```c
/*
 * (C) Copyright 2002
 * Sysgo Real-Time Solutions, GmbH <www.elinos.com>
 * Alex Zuepke <azu@sysgo.de>
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

#include <common.h>

ulong myflush(void);


#define FLASH_BANK_SIZE PHYS_FLASH_SIZE
#define MAIN_SECT_SIZE  0x10000         /* 64 KB */

flash_info_t    flash_info[CFG_MAX_FLASH_BANKS]; // include/configs/smdk2410.h에 1로 지정


#define CMD_READ_ARRAY          0x000000F0
#define CMD_UNLOCK1             0x000000AA
#define CMD_UNLOCK2             0x00000055
#define CMD_ERASE_SETUP         0x00000080
#define CMD_ERASE_CONFIRM       0x00000030
#define CMD_PROGRAM             0x000000A0
#define CMD_UNLOCK_BYPASS       0x00000020

#define MEM_FLASH_ADDR1         (*(volatile u16 *)(CFG_FLASH_BASE + (0x00000555 << 1)))
#define MEM_FLASH_ADDR2         (*(volatile u16 *)(CFG_FLASH_BASE + (0x000002AA << 1)))

#define BIT_ERASE_DONE          0x00000080
#define BIT_RDY_MASK            0x00000080
#define BIT_PROGRAM_ERROR       0x00000020
#define BIT_TIMEOUT             0x80000000 /* our flag */

#define READY 1
#define ERR   2
#define TMO   4

/*-----------------------------------------------------------------------
 */

ulong flash_init(void)
{
    int i, j;
    ulong size = 0;

    for (i = 0; i < CFG_MAX_FLASH_BANKS; i++)
    {
        ulong flashbase = 0;
        flash_info[i].flash_id =
#if defined(CONFIG_AMD_LV400)
          (AMD_MANUFACT & FLASH_VENDMASK) |
          (AMD_ID_LV400B & FLASH_TYPEMASK);
#elif defined(CONFIG_AMD_LV800)
          (AMD_MANUFACT & FLASH_VENDMASK) |
          (AMD_ID_LV800B & FLASH_TYPEMASK);
#else
#error "Unknown flash configured"
#endif
        flash_info[i].size = FLASH_BANK_SIZE;
        flash_info[i].sector_count = CFG_MAX_FLASH_SECT;
        memset(flash_info[i].protect, 0, CFG_MAX_FLASH_SECT);
        if (i == 0)
          flashbase = PHYS_FLASH_1;
        else
```

```
 83           panic("configured to many flash banks!\n");
 84         for (j = 0; j < flash_info[i].sector_count; j++)
 85         {
 86             if (j <= 3)
 87             {
 88                 /* 1st one is 16 KB */
 89                 if (j == 0)
 90                 {
 91                         flash_info[i].start[j] = flashbase + 0;
 92                 }
 93
 94                 /* 2nd and 3rd are both 8 KB */
 95                 if ((j == 1) || (j == 2))
 96                 {
 97                         flash_info[i].start[j] = flashbase + 0x4000 + (j-1)*0x2000;
 98                 }
 99
100                 /* 4th 32 KB */
101                 if (j == 3)
102                 {
103                         flash_info[i].start[j] = flashbase + 0x8000;
104                 }
105             }
106             else
107             {
108                 flash_info[i].start[j] = flashbase + (j - 3)*MAIN_SECT_SIZE;
109             }
110         }
111         size += flash_info[i].size;
112     }
113
114     flash_protect(FLAG_PROTECT_SET,
115                   CFG_FLASH_BASE,
116                   CFG_FLASH_BASE + monitor_flash_len - 1,
117                   &flash_info[0]);
118
119     flash_protect(FLAG_PROTECT_SET,
120                   CFG_ENV_ADDR,
121                   CFG_ENV_ADDR + CFG_ENV_SIZE - 1,
122                   &flash_info[0]);
123
124     return size;
125 }
126
127 /*-----------------------------------------------------------------------
128  */
129 void flash_print_info  (flash_info_t *info)
130 {
131     int i;
132
133     switch (info->flash_id & FLASH_VENDMASK)
134     {
135     case (AMD_MANUFACT & FLASH_VENDMASK):
136         printf("AMD: ");
137         break;
138     default:
139         printf("Unknown Vendor ");
140         break;
141     }
142
143     switch (info->flash_id & FLASH_TYPEMASK)
144     {
145     case (AMD_ID_LV400B & FLASH_TYPEMASK):
146         printf("1x Amd29LV400BB (4Mbit)\n");
147         break;
148     case (AMD_ID_LV800B & FLASH_TYPEMASK):
149         printf("1x Amd29LV800BB (8Mbit)\n");
150         break;
151     default:
152         printf("Unknown Chip Type\n");
153         goto Done;
154         break;
155     }
156
157     printf("  Size: %ld MB in %d Sectors\n",
158            info->size >> 20, info->sector_count);
159
160     printf("  Sector Start Addresses:");
161     for (i = 0; i < info->sector_count; i++)
162     {
163         if ((i % 5) == 0)
164         {
```

```
165             printf ("\n   ");
166         }
167         printf (" %08lX%s", info->start[i],
