How to build up UVC Driver on Linux Ver.1.1

The iP297X is compatible with UVC(USB Video Class) specification. In Linux, you need to modify UVC Driver to make the iP297X to work.

- There is a C file called UVC_driver.c in Kernel source. Find the static struct usb_device_id uvc_ids[]. Add the following into the structure.

```
/*iPassion USB Web Camera */
 { .match_flags    = USB_DEVICE_ID_MATCH_DEVICE
                   | USB_DEVICE_ID_MATCH_INT_INFO,
   .idVendor       = 0x1B3B,
   .idProduct      = 0x2970,/*If you use iP2977, then type "0x2977" */
   .bInterfaceClass    = USB_CLASS_VIDEO,
   .bInterfaceSubClass     = 1,
   .bInterfaceProtocol     = 0,
   .driver_info          = UVC_QUIRK_PROBE_MINMAX
         | UVC_QUIRK_IGNORE_SELECTOR_UNIT}
```

- Improve compatibility : Find the C file : uvc_video.c and modify the following functions

1) static int uvc_video_decode_start function

```
static int uvc_video_decode_start(struct uvc_streaming *stream,
        struct uvc_buffer *buf, const __u8 *data, int len)
{
    static __u8 fid;                                    // modify to static style

    /* Sanity checks:
     * - packet must be at least 2 bytes long
     * - bHeaderLength value must be at least 2 bytes (see above)
     * - bHeaderLength value can't be larger than the packet size.
     */
    if (len < 2 || data[0] < 2 || data[0] > len)
        return -EINVAL;

    /* Skip payloads marked with the error bit ("error frames"). */
    if (data[1] & UVC_STREAM_ERR) {
        uvc_trace(UVC_TRACE_FRAME, "Dropping payload (error bit "
                "set).\n");
        return -ENODATA;
    }

    if ( len >= 16 )            // have data in buffer
    {
        // data must judge from data[12]，data before data[12] is for package only
        if ( (data[12]==0xFF && data[13]==0xD8 && data[14]==0xFF) ||
```

```
            (data[12]==0xD8 && data[13]==0xFF && data[14]==0xC4))
        {
            if(stream->last_fid)
                fid &= ~UVC_STREAM_FID;
            else
                fid |= UVC_STREAM_FID;
        }
}
//fid = data[1] & UVC_STREAM_FID;
// ================================================================================

/* Increase the sequence number regardless of any buffer states, so
 * that discontinuous sequence numbers always indicate lost frames.
 */
if (stream->last_fid != fid)
        stream->sequence++;

/* Store the payload FID bit and return immediately when the buffer is
 * NULL.
 */
if (buf == NULL) {
        stream->last_fid = fid;
        return -ENODATA;
}

/* Synchronize to the input stream by waiting for the FID bit to be
 * toggled when the the buffer state is not UVC_BUF_STATE_ACTIVE.
 * stream->last_fid is initialized to -1, so the first isochronous
 * frame will always be in sync.
 *
 * If the device doesn't toggle the FID bit, invert stream->last_fid
 * when the EOF bit is set to force synchronisation on the next packet.
 */
if (buf->state != UVC_BUF_STATE_ACTIVE) {
        struct timespec ts;

        if (fid == stream->last_fid) {
            uvc_trace(UVC_TRACE_FRAME, "Dropping payload (out of "
                    "sync).\n");
            if ((stream->dev->quirks & UVC_QUIRK_STREAM_NO_FID) &&
                (data[1] & UVC_STREAM_EOF))
                stream->last_fid ^= UVC_STREAM_FID;
            return -ENODATA;
        }

        if (uvc_clock_param == CLOCK_MONOTONIC)
            ktime_get_ts(&ts);
        else
            ktime_get_real_ts(&ts);

        buf->buf.sequence = stream->sequence;
        buf->buf.timestamp.tv_sec = ts.tv_sec;
        buf->buf.timestamp.tv_usec = ts.tv_nsec / NSEC_PER_USEC;
```

```
                /* TODO: Handle PTS and SCR. */
                buf->state = UVC_BUF_STATE_ACTIVE;
        }

        /* Mark the buffer as done if we're at the beginning of a new frame.
