The Linux gpio-104 device-driver

[Document number: ??, revision 1]

Martin Shepherd
California Institute of Technology

November 2, 2005

This page intentionally left blank.

Abstract

The gpio-104 board is a PC/104 general-purpose I/O board. It provides 24-bits of digital I/O, can digitize one-at-a-time of 8 input voltages, with 12-bit resolution, and generate 4 separate analog output voltages, each with 12-bit resolution.

The CCB uses 16 of the digital outputs to control front-panel LEDs, 1 of the digital outputs to force a reload of the CCB firmware, and 3 digital outputs to select which CCB PCB is currently to be sampled by the subset of the analog and digital inputs that are specific to a given CCB PCB. The remaining analog and digital inputs monitor voltages and status bits that are global to the system.

The 4 voltage outputs of the gpio-104 board aren't currently used. However the driver provides commands for setting their values.

Contents

Chapter 1

Design considerations

1.1 The readout speed

The gpio-104 board relies on ISA-bus transactions to transfer data. This is very slow, and because transfering a single byte using ISA-bus transactions, is achieved with a single unbreakable inb or outb CPU instruction, each byte that is transfered to or from the gpio-104 board stalls the CPU for a whole 0.5μs ISA-bus cycle. If the driver transfered many such bytes in sequence, then the computer would appear to hang until the sequence finished. Given that this board is used by the CCB for low priority monitoring, that shouldn't get in the way of anything else, the driver thus had to be written to explicitly throttle-down the frequency at which bytes are transfered. Each single-byte transaction still unavoidably takes 0.5μs, but the processor is explicitly given up for 1ms between certain of these transactions, to allow other processes to make headway. These 1ms windows cover the period during which an ADC is digitizing a new sample. According to the data-sheet, this delay only needs to be 9μs, so 1ms is much longer than needed. However 1ms is the shortest delay that the Linux kernel offers that isn't implemented using a non-interruptible busy-waiting loop.

The bottom-line is that if the minimal 9μs conversion delays were implemented, then reading out monitoring information from all of the CCB PCBs, would theoretically take about 0.4ms, whereas replacing the 9μs unbreakable delays with 1ms breakable delays, increases this to 40.4ms. Given that the monitoring period of the CCB manager will normally be around 1s, and the mimum period invisaged is 100ms, this should be more than adequate.

1.2 The versatility of the public device-driver interface

There are two competing goals for a device-driver. On the one hand one ideally wants to give the calling process the flexibility to do anything that the board can do, which means exposing

the lowest level of functionality, at the device-driver interface. On the other hand one wants to hide the details of the hardware from the user-level code, to ensure optimal efficiency, prevent the user-level code from doing dangerous things, and to make it easy to accomodate a different board at a future date, without having to change the device-driver interface. The final driver always ends up being a compromise between these aims. For a general purpose device-driver to a general purpose I/O board, the compromise is usually biased towards low-level versatility. This, however has serious drawbacks in terms of the complexity of the device-driver interface, and in a loss of efficiency, due to the need for excessive numbers of system calls to get anything done. In the case of the CCB, on the other hand, there are specific goals for how this board is to be used, thus making a general-purpose interface hard to justify, and it is important that it's usage not impact higher-priority processes on the computer. Thus a higher level interface has been chosen, which is designed around the CCB usage of the board.

In particular, the front-panel LEDs can be turned on and off individually, or as a group, by name, and monitoring information is automatically collected periodically behind the scenes, by the driver, which then notifies the calling process whenever a new set of information has been collected, and then allows all of this information to be collected in a single read operation. This avoids the CCB server having to go through all of the circumlocutions needed to address, request, delay and readout each sample.

Chapter 2

Loading the device-driver module

The device driver is implemented as a loadable module. Since this module doesn't depend on any other loadable driver modules, it can be loaded with a simple insmod command. If the module were in the current directory, then the following command-line would be a typical way to do this.

```
/sbin/insmod ./ccb_gpio_driver.ko major=124 base_addr=0x300 readout_period=100
```

The optional parameters shown above, are defined as follows.

