pxaRC - R/C and robotics software for Linux/PXA255/PXA270

Copyright © 2005, 2006, 2007, 2008 pabr@pabr.org
All rights reserved.

pxaRC is a collection of drivers and utilities for remote-control and robotics applications on Linux/PXA255/PXA270-based platforms such as the Gumstix basix, connex [http://docwiki.gumstix.org/Basix_and_connex] and verdex [http://docwiki.gumstix.org/Verdex] motherboards.

READ THE HYPERTEXT VERSION HERE:

<table>
<thead>
<tr>
<th>Revision History</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3 2008-09-07</td>
</tr>
<tr>
<td>1.2 2007-05-09</td>
</tr>
<tr>
<td>1.0 2005-06-02</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction .................................................................................................................. 3  
2. Disclaimer ...................................................................................................................... 3  
3. RC signals ..................................................................................................................... 3  
   3.1. RC servo control signals .......................................................................................... 3  
   3.2. Multiplexed signals on the radio link ...................................................................... 4  
   3.3. H-bridge control signal ........................................................................................... 4  
   3.4. Accelerometer output signal .................................................................................... 4  
4. Software .......................................................................................................................... 4  
   4.1. Requirements .......................................................................................................... 4  
   4.2. Design principles .................................................................................................... 4  
   4.3. Installation .............................................................................................................. 5  
      4.3.1. Compiling within Buildroot ................................................................................ 5  
      4.3.2. Compiling outside Buildroot ............................................................................. 5  
   4.4. `pxa_rtsched`: simple real-time scheduler ............................................................... 5  
      4.4.1. Overview .......................................................................................................... 5  
      4.4.2. Implementation details ...................................................................................... 6  
      4.4.3. Known problems .............................................................................................. 6  
   4.5. `pxa_opwm`: PWM/PPM signal generator ................................................................. 6  
      4.5.1. Overview .......................................................................................................... 6  
      4.5.2. Usage .............................................................................................................. 7  
      4.5.3. Implementation details ...................................................................................... 7  
      4.5.4. Known problems .............................................................................................. 8  
   4.6. `pxa_ipwm`: PWM/PPM/DCM signal decoder and pulse counter ............................. 8  
      4.6.1. Overview .......................................................................................................... 8  
      4.6.2. Usage .............................................................................................................. 8  
      4.6.3. Implementation details ...................................................................................... 9  
      4.6.4. Known problems .............................................................................................. 10  
   4.7. `iprc_rx`: RC receiver emulator ................................................................................ 10  
   4.8. `iprc_tx`: RC transmitter emulator .......................................................................... 10  
   4.9. `ppmrelay`: Copy a PPM stream ............................................................................. 11  
   4.10. `short2ascii`, `ascii2short`, `long2ascii`, `ascii2long`: Binary to ASCII converters ....... 11  
   4.11. `pxa_cfcam`: CMOS image sensor on the PCMCIA/CF bus ..................................... 12  
      4.11.1. Overview .......................................................................................................... 12  
      4.11.2. `pxa_cfcam_pnm`: convert images to PNM format ............................................. 12  
      4.11.3. Supported sensors ......................................................................................... 13  
      4.11.4. Implementation details .................................................................................... 13  
      4.11.5. Known problems ............................................................................................ 13  
5. Use cases ......................................................................................................................... 14  
   5.1. Remote-control with USB joystick connected to ground station ............................ 14  
   5.2. Remote-control with Bluetooth joystick .................................................................. 14  
6. Roadmap .......................................................................................................................... 14  
   6.1. Bluetooth QoS ......................................................................................................... 14  
   6.2. Software-based GPIO input debouncing ................................................................. 14  
   6.3. Multiple queues in `pxa_ipwm` ................................................................................ 14  
   6.4. Video4Linux compatibility ....................................................................................... 15  
Bibliography ......................................................................................................................... 15  
Glossary .............................................................................................................................. 15
1. Introduction

Table 1, “Overview” shows how pxaRC addresses the requirements of R/C and robotics applications.

