pxc 0.80

A device driver for the PXC200 frame grabber.
September 2001

by Alessandro Rubini (`rubini@linux.it`) and others

The pxc package

This manual documents the 0.80 release of the *pxc* device driver and associated user utilities, the driver works with the PXC200 frame grabber produced by ImageNation. Version 0.27 and later of the driver work with both the "F" and "L" variants of the grabber. Version 0.28 and later also work with generic Bt848/849/878 devices. Version 0.80 and later also support the PXC200A (both "F" and "L") family of grabbers by ImageNation.

The *pxc* package is made up of a device driver implementing *read* and *ioctl* operations and a few user-space demonstration applications. The sample applications are also used by the author in the real world, especially in relation to the *bisce* package (`http://arcana.linux.it/software/bisce`).

1 Incompatibilities

There are a few incompatibilities between this version on the driver and earlier versions. You can skip this section if you are not a previous user of *pxc*.

Buffer Management

After version 0.28 buffer management has been introduced. This means that the device doesn't overwrite the same DMA image over and over. While open/read/close applications won't notice any difference, if you opened a pgm or ppm device once to read it over and over you will always find the same image. The fix here is adding these two lines after acquisition of each image:

```
struct Px_BufControl bufcontrol;
ioctl(grab_fd, PX_IOCINFOBUF, &bufcontrol);
ioctl(grab_fd, PX_IOCDONEBUF, &bufcontrol);
```

See Chapter 5 [Buffer Management], page 4.

Sequence Acquisition

Acquisition of sequences has been disabled with the introduction of buffer management. I plan to reintroduce it implemented in the right way. I don't think this affects you unless you are running something like *bisce* (`http://ar.linux.it/software/#bisce`).

2 Overview

The device driver doesn't support all of the device features, as I only wrote it to scratch my own itches. The implemented features should satisfy most every-day needs, though. Feel free to ask for features if you need them: I'll feel more inclined to write code if someone is going to use it (and possibly pay for such development). I thank ImageNation for funding part of my work, this includes buffer management, trigger support and PXC200A support.

This driver works with Linux-2.2 and 2.4. Support for 2.0 has been removed after version 0.28 As for 2.0, I didn't remove the portability code from the source files, if the need arises I might backport the features of Linux-2.2 I have been using after release 0.28 of this driver.

The latest releases are always available from either of this places:

```
ftp://arcana.linux.it/pub/pxc             (all versions)
ftp://ftp.systemy.it/pub/develop          (original place)
ftp://ftp.linux.it/pub/People/rubini      (nightly mirror of above)
```

There are two mailing lists devoted to this project. The lists are called `pxc@lists.linux.it` (where technical discussions are held) and `pxc-announce@lists.linux.it` (a place to announce

new driver releases). Who subscribes to the former list doesn't need to subscribe to the latter one as I always post announcements to both lists.

In order to get subscribed to either list, try the following commands:

```
echo subscribe | mail pxc-request@lists.linux.it
echo subscribe | mail pxc-announce-request@lists.linux.it
```

2.1 Device Special files

Basically, the driver creates a few streams that can be used to retrieve image data. The devices are created for four grabbers; if you have more devices, you need to create the nodes for them in /dev (there are no limits in the driver, as allocation of data structures is dynamic).

Most accesses to the driver need a DMA buffer (see Chapter 3 [The DMA Buffer], page 3).

The following devices are used to access the first grabber (nr. 0):

`/dev/pxc0`
: A binary device, returning a continuous stream of low resolution (single-field) images. To avoid jitter across successive images, only the even field is grabbed. Continuous acquisition can be disabled via ioctl() commands, so that you can acquire a specific number of images, either using a hardware trigger or software control. A DMA buffer is needed in order to be able to read this device.

`/dev/pxc0H`
: The high-resolution binary node: it returns a continuous stream of high resolution (two fields) images. Ioctl() can be used to configure this device, like /dev/pxc0. A DMA buffer is needed to access this device.

`/dev/pxc0pgm`
: The *pgm* device associated to the previous one. When being read, it returns a single *pgm* image, in low resolution. You can just invoke "`display /pxc0pgm`", or anything equivalent, to preview your data. Once again, DMA is needed to use this device. By default, DMA is stopped as soon as one field has been acquired. In version 0.27 and earlier of the driver, the driver continued overwriting the DMA buffer with new images; therefore you could get slices of different images if reading slowly. Anyway, you can set `PX_FLAG_CONTACQ` if you need continuous acquisition.

`/dev/pxc0Hpgm`
: The *pgm* node that returns an high resolution (two fields) gray-scale image. By default, DMA is stopped as soon as one frame has been acquired. In version 0.27 and earlier of the driver, the driver continued overwriting the DMA buffer with new images; therefore you could get slices of different images if reading slowly.

`/dev/pxc0ctl`
: A control device. Opening it doesn't allocate the DMA buffer. Reading returns the Bt848 registers, ioctl() can be used as well on this device.

`/dev/pxc0rgb`
`/dev/pxc0Hrgb`
: These devices return a continuous stream of RGB color data, low and high resolution, respectively. Each pixel is identified by three data bytes. They require a DMA buffer, and continuous acquisition is enabled while reading either of these files.

`/dev/pxc0ppm`
`/dev/pxc0Hppm`
: Color "portable pixmap" devices, similar in concept to the *pgm* entry points. The require a DMA buffer. Only one field or one frame is acquired.

`/dev/pxc0bgr`
`/dev/pxc0Hbgr`

The device transfers data in BGR format instead of RGB. While both the `rgb` and `ppm` devices take care of pixel swapping, these devices do not. They therefore provide faster access. Note that when you issue mmap() on a color device you will always access BGR data, as the RGB conversion is only performed by the implementation of read().

