Accessing Intel ICH/PCH GPIOs

2017-11-08

Tags: software, electronics.

In this note I describe the method required for accessing GPIOs on (in theory) every Intel chipset from ICH0 to (at least) Series 100 PCH.

Table of Contents

Overview

Historically (going as far back as ICH0), Intel used a special PCI function to implement a multitude of legacy functions (such as `A20#`, the LPC bus, etc). The Intel lingo for this is “D31:F0”, as in “device 31, function 0”, but it shows up in `lspci` as `00:1f.0 ISA bridge`. In this note, I will use the Intel terminology for PCI devices.

GPIOs on ICH0..9 and Series 5..9 PCH

On these platforms, D31:F0 has an I/O BAR. The size of the I/O space was increased for ICH6, but otherwise the GPIO interface is identical. It is comprised of a set of I/O ports. To access them, it is enough to place the I/O space of the function in the global I/O space, and enable decoding.

The implementation is as follows:

- The vendor and device IDs are checked to make sure we’re not crashing the system.
- The I/O BAR is configured via libpci.
- The I/O permissions are requested via ioperm.
- Finally, GPIO registers can be used.

GPIOs on Series 100 PCH

On these platforms, D31:F0 is solely dedicated to being an LPC bridge. The GPIOs are located in what Intel calls “private configuration space”, accessible through a “primary to sideband bridge” through “target port IDs”. All this seems extremely opaque, but in reality very little has changed.

The “primary to sideband bridge” is simply a PCI function (located at D31:F1) that has BAR0, a memory BAR, “private configuration space”, initialized by platform firmware to point to some location it finds convenient. To prevent the OS from reassigning the BAR, the firmware “hides” the device, namely sets a bit in the configuration space that causes all reads to return all-ones. (Writes still go through.) The “target port ID” is the bits [23:16] of the D31:F1 BAR0.

The implementation is as follows:

- The vendor and device IDs are checked to make sure we're not crashing the system.
- The D31:F1 function is blindly enabled bypassing the operating system as to avoid changing the BAR, and verified to have been enabled correctly.
- The memory BAR is read from D31:F1.
- The D31:F1 function is quickly disabled again.
- The address read from BAR is mapped into the process' address space.
- Finally, GPIO registers can be used.

Intel documentation

- Intel® 82801EB I/O Controller Hub 5 (ICH5) / Intel® 82801ER I/O Controller Hub 5 R (ICH5R) Datasheet (Document 252516-001)
- Intel® 9 Series Chipset Family Platform Controller Hub (PCH) Datasheet (Document 330550-002)
- Intel® 100 Series and Intel® C230 Series Chipset Family Platform Controller Hub (PCH) Datasheet – Volume 1 of 2 (Document 332690-004EN)
- Intel® 100 Series and Intel® C230 Series Chipset Family Platform Controller Hub (PCH) Datasheet - Volume 2 of 2 (Document 332691-002EN)

Code

Run `make && sudo ./gpioke` and enjoy a printout of GPIO status live from your chipset. This demo was written very carefully and is not supposed to ever crash your machine. However, it has not undergone a lot of live testing.

Makefile

```
CFLAGS = -std=c11 -Wall -g
LIBS   = -lpci

gpioke: gpioke.c
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)
```