<table>
<thead>
<tr>
<th></th>
<th>R/C models</th>
<th>pxaRC</th>
<th>Robotics</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-board computer</td>
<td>R/C receiver with little or no programmability</td>
<td>200-400 MHz ARM-based board</td>
<td>PC/104, Mini-ITX and misc. proprietary platform</td>
</tr>
<tr>
<td>Operating system</td>
<td>No open OS</td>
<td>Linux with simple real-time extensions</td>
<td>Linux, vxWorks, Windows, WinCE</td>
</tr>
<tr>
<td>C&amp;C channel</td>
<td>One-way 41/72 MHz PPM or PCM radio</td>
<td>TCP/IP over Bluetooth or Wi-Fi</td>
<td>Misc. protocols on 433 MHz, 868 MHz, 2.4 GHz; TCP/IP</td>
</tr>
<tr>
<td>C&amp;C user interface</td>
<td>41/72 MHz R/C transmitter with analog sticks</td>
<td>USB joystick and iprc_tx on ground-based laptop; Bluetooth joystick.</td>
<td>Workstation with dedicated GUI</td>
</tr>
<tr>
<td>Receiver</td>
<td>41/72 MHz R/C receiver</td>
<td>iprc_rx and pxa_opwm</td>
<td>Misc. proprietary systems</td>
</tr>
<tr>
<td>Actuators</td>
<td>R/C servos and speed controllers</td>
<td>R/C servos and speed controllers; I2C servos</td>
<td>High-end servos with position and torque feedback; misc. proprietary hardware</td>
</tr>
<tr>
<td>Video processing</td>
<td>2.4 GHz wireless video camera</td>
<td>pxa_cfcam and CMOS sensor</td>
<td>USB cameras; cameras with PC104 or PCI video acquisition boards</td>
</tr>
<tr>
<td>I/O</td>
<td>R/C servo outputs; few inputs</td>
<td>GPIO, I2C, CF; Audiostix, Robostix</td>
<td>Parallel/serial/USB ports; misc. proprietary interface boards</td>
</tr>
<tr>
<td>Sensors</td>
<td>Gyros</td>
<td>MEMS accelerometers with I2C or duty-cycle outputs</td>
<td>Misc</td>
</tr>
</tbody>
</table>

2. Disclaimer

- Use these instructions and the related software at your own risk.

- This is an experimental project with none of the fail-safe features one could expect from a commercial product.

- Class 2 Bluetooth devices (the most common type) have a range of 10 m. This is not appropriate for some applications.

- Unlike 41/72 MHz R/C radio equipment, Bluetooth and Wi-Fi devices use shared radio spectrum which can be jammed by various sources.

3. RC signals

This section clarifies the naming of modulations used in RC signals.

3.1. RC servo control signals

Figure 1. RC servo signal
The target angular position of the servo is encoded as the duration of the "high" states of a square-wave signal, between 1 ms and 2 ms. The duration of the "low" states does not encode information. This is a pulse-width modulation (PWM).

### 3.2. Multiplexed signals on the radio link

**Figure 2. Six-channel "PPM" signal**

Channel values are encoded as the absolute duration between successive low-to-high transitions, between 1 ms and 2 ms. Independent channels are transmitted sequentially. A "low" state longer than about 5 ms signals the beginning of a new frame. Strictly speaking, this is a multiplexed differential PPM modulation. Since the "high" pulses generally have a fixed duration of 0.5 ms, the signal can also be seen as a multiplexed PWM modulation of the "low" states between 0.5 and 1.5 ms.

### 3.3. H-bridge control signal

**Figure 3. H-bridge control signal**

An H-bridge is driven by the variable duty-cycle of a square-wave signal. Strictly speaking, this is a duty-cycle modulation (DCM). It is sometimes called PWM, although this is only correct when the repeat rate of the pulses is constant (which is generally not the case for a RC servo signal).

### 3.4. Accelerometer output signal

**Figure 4. Accelerometer output signal**

Some solid-state accelerometers output a duty-cycle-modulated square-wave signal. Note that it would be wrong to decode this as a PWM signal, since the frequency is not tightly regulated.

### 4. Software

#### 4.1. Requirements

The software is tested with Gumstix Buildroot svn rev 1161 (linux-2.6.18gum) and rev 1321 (linux-2.6.18gum).

