

```

1  /*
2  *   ALSA driver for VIA VT82xx (South Bridge)
3  *
4  *   VT82C686A/B/C, VT8233A/C, VT8235
5  *
6  *   Copyright (c) 2000 Jaroslav Kysela <perex@perex.cz>
7  *   Tjeerd.Mulder <Tjeerd.Mulder@fujitsu-siemens.com>
8  *   2002 Takashi Iwai <tiwai@suse.de>
9  *
10 *   This program is free software; you can redistribute it and/or modify
11 *   it under the terms of the GNU General Public License as published by
12 *   the Free Software Foundation; either version 2 of the License, or
13 *   (at your option) any later version.
14 *
15 *   This program is distributed in the hope that it will be useful,
16 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
17 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
18 *   GNU General Public License for more details.
19 *
20 *   You should have received a copy of the GNU General Public License
21 *   along with this program; if not, write to the Free Software
22 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
23 *
24 */
25
26 /*
27  * Changes:
28  *
29  * Dec. 19, 2002 Takashi Iwai <tiwai@suse.de>
30  *   - use the DSX channels for the first pcm playback.
31  *     (on VIA8233, 8233C and 8235 only)
32  *     this will allow you play simultaneously up to 4 streams.
33  *     multi-channel playback is assigned to the second device
34  *     on these chips.
35  *   - support the secondary capture (on VIA8233/C,8235)
36  *   - SPDIF support
37  *     the DSX3 channel can be used for SPDIF output.
38  *     on VIA8233A, this channel is assigned to the second pcm
39  *     playback.
40  *     the card config of alsa-lib will assign the correct
41  *     device for applications.
42  *   - clean up the code, separate low-level initialization
43  *     routines for each chipset.
44  *
45  * Sep. 26, 2005 Karsten Wiese <annabellesgarden@yahoo.de>
46  *   - Optimize position calculation for the 823x chips.
47  */
48
49 #include <asm/io.h>
50 #include <linux/delay.h>
51 #include <linux/interrupt.h>
52 #include <linux/init.h>
53 #include <linux/pci.h>
54 #include <linux/slab.h>
55 #include <linux/gameport.h>
56 #include <linux/module.h>
57 #include <sound/core.h>
58 #include <sound/pcm.h>
59 #include <sound/pcm_params.h>
60 #include <sound/info.h>
61 #include <sound/tlv.h>
62 #include <sound/ac97_codec.h>
63 #include <sound/mpu401.h>
64 #include <sound/initval.h>
65
66 #if 0
67 #define POINTER_DEBUG
68 #endif

```

```

69
70 MODULE_AUTHOR("Jaroslav Kysela <perex@perex.cz>");
71 MODULE_DESCRIPTION("VIA VT82xx audio");
72 MODULE_LICENSE("GPL");
73 MODULE_SUPPORTED_DEVICE(("{VIA,VT82C686A/B/C,pci},{VIA,VT8233A/C,8235}"));
74
75 #if defined(CONFIG_GAMEPORT) || (defined(MODULE) && defined(CONFIG_GAMEPORT_MODULE))
76 #define SUPPORT_JOYSTICK 1
77 #endif
78
79 static int index = SNDRV_DEFAULT_IDX1;    /* Index 0-MAX */
80 static char *id = SNDRV_DEFAULT_STR1;    /* ID for this card */
81 static long mpu_port;
82 #ifdef SUPPORT_JOYSTICK
83 static int joystick;
84 #endif
85 static int ac97_clock = 48000;
86 static char *ac97_quirk;
87 static int dxs_support;
88 static int dxs_init_volume = 31;
89 static int nodelay;
90
91 module_param(index, int, 0444);
92 MODULE_PARM_DESC(index, "Index value for VIA 82xx bridge.");
93 module_param(id, charp, 0444);
94 MODULE_PARM_DESC(id, "ID string for VIA 82xx bridge.");
95 module_param(mpu_port, long, 0444);
96 MODULE_PARM_DESC(mpu_port, "MPU-401 port. (VT82C686x only)");
97 #ifdef SUPPORT_JOYSTICK
98 module_param(joystick, bool, 0444);
99 MODULE_PARM_DESC(joystick, "Enable joystick. (VT82C686x only)");
100 #endif
101 module_param(ac97_clock, int, 0444);
102 MODULE_PARM_DESC(ac97_clock, "AC'97 codec clock (default 48000Hz).");
103 module_param(ac97_quirk, charp, 0444);
104 MODULE_PARM_DESC(ac97_quirk, "AC'97 workaround for strange hardware.");
105 module_param(dxs_support, int, 0444);
106 MODULE_PARM_DESC(dxs_support, "Support for DXS channels (0 = auto, 1 = enable, 2 =
107 disable, 3 = 48k only, 4 = no VRA, 5 = enable any sample rate)");
108 module_param(dxs_init_volume, int, 0644);
109 MODULE_PARM_DESC(dxs_init_volume, "initial DXS volume (0-31)");
110 module_param(nodelay, int, 0444);
111 MODULE_PARM_DESC(nodelay, "Disable 500ms init delay");
112
113 /* just for backward compatibility */
114 static int enable;
115 module_param(enable, bool, 0444);
116
117
118 /* revision numbers for via686 */
119 #define VIA_REV_686_A 0x10
120 #define VIA_REV_686_B 0x11
121 #define VIA_REV_686_C 0x12
122 #define VIA_REV_686_D 0x13
123 #define VIA_REV_686_E 0x14
124 #define VIA_REV_686_H 0x20
125
126 /* revision numbers for via8233 */
127 #define VIA_REV_PRE_8233 0x10 /* not in market */
128 #define VIA_REV_8233C 0x20 /* 2 rec, 4 pb, 1 multi-pb */
129 #define VIA_REV_8233 0x30 /* 2 rec, 4 pb, 1 multi-pb, spdif */
130 #define VIA_REV_8233A 0x40 /* 1 rec, 1 multi-pb, spdif */
131 #define VIA_REV_8235 0x50 /* 2 rec, 4 pb, 1 multi-pb, spdif */
132 #define VIA_REV_8237 0x60
133 #define VIA_REV_8251 0x70
134
135
136

```

```

137 /*
138  *   Direct registers
139  */
140
141 #define VIAREG(via, x) ((via)->port + VIA_REG_##x)
142 #define VIADEV_REG(viadev, x) ((viadev)->port + VIA_REG_##x)
143
144 /* common offsets */
145 #define VIA_REG_OFFSET_STATUS 0x00 /* byte - channel status */
146 #define VIA_REG_STAT_ACTIVE 0x80 /* RO */
147 #define VIA8233_SHADOW_STAT_ACTIVE 0x08 /* RO */
148 #define VIA_REG_STAT_PAUSED 0x40 /* RO */
149 #define VIA_REG_STAT_TRIGGER_QUEUED 0x08 /* RO */
150 #define VIA_REG_STAT_STOPPED 0x04 /* RWC */
151 #define VIA_REG_STAT_EOL 0x02 /* RWC */
152 #define VIA_REG_STAT_FLAG 0x01 /* RWC */
153 #define VIA_REG_OFFSET_CONTROL 0x01 /* byte - channel control */
154 #define VIA_REG_CTRL_START 0x80 /* WO */
155 #define VIA_REG_CTRL_TERMINATE 0x40 /* WO */
156 #define VIA_REG_CTRL_AUTOSTART 0x20
157 #define VIA_REG_CTRL_PAUSE 0x08 /* RW */
158 #define VIA_REG_CTRL_INT_STOP 0x04
159 #define VIA_REG_CTRL_INT_EOL 0x02
160 #define VIA_REG_CTRL_INT_FLAG 0x01
161 #define VIA_REG_CTRL_RESET 0x01 /* RW - probably reset?
162 undocumented */
163 #define VIA_REG_CTRL_INT (VIA_REG_CTRL_INT_FLAG | VIA_REG_CTRL_INT_EOL |
164 VIA_REG_CTRL_AUTOSTART)
165 #define VIA_REG_OFFSET_TYPE 0x02 /* byte - channel type (686 only)
166 */
167 #define VIA_REG_TYPE_AUTOSTART 0x80 /* RW - autostart at EOL */
168 #define VIA_REG_TYPE_16BIT 0x20 /* RW */
169 #define VIA_REG_TYPE_STEREO 0x10 /* RW */
170 #define VIA_REG_TYPE_INT_LLINE 0x00
171 #define VIA_REG_TYPE_INT_LSAMPLE 0x04
172 #define VIA_REG_TYPE_INT_LESSONE 0x08
173 #define VIA_REG_TYPE_INT_MASK 0x0c
174 #define VIA_REG_TYPE_INT_EOL 0x02
175 #define VIA_REG_TYPE_INT_FLAG 0x01
176 #define VIA_REG_OFFSET_TABLE_PTR 0x04 /* dword - channel table pointer */
177 #define VIA_REG_OFFSET_CURR_PTR 0x04 /* dword - channel current pointer
178 */
179 #define VIA_REG_OFFSET_STOP_IDX 0x08 /* dword - stop index, channel
180 type, sample rate */
181 #define VIA8233_REG_TYPE_16BIT 0x00200000 /* RW */
182 #define VIA8233_REG_TYPE_STEREO 0x00100000 /* RW */
183 #define VIA_REG_OFFSET_CURR_COUNT 0x0c /* dword - channel current count (24 bit)
184 */
185 #define VIA_REG_OFFSET_CURR_INDEX 0x0f /* byte - channel current index (for
186 via8233 only) */
187
188 #define DEFINE_VIA_REGSET(name, val) \
189 enum {\
190     VIA_REG_##name##_STATUS = (val),\
191     VIA_REG_##name##_CONTROL = (val) + 0x01,\
192     VIA_REG_##name##_TYPE = (val) + 0x02,\
193     VIA_REG_##name##_TABLE_PTR = (val) + 0x04,\
194     VIA_REG_##name##_CURR_PTR = (val) + 0x04,\
195     VIA_REG_##name##_STOP_IDX = (val) + 0x08,\
196     VIA_REG_##name##_CURR_COUNT = (val) + 0x0c,\
197 }
198
199 /* playback block */
200 DEFINE_VIA_REGSET(PLAYBACK, 0x00);
201 DEFINE_VIA_REGSET(CAPTURE, 0x10);
202 DEFINE_VIA_REGSET(FM, 0x20);
203
204

```

```

205
206 /* AC'97 */
207 #define VIA_REG_AC97 0x80 /* dword */
208 #define VIA_REG_AC97_CODEC_ID_MASK (3<<30)
209 #define VIA_REG_AC97_CODEC_ID_SHIFT 30
210 #define VIA_REG_AC97_CODEC_ID_PRIMARY 0x00
211 #define VIA_REG_AC97_CODEC_ID_SECONDARY 0x01
212 #define VIA_REG_AC97_SECONDARY_VALID (1<<27)
213 #define VIA_REG_AC97_PRIMARY_VALID (1<<25)
214 #define VIA_REG_AC97_BUSY (1<<24)
215 #define VIA_REG_AC97_READ (1<<23)
216 #define VIA_REG_AC97_CMD_SHIFT 16
217 #define VIA_REG_AC97_CMD_MASK 0x7e
218 #define VIA_REG_AC97_DATA_SHIFT 0
219 #define VIA_REG_AC97_DATA_MASK 0xffff
220
221 #define VIA_REG_SGD_SHADOW 0x84 /* dword */
222 /* via686 */
223 #define VIA_REG_SGD_STAT_PB_FLAG (1<<0)
224 #define VIA_REG_SGD_STAT_CP_FLAG (1<<1)
225 #define VIA_REG_SGD_STAT_FM_FLAG (1<<2)
226 #define VIA_REG_SGD_STAT_PB_EOL (1<<4)
227 #define VIA_REG_SGD_STAT_CP_EOL (1<<5)
228 #define VIA_REG_SGD_STAT_FM_EOL (1<<6)
229 #define VIA_REG_SGD_STAT_PB_STOP (1<<8)
230 #define VIA_REG_SGD_STAT_CP_STOP (1<<9)
231 #define VIA_REG_SGD_STAT_FM_STOP (1<<10)
232 #define VIA_REG_SGD_STAT_PB_ACTIVE (1<<12)
233 #define VIA_REG_SGD_STAT_CP_ACTIVE (1<<13)
234 #define VIA_REG_SGD_STAT_FM_ACTIVE (1<<14)
235 /* via8233 */
236 #define VIA8233_REG_SGD_STAT_FLAG (1<<0)
237 #define VIA8233_REG_SGD_STAT_EOL (1<<1)
238 #define VIA8233_REG_SGD_STAT_STOP (1<<2)
239 #define VIA8233_REG_SGD_STAT_ACTIVE (1<<3)
240 #define VIA8233_INTR_MASK(chan)
241 ((VIA8233_REG_SGD_STAT_FLAG|VIA8233_REG_SGD_STAT_EOL) << ((chan) * 4))
242 #define VIA8233_REG_SGD_CHAN_SDX 0
243 #define VIA8233_REG_SGD_CHAN_MULT 4
244 #define VIA8233_REG_SGD_CHAN_REC 6
245 #define VIA8233_REG_SGD_CHAN_REC1 7
246
247 #define VIA_REG_GPI_STATUS 0x88
248 #define VIA_REG_GPI_INTR 0x8c
249
250 /* multi-channel and capture registers for via8233 */
251 #define VIA_REGSET(MULTPLAY, 0x40);
252 #define VIA_REGSET(CAPTURE_8233, 0x60);
253
254 /* via8233-specific registers */
255 #define VIA_REG_OFS_PLAYBACK_VOLUME_L 0x02 /* byte */
256 #define VIA_REG_OFS_PLAYBACK_VOLUME_R 0x03 /* byte */
257 #define VIA_REG_OFS_MULTPLAY_FORMAT 0x02 /* byte - format and channels */
258 #define VIA_REG_MULTPLAY_FMT_8BIT 0x00
259 #define VIA_REG_MULTPLAY_FMT_16BIT 0x80
260 #define VIA_REG_MULTPLAY_FMT_CH_MASK 0x70 /* # channels <= 4 (valid =
261 1,2,4,6) */
262 #define VIA_REG_OFS_CAPTURE_FIFO 0x02 /* byte - bit 6 = fifo enable */
263 #define VIA_REG_CAPTURE_FIFO_ENABLE 0x40
264
265 #define VIA_DXS_MAX_VOLUME 31 /* max. volume (attenuation) of
266 reg 0x32/33 */
267
268 #define VIA_REG_CAPTURE_CHANNEL 0x63 /* byte - input select */
269 #define VIA_REG_CAPTURE_CHANNEL_MIC 0x4
270 #define VIA_REG_CAPTURE_CHANNEL_LINE 0
271 #define VIA_REG_CAPTURE_SELECT_CODEC 0x03 /* recording source codec (0 =
272 primary) */

