# The Linux EPP-port driver that controls the CCB firmware [Document number: ??, revision 1]

Martin Shepherd
California Institute of Technology

November 2, 2005

This page intentionally left blank.

**Abstract**

The linux computer that is responsible for controlling, monitoring and reading data from the CCB FPGA firmware, includes a parallel port, configured to use the IEEE-1284 EPP standard. This implements a protocol that provides separate hardware-generated I/O cycles, for reading and writing 8-bit addresses and data. These are used by the CCB firmware, to implement a register based interface, to which the computer sends commands and configuration data. The interrupt pin of the parallel port is also used by the firmware, to alert the computer to various events.

This document describes the external behavior of the device driver that sits between the EPP-enabled parallel port and the CCB server program.

# Contents

# List of Figures

# Chapter 1

# Overview

The CCB EPP device driver controls, schedules and configures the CCB firmware. It interfaces with the CCB server via a standard device file in /dev. Note that only one process is allowed to open this device at a time, and that the driver is only designed to manage one CCB peripheral, since the real-time computer only has one parallel port.

Once the device file has been opened, the server sends commands and configuration information via ioctl() calls. Notification of asynchronous events, such the availability of the start-time of the latest scan, or the availability of a log message to be picked up, are signalled initially by select() indicating that the device-file descriptor is readable, followed by the server reading a single-byte bit-mask, using read(), whose individual bits indicate which events occurred. If a particular event has associated data, such as the start time of a scan, then the server retrieves this information by using an ioctl() call.

None of the driver's ioctl() calls ever block the caller. If fcntl() is used to make the file-descriptor non-blocking, or the server is careful to only call read() when select() indicates that there are data to be read, then read() will also never block. If, however non-blocking I/O hasn't been selected, and read() is called when select() hasn't indicated that there are data available, then read() will block the calling process until the next event occurs.

New scans can be commanded at any time, via the appropriate ioctl() call. Dump-scan and intra-scan commands are forwarded to the firmware immediately, before the commanding ioctl() call returns. On the other hand, normal observing scans, which are required to start at a specified future time, are queued for later initiation, potentially long after the commanding ioctl() call returns. If the driver receives another command to start a scan, before a pending normal observing scan is initiated, then the pending scan is cancelled before the firmware ever hears about it, and the newly requested scan is scheduled instead.

Configuration information for the next start-scan command can be sent at any time. The driver guarantees that such information does not change the configuration of any previously requested scan, regardless of whether or not the previously requested scan has actually been

initiated yet.

The treatment of time is somewhat complicated. First of all, the CCB computer uses an NTP server to syncrhonize its clock with the observatory clock. This should yield accuracies of a few milliseconds. Secondly, the CCB firmware receives a 1-second tick from the observatory clock, and translates this into a parallel-port CPU interrupt, within a few microseconds. In response, the CCB interrupt handler looks at the absolute time reported by the NTP-synchronized CPU clock, and rounds this to the nearest second, to deduce the absolute time of that one-second tick, to much higher accuracy than provided by the NTP-synchronized CPU clock. This works, provided that the CPU clock is synchronized to within at worst half a second of the observatory clock.

When the interrupt handler notices that the absolute time that it just calculated matches (or has passed) the time at which it is supposed to tell the firmware about a pending scan, then it also stages a work-queue function to run at a safer time, to initiate the scan. This function, which runs in the context of a special kernel process, tells the firmware to initiate the scan, and keeps a record of when it did this.

Dump-scan and intra-scan commands are initiated directly from the ioctl() call that command them, and thus normally don't start on a 1-second tick.

Whithin less than 1$\mu$s of the driver telling the firmware to initiate a scan, the firmware zeroes a 32-bit timestamp counter. The value of this counter, which increments by one every 100ns, and wraps around to zero every 7.15 minutes, is included in the header of each packet of integrated or dump-mode data that is sent to the CCB server over the separate USB link. In order to be able to convert this time offset into an absolute time, the CCB server needs to know the absolute time at which the driver told the firmware to initiate the scan. This means that the driver needs to have a means of accurately computing timestamps whenever it initiates a scan. To this end, the 1-second interrupt handler, in addition to noting the absolute time rounded to the nearest second, also makes a note of the raw CPU clock time at this instant. Using these values, the driver can subsequently accurately figure out the current time, by taking the rounded absolute time of the 1-second interrupt, and adding to this the difference between the current CPU clock time, and the CPU clock time of the 1-second interrupt. Since the raw CPU clock times are determined by calling the Linux kernel's do_gettimeofday(), which is purported to interpolate its internal jiffies counter to microsecond accuracy, this yields an accurate scan-initiation timestamp.

