Personal Video Recording performance improvemnt

This release of the linux drivers for the stb043xx controller has improved performance in the demux driver for both the recording and playback of audio video data. There are two devices implemented in the demux driver - the demux device and the PVR device. The demux device provides the capability of filtering audio and video data from the incoming MPEG transport stream for recording to a hard disk or for decoding by the audio and video decoders.. The PVR device provides a DMA interface from memory to the demux for delivering recorded audio and video data to the demux hardware for playback. This document describes the changes made to the demux device and the PVR device.

DEMUX DEVICE

For the recording of audio/video data through the demux driver, a non-buffered filter mode of operation has been added. Prior to this release only buffered mode was available. In buffered mode the data being filtered through the demux driver passes through two additional buffers before being written to the hard disk. The data first is written to a circular memory buffer by the demux hardware. When the BTI threshold is reached the driver is interrupted and copies the data from the circular buffer into a backup buffer. The application then performs a read to the demux driver that copies the data from the backup buffer to an application buffer. The application then writes the data to the hard disk. If the new non-buffered mode of operation is enabled, the data is written to the circular memory buffer by the demux hardware as in buffered mode. However the application is now able to write the data to the hard disk directly from the memory buffer without being copied.

Non-buffered mode is only valid for pes filters (DEMUX_FILTER_PES_SET), transport filters (DEMUX_FILTER_TS_SET) and bucket filters (DEMUX_FILTER_BUCKET_SET). If non-buffered mode is set for a section filter (DEMUX_FILTER_SET) the filter will not operate correctly.

The following functionality has been added to the demux driver to implement non-buffered filters..

mmap function

New demux ioctl's:

- DEMUX_FILTER_SET_FLAGS
- DEMUX_FILTER_GET_QUEUE
- DEMUX_FILTER_SET_READPTR

Asynchronous I/O capability

Filters are set to buffered mode by default when they are opened. A new ioctl has been added to enable non-buffered mode - DEMUX_FILTER_SET_FLAGS. This ioctl is only valid after opening the filter and before the filter is set using DEMUX_FILTER_PES_SET, DEMUX_FILTER_TS_SET or DEMUX_FILTER_BUCKET_SET.

The application gets access to the circular memory buffer via mmap. mmap is only valid after the filter is set and before the filter is started. It will map the demux hardware circular buffer for the specified filter into user address space as a non-cachable memory region. If the MAP_LOCKED flag is set then the buffer will be locked in memory and not swapped out to disk.

After the filter is started the application calls DEMUX_FILTER_GET_QUEUE to be notified when data is available in the queue. DEMUX_FILTER_GET_QUEUE returns the offset into the circular buffer of the current read and write pointers. Using the address returned by mmap and the read/write pointers the application writes the available data to the hard disk. After the write completes DEMUX_FILTER_SET_READPTR is called to update the filter read pointer. If the driver is opened in blocking mode the driver will block if no data is available until data becomes available.

Asynchronous notification is implemented for the demux filters as described in LINUX DEVICE DRIVERS, 2nd Edition (O'REILLY)- CH5: Enhanced Char Driver Operations. When the applicatioin enables asynchronous notificaton the driver will send a SIGIO to the application when data is available and the BTI threshold has been reached. The application signal handler can then use DEMUX_FILTER_GET_QUEUE and write the data to the hard disk.

The following code is an example for setting up a bucket filter in non-buffered mode.

```
// open the demux device
fd = open("/dev/demuxapi0", O_RDWR | O_NONBLOCK);

// set filter to non-buffered mode
ioctl(fd, DEMUX_FILTER_SET_FLAGS, FILTER_FLAG_NONBUFFERED);

// set filter buffer size
ioctl(fd, DEMUX_SET_BUFFER_SIZE, qsize);

// set bucket filter parameters
para.unloader.threshold = bti;
para.unloader.unloader_type = UNLOADER_TYPE_TRANSPORT;

// set the bucket filter
ioctl(fd, DEMUX_FILTER_BUCKET_SET, &para);

// mmap the buffer
pbktq = mmap(NULL, qsize, PROT_WRITE, MAP_SHARED|MAP_LOCKED, fd, 0);
```

To set up for asynchronous I/O.

```
struct sigaction sa;

// set a signal handler
sa.sa_handler = sgh;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
sigaction(SIGIO, &sa, 0);

// set file ownership
fcntl(fd, F_SETOWN, getpid());

// set asynchronous mode
oflags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, oflags | FASYNC);
```

