```
<title>DVB Demux Device</title>

<para>The DVB demux device controls the filters of the DVB hardware/software. It
can be
accessed through <emphasis role="tt">/dev/adapter0/demux0</emphasis>. Data types
and and ioctl definitions can be
accessed by including <emphasis role="tt">linux/dvb/dmx.h</emphasis> in your
application.
</para>
<section id="dmx_types">
<title>Demux Data Types</title>

<section id="dmx_output_t">
<title>dmx_output_t</title>
 <programlisting>
 typedef enum
 {
         DMX_OUT_DECODER,
         DMX_OUT_TAP,
         DMX_OUT_TS_TAP
 } dmx_output_t;
</programlisting>
<para><emphasis role="tt">DMX_OUT_TAP</emphasis> delivers the stream output to
the demux device on which the ioctl is
called.
</para>
<para><emphasis role="tt">DMX_OUT_TS_TAP</emphasis> routes output to the logical
DVR device <emphasis role="tt">/dev/dvb/adapter0/dvr0</emphasis>,
which delivers a TS multiplexed from all filters for which <emphasis
role="tt">DMX_OUT_TS_TAP</emphasis> was
specified.
</para>
</section>

<section id="dmx_input_t">
<title>dmx_input_t</title>
 <programlisting>
 typedef enum
 {
         DMX_IN_FRONTEND,
         DMX_IN_DVR
 } dmx_input_t;
</programlisting>
</section>

<section id="dmx_pes_type_t">
<title>dmx_pes_type_t</title>
 <programlisting>
 typedef enum
 {
         DMX_PES_AUDIO,
         DMX_PES_VIDEO,
         DMX_PES_TELETEXT,
         DMX_PES_SUBTITLE,
         DMX_PES_PCR,
         DMX_PES_OTHER
```

```
 } dmx_pes_type_t;
</programlisting>
</section>

<section id="dmx_event_t">
<title>dmx_event_t</title>
 <programlisting>
 typedef enum
 {
        DMX_SCRAMBLING_EV,
        DMX_FRONTEND_EV
 } dmx_event_t;
</programlisting>
</section>

<section id="dmx_scrambling_status_t">
<title>dmx_scrambling_status_t</title>
 <programlisting>
 typedef enum
 {
        DMX_SCRAMBLING_OFF,
        DMX_SCRAMBLING_ON
 } dmx_scrambling_status_t;
</programlisting>
</section>

<section id="dmx_filter">
<title>struct dmx_filter</title>
 <programlisting>
 typedef struct dmx_filter
 {
        uint8_t         filter[DMX_FILTER_SIZE];
        uint8_t         mask[DMX_FILTER_SIZE];
 } dmx_filter_t;
</programlisting>
</section>

<section id="dmx_sct_filter_params">
<title>struct dmx_sct_filter_params</title>
 <programlisting>
 struct dmx_sct_filter_params
 {
        uint16_t            pid;
        dmx_filter_t        filter;
        uint32_t            timeout;
        uint32_t            flags;
 #define DMX_CHECK_CRC       1
 #define DMX_ONESHOT         2
 #define DMX_IMMEDIATE_START 4
 };
</programlisting>
</section>

<section id="dmx_pes_filter_params">
<title>struct dmx_pes_filter_params</title>
 <programlisting>
```

```
 struct dmx_pes_filter_params
 {
        uint16_t            pid;
        dmx_input_t         input;
        dmx_output_t        output;
        dmx_pes_type_t      pes_type;
        uint32_t            flags;
 };
</programlisting>
</section>

<section id="dmx_event">
<title>struct dmx_event</title>
 <programlisting>
 struct dmx_event
 {
        dmx_event_t          event;
        time_t               timeStamp;
        union
        {
                dmx_scrambling_status_t scrambling;
        } u;
 };
</programlisting>
</section>

<section id="dmx_stc">
<title>struct dmx_stc</title>
 <programlisting>
 struct dmx_stc {
        unsigned int num;       /&#x22C6; input : which STC? 0..N &#x22C6;/
        unsigned int base;      /&#x22C6; output: divisor for stc to get 90 kHz
clock &#x22C6;/
        uint64_t stc;           /&#x22C6; output: stc in 'base'&#x22C6;90 kHz
units &#x22C6;/
 };
</programlisting>
 </section>

</section>

<section id="dmx_fcalls">
<title>Demux Function Calls</title>

<section id="dmx_fopen">
<title>open()</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This system call, used with a device name of /dev/dvb/adapter0/demux0,
 allocates a new filter and returns a handle which can be used for subsequent
 control of that filter. This call has to be made for each filter to be used,
i.e. every
 returned file descriptor is a reference to a single filter.