gpioke.c (download)

```
/*
 * See:
 * - http://lxr.free-electrons.com/source/drivers/mfd/lpc_ich.c
 * - http://lxr.free-electrons.com/source/drivers/gpio/gpio-ich.c
 * - Intel document 252516-001 (ICH5)
 * - Intel document 330550-002 (9 Series PCH)
 * - Intel documents 332690-004 and 332691-002EN (100 Series PCH)
 */

#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <fcntl.h>
```

```
14  #include <pci/pci.h>
15  #include <sys/io.h>
16  #include <sys/mman.h>
17  #include <sys/errno.h>
18
19  /*
20   * GPIO register offsets in GPIO I/O space.
21   * Each chunk of 32 GPIOs is manipulated via its own USE_SELx, IO_SELx, and
22   * LVLx registers.  Logic in the read/write functions takes a register and
23   * an absolute bit number and determines the proper register offset and bit
24   * number in that register.  For example, to read the value of GPIO bit 50
25   * the code would access offset ichx_regs[2(=GPIO_LVL)][1(=50/32)],
26   * bit 18 (50%32).
27   */
28  enum GPIO_REG {
29    GPIO_USE_SEL = 0,
30    GPIO_IO_SEL,
31    GPIO_LVL,
32  };
33
34  static const uint8_t ichx_regs[4][3] = {
35    {0x00, 0x30, 0x40},     /* USE_SEL[1-3] offsets */
36    {0x04, 0x34, 0x44},     /* IO_SEL[1-3] offsets */
37    {0x0c, 0x38, 0x48},     /* LVL[1-3] offsets */
38  };
39
40  /*
41   * Generic PCI configuration space registers.
42   */
43  #define REG_VENDOR          0x00
44  #define REG_DEVICE          0x04
45
46  /*
47   * D31:F0 configuration space registers.
48   */
49  #define REG_ICH0_GPIOBASE   0x58
50  #define REG_ICH0_GC         0x5c
51
52  #define REG_ICH6_GPIOBASE   0x48
53  #define REG_ICH6_GC         0x4c
54
55  #define REG_ICHx_GC_EN      0x10
56  #define REG_ICHx_GC_GLE     0x01
57
58  /*
59   * D31:F1 configuration space registers.
60   */
61  #define REG_P2SB_BAR        0x10
62  #define REG_P2SB_BARH       0x14
63  #define REG_P2SB_CTRL       0xe0
64
65  #define REG_P2SB_CTRL_HIDE  0x0100
66
67  /*
68   * P2SB private registers.
69   */
70  #define P2SB_PORTID_SHIFT   16
71  #define P2SB_PORT_GPIO3     0xAC
72  #define P2SB_PORT_GPIO2     0xAD
73  #define P2SB_PORT_GPIO1     0xAE
74  #define P2SB_PORT_GPIO0     0xAF
75
76  /*
77   * GPIO sideband registers.
78   */
79  #define REG_PCH_GPIO_FAMBAR 0x8
80  #define REG_PCH_GPIO_PADBAR 0xc
81
82  #define REG_PCH_GPIO_PAD_OWN      0x20
```

```
#define REG_PCH_GPIO_HOSTSW_OWN   0xd0
#define REG_PCH_GPIO_GPI_IS       0x100
#define REG_PCH_GPIO_GPI_IE       0x120
#define REG_PCH_GPIO_GPE_STS      0x140
#define REG_PCH_GPIO_GPE_EN       0x160
#define REG_PCH_GPIO_SMI_STS      0x184
#define REG_PCH_GPIO_SMI_EN       0x1a4
#define REG_PCH_GPIO_NMI_STS      0x1c4
#define REG_PCH_GPIO_NMI_EN       0x1e4

#define REG_PCH_GPIO_DW0_PMODE    0x1600
#define REG_PCH_GPIO_DW0_RXDIS    0x0200
#define REG_PCH_GPIO_DW0_TXDIS    0x0100
#define REG_PCH_GPIO_DW0_RXSTATE  0x0002
#define REG_PCH_GPIO_DW0_TXSTATE  0x0001

#define REG_PCH_GPIO_DW1_TERM_NONE    0x0
