# The software installation and configuration of the CCB computer

[Document number: A48001N010, revision 1]

Martin Shepherd
California Institute of Technology

December 29, 2005

This page intentionally left blank.

**Abstract**

This document describes the installation, configuration and future maintenance of both the Linux operating-system, and the CCB software, on the embedded computer of the Caltech Continuum Backend.

# Contents

# Chapter 1

# Introduction

The Caltech Continuum Backend (CCB) contains a Lippert, "*Cool Roadrunner III*" PC104+ single-board computer, that acts as an intelligent intermediary between the CCB Manager and the CCB hardware. This computer runs a stock copy of the Fedora Linux operating-system, with non-essential services turned off, and uses custom CCB device-drivers to communicate with the CCB hardware via a USB link, an EPP-enabled parallel-port link, and a PC104 general-purpose I/O (GPIO) board. Two custom programs run continuously on this computer, running as background daemons. One controls status LEDs on the front panel of the CCB, according to monitoring information that it acquires through the GPIO board. The other is the CCB server, which allows remote programs, such as the CCB Manager, to use TCP/IP links both to control and receive data from the CCB, over the network. The computer's clock runs in UTC and is synchronized to the Green Bank observatory's clock, using a standard Linux NTP client. Time-scheduling of commands and time-stamping of data, which require even greater precision, use a combination of the computer's clock, and a 1-pulse-per-second (1PPS) interrupt.

The architecture of the system is summarized in figure 1.1.

The CCB computer has a 2GB microdrive, plugged into a compact-flash socket on the CPU board. This is where the operating system and the CCB software are installed. The computer thus operates in a standalone configuration, without any dependence on other computers for booting or software. The CPU on the board is an Intel Pentium-III, running at 933MHz. The CPU board allows a maximum of 512MB of RAM to be installed, and boards with this maximum were bought. The historical reason for requiring this amount of RAM was to allow the root filesystem, the running kernel and CCB software, to reside entirely in RAM, after being uncompressed from an image on a flash drive. In practice, a large microdrive was used, instead of a flash drive, and the RAM was not needed for this purpose. However it is still useful to have a lot of RAM, both to avoid the CCB software from ever having to suffer from paging delays, and to allow another cut-down OS to be booted into RAM to facilitate off-line maintenance of the root filesystem on the microdrive.

