MEMORANDUM

To: File

From: Rex K. Andrew

Date: December 31, 2001

SUBJ: MIO64F5 Device Driver and C Interface

1 Introduction

We have a data collection program called atocsam written in C by Kurt Metzger of the University of Michigan, compiled under the Watcom C/C++ compiler and executed under DOS on remotely deployed IBM-compatible PCs. It is 16-bit code. We want to upgrade our software and possibly operating system capabilities on these hosts. The main impediment is (re-)implementing the data collection routine on any new system.

For various reasons (maybe someday we’ll put them in here) I want to try implementing data collection under Linux. This is approached by writing a device driver for the DAQ card. A user program (ostensibly the follow-on to atocsam ) then makes calls to the device driver.

For simplicity, I propose a software design with a fair bit of separation between the hardware and the “user code”. See Fig. 1. Originally, I wanted to provide a C++ class DAQstream which will act as an interface between the user code and the DAQ card. This did not work out so instead I implemented a C interface. (Theoretically, changing the DAQ card would then require recoding the device driver, the interface code, but not the main user code.)

For even more simplicity, I have not tried to follow the conventions suggested by the COMEDI project. This is meant to be a simple DAQ device driver that only does one thing — collects multichannel data. ¹

This memo gives a design overview of the device driver and the C interface.

¹It might have been really attractive to build the driver with a COMEDI-compliant interface. The advantage would come if the code is ported to a more modern PC with a more recent A/D card — most recent A/D cards from National Instruments appear to have COMEDI-compliant Linux device drivers available over the Internet. Then the user code and the interface code would not have to change at all! But I was too lazy to try to do this.
Figure 1: DAQ sampler and the interface routines

1.1 References

- Source code for atocsam.
- *Linux Device Drivers, Second edition*, Alessandro Rubini, O’Reilly 2001. (The second edition covers kernels 2.0, 2.2 and into 2.4.)
- The AT-MIO-64E-3 device driver written by Christian Fughe (C.M.Fughe@wtb.tue.nl) and Rob de Kluijver (R.d.Kluyver@ctd.tue.nl). This is for the E-series boards, but shows The Way.
- COMEDI (Linux control and measurement device interface) ([http://stm.lbl.gov/comedi](http://stm.lbl.gov/comedi)). Links to many many device drivers, but not for this board. :-(

1.2 Conventions

I don’t have a lot of coding conventions, but the following Hungarian-style conventions do appear and should be noted.

- File globals are prefixed by “G” to distinguish them from local-scope variables.
- Sometimes pointers in C code are prefixed with p.
Class member names in C++ and structure member names in C often have appended underscores (_). Some people put these in front. Some purists argue not to do this at all because C compilers add underscores, and this can get confusing. I am not confused. I write code too simple to get myself confused.

- C++ Classes tend to start with a capital “C”. Again, vaguely Hungarian.
- Often class data members have postfixed underscores, to distinguish them from local variable.

## 2 Principle of Operation

This device driver is meant to do one thing: collect multichannel data. Typically, data is collected into a buffer, and when the buffer is full, the system is interrupted and some chunk of user code has to do something with the buffered data, usually write it to disk. There is always the question of whether data will be lost if the “system” cannot get back quickly enough from whatever interrupt handling is needed to collecting data into the buffer. In DOS systems, this delay, sometimes called “latency” can be bounded precisely. In Linux/UNIX systems, there is usually no guarantee what the latency can be, because the kernel can decide to just schedule some other task that locks systems resources for seconds/minutes/hours.

Generally, such huge delays do not occur very often in Linux systems. David Horwitt of SIO/UCSD uses a rule-of-thumb for FreeBSD systems that the latency (for a processor of our type) is about “a few milliseconds”. Assume the latency is of order O(10msec). This is about the standard interrupt rate from the system timer. Therefore, a design is sought that will be insensitive to fluctuations in the time-of-arrival of interrupt events on this scale (say, 10 msec) but responsive on a much slower scale, say more than 10 times this, or more than 100 msec. Under the assumption that system pauses of more than 100 msec are extremely rare, this would imply that the chances of missing an interrupt and losing data would be highly unlikely. (Note, this is not a proof of performance.)