#define REG_PCH_GPIO_DW1_TERM_5K_PD   0x2
#define REG_PCH_GPIO_DW1_TERM_20K_PD  0x4
#define REG_PCH_GPIO_DW1_TERM_5K_PU   0xa
#define REG_PCH_GPIO_DW1_TERM_20K_PU  0xc
#define REG_PCH_GPIO_DW1_TERM_NATIVE  0xf

/*
 * Helper functions.
 */

#define MSG(...) do { \
    fprintf(stderr, "[*] " __VA_ARGS__); fprintf(stderr, "\n"); \
  } while(0)
#define ERR(...) do { \
    fprintf(stderr, "[-] " __VA_ARGS__); fprintf(stderr, "\n"); \
    return 1; \
  } while(0)
#define DIE(...) do { *fatal = 1; ERR(__VA_ARGS__) } while(0)

struct pci_dev *pci_find_dev(struct pci_access *pci, uint8_t bus, uint8_t dev, ui
  for(struct pci_dev *it = pci->devices; it; it = it->next) {
    if(it->bus == bus && it->dev == dev && it->func == func) return it;
  }
  return NULL;
}

/*
 * Finally, our main logic!
 */

int try_ich(struct pci_access *pci,
            uint16_t reg_gpiobase, uint16_t reg_gc,
            const char *desc, int *fatal) {
  MSG("Checking for a %s system", desc);

  struct pci_dev *d31f0 = pci_find_dev(pci, 0, 31, 0);
  uint32_t gpiobase = pci_read_long(d31f0, reg_gpiobase);
  uint8_t gc = pci_read_byte(d31f0, reg_gc);
  MSG("GPIOBASE=%08x, GC=%02x", gpiobase, gc);

  if(gpiobase == 0xffffffff) {
    *fatal = 1;
    ERR("Cannot read GPIOBASE, are you running me as root?");
  } else if(gpiobase == 0) {
    ERR("GPIOBASE not implemented at %04x", reg_gpiobase);
  } else if(!(gpiobase & 1)) {
    *fatal = 1;
    ERR("GPIOBASE is not an I/O BAR");
  }

  if(!(gpiobase & 0xfffc)) {
    const uint32_t DEFAULT_GPIOBASE = 0x0480;
```

```
152
153       MSG("GPIOBASE is not configured, setting to %08x and hoping this works", DEFA
154       pci_write_long(d31f0, reg_gpiobase, DEFAULT_GPIOBASE);
155       gpiobase = pci_read_long(d31f0, reg_gpiobase);
156       if((gpiobase & 0xfffc) != DEFAULT_GPIOBASE) {
157         ERR("Cannot set GPIOBASE");
158       }
159     }
160
161     MSG("GPIO decoding is %s", (gc & REG_ICHx_GC_EN) ? "enabled" : "disabled");
162     MSG("GPIO lockdown is %s", (gc & REG_ICHx_GC_GLE) ? "enabled" : "disabled");
163
164     if(!(gc & REG_ICHx_GC_EN)) {
165       MSG("Enabling GPIO decoding");
166       pci_write_byte(d31f0, reg_gc, gc | REG_ICHx_GC_EN);
167       gc = pci_read_byte(d31f0, reg_gc);
168       if(!(gc & REG_ICHx_GC_EN)) {
169         ERR("Cannot enable GPIO decoding");
170       }
171     }
172
173     gpiobase &= 0xfffc;
174     if(ioperm(gpiobase, 128, 1) == -1) {
175       ERR("Cannot access I/O ports %04x:%04x", gpiobase, gpiobase + 128);
176     }
177
178     for(int n = 1; n < 3; n++) {
179       MSG("USE_SEL%d=%08x", n, inl(gpiobase + ichx_regs[GPIO_USE_SEL][n]));
180       MSG("IO_SEL%d=%08x", n, inl(gpiobase + ichx_regs[GPIO_IO_SEL][n]));
181       MSG("LVL%d=%08x", n, inl(gpiobase + ichx_regs[GPIO_LVL][n]));
182     }
183
184     return 0;
185   }
186
187   int get_pch_sbreg_addr(struct pci_access *pci, pciaddr_t *sbreg_addr) {