- major

  This is a unique major device number to assign to the driver (see the file devices.txt in the kernel-distribution's Documentation directory, for a list). If this number isn't specified, then the driver will ask the kernel for a suitable number, and the resulting number can subsequently be determined by looking in the file /proc/devices.

  The specified, or kernel-assigned number should be used to create the device-driver file in /dev, using a command line like the following.

  mknod /dev/ccbgpio c $major 0

  where $major is the selected major number.

- base_addr

  This is the base address of the GPIO-104 card, which is set on the card via jumpers. The factory-default value is 0x300.

- readout_period

  This parameter specifies how often the driver reads out monitoring information from the CCB PCBs, and is interpretted as an integer number of milliseconds. Its default

value is 100ms, and a minimum value of 50ms is quietly enforced. Note that readouts only occur when a process has the device-driver's device-file open. The readout period can subsequently be changed by such a process.

Chapter 3

The system call interface

Once the driver has been loaded, the server program connects to it by opening its device file, as shown below.

```
#include "ccb_gpio_driver"
...
int fd = open(CCB_GPIO_DEV_FILE, O_RDWR);
```

Thereafter the read(), ioctl(), fcntl() and select() system calls can be used to control the front-panel LEDs, reload the CCB firmware, and monitor the health of the CCB hardware.

3.1 fcntl()

This system call can be used to configure the file-descriptor of the device file, for non-blocking I/O, as shown below.

```
int flags = fcntl(fd, F_GETFL, 0);
if(flags < 0 || fcntl(fd, F_SETFL, flags | O_NONBLOCK) < 0)
  ...error...
```

While it does no harm to explicitly switch into non-blocking I/O mode, it isn't necessary if the calling process only invokes read() when the select() system call indicates that there are data to be read.

3.2 select()

This system call is used by the server to determine when a new set of monitoring data is ready to be read from the driver. The following example shows select() being used to wait for the device file to become readable. A more realistic example would have it waiting for multiple files to become readable and/or writeable.

```
  fd_set read_fds;       /* A set of readable file descriptors */
  int nready;            /* The number of files that are ready for I/O */
/*
 * Empty the set of readable file descriptors.
 */
  FD_ZERO(&read_fds);
/*
 * Add the driver's file descrptor to the set that are to be
 * watched for readability.
 */
  FD_SET(fd, &read_fds);
/*
 * Wait for the driver to indicate that fd is readable.
 */
  nread = select(fd+1, &read_fds, NULL, NULL, NULL);
  if(nread < 0)
    ... error ...
/*
 * Was fd marked as readable?
 */
  if(FD_ISSET(fd, &read_fds)) {
    ...read from fd...
  };
```

3.3 read()

Whenever select() indicates that the device-file is ready to be read, this indicates that there is a new set of monitor data available to be read. If this isn't read before the next set of monitor data becomes available, then the older copy is simply discarded. Each call to read() reads either a whole set of monitoring data, if data were avaiable, or nothing if either no data were available in non-blocking mode, or the read was interrupted by a signal, while waiting for a new data-set to arrive. If the caller to read() requests fewer bytes than the number that constitute a data-set, then the remaining bytes of that data-set are simply discarded.

Note that it isn't possible for a call to read() to be interrupted in the middle of delivering a data-set. So there is no need to be able to restart reading from the middle of a data-set. In other words, calls to read() are atomic, in that they either return/consume a full monitoring data-set, or they return nothing at all.