All of the devices except the control node can be accessed via mmap().

3 The DMA Buffer

One of the most compelling problems with any DMA-capable device is the allocation of a suitable memory buffer. The pxc200 driver uses a module of mine, called "allocator". The allocator is able to use high memory (above the one used in normal operation) for DMA allocation.

To prevent the kernel for using high memory, so that it remains available for DMA, you should pass a command line argument to the kernel. Command line arguments can be passed to Lilo, to Loadlin or to whichever loader you are using (unless it's very poor in design). For Lilo, either use "append=" in /etc/lilo.conf or add command-line arguments to the interactive prompt. For example, I have a 64MB box and reserve two megs for DMA:

In lilo.conf:

```
image = /zImage
label = linux
append = "mem=62M"
```

Or, interactively:

```
LILO: linux mem=62M
```

If you modify your lilo.conf, you need to re-run /sbin/lilo to make the new configuration effective. Note that other boot loaders exist, and each of them allows passing a command line to the kernel. For example, *Grub* allows a commandline to be specified in its *kernel* directive. Please refer to the documentation for your specific boot loader.

There are other ways to reserve a DMA buffer, but they are not (yet) supported. These include the *bigphysarea* patch for 2.0/2.2 and the official 2.3/2.4 API.

4 Installing the package

4.1 Compiling the Package

To compile the driver and associated utilities, try `make`. If needed, try `make install`.

Please note that the installation part is the least tested corner of the package. Also, due to kernel-version issues, you might want to check the `Makefile` and edit a few variables.

If your kernel headers are not in '`/usr/src/linux/include/linux`' and '`/usr/src/linux/include/asm`', please specify an include directory or the kernel directory in the command line of '`make`'. This is what I do:

```
make KERNELDIR=/usr/src/linux-2.2
```

You can also use `INCLUDEDIR` (set, for example, to '`/usr/src/linux-2.2/include`'), but support for `INCLUDEDIR` is only there for backward compatibility and will be removed in later versions.

You can set KERNELDIR or INCLUDEDIR in the environment, if you prefer that to the command line of 'make'.

If you install the Debian package I distribute since version 0.25 of the driver, please check which kernel version the driver is compiled against (I tend to compile against the latest stable Linux kernel, but don't trust this statement). In general, don't trust my Debian packages too much, I only use them for easing myself in maintaining specific installations and are not good enough for general use.

4.2 Loading the Kernel Module

The *pxc_load* script deals with loading the device driver. It looks for the module in the current directory and then in the usual places. The script prints the location where the module is found, so you can at least know what is going on. You are expected to edit the script to configure permission bits and owners of the devices being created (by default anyone can access the frame grabber.

I expect most users will simply run `./pxc_load` from the top-level directory of the package.

Any argument that you pass to the pxc_load command line are passed to *insmod*. This allows load-time configuration of internal variables and, if you feel so inclined, to specify the `-f` option.

Parameters that can be set at load time are `major=`*n*, `buffers=`*n*, and `pll=`*n,n,...*. The `major` parameter is used to force a specific major number for the device (the default is currently 60, and you can use 0 to select a dynamically-assigned major number).

The `pll` argument is used to tell which of the devices has a single quartz crystal and needs use of the internal Bt848A phased locked loop (PLL) to generate a PAL frequency. The PXC device by ImageNation have two crystals, but some other Bt848-based device requires the PLL to be used (if images grabbed from PAL cameras have misaligned lines, then you need the PLL). For example, to use a real PXC200 (`/dev/pxc0`) and a plain-Bt848 device (`/dev/pxc1` I use the parameter "`pll=0,1`". The PLL is automatically used for any device based on the Bt849 and Bt878 chips.

The `buffers` argument default to 2, and is used to select the behaviour in buffer management. If you have reserved enough memory you are strongly urged to use `buffers=3`. See Chapter 5 [Buffer Management], page 4.

4.3 Unloading the Module

The *pxc_unload* script takes care of unloading the driver and cleaning up the `/dev/` directory.

5 Buffer Management

Up to version 0.28, I used a single image buffer for each frame grabber device. This lead to several problems, related with DMA going on at the same time as a user space program is reading the buffer.

After version 0.28, ImageNation (producers of the PXC200 frame grabbers) sponsored development of the *pxc* device driver and I switched to a more flexible approach. This section describes how memory is managed within the device driver, and you may well skip over it if reading one image from '`/dev/pxc0pgm`' is all you need.

The basic idea of asynchronous buffers comes from *Cyclical Asyncronous Buffers*, in *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*, by Giorgio Buttazzo (Kluwer Academic Publishers, Boston, 1997), page 291 through 295.

5.1 Multiple Buffers

When the device is initialized it allocates DMA space in high memory (see Chapter 3 [The DMA Buffer], page 3). This happens the first time it is opened, either the first time after loading the driver or the first time after it has been close by all processes.

The driver allocates space for two buffers by default, although for best operation you'll need three or more buffers. The default can be changed by setting the **buffers** option when loading the driver, so you can choose to always allocate three buffers if this best suites your application (see Section 4.2 [Loading], page 4). I chose to keep the default smaller in order to avoid "not enough memory" errors with setups that worked well with older versions of the driver. The current implementation of the driver refuses to run with less than two buffers, because that environment would require to start and stop DMA according to use of the buffer, and I'd better avoid it if possible.

Each of the buffers is either in-use of free. A buffer can be used by either the device driver or a user-space process. The driver owns a buffer when the frame grabber is actively writing to its memory area, a process owns a buffer when it is reading data from it.