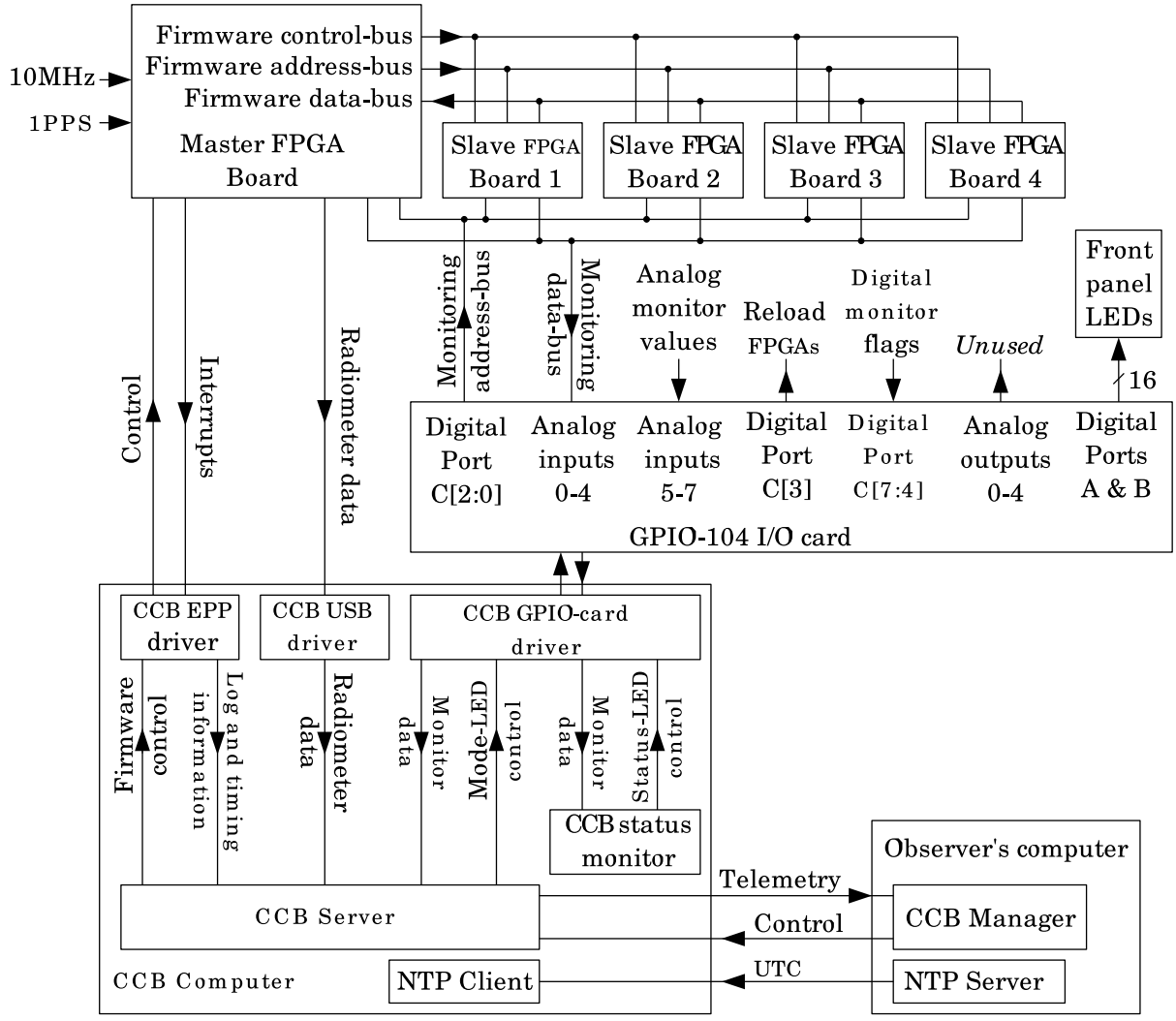


Figure 1.1: The computer architecture of the CCB

The computer board and the GPIO board were left in their manufacturer's default hardware configurations. No jumpers were either installed or removed.

# Chapter 2

# Installation of Fedora Linux

## 2.1 The choice of a stock Fedora Linux distribution

Fedora Core-4 Linux was used as the operating system of the CCB computer. Since the CCB firmware was designed to minimize the real-time requirements of the computer software, there was no need to run a real-time enhanced version of Linux. In principle this means that in the future a different version of Linux could be installed, without having to worry about configuring real-time extensions. However note that the custom CCB device drivers were written for the 2.6 series of the Linux kernel, and that future kernels probably won't be backwards-compatible at the driver level. Thus the device-drivers might need to be ported to a future kernel. In practice, it shouldn't ever be necessary to upgrade to a newer version of Linux, provided that the current version works reliably and that the operating system doesn't contain a remotely exploitable security weakness that can't be resolved without patching the kernel.

## 2.2 Security Maintenance

The only network ports that are left open on the CCB are those of the ssh daemon, and the CCB server. The computer's own firewall, along with the observatory firewall and host-authorization checks by the CCB server, prevent any of these ports from being accessed by computers outside of the observatory. An attacker would thus have to break into another observatory computer before being able to attack the CCB computer.

The CCB server itself provides no access-control, beyond IP-address based host-authorization. Thus an attack launched via a compromised observatory computer could potentially take control of the CCB, or exploit a bug in it to install malicious software. This would, however require an unlikely degree of determination of the attacker, since it would involve studying

the CCB software and its protocols in depth, to find a weakness. More likely would be an attack on the ssh daemon, using a hack developed for a future vulnerability. Thus it may be necessary to upgrade the ssh daemon occasionally. Note that doing this involves logging in as root and running the following command:

```
/usr/bin/yum update ssh
```

## 2.3 The choice of installation medium

The Cool Roadrunner III CPU-board has a non-standard connector for the CDROM drive, and the adapter cables that came with the board, turned out not to fit a standard internal PC CDROM drive. Also, the only USB CDROM drive that was available, wasn't recognized by the BIOS. The BIOS also didn't support booting from a USB flash-drive, and although it did recognize a USB floppy drive, it turned out that there weren't any Fedora network boot-loaders that would fit on a floppy. This presented a serious problem, because even to perform a network installation of Fedora, one first needs to be able to install a Fedora-compatible network boot-loader from some other medium.

The final solution was thus to temporarily remove the microdrive from the CPU board, and install a network boot-loader onto it, using a flash-card reader on another computer.

## 2.4 IMPORTANT: The device-file of the microdrive

Many of the operations in this document are performed on the raw device that Linux uses to access the microdrive. It is vitally important to use the correct device-file. Getting this wrong could erase or corrupt another device, such as a workstation hard-disk.

In this document, whenever this device-file is cited, it is written as $MICRODRIVE, where MICRODRIVE is an environment variable that must be assigned the correct device-file name. This device-file name must be the one that refers to the whole microdrive, not to one of its partitions. Thus /dev/hdc would be a valid value, whereas /dev/hdc1 would not, since it refers to the first partition. In cases where the device-file of a partition is needed, this will be formed by appending the partition number to the file of the whole device. For example, ${MICRODRIVE}1 is the device-file of the first partition of the microdrive.

### 2.4.1 The fixed device-file of the microdrive on a CCB computer

If the microdrive is plugged into a CCB computer, then it will have the fixed device-file that is associated with the IDE port of the flash-card socket on the CCB computer. Since the

BIOS associates the device HDC with this socket, Linux assigns this device the device-file called /dev/hdc. To check that this is correct, type the following:

```
cat /proc/ide/hdc/model
```

If the disk is one of the original Hitachi microdrives, then the result of this command should be the string HMS360402D5CF00, which is the model number of this drive.

Having verified this, type the following to assign the device-file to the MICRODRIVE environment variable, before cutting-and-pasting any operation that operates on the microdrive.

```
MICRODRIVE="/dev/hdc"
```

Since most of these operations must be performed as root, which uses the bash shell, the above assignment is shown as a Bourne-shell statement.

### 2.4.2 The variable device-file of the microdrive on a workstation

Alternatively, if the microdrive is plugged into a USB flash-card reader on a workstation PC, then Linux will assign it a temporary SCSI device-file, such as /dev/sda. Beware that this can vary from one insertion to the next, depending on what other USB or SCSI devices have been inserted before it.