# Chapter 2

# Loading the 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_epp_driver.ko major=123 port_addr=0x378 port_irq=7
```

Beware that this will take a couple of seconds, since the installation is not allowed to complete until the driver has verified that it is receiving 1PPS interrupts from the CCB firmware.

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/ccbepp c $major 0
  ```

  where $major is the selected major number.

- port_addr

  This is the base address of the parallel-port control registers. When not specified, the traditional value of 0x378 is assumed. This is the traditional value for the first parallel

port of a PC, but isn't completely standardized. The traditional values for second and third parallel ports are 0x278 and 0x3bc.

If the wrong address is assigned, then during insertion, the driver will report that it can't detect the CCB firmware, in /var/log/messages, and cause insmod to exit with an error.

- port_irq

  This specifies the interrupt request number that is used by the parallel port. If this isn't specified, then the traditional value of 7 is assumed. This is the traditional number for the first parallel port of a PC, but, like the base-address above, isn't completely standardized. The traditional IRQ numbers for second and third parallel ports are 2 and 5.

  If the wrong number is assigned, then during insertion, the driver will report that it can't see the 1PPS interrupt, in /var/log/messages, and cause insmod to exit with an error.

## 2.1 Load-time checks

When insmod is called upon to load the EPP driver, the driver checks to see whether the CCB firmware is reachable and functioning, before allowing insmod to complete. If the checks fail, then the driver unloads itself, and the exit status of insmod is non-zero.

The steps that the driver goes through to check for the firmware, are as follows.

1. First the driver allocates the resources that it needs to communicate with the parallel port.

2. Next it asserts the parallel-port's reset line for 1ms, to reset the firmware to a known state.

3. It then waits for 10ms, to give the FPGA phase-locked-loop time to lock onto the 10MHz clock.

4. The driver then attempts a single-byte EPP data-read transaction, without first sending an address-byte, to read the value of the firmware identification register. This is supposed to hold the number 27. If this EPP transaction times out, or the driver doesn't receive the number 27, then the driver complains, unloads itself and tells insmod to indicate failure.

5. Next, the driver reads back the post-reset values of all of the firmware registers, into its own shadow register bank. These should all be zero at this point.

6. The driver then enables parallel-port interrupts, and waits for up to 1.5 seconds to receive a 1PPS interrupt from the firmware, via the EPP interrupt pin. If it doesn't receive this interrupt then the driver complains, unloads itself and tells insmod to indicate failure.

7. If all of the above steps succeed, then the loading process is complete, so the driver tells insmod to indicate success.

# 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_epp_driver"
...
int fd = open(CCB_EPP_DEV_FILE, O_RDWR);
```

Thereafter the read(), ioctl(), fcntl() and select() system calls can be used to control the CCB.

## 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 there are data waiting 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 precisely one byte of data waiting to be read. This byte holds a non-zero bit-mask union of CCBEppDrvAlerts enumerators, each of which alerts the server to a specific event, whose details should be picked up via a call to ioctl().

The CCBEppDrvAlerts enumeration is defined in ccb_epp_driver.h as follows:

```
   typedef enum {
     CCB_EPP_DRV_GET_TZERO = 1, /* A new scan has been initiated. The zero */
/* offset of its timestamps is now available */
/* via the CCB_EPP_IOCTL_GET_TZERO ioctl(). */
     CCB_EPP_DRV_GET_LOG   = 2  /* A new log message is available to be picked */
/* up via the CCB_EPP_IOCTL_GET_LOG ioctl(). */
   } CCBEppDrvAlerts;
```

## 3.4 ioctl()

The ioctl() system-call is the primary means of communicating with the driver. This is easier and more efficient than using read() and write() to send and receive parameters. The following ioctl() calls are available.

### 3.4.1 CCB_EPP_IOCTL_GET_TZERO

This ioctl() call is used to get the time offset to add to firmware timestamps of the indicated scan. It should be called whenever the bit-mask returned by read() includes the CCB_DRV-_GET_TZERO enumerator. It's data argument must be a pointer to a CCBScanZeroTime variable, as shown below.

```
#include "ccb_epp_driver"
...
CCBScanZeroTime scan_zero_time;
...
if(ioctl(fd, CCB_EPP_IOCTL_GET_TZERO, &scan_zero_time) < 0)
  perror("ioctl(CCB_EPP_IOCTL_GET_TZERO)");
```

The CCBScanZeroTime type is defined in ccb_epp_driver.h, as follows.