Each successful call to read() returns a data-set encapsulated in a CCBMonitorData structure. This is defined as follows:

```
typedef struct {
  unsigned short fan12v;     /* The voltage of the 12V fan PSU */
  unsigned short a8v;        /* The voltage of the analog 8V PSU */
  unsigned short d5v;        /* The voltage of the digital 5V PSU */
  unsigned short cnf_done;   /* True if the FPGAs have all finished */
                             /* loading their firmware. */
  unsigned short high_temp;  /* True if there is a high-temperature */
                             /* condition. */
  unsigned short ccb_id;     /* The 2-bit ID of the CCB hardware */
  CCBFpgaMonitorData fpga[CCB_NUM_SLAVE+1];
                             /* Data of the master-FPGA CCB, */
                             /* followed by those of the 4 slave */
                             /* FPGAs. */
} CCBMonitorData;
```

In this structure, the 2-bit ccb_id member identifies which CCB is attached, and is used at boot-time to decide which IP address to assign to the CCB computer.

The elements of the embedded fpga[] array are structures of type CCBFpgaMonitorData, which is defined as follows:

```
typedef struct {
  unsigned short d1_2v;   /* The voltage of the digital 1.2V PSU */
  unsigned short d2_5v;   /* The voltage of the digital 2.5V PSU */
  unsigned short d3_3v;   /* The voltage of the digital 3.3V PSU */
  unsigned short a5v;     /* The voltage of the analog 5V PSU */
  unsigned short hb;      /* The mean voltage of the heartbeat */
                          /* signal. */
  unsigned short cnf_error; /* True if an error occurred while */
                            /* loading the FPGA firmware. */
  unsigned short cnf_done;  /* True if the FPGA firmware */
                            /* loading-process has completed. */
} CCBFpgaMonitorData;
```

So, an example of how to read a monitor-data packet is as follows.

```
int fd;              /* The file-descriptor of the opened */
                     /* /dev/ccbgpio device, left configured */
                     /* for blocking I/O. */
ssize_t nread;       /* The number of bytes read */
CCBMonitorData md;   /* A packet of monitoring data */
...
nread = read(fd, &md, sizeof(md));
if(nread < 0) {
  fprintf(stderr, "Error reading %s: %s\n",
          CCB_GPIO_DEV_FILE, strerror(errno));
} else if(nread < sizeof(md)) {
  fprintf(stderr, "Unexpected short read from %s.\n",
          CCB_GPIO_DEV_FILE);
} else {
  printf("CCB ID = %d\n", md.ccb_id);
};
```

In the example, since the file-descriptor has not been configured for non-blocking I/O, the call to read() doesn't return until either data becomes available, or a signal arrives, or an error occurs. Therefore we don't have to separately check for nread being zero, which, in non-blocking mode, would indicate that no data were currently available.

3.3.1 Scaling the returned values to physical units

The returned values that represent analog quantities, are returned as unsigned 12-bit ADC counts. To facilitate converting these into floating-point values to voltages, the ccb_gpio_driver.h header-file defines one scaling macro per item. In the following, where md is a CCBMonitorData variable, and n is the index of an FPGA board, examples are shown of how to use each of the scaling constants.

- The fan voltage:

  float v = md.fan12v * CCB_GPIO_SYS_FAN12V_SCALE

- The shared analog 8V PSU voltage:

  float v = md.a8v * CCB_GPIO_SYS_A8V_SCALE

- The shared digital 5V PSU voltage:

  float v = md.d5v * CCB_GPIO_SYS_D5V_SCALE

- An FPGA's digital 1.2V PSU voltage:

  float v = md.fpga[n].d1_2v * CCB_GPIO_SYS_D1_2V_SCALE

- An FPGA's digital 2.5V PSU voltage:
  float v = md.fpga[n].d2_5v * CCB_GPIO_SYS_D2_5V_SCALE
- An FPGA's digital 3.3V PSU voltage:
  float v = md.fpga[n].d3_3v * CCB_GPIO_SYS_D3_3V_SCALE
- An FPGA's analog 5V PSU voltage:
  float v = md.fpga[n].a5v * CCB_GPIO_SYS_A5V_SCALE
- An FPGA's mean heartbeat voltage:
  float v = md.fpga[n].hb * CCB_GPIO_SYS_HB_SCALE

