The ccbserver program.

[Document number: A48001N009, revision 1]

Martin Shepherd
California Institute of Technology

December 29, 2005

This page intentionally left blank.

Abstract

This document describes the behavior and usage of the CCB server program. This program provides the network interfaces through which the CCB manager controls the CCB, acquires radiometer data, and receives both monitoring data and log message from the CCB. The server also provides a network interface for diagnostic programs to use to acquire raw ADC samples from the CCB.

Contents

Chapter 1

Introduction

The ccbserver program runs on the computer that is contained within each CCB. It implements a layer between the low-level device-drivers that communicate with the CCB hardware, and the network interfaces that the remote CCB manager uses to send the CCB high-level directions, and read back radiometer data, instrument-monitoring data and log messages.

The ccbserver program is invoked as follows.

```
ccbserver [-daemon]
```

Note that in the above line, where square-brackets enclose an argument, this means that the enclosed argument is optional.

If the ccbserver program is invoked without the optional -daemon argument, then the program runs interactively, and writes startup error messages to the terminal. Alternatively, if it is invoked with the -daemon argument, then the ccbserver program re-runs itself as a detached background daemon, without printing anything to a terminal, and sends startup messages to syslog.

On the CCB computer, the ccbserver program is started at boot-time as a background daemon. In this mode, log messages are sent both to any CCB manager that is connected, and also to the standard Linux syslog facility. These messages are recorded by syslog, in the following file:

```
/var/log/messages
```

This file is particularly important for seeing startup error messages, since the CCB manager doesn't have a chance to connect and receive these messages, before the startup-error causes the program to abort. To extract the subset of messages that were written to this file by the CCB server, type:

```
grep ccbserver: /var/log/messages
```

Beware that one needs to be logged in as root to be able to read this file.

Chapter 2

The hardware interfaces of the CCB server program

On the hardware side the CCB server communicates with the CCB hardware through three interfaces.

1. The EPP parallel-port.

   The parallel-port is used, in IEEE-1284 EPP mode, to control the firmware within the master FPGA of the CCB. The CCB EPP device driver translates relatively high-level commands from the ccbserver program, into appropriately scheduled sequences of reads and writes to a register-based interface in the FPGA firmware. The EPP device-driver also handles the interrupts that the CCB firmware raises, without needing to involve the ccbserver program. Thus the ccbserver program doesn't have to deal with any of the intricacies, or time-sensitivities of starting and stopping scans. Instead it simply hands off these responsibilities to the device-driver, and once it has sent a command to the EPP driver, it returns to its other tasks of waiting for commands from the CCB manager, and forwarding radiometer data from the USB port, monitoring data from the GPIO card, and log messages from a variety of sources, to the CCB manager.

   The CCB server also uses the EPP interface to reset the CCB firmware to a known initial state, whenever a new manager program connects. The EPP driver forwards this reset request to the firmware, via the standard EPP-reset line of the parallel port. This causes all flip-flops within the firmware to adopt their desired initial states. The firmware also forwards the reset signal to the USB chip, which then behaves as though it has been power-cycled. This ensures that any partially completed transaction between the CCB firmware and the USB chip isn't left hanging by the firmware reset, and also ensures that the first data that the CCB server receives, after a firmware reset, are those of the first scan that is started after the firmware reset, rather than an unknown number of trailing bytes that were queued to be sent before the reset. Thus the server

doesn't have to worry about detecting where the truncated end of the previous stream of data ends, and where the first byte of the new stream begins.

2. The USB port.

   The USB 1.1 port of the CCB computer, is used to receive integrated radiometer-data or raw ADC samples from the CCB firmware, via a USB digital-I/O chip on the master-FPGA board of the CCB. The radiometer data arrive as an asynchronous stream of data, that doesn't require the ccbserver program to do anything more than read from the USB device-driver's device-file, to solicit it. After a firmware-reset, which also resets the USB-chip, the first byte of data that arrives over the USB link, is the start of the header of the frame of data from the first integration period of the first scan that follows the reset. Data are transmitted over the USB bus, at about 1MByte/s, using USB bulk transfers.

3. The ISA-bus GPIO card.

   The ccbserver program uses the device-driver that controls the general-purpose I/O (GPIO) card, for the following tasks.

   - Reloading the FPGA firmware.

