RouterBOARD 200 Series

Software Developer's Kit for Linux

Rev. B (23-Apr-2004)

Table of Contents

Controlling LEDs and LCD

IOCS0 Port

IOSC0 control port is used to control LEDs and LCD display output. This is write-only register, and its 16 lower bits are used as follows:

bit	LCD	LED
0 .. 7	data	
8	INITX	
9	SLINX	
10	AFDX	
11	backlit	
12		LED1
13		LED2
14		LED3
15		LED4

IOCS0 port number can be read from PCI Configuration Space Function 0 (F0) at index 74h as a 16 bit value (see [GEODE] 5.3.1 pg. 151 and pg. 176 Table 5-29):

```
#define CAR 0xcf8 /* PCI config address register */
#define CDR 0xcfc /* PCI config data register */
static unsigned getIocs0Port(void) {
    static unsigned ret = 0;
    if (!ret) {
      outl(0x80009074, CAR);
      ret = inw(CDR);
    }
    return ret;
}
```

LED Control Example

To control LED1 .. LED4, you may use the following function:

```
#define LED1 0x1000
#define LED2 0x2000
#define LED3 0x4000
#define LED4 0x8000
/* char *s - string of LED numbers to lit, for example "134" */
static int leds(char *s) {
    unsigned mask = 0;
    if (s) while (*s) switch (*(s++)) {
    default: return 1;
    case '1': mask |= LED1; break;
    case '2': mask |= LED2; break;
    case '3': mask |= LED3; break;
    case '4': mask |= LED4; break;
    }
    outw(mask, getIocs0Port());
    return 0;
}
```

LCD Display

This chapter explains programming LCD port using LCD displays based on Hitachi HD44780U Dot Matrix LCD Controller (sold by MikroTik).

LCD interface works as a parallel port, so timing and control signals are important. Many functions in this example use udelay function that runs an idle cycle for the given number of microseconds:

```
/* unsigned usec - duration of the idle cycle in microseconds */
static void udelay(unsigned usec) {
    struct timespec t;
    t.tv_sec = 0;
    t.tv_nsec = usec * 1000;
    nanosleep(&t, 0);
}
```

The port's control signals are defined as follows:

```
#define LCD_INITX 0x100
#define LCD_SLINX 0x200
#define LCD_AFDX 0x400
#define LCD_BACKLIGHT 0x800
```

Note also that since IOCS0 port cannot be used to read data from the attached device, Data Read/write (LCD_SLINX) is never used.

Sending Commands

To send a command to a LCD, you should write the command's code to the IOCS0 port and hold it for about 1µs while Enable Signal (INITX) is active and for about 33µs while INITX is not active [HITACHI]:

```
/* unsigned char d - command to write to the LCD */
static void writeLCDcmd(unsigned char d) {
    outw(LCD_INITX | LCD_BACKLIGHT | d, getIocs0Port());
    udelay(1);
    outw(LCD_BACKLIGHT | d, getIocs0Port());
    udelay(33);
}
```

We will use the following commands (see [HITACHI pg. 191 Table 6] for summary of all commands):

Code	Description
0x39	Set 8 bit data length and 2 display lines
0x0C	Turn display on, cursor & blinking cursor off
0x01	Clear display (note an additional delay of 1607µs after this command)

Positioning on the LCD screen is done by the following command:

```
writeLCDcmd(0x80 + column + row * 0x20);
```

Sending Data

The Hitachi LCD controller supports standard ASCII symbol codes. Sending symbols is similar to sending commands, but Register Select (AFDX) should be active during the writing cycle [HITACHI]:

```
/* unsigned char d - data to write to the LCD */
static void writeLCDdata(unsigned char d) {
    outw(d | LCD_BACKLIGHT | LCD_AFDX | LCD_INITX, getIocs0Port());
    udelay(1);
```

```
    outw(d | LCD_BACKLIGHT | LCD_AFDX, getIocs0Port());
    udelay(33);
}
```

LCD Example

Now we can use the described procedures to write the word "TEST" startinf from the 1st row, 7th column:

```
static int lcd(void) {
    /*
    ** Initialize LCD display controller
    ** First, set 8 bit data length, 2 display lines
    */
    writeLCDcmd(0x39);
    /*
    ** Display on, cursor & blinking cursor off
    */
    writeLCDcmd(0x0C);
    /*
    ** Clear display
    */
    writeLCDcmd(0x01);
    udelay(1607);
    /*
    ** Go to row 1, column 7
    */
    writeLCDcmd(0x80 + 7 + 1 * 0x20);
    writeLCDdata('T');
    writeLCDdata('E');
    writeLCDdata('S');
    writeLCDdata('T');
    return 0;
}
```