         * End of frame detection is better implemented by checking the EOF
         * bit (FID bit toggling is delayed by one frame compared to the EOF
         * bit), but some devices don't set the bit at end of frame (and the
         * last payload can be lost anyway). We thus must check if the FID has
         * been toggled.
         *
         * stream->last_fid is initialized to -1, so the first isochronous
         * frame will never trigger an end of frame detection.
         *
         * Empty buffers (bytesused == 0) don't trigger end of frame detection
         * as it doesn't make sense to return an empty buffer. This also
         * avoids detecting end of frame conditions at FID toggling if the
         * previous payload had the EOF bit set.
         */
        if (fid != stream->last_fid && buf->buf.bytesused != 0) {
                uvc_trace(UVC_TRACE_FRAME, "Frame complete (FID bit "
                                "toggled).\n");
                buf->state = UVC_BUF_STATE_READY;
                return -EAGAIN;
        }

        stream->last_fid = fid;

        return data[0];
}
```

2) static void uvc_video_decode_data function

```
static void uvc_video_decode_data(struct uvc_streaming *stream,
                struct uvc_buffer *buf, const __u8 *data, int len)
{
        struct uvc_video_queue *queue = &stream->queue;
        unsigned int maxlen, nbytes;
        void *mem;

        if (len <= 0)
                return;

        /* Copy the video data to the buffer. */
        maxlen = buf->buf.length - buf->buf.bytesused;
        mem = queue->mem + buf->buf.m.offset + buf->buf.bytesused;
        nbytes = min((unsigned int)len, maxlen);
        memcpy(mem, data, nbytes);
        buf->buf.bytesused += nbytes;

// Declaring an index variable with special style is must when modifying video data. Then, accessing data of memory is correct .
        unsigned char *point_mem;
        static unsigned char *mem_temp = NULL;
```

```
        //    Initialize temporarily memory
        static unsigned int nArrayTemp_Size = 1000;
        if(mem_temp == NULL){
            mem_temp = kmalloc(nArrayTemp_Size, GFP_KERNEL);
        }
        else if(nArrayTemp_Size <= nbytes){          // Allocating memory size is need when receiving data length is longer than the previous.
            kfree(mem_temp);
            nArrayTemp_Size += 500;
            kmalloc(nArrayTemp_Size, GFP_KERNEL);
        }
        memset(mem_temp, 0x00, nArrayTemp_Size);

        // Point to memory location of data storage
        point_mem = (unsigned char *)mem;

        if( *(point_mem) == 0xD8 && *(point_mem + 1) == 0xFF && *(point_mem + 2) == 0xC4){
            memcpy( mem_temp + 1, point_mem, nbytes);
            mem_temp[0] = 0xFF;
            memcpy( point_mem, mem_temp, nbytes + 1);
        }
        //
================================================================================
        /* Complete the current frame if the buffer size was exceeded. */
        if (len > maxlen) {
            uvc_trace(UVC_TRACE_FRAME, "Frame complete (overflow).\n");
            buf->state = UVC_BUF_STATE_READY;
        }
}
```

- If you want to adjust image ,like Brightness, Contrast and Saturation etc. , you need to use ioctl to control sensor. The following is the sample code.