188     MSG("Checking for a Series 10 PCH system");
189
190     struct pci_dev *d31f1 = pci_get_dev(pci, 0, 0, 31, 1);
191     pci_fill_info(d31f1, PCI_FILL_IDENT);
192     if(d31f1->vendor_id == 0xffff) {
193       MSG("Cannot find D31:F1, assuming it is hidden by firmware");
194
195       uint32_t p2sb_ctrl = pci_read_long(d31f1, REG_P2SB_CTRL);
196       MSG("P2SB_CTRL=%02x", p2sb_ctrl);
197       if(!(p2sb_ctrl & REG_P2SB_CTRL_HIDE)) {
198         ERR("D31:F1 is hidden but P2SB_E1 is not 0xff, bailing out");
199       }
200
201       MSG("Unhiding P2SB");
202       pci_write_long(d31f1, REG_P2SB_CTRL, p2sb_ctrl & ~REG_P2SB_CTRL_HIDE);
203
204       p2sb_ctrl = pci_read_long(d31f1, REG_P2SB_CTRL);
205       MSG("P2SB_CTRL=%02x", p2sb_ctrl);
206       if(p2sb_ctrl & REG_P2SB_CTRL_HIDE) {
207         ERR("Cannot unhide PS2B");
208       }
209
210       pci_fill_info(d31f1, PCI_FILL_RESCAN | PCI_FILL_IDENT);
211       if(d31f1->vendor_id == 0xffff) {
212         ERR("P2SB unhidden but does not enumerate, bailing out");
213       }
214     }
215
216     pci_fill_info(d31f1, PCI_FILL_RESCAN | PCI_FILL_IDENT | PCI_FILL_BASES);
217     if(d31f1->vendor_id != 0x8086) {
218       ERR("Vendor of D31:F1 is not Intel");
219     } else if((uint32_t)d31f1->base_addr[0] == 0xffffffff) {
220       ERR("SBREG_BAR is not implemented in D31:F1");
```

```
  }

  *sbreg_addr = d31f1->base_addr[0] &~ 0xf;
  MSG("SBREG_ADDR=%08lx", *sbreg_addr);

  MSG("Hiding P2SB again");
  uint32_t p2sb_ctrl = pci_read_long(d31f1, REG_P2SB_CTRL);
  pci_write_long(d31f1, REG_P2SB_CTRL, p2sb_ctrl | REG_P2SB_CTRL_HIDE);

  pci_fill_info(d31f1, PCI_FILL_RESCAN | PCI_FILL_IDENT);
  if(d31f1->vendor_id != 0xffff) {
    ERR("Cannot hide P2SB");
  }

  return 0;
}

uint32_t sideband_read(void *sbmap, uint8_t port, uint16_t reg) {
  uintptr_t addr = ((uintptr_t)sbmap + (port << P2SB_PORTID_SHIFT) + reg);
  return *((volatile uint32_t *)addr);
}

int try_pch(struct pci_access *pci) {
  pciaddr_t sbreg_addr;
  if(get_pch_sbreg_addr(pci, &sbreg_addr)) {
    MSG("Re-enumerating PCI devices will probably crash the system");
    ERR("Probing Series 100 PCH failed");
  }

  int memfd = open("/dev/mem", O_RDWR);
  if(memfd == -1) {
    ERR("Cannot open /dev/mem");
  }

  void *sbmap = mmap((void*)sbreg_addr, 1<<24, PROT_READ|PROT_WRITE, MAP_SHARED,
                     memfd, sbreg_addr);
  if(sbmap == MAP_FAILED) {
    if(errno == EPERM) {
      MSG("Is your kernel configured with CONFIG_DEVMEM_STRICT=n?");
      MSG("Try rebooting and specifying iomem=relaxed on kernel command line.");
    }
    ERR("Cannot map SBREG");
  }

  close(memfd);

  for(unsigned port = 0; port < 4; port++) {
    uint16_t port_id = P2SB_PORT_GPIO0 - port;
    uint32_t padbar = sideband_read(sbmap, port_id, REG_PCH_GPIO_PADBAR);
    MSG("GPIO%d_PADBAR=%x", port, padbar);

    for(unsigned pad = 0; pad < 32; pad++) {
      uint32_t dw0 = sideband_read(sbmap, port_id, padbar + pad * 8);