Use of three buffers allows a process to always get hold of valid image data with no delay. One of the buffers is always in use as a DMA target, and one is normally in use by the process. If the process needs to get hold of new image data immediately after releasing a buffer, you'll need three buffer. In this case DMA alternates the two buffers not owned by the process, and the process can always find a free (and up to date) buffer in no time).

In general, you'll need R+W+1 buffers, where "W" is the number of writers (in this case, 1), and "R" is the number of readers (the number of processes that need to access image data). The case with several processes has not yet been tested, though, and the current design doesn't allow more than one process waiting for a new buffer; some issues still need to be addresses, but it's a low priority task.

Note that the driver assigns a buffers to a file rather than to a process, so a process reading image data from different file descriptors incurs in the same untested behaviour outlined above.

Currently, only one process (rather, one file) can be waiting for image data, if another file should wait it will receive `EBUSY` ("Device Busy") instead.

5.2 Buffer Allocation Using Read

The device driver allocates actual buffer from high memory (see Chapter 3 [The DMA Buffer], page 3), so you'll need to have enough high memory according to the number of devices you run and the number of buffers you choose.

During driver activity, one of the buffers is always property of the driver itself, because the device is performing DMA to that buffer. When an interrupt signals that DMA is complete on that buffer, the driver looks for a free buffer where the following frame or field can be stored. If it finds one, the previous buffer is released; if it finds none it reuses the same buffer without releasing it.

When a process is reading image data (using the *read* system call), the driver will give it exclusive access to one buffer. The buffer is released when the read position for the file reaches the end of image data (either because the whole image has been read or because *lseek* is used).

When a *read* system call is performed and no buffer is currently owned by this file, the device driver will select the most recent buffer and mark it as property of this file. If the most recent buffer is the one that has already been read, the process will sleep waiting for a new buffer. This may happen either because there is yet no newer image than the one already processed, or because you are using only two buffers, so the other buffer is currently the target of DMA.

If the `O_NONBLOCK` flag is set on the file and no buffer is owned by it when it calls *read*, the driver returns `-EAGAIN`. The interrupt handler will arrange for the proper image buffer to get assigned to this file.

Similarly, a *select* (or *poll*) system call can be used to express interest in a new image buffer. The file won't be reported as readable before it gets hold of an image buffer.

As soon as an interrupt reports that a new image has been transferred, the device driver will assign the new buffer to the file and will awake the process. DMA will continue to another buffer (the one just released if you are working with two buffers).

If three buffers are used, a process that is slowly processing data will never sleep when calling *read*. A fast process may still sleep if it is done processing an image before the next image is ready.

When a file is closed, its buffer, if any, is released. Extreme care has been used to avoid any race condition in buffer allocation by using proper device locking.

5.3 Mmap

A process can choose to access image data using *mmap* instead of copying it over using *read*. To this aim, it should map from the device driver a memory area big enough to include all image buffers (whose size can be queried via `ioctl(PX_IOCGDMASIZE)`. See Section 5.6 [Ioctl Commands for Buffer Management], page 7.

To proficiently access image buffers using *mmap*, the process must synchronize with the device driver using *ioctl*. It can express interest for a specific image, wait for the image to be available, release the image buffer. Image buffers are identified by offsets within buffer memory area (i.e., offsets from the beginning of the *mmap* base address).

5.4 Detailing Specific Requirements

Warning: this is not yet implemented, some documentation is there but the code is not.

If you need to acquire a series of images equally-spaced in time, you'll be able to ask for that though *ioctl*. To this aim, you need enough free memory to allocate the needed number of buffers.

Each time one of the relevant images has done its way to a buffer, the buffer will be marked "reserved by a file" and will not be used again for DMA unless no more free buffers are there. When the reading process is done with an image, it will get hold of the next one in its list of reserved buffers. If the reading process is slower than the driver in producing images, you'll have to choose your policy of error recovery between "release them all and start over", "drop the oldest one but keep the pace", "drop the newest one and keep the pace", "drop the newest one and get one as soon as possible".

5.5 Receiving a Signal

A process can ask to receive a signal when a new buffer is ready. By calling `PX_IOCSIGNAL`, the process tells which signal it wants to receive. After the process has expressed interest in a new image buffer (either by issuing a non-blocking *read*, by calling *select*, or by using *ioctl*), the process can attend other tasks. The interrupt handler that assigns a buffer to this file will also send a signal to the relevant process. The next *read* call will return that image.

In addition to *read*, the signalled process can also use `ioctl`, possibly associated with *mmap*. The point of the signal is simply to tell the process that it now owns an image buffer.

The process must release the image buffer before receiving a new signal, because the device driver can't give more than one image buffer to each process (or, rather, each open file).

To activate signal reporting, the third argument to *ioctl* is used as a signal number, and signal 0 is used to disable signal reporting. The exact image being assigned to the process depends on other settings. See Section 5.6 [Ioctl Commands for Buffer Management], page 7.

5.6 Ioctl Commands for Buffer Management

This section lists all *ioctl* commands related to buffer management. Most communication between user space and the device driver is based on the following data structure:

```
typedef struct Px_BufControl {
    unsigned long flags;        /* currently unused */
    unsigned long frame;        /* frame number for this request */
    unsigned long currentframe; /* number of last frame acquired */
    unsigned long offset;       /* offset in buffer memory */
    unsigned short wid, hei;    /* image size */
    unsigned short pixelsize;   /* 1 or 3 (b/w or color) */
    unsigned short unused;      /* for alignment purposes */
} Px_BufControl;
```

All commands but `PX_IOCGDMASIZE` use a pointer to `Px_BufControl` as third argument.