```
/*************************************************************
#            I2C and Register I/O Sample code                                    #
#This package work with the iPassion UVC based webcams with the mjpeg feature. #
#.                                                                                                    #
*************************************************************/

//=================================================//
#define V4L2_CID_REG_RELATIVE_IPASSION      0x0A046D01
#define V4L2_CID_I2C_RELATIVE_IPASSION    0x0A046D02
#define V4L2_CID_MOTION_RELATIVE_IPASSION    0x0A046D03


#define UVC_GUID_IPASSION_CONTROL {0x3a, 0xab, 0x91, 0x99, 0xef, 0xb2, 0xc9, 0x48, 0x8f,
0xe9, 0x8f, 0xe3, 0x63, 0x47, 0x71, 0xd0}
#define VC_EXTENSION_UNIT       4
#define XU_REGISTER 1
#define XU_I2C       4
#define XU_MOTIONCONTROL                    3
#define EXT_COMMAND_COUNT  3
/* iPassion    controls */
static struct uvc_xu_control_info xu_ctrls[] = {
   {
      .entity      = UVC_GUID_IPASSION_CONTROL,
      .selector = XU_REGISTER,
      .index       = 0,
      .size         = 3,
      .flags       = UVC_CONTROL_SET_CUR | UVC_CONTROL_GET_CUR
   },
   {
      .entity      = UVC_GUID_IPASSION_CONTROL,
      .selector = XU_I2C,
      .index       = 1,
      .size         = 5,
      .flags       = UVC_CONTROL_SET_CUR | UVC_CONTROL_GET_CUR
   },
```

```
    {
        .entity     = UVC_GUID_IPASSION_CONTROL,
        .selector = XU_MOTIONCONTROL,
        .index       = 2,
        .size         = 3,
        .flags       = UVC_CONTROL_SET_CUR | UVC_CONTROL_GET_CUR
    }
};

/* mapping for Pan/Tilt/Focus */
static struct uvc_xu_control_mapping xu_mappings[] = {
    {
        .id              = V4L2_CID_REG_RELATIVE_IPASSION,
        .name           = "Reg (relative)",
        .entity       = UVC_GUID_IPASSION_CONTROL,
        .selector    = XU_REGISTER,
        .size          = 24,
        .offset       = 0,
        .v4l2_type = V4L2_CTRL_TYPE_INTEGER,
        .data_type = UVC_CTRL_DATA_TYPE_SIGNED
    },
    {
        .id              = V4L2_CID_I2C_RELATIVE_IPASSION,
        .name           = "I2C (relative)",
        .entity       = UVC_GUID_IPASSION_CONTROL,
        .selector    = XU_I2C,
        .size          = 40,
        .offset       = 0,
        .v4l2_type = V4L2_CTRL_TYPE_INTEGER,
        .data_type = UVC_CTRL_DATA_TYPE_SIGNED
    },
    {
        .id              = V4L2_CID_MOTION_RELATIVE_IPASSION,
        .name           = "Motion (relative)",
        .entity       = UVC_GUID_IPASSION_CONTROL,
        .selector    = XU_MOTIONCONTROL,
        .size          = 32,
        .offset       = 0,
        .v4l2_type = V4L2_CTRL_TYPE_INTEGER,
```

```
        .data_type = UVC_CTRL_DATA_TYPE_SIGNED
    }
};

int ReadI2C(int fd, unsigned char *data)//read I2C
{
    int err;
    struct uvc_xu_control xctrl;
    xctrl.unit = VC_EXTENSION_UNIT;
    xctrl.selector=XU_I2C;
    xctrl.size=5;
    xctrl.data=data;

    if ((err = ioctl(vd->fd, UVCIOC_CTRL_SET, &xctrl)) < 0) {
        printf("ioctl XU_I2C error\n");
        return -1;
    }
    else
        printf("ioctl XU_I2C Success\n");
    if ((err = ioctl(vd->fd, UVCIOC_CTRL_GET, &xctrl)) < 0) {
        printf("ioctl XU_I2C error\n");
        return -1;
    }
    else
        printf("ioctl XU_I2C Success\n");