```

```

273
274 #define VIA_TBL_BIT_FLAG 0x40000000
275 #define VIA_TBL_BIT_EOL 0x80000000
276
277 /* pci space */
278 #define VIA_ACLINK_STAT 0x40
279 #define VIA_ACLINK_C11_READY 0x20
280 #define VIA_ACLINK_C10_READY 0x10
281 #define VIA_ACLINK_C01_READY 0x04 /* secondary codec ready */
282 #define VIA_ACLINK_LOWPOWER 0x02 /* low-power state */
283 #define VIA_ACLINK_C00_READY 0x01 /* primary codec ready */
284 #define VIA_ACLINK_CTRL 0x41
285 #define VIA_ACLINK_CTRL_ENABLE 0x80 /* 0: disable, 1: enable */
286 #define VIA_ACLINK_CTRL_RESET 0x40 /* 0: assert, 1: de-assert */
287 #define VIA_ACLINK_CTRL_SYNC 0x20 /* 0: release SYNC, 1: force SYNC hi */
288 #define VIA_ACLINK_CTRL_SDO 0x10 /* 0: release SDO, 1: force SDO hi */
289 #define VIA_ACLINK_CTRL_VRA 0x08 /* 0: disable VRA, 1: enable VRA */
290 #define VIA_ACLINK_CTRL_PCM 0x04 /* 0: disable PCM, 1: enable PCM */
291 #define VIA_ACLINK_CTRL_FM 0x02 /* via686 only */
292 #define VIA_ACLINK_CTRL_SB 0x01 /* via686 only */
293 #define VIA_ACLINK_CTRL_INIT (VIA_ACLINK_CTRL_ENABLE|\
294 VIA_ACLINK_CTRL_RESET|\
295 VIA_ACLINK_CTRL_PCM|\
296 VIA_ACLINK_CTRL_VRA)
297 #define VIA_FUNC_ENABLE 0x42
298 #define VIA_FUNC_MIDI_PNP 0x80 /* FIXME: it's 0x40 in the datasheet! */
299 #define VIA_FUNC_MIDI_IRQMASK 0x40 /* FIXME: not documented! */
300 #define VIA_FUNC_RX2C_WRITE 0x20
301 #define VIA_FUNC_SB_FIFO_EMPTY 0x10
302 #define VIA_FUNC_ENABLE_GAME 0x08
303 #define VIA_FUNC_ENABLE_FM 0x04
304 #define VIA_FUNC_ENABLE_MIDI 0x02
305 #define VIA_FUNC_ENABLE_SB 0x01
306 #define VIA_PNP_CONTROL 0x43
307 #define VIA_FM_NMI_CTRL 0x48
308 #define VIA8233_VOLCHG_CTRL 0x48
309 #define VIA8233_SPDIF_CTRL 0x49
310 #define VIA8233_SPDIF_DX3 0x08
311 #define VIA8233_SPDIF_SLOT_MASK 0x03
312 #define VIA8233_SPDIF_SLOT_1011 0x00
313 #define VIA8233_SPDIF_SLOT_34 0x01
314 #define VIA8233_SPDIF_SLOT_78 0x02
315 #define VIA8233_SPDIF_SLOT_69 0x03
316
317 /*
318 */
319
320 #define VIA_DXS_AUTO 0
321 #define VIA_DXS_ENABLE 1
322 #define VIA_DXS_DISABLE 2
323 #define VIA_DXS_48K 3
324 #define VIA_DXS_NO_VRA 4
325 #define VIA_DXS_SRC 5
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340

```

```

341  /*
342  * pcm stream
343  */
344
345  struct snd_via_sg_table {
346      unsigned int offset;
347      unsigned int size;
348  } ;
349
350  #define VIA_TABLE_SIZE    255
351  #define VIA_MAX_BUFSIZE   (1<<24)
352
353  struct viadev {
354      unsigned int reg_offset;
355      unsigned long port;
356      int direction; /* playback = 0, capture = 1 */
357      struct snd_pcm_substream *substream;
358      int running;
359      unsigned int tbl_entries; /* # descriptors */
360      struct snd_dma_buffer table;
361      struct snd_via_sg_table *idx_table;
362      /* for recovery from the unexpected pointer */
363      unsigned int lastpos;
364      unsigned int fragsize;
365      unsigned int bufsize;
366      unsigned int bufsize2;
367      int hwptr_done; /* processed frame position in the buffer */
368      int in_interrupt;
369      int shadow_shift;
370  };
371
372
373  enum { TYPE_CARD_VIA686 = 1, TYPE_CARD_VIA8233 };
374  enum { TYPE_VIA686, TYPE_VIA8233, TYPE_VIA8233A };
375
376  #define VIA_MAX_DEVS      7 /* 4 playback, 1 multi, 2 capture */
377
378  struct via_rate_lock {
379      spinlock_t lock;
380      int rate;
381      int used;
382  };
383
384  struct via82xx {
385      int irq;
386
387      unsigned long port;
388      struct resource *mpu_res;
389      int chip_type;
390      unsigned char revision;
391
392      unsigned char old_legacy;
393      unsigned char old_legacy_cfg;
394  #ifdef CONFIG_PM
395      unsigned char legacy_saved;
396      unsigned char legacy_cfg_saved;
397      unsigned char spdif_ctrl_saved;
398      unsigned char capture_src_saved[2];
399      unsigned int mpu_port_saved;
400  #endif
401
402      unsigned char playback_volume[4][2]; /* for VIA8233/C/8235; default = 0 */
403      unsigned char playback_volume_c[2]; /* for VIA8233/C/8235; default = 0 */
404
405      unsigned int intr_mask; /* SGD_SHADOW mask to check interrupts */
406
407      struct pci_dev *pci;
408      struct snd_card *card;

```

```

409     unsigned int num_devs;
410     unsigned int playback_devno, multi_devno, capture_devno;
411     struct viadev devs[VIA_MAX_DEVS];
412     struct via_rate_lock rates[2]; /* playback and capture */
413     unsigned int dxs_fixed: 1; /* DXS channel accepts only 48kHz */
414     unsigned int no_vra: 1; /* no need to set VRA on DXS channels */
415     unsigned int dxs_src: 1; /* use full SRC capabilities of DXS */
416     unsigned int spdif_on: 1; /* only spdif rates work to external DACs */
417
418     struct snd_pcm *pcms[2];
419     struct snd_rawmidi *rmidi;
420     struct snd_kcontrol *dxs_controls[4];
421
422     struct snd_ac97_bus *ac97_bus;
423     struct snd_ac97 *ac97;
424     unsigned int ac97_clock;
425     unsigned int ac97_secondary; /* secondary AC'97 codec is present */
426
427     spinlock_t reg_lock;
428     struct snd_info_entry *proc_entry;
429
430 #ifdef SUPPORT_JOYSTICK
431     struct gameport *gameport;
432 #endif
433 };
434
435 static DEFINE_PCI_DEVICE_TABLE(snd_via82xx_ids) = {
436     /* 0x1106, 0x3058 */
437     { PCI_VDEVICE(VIA, PCI_DEVICE_ID_VIA_82C686_5), TYPE_CARD_VIA686, },
438     /* 686A */
439     /* 0x1106, 0x3059 */
440     { PCI_VDEVICE(VIA, PCI_DEVICE_ID_VIA_8233_5), TYPE_CARD_VIA8233, }, /*
441 VT8233 */
442     { 0, }
443 };
444
445 MODULE_DEVICE_TABLE(pci, snd_via82xx_ids);
446
447 /*
448 */
449
450 /*
451 * allocate and initialize the descriptor buffers
452 * periods = number of periods
453 * fragsize = period size in bytes
454 */
455
456 static int build_via_table(struct viadev *dev, struct snd_pcm_substream *substream,
457                          struct pci_dev *pci,
458                          unsigned int periods, unsigned int fragsize)
459 {
460     unsigned int i, idx, ofs, rest;
461     struct via82xx *chip = snd_pcm_substream_chip(substream);
462
463     if (dev->table.area == NULL) {
464         /* the start of each lists must be aligned to 8 bytes,
465          * but the kernel pages are much bigger, so we don't care
466          */
467         if (snd_dma_alloc_pages(SNDRV_DMA_TYPE_DEV, snd_dma_pci_data(chip-
468 >pci),
469                                PAGE_ALIGN(VIA_TABLE_SIZE * 2 * 8),
470                                &dev->table) < 0)
471             return -ENOMEM;
472     }
473     if (! dev->idx_table) {
474         dev->idx_table = kmalloc(sizeof(*dev->idx_table) * VIA_TABLE_SIZE,
475 GFP_KERNEL);
476     }

```



```

477     if (! dev->idx_table)
478         return -ENOMEM;
479     }
480
481     /* fill the entries */
482     idx = 0;
483     ofs = 0;
484     for (i = 0; i < periods; i++) {
485         rest = fragsize;
486         /* fill descriptors for a period.
487          * a period can be split to several descriptors if it's
488          * over page boundary.
489          */
490         do {
491             unsigned int r;
492             unsigned int flag;
493             unsigned int addr;
494
495             if (idx >= VIA_TABLE_SIZE) {
496                 snd_printk(KERN_ERR "via82xx: too much table
497 size!\n");
498                 return -EINVAL;
499             }
500             addr = snd_pcm_sgbuf_get_addr(substream, ofs);
501             ((u32 *)dev->table.area)[idx << 1] = cpu_to_le32(addr);
502             r = snd_pcm_sgbuf_get_chunk_size(substream, ofs, rest);
503             rest -= r;
504             if (! rest) {
505                 if (i == periods - 1)
506                     flag = VIA_TBL_BIT_EOL; /* buffer boundary
507 */
508                 else
509                     flag = VIA_TBL_BIT_FLAG; /* period
510 boundary */
511             } else
512                 flag = 0; /* period continues to the next */
513             /*
514              printk(KERN_DEBUG "via: tbl %d: at %d size %d "
515                     "(rest %d)\n", idx, ofs, r, rest);
516              */
517             ((u32 *)dev->table.area)[(idx<<1) + 1] = cpu_to_le32(r |
518 flag);
519             dev->idx_table[idx].offset = ofs;
520             dev->idx_table[idx].size = r;
521             ofs += r;
522             idx++;
523         } while (rest > 0);
524     }
525     dev->tbl_entries = idx;
526     dev->bufsize = periods * fragsize;
527     dev->bufsize2 = dev->bufsize / 2;
528     dev->fragsize = fragsize;
529     return 0;
530 }
531
532
533 static int clean_via_table(struct viadev *dev, struct snd_pcm_substream *substream,
534                          struct pci_dev *pci)
535 {
536     if (dev->table.area) {
537         snd_dma_free_pages(&dev->table);
538         dev->table.area = NULL;
539     }
540     kfree(dev->idx_table);
541     dev->idx_table = NULL;
542     return 0;
543 }
544

```

```

545  /*
546  *   Basic I/O
547  */
548
549  static inline unsigned int snd_via82xx_codec_xread(struct via82xx *chip)
550  {
551      return inl(VIAREG(chip, AC97));
552  }
553
554  static inline void snd_via82xx_codec_xwrite(struct via82xx *chip, unsigned int val)
555  {
556      outl(val, VIAREG(chip, AC97));
557  }
558
559  static int snd_via82xx_codec_ready(struct via82xx *chip, int secondary)
560  {
561      unsigned int timeout = 1000;      /* 1ms */
562      unsigned int val;
563
564      while (timeout-- > 0) {
565          udelay(1);
566          if (!(val = snd_via82xx_codec_xread(chip)) & VIA_REG_AC97_BUSY))
567              return val & 0xffff;
568      }
569      snd_printk(KERN_ERR "codec_ready: codec %i is not ready [0x%x]\n",
570                  secondary, snd_via82xx_codec_xread(chip));
571      return -EIO;
572  }
573
574  static int snd_via82xx_codec_valid(struct via82xx *chip, int secondary)
575  {
576      unsigned int timeout = 1000;      /* 1ms */
577      unsigned int val, val1;
578      unsigned int stat = !secondary ? VIA_REG_AC97_PRIMARY_VALID :
579                               VIA_REG_AC97_SECONDARY_VALID;
580
581      while (timeout-- > 0) {
582          val = snd_via82xx_codec_xread(chip);
583          val1 = val & (VIA_REG_AC97_BUSY | stat);
584          if (val1 == stat)
585              return val & 0xffff;
586          udelay(1);
587      }
588      return -EIO;
589  }
590
591  static void snd_via82xx_codec_wait(struct snd_ac97 *ac97)
592  {
593      struct via82xx *chip = ac97->private_data;
594      int err;
595      err = snd_via82xx_codec_ready(chip, ac97->num);
596      /* here we need to wait fairly for long time.. */
597      if (!nodelay)
598          msleep(500);
599  }
600
601
602
603
604
605
606
607
608
609
610
611
612

```

```

613 static void snd_via82xx_codec_write(struct snd_ac97 *ac97,
614                                     unsigned short reg,
615                                     unsigned short val)
616 {
617     struct via82xx *chip = ac97->private_data;
618     unsigned int xval;
619
620     xval = !ac97->num ? VIA_REG_AC97_CODEC_ID_PRIMARY :
621 VIA_REG_AC97_CODEC_ID_SECONDARY;
622     xval <= VIA_REG_AC97_CODEC_ID_SHIFT;
623     xval |= reg << VIA_REG_AC97_CMD_SHIFT;
624     xval |= val << VIA_REG_AC97_DATA_SHIFT;
625     snd_via82xx_codec_xwrite(chip, xval);
626     snd_via82xx_codec_ready(chip, ac97->num);
627 }
628
629 static unsigned short snd_via82xx_codec_read(struct snd_ac97 *ac97, unsigned short
630 reg)
631 {
632     struct via82xx *chip = ac97->private_data;
633     unsigned int xval, val = 0xffff;
634     int again = 0;
635
636     xval = ac97->num << VIA_REG_AC97_CODEC_ID_SHIFT;
637     xval |= ac97->num ? VIA_REG_AC97_SECONDARY_VALID :
638 VIA_REG_AC97_PRIMARY_VALID;
639     xval |= VIA_REG_AC97_READ;
640     xval |= (reg & 0x7f) << VIA_REG_AC97_CMD_SHIFT;
641     while (1) {
642         if (again++ > 3) {
643             snd_printk(KERN_ERR "codec_read: codec %i is not valid
644 [0x%x]\n",
645                         ac97->num, snd_via82xx_codec_xread(chip));
646             return 0xffff;
647         }
648         snd_via82xx_codec_xwrite(chip, xval);
649         udelay (20);
650         if (snd_via82xx_codec_valid(chip, ac97->num) >= 0) {
651             udelay(25);
652             val = snd_via82xx_codec_xread(chip);
653             break;
654         }
655     }
656     return val & 0xffff;
657 }
658
659 static void snd_via82xx_channel_reset(struct via82xx *chip, struct viadev *viadev)
660 {
661     outb(VIA_REG_CTRL_PAUSE | VIA_REG_CTRL_TERMINATE | VIA_REG_CTRL_RESET,
662          VIADEV_REG(viadev, OFFSET_CONTROL));
663     inb(VIADEV_REG(viadev, OFFSET_CONTROL));
664     udelay(50);
665     /* disable interrupts */
666     outb(0x00, VIADEV_REG(viadev, OFFSET_CONTROL));
667     /* clear interrupts */
668     outb(0x03, VIADEV_REG(viadev, OFFSET_STATUS));
669     outb(0x00, VIADEV_REG(viadev, OFFSET_TYPE)); /* for via686 */
670     // outl(0, VIADEV_REG(viadev, OFFSET_CURR_PTR));
671     viadev->lastpos = 0;
672     viadev->hwptr_done = 0;
673 }
674
675
676
677
678
679
680