These commands are implemented:

`PX_IOCGDMASIZE (unsigned long * argument)`
: Return to user space the size of the DMA buffer, using the pointer argument. The size includes all the buffers currently allocated by the device drivers (the buffers are sequential in high memory).

`PX_IOCNEXTBUF`
: Declare interest in a new buffer. This command reads the `frame` field, and writes the `currentframe` field (to return information to user space). If the file already owns a buffer, that buffer is released. The file is requesting either for frame number `frame` or from the most recent available frame (if `frame` is 0 or generally less than `currentframe`). The process can then wait for the buffer using *read*, or *select* (or *poll*). Or it can wait for a signal (see Section 5.5 [Receiving a Signal], page 6). The field `offset` is set to `~0` by the device driver, to signal that no buffer is currently owned.

`PX_IOCWAITBUF`
: Declare interest in a new buffer and wait for it. This is like `PX_IOCNEXTBUF` but actually waits for the buffer to be acquired. The `frame` field is both read and written to by the device driver. The `currentframe` field is written to, and `offset` is set to the offset of the frame currently owned. The fields `wid`, `hei` and `pixelsize` are updated as well. If the frame being requested is already available the command won't wait. If the frame requested is not available any more (because the request came in too late), the most recent frame will be returned.

`PX_IOCDONEBUF`
: Release a buffer. The data structure passed as argument must refer to a valid buffer (i.e., it must have been filled by `IOCINFOBUF`, described next. The driver updates `currentframe` and sets `offset` to `~0`. Note that the buffer is automatically released when you ask for a new buffer, unless it has been locked (see below).

`PX_IOCINFOBUF`
: Get information. If the file owns a buffer, both `frame` and `offset` will be set to the associated values. If the file owns no buffer, `frame` is set to 0 and `offset` is set to `(unsigned long)-1)` (i.e., `~0` as above). The field `currentframe` is updated to the last frame acquired by the driver.

`PX_IOCLOCKBUF`
: Get information and lock the buffer to this file. The command behaves like `IOCINFOBUF` but also prevents the current buffer for being released next time you ask for a buffer. After the command has run, the current file descriptor doesn't own

a buffer any more (so the buffer won't be released to the device driver when ask for a new buffer); but it can continue to use any locked buffer until it calls `IOCDONEBUF`. If the file doesn't own a buffer, the command returns an error of `EAGAIN`.

`PX_IOCSIGNAL (unsigned long argument)`
: Ask to receive a signal when the file descriptor becomes readable. If the argument is 0, signal reporting is disabled, otherwise it is used as a signal number to send to the current process. The signal is sent by the interrupt handler when the file descriptor becomes readable. You can use the signal either with *read* or *mmap*.

Equipped with these commands, a program can implement most sensible policies for use of image data. For example, a busy-looping program that want to work on any image it can will use:

```
bufctl.frame = 0; /* get first one */
ioctl(fd, PX_IOCWAITBUF, ^bufctl);
while (1) {
    /* work on offset bufctl.offset in the memory map */
    bufctl.frame++; /* need data more recent than this one */
    ioctl(fd, PX_IOCWAITBUF, ^bufctl);
}
```

The source package of the device driver includes a few demonstration source files. All of them are called '`demo-*`' and they are well commented. If you make them save image data to disk, the output files are all pgm images or ppm if you are acquiring color images. Those demonstration programs are not installed by "*make install*".

6 Trigger Support

The driver supports triggered acquisition, with both the PXC200L and the PXC200F device. However, only the first trigger of the PXC200F is supported (the other three triggers are currently unused).

To enable trigger operation you can invoke `ioctl(PX_IOCSTRIG)`, passing a pointer to trigger-related flags in the third argument or *ioctl*.

The available flags are as the following:

`PX_FLAG_TRIGMODE`
: Ask for trigger mode.

`PX_FLAG_TRIGEDGE`
: The trigger is edge-triggered. The default is level-triggered.

`PX_FLAG_TRIGLEVL`
: The trigger is level-sensitive. This is the default.

`PX_FLAG_TRIGPOS`
: Positive edge/level. The default is negative.

`PX_FLAG_TRIGNEG`
: Negative edge/level. This is the default.

`PX_FLAG_TRIGDBB`
: This flag asks to activate the "DeBounce Both edges" option, which is disabled by default. It is not used for PXC200L.

`PX_FLAG_TRIGDBS`
: This flag asks to activate the "DeBounce Short" option, which is disabled by default. It is not used for PXC200L.

Please look at the sample file '`demo-trigger.c`' to see how things work. Also, '`pxc_live`' has a (very basic) support for trigger operation. Please note that the application is dead when waiting for a trigger. This because I didn't want to leave the easy realm of scripting languages to make that simple demonstration.

7 Device Methods

A device driver is plugged in the system by means of a table of "device operations" (or methods) that is takes care of. The implementation (or lack of) used in the *pxc* grabber is described below.

open

The open method allocates resources. On first open the device is initialized and its DMA engine is programmed for correct operation. Since high resolution (frame acquisition) and low resolution (field acquisition) use different DMA programs, they are incompatible; i.e., opening a high-resolution device will return `EBUSY` ("device busy") if a low-resolution device is currently open. The same incompatibility exists for color vs. B/W mode.

