SPCT6100 Application Development Guide

Version 1.0

Sunplus Core Tech SPCT6100 Linux Team

SPCT6100 Application Development Guide: Version 1.0

by Sunplus Core Tech SPCT6100 Linux Team

Revision History

Revision 1.0 2010-02-22 Revised by: Sunplus Core Technology
Original Version

Table of Contents

List of Examples

Introduction

SPCT6100 EVB Board is a powerful video capturing card dedicated to security field. With a PAC DSP inside, it can compress 1 channel live data with D1 resolution to H.264 format, or 4 channels with CIF resolution, at the framerate of 30fps for NTSC, while 25 fps for PAL.

SPCT6100 EVB Board can work on both Linux and Windows. On Linux, our device driver is V4L2 compatible, so that those free software based on V4L2 can access SPCT6100 EVB Board directly or with minor modification, such as ffmpeg/ffserver/vlc/gstreamer. While on windows, DirectShow interface is used.

This document gives you introduction about how to development application for SPCT6100 using V4L2 interface on Linux

For inquiries about this document contact Sunplus Core Technology.

Chapter 1. Getting Started

Welcome to SPCT6100 Application Development Guide! In this document, you will acquire a comprehensive understanding of how to write applications for SPCT6100 using V4L2 interface.

First, let's have an overview of SPCT6100 video capturing system. SPCT6100 Video Capture Card is PCI hosted, so you can plug it into any pci slot. This card has four anolog video inputs, with a PAC DSP as its strong heart, it can compress raw data produced by this four anolog inputs into H.264 format, many disk spaces thus can be saved. This card transfers data using DMA, every time raw data or compressed data is DMAed to your host memory, driver will inform your applications, you can then do some processing on those data.

To develope basic applications for SPCT6100, we assume you are familiar with the following skills:

- C language
- Linux build system (toolchain, make etc.)
- Linux system call (open, ioctl, mmap etc.)

Now, let's begin our SPCT6100 traval!

Chapter 2. Live Path Data Accessing

In Chapter 1, you were given an overview of the SPCT6100 video capturing system. In this chapter, you'll learn how to write your own SPCT6100 applications.

SPCT6100 supports two YUV formats, YUV422 and YUV420P. When there is only one card, live path data is YUV422 while YUV420P when >= two cards. You can not change this in application layer, driver will detect card number to decide which yuv format should be send to user space

Suppose only one card has been pulgged in PCI slot and we want to save 100 frames raw YUV422 data of channel 0 into a local file, let's start coding (you can get all source code in demo package released with this doc).

Generally speaking, every SPCT6100 application should have the following framework.

- Opening the device
- Setting device properties, such as video standard, resolution, picture brightness etc.
- The actual input/output loop
- Closing the device

2.1. Opening The Device

Every SPCT6100 video capturing card will be registered as two devices in the system. One for live path, the other for compress path. Device registering process is handled by v4l2 kernel module, they will both have the major device number '81' and two consistant minor device numbers. Suppose the only v4l2 device on your system is SPCT6100, then the device name of live path and compress path will be "/dev/video0" and "/dev/video1", 0 and 1 is there minor number.

Now, we know what the device name should be. We want to get live path raw data, so we should open "/dev/video0".

```
/*
 *  spct6100-v4l2-live.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/poll.h>
#include <sys/mman.h>
#include <errno.h>
#include <fcntl.h>
#include <linux/videodev2.h>
#include <spct6100.h>

#define CLEAR(x) memset (&(x), 0, sizeof (x))

int fd;
```

```
int totalframes;
struct buffer
{
  void *start;
  size_t length;
};
struct buffer *buffers = NULL;
FILE *output = NULL;
void save_one_frame_yuv (void *addr);
void process_data (void);

static void
errno_exit (const char *s)
{
  fprintf (stderr, "%s error %d, %s\n", s, errno, strerror (errno));
  exit (EXIT_FAILURE);
}

int
main (int argc, char **argv)
{
  int index;
  int n = 0;
  unsigned int i;
  struct pollfd pfd;
  struct v4l2_input input;
  enum v4l2_buf_type type;
  struct v4l2_control change_mode;
  struct v4l2_control change_resolution;
  struct v4l2_requestbuffers reqbuf;

  fd = open ("/dev/video0", O_RDWR | O_SYNC);
  if (fd < 0) {

    printf ("Open device failed\n");
  }
```

Now, we have got the file handler, we can operate on the live path device through this handler.

2.2. Change Device Properity

When a SPCT6100 video processing card start working, it works on a deafult setting, including video standard, resolution etc. So your application should change them to what you wanted.

In this example, we want the card to work on NTSC mode.

```
  change_mode.id = V4L2_CID_MODE;
  change_mode.value = 1;    // 0: PAL, 1: NTSC

  for (index = 0; index < 4; index++) {

    if (-1 == ioctl (fd, VIDIOC_S_INPUT, &index)) {

      perror ("VIDIOC_S_INPUT");
      exit (EXIT_FAILURE);
```

```
    }

    if (-1 == ioctl (fd, VIDIOC_S_CTRL, &change_mode)) {

      perror ("VIDIOC_S_CTRL");
      exit (EXIT_FAILURE);
    }
  }
```

Note: for SPCT6100, different channel can have different value on the same properity. So, every time you want to set the properity of a specified channel, you should using `VIDIOC_S_INPUT`. To here, we have make the card work as we needed, we can begin to save data.

2.3. Streaming IO

Streaming is an I/O method where only pointers to buffers are exchanged between application and driver, the data itself is not copied. Memory mapping is primarily intended to map buffers in device memory into the application's address space. Device memory can be for example the video memory on a graphics card with a video capture add-on. However, being the most efficient I/O method available for a long time, many other drivers support streaming as well, allocating buffers in DMA-able main memory.

A driver can support many sets of buffers. Each set is identified by a unique buffer type value. The sets are independent and each set can hold a different type of data. To access different sets at the same time different file descriptors must be used.

To use streaming io(Memory Mapping) you should do the following step:

Step 1: Get the number of supported device buffers. Call the `VIDIOC_REQBUFS` ioctl with the desired number of buffers and buffer type(for SPCT6100 it should always be V4L2_BUF_TYPE_VIDEO_CAPTURE). If the number you requested is not supported, driver will change that number to the correct one.

Step 2: Memory Map and initialize all buffers. Before applications can access the buffers they must map them into their address space with the mmap function. The location of the buffers in device memory can be determined with the `VIDIOC_QUERYBUF` ioctl. The *`m.offset`* and *`length`* returned in a struct v4l2_buffer are passed as sixth and second parameter to the `mmap()` funciton. the offset and length values must not be modified. `VIDIOC_QBUF` ioctl should be called later to initialize all buffers. Remember the buffers are allocated in physical memory, as opposed to virtual memory which can be swapped out to disk. Applications should free the buffers as soon as possible with the munmap function.

Step 3: Make the buffer queues working. After memory mapping all buffers, you could inform the driver to begin fill data into these buffers with the `VIDIOC_STREAMON` ioctl with the desired buffer type(V4L2_BUF_TYPE_VIDEO_CAPTURE).

Step 4: Processing data. After the first three steps, data will be continusly filled into buffers, driver are now data producer, your application should be data consumer. Using `VIDIOC_DQBUF` ioctl you can determine which buffer has data now, and process data in this buffer. When finishing data processing, you should using `VIDIOC_QBUF` to mask this buffer empty, so that data can be filled into it again.

Step 5: Make the buffer queues stop working. Whenever your application do not need data ever, you should inform driver to stop fill data into these buffers using `VIDIOC_STREAMOFF` ioctl.

Code for this part is listed below.