#### 4.2. Design principles

- For performance reasons, APIs use binary encodings whenever possible. Utilities are provided to access them in ASCII format: see Section 4.10, "short2ascii, ascii2short, long2ascii, ascii2long": Binary to ASCII converters ".
- Device interfaces use read() and write() whenever possible because these system calls are easier to use than ioctl() in most programming languages.
- Devices are configured with module parameters rather than with a dedicated API. This reflects the fact that configurations are generally tied to hardware and therefore need not be changed at run-time.
- Access control is based on permissions of the device special files.
4.3. Installation

Source code can be downloaded from http://www.pabr.org/pxarc/dist/.

4.3.1. Compiling within Buildroot

$ tar zxvf pxarc-$VERSION.tar.gz
$ cp -r pxarc-$VERSION/package/buildroot $SOMEWHERE/gumstix-buildroot/package/pxarc
$ cd $SOMEWHERE/gumstix-buildroot/
$ echo 'source "package/pxarc/Config.in"' >> package/Config.in    # Optional (to see pxaRC in menuconfig).
$ make                                                            # Makes sure the toolchain is built.
$ make TARGETS="linux pxarc"                                      # Compiles pxaRC itself.
$ ls build_arm_nofpu/root/lib/modules/*/misc/                    # Newly-created modules should be there.

Flash the root filesystem, then configure the modules you need in /etc/modprobe.conf and/or /etc/modules. If you are using udev, devices will be registered in /dev automatically on each boot. Otherwise, you will need to set dev_major=... for each module and create devices with mkod.

4.3.2. Compiling outside Buildroot

$ tar zxvf pxarc-$VERSION.tar.gz
$ cd pxarc-$VERSION/src
$ vi Makefile    # Adjust $(BR) and $(LINUX_VERSION)
$ make
$ scp *.ko root@$IPADDRESS:
$ scp iprc_rx.target root@$IPADDRESS:iprc_rx
$ scp ppmrelay.target root@$IPADDRESS:ppmrelay
$ scp short2ascii.target root@$IPADDRESS:short2ascii
$ scp ascii2short.target root@$IPADDRESS:ascii2short
$ scp long2ascii.target root@$IPADDRESS:long2ascii
$ scp ascii2long.target root@$IPADDRESS:ascii2long

You will need to manually insmod the modules you need.

4.4. pxa_rtsched: simple real-time scheduler

4.4.1. Overview

pxa_rtsched.ko provides real-time capabilities based on the PXA's fast interrupts (FIQ). Its main features are:

- Scheduling of tasks and timed sequences of tasks, with microsecond-scale average jitter.
- Processing of edge-triggered GPIO interrupts, with microsecond-scale average interrupt latency.

pxa_rtsched is intended to be used by other kernel modules such as pxa_opwm and pxa_ipwm through the API in pxa_rtsched.h. It has no user-accessible interface, apart from printing statistics in dmesg when unloaded.

The timer period is 271 ns for the PXA255 and 308 ns for the PXA270. This translates into about 12 bits of resolution when encoding or decoding typical R/C signals. In practice, due to as yet unidentified reasons, worst-case jitter and latency can exceed ±3 μs on a heavily loaded 200 MHz processor. This is good enough for most R/C applications, where servos have a dead-band (hysteresis) of several microseconds.
4.4.2. Implementation details

pxa_rtsched uses OSMR1 (OS Match Register 1), which is normally available on ARM Linux. OSMR0 is used by the system timer, and OSMR3 is used by the watchdog.

The following tricks are used in order to reduce latency:

- Use FIQ interrupts, which have higher priority than the regularIRQs used by the Linux kernel
- Use the shadow r8-r14 FIQ registers
- Lock the FIQ handler code into the I-cache
- Lock parts of the FIQ handler code into the D-cache as well (this is required because GCC stores constants in the .text section for PC-relative addressing)
- Lock all data structures used in FIQ context into the D-cache, including the stack
- Lock addresses of built-in peripherals into the D-TLB
- Use the Performance Monitor Count Registers (PMN0, PMN1) of the PXA to confirm that code and data are really locked into the caches (several bugs were discovered that way).

4.4.3. Known problems

- The scheduling algorithm has unbounded execution time. It may fail to enforce hard-real-time behaviour when used in excessively complex scenarios (e.g. trying to schedule a long sequence of tasks when many tasks are already scheduled). Reports of overruns and overloads can be seen with dmesg when pxa_rtsched.ko is unloaded.