On the other hand, opening the control entry point, /dev/pxc0ctl, won't ever return EBUSY, so you can use it to issue configuration commands independent of current device setup.

close

Closing the device releases the resources it uses. On last close the grabber is shut down unless *PX_FLAG_PERSIST* is (see Chapter 8 [Ioctl], page 10).

read

Reading the device returns acquired data. According to the device being read you'll get different results (see Chapter 2 [Overview], page 1) Reading the control node returns the BT848 configuration registers, as a continuous stream of binary data. After the device is first opened, the *read* system call will block until data is available. This can take up to two frames (80ms). To avoid blocking your application can choose to keep the device open.

write

No *write* is implemented. If and when it is implemented, it will allow text-mode control of grabber features, without the need to go through *ioctl*. However, given the availability of *pxc_control* (see Section 10.2 [pxc_control], page 16), I doubt I'll ever implement *write*.

select/poll

The *select* (or *poll*) methods should tell whether read() or write() will block. It is currently unimplemented.

mmap

You can memory-map the image region. Both low-resolution and high-resolution regions can be memory-mapped, both in black-and-white and in color mode. While I routinely use *mmap* in my application, I didn't make extensive tests of potential dangerous setups; in particular, things may fail if you close the device before releasing the mapping. This happens because open() and close() are used to control device initialization and shutdown even though a memory map can survive after the device is closed. Memory-mapping can be tested by running the "mapper" program (see Section 10.3 [Other Tools], page 17).

During continuous grabbing (the default when the device is opened), the mmap()able region is as large as one frame (or field, if you access the low-res image). If the device is instructed to leave continuous acquisition and just grab a few fields instead, you'll be able to map exactly that number of fields as a continuous memory region. This feature is untested and you are urged to call me beforehand if you need it.

ioctl

The *ioctl* entry points is used to configure the device. The list of supported commands is large enough to deserve a chapter of its own. See Chapter 8 [Ioctl], page 10.

8 Ioctl

The *ioctl* entry point is used to see and/or change what is happening in the internals. A few of the ioctl commands are very low level, and I'd suggest using them only if you have hardware knowledge of the device.

I implemented no kind of protection on the device: you must protect it using the normal Unix permission/owner techniques (however, by default the devices are open to everyone, feel free to change *pxc_load* if needed). It might make sense to implement some access restriction in the device, but I'm not sure about it.

The following list describes all the commands currently implemented in the driver excluding the ones related to buffer management, that have already been described in Chapter 5 [Buffer Management], page 4. The type of the third argument (if any) is specified in the parenthesis. All of the commands can also be issued by means of the *pxc_control* application as well (see Section 10.2 [pxc_control], page 16).

`PX_IOCRESET (no arguments needed)`

The command resets the device by bringing it down and up again.

`PX_IOCHARDRESET (no arguments needed)`

This is mainly a debugging tool: it decreases the usage count of the module to one (and resets the device as well). The command is needed when oops() happen; in this case the faulty process won't close the device, and the usage count of the module won't decrease to zero any more. Opening the device once more to issue PX_IOCHARDRESET will bring the usage count to zero when the device will be closed, thus allowing module unload.

`PX_IOCGFLAGS (unsigned long * argument)`

Return to user space the current value of device flags. See pxc200.h for the meaning of each bit.

PX_IOCSFLAGS (unsigned long * argument)

Set the device flags from the value pointed by the argument. Only `PX_USER_FLAGS` are copied to the device. Currently only two flags are defined: `PX_FLAG_PERSIST` prevents the device from being shutdown on last close and re-initialized at first open. `PX_FLAG_CONTACQ` can be used to require continuous acquisition even when reading from a *pgm* or *ppm* device. Continuous acquisition was the default in versions up to 0.27 of the device, but wasn't the right choice. Version 0.28 and later only set continuous acquisition when reading from the one of the raw devices (and you can reset the flag to stop acquisition).

`PX_IOCGDMASIZE (unsigned long * argument)`

Return to user space the size of the DMA buffer, using the pointer argument. The size includes all the buffers currently allocated by the device drivers (the buffers are sequential in high memory).

`PX_IOCGDMABUF (unsigned long * argument)`
: Return to user space the physical address of the DMA buffer being used. Only informative.

`PX_IOCGIRQCOUNT (unsigned long * argument)`
: Return to user space the number of interrupts the device received. The counter is reset at each device initialization; set PX_FLAG_PERSIST to avoid resetting the device each time its user count drops to zero.

`PX_IOCGRISCADDE (unsigned long * argument)`

`PX_IOCGRISCADDO (unsigned long * argument)`
: Return to user space the physical address of the RISC program used to control data acquisition for each field. The driver uses two different physical pages to store the programs for even-field and odd-field acquisition.

`PX_IOCGPROGRAME (void *argument)`

`PX_IOCGPROGRAMO (void *argument)`
: Return to user space the actual program used by the RISC DMA controller. The program is at most one page long (PAGE_SIZE, as defined in <asm/page.h>). Two different pages (and two different programs are used for the even and the odd field.

`PX_IOCGREFV (unsigned long * argument)`
: Retrieves the reference voltage set in the DAC chip. This defaults to 128 at initialization time (i.e., 1.25 volts).

`PX_IOCSREFV (unsigned long * argument)`
: Set the reference voltage value: the voltage is used by the By848 to scale video samples. The PXC200 generates the reference using an 8-bit DAC powered by a 2.5V voltage reference.

`PX_IOCSMUX (unsigned long * argument)`

`PX_IOCGMUX (unsigned long * argument)`
: These commands are used to control the input multiplexer of the Bt848 chip. The chip can choose between three or 4 (Bt848A) video signals. The default is using input 0 (the one connected to the BNC connector of the PXC200). If the device is a plain Bt848 or Bt878 device, input 3 is the default.

`PX_IOCSTRIG (unsigned long * argument)`
: The command is used to set the trigger mode used in acquisition. The value pointed to by the argument is a combination of trig-related flags (which can't be set by IOCSFLAGS). No "IOCGTRIG" is there, as the information can be retrieved via "IOCGFLAGS". See Chapter 6 [Trigger Support], page 8.