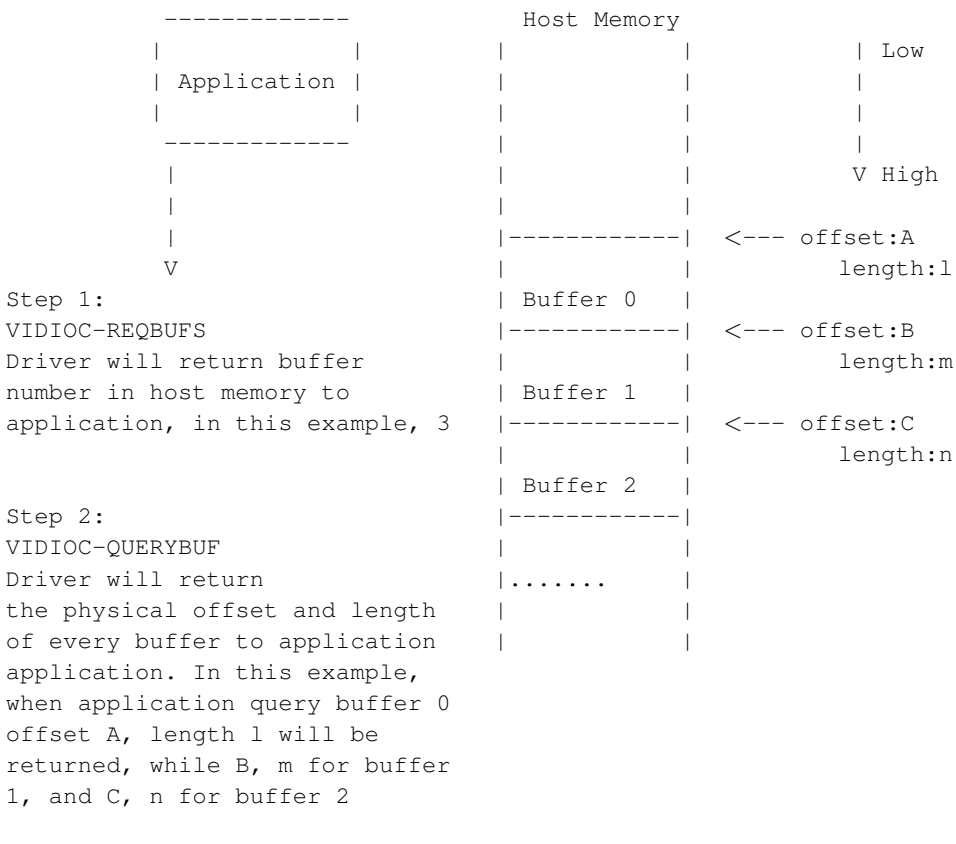
```
/*** Step 1 Start ***/
memset (&reqbuf, 0, sizeof (reqbuf));
reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.memory = V4L2_MEMORY_MMAP;
reqbuf.count = 12;

if (-1 == ioctl (fd,  VIDIOC_REQBUFS, &reqbuf)) {
        if (errno == EINVAL)
                printf ("Video capturing or mmap-streaming is not supported\n");
        else
                perror ("VIDIOC_REQBUFS");

        exit (EXIT_FAILURE);
}
/*** Step 1 End ***/

/*** Step 2 Start ***/
buffers = calloc (reqbuf.count, sizeof (*buffers));

  // Memory Mapping
for (i = 0; i < reqbuf.count; i++) {
        struct  v4l2_buffer buffer;

        memset (&buffer, 0, sizeof (buffer));
        buffer.type = reqbuf.type;
        buffer.memory = V4L2_MEMORY_MMAP;
        buffer.index = i;

        if (-1 == ioctl (fd,  VIDIOC_QUERYBUF, &buffer)) {
                perror ("VIDIOC_QUERYBUF");
                exit (EXIT_FAILURE);
        }

        buffers[i].length = buffer.length; /* remember for munmap() */

        buffers[i].start = mmap (NULL, buffer.length,
                                 PROT_READ | PROT_WRITE, /* recommended */
                                 MAP_SHARED,             /* recommended */
                                 fd, buffer.m.offset);

        if (MAP_FAILED == buffers[i].start) {
                /* If you do not exit here you should unmap() and free()
                   the buffers mapped so far. */
                perror ("mmap");
                exit (EXIT_FAILURE);
        }
}

  // Initialize all buffers ***/
for (index = 0; index < reqbuf.count; index++) {

  struct v4l2_buffer buf;
  CLEAR(buf);
  buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
  buf.memory = V4L2_MEMORY_MMAP;
  buf.index = index;
  if (-1 == ioctl(fd, VIDIOC_QBUF, &buf))
```

```
    errno_exit("VIDIOC_QBUF");
}

/*** Step 2 End ***/

/*** Step 3 Start ***/

type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (-1 == ioctl(fd, VIDIOC_STREAMON, &type))
  errno_exit("VIDIOC_STREAMON");

/*** Step 3 End ***/
```

2.4. Memory Mapping

Using the upper code, we have establish a whole memory maping system, through which we can access the DMA memory in user space directly.

Memory mapping for SPCT6100 system can be described in the following map:



```
space to physical memory.

the structure buffers will record
all those user space information,
so you can using them later.
```

2.5. Live Data Accessing

Up to now, the data queue has been established and driver has began to fill data into those buffer. Then how should you get informed when data is ready? using poll.

```
  output = fopen("data.yuv", "wb");

  pfd.fd =  fd;
  pfd.events = POLLIN;
  while(1) {

    n= poll(&pfd, 1, -1);
    if(n < 0) {

      break;
    } else {

      process_data();
      if(totalframes >= 100)
      break;
    }
  }

  type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
  if (-1 == ioctl(fd, VIDIOC_STREAMOFF, &type))
    errno_exit("VIDIOC_STREAMOFF");

  fclose(output);
  close(fd);
} // End of main
```

When the system call *poll* is called, the process will be blocked until driver informed the process that some buffers have data ready to be processed. After processing the data, process should call poll again to wait for the next time driver inform.

Then how should we process those data ready buffer ? Let's continue.

```
void process_data(void)
{
  struct v4l2_buffer buf;
  CLEAR(buf);
  buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
  buf.memory = V4L2_MEMORY_MMAP;
  if (-1 == ioctl (fd, VIDIOC_DQBUF, &buf))
    errno_exit("VIDIOC_DQBUF");
  save_one_frame_yuv(buf);
  if (-1 == ioctl (fd, VIDIOC_QBUF, &buf))
    errno_exit("VIDIOC_QBUF");
}
```

To process the data-ready buffer, you should first get them from the data queue using `VIDIOC_DQBUF`, driver pass many information to the process through those fields in the structure you has passed when calling `VIDIOC_DQBUF`. Using those information, the process can know the address, length, properity of the data, then some processing can be done. After all this, `VIDIOC_QBUF` should be called to inform driver that these buffer is empty again and data can be refilled to them.

Here comes the finally section, how can we know the address, length, properity etc of the data ?

```
void save_one_frame_yuv(struct v4l2_buffer buf)
{
  int buf_index, ch;

  for (ch = 0; ch < 4; ch++) {
    buf_index = (buf.index >> (ch*8)) & 0xFF;
    if (ch != 2 || buf_index == 0xFF) {
      continue;
    }

    fwrite(buffers[buf_index].start, buf.length, 1, output);
    totalframes++;
  }
}
```

We assume that you know meanings of those fields in struct *v4l2_buffer*, or please see v4l2 specification from the website first. For live path, driver will pass information to user space in the following format: (only two fields are used for live path)

```
   struct v4l2_buffer {
     ......
     unsigned int index;
                   |
                   |
                   v

              bit 31     bit 24   bit 16  bit 8   bit 0
                  |         |        |        |        |
                  V         V        V        V        V
                   ------------------------------------
                  | ch 3   | ch 2   | ch 1   | ch 0   |
                  | index  | index  | index  | index  |
                   ------------------------------------

                  if index == 0xFF
                    there is no data for this channel

                  if index != 0xFF {
                    there is data for this channel
                    data address in user space
                         = your previous mmap array [ index ]
                  }
     ......
     unsigned int length;
                   |
                   |
                   V
                   Length of the data, for NTSC/CIF/YUV422,
```

```
                driver will filled this with 352*288*2
};
```

Everytime driver wakes up user space process, information about every channel will be recorded on these fields. If the user space process does not care about some of them, like this example we only care about data from channel 2, it can simply ignore those information.

Up to now, all the code have been listed. You can compile this file and run it.

In the following chapters, details about compress path data access will be given.