      uint32_t dw1 = sideband_read(sbmap, port_id, padbar + pad * 8 + 4);
      if(dw1 == 0) {
        // Not documented as such, but appears to be a reliable last pad marker.
        break;
      }

      const char *state = "???", *rxstate = "", *txstate = "";
      if((dw0 & REG_PCH_GPIO_DW0_PMODE) != 0) {
        state = "Native";
      } else if((dw0 & REG_PCH_GPIO_DW0_TXDIS) != 0 &&
                (dw0 & REG_PCH_GPIO_DW0_RXDIS) != 0) {
        state = "Off";
      } else {
        state = "GPIO";
        if((dw0 & REG_PCH_GPIO_DW0_RXDIS) == 0) {
          if((dw0 & REG_PCH_GPIO_DW0_RXSTATE) != 0) {
```

```
            rxstate = " InHigh";
          } else {
            rxstate = " InLow";
          }
        }

        if((dw0 & REG_PCH_GPIO_DW0_TXDIS) == 0) {
          if((dw0 & REG_PCH_GPIO_DW0_TXSTATE) != 0) {
            txstate = " OutHigh";
          } else {
            txstate = " OutLow";
          }
        }
      }

      const char *pull = "???";
      switch(dw1 >> 10) {
        case REG_PCH_GPIO_DW1_TERM_NONE:   pull = "None";   break;
        case REG_PCH_GPIO_DW1_TERM_5K_PD:  pull = "Dn5k";   break;
        case REG_PCH_GPIO_DW1_TERM_20K_PD: pull = "Dn20k";  break;
        case REG_PCH_GPIO_DW1_TERM_5K_PU:  pull = "Up5k";   break;
        case REG_PCH_GPIO_DW1_TERM_20K_PU: pull = "Up20k";  break;
        case REG_PCH_GPIO_DW1_TERM_NATIVE: pull = "Native"; break;
      }

      printf("[+] GPIO%d_PAD%d: DW0=%08x DW1=%08x State=%s%s%s Pull=%s\n",
             port, pad, dw0, dw1, state, rxstate, txstate, pull);
    }
  }

  return 0;
}

int create_pci(int method, struct pci_access **pci_out)  {
  struct pci_access *pci = pci_alloc();
  pci->method = method;
  pci_init(pci);
  pci_scan_bus(pci);

  struct pci_dev *d31f0 = pci_find_dev(pci, 0, 31, 0);
  if(!d31f0) {
    ERR("Cannot find D31:F0");
  }

  pci_fill_info(d31f0, PCI_FILL_IDENT | PCI_FILL_BASES);
  if(d31f0->vendor_id != 0x8086) {
    ERR("Vendor of D31:F0 is not Intel");
  }

  *pci_out = pci;
  return 0;
}

int main() {
  struct pci_access *pci;
  if(create_pci(PCI_ACCESS_AUTO, &pci)) {
    MSG("Is this an Intel platform?");
    return 1;
  }

  int fatal = 0;
  if(try_ich(pci, REG_ICH0_GPIOBASE, REG_ICH0_GC,
             "ICH0..ICH5", &fatal) && fatal) {
    return 1;
  } else if(try_ich(pci, REG_ICH6_GPIOBASE, REG_ICH6_GC,
                    "ICH6..ICH9 or Series 5..9 PCH", &fatal) && fatal) {
    return 1;
  } else {
    pci_cleanup(pci);
```

```
359
360       // Letting Linux discover P2SB (and reassign its BAR) hangs the system,
361       // so we need to enumerate the device bypassing it.
362       if(create_pci(PCI_ACCESS_I386_TYPE1, &pci)) {
363         return 1;
364       }
365
366       if(try_pch(pci)) {
367         return 1;
368       }
369     }
370
371     printf("[+] Done\n");
372     return 0;
373   }
```

Want to discuss this note? Drop me a letter.

Drafts