUFFERS
DAQDRV_IOC_RELEASE_OUTPUT_BUFFERS
```

The functions release all of the input or output buffers, the DAQDRV_Fifo data is ignored.

```
int ReleaseInputBuffers( int fd )
{
    DAQDRV_Fifo fiforeq = { 0 };
    return( ioctl( fd, DAQDRV_IOC_RELEASE_INPUT_BUFFERS, &fiforeq ) );
}
```

3.3.8 Register access

Register access functions provide read, write, and bitwise AND and OR operations for the data acquisition card registers and for the PLX PCI interface registers. Register access is rarely

needed alongside of the other driver functions, apart from triggering or starting the data acquisition.

The IOCTL function codes for local register access are as follows.

DAQDRV_IOC_REGISTER_READ
DAQDRV_IOC_REGISTER_WRITE
DAQDRV_IOC_REGISTER_OR
DAQDRV_IOC_REGISTER_AND

The IOCTL codes for accessing the PLX registers are

DAQDRV_IOC_PLX_READ
DAQDRV_IOC_PLX_WRITE
DAQDRV_IOC_PLX_OR
DAQDRV_IOC_PLX_AND

The IOCTL register access functions use the DAQDRV_RegReq structure type.

```
typedef struct daqdrv_regreq {
    unsigned int offset;
    unsigned int value;
} DAQDRV_RegReq;
```

The register offset in bytes is loaded into the **offset** field. The register value or bit mask, for register write or bitwise AND or OR functions, is loaded into the **value** field. The value read from the register (for read functions) is returned in the **value** field.

The following example uses register access to set the INPUT S/W TRIGGER bit in the Board Control Register (BCR, offset 0x0);

```
#include <sys/ioctl.h>
#include “daqdrv.h”
unsigned int TriggerInputBurst( int fd )
{
    DAQDRV_RegReq regreq = { 0x0, 1<<11 };
    retv = ioctl( fd, DAQDRV_IOC_REGISTER_OR, &regreq );
    return( (retv ? retv : regreq.value) ); // -1 if error
}
```

4 Epilogoue

This completes the description of the daqdrv device driver. Bug reports and feature requests are welcome.