- Timer interrupts and edge-triggered GPIO interrupts compete for the same resource, i.e. FIQ-mode CPU cycles. If both features are enabled simultaneously, hard-real-time behaviour cannot be guaranteed. Excessively fast transitions on a GPIO input may even freeze the system entirely. See Section 6.2, “Software-based GPIO input debouncing”.

- Due to subtle issues in kernel memory management and implementation details of pxa_rtsched, unloading RT modules will decrease the accuracy of the scheduler and may make the system vulnerable to a race condition leading to a crash. As a workaround, use rmmod only during development, and always unload the whole stack of modules, including pxa_rtsched.ko.

- RT tasks written in C must be carefully designed to avoid overflowing the FIQ stack.

4.5. pxa_opwm: PWM/PPM signal generator

4.5.1. Overview

pxa_opwm.ko is a timer-based PWM/PPM signal generator for the PXA255 and PXA270, implemented as a loadable Linux kernel module. It can be configured to generate a single multi-channel PPM signal and/or the associated single-channel PWM signals. Signals can be output on any GPIO pin.
pxa_opwm does not use the hardware PWM generators of the PXA.

pxa_opwm replaces pxa_mpwm, a previous implementation with 8-bit resolution.

4.5.2. Usage

Figure 6. OPWM waveforms (PPM mode); meaning of timing parameters

GPIO pins, channels and timings are configured with module parameters as illustrated in Table 2, “pxa_opwm module configuration examples”.

A user-mode process controls pxa_opwm by writing 16-bit unsigned integers in host endianness to device special files. See Table 3, “pxa_opwm devices”. Signal generation begins after a program has opened the device and written values for all channels.

When the process releases the device, all outputs are reverted to logic 0. This should cause most ESCs to shutdown and most servos to become idle. This safety feature can be disabled with the module parameter persistent=1.

Table 2. pxa_opwm module configuration examples

<table>
<thead>
<tr>
<th>Application</th>
<th>Module parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>One RC PPM signal with 6 channels</td>
<td>modprobe pxa_opwm gpio=61 nchans=6 tmin=1000 tmax=2000 tpause=500 tsync=12000</td>
</tr>
<tr>
<td>Same with associated PWM servo outputs, except CH5</td>
<td>modprobe pxa_opwm gpio=61 nchans=6 servo=58,59,60,62,-1,63 tmin=1000 tmax=2000 tpause=500 tsync=12000</td>
</tr>
<tr>
<td>PWM servo outputs only</td>
<td>modprobe pxa_opwm nchans=6 servo=58,59,60,62,-1,63 tmin=1000 tmax=2000 tpause=500 tsync=12000</td>
</tr>
</tbody>
</table>

Table 3. pxa_opwm devices

<table>
<thead>
<tr>
<th>Device</th>
<th>Minor</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/opwm0-0</td>
<td>0</td>
<td>Channel #1 of PPM signal #1</td>
</tr>
<tr>
<td>/dev/opwm0-1</td>
<td>1</td>
<td>Channel #2 of PPM signal #1</td>
</tr>
<tr>
<td>/dev/opwm0-2</td>
<td>2</td>
<td>Channel #3 of PPM signal #1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>/dev/opwm0</td>
<td>15</td>
<td>PPM signal #1 (all channels written synchronously)</td>
</tr>
<tr>
<td>/dev/opwm1-0</td>
<td>16</td>
<td>Channel #1 of PPM signal #2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>/dev/opwm1</td>
<td>31</td>
<td>PPM signal #2 (all channels written synchronously)</td>
</tr>
</tbody>
</table>

4.5.3. Implementation details

pxa_opwm is a straightforward application of pxa_rtsched. To generate a PPM frame of N pulses, pxa_opwm schedules a timed sequence of 2*N tasks (N for rising edges, N for falling edges). The associated optional PWM signals are toggled by the same tasks.
4.5.4. Known problems

- GPIO outputs use 3.3 V levels (VOL=0.4, VOH=3.2 - see [PXA255_ELEC] and [PXA270_ELEC]). Most RC servos will recognize PWM signals with an amplitude as low as 2 V, but some with CMOS circuits may fail to decode a logic 1 at 3.2 V.