The collection process implemented with these considerations proceeds as follows. The DAQ card is programmed to “ping-pong” between two DMA channels, channels A and B. Two large buffers are allocated by the driver and attached to channels A and B. The DMA controller is programmed to transfer one buffer’s worth of data, then signal “terminal count reached”, autoinitialize its counter, and proceed to transfer another buffer’s worth of data. Upon receiving the terminal count signal, the DAQ switches the DMA channel it is using (A to B or vice versa) and sends a system interrupt. As long as the interrupt routine does not disable the DMA or turn off the DAQ, the hardware continues to transfer data from the DAQ to memory under DMA control regardless of what the interrupt routine, other kernel routines, the kernel, or the user application is doing. Thus, the hardware does not care if the data in the two buffers is serviced or not: once collection starts, buffers A and B are filled alternately forever. (There is a small FIFO on the DAQ board that can accommodate 512 samples. This serves as a little buffer between the actual A/D clock rates, which are deterministic, and the response of the DMA controller and lines, which can fluctuate a little depending on bus contention. However, as long as the DMA channels are not shared, the DMA controller is not expected to “go out to lunch” for as long as the kernel itself might. In other words, the “latency” associated with the DMA controller is different than — and much less than – the latency associated with the kernel.)
The interrupt service routine (ISR) in the driver is very simple: it merely sets a flag depending on which buffer was filled, awakens any pending read() task, clears the interrupt condition, and exits. (Well, it does a bit more, but not much. It is intended to execute very quickly.) The ISR also does not care if the data in the buffers is serviced or not— it has no concept of what “servicing” should mean.

The onus for “not losing data” thus falls on the user application. A typical user application for collecting lots of data might have this pseudo-code:

```plaintext
allocate a buffer
loop forever
{
    read data into the buffer
    do something with the buffer
}
```

The read() function is executed once each time through the loop. As long as the loop body executes faster than the time it takes to fill a buffer, the read() function will have to wait (or “sleep” or “block”) until a driver-side buffer becomes full. As mentioned above, when this happens, the ISR is called. DMA collection resumes into the alternate buffer! The ISR unblocks the read() command, which then grabs a bufferfull of data. (Copying data requires bus resources, so this activity must interleave its bus calls with the ongoing DMA operations.)

How fast does the user code have to execute the loop body? In normal ATOC tasks, 42 channels are sampled at 300 MHz. This is a rate of 25.2 kB/sec. Thus, if the DMA buffers are 25.2kB in size, the interrupts will come about one every second. atocsam simply writes the data to disk, so the data copy from driver-side buffer to user-side buffer (fast) plus the write to disk (slow) plus some bookkeeping overhead(small) must take less than 1 sec to avoid losing data with this buffer size. A ballpark estimate of disk throughput might be 200 kB/sec. This would imply that the buffer write to disk would consume 100 msec. Even if the disk write was slow, the seek time awful, the phase of the moon disadvantageous, etc etc etc and the operations took twice as long — i.e., 200 msecs — such code would probably not lose any data.

The maximum size of a data buffer is 131072 bytes. If two such buffers are allocated and used in ping-pong fashion, the time between interrupts will be about 5.2 seconds. Given the ballpark estimate above for disk throughput, this must clearly be a huge amount of time, even for a 80486 PC (!), to respond to a driver interrupt, so the possibility of losing data must be quite small.