```
typedef struct {
  unsigned long scan;      /* The ID number of the scan */
  struct timeval tzero;    /* The absolute time offset */
} CCBScanZeroTime;
```

The scan member of this type specifies which scan the timestamp refers to, using the value of the parameter of the same name, taken from the associated start-scan command.

The CCB server uses this time to offset the timestamps of the CCB packets associated with the newly started scan, as these packets are subsequently received over the separate USB link.

Note that this time isn't the same as the start-time of the scan. It is actually the time at which the scan-initiation command was sent to the firmware, and this may precede the actual start time of the scan by as much as a second.

### 3.4.2 CCB_EPP_IOCTL_GET_LOG

This ioctl() call is used to retrieve occasional error or warning messages from the driver. It should be called whenever the bit-mask returned by read() includes the CCB_DRV_GET_LOG enumerator. It's data argument must be a pointer to a CCBEppDrvLogMsg variable, as shown below.

```
#include "ccb_epp_driver"
...
CCBEppDrvLogMsg driver_log_msg;
...
if(ioctl(fd, CCB_EPP_IOCTL_GET_LOG, &driver_log_msg) < 0)
  perror("ioctl(CCB_EPP_IOCTL_GET_LOG)");
```

The CCBEppDrvLogMsg object is defined in ccb_epp_driver.h, as follows.

```
typedef struct {
  char text[CCB_MAX_LOG];   /* The text of the message. */
  CCBLogLevel level;        /* The importance of the message. */
  unsigned long id;         /* The unique identifier of the message. */
} CCBEppDrvLogMsg;
```

The CCB_MAX_LOG parameter is defined in ccbconstants.h, and the CCBLogLevel enumeration is defined in ccbcommon.h. Both of these files are indirectly included by ccb_epp_driver.h. CCB_MAX_LOG specifies the maximum length of a log message, including the '
0' terminator, while CCBLogLevel values enumerate the standard Ygor logging-levels.

Note that since log messages from the driver should be rare, and because the first message caused by any particular error condition, is usually the important one, only the first message to arrive after a call to this ioctl() is recorded for subsequent retrieval by the next call.

### 3.4.3 CCB_EPP_IOCTL_VERIFY_REGS

This ioctl() tells the driver to read-back the current values of the CCB firmware registers, and compare their values with the last values that were sent to them by the driver. This is a simple diagnostic which helps to verify the integrity of the EPP connection to the firmware, and the correct functioning of the firmware itself. It can be called at any time, as shown in the example below.

```
#include "ccb_epp_driver"
...
int discrepant_bytes;  /* The number of discrepant register bytes */
...
if(ioctl(fd, CCB_EPP_IOCTL_VERIFY_REGS, &discrepant_bytes) < 0)
  perror("ioctl(CCB_EPP_IOCTL_VERIFY_REGS)");
```

The data argument of the ioctl() call must be a pointer to an int variable. On return, this contains a count of the number of register bytes that differed from their expected values, and is thus hopefully zero.

### 3.4.4 CCB_EPP_IOCTL_GET_CLOCK_ERROR

This ioctl() returns the time offset, in microseconds, between the latest 1-second interrupt from the firmware and the nearest 1-second tick of the CPU clock. It can be called at any time, as shown in the example below.

```
#include "ccb_epp_driver"
...
long usec;  /* The clock-offset in microseconds */
...
if(ioctl(fd, CCB_EPP_IOCTL_GET_CLOCK_ERROR, &usec) < 0)
  perror("ioctl(CCB_EPP_IOCTL_GET_CLOCK_ERROR)");
```

The data argument of the ioctl() call must be a pointer to a long variable. On return, this contains the apparent clock offset, measure in microseconds.

This value isn't used during normal operation. Instead, it is intended for use as a diagnostic to determine how accurately the NTP server is keeping the computer clock synchronized to the observatory clock. Beware that this error is modulo one second, since the driver has no

way of determining which absolute second a given 1PPS tick is meant to correspond to. The assumption is thus that the NTP server will keep the computer clock synchronized to within much less than half a second of the observatory clock, and that the 1PPS pulses that the firmware received, are synchronized with the 1-second tick of the observatory clock.

### 3.4.5 CCB_EPP_IOCTL_TELL

This ioctl is used to send firmware commands and configuration data to the driver. Its data argument is a CCBDriverCmd object, which is the type of object that the CCB server uses to send commands to either the simulation driver, or the real driver. It is defined in ccbserverlink.h, which is included by ccb_epp_driver.h, and is described in the ccb_network_interfaces document. The following is a trivial example of this ioctl call.

```
#include "ccb_epp_driver"
...
CCBDriverCmd cmd;  /* The command to send to the firmware */
/*
 * Tell the driver to start an intra-scan.
 */
cmd.type = CCB_DRV_INTRA_SCAN;
cmd.pars.dump.scan = 0;
if(ioctl(fd, CCB_EPP_IOCTL_TELL, &cmd) < 0)
  perror("ioctl(CCB_EPP_IOCTL_TELL)");
```