Since USB mass-storage devices are represented as SCSI devices under Linux, the appropriate device-file can be figured out by invoking the scsi_info command for each possible SCSI device, until something that looks like the microdrive is indicated. SCSI device files are assigned in the order in which the device is detected, starting with /dev/sda, followed by /dev/sdb etc...

For example, on the author's laptop, which has a vacant Sony memory-stick reader in one USB port, and a SanDisk USB flash-drive plugged in another, the first three SCSI devices are reported as follows:

```
% scsi_info /dev/sda
SCSI_ID="0,0,0"
MODEL="Sony MSC-U01"
FW_REV="1.00"
% scsi_info /dev/sdb
SCSI_ID="0,0,0"
MODEL="SanDisk Cruzer Mini"
FW_REV="0.2"
% scsi_info /dev/sdc
open() failed: No such device or address
```

Similarly, when the microdrive is plugged into a flash-card reader, the name of its manufacturer, or its model number should be displayed in the MODEL="" field. In general, unless another SCSI or USB device has been plugged in since the microdrive was inserted, the microdrive should be the last device that is listed, before receiving the No such device or address message.

Having determined which device is attached to the microdrive, assign it to the MICRODRIVE environment variable, before following any instructions that refer to the microdrive.

## 2.5 Remedial steps after a failed installation

During the first attempt at installing Fedora, the installer crashed, due to there being insufficient space on the microdrive, for all of the selected packages. This unfortunately corrupted the MBR (Master Boot Record) of the microdrive, such that subsequent attempts to boot from boot-images loaded onto the microdrive, resulted in the BIOS of the CCB computer simply saying:

```
No operating system found
```

It also left a partition table that contained an LVM partition that wasn't recognized by the GRUB or LILO boot-loaders. Before another attempt to install Fedora could be made, the following remedial steps thus had to be taken. These steps shouldn't be necessary when installing on a previously unused microdrive.

### 2.5.1 Fixing the partition table

With the microdrive placed in a flash-card reader on another computer, the Linux fdisk command was used to first remove all of the unwanted partitions that the installer had created, and then to replace them with a single normal partition that encompassed the whole disk.

The microdrive had 2 partitions on it, so the repartitioning consisted of individually deleting these two partitions, using the fdisk d command, followed by creating the new partition with the n command. Specifically, the commands used were as follows.

```
/sbin/fdisk $MICRODRIVE
d
1
d
```

```
2
n
p
1
<return>
<return>
w
```

See section 2.4 for an explanation of the MICRODRIVE environment variable.

### 2.5.2 Fixing the MBR of the microdrive

When the Master Boot Record of a disk has been corrupted, the only way to fix it, is to install a new boot-loader there. So the microdrive was placed in a flash-card reader on a separate computer to do this. Since the grub boot-loader, which is what comes with all recent linux distributions, refuses to install itself on a disk that isn't known by the BIOS, and removable drives, such as the microdrive placed in a flash-card reader of a desktop computer, aren't known by the BIOS, the older LILO boot-loader had to be installed. This was done as follows.

```
mkdir ~/src/lilo
cd ~/src/lilo
wget http://gd.tuwien.ac.at/opsys/linux/lilo/lilo-22.6.1.binary.tar.gz
tar xzf lilo-22.6.1.binary.tar.gz

cat > lilo.conf << EOF
boot=$MICRODRIVE
EOF

sbin/lilo -b $MICRODRIVE -C ./lilo.conf -P fix -v -M $MICRODRIVE mbr
```

See section 2.4 for an explanation of the MICRODRIVE environment variable.

After doing this, the CCB computer returned to being able to boot from any boot-image that was loaded into the first (and only) partition of the microdrive.

## 2.6 Where to find copies of the Fedora Core distribution

The installation was performed from Caltech's mirror of the Fedora Linux CDs, at toughguy.caltech.edu. To perform a new installation, select a site from one of the official mirrors listed at:

```
http://fedora.redhat.com/download/mirrors.html
```

## 2.7 Creating a Fedora boot-disk image on the microdrive

The first step was to download a copy of the first installation CD of Fedora Core 4.

```
wget ftp://toughguy.caltech.edu:/pub/linux/fedora/linux/core/4/i386/iso/FC4-i386-disc1.iso
```

This was written to a blank CD by placing the CD in a CD-writer and typing:

```
cdrecord -v FC4-i386-disc1.iso
```

The CD was then mounted.

```
mount cdrom
```

Note that the need to burn and mount a CD could have been avoided by loopback mounting the ISO filesystem. This wasn't done, simply because the author didn't have root permission on the PC that this was being performed on.

After placing the CCB microdrive in a flash-card reader, and assigning the corresponding device-file name to the MICRODRIVE environment variable (see section 2.4), the Fedora boot-image was then copied from the CD, to the first partition of the microdrive.

```
dd if=/mnt/cdrom/images/diskboot.img of=${MICRODRIVE}1
```

The microdrive was then temporarily mounted, using the /mnt/flash mount-point that was assigned to it in /etc/fstab, and its contents listed, to check that the above copy resulted in a mountable file-system.

```
mount /mnt/flash
ls /mnt/flash
umount /mnt/flash
```

This yielded the following listing.

```
boot.msg     isolinux.bin  options.msg  snake.msg     vmlinuz
general.msg  ldlinux.sys   param.msg    splash.lss
initrd.img   memtest       rescue.msg   syslinux.cfg
```

## 2.8 Telling the BIOS to boot from the microdrive

With the CCB computer turned off, the microdrive was placed in the flash-card carrier of the CCB computer, and a keyboard and monitor were plugged in to the CCB computer board, using the adapter cables that came with the computer. The CCB computer was then switched on, and as the computer was going through its power-on self-test, the DEL key was hit repeatedly, until the monitor warmed up and showed that this had selected the BIOS configuration screen.