Chapter 3. Variable length Data Accessing

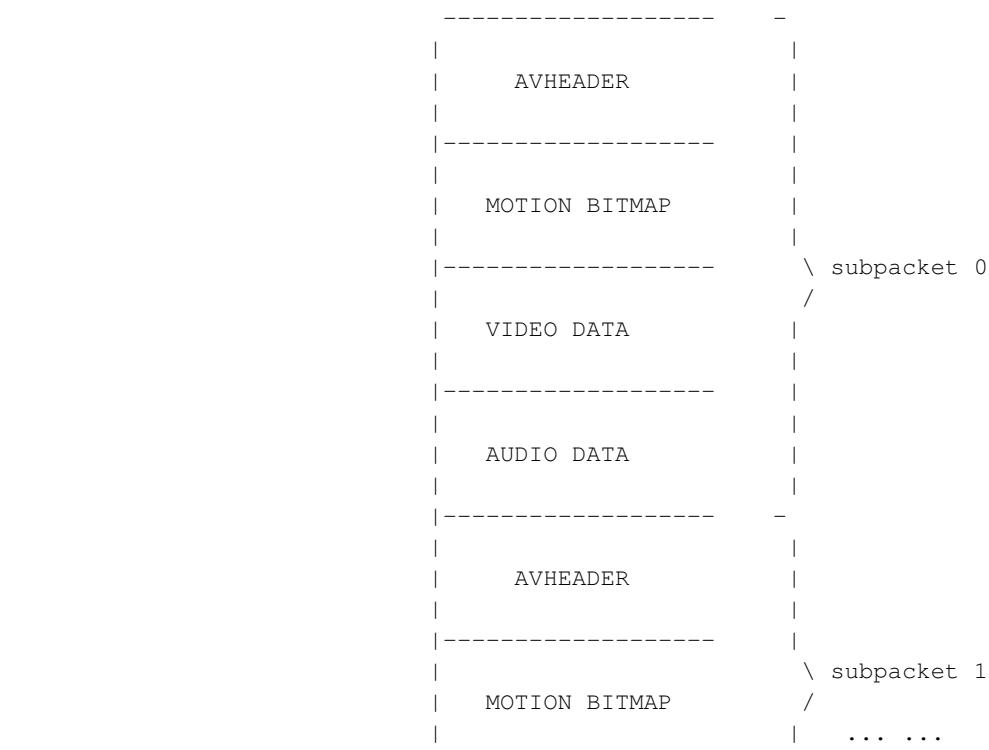
3.1. Separate H.264, Audio, Motion

Compressed H.264 data, ADPCM audio data, Motion detection bitmap are all variable length data, packed in one packet and passed to user space through compress path. How can we separate them ?

Remember that in the previous section, when `VIDIOC_DQBUF` is called, driver will fill some information in those fields `VIDIOC_DQBUF` passed down. For live path (/dev/video0), only two fields are used(index/length), for compress path (/dev/video1), the meaning of the field 'length' has been changed

```
struct v4l2_buffer {
  ......
  unsigned int index;
                |
                |
                v

           bit 31     bit 24   bit 16  bit 8   bit 0
               |         |        |        |        |
               V         V        V        V        V
                -----------------------------------
               | ch 3   | ch 2   | ch 1   | ch 0   |
               | index  | index  | index  | index  |
                -----------------------------------

               if index == 0xFF
                 there is no data for this channel

               if index != 0xFF {
                 there is data for this channel
                 data address in user space
                      = your previous mmap array [ index ]

               }

  unsigned int length;
                |
                |
                v

           bit 31     bit 24   bit 16  bit 8   bit 0
               |         |        |        |        |
               V         V        V        V        V
                -----------------------------------
               | ch 3   | ch 2   | ch 1   | ch 0   |
               | subnum | subnum | subnum | subnum |
                -----------------------------------


           Here, to understand the word 'subnum', detailed information about
           compress path packet must be introduced.

           Every compress path packet have 'subnum' subpackets, so the packet
```

has the following structure:



what's the offset and length of motion/video/audio in each subpacket?
see AVHEADER.

```
typedef struct {
  u32 AVH_uReserved0             // 0x00

  u32 AVH_uTotalDataSize;        // 0x04
          total Data length of this subpacket
          with this header included.

  u32 AVH_uReserved1;            // 0x08

  u32 AVH_uFrameType;            // 0x0C
          video frame type
          0 ~ I frame
          2 ~ P frame
          3 ~ B frame

  u32 AVH_uVideoTimeStamp;       // 0x10
          video frame time stamp
          the unit is 1/32768 second

          Note: this field is 32 bits, so
          you must pay attention to counter
          overflow.

  u32 AVH_uVideoDataLen;        // 0x14
          video frame data length
```

```
  u32 AVH_uAudioDataLen;        // 0x18
        audio frame data length

  u32 AVH_uSpsLen;              // 0x1C
        sps header length

  u32 AVH_uVideoDataAddrOffset; // 0x20
        video frame data offset in subpacket

  u32 AVH_uAudioDataAddrOffset; // 0x24
        audio frame data offset in subpacket

  u32 AVH_uReserved2;           // 0x28

  u8  AVH_uReserved3;           // 0x2C

  u8  AVH_Func_Motion_En;       // 0x2D
        motion detection unit status

  u8  AVH_Func_Encode_En;       // 0x2E
        video encoder unit status

  u8  AVH_Func_AudioEncode_En;  // 0x2F
        audio encoder unit status

  u32 AVH_uVideoLost;           // 0x30
        video lost status
        0 ~ no video lost
        1 ~ video lost

  u32 AVH_uMotionDataLen;       // 0x34
        motion detection bitmap length

  u32 AVH_uAudioTimeStamp;      // 0x38
        audio frame time stamp
        the unit is 1/32768 second also

  u8  AVH_uReserved4[88 - 0x3c];
} __attribute__((packed)) DMAVHEADER, *PDMAVHEADER;
```

everytime you `VIDIOC_DQBUF` a compress path packet, you can using the following process logic:

```
  int buf_index = v4l2_buf.index;
  int buf_len =   v4l2_buf.length;
  int index, subpacket_num
  int channel_index, subpacket_index;
  unsigned char *cur_addr = NULL;

  for (channel_index = 0; channel_index < 4; channel_index++) {

    index = buf_index & 0xff;
    subpacket_num = buf_len & 0xff;
    buf_index >>= 8;
    buf_len >>= 8;

    /* suppose we only process data from channel 2 */
```

```
    if (channel_index == 2 && index != 0xff) {
      cur_addr = buffers[index].start;
      for (subpacket_index = 0;
           subpacket_index < subpacket_num;
           subpacket_index++) {

        if (PACKET_WITH_VIDEO(cur_addr)) {
          video_data_address =
              mmap_addr[index].start + AV_VOFFSET(cur_addr);
          video_data_length =  AV_VLEN(cur_addr);

          do_video_process(video_data_address,
                           video_data_length);
        }

        if (PACKET_WITH_AUDIO(cur_addr)) {
          audio_data_address =
              mmap_addr[index].start + AV_AOFFSET(cur_addr);
          audio_data_length =  AV_ALEN(cur_addr);

          do_audio_process(audio_data_address,
                           audio_data_length);
        }

        if (PACKET_WITH_MOTION(cur_addr)) {
          motion_bitmap_address =
              mmap_addr[index].start + sizeof(DMAVHEADER);
          motion_bitmap_length = 72;  // bitmap has fixed length
          do_motion_process(motion_bitmap_address,
                            motion_bitmap_length);
        }

        if (PACKET_WITH_VIDEOLOST(cur_addr)) {
          do_videolost_process();
        }

        // update cur_addr to the next subpacket
        cur_addr+=AV_TOTAL(cur_addr);
      }
    }
  }
```

all these help macros:

```
AV_TOTAL
AV_ALEN
AV_AOFFSET
AV_VLEN
AV_VOFFSET
PACKET_WITH_VIDEO
PACKET_WITH_AUDIO
PACKET_WITH_MOTION
PACKET_WITH_VIDEOLOST
```

can be found in spct6100 sdk development header file 'spct6100.h'

Now, with knowledge getting from the previous chapter and information from the upper structure, you know whether it is H.264 data or audio data or motion detection bitmap, you also know where to get the data and the length of the data, you surely can process variable length data now!