Thus, the data collection implementation designed here runs a double-buffered “ping-pong” DMA scheme between the DAQ card, the DMA controller and the system memory. With ATOC-like data rates and maximally-sized buffers, the interrupt rate is of the order of several seconds. Data is collected continuously forever by the hardware— data should never be lost between the DAQ board and the driver buffers. During data collection, the driver mostly sleeps and does not do very much at all, except at each interrupt when it awakens to service the interrupt, transfer data up to the user, and then go back to sleep. The user application has the sole responsibility of reading the device file often enough to avoid losing data, and, for applications like atocsam, there seems to be plenty of time to do simple tasks such as writing data to disk.
This approach has the disadvantage of tying up two DMA channels, whereas atosam needed one channel for D/A operations. This implementation will therefore not work for the dual-mode A/D and D/A capability of atosam.

3 C Interface

The main purpose of this code is to provide a simple and intuitive interface to the application code (and coder) and hide the specific coding details related to the device driver. It would be nice to someday put all of this stuff into a C++ class, so the implementation has been designed to look a bit like a raw implementation of a class. I now think a C++ interface class could be constructed on top of these interface routines.

The interface is implemented using a struct DAQdevice defined in DAQdevice.h as:

```c
struct DAQdevice
{
    char * device_name_[256];
    int driver_version_major_;
    int driver_version_minor_;
    int file_descriptor_;
    struct channel_list chan_list_;  
    float fill_factor_; 
    int total_states_; 
    unsigned long int DMA_buf_size_; /* in bytes */ 
    char * error_msg_[256]; 
    unsigned int driver_status_; 
    unsigned int scan_preload_; 
    unsigned int sample_preload_; 
    unsigned long main_clock_; 
};
```

User code allocates this structure and then passes this structure (via its pointer) to all interface routines. (This is very similar to the this pointer implicitly accompanying all C++ class methods.)

The “interface” consists of a few routines implemented in the module DAQdevice.c and described in section 3.2. It helps to understand the actions of these methods better by detailing the purpose of the struct DAQdevice elements, done in section 3.1.
3.1 Structure Elements

- **device_name** intended to hold the name of the device. This is likely to be the name of a device file such as `/dev/atmio`, which is anticipated to be a character device with read-only privileges.

- **driver_version_major** and **driver_version_minor** implement a decimal version such as “1.1”. Driver code does not support floating point, so versions with decimal parts have to be split into major and minor digits.

- **file_descriptor** is the “raw file descriptor” of the device file. This is actually a low-level integer handle which maps to a file descriptor structure describing the file.

- **chan_list** is a linked list of **channel_node** structures. There is an element in the list for each channel to be collected. Each element contains a word and a pointer. The word bits identify the channel number, the (board) gain to be applied, the polarity of the input, and the input mode. There will be a mirror of this data structure in the device driver. The next pointer points to the next element in the list: the last element has `next` equal to NULL.

- **fill_factor** See the discussion on the computation of the buffer size in the next section.

- **total_states** A convenient copy of the number of MUX states.

- **DMA_buf_size** See the discussion on the computation of the buffer size in the next section.

- **errmsg** The error message associated with the driver return value: see **driver_status**. Error message strings are hardcoded in `ioctl_defs.h` and are not stored in the driver.

- **driver_status** The return code from the driver, updated whenever a call which returns a value is made to the driver.

- **scan_preload** The time between successive conversions on the same channel, conventionally known as the sample rate, is called in the National Instruments documentation the scan interval. This is defined as the number **scan_preload** of periods of the counter clock. We have assumed a hardcoded choice (in the driver) of a 5 MHz clock, so the fundamental period is 0.20 μsec. The value of **scan_preload** is then computed as

\[
\text{scan_preload} = \text{floor}(5 \text{ MHz} / f_s)
\]

where \( f_s \) is the (real) desired (single-channel) sample rate.