Controlling GPIO

F0BAR0 Port

F0 Base Address Register 0 (F0BAR0) is used as base port to configure SC1100 GPIOs. F0BAR0 port number can be read from PCI Configuration Space Function 0 (F0) at index 10h as a 32 bit value aligned to 64 byte boundary (see [GEODE] table 5-29, pg. 168):

```
#define CAR 0xcf8 /* PCI config address register */
#define CDR 0xcfc /* PCI config data register */
static unsigned getF0Bar0(void) {
    static unsigned ret = 0;
    if (!ret) {
      outl(0x80009010, CAR);
      ret = inl(CDR) & ~0x3f;
    }
    return ret;
}
```

GPIO Configuration

Before using GPIOs they must be configured ([GEODE] table 5-30, pg. 197-199). There are some configuration options that are explained in [GEODE], which will not be used in the given examples. The following function may be used to write the GPIO configuration:

```
/* unsigned gpio - the number of the GPIO port to configure */
/* unsigned cfg - the configuration of the GPIO port */
static void setGpioCfg(unsigned gpio, unsigned cfg) {
    /*
    ** Select GPIO for configuration
    */
    outl(gpio, getF0Bar0() + 0x20);
    /*
    ** Write GPIO configuration
    */
    outl(cfg, getF0Bar0() + 0x24);
}
```

GPIO Input/Output

GPIOs are separated in two banks, 32 in each. Once a GPIO is configured, it can be used in read/write operations. The GPIOs maintain two states simultaneously: one for input and one for output ([GEODE] table 5-30, pg. 196-197).

GPIO outputs are controlled by setting bits in GPDO0 and GPDO1 registers. They are read/write, so it is possible not only to trigger the output state, but also to check the current output state:

```
/* unsigned gpio - the number of the GPIO port to write to */
/* unsigned enable - the value to write to the GPIO port */
static unsigned setGpio(unsigned gpio, unsigned enable) {
    unsigned dataPort;
    unsigned value;
    if (gpio & 0x20) dataPort = getF0Bar0() + 0x10;
    else dataPort = getF0Bar0() + 0x00;
    value = inl(dataPort);
    if (enable) value |= 1 << (gpio & 0x1f);
    else value &= ~(1 << (gpio & 0x1f));
    outl(value, dataPort);
}

/* unsigned gpio - the number of the GPIO port to read from */
static unsigned getCurrentGpio(unsigned gpio) {
    unsigned dataPort;
    if (gpio & 0x20) dataPort = getF0Bar0() + 0x10;
    else dataPort = getF0Bar0() + 0x00;
    return !!(inl(dataPort) & (1 << (gpio & 0x1f)));
}
```

GPIO input states are read from GPDI0 and GPDI1 registers:

```
/* unsigned gpio - the number of the GPIO port to read from */
static unsigned getGpio(unsigned gpio) {
    unsigned dataPort;
    if (gpio & 0x20) dataPort = getF0Bar0() + 0x14;
    else dataPort = getF0Bar0() + 0x04;
    return !!(inl(dataPort) & (1 << (gpio & 0x1f)));
}
```

GPIO Example

Now we can use the described procedures to write a function that can set output state of a GPIO and get status of the GPIOs:

```
/* char *s - text string of a statement in form of number=state:
** number -  the number of the GPIO from 0 to 63;
** state - either 0 or 1
```

```
**
** If char *s=="", displays the input/output status of all the GPIOs
*/
static int gpio(char *s) {
    if (s && *s) {
      unsigned num;
      unsigned val;
      char dummy;
      if (2 != sscanf(s, "%d=%d%c", &num, &val, &dummy)) return 1;
      if (num >= 63) return usage();
      if (val >= 2) return usage();
      setGpioCfg(num, 1);
      setGpio(num, val);
    }
    else {
      unsigned num;
      for (num = 0; num != 64; ++num) {
          setGpioCfg(num, 0x0);
      }
      printf("             INPUT/OUTPUT\n");
      printf("GPIO  0..15: ");
      for (num = 0; num != 16; ++num) {
          printf("%d/%d ", getGpio(num), getCurrentGpio(num));
      }
      printf("\nGPIO 16..31: ");
      for (num = 16; num != 32; ++num) {
          printf("%d/%d ", getGpio(num), getCurrentGpio(num));
      }
      printf("\nGPIO 32..47: ");
      for (num = 32; num != 48; ++num) {
          printf("%d/%d ", getGpio(num), getCurrentGpio(num));
      }
      printf("\nGPIO 48..63: ");
      for (num = 48; num != 64; ++num) {
          printf("%d/%d ", getGpio(num), getCurrentGpio(num));
      }
      printf("\n");
    }
    return 0;
}
```

Watchdog

The SC1100 CPU has an embedded hardware watchdog. A driver is required to control the watchdog. The driver called **scx200_wdt** is in the recent Linux kernels (the one from 2.4.23 stock kernel was tested), but you will also need a small patch (distributed with the SDK) for it to recognize the watchdog (as the driver is designed for SCx200 chips, not for SC1100).