3.2. H.264 Data Accessing

Only minor difference exists beteen H.264 and live yuv data accessing, when access H.264 data we should analyze those extra fields of struct *v4l2_buffer*. This time we want to save 100 frame H.264 data from channel 2, just modify the function *save_one_frame_yuv* in spct6100-v4l2-live.c, and the device to open should be changed from '/dev/video0' to '/dev/video1' and output file name to 'data.264' then everything will OK! For full code list, please see spct6100-v4l2-compress.c in demos package

Example 3-1. spct6100-v4l2-compress.c

```
      ... ...

/*Here we change the function name from

       save_one_frame_yuv
  to
       save_one_frame_H264 */

void save_one_frame_H264(struct v4l2_buffer buf)
{
  int buf_index = buf.index;
  int buf_len = buf.length;
  int index, subpacket_num, channel_index, subpacket_index;
  unsigned char *cur_addr = NULL;
  for (channel_index = 0; channel_index < 4; channel_index++) {
    index = buf_index & 0xff;
    subpacket_num = buf_len & 0xff;
    buf_index >>= 8;
    buf_len >>= 8;

    /* We only process data from channel 2 */

    if (channel_index == 2 && index != 0xff) {
      cur_addr = buffers[index].start;
      for (subpacket_index = 0; subpacket_index < subpacket_num; subpacket_index++) {
        if (!PACKET_WITH_VIDEO(cur_addr)) {
          cur_addr+=AV_TOTAL(cur_addr);
          continue;
        }
        if (PACKET_IS_ISPSFRAME(cur_addr))
          got_sps = 1;

        if (got_sps) {
          printf("Save video frame %d\n", totalframes);
          fwrite(cur_addr + AV_VOFFSET(cur_addr), AV_VLEN(cur_addr), 1, output);
          totalframes++;
        }
        cur_addr+=AV_TOTAL(cur_addr);
      }
```

```
    }
  }
}
```

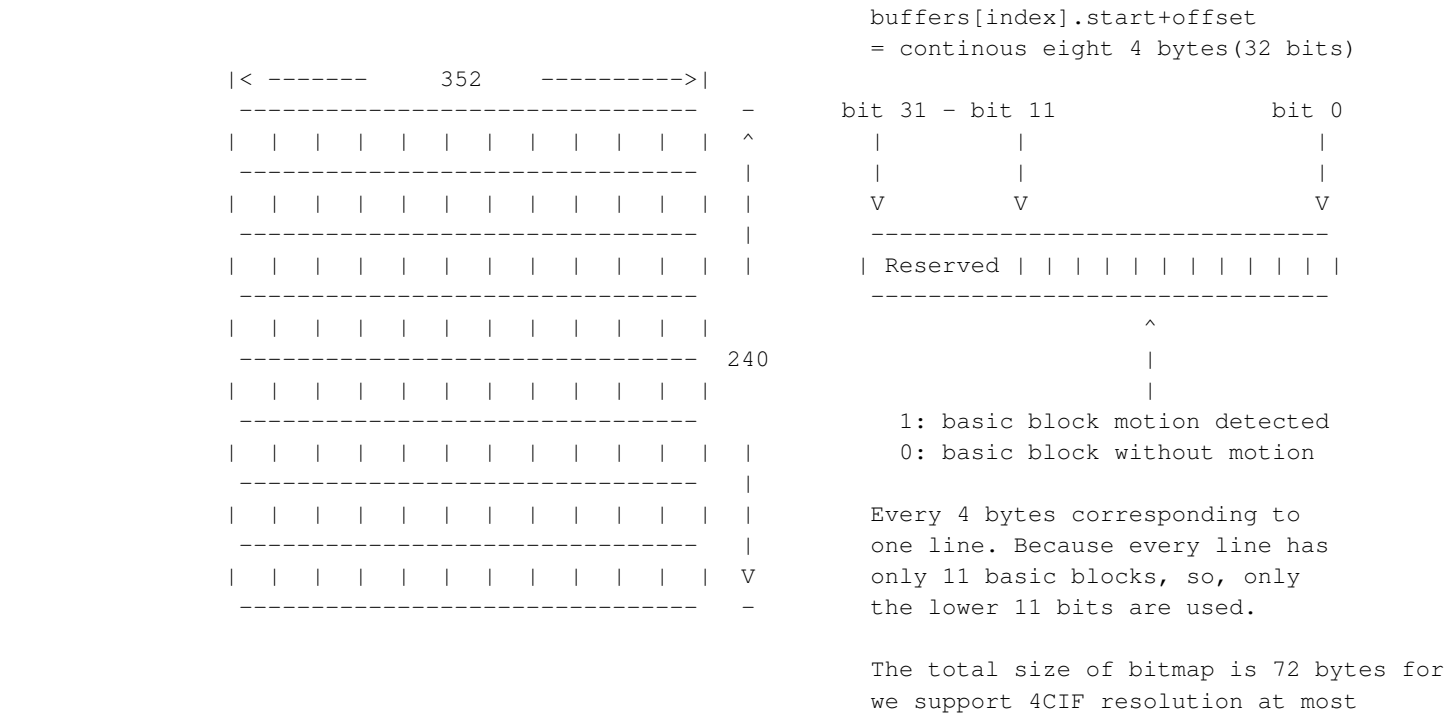
We only need to override data processing function, then everything will correct for compress path.

3.3. Motion Detection Bitmap

SPCT6100, with DSP inside, can do some software processing on the data, such as motion detection. Driver will send you motion detection results through compress path.

In the code above, if PACKET_WITH_MOTION(subpacket_address) is true, then the data has motion detection result. Instead of writing the results to file, you should do some analysis on the results.

When detect motions, SPCT6100 see the screen as many 32x32 basic blocks. If any basic block has motion detected then driver will send you the results in the following format:



3.4. Audio Data Accessing

Audio data has the format G.723, you can processing audio data as you like

By default audio was closed on the board, you can enable it using VIDIOC_S_CTRL with id equals V4L2_CID_AUDIO

Chapter 4. Card Properity

Devices typically have a number of user-settable controls such as brightness, saturation and so on, so does for SPCT6100 video capture card. SPCT6100 V4L2 Linux driver gives you ability to operate nearly every settable properity of this card. You can change these properities to new value or reading the current value of a specified properity of a specified channel.

4.1. Set The Properity

To set the specified properity of the card to new value, you should call ioctl with `VIDIOC_S_CTRL` and a pointer to a struct v4l2_control struct as the second and third parameters. struct v4l2_control has two field, you should set the *`id`* field to the corresponding id, and *`value`* field to the new value.