After selecting the "*Advanced BIOS Features*" BIOS-screen entry, followed by moving the cursor to the "*First Boot Device*" entry that this listed, and hitting the Enter key, a list of potential boot devices was presented. From this, the entry for HDD-0 (the microdrive) was selected. The new settings were then saved, and the BIOS-screen exited, by first hitting the F10 function key, followed by Enter. This then initiated the boot process.

## 2.9 Running the Fedora Linux installer

After switching on the computer, and completing the BIOS configuration to boot from the microdrive, the BIOS proceeded to load the Fedora boot-image that had been installed there (see section 2.7). This then prompted for the installation method. Since a previous attempt had used graphical mode, and this had failed, text-mode was selected, by typing:

```
linux text
```

at the Boot: prompt, and then hitting return. The following is a summary of the prompts that the installer presented, and the corresponding responses. Note that, because previous installation attempts had suffered problems with LVM partitions, normal partitions were selected in this attempt.

```
Language: English
Keyboard: us
Installation method: FTP
Configure TCP/IP: DHCP
FTP Setup:
  FTP site name: toughguy.caltech.edu
  Fedora Core directory: /pub/linux/fedora/linux/core/4/i386/os
Mouse Not Detected: "Use Text Mode"
Welcome to Fedora Core: OK
Installation Type: Custom
Disk Partitioning Setup: Autopartion
```

At this point there was a warning, saying that this would delete the current contents of the disk. This could safely be ignored.

```
(Selected the "Format Drive" button)
(Selected the "Remove all partitions" menu entry)
(Kept the default selection of drive "hdc" [which is the only drive])
Are you sure? Yes

Partitioning:

  (Deleted all but the 100MB /boot partition, including LogVol00)
  (Created a new primary swap partition of 128MB)
  (Created a new primary / partition with the remaining space [1717MB])

 Partioning Warning (regarding non-optimal swap size): Yes
 Boot loader configuration: Grub
```

The next question that the installer asked was which options should be passed to the kernel at boot time. It was decided to tell the kernel to not use DMA, because, as had previously been discovered, the boot process would otherwise repeatedly attempt to access the microdrive using DMA, only to eventually give up and fall back to non-DMA access. Turning off DMA thus avoided a lengthy wait, and lots of error messages, during boot.

ACPI, which provides enhanced power-management, was also disabled, since this is generally the first suspect in many device-driver related problems, and is unnecessary for the CCB computer.

```
 Boot loader options: ide=nodma acpi=off
 Boot loader password: (none)
 Boot loader labels: (leave at defaults)
```

```
   Boot loader installation location: MBR
   Network Configuration: DHCP  +  Activate on boot
   Hostname configuration: DHCP
   Firewall:
      (*) Enable firewall ( ) No firewall
      Customize: Remote login (ssh)
```

For the following "Security Enhanced Linux" question, it was decided that SELinux should be disabled. During a previous installation, SELinux had prevented the NTP daemon from working, and had just kept getting in the way of many system-administration tasks. Furthermore, since there weren't going to be any public user-accounts on this computer, and the computer has its own firewall, enabling SELinux would have been pointless.

```
   Security Enhanced Linux: disabled
   Language support: English (USA)
   Time Zone Selection:
      [*] System clock uses UTC
      US Eastern
   Root Password: ******
```

The next step was package-selection. The 2GB size of the microdrive made it difficult to find a minimal selection of packages that would leave sufficient room for a home directory with space for both software and diagnostic data files. In particular, it was necessary to avoid installing anything that had any dependencies on X-windows, since X takes a lot of space. For example, this meant that the emacs editor could not be selected. Similarly, many package groups included not only things that were needed, but also lots of things that weren't needed. After a previous installation, many of these unwanted parts were removed by hand, using the rpm -e command. However the next time that the yum program was used to install patches, yum reinstalled everything that had been removed in this way. Thus, to ease future maintenance, it was necessary to settle on a group of packages, at install time, that didn't take up too much space.

```
   Package Group Selection:
      Editors   (press F2 to make sure that only vi is selected)
      Development tools (press F2)
autoconf
automake*
binutils
gcc
gdb
ltrace
make
```

```
patchutils
pstack
strace
   Complete: Reboot
```

The CCB computer booted itself successfully from the microdrive.

## 2.10 Patching Fedora

The next step was to bring the installation up to date with any patches that had been released since Fedora Core-4 was released. This was done by logging in to the CCB computer as root, and typing:

```
yum update
reboot
```

## 2.11 Install the NTP daemon

The *Network Time Protocol* daemon conditions the rate and offset of the computer clock to synchronize it with the observatory clock. This was installed and activated by typing:

```
yum install ntp
/sbin/chkconfig ntpd on
reboot
```

Note that it took about 10 minutes for this to initially correct the clock, because the clock was initially many hours wrong.