```

```

681  /*
682  *   Interrupt handler
683  *   Used for 686 and 8233A
684  */
685  static irqreturn_t snd_via686_interrupt(int irq, void *dev_id)
686  {
687      struct via82xx *chip = dev_id;
688      unsigned int status;
689      unsigned int i;
690
691      status = inl(VIAREG(chip, SGD_SHADOW));
692      if (!(status & chip->intr_mask)) {
693          if (chip->rmidi)
694              /* check mpu401 interrupt */
695              return snd_mpu401_uart_interrupt(irq, chip->rmidi-
696 >private_data);
697          return IRQ_NONE;
698      }
699
700      /* check status for each stream */
701      spin_lock(&chip->reg_lock);
702      for (i = 0; i < chip->num_devs; i++) {
703          struct viadev *viadev = &chip->devs[i];
704          unsigned char c_status = inb(VIADEV_REG(viadev, OFFSET_STATUS));
705          if (!(c_status &
706 (VIA_REG_STAT_EOL|VIA_REG_STAT_FLAG|VIA_REG_STAT_STOPPED)))
707              continue;
708          if (viadev->substream && viadev->running) {
709              /*
710               * Update hwptr_done based on 'period elapsed'
711               * interrupts. We'll use it, when the chip returns 0
712               * for OFFSET_CURR_COUNT.
713               */
714              if (c_status & VIA_REG_STAT_EOL)
715                  viadev->hwptr_done = 0;
716              else
717                  viadev->hwptr_done += viadev->fragsize;
718              viadev->in_interrupt = c_status;
719              spin_unlock(&chip->reg_lock);
720              snd_pcm_period_elapsed(viadev->substream);
721              spin_lock(&chip->reg_lock);
722              viadev->in_interrupt = 0;
723          }
724          outb(c_status, VIADEV_REG(viadev, OFFSET_STATUS)); /* ack */
725      }
726      spin_unlock(&chip->reg_lock);
727      return IRQ_HANDLED;
728  }
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748

```

```

749  /*
750  *   Interrupt handler
751  */
752  static irqreturn_t snd_via8233_interrupt(int irq, void *dev_id)
753  {
754      struct via82xx *chip = dev_id;
755      unsigned int status;
756      unsigned int i;
757      int irqreturn = 0;
758
759      /* check status for each stream */
760      spin_lock(&chip->reg_lock);
761      status = inl(VIAREG(chip, SGD_SHADOW));
762
763      for (i = 0; i < chip->num_devs; i++) {
764          struct viadev *viadev = &chip->devs[i];
765          struct snd_pcm_substream *substream;
766          unsigned char c_status, shadow_status;
767
768          shadow_status = (status >> viadev->shadow_shift) &
769                          (VIA8233_SHADOW_STAT_ACTIVE|VIA_REG_STAT_EOL|
770                           VIA_REG_STAT_FLAG);
771          c_status = shadow_status & (VIA_REG_STAT_EOL|VIA_REG_STAT_FLAG);
772          if (!c_status)
773              continue;
774
775          substream = viadev->substream;
776          if (substream && viadev->running) {
777              /*
778               * Update hwptr_done based on 'period elapsed'
779               * interrupts. We'll use it, when the chip returns 0
780               * for OFFSET_CURR_COUNT.
781               */
782              if (c_status & VIA_REG_STAT_EOL)
783                  viadev->hwptr_done = 0;
784              else
785                  viadev->hwptr_done += viadev->fragsize;
786              viadev->in_interrupt = c_status;
787              if (shadow_status & VIA8233_SHADOW_STAT_ACTIVE)
788                  viadev->in_interrupt |= VIA_REG_STAT_ACTIVE;
789              spin_unlock(&chip->reg_lock);
790
791              snd_pcm_period_elapsed(substream);
792
793              spin_lock(&chip->reg_lock);
794              viadev->in_interrupt = 0;
795          }
796          outb(c_status, VIADEV_REG(viadev, OFFSET_STATUS)); /* ack */
797          irqreturn = 1;
798      }
799      spin_unlock(&chip->reg_lock);
800      return IRQ_RETVAL(irqreturn);
801  }
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816

```

```

817  /*
818  *   PCM callbacks
819  */
820
821  /*
822  *   trigger callback
823  */
824  static int snd_via82xx_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
825  {
826      struct via82xx *chip = snd_pcm_substream_chip(substream);
827      struct viadev *viadev = substream->runtime->private_data;
828      unsigned char val;
829
830      if (chip->chip_type != TYPE_VIA686)
831          val = VIA_REG_CTRL_INT;
832      else
833          val = 0;
834      switch (cmd) {
835      case SNDRV_PCM_TRIGGER_START:
836      case SNDRV_PCM_TRIGGER_RESUME:
837          val |= VIA_REG_CTRL_START;
838          viadev->running = 1;
839          break;
840      case SNDRV_PCM_TRIGGER_STOP:
841      case SNDRV_PCM_TRIGGER_SUSPEND:
842          val = VIA_REG_CTRL_TERMINATE;
843          viadev->running = 0;
844          break;
845      case SNDRV_PCM_TRIGGER_PAUSE_PUSH:
846          val |= VIA_REG_CTRL_PAUSE;
847          viadev->running = 0;
848          break;
849      case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
850          viadev->running = 1;
851          break;
852      default:
853          return -EINVAL;
854      }
855      outb(val, VIADEV_REG(viadev, OFFSET_CONTROL));
856      if (cmd == SNDRV_PCM_TRIGGER_STOP)
857          snd_via82xx_channel_reset(chip, viadev);
858      return 0;
859  }
860
861  /*
862  *   pointer callbacks
863  */
864
865  /*
866  *   calculate the linear position at the given sg-buffer index and the rest count
867  */
868
869  #define check_invalid_pos(viadev,pos) \
870      ((pos) < viadev->lastpos && ((pos) >= viadev->bufsize2 || \
871                                     viadev->lastpos < viadev->bufsize2))
872
873  static inline unsigned int calc_linear_pos(struct viadev *viadev, unsigned int idx,
874                                             unsigned int count)
875  {
876      unsigned int size, base, res;
877
878      size = viadev->idx_table[idx].size;
879      base = viadev->idx_table[idx].offset;
880      res = base + size - count;
881      if (res >= viadev->bufsize)
882          res -= viadev->bufsize;
883
884

```

```

885     /* check the validity of the calculated position */
886     if (size < count) {
887         snd_printd(KERN_ERR "invalid via82xx_cur_ptr (size = %d, count =
888         %d)\n",
889         (int)size, (int)count);
890         res = viadev->lastpos;
891     } else {
892         if (!count) {
893             /* Some mobos report count = 0 on the DMA boundary,
894             * i.e. count = size indeed.
895             * Let's check whether this step is above the expected
896             size.
897             */
898             int delta = res - viadev->lastpos;
899             if (delta < 0)
900                 delta += viadev->bufsize;
901             if ((unsigned int)delta > viadev->fragsize)
902                 res = base;
903         }
904         if (check_invalid_pos(viadev, res)) {
905 #ifdef POINTER_DEBUG
906             printk(KERN_DEBUG "fail: idx = %i/%i, lastpos = 0x%x, "
907             "bufsize2 = 0x%x, offsize = 0x%x, size = 0x%x, "
908             "count = 0x%x\n", idx, viadev->tbl_entries,
909             viadev->lastpos, viadev->bufsize2,
910             viadev->idx_table[idx].offset,
911             viadev->idx_table[idx].size, count);
912 #endif
913             /* count register returns full size when end of buffer is
914             reached */
915             res = base + size;
916             if (check_invalid_pos(viadev, res)) {
917                 snd_printd(KERN_ERR "invalid via82xx_cur_ptr (2), "
918                 "using last valid pointer\n");
919                 res = viadev->lastpos;
920             }
921         }
922     }
923     return res;
924 }
925
926 /*
927  * get the current pointer on via686
928  */
929 static snd_pcm_uframes_t snd_via686_pcm_pointer(struct snd_pcm_substream *substream)
930 {
931     struct via82xx *chip = snd_pcm_substream_chip(substream);
932     struct viadev *viadev = substream->runtime->private_data;
933     unsigned int idx, ptr, count, res;
934
935     if (snd_BUG_ON(!viadev->tbl_entries))
936         return 0;
937     if (!(inb(VIADEV_REG(viadev, OFFSET_STATUS)) & VIA_REG_STAT_ACTIVE))
938         return 0;
939
940     spin_lock(&chip->reg_lock);
941     count = inl(VIADEV_REG(viadev, OFFSET_CURR_COUNT)) & 0xffffffff;
942     /* The via686a does not have the current index register,
943     * so we need to calculate the index from CURR_PTR.
944     */
945     ptr = inl(VIADEV_REG(viadev, OFFSET_CURR_PTR));
946     if (ptr <= (unsigned int)viadev->table.addr)
947         idx = 0;
948     else /* CURR_PTR holds the address + 8 */
949         idx = ((ptr - (unsigned int)viadev->table.addr) / 8 - 1) % viadev-
950 >tbl_entries;
951
952

```

```

953     res = calc_linear_pos(viadev, idx, count);
954     viadev->lastpos = res; /* remember the last position */
955     spin_unlock(&chip->reg_lock);
956
957     return bytes_to_frames(substream->runtime, res);
958 }
959
960 /*
961  * get the current pointer on via823x
962  */
963 static snd_pcm_uframes_t snd_via8233_pcm_pointer(struct snd_pcm_substream
964 *substream)
965 {
966     struct via82xx *chip = snd_pcm_substream_chip(substream);
967     struct viadev *viadev = substream->runtime->private_data;
968     unsigned int idx, count, res;
969     int status;
970
971     if (snd_BUG_ON(!viadev->tbl_entries))
972         return 0;
973
974     spin_lock(&chip->reg_lock);
975     count = inl(VIADEV_REG(viadev, OFFSET_CURR_COUNT));
976     status = viadev->in_interrupt;
977     if (!status)
978         status = inb(VIADEV_REG(viadev, OFFSET_STATUS));
979
980     /* An apparent bug in the 8251 is worked around by sending a
981      * REG_CTRL_START. */
982     if (chip->revision == VIA_REV_8251 && (status & VIA_REG_STAT_EOL))
983         snd_via82xx_pcm_trigger(substream, SNDRV_PCM_TRIGGER_START);
984
985     if (!(status & VIA_REG_STAT_ACTIVE)) {
986         res = 0;
987         goto unlock;
988     }
989     if (count & 0xffffffff) {
990         idx = count >> 24;
991         if (idx >= viadev->tbl_entries) {
992 #ifdef POINTER_DEBUG
993             printk(KERN_DEBUG "fail: invalid idx = %i/%i\n", idx,
994                    viadev->tbl_entries);
995 #endif
996             res = viadev->lastpos;
997         } else {
998             count &= 0xffffffff;
999             res = calc_linear_pos(viadev, idx, count);
1000         }
1001     } else {
1002         res = viadev->hwptr_done;
1003         if (!viadev->in_interrupt) {
1004             if (status & VIA_REG_STAT_EOL) {
1005                 res = 0;
1006             } else
1007                 if (status & VIA_REG_STAT_FLAG) {
1008                     res += viadev->fragsize;
1009                 }
1010         }
1011     }
1012 unlock:
1013     viadev->lastpos = res;
1014     spin_unlock(&chip->reg_lock);
1015
1016     return bytes_to_frames(substream->runtime, res);
1017 }
1018
1019
1020

```



```

1021  /*
1022  * hw_params callback:
1023  * allocate the buffer and build up the buffer description table
1024  */
1025  static int snd_via82xx_hw_params(struct snd_pcm_substream *substream,
1026                                  struct snd_pcm_hw_params *hw_params)
1027  {
1028      struct via82xx *chip = snd_pcm_substream_chip(substream);
1029      struct viadev *viadev = substream->runtime->private_data;
1030      int err;
1031
1032      err = snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
1033      if (err < 0)
1034          return err;
1035      err = build_via_table(viadev, substream, chip->pci,
1036                           params_periods(hw_params),
1037                           params_period_bytes(hw_params));
1038      if (err < 0)
1039          return err;
1040
1041      return 0;
1042  }
1043
1044  /*
1045  * hw_free callback:
1046  * clean up the buffer description table and release the buffer
1047  */
1048  static int snd_via82xx_hw_free(struct snd_pcm_substream *substream)
1049  {
1050      struct via82xx *chip = snd_pcm_substream_chip(substream);
1051      struct viadev *viadev = substream->runtime->private_data;
1052
1053      clean_via_table(viadev, substream, chip->pci);
1054      snd_pcm_lib_free_pages(substream);
1055      return 0;
1056  }
1057
1058  /*
1059  * set up the table pointer
1060  */
1061  static void snd_via82xx_set_table_ptr(struct via82xx *chip, struct viadev *viadev)
1062  {
1063      snd_via82xx_codec_ready(chip, 0);
1064      outl((u32)viadev->table.addr, VIADEV_REG(viadev, OFFSET_TABLE_PTR));
1065      udelay(20);
1066      snd_via82xx_codec_ready(chip, 0);
1067  }
1068
1069  /*
1070  * prepare callback for playback and capture on via686
1071  */
1072  static void via686_setup_format(struct via82xx *chip, struct viadev *viadev,
1073                                  struct snd_pcm_runtime *runtime)
1074  {
1075      snd_via82xx_channel_reset(chip, viadev);
1076      /* this must be set after channel_reset */
1077      snd_via82xx_set_table_ptr(chip, viadev);
1078      outb(VIA_REG_TYPE_AUTOSTART |
1079           (runtime->format == SNDRV_PCM_FORMAT_S16_LE ? VIA_REG_TYPE_16BIT : 0) |
1080           (runtime->channels > 1 ? VIA_REG_TYPE_STEREO : 0) |
1081           ((viadev->reg_offset & 0x10) == 0 ? VIA_REG_TYPE_INT_LSAMPLE : 0) |
1082           VIA_REG_TYPE_INT_EOL |
1083           VIA_REG_TYPE_INT_FLAG, VIADEV_REG(viadev, OFFSET_TYPE));
1084  }
1085
1086
1087
1088