All controls are accessed using an ID value. V4L2 defines several IDs for specific purposes. SPCT6100 also implement some other custom controls using `V4L2_CID_PRIVATE_BASE` and higher values. The pre-defined and customed control IDs have the prefix `V4L2_CID_`. The ID is used when querying the attributes of a control, and when getting or setting the current value. All `VIDIOC_S_CTRL` should with the same parameter

```
struct v4l2_control {
  unsigned int id;
  unsigned int value;
};


//set color brightness
id = V4L2_CID_BRIGHTNESS
value: 0 ~ 255;
stage: any time
stage means when this ioctl can be used


//set color contrast
id = V4L2_CID_CONTRAST
value: 0 ~ 255
stage: any time


//set color saturation
id = V4L2_CID_SATURATION
value: 0 ~ 255
stage: any time


//set color hue
id = V4L2_CID_HUE
value: 0 ~ 255
stage: any time


//set video standard (NTSC/PAL)
id = V4L2_CID_MODE
value: 0 ~ NTSC
       1 ~ PAL
```

```
stage: before STREAMON


//live and compress status control
id = V4L2_CID_STATUS
value = 0 ~ close both live and compress path
        1 ~ open compress path only
        2 ~ open live path only
        3 ~ open both live and compress path
stage: before STREAMON


//set live raw yuv frame resolution
id = V4L2_CID_LSHOWSIZE
value = 0 ~ 4CIF
        1 ~ CIF
        2 ~ QCIFF
        3 ~ 2CIFF
stage: before STREAMON


//set compress H.264 frame resolution
id = V4L2_CID_CSHOWSIZE
value = 0 ~ 4CIF
        1 ~ CIF
        2 ~ QCIF
        3 ~ 2CIF
stage: before STREAMON


//set fps
id = V4L2_CID_EENSPEED
value = 0 ~ 30
stage: anytime


//video lost control
id = V4L2_CID_VLOSTEN
value = 0 ~ disable video lost
        1 ~ enable  video lost
stage: any time

//enable or disable audio
id = V4L2_CID_AUDIO
value = 0 ~ disable audio
        1 ~ enable  audio
stage: before STREAMON

//request I frame with sps header
id = V4L2_CID_REQIFRAME
value = 0 ~ 3 specified channel request
        4     all channel request
stage: anytime


//change H.264 frame interval
id = V4L2_CID_GOPSET
```

```
value = pointer to struct spct6100_gopset

        struct spct6100_gopset {
          unsigned int gopp;          // how many P frame between
                                      // two I frame

          unsigned int gopb;          // how many B frame between
                                      // two I frame
        };
stage: anytime


//set H.264 encode parameter
id = V4L2_CID_EQUALITY
value = pointer to struct spct6100_eqset

        struct spct6100_eqset {
          unsigned int type;          // 0 ~ CBR   2 ~ VBR

          unsigned int minbitrate;    // when type == 0, minbitrate
          unsigned int maxbitrate;    // and maxbitrate must be specified

          unsigned int qpi;
          unsigned int qpb;           // when type == 2, qpi,qpb,qpp
          unsigned int qpp;           // must be specified
        };
stage: anytime


//enable motion detection
id = V4L2_CID_MOTIONDETECT
value = 0 ~ disable  motion detection
        1 ~ enable   motion detection
stage: anytime


//set motion detection threshold
//this setting should be followed by V4L2_CID_MOTIONDETECT
//with enable == 1 to make the change valid at once
id = V4L2_CID_MTHRESHOLD
value = new threshold value
stage: anytime


//set motion detection interval
//this setting should be followed by V4L2_CID_MOTIONDETECT
//with enable == 1 to make the change valid at once
id = V4L2_CID_MINTERVAL
value = new interval value
stage: anytime


//set screen osd
//currently, SPCT6100 allows a osd timer and a at most 15
//bytes string on the screen simutaneously.
//the osd timer will always on the top-left corner of the screen,
//while you can specifiy the position of the string
```

```
id = V4L2_CID_OSDSET
value = pointer to struct spct6100_osdset

        struct spct6100_osdset {
          unsigned int tran;            // the following 4 fields
          unsigned int acolor;          // are invalid now
          unsigned int vscale;
          unsigned int hscale;
          unsigned int timer_enable;    // 0 ~ don't show osd timer
                                        // 1 ~ show osd timer

          unsigned int osd_enable;      // 0 ~ disable the string
                                        // 1 ~ enable  the string

          unsigned int osd_pos_x;       // x position of the string
          unsigned int osd_pos_y;       // y position of the string
          unsigned int str[16];         // string contents
        };
stage: anytime


//set mosaic area
id = V4L2_CID_MOSAIC
value = pointer to struct spct6100_mosaic

        struct spct6100_mosaic {
          unsigned int num;             // number of mosaic area
                                        // SPCT6100 support at most 6

          unsigned int x[6];            // x position of top-left corner
          unsigned int y[6];            // y position of top-left corner
          unsigned int w[6];            // width of each mosaic area
          unsigned int h[6];            // height of each mosaic area
        };
stage: anytime

//write eeprom through i square c bus
id = V4L2_CID_WI2C
value = pointer to struct spct6100_eeprom

        struct spct6100_eeprom {
          unsigned int deviceid;        // device id, for eeprom, 0xa0
          unsigned int addr;            // address with eeprom address space
          unsigned int length;          // length of data
          char *buf                     // data should be written
        };
stage: anytime


//read eeprom through i square c bus
id = V4L2_CID_RI2C
value = pointer to struct spct6100_eeprom

        struct spct6100_eeprom {
          unsigned int deviceid;        // device id, for eeprom, 0xa0
          unsigned int addr;            // address with eeprom address space
          unsigned int length;          // length of data
```

```
          char *buf                     // driver will fill data here
                                        // application should malloc this space
        };
stage: anytime

//set gpio direction
id = V4L2_CID_SGPIOD
value = pointer to struct spct6100_gpio

        struct spct6100_gpio {
          unsigned int pinid;           // pin id;
          void *value;                  // pointer to the direction
                                        // 0 ~ out
                                        // 1 ~ in
        };
stage: anytime


//write gpio pin
id = V4L2_CID_WGPIO
value = pointer to struct spct6100_gpio

        struct spct6100_gpio {
          unsigned int pinid;           // pin id;
          void *value;                  // pointer to the value to write
        };
stage: anytime


//read gpio pin
id = V4L2_CID_RGPIO
value = pointer to struct spct6100_gpio

        struct spct6100_gpio {
          unsigned int pinid;           // pin id;
          void *value;                  // driver will fill data to memory
                                        // pointed to by value
        };
stage: anytime

//enable watchdog
id = V4L2_CID_WATCHEN
value = 0 ~ disable
        1 ~ enable
stage: anytime


//watchdog setting
id = V4L2_CID_WATCHSET
value = pointer to struct spct6100_watch

        struct spct6100_watch {
          unsigned int timeout;         // timeout value
                                        // period in second before reset
        };
stage: anytime
```

4.2. Get The Properity

To get the specified properity, things almost the same with that of setting, except that you should call ioctl with `VIDIOC_G_CTRL` and need only to fill the *`id`* field, then driver will fill the *`value`* field to the current value of the properity. If the *`value`* field of the corresponding `VIDIOC_S_CTRL` is a pointer to a structure, then it also should be that in `VIDIOC_G_CTRL`.

4.3. Example

Suppose we want to change the brightness to 75, if it is now below 70. So we first should get the current value, then decide what to do next.

Example 4-1. spct6100-set-brightness.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/poll.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <linux/videodev2.h>
#include <spct6100.h>

#define CLEAR(x) memset (&(x), 0, sizeof (x))

  static void
errno_exit(const char *s)
{
  fprintf(stderr, "%s error %d, %s\n", s, errno, strerror(errno));
  exit(EXIT_FAILURE);
}

int main (int argc, char *argv[])
{
  int err = 0;
  int fd = -1;
  int channelnum = 1;
  struct v4l2_control brightness_control;

  fd = open( "/dev/video0", O_RDWR);
  if (fd < 0) {

    printf( "Can not open SPCT6100 PCI device....\n" );
    exit(0);
  }

  if (-1 == ioctl(fd, VIDIOC_S_INPUT, &channelnum))
    errno_exit("VIDIOC_S_INPUT");

  CLEAR(brightness_control);
  brightness_control.id = V4L2_CID_BRIGHTNESS;
```

```
if (-1 == ioctl(fd, VIDIOC_QUERYCTRL, &brightness_control)) {

  errno_exit("VIDIOC_QUERYCTRL");
}
else {

  // You should check if this control is supported first.
  if (brightness_control.flags & V4L2_CTRL_FLAG_DISABLED) {

    errno_exit("This ID is not supported");
  }
}

if (-1 == ioctl(fd, VIDIOC_G_CTRL, &brightness_control))
  errno_exit("VIDIOC_G_CTRL");
else
  printf("The current brightness of channel 1 is: %d\n", brightness_control.value);

if (brightness_control.value < 70) {

  brightness_control.id = V4L2_CID_BRIGHTNESS;
  brightness_control.value = 75;
  if (-1 == ioctl(fd, VIDIOC_S_CTRL, &brightness_control))
    errno_exit("VIDIOC_S_CTRL");
  else
    printf("Changing brightness of channel 1 success!\n");

  return 0;
}
```

Chapter 5. Multicard Accessing

Some main board has several PCI slot, so you can plug two, three, four or even more SPCT6100 video capture card on your PC, our linux driver can support these card to work simutaneously.

When more than one card are plugged, every card will have a unique name. Suppose your PC has no other video capture card plugged in, so the first SPCT6100 card will occupy device file /dev/video0 (Live Path) and /dev/video1 (Compress Path), the second will occupy /dev/video1 and /dev/video2, then the third, the fourth and the more. You can open different device file to access different card.

Every card should have it's own buffer queue, not shared with other cards. Every card can have there own properity (brightness, contrast, etc).

In on word, you can use SPCT6100 video capture card to construct your 4, 8, 16, or even more channels security system!

I. Function Reference

Table of Contents

ioctl VIDIOC_ENUM_FMT

Name

VIDIOC_ENUM_FMT — Enumerate image formats

Synopsis

```
int ioctl(int fd, int request, struct v4l2_fmtdesc *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_ENUM_FMT

argp

Description

To enumerate image formats applications initialize the *type* and *index* field of struct v4l2_fmtdesc and call the VIDIOC_ENUM_FMT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code. All formats are enumerable by beginning at index zero and incrementing by one until EINVAL is returned.