`PX_IOCACQNOW (no arguments)`
: The command starts acquisition (either continuous acquisition or a single image, according to `PX_FLAG_CONTACQ`). Currently the command is useful to request acquisition of a single field or frame while reading (or memory-mapping a pgm/ppm device).

`PX_IOCACWAITVB (unsigned long * argument)`
: If the integer pointed-to by the argument is 0, this command will wait until a full image is acquired (thus, two vertical blanks in high resolution, one in low resolution). If the integer is not 0, the system call only returns after that many images are received. Thus, 0 and 1 have the same meaning.

  The command can only be used during acquisition, as the driver is currently not able to count VB's unless it is grabbing. The command returns an error of `EAGAIN` if no acquisition is active. The system call can block indefinitely if acquisition is

interrupted before it terminates; however, it can be interrupted by non-blocked signals.

`PX_IOCSEQUENCE (Px_AcqControl * argument)`

This command is considered obsolete, I plan to remove it in the future as buffer management has been introduced, and it offers much better performance.

The command tells the frame grabber to acquire a sequence of images/frames. When issued on an hires device this instructs to acquire whole frames, when issued on a low-res device this instructs the driver to acquire fields (but always the same field). The structure, defined in pxc200.h, includes the following fields:

```
    __u32 flags    Various flags, currently unused.
    __u32 count    How many images to grab before EOF is seen.
    __u32 step     Grab one every that many frames (this is
independent of the camera being PAL
or NTSC, the user program must make its
own timing calculations.
    __u32 buflen   Allocate a vmalloc() buffer that big. The
number is an image count, so you can
for example require 50 images using
a 10-image buffer.
```

The command can return -ENOMEM if the requested buffer cannot be allocated.

If this ioctl() returns success, successive read() calls will read from the allocated buffer instead of the current acquired image. This effectively allows to read a sequence of evenly-spaced images to user space. Data being read will always be in raw format, independently of whether you are reading the *pgm* or non-*pgm* device node. When every field has been transferred to user space, the reader will se EOF.

If the device is opened multiple times, only one file descriptor at a time can acquire a sequence. This means that you can spawn acquisition of a sequence even while other program are accessing the device in the default way. If you invoke IOCSEQUENCE on a device that already has a sequence defined, you'll get -EBUSY.

Acquisition of a sequence is terminated at file close. Disabling an ongoing sequence to get back to continuous acquisition is not possible at the time being.

The command is used in pxc_grab.

`PX_IOCGHWOVERRUN (unsigned long * argument)`

Returns to user space the number of hardware overruns experienced up to now. The counter is reset at device bringup time (see the description of IOCGIRQCOUNT above).

`PX_IOCGSWOVERRUN (unsigned long * argument)`

Returns to user space the number of software overruns experienced up to now. The counter is reset at device bringup time (see the description of IOCGIRQCOUNT above). Software overruns can only happen during sequence acquisition.

`PX_IOCGBRIGHT (signed long * argument)`

`PX_IOCSBRIGHT (signed long * argument)`

These commands get/set the number of brightness value. Brightness value for bt848 device ranges from -128 to 127 and 0 is the default value.

`PX_IOCGCONTRAST (unsigned long * argument)`

`PX_IOCSCONTRAST (unsigned long * argument)`

These commands get/set the number of contrast value. Contrast value for bt848 device ranges from 0 to 511 and 216 is the default value.

`PX_IOCGHUE (signed long * argument)`

`PX_IOCSHUE (signed long * argument)`
: These commands get/set the number of hue value. Hue value for bt848 device ranges from -128 to 127 and 0 is the default value.

`PX_IOCGSATU (unsigned long * argument)`
`PX_IOCSSATU (unsigned long * argument)`
: These commands get/set the number of chroma (u) value. Chroma (U) value for bt848 device ranges from 0 to 511 and 254 is the default value.

`PX_IOCGSATV (unsigned long * argument)`
`PX_IOCSSATV (unsigned long * argument)`
: These commands get/set the number of chroma (v) value. Chroma (V) value for bt848 device ranges from 0 to 511 and 180 is the default value.

`PX_IOCGVTYPE (unsigned long * argument)`
`PX_IOCSVTYPE (unsigned long * argument)`
: These commands get/set video-type. Video type could be either PX_CVID (composite video) which is the Default one or PX_SVID (S video). These commands are needed only for people who use pxc-200F board and would like to use S-Video on all 4 channels. Since pxc-200F board share channel 0 between composite video and S video, there is no other way around to set the video type automatically.

`PX_IOCGIFORM (unsigned long * argument)`
`PX_IOCSIFORM (unsigned long * argument)`
: The commands are used to get and set the input video format. Allowed values are listed in '`bt848.h`', and are `BT848_IFORM_NTSC`, `BT848_IFORM_PAL_M` etc. The default format is `BT848_IFORM_AUTO`, so the driver automatically adapts to NTSC/PAL video input. You may need to force a specific input format in order for the device to reliably output stable hsync and vsync drives (this only applies to the PXC200F device, as the L flavour has no output drives). The Bt848 device might lock if changing video format too often; in order to prevent this problem, the driver ensures that each change happens at least 5 seconds after the previous change; you can verify the delay by running `./pxc_control setiform 1 setiform 0`.

9 Bugs and Pending Issues.

Some features are missing. Some of them are just "not yet" features, and some are missing by design – i.e., I have no plans to implement them in the near future, unless someone shows interest in them. This section describes only those kind of deficiencies (or ideas that are not scheduled to be implemented).