## 2.12 Configure the firewall

Under Linux the iptables firewall allows one to specify which remote computers are allowed to connect to particular network services. While logged in as root, the following statements configured the firewall to allow all observatory computers to connect to the ssh daemon, and to the CCB server's three TCP/IP ports (5322, 5323 and 5324).

```
for src in 199.88.192.0/24 192.33.116.0/24 172.23.1.0/24 ; do
 iptables -I RH-Firewall-1-INPUT 8 -p tcp -s $src --dport 22 -j ACCEPT
 iptables -I RH-Firewall-1-INPUT 8 -p tcp -s $src --dport 5322 -j ACCEPT
 iptables -I RH-Firewall-1-INPUT 8 -p tcp -s $src --dport 5323 -j ACCEPT
 iptables -I RH-Firewall-1-INPUT 8 -p tcp -s $src --dport 5324 -j ACCEPT
done
/sbin/service iptables save
```

## 2.13 Turn off unnecessary services

To speed up reboots, free-up resources, potentially improve reliability, and improve security, a number of unnecessary services were turned off, by typing the following, as root.

```
for s in bluetooth cups gpm sendmail isdn kudzu mdmonitor \
         netfs nfslock rhnsd pcmcia acpid rpcidmapd rpcgssd \
         portmap atd haldaemon; do
  /sbin/chkconfig $s off
done
```

## 2.14 Fix the time-zone

Even though the "*System clock uses UTC*" field was selected during Fedora installation, somehow the hardware clock ended up storing time in local-time, instead of UTC. Fixing this involved typing the following, root.

```
cd /etc

ln -f ../usr/share/zoneinfo/US/Eastern localtime

cat > /etc/sysconfig/clock << EOF
ZONE="US/Eastern"
UTC=true
ARC=false
EOF
```

## 2.15 Turn off IPV6

According to comments found on the web, enabling IPV6 support can slow down IPV4 internet connections if the local router is configured for IPV6, but doesn't actually have any IPV6 endpoints.

Wolfgang turned off IPV6 support by adding the following line to /etc/modprobe.conf.

```
alias net-pf-10 off
```

## 2.16 Turn off auto-loading of the serial driver for the FTDI USB chip

The CCB hardware uses a USB chip from FTDI to communicate with the CCB computer over the USB bus. This chip is the successor to an older serial-I/O USB chip from the same manufacturer. Although there is no open-source Linux device-driver for the newer parallel-I/O chip, there is one for the older serial-I/O chip, and this works, in a backwards compatible fashion with the newer chip. Thus, when any module containing this chip is plugged into the USB port of the computer, Fedora Linux loads the serial-I/O driver for it. However, since this driver doesn't support the block-transfer capabilities of the newer chip, a custom CCB driver was written to take advantage of the extra speed of the new chip. It was thus necessary to prevent the older driver from being automatically loaded when this chip was detected by the USB hot-plug scripts.

The name of the old serial-I/O driver is ftdi_sio, and the name of the custom CCB parallel-I/O driver is ftdi_pio. Thus to have the hot-plug scripts associate the chip with the newer driver, the latter name was substituted for the former one, wherever it was found in the hot-plug scripts, as follows.

```
cd /etc/hotplug
for file in usb.rc usb.distmap; do
  ed -s $file << EOF
   ,s|ftdi_sio|ftdi_pio|g
   w
   q
EOF
done
```

The relative timing of the loading and unloading of all of the custom CCB device-drivers needs to be carefully controlled, for reasons that will become apparent later. So it was

decided not to have the CCB driver modules loaded and unloaded automatically by the hot-plug scripts. Thus, instead of replacing the old serial-I/O driver, in the /lib/modules hierarchy, with the CCB parallel-I/O USB driver, the entry of the serial-I/O driver, in the map of USB modules, was simply commented out. This thereafter prevented the USB subsystem from loading this driver.

```
cd  /lib/modules/`uname -r`
ed -s modules.usbmap << EOF
  ,s|^ftdi_sio|#ftdi_sio|g
  w
  q
EOF
```

## 2.17 Keeping a copy of the DHCP network-startup script

When Fedora was installed above, DHCP networking was selected. Although the CCB IP-address allocation scheme subsequently displaces the script that initiates DHCP networking, a copy of the original script should be kept, just in case it is later necessary to revert to DHCP. The contents of this script include a line which specifies the hardware address of the computer's ethernet interface. This line isn't actually needed, and is a liability, because when the root-filesystem is cloned to the microdrive of another CCB computer, the hardware address will be wrong. Thus the following statements were executed, as root, to both copy this file to a backup location, and remove the redundant hardware-address line.

```
cd /etc/sysconfig/network-scripts
cp -p ifcfg-eth0 original_ifcfg-eth0
ed -s original_ifcfg-eth0 << EOF
  /HWADDR/d
EOF
```

## 2.18 Set up the COM1 port

By default, all console messages, including the BIOS configuration screen, the boot-loader menu, the sequence of log messages that are displayed while Linux is booting, and the initial login screen, to be directed to the VGA port, for display on a monitor. Similarly, all keyboard input is taken from the PS2 input. However the monitor and keyboard sockets aren't available when the CCB is on the telescope, whereas the serial COM1 port is. Thus it was necessary to find a way to switch to using the COM1 port for console input and output.