- **sample_preload** is a variable currently needed for compatibility with Metzger’s post-processing codes. Metzger’s post-processing codes (primarily the beamformers) relied on a variable **tic_cnt** which I believe was intended to be the number of 0.25 μsec “time-ticks” between consecutive conversions (typically on different channels) of the ADC. Thus, the time in seconds between consecutive conversions (called the “sample time” in the National Instruments manual) is **tic_cnt** × 0.25 μsec. (The origin of this 0.25 μsec value is a bit murky, particularly as his `atocsam` code sets the DAQ sample clock to 5 MHz, which results in a 0.20 μsec-long “time-tick”. Elsewhere in at least one chunk of beamforming code he threw in a hack to correct this, but....). This value is simply a mirror of a value internal to the driver. (The value in the driver is hardcoded, and must be passed up via an `ioctl()` command.)
**main_clock** is the frequency of the main time base in the DAQ. This will usually be 5000000 (for 5 MHz) but may be several other values, all less than 5 MHz. The selection of the primary time base is hardcoded in the driver (to 5 MHz) but there is some provision for altering this to 1 MHz. This flexibility is not very well implemented, though.

### 3.2 Interface Routines

- **void DAQ_Open (struct DAQdevice *pD, int N, float fs, int *MUXmap, int *gainmap)** This is like a C++ constructor. **N** is the number of states: **MUXmap** and **gainmap** must be vectors of this length.

  This routine first checks that the sample rate **fs** and the number of states **N** are positive. **fill_factor** is initialized (with a hardcoded constant) and **device_name** is initialized with the constant string /dev/atmio. The DMA buffer size is calculated. (See below.) Then the device file is opened and the resulting handle stored in **file_descriptor**. Assuming this succeeds, the driver is queried for its version (which is returned with major and minor compressed into a single byte). The driver is next commanded to initialize itself. The value of **scan_preload** is computed and then written down to the driver. The values of **sample_preload** and **main_clock** are retrieved from the driver. Finally, the local channel list linked list is built up from the supplied vectors **MUXmap** and **gainmap**, and the entire linked list structure copied down ("mirrored") to the driver.

  The DMA buffer size is calculated by this routine based on the inputs supplied. It uses a hardcoded fudge factor **fill_factor** specified at compile time (i.e., there is no intention that the user will want to fiddle with this.) The DMA buffers are sized, as described below, to contain exactly an integer number of scans.

  Let **Nstates** be the number of states in the scan sequence. Typically, **Nstates** is 40 for **atocsam**. Each conversion produces a 2 byte word, so the storage for one scan is **2Nstates**. The maximum buffer size of 16-bit DMA is **128KB = 131072 bytes**. Therefore, the maximum number of scans that can be accomodated in one buffer is **floor(131072/2Nstates)** where **floor()** rounds down towards zero.

  We may not always want to use DMA buffers with this maximum size, particularly during testing and debugging. Define a “target size” containing a target number **m_target** of scans where

  \[ m_{\text{target}} = \text{floor}(\alpha_{\text{ff}}131072/2N_{\text{states}}) \]  

  using the fill factor \( \alpha_{\text{ff}} \in [0, 1] \).

  The size of the DMA buffers is then the value

  \[ m_{\text{target}}2N_{\text{states}} \]  

  in bytes. This value is calculated in this class and is passed back up to the user application so the application code can allocate output buffers the same size for output buffering. (See the example code in section 5.)

  This value must always be even.

  The quantity **m_targetNstates** is the total number of (16-bit) samples collected in the buffer. This is computed in the driver by using a 1-bit shift on the buffer size to simulate a divide by 2.
At the completion of this routine, the `struct DAQdevice` should be fully initialized, and the driver should be initialized with all pertinent sample rates and the channel list. The DAQ board is also initialized to a known state. (This is right out of their manual.) The driver will also have requested its IRQ and DMA channels.

Error conditions result in setting `driver_status` and filling in `error_msg`.

- **void DAQ_Close(struct DAQdevice *pD)** This closes the file. The driver releases its IRQ line and DMA channels. No error conditions occur.

- **void DAQ_Setup(struct DAQdevice *pD)** The driver is commanded to set itself up for a collection. This mostly means that driver-side data is written to the DAQ itself. The DMA parameters (buffer size, etc.) are written to the DMA controller.
  Error conditions result in setting `driver_status` and filling in `error_msg`.