/dev/dvb/adapter0/dvr0
```

 is a logical device to be used for retrieving Transport Streams for digital
 video recording. When reading from this device a transport stream containing
 the packets from all PES filters set in the corresponding demux device
 (/dev/dvb/adapter0/demux0) having the output set to DMX_OUT_TS_TAP. A
 recorded Transport Stream is replayed by writing to this device. </para>
<para>The significance of blocking or non-blocking mode is described in the
 documentation for functions where there is a difference. It does not affect the
 semantics of the open() call itself. A device opened in blocking mode can later
 be put into non-blocking mode (and vice versa) using the F_SETFL command
 of the fcntl system call.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int open(const char ⋆deviceName, int flags);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>const char
 *deviceName</para>
</entry><entry
 align="char">
<para>Name of demux device.</para>
</entry>
 </row><row><entry
 align="char">
<para>int flags</para>
</entry><entry
 align="char">
<para>A bit-wise OR of the following flags:</para>
</entry>
 </row><row><entry
 align="char">
</entry><entry
 align="char">
<para>O_RDWR read/write access</para>
</entry>
 </row><row><entry
 align="char">
</entry><entry
 align="char">
<para>O_NONBLOCK open in non-blocking mode</para>
</entry>
 </row><row><entry
 align="char">
</entry><entry
 align="char">
<para>(blocking mode is the default)</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS

```
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>ENODEV</para>
</entry><entry
 align="char">
<para>Device driver not loaded/available.</para>
</entry>
 </row><row><entry
 align="char">
<para>EINVAL</para>
</entry><entry
 align="char">
<para>Invalid argument.</para>
</entry>
 </row><row><entry
 align="char">
<para>EMFILE</para>
</entry><entry
 align="char">
<para>&#8220;Too many open files&#8221;, i.e. no more filters available.</para>
</entry>
 </row><row><entry
 align="char">
<para>ENOMEM</para>
</entry><entry
 align="char">
<para>The driver failed to allocate enough memory.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_fclose">
<title>close()</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This system call deactivates and deallocates a filter that was previously
 allocated via the open() call.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int close(int fd);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
```

<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid open file descriptor.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_fread">
<title>read()</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This system call returns filtered data, which might be section or PES
data. The
 filtered data is transferred from the driver’s internal circular buffer
to buf. The
 maximum amount of data to be transferred is implied by count.</para>
</entry>
 </row><row><entry
 align="char">
<para>When returning section data the driver always tries to return a complete
single
 section (even though buf would provide buffer space for more data). If the size
 of the buffer is smaller than the section as much as possible will be returned,
 and the remaining data will be provided in subsequent calls.</para>
</entry>
 </row><row><entry
 align="char">
<para>The size of the internal buffer is 2 * 4096 bytes (the size of two maximum
 sized sections) by default. The size of this buffer may be changed by using the
 DMX_SET_BUFFER_SIZE function. If the buffer is not large enough, or if
 the read operations are not performed fast enough, this may result in a buffer
 overflow error. In this case EOVERFLOW will be returned, and the circular
 buffer will be emptied. This call is blocking if there is no data to return,
i.e. the
 process will be put to sleep waiting for data, unless the O_NONBLOCK flag
 is specified.</para>
</entry>
 </row><row><entry
 align="char">
<para>Note that in order to be able to read, the filtering process has to be
started
 by defining either a section or a PES filter by means of the ioctl functions,
 and then starting the filtering process via the DMX_START ioctl function
 or by setting the DMX_IMMEDIATE_START flag. If the reading is done
 from a logical DVR demux device, the data will constitute a Transport Stream
 including the packets from all PES filters in the corresponding demux device

 /dev/dvb/adapter0/demux0 having the output set to DMX_OUT_TS_TAP.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>size_t read(int fd, void ⋆buf, size_t count);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>void *buf</para>
</entry><entry
 align="char">
<para>Pointer to the buffer to be used for returned filtered data.</para>
</entry>
 </row><row><entry
 align="char">
<para>size_t count</para>
</entry><entry