     One of the digital output bits of the GPIO card, is dedicated to reloading the firmware of the CCB FPGAs. Asserting this output causes all of the FPGAs in the CCB to reload their firmware from flash memory. After telling the GPIO driver to do this, the CCB server then calls upon the EPP driver to reset the flip-flops of the downloaded firmware into their initial states.

   - Reading from the instrument monitoring bus.

     The CCB hardware includes a monitoring bus, which allows a number of global and per-FPGA statistics, such as power-supply voltages, to be sampled. Since there are more parameters to be monitored than the number of analog and digital input channels on the GPIO card, each per-FPGA value needs to be actively solicited, by first writing the address of the parent FPGA to particular digital output bits of the GPIO card, then telling the GPIO card to initiate an analog conversion, and then finally, after a readout delay, reading back the digitized value. As a result reading out all of the monitoring statistics requires a lot of read/write transactions with the GPIO card, and careful timing.

     Communication with the GPIO card is via the ISA bus, which due to its synchronous nature, has the awkward side-effect of stalling the CPU for 0.5μs per transaction. Given the large number of transactions that are required to read out a snapshot of all of the monitor values, it is important to intersperse delays between each of these transactions, such that other processes on the computer can make headway. Without these delays, the CPU would be stalled for significant blocks of time, and this would adversely affect interrupt latency.

     The facts that many transactions are needed to read out each snapshot of all of the monitoring values, and that delays need to be inserted to prevent stalling the CPU for too long, mean that a simple device driver that merely provided the CCB

server with a way to directly initiate transactions and receive the corresponding responses, would involve a large number of expensive system calls. The need to add delays, to prevent hogging the CPU, and the need for ADC readout delays, would also make the server rather complex, especially given that it would need to be able to service its other duties of communicating with the manager and the USB driver, during these delays. For this reason, a higher-level device-driver was written that handles all of the low-level complications of collecting a snapshot of data in a suitably rate-limited fashion, and that allows the server to get on with whatever else it needs to do, while the device-driver is slowly collecting the next monitoring snapshot.

The choice of this higher-level design for the device-driver also made it possible for multiple programs to read-out snapshots, without them each having to redundantly perform the same individual transactions with the GPIO card. This is convenient, because it meant that the task of updating the front-panel status LEDs to correspond to the latest monitoring data, could efficiently be performed by a separate program from the CCB server.

The GPIO device driver periodically collects snapshots of the monitoring data, at a rate that is configurable via an ioctl() call, and the CCB server simply uses the select() system-call to watch for the availability of a new snapshot of data, while at the same time watching for USB and TCP/IP transactions that it can attend to, while waiting.

Then, whenever select() indicates that a new snapshot of monitoring data is available to be read, the CCB server reads the whole snapshot, from the driver's internal buffer, in a single read() system-call, and queues it to be sent to the CCB manager.

- Controlling the mode LEDs on the front-panel.

  Whereas the front-panel status-LEDs of the CCB are controlled by the ccb_monitor_status program, to ensure that they continue to be updated when the CCB server isn't active, the mode LEDs, which indicate the operating mode that was most recently requested by the CCB server, are controlled by the CCB server, via corresponding digital output pins on the GPIO card.

2.1 Working around motherboard hardware problems

The super-I/O interface chip on the computer mother-board is unfortunately rather buggy. The USB host-controller, in particular has serious problems, that are known, and have been discussed widely on the internet. The following sections describe these problems, and the workarounds that were used to either avoid triggering these problems, or to recover from them.

2.1.1 Problems in the parallel-port interface chip

The motherboard's parallel-port hardware doesn't implement EPP transaction-timeouts correctly. For example when the CCB firmware is reloaded while an EPP transaction is in progress, or an EPP transaction is initiated while a CCB firmware-reload operation is in progress, then the CCB firmware isn't there to complete the transaction, and at worst, the parallel-port controller on the motherboard ought to timeout. Although the controller does appear to timeout the transaction, the EPP timeout bit doesn't then indicate that it ended with a timeout. This wouldn't be particularly problematic, if it weren't for the fact that all subsequent attempts to perform EPP transactions do get aborted with the time-out bit set, even once the CCB firmware has been reloaded, and is able to respond correctly to EPP transactions. The conventional prescriptions for clearing the timeout bit, as taken from the standard parallel-port device-driver, don't seem to actually do so. In fact, once this has happened, the only way to recover the parallel port seems to be to reboot the computer. This presumably performs a hard reset of the super-I/O chip. In an attempt to find a less extreme solution, the data-sheet of the interface chip was obtained from the manufacturer, under an NDA. However there was no mention of EPP timeouts, or how to clear the EPP timeout bit.