Table 1. struct v4l2_fmtdesc

__u32	*index*	Number of the format in the enumeration, set by the application. This is in no way related to the *pixelformat* field.
enum v4l2_buf_type	*type*	Type of the data stream, set by the application. For SPCT6100 it should be V4L2_BUF_TYPE_VIDEO_CAPTURE.
__u32	*flags*	
__u8	*description*[32]	Description of the format, a NUL-terminated ASCII string. This information is intended for the user, for example: "YUV 4:2:2". Driver will fill this field.
__u32	*pixelformat*	The image format identifier. This is a four character code as computed by the v4l2_fourcc macro.

__u32	*reserved*[4]	Reserved for future extensions. Drivers must set the array to zero.

Table 2. Image Format Description Flags

V4L2_FMT_FLAG_COMPRESSED	0x0001	This is a compressed format.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_fmtdesc *type* is not supported or the *index* is out of bounds.

ioctl VIDIOC_ENUMINPUT

Name

VIDIOC_ENUMINPUT — Enumerate video inputs

Synopsis

```
int ioctl(int fd, int request, struct v4l2_input *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_ENUMINPUT

argp

Description

To get the data or query the attributes of a specified video channel applications initialize the *index* field of struct v4l2_input and call the VIDIOC_ENUMINPUT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all inputs applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

Table 1. struct v4l2_input

__u32	*index*	Identifies the input, set by the application.
__u8	*name*[32]	Name of the video channel, a NUL-terminated ASCII string, for example: "Card 0 Channel 2". This information is intended for the user, filled by the driver.
__u32	*type*	Type of the input. For SPCT6100, always V4L2_INPUT_TYPE_CAMERA.
v4l2_std_id	*std*	Every video input supports one or more different video standards. This field is a set of all supported standards.
__u32	*reserved*[4]	Reserved for future extensions. Drivers must set the array to zero.

Table 2. Input Types

V4L2_INPUT_TYPE_TUNER	1	This input uses a tuner (RF demodulator).
V4L2_INPUT_TYPE_CAMERA	2	Analog baseband input, for example CVBS / Composite Video, S-Video, RGB.

Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_input *index* is out of bounds.

ioctl VIDIOC_ENUMSTD

Name

VIDIOC_ENUMSTD — Enumerate supported video standards

Synopsis

```
int ioctl(int fd, int request, struct v4l2_standard *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_ENUMSTD

argp

Description

To query the attributes of a video standard, especially a custom (driver defined) one, applications initialize the *index* field of struct v4l2_standard and call the VIDIOC_ENUMSTD ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all standards applications shall begin at index zero, incrementing by one until the driver returns EINVAL. Drivers may enumerate a different set of standards after switching the video input or output.

Table 1. struct v4l2_standard

__u32	*index*	Number of the video standard, set by the application.
v4l2_std_id	*id*	For SPCT6100, V4L2_STD_NTSC_M or V4L2_STD_PAL_M are returned
__u8	*name*[24]	Name of the standard, a NUL-terminated ASCII string, for example: "PAL-B/G", "NTSC Japan". This information is intended for the user.
struct v4l2_fract	*frameperiod*	The frame period (not field period) is numerator / denominator. For example M/NTSC has a frame period of 1001 / 30000 seconds.
__u32	*framelines*	Total lines per frame including blanking, e. g. 625 for B/PAL.

__u32	*reserved*[4]	Reserved for future extensions. Drivers must set the array to zero.

Table 2. struct v4l2_fract

__u32	*numerator*
__u32	*denominator*

Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_standard *index* is out of bounds.

ioctl VIDIOC_G_CTRL, VIDIOC_S_CTRL

Name

VIDIOC_G_CTRL, VIDIOC_S_CTRL — Get or set the value of a control

Synopsis

```
int ioctl(int fd, int request, struct v4l2_control *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_G_CTRL, VIDIOC_S_CTRL

argp

Description

To get the current value of a control applications initialize the *id* field of a struct v4l2_control and call the VIDIOC_G_CTRL ioctl with a pointer to this structure. To change the value of a control, applications initialize the *id* and *value* fields of a struct v4l2_control and call the VIDIOC_S_CTRL ioctl.

When the *id* is invalid drivers return an EINVAL error code. When the *value* is out of bounds drivers can choose to take the closest valid value or return an ERANGE error code, whatever seems more appropriate. However, VIDIOC_S_CTRL is a write-only ioctl, it does not return the actual new value.

These ioctls work only with user controls.

Table 1. struct v4l2_control

__u32	*id*	Identifies the control, set by the application.
__s32	*value*	New value or current value.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_control `id` is invalid.

ERANGE

The struct v4l2_control `value` is out of bounds.

EBUSY

The control is temporarily not changeable, possibly because another applications took over control of the device function this control belongs to.

ioctl VIDIOC_G_FMT, VIDIOC_S_FMT, VIDIOC_TRY_FMT

Name

VIDIOC_G_FMT, VIDIOC_S_FMT, VIDIOC_TRY_FMT — Get or set the data format, try a format

Synopsis

```
int ioctl(int fd, int request, struct v4l2_format *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_G_FMT, VIDIOC_S_FMT, VIDIOC_TRY_FMT

argp

Description

These ioctls are used to negotiate the format of data (typically image format) exchanged between driver and application.

To query the current parameters applications set the *type* field of a struct v4l2_format to the respective buffer (stream) type. For example video capture devices use V4L2_BUF_TYPE_VIDEO_CAPTURE. When the application calls the VIDIOC_G_FMT ioctl with a pointer to this structure the driver fills the respective member of the *fmt* union. In case of video capture devices that is the struct v4l2_pix_format *pix* member. When the requested buffer type is not supported drivers return an EINVAL error code.

To change the current format parameters applications initialize the *type* field and all fields of the respective *fmt* union member. Good practice is to query the current parameters first, and to modify only those parameters not suitable for the application. When the application calls the VIDIOC_S_FMT ioctl with a pointer to a v4l2_format structure the driver checks and adjusts the parameters against hardware abilities. Drivers should not return an error code unless the input is ambiguous, this is a mechanism to fathom device capabilities and to approach parameters acceptable for both the application and driver. On success the driver may program the hardware, allocate resources and generally prepare for data exchange. Finally the VIDIOC_S_FMT ioctl returns the current format parameters as VIDIOC_G_FMT does. Very simple, inflexible devices may even ignore all input and always return the default parameters. However all V4L2 devices exchanging data with

the application must implement the VIDIOC_G_FMT and VIDIOC_S_FMT ioctl. When the requested buffer type is not supported drivers return an EINVAL error code on a VIDIOC_S_FMT attempt. When I/O is already in progress or the resource is not available for other reasons drivers return the EBUSY error code.

The VIDIOC_TRY_FMT ioctl is equivalent to VIDIOC_S_FMT with one exception: it does not change driver state. It can also be called at any time, never returning EBUSY. This function is provided to negotiate parameters, to learn about hardware limitations, without disabling I/O or possibly time consuming hardware preparations. Although strongly recommended drivers are not required to implement this ioctl.