Start the filter.

```
ioctl(fd_bkt, DEMUX_START)
```

Sample signal handler

```
void sgh(int s)
{
    queue_para qpara;

    ioctl(fd, DEMUX_FILTER_GET_QUEUE,&qpara);
    if(qpara.writeptr < qpara.readptr)
    {
        write(fd_data,  pbktq+qpara.readptr, qsize - qpara.readptr);
        write(fd_data,  pbktq, qpara.write);
    }
    else
        write(tsfile, pbktq+qpara.readptr, qpara.writeptr - qpara.readptr);

    ioctl(fd_bkt, DEMUX_FILTER_SET_READPTR, qpara.writeptr);
}
```

New demux device ioctls

ioctl():set flags

Synopsis:

```
int ioctl
{
    int fd,
    int request = DEMUX_FILTER_SET_FLAGS,
    ULONG flags
}
```

Description:

This ioctl call is only valid for filters of type PES, BUCKET and TS. It is not valid for section filters. It is also only valid to set the filter flags after the filter is opened and before the filter is set using DEMUX_FILTER_PES_SET, DEMUX_FILTER_TS_SET or DEMUX_FILTER_BUCKET_SET.

The demux filter set ioctl allocates a circular buffer to receive the filter data. If FILTER_FLAG_NONBUFFERED is set then the filter will be set to non-buffered mode. In non-buffered mode data can not be read from the filter by a read call. The application gains access to the circular buffer using mmap. mmap can be called after the filter is set and before the filter is started.The MAP_LOCKED flag should be set to prevent the buffer from being swapped out. The application then must call DEMUX_FILTER_GET_QUEUE to obtain the read and write pointers for the buffer. The read pointer is an offset into the buffer where the incoming data begins. The write pointer is an offset into the buffer where the next byte of data will be written. On return form DEMUX_FILTER_GET_QUEUE the application processes the data in the buffer and then calls DEMUX_FILTER_SET_READPTR to update the read pointer. Since the buffer is a circular buffer the application must be able to handle buffer wrap conditions.

Parameters:

int fd	File descriptor returned by open().
int request	DEMUX_FILTER_SET_FLAGS
ULONG flags	FILTER_FLAG_NONBUFFERED

Returns:

Returns 0 if succeeded.

Ioctl():filter get queue

Synopsis:

```
int ioctl
{
    int fd,
    int request = DEMUX_FILTER_GET_QUEUE,
    struct  demux_queue_para  *para
}
```

Description:

This ioctl call is only valid for filters of type PES, BUCKET and TS which have been set in non-buffered mode. It is also only valid to call DEMUX_FILTER_GET_QUEUE after the filter has been started.

DEMUX_FILTER_GET_QUEUE returns the current values of the filter circular buffer read and write pointer offsets. The read pointer specifies the offset into the circular buffer where the data begins. The write pointer specifies the offset into the circular buffer where the next byte of data will be written. Using the buffer pointer returned from mmap and the read/write offsets the application processes the data in the buffer and then calls DEMUX_FILTER_SET_READPTR to update the read pointer. The application gains access to the circular buffer using mmap. mmap can be called after the filter is set and before the filter is started.The MAP_LOCKED flag should be set to prevent the buffer from being swapped out. Since the buffer is a circular buffer the application must be able to handle buffer wrap conditions.

If the filter is opened in blocking mode and there is no data in the queue (the read pointer equals the write pointer) the call will block until data is available. If the filter is opened in non-blocking mode and there is no data in the queue the driver sets the return code to -EWOULDBLOCK and returns immediately.

In non-buffered mode it is the responsibility of the application code to process the data fast enough to prevent the demux hardware from wrapping the circular queue and overwriting unprocessed data.

Parameters:

int fd	File descriptor returned by open().
int request	DEMUX_FILTER_GET_QUEUE
struct demux_queue_para *para	Pointer to ioctl parameter structure

```
struct demux_queue_para
{
    unsigned readptr;           //queue read pointer set by driver
    unsigned writeptr;          //queue write pointer set by driver
}
```

Returns:

Returns 0 if succeeded.

Ioctl():set read pointer

Synopsis:

```
int ioctl
{
    int fd,
    int request = DEMUX_FILTER_SET_READPTR,
    ULONG readptr
}
```

Description:

This ioctl call is only valid for filters of type PES, BUCKET and TS which have been set in non-buffered mode. The application calls DEMUX_FILTER_SET_READPTR after processing the data specified by DEMUX_FILTER_GET_QUEUE. The value of the read pointer should be set to the offset of the next byte in the queue to be read by the application. Generally this would be equal to the write pointer value returned by DEMUX_FILTER_GET_QUEUE.