 align="char">
<para>Size of buf.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EWOULDBLOCK</para>
</entry><entry
 align="char">
<para>No data to return and O_NONBLOCK was specified.</para>
</entry>
 </row><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid open file descriptor.</para>
</entry>
 </row><row><entry
 align="char">
<para>ECRC</para>
</entry><entry
 align="char">
<para>Last section had a CRC error - no data returned. The

```
 buffer is flushed.</para>
</entry>
 </row><row><entry
 align="char">
<para>EOVERFLOW</para>
</entry><entry
 align="char">
</entry>
 </row><row><entry
 align="char">
</entry><entry
 align="char">
<para>The filtered data was not read from the buffer in due
 time, resulting in non-read data being lost. The buffer is
 flushed.</para>
</entry>
 </row><row><entry
 align="char">
<para>ETIMEDOUT</para>
</entry><entry
 align="char">
<para>The section was not loaded within the stated timeout
 period. See ioctl DMX_SET_FILTER for how to set a
 timeout.</para>
</entry>
 </row><row><entry
 align="char">
<para>EFAULT</para>
</entry><entry
 align="char">
<para>The driver failed to write to the callers buffer due to an
 invalid *buf pointer.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_fwrite">
<title>write()</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This system call is only provided by the logical device
/dev/dvb/adapter0/dvr0,
 associated with the physical demux device that provides the actual DVR
 functionality. It is used for replay of a digitally recorded Transport Stream.
 Matching filters have to be defined in the corresponding physical demux
 device, /dev/dvb/adapter0/demux0. The amount of data to be transferred is
 implied by count.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>ssize_t write(int fd, const void &amp;#x22C6;buf, size_t
```

```
 count);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>void *buf</para>
</entry><entry
 align="char">
<para>Pointer to the buffer containing the Transport Stream.</para>
</entry>
 </row><row><entry
 align="char">
<para>size_t count</para>
</entry><entry
 align="char">
<para>Size of buf.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EWOULDBLOCK</para>
</entry><entry
 align="char">
<para>No data was written. This
 might happen if O_NONBLOCK was specified and there
 is no more buffer space available (if O_NONBLOCK is
 not specified the function will block until buffer space is
 available).</para>
</entry>
 </row><row><entry
 align="char">
<para>EBUSY</para>
</entry><entry
 align="char">
<para>This error code indicates that there are conflicting
 requests. The corresponding demux device is setup to
 receive data from the front- end. Make sure that these
 filters are stopped and that the filters with input set to
 DMX_IN_DVR are started.</para>
</entry>
 </row><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
```

```
<para>fd is not a valid open file descriptor.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_start">
<title>DMX_START</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call is used to start the actual filtering operation defined
via the ioctl
 calls DMX_SET_FILTER or DMX_SET_PES_FILTER.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request = DMX_START);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
 align="char">
<para>Equals DMX_START for this command.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
 </row><row><entry
 align="char">
<para>EINVAL</para>
</entry><entry
 align="char">
<para>Invalid argument, i.e. no filtering parameters provided via
 the DMX_SET_FILTER or DMX_SET_PES_FILTER
```

```
 functions.</para>
</entry>
 </row><row><entry
 align="char">
<para>EBUSY</para>
</entry><entry
 align="char">
<para>This error code indicates that there are conflicting
 requests. There are active filters filtering data from
 another input source. Make sure that these filters are
 stopped before starting this filter.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_stop">
<title>DMX_STOP</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call is used to stop the actual filtering operation defined via
the
 ioctl calls DMX_SET_FILTER or DMX_SET_PES_FILTER and started via
 the DMX_START command.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request = DMX_STOP);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
 align="char">
<para>Equals DMX_STOP for this command.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
```

```
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_set_filter">
<title>DMX_SET_FILTER</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call sets up a filter according to the filter and mask
parameters
 provided. A timeout may be defined stating number of seconds to wait for a
 section to be loaded. A value of 0 means that no timeout should be applied.