    return 0;
}
int WriteI2C(int fd, unsigned char *data)//write I2C
{
    int err;
    struct uvc_xu_control xctrl;
    xctrl.unit = VC_EXTENSION_UNIT;
    xctrl.selector=XU_I2C;
    xctrl.size=5;
    xctrl.data=data;

    if ((err = ioctl(vd->fd, UVCIOC_CTRL_SET, &xctrl)) < 0) {
        printf("ioctl XU_I2C error\n");
```

```
            return -1;
        }
        else
            printf("ioctl XU_I2C Success\n");
        if ((err = ioctl(vd->fd, UVCIOC_CTRL_GET, &xctrl)) < 0) {
            printf("ioctl XU_I2C error\n");
            return -1;
        }
        else
            printf("ioctl XU_I2C Success\n");

        return 0;
}
int WriteRegister(int fd, unsigned char *data)
{
        int err;
        struct uvc_xu_control xctrl;
        xctrl.unit = VC_EXTENSION_UNIT;
        xctrl.selector=XU_REGISTER;
        xctrl.size = 3;
        xctrl.data=data;
        if ((err = ioctl(fd, UVCIOC_CTRL_SET, &xctrl)) < 0) {
            printf("ioctl XU_REGISTER error\n");
            return -1;
        }
        else
            printf("ioctl XU_REGISTER Success\n");
        return 0;
}
int ReadRegister(int fd, unsigned char *data)
{
        int err;
        struct uvc_xu_control xctrl;
        xctrl.unit = VC_EXTENSION_UNIT;
        xctrl.selector=XU_REGISTER;
        xctrl.size=3;
        xctrl.data=data;

        if ((err = ioctl(vd->fd, UVCIOC_CTRL_SET, &xctrl)) < 0) {
```

```
            printf("ioctl XU_REGISTER error\n");
            return -1;
        }
        else
            printf("ioctl XU_REGISTER Read Success\n");
        if ((err = ioctl(vd->fd, UVCIOC_CTRL_GET, &xctrl)) < 0) {
            printf("ioctl XU_REGISTER error\n");
            return -1;
        }
        else
            printf("ioctl XU_REGISTER Read Success\n");
        return 0;
}
int InitExtensionCommand(int fd)
{

            int err,index;

        // Add XU Selector
        for(index =0;index < EXT_COMMAND_COUNT;index++)
        {
            if ((err = ioctl(fd, UVCIOC_CTRL_ADD, &xu_ctrls[index])) < 0) {
                printf("ioctl UVCIOC_CTRL_ADD error\n");
                return -1;
            }
            else
                printf("ioctl UVCIOC_CTRL_ADD success\n");
            if ((err = ioctl(fd,    UVCIOC_CTRL_MAP, &xu_mappings[index])) < 0) {
                printf("ioctl UVCIOC_CTRL_MAP error\n");
                return -1;
            }
            else
                printf("ioctl UVCIOC_CTRL_MAP success\n");
        }

        return 0;
}

void main()
```

```
{
    int fd;
    unsigned char data[5];
    fd = open(...);//open device
    //Add Extension Unit Command into UVC Driver
    InitExtensionCommand(fd);
    //Read I2C
    data[0] =0x81;//Status
    data[1] =0x40;//Sensor ID
    data[2] =0x0c;//Sensor Reg. Address
    data[3] =0x00;
    ReadI2C(fd, &data);//read I2C
    //Write I2C
    data[0] =0x01;//Status
    data[1] =0x40;//Sensor ID
    data[2] =0x0c;//Sensor Reg. Address
    data[3] =0x00;
    WriteI2C(fd, &data);//write I2C
    //Read Register
    data[0] = 0x01;//Read Status
    data[1] = 0xac;//ip297x Register Address
    data[2] = 0x00;//ip297x Register Value
    ReadRegister(fd, &data)
    //Write Register
    data[0] = 0x00;//Write Status
    data[1] = 0xac;//ip297x Register Address
    data[2] = 0x00;//ip297x Register Value
    WriteRegister(fd, &data);
}
```