- **void DAQ_Arm(struct DAQdevice *pD)** The driver arms the DAQ. This mostly means that certain counters are enabled. DMA is enabled. Can fail if the driver has not been told to set up.
  Error conditions result in setting `driver_status` and filling in `error_msg`.

- **void DAQ_Start(struct DAQdevice *pD)** The board is told to start collection. Can fail if the driver has not been armed.
  Error conditions result in setting `driver_status` and filling in `error_msg`.

- **void DAQ_Stop(struct DAQdevice *pD)** The board is told to stop collection. Cannot fail.

- **void DAQ_Dump(struct DAQdevice *pD)** The driver writes out a bunch of internal variables to the system log. Cannot fail.

- **void DAQ_GetStatus(struct DAQdevice *pD, char *msg, int len)** Gets the status code from the driver, associates this with a human-readable error message, and copies at most `len` bytes of this message into the buffer pointed to by `msg`. Cannot fail.

- **double DAQ_Conversion_Interval(struct DAQdevice *pD)** Returns the time in seconds between successive conversions (the “sample” time). Uses only elements of `struct DAQdevice`. Will barf with an error message if `main_clock` is zero (catches divide-by-zero) and returns 0.0 in this case.

- **double DAQ_Scan_Interval(struct DAQdevice *pD)** Returns the time in seconds between successive scans (the “scan” time, a.k.a. the sample rate in normal parlance.) Uses only elements of `struct DAQdevice`. Will barf with an error message if `main_clock` is zero (catches divide-by-zero) and returns 0.0 in this case.

The following errors are caught, but no fancy error control is implemented at the interface level.

Error situations anticipated:

- On open: sampling rate is 0.0 (or negative!)

- On open: gain or MUX map has no states in it.
On open: device file not found!

On open: device file does not open properly.

During a setup call using \texttt{ioctl}(): funky return value.

During a read: the collection bombs and not enough bytes are read.

During a read: both buffers are ready to read. (Meaning: the user code is not keeping up with the data rate.)

During a read: driver time out. (Meaning: the buffer did not fill after TIMEOUT number of seconds. This is hardcoded to about 20 seconds. A better implementation would adaptively set this according to the buffer size and sample rate.)

4 Linux Device Driver

As of this writing, kernel 2.4.x has been released; not everyone on the Internet is fully enamoured with this version, so I’m proceeding with device driver development based on 2.2.18.

Briefly, the driver sets up two DMA channels (called here A and B) and two buffers (BufferA and BufferB) and commands the DAQ board into a mode where the board uses channel A to move data into BufferA until it is full, then switches automatically to channel B, moving data into BufferB. When that is full, the operation switches back to channel A and BufferA, and so on. The idea is to run continuously. If for some reason the user application cannot keep up, data will be lost. The driver does not care if the user application keeps up or not.

The DAQ will interrupt when a DMA buffer is full. For error catching, we will also enable an interrupt if the on-board FIFO is overrun. The interrupt service routine will have to determine which event has occurred and set the appropriate flags.

The transfer of data out of (and into!) the double buffers can be monitored by the driver, which will set software flags to indicate if the buffers overflow. The interface class will have to watch for this by querying the state of these software flags in between buffer reads.

4.1 Module Operations

4.1.1 \texttt{init}module

Tries to register the driver and reserves the DAQ board I/O space. If this fails, returns -EIO.

4.1.2 \texttt{cleanup}module

If the module is still in use, prints out a message and exits. Otherwise, tries to cleanup by freeing the DMA buffers and the channel list structures (if they have not already been freed). Releases the DAQ board I/O space, unregisters the driver. Cannot fail. Returns nothing.
4.2 File Operations

The “file operations” supported by the driver are described below.

4.2.1 mio64f5_open

Called when the device file is opened. This routine will request the IRQ line and the two DMA channels. These will not be shared because (1) we don’t want problems, and (2) the host PC should not have so many peripherals in action that sharing resources will be a problem.