```

```

1089 static int snd_via686_playback_prepare(struct snd_pcm_substream *substream)
1090 {
1091     struct via82xx *chip = snd_pcm_substream_chip(substream);
1092     struct viadev *viadev = substream->runtime->private_data;
1093     struct snd_pcm_runtime *runtime = substream->runtime;
1094
1095     snd_ac97_set_rate(chip->ac97, AC97_PCM_FRONT_DAC_RATE, runtime->rate);
1096     snd_ac97_set_rate(chip->ac97, AC97_SPDIF, runtime->rate);
1097     via686_setup_format(chip, viadev, runtime);
1098     return 0;
1099 }
1100
1101 static int snd_via686_capture_prepare(struct snd_pcm_substream *substream)
1102 {
1103     struct via82xx *chip = snd_pcm_substream_chip(substream);
1104     struct viadev *viadev = substream->runtime->private_data;
1105     struct snd_pcm_runtime *runtime = substream->runtime;
1106
1107     snd_ac97_set_rate(chip->ac97, AC97_PCM_LR_ADC_RATE, runtime->rate);
1108     via686_setup_format(chip, viadev, runtime);
1109     return 0;
1110 }
1111
1112 /*
1113  * lock the current rate
1114  */
1115 static int via_lock_rate(struct via_rate_lock *rec, int rate)
1116 {
1117     int changed = 0;
1118
1119     spin_lock_irq(&rec->lock);
1120     if (rec->rate != rate) {
1121         if (rec->rate && rec->used > 1) /* already set */
1122             changed = -EINVAL;
1123         else {
1124             rec->rate = rate;
1125             changed = 1;
1126         }
1127     }
1128     spin_unlock_irq(&rec->lock);
1129     return changed;
1130 }
1131
1132 /*
1133  * prepare callback for DSX playback on via823x
1134  */
1135 static int snd_via8233_playback_prepare(struct snd_pcm_substream *substream)
1136 {
1137     struct via82xx *chip = snd_pcm_substream_chip(substream);
1138     struct viadev *viadev = substream->runtime->private_data;
1139     struct snd_pcm_runtime *runtime = substream->runtime;
1140     int ac97_rate = chip->dxs_src ? 48000 : runtime->rate;
1141     int rate_changed;
1142     u32 rbits;
1143
1144     if ((rate_changed = via_lock_rate(&chip->rates[0], ac97_rate)) < 0)
1145         return rate_changed;
1146     if (rate_changed)
1147         snd_ac97_set_rate(chip->ac97, AC97_PCM_FRONT_DAC_RATE,
1148             chip->no_vra ? 48000 : runtime->rate);
1149     if (chip->spdif_on && viadev->reg_offset == 0x30)
1150         snd_ac97_set_rate(chip->ac97, AC97_SPDIF, runtime->rate);
1151
1152     if (runtime->rate == 48000)
1153         rbits = 0xffffffff;
1154     else
1155         rbits = (0x100000 / 48000) * runtime->rate +
1156             ((0x100000 % 48000) * runtime->rate) / 48000;

```

```

1157     snd_BUG_ON(rbits & ~0xffff);
1158     snd_via82xx_channel_reset(chip, viadev);
1159     snd_via82xx_set_table_ptr(chip, viadev);
1160     outb(chip->playback_volume[viadev->reg_offset / 0x10][0],
1161           VIADEV_REG(viadev, OFS_PLAYBACK_VOLUME_L));
1162     outb(chip->playback_volume[viadev->reg_offset / 0x10][1],
1163           VIADEV_REG(viadev, OFS_PLAYBACK_VOLUME_R));
1164     outl((runtime->format == SNDRV_PCM_FORMAT_S16_LE ? VIA8233_REG_TYPE_16BIT :
1165 0) | /* format */
1166         (runtime->channels > 1 ? VIA8233_REG_TYPE_STEREO : 0) | /* stereo */
1167         rbits | /* rate */
1168         0xff000000, /* STOP index is never reached */
1169         VIADEV_REG(viadev, OFFSET_STOP_IDX));
1170     udelay(20);
1171     snd_via82xx_codec_ready(chip, 0);
1172     return 0;
1173 }
1174
1175 /*
1176  * prepare callback for multi-channel playback on via823x
1177  */
1178 static int snd_via8233_multi_prepare(struct snd_pcm_substream *substream)
1179 {
1180     struct via82xx *chip = snd_pcm_substream_chip(substream);
1181     struct viadev *viadev = substream->runtime->private_data;
1182     struct snd_pcm_runtime *runtime = substream->runtime;
1183     unsigned int slots;
1184     int fmt;
1185
1186     if (via_lock_rate(&chip->rates[0], runtime->rate) < 0)
1187         return -EINVAL;
1188     snd_ac97_set_rate(chip->ac97, AC97_PCM_FRONT_DAC_RATE, runtime->rate);
1189     snd_ac97_set_rate(chip->ac97, AC97_PCM_SURR_DAC_RATE, runtime->rate);
1190     snd_ac97_set_rate(chip->ac97, AC97_PCM_LFE_DAC_RATE, runtime->rate);
1191     snd_ac97_set_rate(chip->ac97, AC97_SPDIF, runtime->rate);
1192     snd_via82xx_channel_reset(chip, viadev);
1193     snd_via82xx_set_table_ptr(chip, viadev);
1194
1195     fmt = (runtime->format == SNDRV_PCM_FORMAT_S16_LE) ?
1196           VIA_REG_MULTPLAY_FMT_16BIT : VIA_REG_MULTPLAY_FMT_8BIT;
1197     fmt |= runtime->channels << 4;
1198     outb(fmt, VIADEV_REG(viadev, OFS_MULTPLAY_FORMAT));
1199
1200     #if 0
1201     if (chip->revision == VIA_REV_8233A)
1202         slots = 0;
1203     else
1204     #endif
1205     {
1206         /* set sample number to slot 3, 4, 7, 8, 6, 9 (for VIA8233/C,8235)
1207         */
1208         /* corresponding to FL, FR, RL, RR, C, LFE ?? */
1209         switch (runtime->channels) {
1210             case 1: slots = (1<<0) | (1<<4); break;
1211             case 2: slots = (1<<0) | (2<<4); break;
1212             case 3: slots = (1<<0) | (2<<4) | (5<<8); break;
1213             case 4: slots = (1<<0) | (2<<4) | (3<<8) | (4<<12); break;
1214             case 5: slots = (1<<0) | (2<<4) | (3<<8) | (4<<12) | (5<<16); break;
1215             case 6: slots = (1<<0) | (2<<4) | (3<<8) | (4<<12) | (5<<16) |
1216                   (6<<20); break;
1217             default: slots = 0; break;
1218         }
1219         /* STOP index is never reached */
1220         outl(0xff000000 | slots, VIADEV_REG(viadev, OFFSET_STOP_IDX));
1221         udelay(20);
1222         snd_via82xx_codec_ready(chip, 0);
1223         return 0;
1224     }

```

```

1225
1226 /*
1227  * prepare callback for capture on via823x
1228  */
1229 static int snd_via8233_capture_prepare(struct snd_pcm_substream *substream)
1230 {
1231     struct via82xx *chip = snd_pcm_substream_chip(substream);
1232     struct viadev *viadev = substream->runtime->private_data;
1233     struct snd_pcm_runtime *runtime = substream->runtime;
1234
1235     if (via_lock_rate(&chip->rates[1], runtime->rate) < 0)
1236         return -EINVAL;
1237     snd_ac97_set_rate(chip->ac97, AC97_PCM_LR_ADC_RATE, runtime->rate);
1238     snd_via82xx_channel_reset(chip, viadev);
1239     snd_via82xx_set_table_ptr(chip, viadev);
1240     outb(VIA_REG_CAPTURE_FIFO_ENABLE, VIADEV_REG(viadev, OFS_CAPTURE_FIFO));
1241     outl((runtime->format == SNDRV_PCM_FORMAT_S16_LE ? VIA8233_REG_TYPE_16BIT :
1242 0) |
1243         (runtime->channels > 1 ? VIA8233_REG_TYPE_STEREO : 0) |
1244         0xff000000, /* STOP index is never reached */
1245         VIADEV_REG(viadev, OFFSET_STOP_IDX));
1246     udelay(20);
1247     snd_via82xx_codec_ready(chip, 0);
1248     return 0;
1249 }
1250
1251 /*
1252  * pcm hardware definition, identical for both playback and capture
1253  */
1254 static struct snd_pcm_hw snd_via82xx_hw =
1255 {
1256     .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
1257             SNDRV_PCM_INFO_BLOCK_TRANSFER |
1258             SNDRV_PCM_INFO_MMAP_VALID |
1259             /* SNDRV_PCM_INFO_RESUME | */
1260             SNDRV_PCM_INFO_PAUSE),
1261     .formats = SNDRV_PCM_FMTBIT_U8 | SNDRV_PCM_FMTBIT_S16_LE,
1262     .rates = SNDRV_PCM_RATE_48000,
1263     .rate_min = 48000,
1264     .rate_max = 48000,
1265     .channels_min = 1,
1266     .channels_max = 2,
1267     .buffer_bytes_max = VIA_MAX_BUFSIZE,
1268     .period_bytes_min = 32,
1269     .period_bytes_max = VIA_MAX_BUFSIZE / 2,
1270     .periods_min = 2,
1271     .periods_max = VIA_TABLE_SIZE / 2,
1272     .fifo_size = 0,
1273 };
1274
1275 /*
1276  * open callback skeleton
1277  */
1278 static int snd_via82xx_pcm_open(struct via82xx *chip, struct viadev *viadev,
1279                                struct snd_pcm_substream *substream)
1280 {
1281     struct snd_pcm_runtime *runtime = substream->runtime;
1282     int err;
1283     struct via_rate_lock *ratep;
1284     bool use_src = false;
1285
1286     runtime->hw = snd_via82xx_hw;
1287
1288     /* set the hw rate condition */
1289     ratep = &chip->rates[viadev->direction];
1290     spin_lock_irq(&ratep->lock);
1291

```

```

1293     ratep->used++;
1294     if (chip->spdif_on && viadev->reg_offset == 0x30) {
1295         /* DXS#3 and spdif is on */
1296         runtime->hw.rates = chip->ac97->rates[AC97_RATES_SPDIF];
1297         snd_pcm_limit_hw_rates(runtime);
1298     } else if (chip->dxs_fixed && viadev->reg_offset < 0x40) {
1299         /* fixed DXS playback rate */
1300         runtime->hw.rates = SNDRV_PCM_RATE_48000;
1301         runtime->hw.rate_min = runtime->hw.rate_max = 48000;
1302     } else if (chip->dxs_src && viadev->reg_offset < 0x40) {
1303         /* use full SRC capabilities of DXS */
1304         runtime->hw.rates = (SNDRV_PCM_RATE_CONTINUOUS |
1305                             SNDRV_PCM_RATE_8000_48000);
1306         runtime->hw.rate_min = 8000;
1307         runtime->hw.rate_max = 48000;
1308         use_src = true;
1309     } else if (!ratep->rate) {
1310         int idx = viadev->direction ? AC97_RATES_ADC : AC97_RATES_FRONT_DAC;
1311         runtime->hw.rates = chip->ac97->rates[idx];
1312         snd_pcm_limit_hw_rates(runtime);
1313     } else {
1314         /* a fixed rate */
1315         runtime->hw.rates = SNDRV_PCM_RATE_KNOT;
1316         runtime->hw.rate_max = runtime->hw.rate_min = ratep->rate;
1317     }
1318     spin_unlock_irq(&ratep->lock);
1319
1320     /* we may remove following constraint when we modify table entries
1321        in interrupt */
1322     if ((err = snd_pcm_hw_constraint_integer(runtime,
1323 SNDRV_PCM_HW_PARAM_PERIODS)) < 0)
1324         return err;
1325
1326     if (use_src) {
1327         err = snd_pcm_hw_rule_noresample(runtime, 48000);
1328         if (err < 0)
1329             return err;
1330     }
1331
1332     runtime->private_data = viadev;
1333     viadev->substream = substream;
1334
1335     return 0;
1336 }
1337
1338 /*
1339  * open callback for playback on via686
1340  */
1341 static int snd_via686_playback_open(struct snd_pcm_substream *substream)
1342 {
1343     struct via82xx *chip = snd_pcm_substream_chip(substream);
1344     struct viadev *viadev = &chip->devs[chip->playback_devno + substream-
1345 >number];
1346     int err;
1347
1348     if ((err = snd_via82xx_pcm_open(chip, viadev, substream)) < 0)
1349         return err;
1350     return 0;
1351 }
1352
1353
1354
1355
1356
1357
1358
1359
1360

```

```

1361  /*
1362  * open callback for playback on via823x DXS
1363  */
1364  static int snd_via8233_playback_open(struct snd_pcm_substream *substream)
1365  {
1366      struct via82xx *chip = snd_pcm_substream_chip(substream);
1367      struct viadev *viadev;
1368      unsigned int stream;
1369      int err;
1370
1371      viadev = &chip->devs[chip->playback_devno + substream->number];
1372      if ((err = snd_via82xx_pcm_open(chip, viadev, substream)) < 0)
1373          return err;
1374      stream = viadev->reg_offset / 0x10;
1375      if (chip->dxs_controls[stream]) {
1376          chip->playback_volume[stream][0] =
1377              VIA_DXS_MAX_VOLUME - (dxs_init_volume & 31);
1378          chip->playback_volume[stream][1] =
1379              VIA_DXS_MAX_VOLUME - (dxs_init_volume & 31);
1380          chip->dxs_controls[stream]->vd[0].access &=
1381              ~SNDRV_CTL_ELEM_ACCESS_INACTIVE;
1382          snd_ctl_notify(chip->card, SNDRV_CTL_EVENT_MASK_VALUE |
1383                          SNDRV_CTL_EVENT_MASK_INFO,
1384                          &chip->dxs_controls[stream]->id);
1385      }
1386      return 0;
1387  }
1388
1389  /*
1390  * open callback for playback on via823x multi-channel
1391  */
1392  static int snd_via8233_multi_open(struct snd_pcm_substream *substream)
1393  {
1394      struct via82xx *chip = snd_pcm_substream_chip(substream);
1395      struct viadev *viadev = &chip->devs[chip->multi_devno];
1396      int err;
1397      /* channels constraint for VIA8233A
1398       * 3 and 5 channels are not supported
1399       */
1400      static unsigned int channels[] = {
1401          1, 2, 4, 6
1402      };
1403      static struct snd_pcm_hw_constraint_list hw_constraints_channels = {
1404          .count = ARRAY_SIZE(channels),
1405          .list = channels,
1406          .mask = 0,
1407      };
1408
1409      if ((err = snd_via82xx_pcm_open(chip, viadev, substream)) < 0)
1410          return err;
1411      substream->runtime->hw.channels_max = 6;
1412      if (chip->revision == VIA_REV_8233A)
1413          snd_pcm_hw_constraint_list(substream->runtime, 0,
1414                                     SNDRV_PCM_HW_PARAM_CHANNELS,
1415                                     &hw_constraints_channels);
1416      return 0;
1417  }
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428