Current bugs are

- Support the *bigphysarea* patch if installed (bug #25)
- Support acquisition of a region of interest (bug #34)
- Support *devfs* (bug #36)
- Document behaviour for *close on exec* (bug #43)
- Allow fine-tuning of parameters and use a configuration file whence default values are loaded (bug #49)
- Implement acquisition of several images evenly spaced in time (bug #50)
- Document problems if physical RAM is 1GB or more (bug #53)
- Maintain a ChangeLog file (bug #55)
- Add *Video for Linux* support (bug #56)

10 User space programs

The device driver package includes a few applications, to test and/or use the frame grabber. One of these programs, *pxc_xgrab* is especially designed for user consumption (read, for the non-technical user), while the other ones are more tools for the Unix-savvy user.

10.1 pxc_xgrab

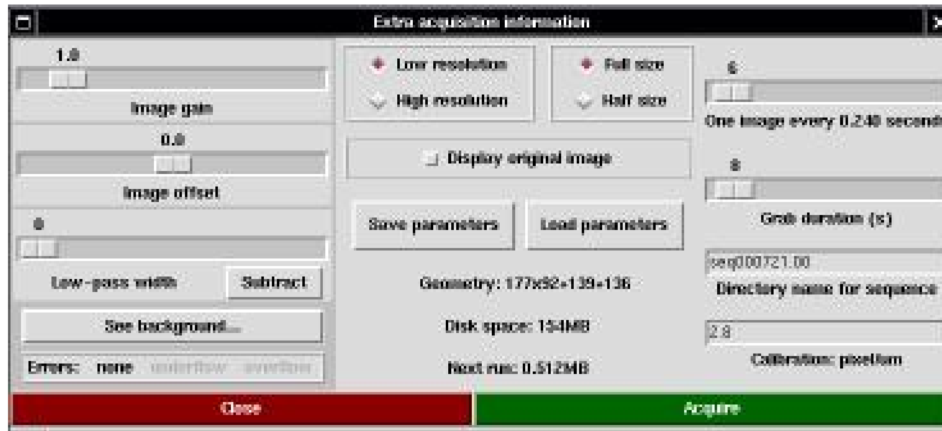
The program is a tool to acquire a sequence of images to disk.

Allowed arguments are `-hires` to select high-resolution mode and/or a number to select a specific device (in case you have more than one). For example, `pxc_xgrab 1` will use `/dev/pxc1pgm` as input device. If both arguments are present, `-hires` must appear first.

Its main window is similar to *pxc_live*, the simple live-video and save-one application. In addition to *pxc_live*, *pxc_xgrab* has an "`Extra...`" button to open a control window. This document only deals with the control window, as the main window is definitely plain and self-explanatory.

The control window has a different look according to whether the program is running under *Pacco* or under a plain wish. When running under *Pacco* the program offers image operations (background subtraction, gain, offset), displaying at half size (faster), and a "save set-

tings"/"load settings" functionality. Other features are available in both cases, and the following description applies to the *Pacco* case.



Whenever the control window is active, the program draws a region of interest in the main window (over the currently-acquired image). To move such a region the user must drag the sides of the region using the mouse pointer. To avoid clicking exactly over the side of the region, you can click anywhere in the image and then move towards the side you want to move; the side will stick to the mouse pointer as soon as they get near enough.

The region of interest marks the part of the input image that will be saved to disk. The smaller the region of interest, the best the results (although modern computers should have no problems in acquiring whole images at full speed, I prefer to be as conservative as possible).

The control window is organized as three columns of controls: the left column deals with image operations, the right column with acquisition parameter and the middle column with miscellaneous control and information.

image operations

The left column of controls offers background subtraction and gain/offset of the image data by means of three scales and two buttons (as well as error indicators for overflow and underflow).

Background subtraction is performed by taking a snapshot of the input image and applying a low-pass filter. The resulting image is subtracted from any further acquired image. The low-pass filter is a square filter: the low-pass image is obtained by the convolution of the input image and a square are whose integral is 1.0. Actually, for performance reasons the program performs two convolutions, one with an horizontal bar and one with a vertical bar, both of integral 1.0.

Please note that when the user acts on the scale to change the width of the low-pass filter, the background is not immediately updated; you need to press the "`Subtract`" button in order to update the background.

If you want to see what the calculated background looks like, you can use the "`See background...`" button. The program will open a new window showing the background currently in charge.

After the background has been subtracted (or not subtracted if the low-pass filter has zero-width), the image undergoes a gain and offset pass. The gray-level of each pixel (which is represented as a floating-point number between 0.0 and 1.0 – black and white) is first multiplied by the specified gain (default 1.0) and then offset by the specified offset (default 0.0). If any underflow (less than 0.0) or overflow (more than 1.0) occurs, the underflow and overflow markers will be lit, and the pixel values in the output image are limited to be between 0.0 and 1.0 anyways.

miscellaneous controls

The central part of the control window offers miscellaneous controls and information. The user can choose whether acquisition must be performed to low or high resolution, and whether the on-screen image must be full-size or half-size (half-size display is faster, but you are not allowed to move the region of interest when visualizing at half-size).

The "`Display original image`" check-button can be used to disable and re-enable image operations on the on-screen image.

To ease people who performs repeated experiments, the program includes two buttons called "`Save parameters`" and "`Load parameters`". The region of interest and acquisition parameters are always saved to the same file, so the user is not allowed to "`Save As`" a different name. The parameters are saved to file called "`.pxc_xgrabrc`" in the user's home directory).

To the bottom, the program reports some information: a numerical representation of the current region-of-interest (in the form *width x height + xoffset + yoffset)*), the disk space available in the current directory and the size of the next acquisition, if it is performed with the current region of interest and acquisition parameters.

acquisition parameters

The right column of the window is used to choose the frame rate, the duration of the acquisition, the on-disk name for the sequence of images to acquire and the microscope's calibration (if you have a microscope).