168                 info->protect[i] ? " (RO)" : "     ");
169     }
170     printf ("\n");
171
172 Done:
173 }
174
175 /*-----------------------------------------------------------------------
176  */
177
178 int     flash_erase (flash_info_t *info, int s_first, int s_last)
179 {
180     ushort result;
181     int iflag, cflag, prot, sect;
182     int rc = ERR_OK;
183     int chip;
184
185     /* first look for protection bits */
186
187     if (info->flash_id == FLASH_UNKNOWN)
188         return ERR_UNKNOWN_FLASH_TYPE;
189
190     if ((s_first < 0) || (s_first > s_last)) {
191         return ERR_INVAL;
192     }
193
194     if ((info->flash_id & FLASH_VENDMASK) !=
195         (AMD_MANUFACT & FLASH_VENDMASK)) {
196         return ERR_UNKNOWN_FLASH_VENDOR;
197     }
198
199     prot = 0;
200     for (sect=s_first; sect<=s_last; ++sect) {
201         if (info->protect[sect]) {
202             prot++;
203         }
204     }
205     if (prot)
206         return ERR_PROTECTED;
207
208     /*
209      * Disable interrupts which might cause a timeout
210      * here. Remember that our exception vectors are
211      * at address 0 in the flash, and we don't want a
212      * (ticker) exception to happen while the flash
213      * chip is in programming mode.
214      */
215     cflag = icache_status();
216     icache_disable();
217     iflag = disable_interrupts();
218
219     /* Start erase on unprotected sectors */
220     for (sect = s_first; sect<=s_last && !ctrlc(); sect++)
221     {
222         printf("Erasing sector %2d ... ", sect);
223
224         /* arm simple, non interrupt dependent timer */
225         reset_timer_masked();
226
227         if (info->protect[sect] == 0)
228         {       /* not protected */
229             vu_short *addr = (vu_short *)(info->start[sect]);
230
231             MEM_FLASH_ADDR1 = CMD_UNLOCK1;
232             MEM_FLASH_ADDR2 = CMD_UNLOCK2;
233             MEM_FLASH_ADDR1 = CMD_ERASE_SETUP;
234
235             MEM_FLASH_ADDR1 = CMD_UNLOCK1;
236             MEM_FLASH_ADDR2 = CMD_UNLOCK2;
237             *addr = CMD_ERASE_CONFIRM;
238
239             /* wait until flash is ready */
240             chip = 0;
241
242             do
243             {
244                 result = *addr;
245
246                 /* check timeout */
```

```
247                 if (get_timer_masked() > CFG_FLASH_ERASE_TOUT)
248                 {
249                     MEM_FLASH_ADDR1 = CMD_READ_ARRAY;
250                     chip = TMO;
251                     break;
252                 }
253
254                 if (!chip && (result & 0xFFFF) & BIT_ERASE_DONE)
255                         chip = READY;
256
257                 if (!chip && (result & 0xFFFF) & BIT_PROGRAM_ERROR)
258                         chip = ERR;
259
260             }  while (!chip);
261
262             MEM_FLASH_ADDR1 = CMD_READ_ARRAY;
263
264             if (chip == ERR)
265             {
266                 rc = ERR_PROG_ERROR;
267                 goto outahere;
268             }
269             if (chip == TMO)
270             {
271                 rc = ERR_TIMOUT;
272                 goto outahere;
273             }
274
275             printf("ok.\n");
276         }
277         else /* it was protected */
278         {
279             printf("protected!\n");
280         }
281     }
282
283     if (ctrlc())
284       printf("User Interrupt!\n");
285
286 outahere:
287     /* allow flash to settle - wait 10 ms */
288     udelay_masked(10000);
289
290     if (iflag)
291       enable_interrupts();
292
293     if (cflag)
294       icache_enable();
295
296     return rc;
297 }
298
299 /*-----------------------------------------------------------------------
300  * Copy memory to flash
301  */
302
303 volatile static int write_hword (flash_info_t *info, ulong dest, ushort data)
304 {
305     vu_short *addr = (vu_short *)dest;
306     ushort result;
307     int rc = ERR_OK;
308     int cflag, iflag;
309     int chip;
310
311     /*
312      * Check if Flash is (sufficiently) erased
313      */
314     result = *addr;
315     if ((result & data) != data)
316         return ERR_NOT_ERASED;
317
318
319     /*
320      * Disable interrupts which might cause a timeout
321      * here. Remember that our exception vectors are
322      * at address 0 in the flash, and we don't want a
323      * (ticker) exception to happen while the flash
324      * chip is in programming mode.