Table 1. struct v4l2_format

enum v4l2_buf_type	*type*		Type of the data stream.
union	*fmt*		
	struct v4l2_pix_format	*pix*	Definition of an image format, used by video capture and output devices. (SPCT6100 use this field)
	struct v4l2_window	*win*	Definition of an overlaid image, used by video overlay devices.
	struct v4l2_vbi_format	*vbi*	Raw VBI capture or output parameters. Used by raw VBI capture and output devices.
	struct v4l2_sliced_vbi_format	*sliced*	Sliced VBI capture or output parameters. Used by sliced VBI capture and output devices.
	__u8	*raw_data*[200]	Place holder for future extensions and custom (driver defined) formats with *type* V4L2_BUF_TYPE_PRIVATE and higher.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EBUSY

The data format cannot be changed at this time, for example because I/O is already in progress.

EINVAL

The struct v4l2_format *type* field is invalid, the requested buffer type not supported, or VIDIOC_TRY_FMT was called and is not supported with this buffer type.

ioctl VIDIOC_G_INPUT, VIDIOC_S_INPUT

Name

VIDIOC_G_INPUT, VIDIOC_S_INPUT — Query or select the current video input channel

Synopsis

```
int ioctl(int fd, int request, int *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_G_INPUT, VIDIOC_S_INPUT

argp

Description

To query the current video input channel applications call the VIDIOC_G_INPUT ioctl with a pointer to an integer where the driver stores the number of the input, as in the struct v4l2_input *index* field. This ioctl will fail only when there are no video input channel, returning EINVAL.

To select a video input channel applications store the number of the desired channel in an integer and call the VIDIOC_S_INPUT ioctl with a pointer to this integer. Side effects are possible. For example inputs may support different video standards, so the driver may implicitly switch the current standard. It is good practice to select an channel before querying or negotiating any other parameters.

Information about video input channel is available using the VIDIOC_ENUMINPUT ioctl.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The number of the video input is out of bounds, or there are no video inputs at all and this ioctl is not supported.

EBUSY

I/O is in progress, the input cannot be switched.

ioctl VIDIOC_G_STD, VIDIOC_S_STD

Name

VIDIOC_G_STD, VIDIOC_S_STD — Query or select the video standard of the current input

Synopsis

```
int ioctl(int fd, int request, v4l2_std_id *argp);
```

```
int ioctl(int fd, int request, const v4l2_std_id *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_G_STD, VIDIOC_S_STD

argp

Description

To query and select the current video standard applications use the VIDIOC_G_STD and VIDIOC_S_STD ioctls which take a pointer to a v4l2_std_id type as argument. VIDIOC_G_STD can return a single flag or a set of flags as in struct v4l2_standard field *id*. The flags must be unambiguous such that they appear in only one enumerated v4l2_standard structure.

VIDIOC_S_STD accepts one or more flags, being a write-only ioctl it does not return the actual new standard as VIDIOC_G_STD does. When no flags are given or the current input does not support the requested standard the driver returns an EINVAL error code. When the standard set is ambiguous drivers may return EINVAL or choose any of the requested standards.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

This ioctl is not supported, or the VIDIOC_S_STD parameter was unsuitable.

ioctl VIDIOC_QBUF, VIDIOC_DQBUF

Name

VIDIOC_QBUF, VIDIOC_DQBUF — Exchange a buffer with the driver

Synopsis

```
int ioctl(int fd, int request, struct v4l2_buffer *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_QBUF, VIDIOC_DQBUF

argp

Description

Applications call the VIDIOC_QBUF ioctl to enqueue an empty buffer in the driver's incoming queue. The semantics depend on the selected I/O method.

To enqueue a memory mapped buffer applications set the *type* field of a struct v4l2_buffer to the same buffer type as previously struct v4l2_format *type* and struct v4l2_requestbuffers *type*, the *memory* field to V4L2_MEMORY_MMAP and the *index* field. Valid index numbers range from zero to the number of buffers allocated with VIDIOC_REQBUFS (struct v4l2_requestbuffers *count*) minus one. The contents of the struct v4l2_buffer returned by a VIDIOC_QUERYBUF ioctl will do as well.

Applications call the VIDIOC_DQBUF ioctl to dequeue a filled buffer from the driver's outgoing queue. They just set the *type* and *memory* fields of a struct v4l2_buffer as above, when VIDIOC_DQBUF is called with a pointer to this structure the driver fills the remaining fields or returns an error code.

By default VIDIOC_DQBUF returns immediately with an EAGAIN error code when no buffer is available.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EAGAIN

Non-blocking I/O has been selected using `O_NONBLOCK` and no buffer was in the outgoing queue.

EINVAL

The buffer *`type`* is not supported, or the *`index`* is out of bounds, or no buffers have been allocated yet, or the *`userptr`* or *`length`* are invalid.

ENOMEM

Not enough physical or virtual memory was available to enqueue a user pointer buffer.

EIO

`VIDIOC_DQBUF` failed due to an internal error. Can also indicate temporary problems like signal loss. Note the driver might dequeue an (empty) buffer despite returning an error, or even stop capturing.

ioctl VIDIOC_QUERYBUF

Name

VIDIOC_QUERYBUF — Query the status of a buffer

Synopsis

```
int ioctl(int fd, int request, struct v4l2_buffer *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_QUERYBUF

argp

Description

This ioctl is part of the memory mapping I/O method. It can be used to query the status of a buffer at any time after buffers have been allocated with the VIDIOC_REQBUFS ioctl.

Applications set the *type* field of a struct v4l2_buffer to the same buffer type as previously struct v4l2_format *type* and struct v4l2_requestbuffers *type*, and the *index* field. Valid index numbers range from zero to the number of buffers allocated with VIDIOC_REQBUFS (struct v4l2_requestbuffers *count*) minus one. After calling VIDIOC_QUERYBUF with a pointer to this structure drivers return an error code or fill the rest of the structure.

In the *flags* field the V4L2_BUF_FLAG_MAPPED, V4L2_BUF_FLAG_QUEUED and V4L2_BUF_FLAG_DONE flags will be valid. The *memory* field will be set to V4L2_MEMORY_MMAP, the *m.offset* contains the offset of the buffer from the start of the device memory, the *length* field its size. The driver may or may not set the remaining fields and flags, they are meaningless in this context.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The buffer *type* is not supported, or the *index* is out of bounds.

ioctl VIDIOC_QUERYCAP

Name

VIDIOC_QUERYCAP — Query device capabilities

Synopsis

```
int ioctl(int fd, int request, struct v4l2_capability *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_QUERYCAP

argp

Description

All V4L2 devices support the VIDIOC_QUERYCAP ioctl. It is used to identify kernel devices compatible with this specification and to obtain information about driver and hardware capabilities. The ioctl takes a pointer to a struct v4l2_capability which is filled by the driver. When the driver is not compatible with this specification the ioctl returns an EINVAL error code.

Table 1. struct v4l2_capability

__u8	*driver*[16]	Name of the driver, a unique NUL-terminated ASCII string. For example: "SPCT6100-V4L2-L". Driver specific applications can use this information to verify the driver identity. It is also useful to work around known bugs, or to identify drivers in error reports. The driver version is stored in the *version* field.

__u8	*card*[32]	Name of the device, a NUL-terminated ASCII string. For example: "SPCT6100". One driver may support different brands or models of video hardware. This information is intended for users, for example in a menu of available devices. Since multiple TV cards of the same brand may be installed which are supported by the same driver, this name should be combined with the character device file name (e. g. `/dev/video2`) or the *bus_info* string to avoid ambiguities.
__u8	*bus_info*[32]	Location of the device in the system, a NUL-terminated ASCII string. For example: "PCI Slot 4". This information is intended for users, to distinguish multiple identical devices. If no such information is available the field may simply count the devices controlled by the driver, or contain the empty string (*bus_info*[0] = 0).
__u32	*version*	Version number of the driver. Together with the *driver* field this identifies a particular driver. The version number is formatted using the `KERNEL_VERSION` macro.
__u32	*capabilities*	Device capabilities.
__u32	*reserved*[4]	Reserved for future extensions. Drivers must set this array to zero.

Table 2. Device Capabilities Flags

`V4L2_CAP_VIDEO_CAPTURE`	0x00000001	The device supports the Video Capture interface.
`V4L2_CAP_STREAMING`	0x04000000	The device supports the streaming I/O method. [a]

Notes:
a. This table only list those capability SPCT6100 focus on.

Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The device is not compatible with this specification.

ioctl VIDIOC_QUERYCTRL

Name

VIDIOC_QUERYCTRL — Enumerate controls items

Synopsis

```
int ioctl(int fd, int request, struct v4l2_queryctrl *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_QUERYCTRL

argp

Description

To query the attributes of a control applications set the *id* field of a struct v4l2_queryctrl and call the VIDIOC_QUERYCTRL ioctl with a pointer to this structure. The driver fills the rest of the structure or returns an EINVAL error code when the *id* is invalid.