As such, the CCB server was modified to avoid doing anything that might provoke an EPP timeout. In particular:

1. The CCB server is careful to only reload the CCB firmware when the EPP device-driver is not loaded. Since the EPP device-driver responds to 1PPS and integration-done interrupts, by communicating with the firmware over the EPP port, unloading the EPP device-driver is the only way to ensure that no EPP transaction is in progress when a firmware-reload is initiated.

2. After the GPIO card has been told to initiate a CCB firmware-reload, the CCB server delays loading the EPP driver for 1 second, to ensure that the firmware is ready to respond appropriately to an EPP transaction, before the first EPP transaction is attempted.

2.1.2 Problems in the USB host-controller chip

The USB host-controller has two serious problems.

1. If one alternately inserts and removes a USB device from the USB interface of the CCB computer, after a few such insertions and removals, the USB host-controller stops recognizing new USB devices. Note that this happens with any USB device, not only the USB chip on the CCB master card. Discussions of this problem were found on

the internet, for the particular host controller on the CCB's computer motherboard, but nobody had come up with a workaround that didn't involve rebooting.

This was a serious problem for the CCB, since every time that the CCB firmware is reset, the USB chip on the master board of the CCB is also reset, and this appears to the computer, as though the USB device had been removed and then reinserted again. This occurs whenever the CCB manager makes a new connection to the CCB server.

Fortunately a workaround for the CCB computer was found. By experimentation, it was found that unloading and then reloading the Linux device-driver of the USB host controller, would allow new devices to be successfully recognized on insertion. This could be done without having to first remove the USB device-drivers of currently connected devices, since the host controller is only involved in responding to USB insertion and removal events. Presumably, reloading the device-driver of the host controller, has the side-effect of resetting the host-controller hardware.

Thus, whenever the CCB firmware and USB chip are reset, and the CCB server attempts to open a new connection to the USB chip, via the CCB's USB device-driver for this chip, if the attempt to open the connection fails, then the CCB server unloads and then reloads the device-driver of the Linux USB host-controller, before trying again to open a connection to the USB chip. This workaround is triggered quite frequently, and works reliably.

2. Whenever the CCB manager disconnects from the CCB server, the CCB server closes its connection to the USB chip of the CCB master card. The CCB's USB device-driver for this chip, responds by no longer requesting data from the USB chip, and by canceling any previously requested, but incomplete transfer. However the USB hardware continues to receive the data of the incomplete bulk-transfer, until it has completed.

   When the CCB manager (or a CCB diagnostic program) establishes a new connection to the CCB server, the CCB server resets the CCB firmware and the CCB USB chip. If this happens too quickly, after the previous connection to the USB chip was closed, then the USB hardware stops working completely. Presumably the truncation of the final incomplete transfer from the USB chip, by the hard-reset of the USB chip, confuses the USB hardware. Even if one then unloads and reloads both the device-driver of the USB host controller, and the CCB's USB device-driver, the USB interface remains unresponsive. The only way that has been found to recover the USB interface is to reboot the computer. Thus, the CCB server attempts to avoid ever triggering this problem, by ensuring that the CCB USB chip is never reset within a few seconds of the USB device-driver being told to drop its connection to that chip.

2.1.3 An unfortunately side-effect of initializing the GPIO card

As previously mentioned, one of the digital output pins of the GPIO card is used to tell the CCB FPGAs to reload their firmware from external flash memory. A reload occurs whenever

this output is driven low. Unfortunately this doesn't only happen when one explicitly tells the GPIO card to drive this pin low, but also every time that one writes to the configuration register of the GPIO card. Thus a reload occurs whenever the CCB's device-driver for the GPIO card is loaded, since the first thing that it has to do is configure the GPIO card.

As a result, because of the previously noted requirement to not have the EPP device-driver loaded when the CCB firmware is reset, it is important that the device-driver of the GPIO card be loaded before that of the EPP port, and that a delay be inserted between the two operations, to give the firmware a chance to start running, before the EPP driver potentially starts trying to communicate with it.