- The usual motivation for generating PWM signals in software is the low resulting cost. For applications which require high-resolution signals, a hardware PWM generator should be used instead.

- pxa_opwm allows pxa_rtsched to extend synchronization pulses beyond tsync if this helps solve scheduling conflicts. This will occur especially when generating several PPM signals simultaneously (but not when generating several PWM signals from a single "virtual" PPM signal by configuring pins with servo=... only).

- As a consequence of the above item, pxa_opwm should not be used to generate duty-cycle-modulated or frequency-modulated signals, except in very simple cases where scheduling is entirely deterministic (for example: single signal and no other RT task).

- The caveats for pxa_rtsched apply to pxa_opwm as well.

There are at least two other ways to generate PWM signals on a Gumstix platform:

- Use the hardware PWM generator of the PXA (up to four independent 3.3 V PWM signals; can also do PPM with additional software)

- Add a daughterboard (see [ROBOSTIX_SIMPLE_SERVO]).

4.6. pxa_ipwm: PWM/PPM/DCM signal decoder and pulse counter

4.6.1. Overview

pxa_ipwm.ko is a PWM/PPM/DCM signal decoder and pulse counter based on edge-triggered GPIO interrupts, implemented as a loadable Linux kernel module.

4.6.2. Usage

GPIO pins, signal formats and timings are configured with module parameters as illustrated in Table 4, “pxa_ipwm module configuration examples”.

Figure 7. PWM decoding; meaning of timing parameters

Figure 8. PPM decoding; meaning of timing parameters

Figure 9. DCM decoding; meaning of timing parameters
Table 4. pxa_ipwm module configuration examples

<table>
<thead>
<tr>
<th>Application</th>
<th>Module parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-channel PWM servo signal</td>
<td>modprobe pxa_ipwm mode=0 gpio=61 tmin=1000 tmax=2000 tsync=4000 timeout=100000</td>
</tr>
<tr>
<td>One RC PPM signal with 6 channels</td>
<td>modprobe pxa_ipwm mode=1 gpio=61 nchans=6 tmin=1000 tmax=2000 tsync=4000 timeout=100000</td>
</tr>
<tr>
<td>2 kHz ±10% DCM signal</td>
<td>modprobe pxa_ipwm mode=2 gpio=61 tmin=455 tmax=556 timeout=100000</td>
</tr>
<tr>
<td>Pulse counter mode, read delayed by 3 ms</td>
<td>modprobe pxa_ipwm mode=3 gpio=61 timeout=3000</td>
</tr>
<tr>
<td>Two pulse counters, read delayed by 3 ms</td>
<td>modprobe pxa_ipwm mode=3,3 gpio=61,62 timeout=3000,3000</td>
</tr>
</tbody>
</table>

User-mode processes access pxa_ipwm by reading from a device special file. See Table 5, “pxa_ipwm devices”.

In PWM/PPM/DCM mode, read() fails with -ETIMEDOUT if no measurement more recent than timeout is available. If the signal does not match the configured timings, read() fails with -EIO (see figures). Otherwise it returns the latest measurement (without waiting for the next pulse). The channel value format is the same as for pxa_opwm.

In pulse counter mode, read() returns a 32-bit unsigned integer in host endianness. The first invocation of read() immediately returns the current value of the counter. Subsequent invocations wait until the counter increases. There may be some latency (configured with timeout=...); this is because there is no easy way for a FIQ routine to wake-up a user process.

A pulse counter can be set to an arbitrary value by writing to the device. However, there is no mechanism for atomically reading and resetting a counter.

Table 5. pxa_ipwm devices

<table>
<thead>
<tr>
<th>Device</th>
<th>Minor</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/ipwm0-0</td>
<td>0</td>
<td>Channel #1 of signal #1 (PPM mode only)</td>
</tr>
<tr>
<td>/dev/ipwm0-1</td>
<td>1</td>
<td>Channel #2 of signal #1</td>
</tr>
<tr>
<td>/dev/ipwm0-2</td>
<td>2</td>
<td>Channel #3 of signal #1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>/dev/ipwm0</td>
<td>15</td>
<td>Signal #1, all channels read at once (PPM/PWM/DCM mode only); Or counter #1</td>
</tr>
<tr>
<td>/dev/ipwm1-0</td>
<td>16</td>
<td>Channel #1 of signal #2 (PPM mode only)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>/dev/ipwm1</td>
<td>31</td>
<td>Signal #2, all channels read at once (PPM/PWM/DCM mode only); Or counter #2</td>
</tr>
</tbody>
</table>

4.6.3. Implementation details

pxa_ipwm registers with pxa_rtsched to receive edge-triggered GPIO interrupts. The FIQ handler records a timestamp for each edge into a FIFO queue (implemented as a circular buffer), and timestamps
are translated into PWM/PPM/DCM values when `read()` is called. The FIQ handler also increases the pulse counters.