Since the BIOS that came with the *Cool Roadrunner III* computer doesn't support console redirection, the scheme that was implemented, doesn't start redirecting console I/O until the boot-loader takes over from the BIOS.

### 2.18.1 Telling the boot-loader to use the COM1 port

When the BIOS hands over control to the GRUB boot-loader, GRUB displays a menu of booting options, taken from /etc/grub.conf, and for a few seconds, prompts, either for one of these menu options to be entered, or for alternate booting directions to be entered, before defaulting to using the first menu option to boot the computer. To switch this menu screen and its keyboard input, to the COM1 port, it was necessary to edit the GRUB configuration file, as follows:

```
cd /etc
vi grub.conf
```

On the line that preceded the first 'title' command, the following lines were inserted.

```
serial --unit=0 --speed=9600 --word=8 --parity=no --stop=1
terminal --dumb --silent serial
```

The first of these lines configured GRUB to use the COM1 port (unit 0), as its serial port, with a speed of 9600 baud, 8-bits, no parity and one stop-bit.

The second line then configured GRUB to interact with a dumb-terminal connected to the above serial port, without either prompting for, or requiring that an operator first hit a key. Note that without the above –dumb argument, GRUB would assume that the terminal was VT100 compatible, and thus be able to control the terminal more efficiently. This would probably be beneficial, but hasn't been tested.

### 2.18.2 Telling Linux to use the COM1 port as its console

Once the boot-loader hands over control to the Linux kernel, the kernel also needs to be told to use the COM1 port as its console. This was done by again editing the GRUB configuration file, but this time to specify arguments to the boot parameters that GRUB passes to the kernel.

```
cd /etc
vi grub.conf
```

The following arguments were then appended to the first line that started with the command-name, kernel.

```
console=tty0 console=ttyS0,9600n8
```

This told the kernel to direct all console messages both to the VGA monitor, via /dev/tty0, and to the COM1 serial port, via /dev/ttys0. Again, the COM1 port was configured to run at 9600 baud, with no parity, 8 bits, and a default of one stop-bit.

When the kernel is told about multiple console devices, in this way, the final device that is specified, is the one from which keyboard input is solicited. Thus this also arranged that the COM1 port be used for keyboard input.

The resulting kernel line, in /etc/grub.conf looked as follows.

```
kernel /vmlinuz-2.6.12-1.1398_FC4 ro root=LABEL=/ ide=nodma acpi=off console=tty0 console=ttyS0,9600n8
```

### 2.18.3 Configuring Linux to allow logins on the COM1 port

In addition to having console messages displayed to the COM1 port, it is possible to configure the COM1 port to allow logins. This was done by editing /etc/inittab, as follows.

```
cd /etc
vi inittab
```

The following lines were then appended to this file.

```
# Serial console login
S0:2345:respawn:/sbin/agetty -L ttyS0 9600
```

The agetty program thereafter presents a login prompt on the COM1 port, when pertinent, and then hands over control of the COM1 port to the login program, when a user-name and password are entered.

### 2.18.4 Authorizing root logins over the COM1 port

By default, root is only allowed to login on certain terminals, and this doesn't include terminals that are connected to the COM1 serial-port. Root access was thus enabled on the COM1 port by editing /etc/securettys, as follows.

```
cd /etc
vi securettys
```

The following line was then appended to this file.

```
ttyS0
```

## 2.19 Updating the man pages and file database

Once most of the installation had been done, the searchable indexes of man pages and files was brought up to date.

```
updatedb
makewhatis
```

The first of these commands made an index of a snapshot of all files in the system, for subsequent use by the locate command. The second command made an index of all of the man pages that were installed in standard locations, for use by the apropos and man -k commands.

## 2.20 Turn off unnecessary cron jobs

By default, Fedora is configured to periodically run a number of maintenance tasks. Many of these can load down both the CPU and the disk for several minutes at a time, and this could cause the CCB software to become sluggish, and to potentially drop integrations. None of the worst offenders are needed by the CCB, and were thus turned off, as follows. See later for why the rpm cron script is run first.

```
cd /etc/cron.daily
./rpm
chmod -x cups makewhatis.cron prelink rpm slocate.cron yum.cron
cd /etc/cron.weekly
chmod -x makewhatis.cron yum.cron
```

The tasks that were disabled by this, were the following.

- cups

  This job deletes files related to print-jobs that have accumulated in the /var/spool/cups/tmp/ directory. The CCB computer isn't configured for printing, so this was redundant, albeit trivially quick.

- makewhatis.cron

  This job reads every man page on the computer, in order to update a table of man-page subjects. Since new man pages won't be being installed very often, and this facility is no more than a convenience, it makes no sense to have this job run automatically every week. It can be done by hand, if ever needed.

- prelink

  This job goes through every shared library and program in the standard system directories and modifies them to speed up the time that it takes to preform relocations of shared-library symbols, when a program is run. This is another CPU and disk hog, and was designed to speed up the loading times of applications, that needed to load lots of large C++ shared libraries. This doesn't apply to any programs used on the CCB computer.