2.2 Loading the CCB device drivers

The CCB device drivers are loaded once, at boot-time before the ccbserver program is started, by the ccbserver system-V initialization script. To handle the problems noted above, the loading order and timing are as follows:

1. Load the CCB device-driver of the GPIO card.
2. Wait for 2 seconds, to accommodate the firmware reload that inserting the GPIO driver will have initiated.
3. Load the CCB device-driver that manages the EPP-port connection to the CCB firmware, before loading the USB driver, since loading this driver has the side-effect of also resetting the USB chip.
4. Load the CCB device-driver that communicates with the USB chip on the CCB's master-FPGA board.

2.3 Resetting the CCB hardware

Whenever no program is connected to the paired control and telemetry links of the CCB server, the CCB server does not maintain any connections to the CCB device-drivers. This is both to ensure that the CCB hardware gets fully reset to a known state whenever the CCB manager or other program connects to the server, and also to allow the CCB server to run in simulation mode, without the need for the CCB device drivers to be present, whenever the server is being run off-line on a non-CCB computer.

Whenever a manager establishes a new connection to the CCB server, and the manager selects the real CCB hardware, as opposed to the simulated hardware, the CCB server opens connections to all of the CCB device drivers, and resets the CCB to a known initial state.

To deal with the hardware problems described previously, this has to be done with care. The procedure that is performed, is timed and ordered as follows:

1. Open a connection to the CCB's device-driver of the GPIO-card.
2. Tell the GPIO driver to reload the CCB firmware.
3. Wait for one second, to give the firmware-reload time to complete.
4. Open a connection to the CCB device-driver that uses the EPP port to control and receive interrupts from the CCB firmware. Doing this has the side-effect of resetting the CCB firmware to its desired initial state.
5. Repeat the following attempts to open a connection to the USB chip, up to 5 times before giving up and aborting.
   (a) Wait for 1 second. This gives the USB subsystem time to notice and respond to the USB chip disengaging and then reengaging to the USB bus.
   (b) Attempt to open a connection to the USB device-driver.
   (c) If the attempted open fails, unload and then reload the Linux uhci-hcd device-driver of the USB host-controller, and go back to the start of the retry loop.
   (d) If the open succeeded, exit the retry loop, and continue to set things up for the new manager connection.

Subsequently, when the CCB manager or diagnostic program disconnects from the control and telemetry links of the CCB server, the CCB server disconnects from the CCB hardware as follows:

1. Close the connection to the device-driver of the GPIO board.
2. Close the connection to the device-driver of the EPP port.
3. Close the connection to the device-driver of the USB chip.
4. Wait for 3 seconds, to prevent an immediate reconnection attempt from the manager or other program, resulting in the USB chip being reset too quickly after its connection has been terminated.

Chapter 3

The simulated hardware interface

To enable the CCB manager and server to be tested on non-CCB computers, which don't have access to the CCB hardware, the CCB server contains two hardware interface layers. One interface communicates with the real CCB hardware, as described previously, while a second communicates with an embedded software simulation of the CCB hardware. The remote CCB manager can dynamically select which of these two layers it interacts with, by sending a command to the CCB server over the control network connection of the server.

As far as is reasonably possible, the simulated hardware behaves like the real hardware, such that commands sent by the CCB manager to the CCB server, change the timing and contents of the radiometer data, log data and monitoring data that the simulation sends back to the manager. Simulated dump-mode data, when commanded, are also dispatched to any dump-mode diagnostic program that is connected, to enable the off-line testing of such programs.

Chapter 4

The network interfaces of the CCB server

The CCB server implements the following three communication channels.

- The control link, at TCP/IP port 5323.

  The control-link is used by the CCB manager to send commands and configuration information to the CCB server. The only messages that go the other way, are replies to the ping commands that check for link continuity, and command-completion messages, which tell the manager whether a command was accepted or rejected by the server.

- The telemetry link, at TCP/IP port 5324.

  The telemetry link is used by the CCB server to send asynchronous information, such as radiometer data, log messages, and monitoring data, to the CCB manager. No messages are ever sent over this link in the other direction.

- The dump-mode link, at TCP/IP port 5322.

  The dump-mode link is not used by the CCB manager. Instead, when some program, such as the CCB manager, the ccb_demo_client program, or a standalone diagnostic program sends a command over the control-link to switch the CCB firmware into dump-mode, the CCB server stops sending radiometer data over the telemetry link, and instead dispatches dump-mode data over the dump-mode link. This allows small custom dump-mode diagnostic programs to be written that don't have to be complicated by having to also control the CCB.