The device can be opened multiple times. The first open will cause the driver to request the IRQ line and the two DMA channels. Subsequent open calls will skip this logic and return immediately, otherwise the open will fail because the IRQ and DMA lines will already be reserved. (The driver is not meant to be fully re-entrant: only one user at a time is expected. However, concurrent debugging of the driver (by user apps) cannot be accomplished if additional processes cannot open the device.)

Successful execution sets the global error to ENOERROR. Possible errors are EIRQDENIED, EDMAINVALID, EDMABUSY. Possible return values are 0 if there is no error, or -EIO if there has been a problem.

4.2.2 mio64f5_release

Called when the device file is closed. This should disable the DMA operation, if it is still going on, and release the IRQ and DMA channels. The DMA buffers and allocated channel list structures are all freed. Returns 0, sets ENOERROR, cannot fail.

4.2.3 mio64f5_read

When a user process tries to read from the device, execution transfers to this routine, which goes immediately to sleep on a local wait queue. This routine is awakened by the ISR if a data buffer is ready, or if the sleep has timed out. When awoken, this routine looks to see if either DMA buffer is ready (i.e., full.) (If so, the DMA should be writing into the other buffer.) If one of the buffers is ready, the routine transfers the contents from the buffer into the user space buffer, counts the number of bytes written out, and returns that count. If both buffers are ready, then the user is not reading fast enough. This is a “soft overflow” condition. Sets ESOFTOVERFLOW and returns -EIO. If neither buffer is ready, then the task must have awakened after a timeout. Sets ETIMEOUT and returns -EIO.

Some other DAQ-related error conditions are checked: if anything untoward has happened, the appropriate error code is set and the routine returns -EIO.

If there is a page fault, the routine sets EPAGEFAULT and returns -EFAULT. (This is the situation for any driver code that page faults.)
4.2.4 mio64f5_ioctl

Oh boy, here we provide the interface to all the special features of the driver. In general, the ioctl function passes a command and an argument down to the driver. It returns a signed integer return value: I tried to use the convention that 0 means OK and -1 means a problem. If the command is one the driver does not recognize, it returns -ENOTTY. (This is a POSIX issue.) If the driver experiences no problems executing the command, the global error is set to ENOERROR. Available commands are described below.

- **INIT**: This sets the driver into a known state. Global variables are (re-)initialized to default values. The AT-MIO-64F5 is configured into a known state. This is directly out of their manual. atocsam had a different default known state. Any information in the channel list is deleted. Cannot fail, so global error will always be set to ENOERROR.

- **MIRROR_CHANNELS**: This command copies the channel list from the interface class to a local copy within the driver. This command does NOT write the channel list to the DAQ board. Possible global errors are ENOCHANDATA, EPAGEFAULT, EBADLASTNODE.

- **SET_BUFFERSIZE**: This command passes the desired DMA buffer size in bytes down to the driver. The driver does not allocate DMA buffers with this command. Can set EPAGEFAULT.

- **SET_SCAN_PRELOAD**: This command passes the scan preload value down to the driver. The preload value is not written to the STC with this command. Can set EPAGEFAULT.

- **SETUP**: Causes the driver to write all its variable internal data to the AT-MIO-64F5 board and get ready for data acquisition. Actions attempted: (1) DMA buffers A and B are allocated, (2) the DMA controller is programmed for channels A and B, (3) the channel data is written to the DAQ, (4) the scan preload value is written to the STC, and (5) the sample preload value is written to the STC. Possible global errors are ENOBUFFSIZE, ENOBUFFMEM, ENOCHANNELS, ENOSCANLOAD, EBADCLOCK, EIRQDENIED. Sets a global “setup success” flag.

- **ARM**: This call “gets everything ready for the collection.” This chiefly enables the counters. Fails if the global “setup success” flag has not been set, or something is not right. Must be called before a collection can proceed. Set a global “is armed” flag. The driver may not be set up for a variety of reasons, so to avoid endless error checking, the return code in this case is simply EOTHER.