3.3.2 Determining whether an FPGA board is present

All of the voltages that are monitored in the CCB are presented to the ADC channels in a scaled form, such that their nominal voltages fall at the mid point of the ADC input range. A power-supply voltage should thus never be close to zero ADC units unless the power-supply is broken or the corresponding CCB board isn't present . If all of the power-supply voltages of a given FPGA-board are close to zero, then this is used by the ccbserver and ccb_monitor_status programs as convincing evidence that said FPGA board is not present in the system. Since an unconnected ADC input won't generally yield exactly zero ADC counts, due to noise and offsets, an appropriate limiting value of CCB_GPIO_ABSENCE_LIMIT is defined. This is expressed in ADC counts, and is currently set to 10% of the nomimal mid-point voltage. In practice 10% is very conservative, since the input noise for an unconnected input is closer to 0.5% of the nomimal mid-point value.

3.3.3 Front-panel LED limits

The ccbserver and ccb_monitor_status programs set LEDs on the front panel of the CCB, according whether the returned analog values are within reasonable limits. To ensure that both programs use the same limits, the limits are parameterized by macros in ccb_gpio_driver.h. The allowable ranges of values that are specified by these macros are defined as follows.

- $\texttt{CCB\_GPIO\_SYS\_FAN12V\_MIN} \leq \texttt{md.fan12v} \leq \texttt{CCB\_GPIO\_SYS\_FAN12V\_MAX}$
- $\texttt{CCB\_GPIO\_SYS\_A8V\_MIN} \leq \texttt{md.a8v} \leq \texttt{CCB\_GPIO\_SYS\_A8V\_MAX}$
- $\texttt{CCB\_GPIO\_SYS\_D5V\_MIN} \leq \texttt{md.d5v} \leq \texttt{CCB\_GPIO\_SYS\_D5V\_MAX}$
- $\texttt{CCB\_GPIO\_FPGA\_D1\_2V\_MIN} \leq \texttt{md.fpga[n].d1\_2v} \leq \texttt{CCB\_GPIO\_FPGA\_D1\_2V\_MAX}$

- CCB_GPIO_FPGA_D2_5V_MIN $\leq$ md.fpga[n].d2_5v $\leq$ CCB_GPIO_FPGA_D2_5V_MAX
- CCB_GPIO_FPGA_D3_3V_MIN $\leq$ md.fpga[n].d3_3v $\leq$ CCB_GPIO_FPGA_D3_3V_MAX
- CCB_GPIO_FPGA_A5V_MIN $\leq$ md.fpga[n].a5v $\leq$ CCB_GPIO_FPGA_A5V_MAX
- CCB_GPIO_FPGA_HB_MIN $\leq$ md.fpga[n].hb $\leq$ CCB_GPIO_FPGA_HB_MAX

3.4 ioctl()

The ioctl() system-call is used to send commands to the driver to set the states of the front-panel LEDs, reload the CCB firmware, or configure the driver.

3.4.1 CCB_GPIO_IOCTL_LIGHT_LEDS

Use the following ioctl() call to turn on a specified set of the front-panel LEDs, without affecting the states of any of the other LEDs. The (int) argument of this call is a bit-mask union of CCBPanelLed enumerators, denoting which LEDs should be turned on. For example, to turn on the cal-mode and dump-mode LEDs, you would call ioctl() as follows:

```
#include "ccb_gpio_driver"
...
if(ioctl(fd, CCB_GPIO_IOCTL_LIGHT_LEDS,
             CCB_CAL_MODE_LED | CCB_DUMP_MODE_LED) < 0)
  perror("ioctl(CCB_IOCTL_LIGHT_LEDS)");
```