It is possible to enumerate controls by calling VIDIOC_QUERYCTRL with successive *id* values starting from V4L2_CID_BASE up to and exclusive V4L2_CID_BASE_LASTP1. Drivers may return EINVAL if a control in this range is not supported. Further applications can enumerate private controls, which are not defined in this specification, by starting at V4L2_CID_PRIVATE_BASE and incrementing *id* until the driver returns EINVAL.

In both cases, when the driver sets the V4L2_CTRL_FLAG_DISABLED flag in the *flags* field this control is permanently disabled and should be ignored by the application.[1]

When the application ORs *id* with V4L2_CTRL_FLAG_NEXT_CTRL the driver returns the next supported control, or EINVAL if there is none. Drivers which do not support this flag yet always return EINVAL.

Table 1. struct v4l2_queryctrl

__u32	*id*	Identifies the control, set by the application. When the ID is ORed with V4L2_CTRL_FLAG_NEXT_CTRL the driver clears the flag and returns the first control with a higher ID. Drivers which do not support this flag yet always return an EINVAL error code.
enum v4l2_ctrl_type	*type*	Type of control.
__u8	*name*[32]	Name of the control, a NUL-terminated ASCII string. This information is intended for the user.
__s32	*minimum*	Minimum value, inclusive. This field gives a lower bound for `V4L2_CTRL_TYPE_INTEGER` controls. It may not be valid for any other type of control, including `V4L2_CTRL_TYPE_INTEGER64` controls. Note this is a signed value.
__s32	*maximum*	Maximum value, inclusive. This field gives an upper bound for `V4L2_CTRL_TYPE_INTEGER` controls and the highest valid index for `V4L2_CTRL_TYPE_MENU` controls. It may not be valid for any other type of control, including `V4L2_CTRL_TYPE_INTEGER64` controls. Note this is a signed value.
__s32	*step*	This field gives a step size for `V4L2_CTRL_TYPE_INTEGER` controls. It may not be valid for any other type of control, including `V4L2_CTRL_TYPE_INTEGER64` controls. Generally drivers should not scale hardware control values. It may be necessary for example when the *name* or *id* imply a particular unit and the hardware actually accepts only multiples of said unit. If so, drivers must take care values are properly rounded when scaling, such that errors will not accumulate on repeated read-write cycles. This field gives the smallest change of an integer control actually affecting hardware. Often the information is needed when the user can change controls by keyboard or GUI buttons, rather than a slider. When for example a hardware register accepts values 0-511 and the driver reports 0-65535, step should be 128. Note although signed, the step value is supposed to be always positive.

__s32	*default_value*	The default value of a V4L2_CTRL_TYPE_INTEGER, _BOOLEAN or _MENU control. Not valid for other types of controls. Drivers reset controls only when the driver is loaded, not later, in particular not when the func-open; is called.
__u32	*flags*	Control flags.
__u32	*reserved*[2]	Reserved for future extensions. Drivers must set the array to zero.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_queryctrl *id* is invalid. The struct v4l2_querymenu *id* or *index* is invalid.

Notes

1. V4L2_CTRL_FLAG_DISABLED was intended for two purposes: Drivers can skip predefined controls not supported by the hardware (although returning EINVAL would do as well), or disable predefined and private controls after hardware detection without the trouble of reordering control arrays and indices (EINVAL cannot be used to skip private controls because it would prematurely end the enumeration).

ioctl VIDIOC_QUERYSTD

Name

VIDIOC_QUERYSTD — Sense the video standard received by the current input

Synopsis

```
int ioctl(int fd, int request, v4l2_std_id *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_QUERYSTD

argp

Description

The hardware may be able to detect the current video standard automatically. To do so, applications call VIDIOC_QUERYSTD with a pointer to a v4l2_std_id type. The driver stores here a set of candidates, this can be a single flag or a set of supported standards if for example the hardware can only distinguish between 50 and 60 Hz systems. When detection is not possible or fails, the set must contain all standards supported by the current video input or output.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

This ioctl is not supported.

ioctl VIDIOC_REQBUFS

Name

VIDIOC_REQBUFS — Initiate Memory Mapping I/O

Synopsis

```
int ioctl(int fd, int request, struct v4l2_requestbuffers *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_REQBUFS

argp

Description

This ioctl is used to initiate memory mapped I/O. Memory mapped buffers are located in device memory and must be allocated with this ioctl before they can be mapped into the application's address space.

To allocate device buffers applications initialize three fields of a v4l2_requestbuffers structure. They set the *type* field to the respective stream or buffer type, the *count* field to the desired number of buffers, and *memory* must be set to V4L2_MEMORY_MMAP. When the ioctl is called with a pointer to this structure the driver attempts to allocate the requested number of buffers and stores the actual number allocated in the *count* field. It can be smaller than the number requested, even zero, when the driver runs out of free memory. A larger number is possible when the driver requires more buffers to function correctly. When memory mapping I/O is not supported the ioctl returns an EINVAL error code.

Applications can call VIDIOC_REQBUFS again to change the number of buffers, however this cannot succeed when any buffers are still mapped. A *count* value of zero frees all buffers, after aborting or finishing any DMA in progress, an implicit VIDIOC_STREAMOFF.

Table 1. struct v4l2_requestbuffers

__u32	*count*	The number of buffers requested or granted. This field is only used when *memory* is set to V4L2_MEMORY_MMAP.

enum v4l2_buf_type	*type*	Type of the stream or buffers, this is the same as the struct v4l2_format *type* field.
enum v4l2_memory;	*memory*	For SPCT6100 applications should set this field to V4L2_MEMORY_MMAP.
__u32	*reserved*[2]	A place holder for future extensions and custom (driver defined) buffer types V4L2_BUF_TYPE_PRIVATE and higher.

Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EBUSY

The driver supports multiple opens and I/O is already in progress, or reallocation of buffers was attempted although one or more are still mapped.

EINVAL

The buffer type (*type* field) or the requested I/O method (*memory*) is not supported.

ioctl VIDIOC_STREAMON, VIDIOC_STREAMOFF

Name

VIDIOC_STREAMON, VIDIOC_STREAMOFF — Start or stop streaming I/O

Synopsis

```
int ioctl(int fd, int request, const int *argp);
```

Arguments

fd

File descriptor returned by open.

request

VIDIOC_STREAMON, VIDIOC_STREAMOFF

argp

Description

The VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl start and stop the capture during streaming (memory mapping) I/O.

Specifically the capture hardware is disabled and no input buffers are filled (if there are any empty buffers in the incoming queue) until VIDIOC_STREAMON has been called. Accordingly the output hardware is disabled, no video signal is produced until VIDIOC_STREAMON has been called. The ioctl will succeed only when at least one output buffer is in the incoming queue.

The VIDIOC_STREAMOFF ioctl, apart of aborting or finishing any DMA in progress, unlocks any user pointer buffers locked in physical memory, and it removes all buffers from the incoming and outgoing queues. That means all images captured but not dequeued yet will be lost, likewise all images enqueued for output but not transmitted yet. I/O returns to the same state as after calling VIDIOC_REQBUFS and can be restarted accordingly.

Both ioctls take a pointer to an integer, the desired buffer or stream type. This is the same as struct v4l2_requestbuffers *type*.

Note applications can be preempted for unknown periods right before or after the VIDIOC_STREAMON or VIDIOC_STREAMOFF calls, there is no notion of starting or stopping "now". Buffer timestamps can be used to synchronize with other events.

Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

Streaming I/O is not supported, the buffer *`type`* is not supported, or no buffers have been allocated (memory mapping) yet.

Appendix A. Demos

All demos are in demo packages including command line demos and GUI demos.

Appendix B. Audio Decoding

Introduction to how to use the audio decoding library. To be done.