- **START**: If the driver is armed, enables the DAQ interrupts, clears the global buffer ready flags, tells the DAQ to start collecting, sets ENOERROR. If the driver is not armed, sets EOTHER.

- **STOP**: Stops the interrupts from the DAQ. Sets ENOERROR.

- **DUMP**: Ubiquitous debug utility, causes the driver to write the values of internal stuff to the log file.

- **HARD_RESET**: Shuts down any collection going on, resets the DAQ board circuitry, and decrements the module count to zero. This is a brutal command Rubini thought useful should the driver get screwed up and prove difficult to remove from the kernel. Primarily intended for debugging. Recently it did not work right.
• GET_STATUS: Get the driver status code. For more info, see section 4.3.

• GET_SAMPLE_PRELOAD: The sample preload value is generated in the driver: this command allows one to extract it.

• GET_MAIN_CLOCK: The frequency of the main time base is set in the driver: this command allows one to extract it.

• GET_VERSION: Compresses the major and minor values into a single word and returns it. Major is in the upper byte, minor in the lower byte.

4.3 ioctl Error Codes

What can go wrong?

• ENOERROR No error.

• EIRQDENIED IRQ request denied.

• EDMAINVALID Invalid DMA channel.

• EDMABUSY Selected DMA channel busy and not available.

• EFiFOFULL ATMIO64F5 on-board FIFO overflow.

• ESOFfOVERFLOW A buffer that was supposed to be read (and hence no longer ready) was ready to be read. I.e., user application has not kept up with the data rate.

• ENOCHANNELS No channels set up for collection.

• ENOBUFFSIZE No DMA buffer size loaded.

• ENOSCANLOAD No scan interval preload value loaded.

• EPAGEFAULT Typically, a page fault when the driver tried to access user memory.

• ENOBUFFMEM Failure to allocate memory for the DMA buffers.

• ENOCHANNELDATA Attempt to mirror the channel list found nothing in the list!

• EBADLASTNODE Attempt to mirror the channel list found the last node was funky.

• EBADCLOCK The main clock did not initialize to either 5 MHz of 1 MHz.

• ENOCOUNT The sample count was invalid. (Probably zero.)

• ETIMEOUT Timed out waiting for an A/D buffer to fill.

• EOFERRRN ATMIO64F5 overrun. (Conversions too fast, walking on each other.)

• EOTHER Non specific error (see Manuel).
4.4 Interrupt Service Routine mio64f5_interrupt

This routine is really simple and is just a “fast” handler with no bottom half. The ISR queries the DAQ to find out why the board sent an interrupt. If the interrupt is because DMA channel A is ready, the BuffARdy flag is set. If the interrupt is because DMA channel B is ready, the BuffBRdy flag is set. Then any task on the wait queue (hopefully a sleeping read()) is awakened, and then the ISR exits.

5 Application Code Example

Make sure the character device /dev/atmio has been created with major 64!

The following code fragment shows how to program a collection with the C interface.

```c
#include <stdio.h>
#include <time.h>
#include "DAQdevice.h"

/*******************
int main(void)
{
    int total_channels, i;
    int *MUXmap, *gainmap;
    float fs;
    struct DAQdevice *pD;
    short * buffer;
    long buffersize;

    pD = (struct DAQdevice *) malloc( sizeof(struct DAQdevice) );

    total_channels = 42;
    MUXmap = (int *)malloc( total_channels*sizeof(int) );
    gainmap = (int *)malloc( total_channels*sizeof(int) );
    for (i=0; i<total_channels; i++)
    {
        MUXmap[i] = i;
        gainmap[i] = 1;
    }

    fs = 300.0;

    DAQ_Open( pD, total_channels, fs, MUXmap, gainmap );
    buffersize = pD->DMA_buf_size_
```
buffer = (short *)malloc( buffersize );