The complete set of LEDs that can be controlled, are members of the CCBPanelLed enumeration, which is defined, along with the functions of each LED, as follows.

```
typedef enum {
  CCB_SAMPLE_MODE_LED = 1,    /* CCB is in sample mode */
  CCB_SLAVE4_READY_LED = 2,   /* Slave-FPGA 4 firmware loaded */
  CCB_SLAVE3_READY_LED = 4,   /* Slave-FPGA 3 firmware loaded */
  CCB_SLAVE2_READY_LED = 8,   /* Slave-FPGA 2 firmware loaded */
  CCB_SLAVE1_READY_LED = 16,  /* Slave-FPGA 1 firmware loaded */
  CCB_MASTER_READY_LED = 32,  /* Master-FPGA firmware loaded */
  CCB_HEARTBEATS_OK_LED = 64, /* All FPGAs have healthy heartbeats. */
  CCB_POWER_OK_LED = 128,     /* All power-supllies ok */
  CCB_SLAVE4_ERROR_LED = 256, /* Slave-FPGA 4 firmware load-error */
```

```
  CCB_SLAVE3_ERROR_LED = 512, /* Slave-FPGA 3 firmware load-error */
  CCB_SLAVE2_ERROR_LED = 1024,/* Slave-FPGA 2 firmware load-error */
  CCB_SLAVE1_ERROR_LED = 2048,/* Slave-FPGA 1 firmware load-error */
  CCB_MASTER_ERROR_LED = 4096,/* Master-FPGA firmware load-error */
  CCB_CAL_MODE_LED = 8192,    /* Cal test-mode */
  CCB_DUMP_MODE_LED = 16384,  /* Dump mode */
  CCB_TEST_MODE_LED = 32768   /* Test mode */
} CCBPanelLed;
```

Note that the value of each enumerator is the value of a single bit, and that this thus allows them to be OR'd together to indicate any particular set of LEDs.

3.4.2 CCB_GPIO_IOCTL_SNUFF_LEDS

Use the following ioctl() call to turn off a specified set of the front-panel LEDs, without affecting the states of any of the other LEDs. The (int) argument of this call is a bit-mask union of CCBPanelLed enumerators, denoting which LEDs should be turned off. For example, to turn off the cal-mode and dump-mode LEDs, you would call ioctl() as follows:

```
#include "ccb_gpio_driver"
...
if(ioctl(fd, CCB_GPIO_IOCTL_SNUFF_LEDS,
             CCB_CAL_MODE_LED | CCB_DUMP_MODE_LED) < 0)
  perror("ioctl(CCB_GPIO_IOCTL_SNUFF_LEDS)");
```

3.4.3 CCB_GPIO_IOCTL_SET_LEDS

Use the following ioctl() call to set the states of ALL of the front-panel LEDs. The (int) argument is a bit-mask union of CCBPanelLed enumerators, in which the presence of an enumerator indicates that the corresponding LED should be turned on, and the absence of an enumerator indicates that the corresponding LED should be turned off.

For example, to turn all of the LEDs off, one would call ioctl() as follows.

```
#include "ccb_gpio_driver"
...
if(ioctl(fd, CCB_GPIO_IOCTL_SET_LEDS, 0) < 0)
  perror("ioctl(CCB_GPIO_IOCTL_SET_LEDS)");
```

3.4.4 CCB_GPIO_IOCTL_SET_PERIOD

Use the following ioctl() opcode to change the monitoring interval. This is the interval between readouts of all monitored values. The (int) argument is the integer number of milliseconds between readouts. The special value of 0, selects the default readout period, which is either the value of the readout_period parameter, that was optionally specified when the module was loaded, or the value of the CCB_GPIO_DEFAULT_READOUT_PERIOD macro in ccb_gpio_driver.h, if the readout_period parameter was not specified at load-time.

All other values are quietly rounded up to the value of the CCB_GPIO_MIN_READOUT-_PERIOD macro.

For example, to change the readout interval to 250ms, you would invoke ioctl() as follows:

```
#include "ccb_gpio_driver"
...
if(ioctl(fd, CCB_GPIO_IOCTL_SET_PERIOD, 250) < 0)
  perror("ioctl(CCB_GPIO_IOCTL_SET_PERIOD)");
```

When this call is received, the next scheduled readout-time is cancelled, and the readout timer is reconfigured to next go off at the specified interval in the future.