```

```

1429  /*
1430  * open callback for capture on via686 and via823x
1431  */
1432  static int snd_via82xx_capture_open(struct snd_pcm_substream *substream)
1433  {
1434      struct via82xx *chip = snd_pcm_substream_chip(substream);
1435      struct viadev *viadev = &chip->devs[chip->capture_devno + substream->pcm-
1436  >device];
1437
1438      return snd_via82xx_pcm_open(chip, viadev, substream);
1439  }
1440
1441  /*
1442  * close callback
1443  */
1444  static int snd_via82xx_pcm_close(struct snd_pcm_substream *substream)
1445  {
1446      struct via82xx *chip = snd_pcm_substream_chip(substream);
1447      struct viadev *viadev = substream->runtime->private_data;
1448      struct via_rate_lock *ratep;
1449
1450      /* release the rate lock */
1451      ratep = &chip->rates[viadev->direction];
1452      spin_lock_irq(&ratep->lock);
1453      ratep->used--;
1454      if (! ratep->used)
1455          ratep->rate = 0;
1456      spin_unlock_irq(&ratep->lock);
1457      if (! ratep->rate) {
1458          if (! viadev->direction) {
1459              snd_ac97_update_power(chip->ac97,
1460                                  AC97_PCM_FRONT_DAC_RATE, 0);
1461              snd_ac97_update_power(chip->ac97,
1462                                  AC97_PCM_SURR_DAC_RATE, 0);
1463              snd_ac97_update_power(chip->ac97,
1464                                  AC97_PCM_LFE_DAC_RATE, 0);
1465          } else
1466              snd_ac97_update_power(chip->ac97,
1467                                  AC97_PCM_LR_ADC_RATE, 0);
1468      }
1469      viadev->substream = NULL;
1470      return 0;
1471  }
1472
1473  static int snd_via8233_playback_close(struct snd_pcm_substream *substream)
1474  {
1475      struct via82xx *chip = snd_pcm_substream_chip(substream);
1476      struct viadev *viadev = substream->runtime->private_data;
1477      unsigned int stream;
1478
1479      stream = viadev->reg_offset / 0x10;
1480      if (chip->dxs_controls[stream]) {
1481          chip->dxs_controls[stream]->vd[0].access |=
1482              SNDRV_CTL_ELEM_ACCESS_INACTIVE;
1483          snd_ctl_notify(chip->card, SNDRV_CTL_EVENT_MASK_INFO,
1484                        &chip->dxs_controls[stream]->id);
1485      }
1486      return snd_via82xx_pcm_close(substream);
1487  }
1488
1489
1490
1491
1492
1493
1494
1495
1496

```



```

1497 /* via686 playback callbacks */
1498 static struct snd_pcm_ops snd_via686_playback_ops = {
1499     .open =      snd_via686_playback_open,
1500     .close = snd_via82xx_pcm_close,
1501     .ioctl = snd_pcm_lib_ioctl,
1502     .hw_params =      snd_via82xx_hw_params,
1503     .hw_free =      snd_via82xx_hw_free,
1504     .prepare =      snd_via686_playback_prepare,
1505     .trigger =      snd_via82xx_pcm_trigger,
1506     .pointer =      snd_via686_pcm_pointer,
1507     .page =      snd_pcm_sgbuf_ops_page,
1508 };
1509
1510 /* via686 capture callbacks */
1511 static struct snd_pcm_ops snd_via686_capture_ops = {
1512     .open =      snd_via82xx_capture_open,
1513     .close = snd_via82xx_pcm_close,
1514     .ioctl = snd_pcm_lib_ioctl,
1515     .hw_params =      snd_via82xx_hw_params,
1516     .hw_free =      snd_via82xx_hw_free,
1517     .prepare =      snd_via686_capture_prepare,
1518     .trigger =      snd_via82xx_pcm_trigger,
1519     .pointer =      snd_via686_pcm_pointer,
1520     .page =      snd_pcm_sgbuf_ops_page,
1521 };
1522
1523 /* via823x DSX playback callbacks */
1524 static struct snd_pcm_ops snd_via8233_playback_ops = {
1525     .open =      snd_via8233_playback_open,
1526     .close = snd_via8233_playback_close,
1527     .ioctl = snd_pcm_lib_ioctl,
1528     .hw_params =      snd_via82xx_hw_params,
1529     .hw_free =      snd_via82xx_hw_free,
1530     .prepare =      snd_via8233_playback_prepare,
1531     .trigger =      snd_via82xx_pcm_trigger,
1532     .pointer =      snd_via8233_pcm_pointer,
1533     .page =      snd_pcm_sgbuf_ops_page,
1534 };
1535
1536 /* via823x multi-channel playback callbacks */
1537 static struct snd_pcm_ops snd_via8233_multi_ops = {
1538     .open =      snd_via8233_multi_open,
1539     .close = snd_via82xx_pcm_close,
1540     .ioctl = snd_pcm_lib_ioctl,
1541     .hw_params =      snd_via82xx_hw_params,
1542     .hw_free =      snd_via82xx_hw_free,
1543     .prepare =      snd_via8233_multi_prepare,
1544     .trigger =      snd_via82xx_pcm_trigger,
1545     .pointer =      snd_via8233_pcm_pointer,
1546     .page =      snd_pcm_sgbuf_ops_page,
1547 };
1548
1549 /* via823x capture callbacks */
1550 static struct snd_pcm_ops snd_via8233_capture_ops = {
1551     .open =      snd_via82xx_capture_open,
1552     .close = snd_via82xx_pcm_close,
1553     .ioctl = snd_pcm_lib_ioctl,
1554     .hw_params =      snd_via82xx_hw_params,
1555     .hw_free =      snd_via82xx_hw_free,
1556     .prepare =      snd_via8233_capture_prepare,
1557     .trigger =      snd_via82xx_pcm_trigger,
1558     .pointer =      snd_via8233_pcm_pointer,
1559     .page =      snd_pcm_sgbuf_ops_page,
1560 };
1561
1562
1563
1564

```

```

1565 static void init_viadev(struct via82xx *chip, int idx, unsigned int reg_offset,
1566                        int shadow_pos, int direction)
1567 {
1568     chip->devs[idx].reg_offset = reg_offset;
1569     chip->devs[idx].shadow_shift = shadow_pos * 4;
1570     chip->devs[idx].direction = direction;
1571     chip->devs[idx].port = chip->port + reg_offset;
1572 }
1573
1574 /*
1575  * create pcm instances for VIA8233, 8233C and 8235 (not 8233A)
1576  */
1577 static int __devinit snd_via8233_pcm_new(struct via82xx *chip)
1578 {
1579     struct snd_pcm *pcm;
1580     int i, err;
1581
1582     chip->playback_devno = 0; /* x 4 */
1583     chip->multi_devno = 4; /* x 1 */
1584     chip->capture_devno = 5; /* x 2 */
1585     chip->num_devs = 7;
1586     chip->intr_mask = 0x33033333; /* FLAG/EOL for rec0-1, mc, sdx0-3 */
1587
1588     /* PCM #0: 4 DSX playbacks and 1 capture */
1589     err = snd_pcm_new(chip->card, chip->card->shortname, 0, 4, 1, &pcm);
1590     if (err < 0)
1591         return err;
1592     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_via8233_playback_ops);
1593     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, &snd_via8233_capture_ops);
1594     pcm->private_data = chip;
1595     strcpy(pcm->name, chip->card->shortname);
1596     chip->pcms[0] = pcm;
1597     /* set up playbacks */
1598     for (i = 0; i < 4; i++)
1599         init_viadev(chip, i, 0x10 * i, i, 0);
1600     /* capture */
1601     init_viadev(chip, chip->capture_devno, VIA_REG_CAPTURE_8233_STATUS, 6, 1);
1602
1603     snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV_SG,
1604                                           snd_dma_pci_data(chip->pci),
1605                                           64*1024, VIA_MAX_BUFSIZE);
1606
1607     /* PCM #1: multi-channel playback and 2nd capture */
1608     err = snd_pcm_new(chip->card, chip->card->shortname, 1, 1, 1, &pcm);
1609     if (err < 0)
1610         return err;
1611     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_via8233_multi_ops);
1612     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, &snd_via8233_capture_ops);
1613     pcm->private_data = chip;
1614     strcpy(pcm->name, chip->card->shortname);
1615     chip->pcms[1] = pcm;
1616     /* set up playback */
1617     init_viadev(chip, chip->multi_devno, VIA_REG_MULTPLAY_STATUS, 4, 0);
1618     /* set up capture */
1619     init_viadev(chip, chip->capture_devno + 1, VIA_REG_CAPTURE_8233_STATUS +
1620 0x10, 7, 1);
1621
1622     snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV_SG,
1623                                           snd_dma_pci_data(chip->pci),
1624                                           64*1024, VIA_MAX_BUFSIZE);
1625     return 0;
1626 }
1627
1628
1629
1630
1631
1632

```

```

1633 /*
1634  * create pcm instances for VIA8233A
1635  */
1636 static int __devinit snd_via8233a_pcm_new(struct via82xx *chip)
1637 {
1638     struct snd_pcm *pcm;
1639     int err;
1640
1641     chip->multi_devno = 0;
1642     chip->playback_devno = 1;
1643     chip->capture_devno = 2;
1644     chip->num_devs = 3;
1645     chip->intr_mask = 0x03033000; /* FLAG|EOL for rec0, mc, sdx3 */
1646
1647     /* PCM #0: multi-channel playback and capture */
1648     err = snd_pcm_new(chip->card, chip->card->shortname, 0, 1, 1, &pcm);
1649     if (err < 0)
1650         return err;
1651     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_via8233_multi_ops);
1652     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, &snd_via8233_capture_ops);
1653     pcm->private_data = chip;
1654     strcpy(pcm->name, chip->card->shortname);
1655     chip->pcms[0] = pcm;
1656     /* set up playback */
1657     init_viadev(chip, chip->multi_devno, VIA_REG_MULTPLAY_STATUS, 4, 0);
1658     /* capture */
1659     init_viadev(chip, chip->capture_devno, VIA_REG_CAPTURE_8233_STATUS, 6, 1);
1660
1661     snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV_SG,
1662                                           snd_dma_pci_data(chip->pci),
1663                                           64*1024, VIA_MAX_BUFSIZE);
1664
1665     /* SPDIF supported? */
1666     if (! ac97_can_spdif(chip->ac97))
1667         return 0;
1668
1669     /* PCM #1: DXS3 playback (for spdif) */
1670     err = snd_pcm_new(chip->card, chip->card->shortname, 1, 1, 0, &pcm);
1671     if (err < 0)
1672         return err;
1673     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_via8233_playback_ops);
1674     pcm->private_data = chip;
1675     strcpy(pcm->name, chip->card->shortname);
1676     chip->pcms[1] = pcm;
1677     /* set up playback */
1678     init_viadev(chip, chip->playback_devno, 0x30, 3, 0);
1679
1680     snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV_SG,
1681                                           snd_dma_pci_data(chip->pci),
1682                                           64*1024, VIA_MAX_BUFSIZE);
1683     return 0;
1684 }
1685
1686 /*
1687  * create a pcm instance for via686a/b
1688  */
1689 static int __devinit snd_via686_pcm_new(struct via82xx *chip)
1690 {
1691     struct snd_pcm *pcm;
1692     int err;
1693
1694     chip->playback_devno = 0;
1695     chip->capture_devno = 1;
1696     chip->num_devs = 2;
1697     chip->intr_mask = 0x77; /* FLAG | EOL for PB, CP, FM */
1698
1699
1700

```

```

1701     err = snd_pcm_new(chip->card, chip->card->shortname, 0, 1, 1, &pcm);
1702     if (err < 0)
1703         return err;
1704     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_via686_playback_ops);
1705     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, &snd_via686_capture_ops);
1706     pcm->private_data = chip;
1707     strcpy(pcm->name, chip->card->shortname);
1708     chip->pcms[0] = pcm;
1709     init_viadev(chip, 0, VIA_REG_PLAYBACK_STATUS, 0, 0);
1710     init_viadev(chip, 1, VIA_REG_CAPTURE_STATUS, 0, 1);
1711
1712     snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV_SG,
1713                                           snd_dma_pci_data(chip->pci),
1714                                           64*1024, VIA_MAX_BUFSIZE);
1715     return 0;
1716 }
1717
1718 /*
1719  * Mixer part
1720  */
1721
1722 static int snd_via8233_capture_source_info(struct snd_kcontrol *kcontrol,
1723                                           struct snd_ctl_elem_info *uinfo)
1724 {
1725     /* formerly they were "Line" and "Mic", but it looks like that they
1726      * have nothing to do with the actual physical connections...
1727      */
1728     static char *texts[2] = {
1729         "Input1", "Input2"
1730     };
1731     uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;
1732     uinfo->count = 1;
1733     uinfo->value.enumerated.items = 2;
1734     if (uinfo->value.enumerated.item >= 2)
1735         uinfo->value.enumerated.item = 1;
1736     strcpy(uinfo->value.enumerated.name, texts[uinfo->value.enumerated.item]);
1737     return 0;
1738 }
1739
1740 static int snd_via8233_capture_source_get(struct snd_kcontrol *kcontrol,
1741                                           struct snd_ctl_elem_value *ucontrol)
1742 {
1743     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1744     unsigned long port = chip->port + (kcontrol->id.index ?
1745 (VIA_REG_CAPTURE_CHANNEL + 0x10) : VIA_REG_CAPTURE_CHANNEL);
1746     ucontrol->value.enumerated.item[0] = inb(port) & VIA_REG_CAPTURE_CHANNEL_MIC
1747 ? 1 : 0;
1748     return 0;
1749 }
1750
1751 static int snd_via8233_capture_source_put(struct snd_kcontrol *kcontrol,
1752                                           struct snd_ctl_elem_value *ucontrol)
1753 {
1754     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1755     unsigned long port = chip->port + (kcontrol->id.index ?
1756 (VIA_REG_CAPTURE_CHANNEL + 0x10) : VIA_REG_CAPTURE_CHANNEL);
1757     u8 val, oval;
1758
1759     spin_lock_irq(&chip->reg_lock);
1760     oval = inb(port);
1761     val = oval & ~VIA_REG_CAPTURE_CHANNEL_MIC;
1762     if (ucontrol->value.enumerated.item[0])
1763         val |= VIA_REG_CAPTURE_CHANNEL_MIC;
1764     if (val != oval)
1765         outb(val, port);
1766     spin_unlock_irq(&chip->reg_lock);
1767     return val != oval;
1768 }