DAQ_Setup( pD );
DAQ_Arm( pD );
DAQ_Start( pD );

while(1)
{
    read(pD->file_descriptor_, (void *)buffer, buffersize );
    /* do something with buffer: next read will overwrite contents....*/
}

DAQ_Stop( pD );
DAQ_Close( pD );

free(buffer);
free(gainmap);
free(MUXmap);
free(pD);
return 1;
}

Clearly, this program does nothing useful with the data. Moreover, there is no loop termination condition. (Do not try this at home!) More normal code would likely terminate after a set number of buffers had been collected, or on occurrence of a ctrl-C from the keyboard, etc. These features have been omitted for clarity.

Neither DAQ_Status() nor DAQ_Dump() have been used in this example.

6 Benchmarks

This driver was tested on a Gateway 33 MHz 80486 PC with 16 MB RAM running Debian 2.2rev3 Linux and a 2.2.18 kernel. The kernel had been built with a very minimal set of capabilities, much like one would anticipate for an embedded Linux application on a remote 486. 42 channels were collected at 300 Hz. The DMA buffersize was the maximum possible: this turns out to be 131040, which accommodates 1560 complete scans. The user application simply gathered the buffer of data and threw it away, and then did a few lines of bookkeeping, which included writing diagnostic messages to the screen and a log file. The test ran continuously for over 60 hours, collecting over 41,000 buffers of data (more than 5.4 gigasamples, or about 8.5 CDs), without losing any data (that the driver could detect.) Specifically, there were no buffer overflows. The test was terminated by a user abort. This is roughly equal to one-third of a year’s worth of data collection using the typical ATOC transmission schedule.
7 Mysteries

There remain some unexplained things going on here, and so I will list them just in case they turn to be important.

**DAQ Card Signals** It is not altogether clear how the on-board data acquisition circuitry handles the SOURCE and GATE pins for the Am9513 chip for counters 2 and 3. I just programmed the STC per the manual, and it seems to work.

**Variances from atocsam** Metzger used the RTSI to route data going out to the DAC back in to an A/D channel. But in order to get all the timing right, he also deviated from the manual example in his set-up of counters 2 and 5. He implemented some conditional logic in `adsetup()` to follow that through, you should know that in standard (receive) mode, `sw_self_clock` and `sw_divide` are both 0.

**DMA and Interrupts** I’m not sure if DMA continues all the time or pauses during execution of the ISR. I don’t think this is a case of bus-mastering, because the DMA interfaces don’t appear to be that smart, but at the same time, the CPU is not involved in the transfer of data from the DAQ card to memory during DMA. The DAQ card is controlling the transfers, but the DMA controller is the only device that knows the number of samples to transfer (per buffer). I suspect the CPU will execute instructions even in the ISR concurrently with DMA execution as long as there is no bus contention.

8 To Do

- Have the I/O library check with the driver version for compatibility.
- Get and set the timeout in the driver?
- Looks like the `main_clock` could be initialized earlier in `DAQ_Open()` and then used in the calculation of the scan preload value. As it is, scan preload value assumes a clock of 5 MHz. This idea of a variable for the main time base frequency needs to be audited because it is not implemented consistently as either a constant, magic number, or a variable.
- Make sure the driver is not losing synch with the channels (channel 0 data is always in the right spot) and the DMA controller is not dropping any data. (Maybe the same symptom.) Probably need to hook the DAQ up to a real timebase to do this.
- Error control is still not the best. Differentiate between errors generated in the interface and those coming up from the driver.
9 Version Record

6/16/01: Initial version.

7/4/01: Updated document to reflect changes that came up during development to date.

11/25/01: Some careful further design cycles have refined this even more. Added much more meat to the ioctl section so users of the class shouldn’t have to read the code to figure out what each ioctl call will do.

12/28/01: Added stuff about the ISR and read(), scrapped the C++ discussion because it did not work, and added the C interface description, plus benchmarks.

12/31/01: Updated the results of continued benchmark test.