- rpm

  This creates an up to date list of all RPM packages that have been installed on the computer, and writes this to /var/log/rpmpkgs. This file doesn't appear to be used by anything. In principle it can be useful if the official database of installed RPMs ever becomes corrupted, and needs to be restored. Thus it may be useful to update this file by hand, if any new RPMs are ever installed/upgraded. This can be done as follows:

  ```
  cd /etc/cron.daily
  chmod +x rpm
  ./rpm
  chmod -x rpm
  ```

- slocate.cron

  This job reads every single directory on all mounted file-systems, in order to update an index of files. This facilitates the locate command. Since the locate command is simply a convenience, it doesn't matter if its database subsequently becomes out of date.

- yum.cron

  This uses the yum command to update itself daily, over the network. This can take a long time, because it needs to load a large number of header files from the yum repositories. It doesn't seem advisable to make the CCB dependent on daily calls to a site that might not even exist in a few years. If yum needs to be updated, it can simply be done by hand.

# Chapter 3

# Installing the CCB software

The CCB software consists of two daemons that run from a user account, 3 custom device-drivers that run within the kernel, a startup script that runs as root, at boot-time, and a shutdown script that is run as root, when the computer is shutdown. There are also a few ancillary scripts and configuration files, that are used by these modules, plus a number of diagnostic programs, that can be run by hand, when needed.

## 3.1 Creating the ccb user-account

The first step in installing the CCB software was to create a user account for it. This account is used for compiling the CCB software, and running the two CCB daemons. Creating the account was done, as root, as follows.

```
echo 'citmbr:x:210' >> /etc/group
useradd -G 210 -u 3000 ccb
passwd ccb
chsh -s /bin/tcsh ccb
```

Since tcsh was chosen as the shell of the `ccb` account, the `~ccb/.cshrc` file was then set up as follows:

```
/bin/su ccb
cd
cat >> ~/.cshrc << \EOF
umask 022
set history=200
```

```
set savehist=50
set notify
set ignoreeof

setenv LESS "-c -i -Q -P \
  ?f%f:Standard input.  ?p%pb\%:.  ?lLines %lt to %lb:. ?l?Lof:. ?L%L:."
setenv PAGER /usr/bin/less

if ($?tcsh) then
  set correct=cmd
  set nobeep
  set autolist
  set rmstar
  unset autologout
  limit coredumpsize 0
endif

alias rm rm -i
unalias ls
EOF
```

## 3.2 Authorizing the ccb account to run privileged programs

To avoid unnecessarily exposing the CCB computer to any security vulnerabilities in the CCB server, the CCB programs are run as normal user processes from the ccb account. This means that they can't execute anything that requires root privilege. However the CCB server needs to be able to reboot the computer, shutdown the computer, and load and unload device drivers, and these are all tasks that must be run as root. The solution that was chosen to address this difficulty, was to set up the sudo command to allow these commands to be run from the ccb account. This was done by editing the sudo configuration file, as follows.

```
cd /etc
chmod +w sudoers
echo "ccb ALL = NOPASSWD: /sbin/reboot, /sbin/poweroff, \
      /usr/local/bin/load_driver, /usr/local/bin/unload_driver" >> sudoers
chmod -w sudoers
```

Note that the load_driver and unload_driver files cited above, are CCB scripts that, when told to load any of the CCB device-drivers, also automatically create any corresponding device

files. Other non-CCB drivers are simply loaded or unloaded, as necessary. The installation of these scripts will be described later.

## 3.3 Compiling the CCB software

A compressed tar file of the CCB distribution can be found on the Green Bank Linux computers, in the home directory of the author. This was copied into the ccb home directory, and unpacked there, while logged in as the ccb user.

```
cd ~ccb
scp mshepher@ssh.gb.nrao.edu:/home/users/mshepher/ccb.tar.gz .
tar xzf ccb.tar.gz
```

This created a hierarchy of directories under a top-level directory called, /home/ccb/CCB. To build the code in this distribution, a separate build-directory was built. The CCB's configure script was then run from there, and the code compiled and linked, using the make command.

```
mkdir -p ~/build
cd ~/build
../CCB/configure
make
```

This built all of the libraries and executables in the build directory. To install them in appropriate sub-directories of the default /usr/local/ installation area, required root privilege. The installation was thus performed as follows.

```
/bin/su -c 'make install'
Password: ******
```

### 3.3.1 The CCB libraries that are installed

The following libraries are installed by the make install command. Each library is installed with three names, starting with one that has the full, *.major.minor.micro*, version number appended to it, followed by one that has just the major component of the version number appended to it, followed by one that has no version number. The latter two names are simply symbolic links to the first. Programs should be linked against the name that has no version number. The linker will then link against the latest version, through the symbolic link. Since the linker records the target of the symbolic link, rather than the link name, as the

dependency of the library; when the program is subsequently run, it will load the library by its full version-numbered name. Thus, when later, a new version of the library is installed, while newly linked programs will see the latest library version, through the version-less symbolic link, previously linked programs will continue to use the older version-numbered name.

In the following list of the CCB libraries, the libraries are referred to by their version-less names. See the ccb_network_interface document for more details regarding these libraries.

- /usr/local/lib/libccbclientlink.so

  This library provides the client end of the network interface to the remote CCB server, that is used by the CCB manager, the ccb_demo_client program, and other test programs, both to send commands to the CCB, and to receive radiometer data back from the CCB.