The frame rate is represented by an integer number, that reads as "save one frame every that many frames". To ease the user the rate is also expressed as time between successive acquisitions (for PAL cameras that value is a multiple of 40ms). The duration of acquisition is expressed in seconds.

The "Directory name for sequence" value is the name of a directory, where the program will save acquired images as a collection of independent *pgm* files called `sequence.`*NNN*, and acquisition parameters (calibration and frame rate) as a file called `params`. Even though the program suggests an experiment name, you are free to replace the name; the program will take care of choosing a new name for every acquisition run, using your suggested name as an hint.

The last entry is the image calibration, meant to use the tool in acquiring microscope data (in association with the *Bisce* package). The calibration figure is expressed as pixels/micrometer, and is automatically adjusted when you switch image resolution. The default value is 2.8 pixel/um in low resolution and 5.6 pixel/um in high resolution. The value can be edited; it will be saved along with the other acquisition parameter.

10.2 pxc_control

Another application that deserve a section of its own, is *pxc_control*, even though it is pretty low-level in design.

The program accesses the frame grabber device and issues *ioctl* commands according to the command-line parameters it receives. The user can tell *pxc_control* to use a different device entry point than the compiled-in default (`/dev/pxc0ctl`) by specifying it as first or last argument of the command-line, or by setting the `PXCDEVICE` environment variable.

If the program is called without any argument, it will print to stderr an usage summary (and the list of supported commands).

I do my best to implement here any command I add to the kernel driver (I don't like to compile special programs just to test things out, shell scripts are easier).

A sample session with the command looks like this:

```
snakes% pxc_control getmux getcontrast setcontrast 200 getcontrast
ioctl("/dev/pxc0ctl", PX_IOCGMUX, ...) = "0x00000000 (0)" 0
ioctl("/dev/pxc0ctl", PX_IOCGCONTRAST, ...) = "0x000000d8 (216)" 0
ioctl("/dev/pxc0ctl", PX_IOCSCONTRAST, ...) = 0
ioctl("/dev/pxc0ctl", PX_IOCGCONTRAST, ...) = "0x000000c8 (200)" 0
```

As you see, the program prints (to stderr) a detailed report of what is going on. Actually, *pxc_control* is the engine that work behind the scenes when *pxc_live* allows changing grabber parameters.

10.3 Other Tools

These simple tools accept the device name as first or last argument of the command-line, or they get it from the `PXCDEVICE` environment variable. The compiled-in default is either `/dev/pxc0ctl` or `/dev/pxc0pgm`.

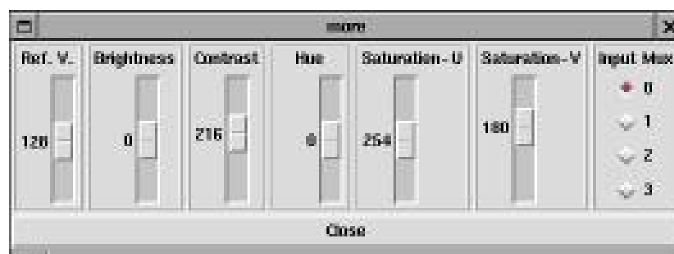
pxc_status

Show the current status of the device (through the informational registers). Since this is quite fussy, you'd better grep for the information you need (information is organized by lines). The program uses /dev/pxc0ctl (or the control file associated to another grabber) to retrieve information, and will complain if the file is not a control entry point.

pxc_show

The program displays on a text screen a live-video display of the data being grabbed. It reads data from a *pgm* device so it can adapt to the real image size, whether it is NTSC or PAL. This is different from earlier versions (pre-0.21) which used the raw entry point and couldn't tell what the image size was.

pxc_live



A simple live-video application. It uses either the "pacco" tool (a GPL package, available from `ftp://ftp.linux.it/pub/People/rubini` or the photo widget of tk (the photo widget is slower). Part of this code is contributed by Daniel Scharstein. The device being used is retrieved from the command line, from the environment variable PXCDEVICE or defaults to /dev/pxc0pgm. Both *pgm* and *ppm* devices do work. The program can also control some camera parameters, using knobs laid out in an optional window, activated by the `More...` button.

pxc_grab

A command line tool for grabbing to disk. Use the '-h' option to get an help screen or check how *pxc_xgrab* calls *pxc_grab*.

mapper

A basic mmap() engine: it prints to stdout data that is accessed via memory map. It takes three arguments: filename, starting offset and final offset. To try out memory mapping, run the following command-line (designed for NTSC video):

```
./mapper /dev/pxc0H 0 `expr 640 "*" 480` | \
rawtopgm 640 480 | xv -
```

The tool is so silly and unrelated to the driver that it isn't installed by "`make install`".

Table of Contents

The pxc package 1

1 Incompatibilities 1

2 Overview 1
2.1 Device Special files 2

3 The DMA Buffer 3

4 Installing the package 3
4.1 Compiling the Package 3
4.2 Loading the Kernel Module 4
4.3 Unloading the Module 4

5 Buffer Management 4
5.1 Multiple Buffers 5
5.2 Buffer Allocation Using Read 5
5.3 Mmap 6
5.4 Detailing Specific Requirements 6
5.5 Receiving a Signal 6
5.6 Ioctl Commands for Buffer Management 7

6 Trigger Support 8

7 Device Methods 9

8 Ioctl 10

9 Bugs and Pending Issues. 13

10 User space programs 14
10.1 pxc_xgrab 14
10.2 pxc_control 16
10.3 Other Tools 17