Parameters:

int fd	File descriptor returned by open().
int request	DEMUX_FILTER_SET_READPTR
ULONG readptr	New read pointer offset

Returns:

Returns 0 if succeeded.

PVR DEVICE

A non-buffered mode of data transfer has also been implemented for the PVR device. The PVR device allows transport stream data to be DMA'd from a memory buffer into the demux hardware for audio/video playback. In buffered mode an application reads the transport stream data from the hard disk into an application buffer. It then writes the data from the buffer to the PVR device. The PVR device write method copies the data from the application buffer into an internal buffer and then sets up the PVR DMA operation. In the non-buffered mode the application allocates a buffer from the PVR device using a new ioctl. This buffer will be non cachable and locked to prevent the buffer from being swapped to hard disk. The application then reads the transport stream data into the allocated buffer and calls the PVR driver (another new ioctl) to DMA the data directly from this buffer into the demux hardware. Up to 16 buffers may be allocated by the application.

The following new ioctl functions have been added to the PVR driver to implement non-buffered mode..

PVR_ALLOCATE_BUFFER

PVR_FREE_BUFFER

PVR_WRITE_BUFFER

New PVR device ioctls

Ioctl():PVR allocate buffer

Synopsis:

```
int ioctl
{
    int fd,
    int request = PVR_ALLOCATE_BUFFER,
    struct pvr_allocate_buffer_t  *param
}
```

Description:

PVR_ALLOCATE_BUFFER allocates a locked non-cached buffer of the specified size and returns a pointer to the buffer. This buffer can be used for DMA'ing MPEG transport data to the demux hardware using PVR_WRITE_BUFFER.

Parameters:

int fd	File descriptor returned by open().
int request	PVR_ALLOCATE_BUFFER
struct pvr_allocate_buffer_t *param	

```
struct pvr_allocate_buffer_t
{
    void * addr;                //buffer pointer returned by driver
    unsigned long len;          //buffer length set by application
}
```

Returns:

Returns 0 if succeeded.

Ioctl():PVR write buffer

Synopsis:

```
int ioctl
{
    int fd,
    int request = PVR_WRITE_BUFFER,
    struct pvr_write_buffer_t  *param
}
```

Description:

PVR_WRITE_BUFFER will DMA the contents of the specified buffer to the demux hardware. The buffer address must be equal to a buffer address previously allocated using PVR_ALLOCATE_BUFFER. The len must be a multiple of 4 bytes if the PVR mode is set to word mode or a multiple of 32 bytes if the PVR mode is set to line mode. This call will not return until the DMA operation has completed.

Parameters:

int fd	File descriptor returned by open().
int request	PVR_WRITE_BUFFER
struct pvr_write_buffer_t *param	

```
struct pvr_write_buffer_t
{
    void * addr;                //buffer pointerr
    unsigned long len;          //data length
}
```

Returns:

Returns len if succeeded else returns error code.

Ioctl():PVR free buffer

Synopsis:

```
int ioctl
{
    int fd,
    int request = PVR_FREE_BUFFER,
    unsigned long addr
}
```

Description:

PVR_FREE_BUFFER frees a PVR buffer. *addr* must be equal to a buffer address previously allocated using PVR_ALLOCATE_BUFFER.

Parameters:

int fd	File descriptor returned by open().
int request	PVR_FREE_BUFFER
unsigned long	addr

Returns:

Returns 0 if succeeded.

Sample Applications

Several sample applications have been added to the stb043xx linux source tree to test and demonstrate the new features. The sample applications can be found in the *demux/pvrtest* subdirectory. There are four programs: test_rec_bkt_nb, test_rec_pes_nb, test_rec_ts_nb and test_pvr_nb. These programs use the non-buffered features introduced in this release.

test_rec_bkt_nb	Demonstrates using a bucket filter to record audio and video transport packets into a single program transport stream onto the hard disk.
test_rec_pes_nb	Demonstrates using PES filters to record audio and video PES data into separate files onto the hard disk.
test_rec_ts_nb	Demonstrates using a TS filter to record an entire transport stream onto the hard disk.
test_pvr_nb	Demonstrates playing back audio and video data from a transport stream that has previously been recorded to the hard disk.