```

```

1769 static struct snd_kcontrol_new snd_via8233_capture_source __devinitdata = {
1770     .name = "Input Source Select",
1771     .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
1772     .info = snd_via8233_capture_source_info,
1773     .get = snd_via8233_capture_source_get,
1774     .put = snd_via8233_capture_source_put,
1775 };
1776
1777 #define snd_via8233_dxs3_spdif_info      snd_ctl_boolean_mono_info
1778
1779 static int snd_via8233_dxs3_spdif_get(struct snd_kcontrol *kcontrol,
1780                                     struct snd_ctl_elem_value *ucontrol)
1781 {
1782     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1783     u8 val;
1784
1785     pci_read_config_byte(chip->pci, VIA8233_SPDIF_CTRL, &val);
1786     ucontrol->value.integer.value[0] = (val & VIA8233_SPDIF_DX3) ? 1 : 0;
1787     return 0;
1788 }
1789
1790 static int snd_via8233_dxs3_spdif_put(struct snd_kcontrol *kcontrol,
1791                                     struct snd_ctl_elem_value *ucontrol)
1792 {
1793     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1794     u8 val, oval;
1795     pci_read_config_byte(chip->pci, VIA8233_SPDIF_CTRL, &oval);
1796     val = oval & ~VIA8233_SPDIF_DX3;
1797     if (ucontrol->value.integer.value[0])
1798         val |= VIA8233_SPDIF_DX3;
1799     /* save the spdif flag for rate filtering */
1800     chip->spdif_on = ucontrol->value.integer.value[0] ? 1 : 0;
1801     if (val != oval) {
1802         pci_write_config_byte(chip->pci, VIA8233_SPDIF_CTRL, val);
1803         return 1;
1804     }
1805     return 0;
1806 }
1807
1808 static struct snd_kcontrol_new snd_via8233_dxs3_spdif_control __devinitdata = {
1809     .name = SNDRV_CTL_NAME_IEC958("Output ", NONE, SWITCH),
1810     .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
1811     .info = snd_via8233_dxs3_spdif_info,
1812     .get = snd_via8233_dxs3_spdif_get,
1813     .put = snd_via8233_dxs3_spdif_put,
1814 };
1815
1816 static int snd_via8233_dxs_volume_info(struct snd_kcontrol *kcontrol,
1817                                       struct snd_ctl_elem_info *uinfo)
1818 {
1819     uinfo->type = SNDRV_CTL_ELEM_TYPE_INTEGER;
1820     uinfo->count = 2;
1821     uinfo->value.integer.min = 0;
1822     uinfo->value.integer.max = VIA_DXS_MAX_VOLUME;
1823     return 0;
1824 }
1825
1826 static int snd_via8233_dxs_volume_get(struct snd_kcontrol *kcontrol,
1827                                       struct snd_ctl_elem_value *ucontrol)
1828 {
1829     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1830     unsigned int idx = kcontrol->id.subdevice;
1831     ucontrol->value.integer.value[0] = VIA_DXS_MAX_VOLUME - chip-
1832 >playback_volume[idx][0];
1833     ucontrol->value.integer.value[1] = VIA_DXS_MAX_VOLUME - chip-
1834 >playback_volume[idx][1];
1835     return 0;
1836 }

```

```

1837
1838 static int snd_via8233_pcmdxs_volume_get(struct snd_kcontrol *kcontrol,
1839                                         struct snd_ctl_elem_value *ucontrol)
1840 {
1841     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1842     ucontrol->value.integer.value[0] = VIA_DXS_MAX_VOLUME - chip-
1843 >playback_volume_c[0];
1844     ucontrol->value.integer.value[1] = VIA_DXS_MAX_VOLUME - chip-
1845 >playback_volume_c[1];
1846     return 0;
1847 }
1848
1849 static int snd_via8233_dxs_volume_put(struct snd_kcontrol *kcontrol,
1850                                       struct snd_ctl_elem_value *ucontrol)
1851 {
1852     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1853     unsigned int idx = kcontrol->id.subdevice;
1854     unsigned long port = chip->port + 0x10 * idx;
1855     unsigned char val;
1856     int i, change = 0;
1857
1858     for (i = 0; i < 2; i++) {
1859         val = ucontrol->value.integer.value[i];
1860         if (val > VIA_DXS_MAX_VOLUME)
1861             val = VIA_DXS_MAX_VOLUME;
1862         val = VIA_DXS_MAX_VOLUME - val;
1863         change |= val != chip->playback_volume[idx][i];
1864         if (change) {
1865             chip->playback_volume[idx][i] = val;
1866             outb(val, port + VIA_REG_OFS_PLAYBACK_VOLUME_L + i);
1867         }
1868     }
1869     return change;
1870 }
1871
1872 static int snd_via8233_pcmdxs_volume_put(struct snd_kcontrol *kcontrol,
1873                                           struct snd_ctl_elem_value *ucontrol)
1874 {
1875     struct via82xx *chip = snd_kcontrol_chip(kcontrol);
1876     unsigned int idx;
1877     unsigned char val;
1878     int i, change = 0;
1879
1880     for (i = 0; i < 2; i++) {
1881         val = ucontrol->value.integer.value[i];
1882         if (val > VIA_DXS_MAX_VOLUME)
1883             val = VIA_DXS_MAX_VOLUME;
1884         val = VIA_DXS_MAX_VOLUME - val;
1885         if (val != chip->playback_volume_c[i]) {
1886             change = 1;
1887             chip->playback_volume_c[i] = val;
1888             for (idx = 0; idx < 4; idx++) {
1889                 unsigned long port = chip->port + 0x10 * idx;
1890                 chip->playback_volume[idx][i] = val;
1891                 outb(val, port + VIA_REG_OFS_PLAYBACK_VOLUME_L +
1892 i);
1893             }
1894         }
1895     }
1896     return change;
1897 }
1898
1899
1900
1901
1902
1903
1904

```

```

1905 static const DECLARE_TLV_DB_SCALE(db_scale_dxs, -4650, 150, 1);
1906
1907 static struct snd_kcontrol_new snd_via8233_pcmdxs_volume_control __devinitdata = {
1908     .name = "PCM Playback Volume",
1909     .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
1910     .access = (SNDRV_CTL_ELEM_ACCESS_READWRITE |
1911                SNDRV_CTL_ELEM_ACCESS_TLV_READ),
1912     .info = snd_via8233_dxs_volume_info,
1913     .get = snd_via8233_pcmdxs_volume_get,
1914     .put = snd_via8233_pcmdxs_volume_put,
1915     .tlv = { .p = db_scale_dxs }
1916 };
1917
1918 static struct snd_kcontrol_new snd_via8233_dxs_volume_control __devinitdata = {
1919     .iface = SNDRV_CTL_ELEM_IFACE_PCM,
1920     .device = 0,
1921     /* .subdevice set later */
1922     .name = "PCM Playback Volume",
1923     .access = SNDRV_CTL_ELEM_ACCESS_READWRITE |
1924                SNDRV_CTL_ELEM_ACCESS_TLV_READ |
1925                SNDRV_CTL_ELEM_ACCESS_INACTIVE,
1926     .info = snd_via8233_dxs_volume_info,
1927     .get = snd_via8233_dxs_volume_get,
1928     .put = snd_via8233_dxs_volume_put,
1929     .tlv = { .p = db_scale_dxs }
1930 };
1931
1932 /*
1933  */
1934
1935 static void snd_via82xx_mixer_free_ac97_bus(struct snd_ac97_bus *bus)
1936 {
1937     struct via82xx *chip = bus->private_data;
1938     chip->ac97_bus = NULL;
1939 }
1940
1941 static void snd_via82xx_mixer_free_ac97(struct snd_ac97 *ac97)
1942 {
1943     struct via82xx *chip = ac97->private_data;
1944     chip->ac97 = NULL;
1945 }
1946
1947 static struct ac97_quirk ac97_quirks[] = {
1948     {
1949         .subvendor = 0x1106,
1950         .subdevice = 0x4161,
1951         .codec_id = 0x56494161, /* VT1612A */
1952         .name = "Soltek SL-75DRV5",
1953         .type = AC97_TUNE_NONE
1954     },
1955     {
1956         /* FIXME: which codec? */
1957         .subvendor = 0x1106,
1958         .subdevice = 0x4161,
1959         .name = "ASRock K7VT2",
1960         .type = AC97_TUNE_HP_ONLY
1961     },
1962     {
1963         .subvendor = 0x110a,
1964         .subdevice = 0x0079,
1965         .name = "Fujitsu Siemens D1289",
1966         .type = AC97_TUNE_HP_ONLY
1967     },
1968     {
1969         .subvendor = 0x1019,
1970         .subdevice = 0x0a81,
1971         .name = "ECS K7VTA3",
1972         .type = AC97_TUNE_HP_ONLY
1973     },
1974 }

```



```

1973     {
1974         .subvendor = 0x1019,
1975         .subdevice = 0x0a85,
1976         .name = "ECS L7VMM2",
1977         .type = AC97_TUNE_HP_ONLY
1978     },
1979     {
1980         .subvendor = 0x1019,
1981         .subdevice = 0x1841,
1982         .name = "ECS K7VTA3",
1983         .type = AC97_TUNE_HP_ONLY
1984     },
1985     {
1986         .subvendor = 0x1849,
1987         .subdevice = 0x3059,
1988         .name = "ASRock K7VM2",
1989         .type = AC97_TUNE_HP_ONLY /* VT1616 */
1990     },
1991     {
1992         .subvendor = 0x14cd,
1993         .subdevice = 0x7002,
1994         .name = "Unknown",
1995         .type = AC97_TUNE_ALC_JACK
1996     },
1997     {
1998         .subvendor = 0x1071,
1999         .subdevice = 0x8590,
2000         .name = "Mitac Mobo",
2001         .type = AC97_TUNE_ALC_JACK
2002     },
2003     {
2004         .subvendor = 0x161f,
2005         .subdevice = 0x202b,
2006         .name = "Arima Notebook",
2007         .type = AC97_TUNE_HP_ONLY,
2008     },
2009     {
2010         .subvendor = 0x161f,
2011         .subdevice = 0x2032,
2012         .name = "Targa Traveller 811",
2013         .type = AC97_TUNE_HP_ONLY,
2014     },
2015     {
2016         .subvendor = 0x161f,
2017         .subdevice = 0x2032,
2018         .name = "m680x",
2019         .type = AC97_TUNE_HP_ONLY, /* http://launchpad.net/bugs/38546 */
2020     },
2021     {
2022         .subvendor = 0x1297,
2023         .subdevice = 0xa232,
2024         .name = "Shuttle AK32VN",
2025         .type = AC97_TUNE_HP_ONLY
2026     },
2027     { } /* terminator */
2028 };
2029
2030 static int __devinit snd_via82xx_mixer_new(struct via82xx *chip, const char
2031 *quirk_override)
2032 {
2033     struct snd_ac97_template ac97;
2034     int err;
2035     static struct snd_ac97_bus_ops ops = {
2036         .write = snd_via82xx_codec_write,
2037         .read = snd_via82xx_codec_read,
2038         .wait = snd_via82xx_codec_wait,
2039     };
2040

```

```

2041     if ((err = snd_ac97_bus(chip->card, 0, &ops, chip, &chip->ac97_bus)) < 0)
2042         return err;
2043     chip->ac97_bus->private_free = snd_via82xx_mixer_free_ac97_bus;
2044     chip->ac97_bus->clock = chip->ac97_clock;
2045
2046     memset(&ac97, 0, sizeof(ac97));
2047     ac97.private_data = chip;
2048     ac97.private_free = snd_via82xx_mixer_free_ac97;
2049     ac97.pci = chip->pci;
2050     ac97.scaps = AC97_SCAP_SKIP_MODEM | AC97_SCAP_POWER_SAVE;
2051     if ((err = snd_ac97_mixer(chip->ac97_bus, &ac97, &chip->ac97)) < 0)
2052         return err;
2053
2054     snd_ac97_tune_hardware(chip->ac97, ac97_quirks, quirk_override);
2055
2056     if (chip->chip_type != TYPE_VIA686) {
2057         /* use slot 10/11 */
2058         snd_ac97_update_bits(chip->ac97, AC97_EXTENDED_STATUS, 0x03 << 4,
2059 0x03 << 4);
2060     }
2061
2062     return 0;
2063 }
2064
2065 #ifdef SUPPORT_JOYSTICK
2066 #define JOYSTICK_ADDR    0x200
2067 static int __devinit snd_via686_create_gameport(struct via82xx *chip, unsigned char
2068 *legacy)
2069 {
2070     struct gameport *gp;
2071     struct resource *r;
2072
2073     if (!joystick)
2074         return -ENODEV;
2075
2076     r = request_region(JOYSTICK_ADDR, 8, "VIA686 gameport");
2077     if (!r) {
2078         printk(KERN_WARNING "via82xx: cannot reserve joystick port 0x%x\n",
2079 JOYSTICK_ADDR);
2080         return -EBUSY;
2081     }
2082
2083     chip->gameport = gp = gameport_allocate_port();
2084     if (!gp) {
2085         printk(KERN_ERR "via82xx: cannot allocate memory for gameport\n");
2086         release_and_free_resource(r);
2087         return -ENOMEM;
2088     }
2089
2090     gameport_set_name(gp, "VIA686 Gameport");
2091     gameport_set_phys(gp, "pci%s/gameport0", pci_name(chip->pci));
2092     gameport_set_dev_parent(gp, &chip->pci->dev);
2093     gp->io = JOYSTICK_ADDR;
2094     gameport_set_port_data(gp, r);
2095
2096     /* Enable legacy joystick port */
2097     *legacy |= VIA_FUNC_ENABLE_GAME;
2098     pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE, *legacy);
2099
2100     gameport_register_port(chip->gameport);
2101
2102     return 0;
2103 }
2104
2105
2106
2107
2108

```

```

2109 static void snd_via686_free_gameport(struct via82xx *chip)
2110 {
2111     if (chip->gameport) {
2112         struct resource *r = gameport_get_port_data(chip->gameport);
2113
2114         gameport_unregister_port(chip->gameport);
2115         chip->gameport = NULL;
2116         release_and_free_resource(r);
2117     }
2118 }
2119 #else
2120 static inline int snd_via686_create_gameport(struct via82xx *chip, unsigned char
2121 *legacy)
2122 {
2123     return -ENOSYS;
2124 }
2125 static inline void snd_via686_free_gameport(struct via82xx *chip) { }
2126 #endif
2127
2128 /*
2129  *
2130  */
2131
2132 static int __devinit snd_via8233_init_misc(struct via82xx *chip)
2133 {
2134     int i, err, caps;
2135     unsigned char val;
2136
2137     caps = chip->chip_type == TYPE_VIA8233A ? 1 : 2;
2138     for (i = 0; i < caps; i++) {
2139         snd_via8233_capture_source.index = i;
2140         err = snd_ctl_add(chip->card,
2141 snd_ctl_new1(&snd_via8233_capture_source, chip));
2142         if (err < 0)
2143             return err;
2144     }
2145     if (ac97_can_spdif(chip->ac97)) {
2146         err = snd_ctl_add(chip->card,
2147 snd_ctl_new1(&snd_via8233_dxs3_spdif_control, chip));
2148         if (err < 0)
2149             return err;
2150     }
2151     if (chip->chip_type != TYPE_VIA8233A) {
2152         /* when no h/w PCM volume control is found, use DXS volume control
2153          * as the PCM vol control
2154          */
2155         struct snd_ctl_elem_id sid;
2156         memset(&sid, 0, sizeof(sid));
2157         strcpy(sid.name, "PCM Playback Volume");
2158         sid.iface = SNDRV_CTL_ELEM_IFACE_MIXER;
2159         if (!snd_ctl_find_id(chip->card, &sid)) {
2160             snd_printd(KERN_INFO "Using DXS as PCM Playback\n");
2161             err = snd_ctl_add(chip->card,
2162 snd_ctl_new1(&snd_via8233_pcmdxs_volume_control, chip));
2163             if (err < 0)
2164                 return err;
2165         }
2166     }
2167     else /* Using DXS when PCM emulation is enabled is really weird */
2168     {
2169         for (i = 0; i < 4; ++i) {
2170             struct snd_kcontrol *kctl;
2171
2172             kctl = snd_ctl_new1(
2173                 &snd_via8233_dxs_volume_control, chip);
2174             if (!kctl)
2175                 return -ENOMEM;
2176             kctl->id.subdevice = i;