### 4.6.4. Known problems

- PXA GPIO inputs are 3.3 V. Do not connect a 5 V signal directly.

- GPIO0 and GPIO1 cannot be used, because they trigger dedicated interrupts.

- All edge timestamps are stored in a single queue. As a result, if multiple pins are decoded, and one of them has much higher frequency than the others, `pxa_ipwm` will fail to decode the slower signals. This problem can be mitigated at compile time by increasing `MAX_EV`. See also Section 6.3, “Multiple queues in `pxa_ipwm`”.

- `pxa_ipwm` takes control of all edge-triggered GPIO interrupts (except those from GPIO0 and GPIO1). On Gumstix motherboards this will especially affect the ethernet driver. The USB driver (`pxa2xx_udc`) should be affected as well because it monitors IRQs from USB_GPIO0; in practice, though, established USB connections still work after `pxa_ipwm` is inserted.

- For performance reasons, all GPIO pins used by `pxa_ipwm` must be within the same 32-bit group.

- Measurement accuracy will decrease as the number of pins and the probability of “simultaneous” edges increase.

- In counter mode, pulses must be longer than 1 µs (according to [PXA255_DEVEL]). The maximum counting frequency depends on interrupt latency and on the duration of the longest RT task. If a pulse occurs before the CPU has cleared the interrupt flag from a previous pulse, then it will not be counted. With no other RT tasks running, `pxa_ipwm` can reliably count pulses at 100 kHz.

- The caveats for `pxa_rtsched` apply to `pxa_ipwm` as well.

For an alternative to `pxa_ipwm` on the Gumstix platform, see [ROBOSTIX_RC_INPUT].

### 4.7. iprc_rx: RC receiver emulator

`iprc_rx [-p <port>] [-t <timeout>] [-c <OPWM device>]`

**Example 1. iprc_rx**

```
iprc_rx -p 9001 -t 2000 -c /dev/opwm0
```

`iprc_rx` is a user-mode program designed to run on the on-board computer. It receives RC commands as UDP packets over a Bluetooth BNEP or Wi-Fi connection and forwards them to `pxa_opwm`.

The program will terminate if it does not receive any UDP packet during a configurable interval.

### 4.8. iprc_tx: RC transmitter emulator

`iprc_tx [-c <joystick>] [-r <refresh rate>] [-m <mixer file>] [-d <dest IP>] [-p <dest port>]`

**Example 2. iprc_tx**

```
iprc_tx -c /dev/js0 -r 50 -m xpad_linear6.mix -d 192.168.10.1 -p 9001
```

`iprc_tx` is a user-mode program designed to run on a Linux workstation with a Bluetooth or Wi-Fi adapter and a USB joystick. It reads the positions of the triggers and analog sticks, mixes them linearly into 6 RC channels, and sends the channel values as UDP packets over a wireless IPv4 link.
The mapping and mixing from joystick axes to RC channels are defined by a matrix in a configuration file. See for example pxarc/examples/xpad_linear6.mix.

Procedure 1. Trimming

1. Bring the sticks to the desired "neutral" position.
2. Depress the "back" button.
3. Allow the sticks to return to their central position.
4. Release the "back" button.

4.9. ppmrelay: Copy a PPM stream

ppmrelay is a user-mode program designed to run on the on-board computer. It receives PPM commands decoded by pxa_ipwm, prints the channel values to standard output, and forwards them to pxa_opwm.

It can be used as a template for more complex applications such as:

• Detecting and mitigating losses of radio signal
• Interpolating commands to a higher frequency (most servos tolerate refresh rates higher than the usual 50 Hz)
• Executing sequences of pre-recorded commands
• Injecting data from on-board sensors into the control loop.