 Finally there is a flag field where it is possible to state whether a section
should
 be CRC-checked, whether the filter should be a &#8221;one-shot&#8221; filter,
i.e. if the
 filtering operation should be stopped after the first section is received, and
 whether the filtering operation should be started immediately (without waiting
 for a DMX_START ioctl call). If a filter was previously set-up, this filter
will
 be canceled, and the receive buffer will be flushed.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request = DMX_SET_FILTER,
 struct dmx_sct_filter_params &#x22C6;params);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
 align="char">
<para>Equals DMX_SET_FILTER for this command.</para>
</entry>
 </row><row><entry
 align="char">
<para>struct
 dmx_sct_filter_params
```

```
 *params</para>
</entry><entry
 align="char">
<para>Pointer to structure containing filter parameters.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
 </row><row><entry
 align="char">
<para>EINVAL</para>
</entry><entry
 align="char">
<para>Invalid argument.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_set_pes_filter">
<title>DMX_SET_PES_FILTER</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call sets up a PES filter according to the parameters provided.
By a
 PES filter is meant a filter that is based just on the packet identifier (PID),
i.e.
 no PES header or payload filtering capability is supported.</para>
</entry>
 </row><row><entry
 align="char">
<para>The transport stream destination for the filtered output may be set. Also
the
 PES type may be stated in order to be able to e.g. direct a video stream
directly
 to the video decoder. Finally there is a flag field where it is possible to
state
 whether the filtering operation should be started immediately (without waiting
 for a DMX_START ioctl call). If a filter was previously set-up, this filter
will
 be cancelled, and the receive buffer will be flushed.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request = DMX_SET_PES_FILTER,
```

```
 struct dmx_pes_filter_params &amp;#x22C6;params);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
 align="char">
<para>Equals DMX_SET_PES_FILTER for this command.</para>
</entry>
 </row><row><entry
 align="char">
<para>struct
 dmx_pes_filter_params
 *params</para>
</entry><entry
 align="char">
<para>Pointer to structure containing filter parameters.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
 </row><row><entry
 align="char">
<para>EINVAL</para>
</entry><entry
 align="char">
<para>Invalid argument.</para>
</entry>
 </row><row><entry
 align="char">
<para>EBUSY</para>
</entry><entry
 align="char">
<para>This error code indicates that there are conflicting
 requests. There are active filters filtering data from
 another input source. Make sure that these filters are
 stopped before starting this filter.</para>
</entry>
 </row></tbody></tgroup></informaltable>
```

```
</section>

<section id="dms_set_buffer_size">
<title>DMX_SET_BUFFER_SIZE</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call is used to set the size of the circular buffer used for
filtered data.