```

```

2177         err = snd_ctl_add(chip->card, kctl);
2178         if (err < 0)
2179             return err;
2180         chip->dxs_controls[i] = kctl;
2181     }
2182 }
2183 }
2184 /* select spdif data slot 10/11 */
2185 pci_read_config_byte(chip->pci, VIA8233_SPDIF_CTRL, &val);
2186 val = (val & ~VIA8233_SPDIF_SLOT_MASK) | VIA8233_SPDIF_SLOT_1011;
2187 val &= ~VIA8233_SPDIF_DX3; /* SPDIF off as default */
2188 pci_write_config_byte(chip->pci, VIA8233_SPDIF_CTRL, val);
2189
2190 return 0;
2191 }
2192
2193 static int __devinit snd_via686_init_misc(struct via82xx *chip)
2194 {
2195     unsigned char legacy, legacy_cfg;
2196     int rev_h = 0;
2197
2198     legacy = chip->old_legacy;
2199     legacy_cfg = chip->old_legacy_cfg;
2200     legacy |= VIA_FUNC_MIDI_IRQMASK; /* FIXME: correct? (disable MIDI) */
2201     legacy &= ~VIA_FUNC_ENABLE_GAME; /* disable joystick */
2202     if (chip->revision >= VIA_REV_686_H) {
2203         rev_h = 1;
2204         if (mpu_port >= 0x200) { /* force MIDI */
2205             mpu_port &= 0xffff;
2206             pci_write_config_dword(chip->pci, 0x18, mpu_port | 0x01);
2207 #ifdef CONFIG_PM
2208             chip->mpu_port_saved = mpu_port;
2209 #endif
2210         } else {
2211             mpu_port = pci_resource_start(chip->pci, 2);
2212         }
2213     } else {
2214         switch (mpu_port) { /* force MIDI */
2215             case 0x300:
2216             case 0x310:
2217             case 0x320:
2218             case 0x330:
2219                 legacy_cfg &= ~(3 << 2);
2220                 legacy_cfg |= (mpu_port & 0x0030) >> 2;
2221                 break;
2222             default: /* no, use BIOS settings */
2223                 if (legacy & VIA_FUNC_ENABLE_MIDI)
2224                     mpu_port = 0x300 + ((legacy_cfg & 0x000c) << 2);
2225                 break;
2226         }
2227     }
2228     if (mpu_port >= 0x200 &&
2229         (chip->mpu_res = request_region(mpu_port, 2, "VIA82xx MPU401"))
2230         != NULL) {
2231         if (rev_h)
2232             legacy |= VIA_FUNC_MIDI_PNP; /* enable PCI I/O 2 */
2233         legacy |= VIA_FUNC_ENABLE_MIDI;
2234     } else {
2235         if (rev_h)
2236             legacy &= ~VIA_FUNC_MIDI_PNP; /* disable PCI I/O 2 */
2237         legacy &= ~VIA_FUNC_ENABLE_MIDI;
2238         mpu_port = 0;
2239     }
2240
2241
2242
2243
2244

```

```

2245 pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE, legacy);
2246 pci_write_config_byte(chip->pci, VIA_PNP_CONTROL, legacy_cfg);
2247 if (chip->mpu_res) {
2248     if (snd_mpu401_uart_new(chip->card, 0, MPU401_HW_VIA686A,
2249                             mpu_port, MPU401_INFO_INTEGRATED |
2250                             MPU401_INFO_IRQ_HOOK, -1,
2251                             &chip->rmidi) < 0) {
2252         printk(KERN_WARNING "unable to initialize MPU-401"
2253                " at 0x%lx, skipping\n", mpu_port);
2254         legacy &= ~VIA_FUNC_ENABLE_MIDI;
2255     } else {
2256         legacy &= ~VIA_FUNC_MIDI_IRQMASK; /* enable MIDI interrupt
2257 */
2258     }
2259     pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE, legacy);
2260 }
2261
2262 snd_via686_create_gameport(chip, &legacy);
2263
2264 #ifdef CONFIG_PM
2265     chip->legacy_saved = legacy;
2266     chip->legacy_cfg_saved = legacy_cfg;
2267 #endif
2268
2269     return 0;
2270 }
2271
2272 /*
2273  * proc interface
2274  */
2275
2276 static void snd_via82xx_proc_read(struct snd_info_entry *entry,
2277                                  struct snd_info_buffer *buffer)
2278 {
2279     struct via82xx *chip = entry->private_data;
2280     int i;
2281
2282     snd_iprintf(buffer, "%s\n\n", chip->card->longname);
2283     for (i = 0; i < 0xa0; i += 4) {
2284         snd_iprintf(buffer, "%02x: %08x\n", i, inl(chip->port + i));
2285     }
2286 }
2287
2288 static void __devinit snd_via82xx_proc_init(struct via82xx *chip)
2289 {
2290     struct snd_info_entry *entry;
2291
2292     if (! snd_card_proc_new(chip->card, "via82xx", &entry))
2293         snd_info_set_text_ops(entry, chip, snd_via82xx_proc_read);
2294 }
2295
2296 /*
2297  *
2298  */
2299
2300 static int snd_via82xx_chip_init(struct via82xx *chip)
2301 {
2302     unsigned int val;
2303     unsigned long end_time;
2304     unsigned char pval;
2305
2306     #if 0 /* broken on K7M? */
2307         if (chip->chip_type == TYPE_VIA686)
2308             /* disable all legacy ports */
2309             pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE, 0);
2310     #endif
2311
2312

```

```

2313     pci_read_config_byte(chip->pci, VIA_ACLINK_STAT, &pval);
2314     if (! (pval & VIA_ACLINK_C00_READY)) { /* codec not ready? */
2315         /* deassert ALink reset, force SYNC */
2316         pci_write_config_byte(chip->pci, VIA_ACLINK_CTRL,
2317                               VIA_ACLINK_CTRL_ENABLE |
2318                               VIA_ACLINK_CTRL_RESET |
2319                               VIA_ACLINK_CTRL_SYNC);
2320         udelay(100);
2321 #if 1 /* FIXME: should we do full reset here for all chip models? */
2322         pci_write_config_byte(chip->pci, VIA_ACLINK_CTRL, 0x00);
2323         udelay(100);
2324 #else
2325         /* deassert ALink reset, force SYNC (warm AC'97 reset) */
2326         pci_write_config_byte(chip->pci, VIA_ACLINK_CTRL,
2327                               VIA_ACLINK_CTRL_RESET|VIA_ACLINK_CTRL_SYNC);
2328         udelay(2);
2329 #endif
2330         /* ALink on, deassert ALink reset, VSR, SGD data out */
2331         /* note - FM data out has trouble with non VRA codecs !! */
2332         pci_write_config_byte(chip->pci, VIA_ACLINK_CTRL,
2333 VIA_ACLINK_CTRL_INIT);
2334         udelay(100);
2335     }
2336
2337     /* Make sure VRA is enabled, in case we didn't do a
2338      * complete codec reset, above */
2339     pci_read_config_byte(chip->pci, VIA_ACLINK_CTRL, &pval);
2340     if ((pval & VIA_ACLINK_CTRL_INIT) != VIA_ACLINK_CTRL_INIT) {
2341         /* ALink on, deassert ALink reset, VSR, SGD data out */
2342         /* note - FM data out has trouble with non VRA codecs !! */
2343         pci_write_config_byte(chip->pci, VIA_ACLINK_CTRL,
2344 VIA_ACLINK_CTRL_INIT);
2345         udelay(100);
2346     }
2347     /* wait until codec ready */
2348     end_time = jiffies + msecs_to_jiffies(750);
2349     do {
2350         pci_read_config_byte(chip->pci, VIA_ACLINK_STAT, &pval);
2351         if (pval & VIA_ACLINK_C00_READY) /* primary codec ready */
2352             break;
2353         schedule_timeout_uninterruptible(1);
2354     } while (time_before(jiffies, end_time));
2355
2356     if ((val = snd_via82xx_codec_xread(chip)) & VIA_REG_AC97_BUSY)
2357         snd_printk(KERN_ERR "AC'97 codec is not ready [0x%x]\n", val);
2358
2359 #if 0 /* FIXME: we don't support the second codec yet so skip the detection now.. */
2360     snd_via82xx_codec_xwrite(chip, VIA_REG_AC97_READ |
2361                               VIA_REG_AC97_SECONDARY_VALID |
2362                               (VIA_REG_AC97_CODEC_ID_SECONDARY <<
2363 VIA_REG_AC97_CODEC_ID_SHIFT));
2364     end_time = jiffies + msecs_to_jiffies(750);
2365     snd_via82xx_codec_xwrite(chip, VIA_REG_AC97_READ |
2366                               VIA_REG_AC97_SECONDARY_VALID |
2367                               (VIA_REG_AC97_CODEC_ID_SECONDARY <<
2368 VIA_REG_AC97_CODEC_ID_SHIFT));
2369     do {
2370         if ((val = snd_via82xx_codec_xread(chip)) &
2371 VIA_REG_AC97_SECONDARY_VALID) {
2372             chip->ac97_secondary = 1;
2373             goto __ac97_ok2;
2374         }
2375         schedule_timeout_uninterruptible(1);
2376     } while (time_before(jiffies, end_time));
2377     /* This is ok, the most of motherboards have only one codec */
2378     __ac97_ok2:
2379 #endif
2380 #endif

```

```

2381     if (chip->chip_type == TYPE_VIA686) {
2382         /* route FM trap to IRQ, disable FM trap */
2383         pci_write_config_byte(chip->pci, VIA_FM_NMI_CTRL, 0);
2384         /* disable all GPI interrupts */
2385         outl(0, VIAREG(chip, GPI_INTR));
2386     }
2387
2388     if (chip->chip_type != TYPE_VIA686) {
2389         /* Workaround for Award BIOS bug:
2390          * DXS channels don't work properly with VRA if MC97 is disabled.
2391          */
2392         struct pci_dev *pci;
2393         pci = pci_get_device(0x1106, 0x3068, NULL); /* MC97 */
2394         if (pci) {
2395             unsigned char data;
2396             pci_read_config_byte(pci, 0x44, &data);
2397             pci_write_config_byte(pci, 0x44, data | 0x40);
2398             pci_dev_put(pci);
2399         }
2400     }
2401
2402     if (chip->chip_type != TYPE_VIA8233A) {
2403         int i, idx;
2404         for (idx = 0; idx < 4; idx++) {
2405             unsigned long port = chip->port + 0x10 * idx;
2406             for (i = 0; i < 2; i++) {
2407                 chip->playback_volume[idx][i] = chip-
2408 >playback_volume_c[i];
2409                 outb(chip->playback_volume_c[i],
2410                     port + VIA_REG_OFS_PLAYBACK_VOLUME_L + i);
2411             }
2412         }
2413     }
2414
2415     return 0;
2416 }
2417
2418 #ifdef CONFIG_PM
2419 /*
2420  * power management
2421  */
2422 static int snd_via82xx_suspend(struct pci_dev *pci, pm_message_t state)
2423 {
2424     struct snd_card *card = pci_get_drvdata(pci);
2425     struct via82xx *chip = card->private_data;
2426     int i;
2427     snd_power_change_state(card, SNDRV_CTL_POWER_D3hot);
2428     for (i = 0; i < 2; i++)
2429         snd_pcm_suspend_all(chip->pcms[i]);
2430     for (i = 0; i < chip->num_devs; i++)
2431         snd_via82xx_channel_reset(chip, &chip->devs[i]);
2432     synchronize_irq(chip->irq);
2433     snd_ac97_suspend(chip->ac97);
2434
2435     /* save misc values */
2436     if (chip->chip_type != TYPE_VIA686) {
2437         pci_read_config_byte(chip->pci, VIA8233_SPDIF_CTRL, &chip-
2438 >spdif_ctrl_saved);
2439         chip->capture_src_saved[0] = inb(chip->port +
2440 VIA_REG_CAPTURE_CHANNEL);
2441         chip->capture_src_saved[1] = inb(chip->port +
2442 VIA_REG_CAPTURE_CHANNEL + 0x10);
2443     }
2444     pci_disable_device(pci);
2445     pci_save_state(pci);
2446     pci_set_power_state(pci, pci_choose_state(pci, state));
2447     return 0;
2448 }

```



```

2449
2450 static int snd_via82xx_resume(struct pci_dev *pci)
2451 {
2452     struct snd_card *card = pci_get_drvdata(pci);
2453     struct via82xx *chip = card->private_data;
2454     int i;
2455
2456     pci_set_power_state(pci, PCI_D0);
2457     pci_restore_state(pci);
2458     if (pci_enable_device(pci) < 0) {
2459         printk(KERN_ERR "via82xx: pci_enable_device failed, "
2460             "disabling device\n");
2461         snd_card_disconnect(card);
2462         return -EIO;
2463     }
2464     pci_set_master(pci);
2465
2466     snd_via82xx_chip_init(chip);
2467
2468     if (chip->chip_type == TYPE_VIA686) {
2469         if (chip->mpu_port_saved)
2470             pci_write_config_dword(chip->pci, 0x18, chip-
2471 >mpu_port_saved | 0x01);
2472         pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE, chip-
2473 >legacy_saved);
2474         pci_write_config_byte(chip->pci, VIA_PNP_CONTROL, chip-
2475 >legacy_cfg_saved);
2476     } else {
2477         pci_write_config_byte(chip->pci, VIA8233_SPDIF_CTRL, chip-
2478 >spdif_ctrl_saved);
2479         outb(chip->capture_src_saved[0], chip->port +
2480 VIA_REG_CAPTURE_CHANNEL);
2481         outb(chip->capture_src_saved[1], chip->port +
2482 VIA_REG_CAPTURE_CHANNEL + 0x10);
2483     }
2484
2485     snd_ac97_resume(chip->ac97);
2486
2487     for (i = 0; i < chip->num_devs; i++)
2488         snd_via82xx_channel_reset(chip, &chip->devs[i]);
2489
2490     snd_power_change_state(card, SNDRV_CTL_POWER_D0);
2491     return 0;
2492 }
2493 #endif /* CONFIG_PM */
2494
2495 static int snd_via82xx_free(struct via82xx *chip)
2496 {
2497     unsigned int i;
2498
2499     if (chip->irq < 0)
2500         goto __end_hw;
2501     /* disable interrupts */
2502     for (i = 0; i < chip->num_devs; i++)
2503         snd_via82xx_channel_reset(chip, &chip->devs[i]);
2504
2505     if (chip->irq >= 0)
2506         free_irq(chip->irq, chip);
2507 __end_hw:
2508     release_and_free_resource(chip->mpu_res);
2509     pci_release_regions(chip->pci);
2510
2511     if (chip->chip_type == TYPE_VIA686) {
2512         snd_via686_free_gameport(chip);
2513         pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE, chip->old_legacy);
2514         pci_write_config_byte(chip->pci, VIA_PNP_CONTROL, chip-
2515 >old_legacy_cfg);
2516     }