4.10. short2ascii, ascii2short, long2ascii, ascii2long : Binary to ASCII converters

These utility programs can be used to access pxa_opwm and pxa_ipwm device special files in ASCII format (instead of binary), for example from a console or shell script.

Warning

ascii2short and ascii2long will parse numbers starting with 0 as octal.

Example 3. Read PWM values once

short2ascii < /dev/ipwm0

Example 4. Read PWM values every 20 ms (approx.)

short2ascii -d 20000 < /dev/ipwm0

Example 5. Read a pulse counter continuously

long2ascii -d 0 < /dev/ipwm0

Example 6. Write PPM values

eocho "32768 32768" | ascii2short > /dev/opwm0
Example 7. Write a sequence of PPM values

```
(echo "16384 49152";
 sleep 1;
 echo "49152 16384";
 sleep 1;
 echo "32768 32768"
)  |  ascii2short  > /dev/opwm0
```

Example 8. Alter channels of a PPM signal

```
short2ascii -d 10000  < /dev/ipwm0  \
| while read v1 v2 v3 v4 v5 v6; do
  echo "Swapping $v1 and $v2, inverting $v3"  1>&2
  echo "$v2 $v1 $((65535-v3)) $v4 $v5 $v6"
 done  \
| ascii2short  > /dev/opwm0
```

4.11. pxa_cfcam: CMOS image sensor on the PCMCIA/CF bus

4.11.1. Overview

pxa_cfcam.ko captures images from a CMOS sensor connected directly to the PCMCIA/CF port of the PXA255. The hardware interfacing is described in [CFCAM].

Features:

- Capture of still images and streaming video
- Plugin API allowing custom on-line processing of raw data.
- Optional I2C bit-banging on any GPIO (useful because the hardware I2C signals of the PXA255 are not available on the CF header).

4.11.2. pxa_cfcam_pnm: convert images to PNM format

pxa_cfcam_pnm.ko is a plugin for pxa_cfcam which decodes CMOS sensor data in Bayer format into uncompressed PNM format (24-bit color or 8-bit greyscale). The PNM header can be disabled with the module parameter \texttt{pnm\_header}=0.

Example 9. Still image capture

```
# modprobe pxa_cfcam
# modprobe pxa_cfcam_pnm
# dd if=/dev/cfcam0 of=/tmp/image.ppm bs=2M count=1
```

Example 10. Still image capture (greyscale)

```
# modprobe pxa_cfcam
# modprobe pxa_cfcam_pnm bpp=1
# dd if=/dev/cfcam0 of=/tmp/image.pgm bs=2M count=1
```
Example 11. Asynchronous video capture

```c
/* Simplified example; error handling not shown. */
int fd = open("/dev/cfcam0", O_RDWR, 0);
unsigned char *mem = mmap(NULL, 320*240*3*2, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
while ( 1 ) {
    u32 offset;
    read(fd, &offset, sizeof(offset));
    unsigned char *rgb = mem + offset;
    /* 'rgb' points to 320*240*3 bytes of RGB data */
}
```

This mode allows capture of consecutive frames. When a buffer is complete, `read()` returns its offset within the mmap'ed area, and the driver starts capturing the next frame into another buffer. The program must finish processing the first buffer and call `read()` again before the second buffer is complete itself.

4.11.3. Supported sensors

The implementation is not yet user friendly. Sensor configurations are hard-coded in `pxa_cfcam.c`. A configuration is selected with the module parameter `sensor=N`. To add support for a new sensor or a new image format, one must edit the code.

To characterize a new sensor, first obtain raw data as follows:

Example 12. Recording raw data from an unknown sensor

```bash
# modprobe pxa_cfcam sensor=0
# modprobe pxa_cfcam_pnm raw=1 bpp=1
# dd if=/dev/cfcam0 of=/tmp/image.pgm bs=2M count=1
```

4.11.4. Implementation details

`pxa_cfcam` first toggles the reset input of the sensor, and optionally configures it with I2C commands. It also sends the required number of clock pulses to cause the sensor to complete its initialization and start producing images.

Afterward, a looping list of DMA descriptor continuously reads raw data from the sensor, clocking it at the same time. Data received during the horizontal and vertical blanking intervals is discarded.