325      */
326     cflag = icache_status();
327     icache_disable();
328     iflag = disable_interrupts();
```

```
329
330     MEM_FLASH_ADDR1 = CMD_UNLOCK1;
331     MEM_FLASH_ADDR2 = CMD_UNLOCK2;
332     MEM_FLASH_ADDR1 = CMD_UNLOCK_BYPASS;
333     *addr = CMD_PROGRAM;
334     *addr = data;
335
336     /* arm simple, non interrupt dependent timer */
337     reset_timer_masked();
338
339     /* wait until flash is ready */
340     chip = 0;
341     do
342     {
343         result = *addr;
344
345         /* check timeout */
346         if (get_timer_masked() > CFG_FLASH_ERASE_TOUT)
347         {
348             chip = ERR | TMO;
349             break;
350         }
351         if (!chip && ((result & 0x80) == (data & 0x80)))
352                 chip = READY;
353
354         if (!chip && ((result & 0xFFFF) & BIT_PROGRAM_ERROR))
355         {
356                 result = *addr;
357
358                 if ((result & 0x80) == (data & 0x80))
359                         chip = READY;
360                 else
361                         chip = ERR;
362         }
363
364     }  while (!chip);
365
366     *addr = CMD_READ_ARRAY;
367
368     if (chip == ERR || *addr != data)
369         rc = ERR_PROG_ERROR;
370
371     if (iflag)
372       enable_interrupts();
373
374     if (cflag)
375       icache_enable();
376
377     return rc;
378 }
379
380 /*-----------------------------------------------------------------------
381  * Copy memory to flash.
382  */
383
384 int write_buff (flash_info_t *info, uchar *src, ulong addr, ulong cnt)
385 {
386     ulong cp, wp;
387     int l;
388     int i, rc;
389     ushort data;
390
391     wp = (addr & ~1);   /* get lower word aligned address */
392
393     /*
394      * handle unaligned start bytes
395      */
396     if ((l = addr - wp) != 0) {
397         data = 0;
398         for (i=0, cp=wp; i<l; ++i, ++cp) {
399             data = (data >> 8) | (*(uchar *)cp << 8);
400         }
401         for (; i<2 && cnt>0; ++i) {
402             data = (data >> 8) | (*src++ << 8);
403             --cnt;
404             ++cp;
405         }
406         for (; cnt==0 && i<2; ++i, ++cp) {
407             data = (data >> 8) | (*(uchar *)cp << 8);
408         }
409
410         if ((rc = write_hword(info, wp, data)) != 0) {
```

```
411             return (rc);
412         }
413         wp += 2;
414     }
415
416     /*
417      * handle word aligned part
418      */
419     while (cnt >= 2) {
420         data = *((vu_short*)src);
421         if ((rc = write_hword(info, wp, data)) != 0) {
422             return (rc);
423         }
424         src += 2;
425         wp  += 2;
426         cnt -= 2;
427     }
428
429     if (cnt == 0) {
430         return ERR_OK;
431     }
432
433     /*
434      * handle unaligned tail bytes
435      */
436     data = 0;
437     for (i=0, cp=wp; i<2 && cnt>0; ++i, ++cp) {
438         data = (data >> 8) | (*src++ << 8);
439         --cnt;
440     }
441     for (; i<2; ++i, ++cp) {
442         data = (data >> 8) | (*(uchar *)cp << 8);
443     }
444
445     return write_hword(info, wp, data);
446 }
447
```