To use the watchdog, you need to create **/dev/watchdog** character (unbuffered) special file with major number of **10** and minor number of **130** (see files Documentation/watchdog.txt and Documentation/watchdog-api.txt in the kernel source). In brief, the /dev/watchdog device works this way:

- When **/dev/watchdog** is opened, the watchdog timer becomes active
- The watchdog timer is restarted by writing to this file (for example, software or watchdog daemon writes to the file every 10 seconds)
- The default watchdog timeout is 30 seconds, when watchdog timeouts second time, system is rebooted.

- Before closing the file, you must write character 'V' to the file, otherwise watchdog timer will remain active

Watchdog Example

This example polls the watchdog every 10 seconds, but then does not close the file (it means that after about 1 minute after the function is interrupted, the watchdog will reboot the system):

```
static int wdog(void) {
    FILE *f;
    f = fopen("/dev/watchdog", "w");
    if (!f) {
      printf("cannot open watchdog device\n");
      exit(1);
    }
    printf("watchdog is active now\n"
         "when you terminate this program, watchdog will reboot system\n"
         "after about one 1 minute\n");
    for (;;) {
        if (!fwrite("", 1, 1, f)) break;
        fflush(f);
        sleep (10);
    }
    printf("failed to write to watchdog\n");
    exit(1);
}
```

ACCESS.bus (I²C)

The I²C bus (it is called ACCESS.bus in SC1100 chip) is a versatile two-wire synchronous serial bus, which is used in RouterBOARD for Hardware monitoring (LM87 sensor chip connected to the second I²C bus) and also available in GPIO header (J19 GPIO header pins 3 and 4 are connected to the first I²C bus).

The stock 2.4 kernels (only 2.4.23 was tested) support the SC1100 chip's I²C subsystem using **scx200_acb** driver (as in case of watchdog device, a special patch provided with this SDK is needed), but it is proved not to be very stable on high load. National Semiconductor has designed an another driver called **i2c-nscacb**, which is also distributed with this SDK (note that we have changed that driver a little bit to improve performance). Whichever driver you will be using, the I²C subsystem core modules **i2c-core** and **i2c-proc** should be loaded first.

Hardware monitor

The RouterBOARD uses LM87 chip as a hardware monitor. The LM87 provides the following measurements:

- CPU core, +3.3V, +5V, +12V voltages
- LM87 chip, CPU area, board temperatures
- Fan rotation speed

To use LM87 chip, the I²C subsystem must be installed. The drivers for LM87 chip is included in 2.6 kernels, but for 2.4 kernels, you need to use LM87 drivers from LM-sensors homepage (www.lm-sensors.nu) or the one provided with the SDK (tested with 2.4.23 stock kernel).

If LM87 module is loaded successfully, the sensor chip's showings become available in **/proc/sys/dev/sensors/lm87-i2c-1-2e/** directory (file format is given in parentheses):

Filename	Description
alarms	bit 12 is active when J16 is open
analog_out	not used
fan1	J11 fan rotation speed in rpm (min current)

Filename	Description
fan2	not used
fan_div	FAN speed divisor: 1, 2, 3 or 4 (fan1 fan2)
in1	CPU core voltage (min max current)
in2	+3.3V voltage (min max current)
in3	+5V voltage (min max current)
in4	+12V voltage (min max current)
temp1	LM87 chip temperature (max min current)
temp2	CPU area temperature (max min current)
temp3	board area temperature (max min current)
vid	not used
vrm	not used

Chassis Intrusion Jumper

The Chassis Intrusion jumper J16 is connected to the LM87 chip.

```
/* char *name - file name to read */
/* int idx - word number in the file counting from 0 */
static int readLM87(char *name, int idx) {
    char n[128];
    FILE *f;
    float b;
    snprintf(n, sizeof(n), "/proc/sys/dev/sensors/lm87-i2c-0-2e/%s", name);
    f = fopen(n, "r");
    for (;idx>=0;idx--) if (!f || !fscanf(f,"%f",&b)) {
        printf("failed to read from lm87 chip\n");
        if (f) fclose(f);
        exit(1);
    }
    fclose(f);
    return (b+0.005)*100;
}

static int cin(void) {
    int alarms;
    alarms = readLM87("alarms", 0);
    printf("Chassis Intrusion: J16 is ");
    if ((alarms/100) & 0x1000) printf("open\n");
    else printf("closed\n");
}
```

Literature

[GEODE] - Geode SC1100 Information Appliance On a Chip
(http://www.national.com/ds/SC/SC1100.pdf)

[HITACHI] - HD44780U (LCD-II) Dot Matrix Liquid Crystal Display Controller/Driver
(http://home.wizard.org/auction_support/hd44780u.pdf)

[LM87] - LM87 Serial Interface System Hardware Monitor with Remote Diode Temperature Sensing
(http://www.national.com/ds/LM/LM87.pdf)