DMA interrupts are generated at the end of each line. When running in synchronous mode (without `mmap()`), `pxa_cfcam` transfers and converts data from the DMA buffer into the buffer referenced in the `read()` system call. When running in asynchronous mode (with `mmap()`), `pxa_cfcam` transfers and converts data from the DMA buffer into one half of a previously mmap'ed buffer, while userspace processes the other half. The two halves are swapped when `read()` is invoked.

`pxa_cfcam` does not use the HSYNC and VSYNC outputs of the sensor. This requires that the number of clock cycles in each frame and each sync interval be known in advance.

4.11.5. Known problems

- `pxa_cfcam` uses some signals on the CF connector in way that it not compatible with the CF standard. Do do not use it while a real CF card is inserted.
• The outputs of a CMOS sensor are not three-state, and would conflict with those of any other device connected to the IO bus. Do not use pxacfcam together with a netcf or wifistix-cf. This would require the driver to power the sensor down while accessing the peripheral, and vice-versa.

• read() does busy wait, which impacts the responsiveness of other processes. For real-time applications, the asynchronous (mmmap()) API should be used instead.

• rootfs versions older than 1161 may boot Linux with GPIO54 configured as "AF2 in" (check with cat /proc/gpio/GPIO54). This may cause conflicts on the data bus of the PXA255 leading to random segmentation faults. To reduce this risk, add "pinit on" at the beginning of bootcmd in u-boot.

5. Use cases

5.1. Remote-control with USB joystick connected to ground station

Figure 10. Remote-control with USB joystick connected to ground station

In this scenario (Figure 10, “Remote-control with USB joystick connected to ground station”), iprc_tx running on a ground station transmits commands from a USB joystick to a remote iprc_rx over Bluetooth (or Wi-Fi).

5.2. Remote-control with Bluetooth joystick

Figure 11. Remote control with a Bluetooth joystick

In this scenario (Figure 11, “Remote control with a Bluetooth joystick”) we take advantage of the fact that modern wireless joysticks are compatible with the Bluetooth HID Profile, and can therefore connect directly to an on-board Linux computer with Bluetooth module. Instructions for using the PS3 "SIXAXIS" controller can be found in [SIXLINUX].

6. Roadmap

6.1. Bluetooth QoS

Goal: Reduce datalink latency (currently about 20 ms).

Bluetooth supports SCO (Synchronous Connection-Oriented) links. This might provide lower latency than BNEP.

6.2. Software-based GPIO input debouncing

Goal: Prevent edge-triggered interrupts from possibly overloading the system.

This could be achieved by masking edge-triggered interrupts for a configurable delay after each event.

6.3. Multiple queues in pxa_ipwm

Goal: Prevent a fast signal from inhibiting the decoding of slower signals.

This could be achieved by having one queue per pin. However, this would make the FIQ handler slower, which would reduce real-time performance.
6.4. Video4Linux compatibility

`pxa_cfcam` implements functionality similar to that of V4L (with a much simpler API). It could be turned into a V4L-compliant device.

**Bibliography**


**Glossary**

**Bluetooth Network Encapsulation Protocol**
Provides an ethernet-like interface (e.g. `bnep0`) at each end of a Bluetooth connection.

**CompactFlash**
A variant of PCMCIA.

**Duty Cycle Modulation**
A modulation which encodes an analog signal into the average value of a square-wave signal. Typically used to drive an electric motor with an H-bridge.

See Also Pulse Width Modulation.

**Electronic Speed Controller**
A motor speed controller, typically driven by a PWM signal in RC models.

**General-Purpose I/O**
The PXA255 has 85 general-purpose pins which can be independently configured for input or output, or connected to integrated peripheral functions (e.g. serial ports and LCD driver). The PXA270 has 119.

**Pulse Code Modulation**
Refers to a digital transmission, in contrast with PWM or PPM.

See Also Pulse Width Modulation, Pulse Position Modulation.

**Pulse Position Modulation**
Refers to a multiple-channel differential PPM signal, in RC terminology. Typically used on the radio link in commercial RC systems.

See Also Pulse Width Modulation, Pulse Code Modulation.
Pulse Width Modulation  A modulation which encodes an analog signal into variable-duration digital pulses. Typically used to drive RC servos. See Also Pulse Position Modulation, Duty Cycle Modulation.