 The default size is two maximum sized sections, i.e. if this function is not
called
 a buffer size of 2 * 4096 bytes will be used.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request =
 DMX_SET_BUFFER_SIZE, unsigned long size);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
 align="char">
<para>Equals DMX_SET_BUFFER_SIZE for this command.</para>
</entry>
 </row><row><entry
 align="char">
<para>unsigned long size</para>
</entry><entry
 align="char">
<para>Size of circular buffer.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
```

```
 </row><row><entry
 align="char">
<para>ENOMEM</para>
</entry><entry
 align="char">
<para>The driver was not able to allocate a buffer of the
 requested size.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_get_event">
<title>DMX_GET_EVENT</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call returns an event if available. If an event is not
available,
 the behavior depends on whether the device is in blocking or non-blocking
 mode. In the latter case, the call fails immediately with errno set to
 EWOULDBLOCK. In the former case, the call blocks until an event becomes
 available.</para>
</entry>
 </row><row><entry
 align="char">
<para>The standard Linux poll() and/or select() system calls can be used with
the
 device file descriptor to watch for new events. For select(), the file
descriptor
 should be included in the exceptfds argument, and for poll(), POLLPRI should
 be specified as the wake-up condition. Only the latest event for each filter is
 saved.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request = DMX_GET_EVENT,
 struct dmx_event &#x22C6;ev);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
```

```
 align="char">
<para>Equals DMX_GET_EVENT for this command.</para>
</entry>
 </row><row><entry
 align="char">
<para>struct dmx_event *ev</para>
</entry><entry
 align="char">
<para>Pointer to the location where the event is to be stored.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
 </row><row><entry
 align="char">
<para>EFAULT</para>
</entry><entry
 align="char">
<para>ev points to an invalid address.</para>
</entry>
 </row><row><entry
 align="char">
<para>EWOULDBLOCK</para>
</entry><entry
 align="char">
<para>There is no event pending, and the device is in
 non-blocking mode.</para>
</entry>
 </row></tbody></tgroup></informaltable>
</section>

<section id="dmx_get_stc">
<title>DMX_GET_STC</title>
<para>DESCRIPTION
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>This ioctl call returns the current value of the system time counter
(which is driven
 by a PES filter of type DMX_PES_PCR). Some hardware supports more than one
 STC, so you must specify which one by setting the num field of stc before the
ioctl
 (range 0...n). The result is returned in form of a ratio with a 64 bit
numerator
 and a 32 bit denominator, so the real 90kHz STC value is stc-&amp;#x003E;stc /
 stc-&amp;#x003E;base
 .</para>
</entry>
 </row></tbody></tgroup></informaltable>
```

```
<para>SYNOPSIS
</para>
<informaltable><tgroup cols="1"><tbody><row><entry
 align="char">
<para>int ioctl( int fd, int request = DMX_GET_STC, struct
 dmx_stc &#x22C6;stc);</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>PARAMETERS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>int fd</para>
</entry><entry
 align="char">
<para>File descriptor returned by a previous call to open().</para>
</entry>
 </row><row><entry
 align="char">
<para>int request</para>
</entry><entry
 align="char">
<para>Equals DMX_GET_STC for this command.</para>
</entry>
 </row><row><entry
 align="char">
<para>struct dmx_stc *stc</para>
</entry><entry
 align="char">
<para>Pointer to the location where the stc is to be stored.</para>
</entry>
 </row></tbody></tgroup></informaltable>
<para>ERRORS
</para>
<informaltable><tgroup cols="2"><tbody><row><entry
 align="char">
<para>EBADF</para>
</entry><entry
 align="char">
<para>fd is not a valid file descriptor.</para>
</entry>
 </row><row><entry
 align="char">
<para>EFAULT</para>
</entry><entry
 align="char">
<para>stc points to an invalid address.</para>
</entry>
 </row><row><entry
 align="char">
<para>EINVAL</para>
</entry><entry
 align="char">
<para>Invalid stc number.</para>
</entry>
 </row></tbody></tgroup></informaltable>
```

</section></section>