```

```

2517     pci_disable_device(chip->pci);
2518     kfree(chip);
2519     return 0;
2520 }
2521
2522 static int snd_via82xx_dev_free(struct snd_device *device)
2523 {
2524     struct via82xx *chip = device->device_data;
2525     return snd_via82xx_free(chip);
2526 }
2527
2528 static int __devinit snd_via82xx_create(struct snd_card *card,
2529                                         struct pci_dev *pci,
2530                                         int chip_type,
2531                                         int revision,
2532                                         unsigned int ac97_clock,
2533                                         struct via82xx ** r_via)
2534 {
2535     struct via82xx *chip;
2536     int err;
2537     static struct snd_device_ops ops = {
2538         .dev_free =      snd_via82xx_dev_free,
2539     };
2540
2541     if ((err = pci_enable_device(pci)) < 0)
2542         return err;
2543
2544     if ((chip = kzalloc(sizeof(*chip), GFP_KERNEL)) == NULL) {
2545         pci_disable_device(pci);
2546         return -ENOMEM;
2547     }
2548
2549     chip->chip_type = chip_type;
2550     chip->revision = revision;
2551
2552     spin_lock_init(&chip->reg_lock);
2553     spin_lock_init(&chip->rates[0].lock);
2554     spin_lock_init(&chip->rates[1].lock);
2555     chip->card = card;
2556     chip->pci = pci;
2557     chip->irq = -1;
2558
2559     pci_read_config_byte(pci, VIA_FUNC_ENABLE, &chip->old_legacy);
2560     pci_read_config_byte(pci, VIA_PNP_CONTROL, &chip->old_legacy_cfg);
2561     pci_write_config_byte(chip->pci, VIA_FUNC_ENABLE,
2562                           chip->old_legacy &
2563                           ~(VIA_FUNC_ENABLE_SB|VIA_FUNC_ENABLE_FM));
2564
2565     if ((err = pci_request_regions(pci, card->driver)) < 0) {
2566         kfree(chip);
2567         pci_disable_device(pci);
2568         return err;
2569     }
2570     chip->port = pci_resource_start(pci, 0);
2571     if (request_irq(pci->irq,
2572                    chip_type == TYPE_VIA8233 ?
2573                    snd_via8233_interrupt : snd_via686_interrupt,
2574                    IRQF_SHARED,
2575                    KBUILD_MODNAME, chip)) {
2576         snd_printk(KERN_ERR "unable to grab IRQ %d\n", pci->irq);
2577         snd_via82xx_free(chip);
2578         return -EBUSY;
2579     }
2580     chip->irq = pci->irq;
2581     if (ac97_clock >= 8000 && ac97_clock <= 48000)
2582         chip->ac97_clock = ac97_clock;
2583     synchronize_irq(chip->irq);
2584

```

```

2585     if ((err = snd_via82xx_chip_init(chip)) < 0) {
2586         snd_via82xx_free(chip);
2587         return err;
2588     }
2589
2590     if ((err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops)) < 0) {
2591         snd_via82xx_free(chip);
2592         return err;
2593     }
2594
2595     /* The 8233 ac97 controller does not implement the master bit
2596      * in the pci command register. IMHO this is a violation of the PCI spec.
2597      * We call pci_set_master here because it does not hurt. */
2598     pci_set_master(pci);
2599
2600     snd_card_set_dev(card, &pci->dev);
2601
2602     *r_via = chip;
2603     return 0;
2604 }
2605
2606 struct via823x_info {
2607     int revision;
2608     char *name;
2609     int type;
2610 };
2611 static struct via823x_info via823x_cards[] __devinitdata = {
2612     { VIA_REV_PRE_8233, "VIA 8233-Pre", TYPE_VIA8233 },
2613     { VIA_REV_8233C, "VIA 8233C", TYPE_VIA8233 },
2614     { VIA_REV_8233, "VIA 8233", TYPE_VIA8233 },
2615     { VIA_REV_8233A, "VIA 8233A", TYPE_VIA8233A },
2616     { VIA_REV_8235, "VIA 8235", TYPE_VIA8233 },
2617     { VIA_REV_8237, "VIA 8237", TYPE_VIA8233 },
2618     { VIA_REV_8251, "VIA 8251", TYPE_VIA8233 },
2619 };
2620
2621 /*
2622  * auto detection of DXS channel supports.
2623  */
2624
2625 static struct snd_pci_quirk dxs_whitelist[] __devinitdata = {
2626     SND_PCI_QUIRK(0x1005, 0x4710, "Avance Logic Mobo", VIA_DXS_ENABLE),
2627     SND_PCI_QUIRK(0x1019, 0x0996, "ESC Mobo", VIA_DXS_48K),
2628     SND_PCI_QUIRK(0x1019, 0x0a81, "ECS K7VTA3 v8.0", VIA_DXS_NO_VRA),
2629     SND_PCI_QUIRK(0x1019, 0x0a85, "ECS L7VMM2", VIA_DXS_NO_VRA),
2630     SND_PCI_QUIRK_VENDOR(0x1019, "ESC K8", VIA_DXS_SRC),
2631     SND_PCI_QUIRK(0x1019, 0xaa01, "ESC K8T890-A", VIA_DXS_SRC),
2632     SND_PCI_QUIRK(0x1025, 0x0033, "Acer Inspire 1353LM", VIA_DXS_NO_VRA),
2633     SND_PCI_QUIRK(0x1025, 0x0046, "Acer Aspire 1524 WLMi", VIA_DXS_SRC),
2634     SND_PCI_QUIRK_VENDOR(0x1043, "ASUS A7/A8", VIA_DXS_NO_VRA),
2635     SND_PCI_QUIRK_VENDOR(0x1071, "Diverse Notebook", VIA_DXS_NO_VRA),
2636     SND_PCI_QUIRK(0x10cf, 0x118e, "FSC Laptop", VIA_DXS_ENABLE),
2637     SND_PCI_QUIRK_VENDOR(0x1106, "ASRock", VIA_DXS_SRC),
2638     SND_PCI_QUIRK(0x1297, 0xa231, "Shuttle AK31v2", VIA_DXS_SRC),
2639     SND_PCI_QUIRK(0x1297, 0xa232, "Shuttle", VIA_DXS_SRC),
2640     SND_PCI_QUIRK(0x1297, 0xc160, "Shuttle Sk41G", VIA_DXS_SRC),
2641     SND_PCI_QUIRK(0x1458, 0xa002, "Gigabyte GA-7VAXP", VIA_DXS_ENABLE),
2642     SND_PCI_QUIRK(0x1462, 0x3800, "MSI KT266", VIA_DXS_ENABLE),
2643     SND_PCI_QUIRK(0x1462, 0x7120, "MSI KT4V", VIA_DXS_ENABLE),
2644     SND_PCI_QUIRK(0x1462, 0x7142, "MSI K8MM-V", VIA_DXS_ENABLE),
2645     SND_PCI_QUIRK_VENDOR(0x1462, "MSI Mobo", VIA_DXS_SRC),
2646     SND_PCI_QUIRK(0x147b, 0x1401, "ABIT KD7(-RAID)", VIA_DXS_ENABLE),
2647     SND_PCI_QUIRK(0x147b, 0x1411, "ABIT VA-20", VIA_DXS_ENABLE),
2648     SND_PCI_QUIRK(0x147b, 0x1413, "ABIT KV8 Pro", VIA_DXS_ENABLE),
2649     SND_PCI_QUIRK(0x147b, 0x1415, "ABIT AV8", VIA_DXS_NO_VRA),
2650     SND_PCI_QUIRK(0x14ff, 0x0403, "Twinhead mobo", VIA_DXS_ENABLE),
2651     SND_PCI_QUIRK(0x14ff, 0x0408, "Twinhead laptop", VIA_DXS_SRC),
2652     SND_PCI_QUIRK(0x1558, 0x4701, "Clevo D470", VIA_DXS_SRC),

```

```

        SND_PCI_QUIRK(0x1584, 0x8120, "Diverse Laptop", VIA_DXS_ENABLE),
        SND_PCI_QUIRK(0x1584, 0x8123, "Targa/Uniwill", VIA_DXS_NO_VRA),
        SND_PCI_QUIRK(0x161f, 0x202b, "Amira Notebook", VIA_DXS_NO_VRA),
        SND_PCI_QUIRK(0x161f, 0x2032, "m680x machines", VIA_DXS_48K),
        SND_PCI_QUIRK(0x1631, 0xe004, "PB EasyNote 3174", VIA_DXS_ENABLE),
        SND_PCI_QUIRK(0x1695, 0x3005, "EPoX EP-8K9A", VIA_DXS_ENABLE),
        SND_PCI_QUIRK_VENDOR(0x1695, "EPoX mobo", VIA_DXS_SRC),
        SND_PCI_QUIRK_VENDOR(0x16f3, "Jetway K8", VIA_DXS_SRC),
        SND_PCI_QUIRK_VENDOR(0x1734, "FSC Laptop", VIA_DXS_SRC),
        SND_PCI_QUIRK(0x1849, 0x3059, "ASRock K7VM2", VIA_DXS_NO_VRA),
        SND_PCI_QUIRK_VENDOR(0x1849, "ASRock mobo", VIA_DXS_SRC),
        SND_PCI_QUIRK(0x1919, 0x200a, "Soltek SL-K8", VIA_DXS_NO_VRA),
        SND_PCI_QUIRK(0x4005, 0x4710, "MSI K7T266", VIA_DXS_SRC),
        { } /* terminator */
};

static int __devinit check_dxs_list(struct pci_dev *pci, int revision)
{
    const struct snd_pci_quirk *w;

    w = snd_pci_quirk_lookup(pci, dxs_whitelist);
    if (w) {
        snd_printdd(KERN_INFO "via82xx: DXS white list for %s found\n",
                    w->name);
        return w->value;
    }

    /* for newer revision, default to DXS_SRC */
    if (revision >= VIA_REV_8235)
        return VIA_DXS_SRC;

    /*
     * not detected, try 48k rate only to be sure.
     */
    printk(KERN_INFO "via82xx: Assuming DXS channels with 48k fixed sample
rate.\n");
    printk(KERN_INFO "           Please try dxs_support=5 option\n");
    printk(KERN_INFO "           and report if it works on your machine.\n");
    printk(KERN_INFO "           For more details, read ALSA-
Configuration.txt.\n");
    return VIA_DXS_48K;
};

static int __devinit snd_via82xx_probe(struct pci_dev *pci,
                                     const struct pci_device_id *pci_id)
{
    struct snd_card *card;
    struct via82xx *chip;
    int chip_type = 0, card_type;
    unsigned int i;
    int err;

    err = snd_card_create(index, id, THIS_MODULE, 0, &card);
    if (err < 0)
        return err;

    card_type = pci_id->driver_data;
    switch (card_type) {
    case TYPE_CARD_VIA686:
        strcpy(card->driver, "VIA686A");
        sprintf(card->shortname, "VIA 82C686A/B rev%x", pci->revision);
        chip_type = TYPE_VIA686;
        break;

```

```

case TYPE_CARD_VIA8233:
    chip_type = TYPE_VIA8233;
    sprintf(card->shortname, "VIA 823x rev%x", pci->revision);
    for (i = 0; i < ARRAY_SIZE(via823x_cards); i++) {
        if (pci->revision == via823x_cards[i].revision) {
            chip_type = via823x_cards[i].type;
            strcpy(card->shortname, via823x_cards[i].name);
            break;
        }
    }
    if (chip_type != TYPE_VIA8233A) {
        if (dxs_support == VIA_DXS_AUTO)
            dxs_support = check_dxs_list(pci, pci->revision);
        /* force to use VIA8233 or 8233A model according to
         * dxs_support module option
         */
        if (dxs_support == VIA_DXS_DISABLE)
            chip_type = TYPE_VIA8233A;
        else
            chip_type = TYPE_VIA8233;
    }
    if (chip_type == TYPE_VIA8233A)
        strcpy(card->driver, "VIA8233A");
    else if (pci->revision >= VIA_REV_8237)
        strcpy(card->driver, "VIA8237"); /* no slog assignment */
    else
        strcpy(card->driver, "VIA8233");
    break;
default:
    snd_printk(KERN_ERR "invalid card type %d\n", card_type);
    err = -EINVAL;
    goto __error;
}

if ((err = snd_via82xx_create(card, pci, chip_type, pci->revision,
                             ac97_clock, &chip)) < 0)
    goto __error;
card->private_data = chip;
if ((err = snd_via82xx_mixer_new(chip, ac97_quirk)) < 0)
    goto __error;

if (chip_type == TYPE_VIA686) {
    if ((err = snd_via686_pcm_new(chip)) < 0 ||
        (err = snd_via686_init_misc(chip)) < 0)
        goto __error;
} else {
    if (chip_type == TYPE_VIA8233A) {
        if ((err = snd_via8233a_pcm_new(chip)) < 0)
            goto __error;
        // chip->dxs_fixed = 1; /* FIXME: use 48k for DXS #3? */
    } else {
        if ((err = snd_via8233_pcm_new(chip)) < 0)
            goto __error;
        if (dxs_support == VIA_DXS_48K)
            chip->dxs_fixed = 1;
        else if (dxs_support == VIA_DXS_NO_VRA)
            chip->no_vra = 1;
        else if (dxs_support == VIA_DXS_SRC) {
            chip->no_vra = 1;
            chip->dxs_src = 1;
        }
    }
    if ((err = snd_via8233_init_misc(chip)) < 0)
        goto __error;
}
/* disable interrupts */
for (i = 0; i < chip->num_devs; i++)
    snd_via82xx_channel_reset(chip, &chip->devs[i]);

```

```
        snprintf(card->longname, sizeof(card->longname),
                 "%s with %s at %#lx, irq %d", card->shortname,
                 snd_ac97_get_short_name(chip->ac97), chip->port, chip->irq);

    snd_via82xx_proc_init(chip);

    if ((err = snd_card_register(card)) < 0) {
        snd_card_free(card);
        return err;
    }
    pci_set_drvdata(pci, card);
    return 0;

__error:
    snd_card_free(card);
    return err;
}

static void __devexit snd_via82xx_remove(struct pci_dev *pci)
{
    snd_card_free(pci_get_drvdata(pci));
    pci_set_drvdata(pci, NULL);
}

static struct pci_driver driver = {
    .name = KBUILD_MODNAME,
    .id_table = snd_via82xx_ids,
    .probe = snd_via82xx_probe,
    .remove = __devexit_p(snd_via82xx_remove),
#ifdef CONFIG_PM
    .suspend = snd_via82xx_suspend,
    .resume = snd_via82xx_resume,
#endif
};

static int __init alsa_card_via82xx_init(void)
{
    return pci_register_driver(&driver);
}

static void __exit alsa_card_via82xx_exit(void)
{
    pci_unregister_driver(&driver);
}

module_init(alsa_card_via82xx_init)
module_exit(alsa_card_via82xx_exit)
```