- /usr/local/lib/libccbdump.so

  This library provides the client end of the network interface to the remote CCB server, that is used by programs that need to receive dump-mode data from the CCB, while the CCB is being separately controlled by the CCB manager, or another test program.

- /usr/local/lib/libccbtclclient.so

  This library provides a TCL wrapper around the libccbclientlink.so and libccbdump.so libraries, which facilitates the writing of interactive TCL/TK test programs, such as the ccb_demo_client program.

- /usr/local/lib/libccbcommon.so

  This program provides functions that are useful to all CCB programs, at both the server and client ends of the CCB communication's interfaces. It is a dependency of all of the other libraries, and is thus automatically loaded by them.

- /usr/local/lib/libccbserverlink.so

  This library implements the network communication's interface used by the CCB server, to talk to remote manager and diagnostic client programs.

- /usr/local/lib/libccbtestclient.so

  This library provides a simplified interface for diagnostic programs to use to control the CCB, and receive either integrated radiometer data, or dump-mode samples from the CCB. It is implemented as a layer on top of the libccbclientlink.so, libccbdump.so and libccbcommon.so libraries.

### 3.3.2 The CCB header files that are installed

The installation procedure installs the following header files. See the ccb_network_interface document for more details regarding their contents.

- /usr/local/include/ccbclientlink.h

  This header file must be included by all CCB client programs that wish to use the interface provided by the libccbclientlink.so library. It provides function prototypes, data-type definitions and macros for the facilities in that library.

- /usr/local/include/ccbconstants.h

  This header file defines constants that parameterize the characteristics of the CCB. With the exception of the ftdi_pio.h header file, this file is automatically included if any of the other header files are included.

- /usr/local/include/ccb_epp_driver.h

  This header file defines the ioctl() commands, constants and data-types that are needed to interact with the CCB device driver that controls the CCB firmware via the computer's EPP-enabled parallel-port. It is only used by the CCB server program.

- /usr/local/include/ccbserverlink.h

  This header file is included by the CCB server. It provides function prototypes, data-type definitions and macros for the facilities in the libccbserverlink.so library.

- /usr/local/include/ftdi_pio.h

  This header file defines the ioctl() commands, constants and data-types that are needed to interact with the CCB device-driver for the FTDI USB module. This driver was written to be a general purpose driver for this chip, and thus has no dependencies on the CCB.

- /usr/local/include/ccbcommon.h

  This header file, which is automatically included by the ccbclientlink.h, ccbserverlink.h, ccbdump.h, and ccbclientlink.h header files, defines function prototypes, data-types and macros for the facilities in the libccbcommon.so library.

- /usr/local/include/ccbdump.h

This header file provides function-prototypes, data-type definitions and macros for use by programs that link with the libccbdump.so library.

- /usr/local/include/ccb_gpio_driver.h

  This header file defines the ioctl() commands, constants and data-types that are needed to talk to read monitoring data, reload the firmware, and control the front-panel LEDs of the CCB, via the GPIO-104 general-purpose-I/O board.

- /usr/local/include/ccbtestclient.h

  This header file provides function-prototypes, data-type definitions and macros for use by programs that link with the libccbtestclient.so library.

### 3.3.3 The CCB configuration files that are installed

The installation procedure installs the following configuration files. See the ccb_network_interface document for more details regarding their contents.

- /usr/local/etc/ccb_authorized_ips

  This file contains a list of the IP addresses of computers that are authorized to connect to the CCB server.

- /usr/local/etc/ccb_ip_addresses

  This file contains a table that associates CCB cable-IDs with IP addresses, netmasks and gateway addresses. This is used at boot time to determine which IP address the CCB computer should be assigned, according to the cable-ID that is read through the GPIO board.

### 3.3.4 The CCB programs that are installed

The installation procedure installs the following programs.

- /usr/local/bin/ccbserver

  This is the CCB server program. On the behalf of the CCB manager, or diagnostic programs, it controls the CCB firmware via the CCB EPP driver, reads back integrated and dump-mode data from the firmware, via the CCB USB driver, and reads

monitoring data via the GPIO driver. It also provides an optional simulation of the CCB hardware and firmware, that can be selected in place of the real hardware, for testing the CCB Manager. This program is started at boot time, by the ccbserver system-V inititialization script, described later, and thereafter runs continuously, as a background daemon.

- /usr/local/bin/ccb_monitor_status

  This program interacts with the CCB GPIO driver to read back monitoring data, and control the front-panel system-status LEDs of the CCB. By default it samples the monitoring data once per second, and updates the LEDs immediately afterward, according to the values that were read.

- /usr/local/bin/ccb_demo_client

  This program was originally written to enable testing of the CCB server, before the manager was written. However, since it runs standalone of the CCB control software, and provides a lower level interface that is more oriented to engineering than that of the CCB Manager, it is also very convenient for debugging CCB hardware problems in the lab.

- /usr/local/bin/ccb_display_dump

  When the CCB Manager or the ccb_demo_client program initiate a dump-scan, this program can be connected to the CCB server's dump-mode port, to display the resulting dump-mode data textually.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_dummy_client

  This is another program that was written to test the CCB server. It simply connects to the server on the specified computer, switches the server into simulation mode, configures the CCB, starts a normal integration-scan, and reports all of the data, monitoring and log messages that are received back from the server. It remains useful for quick tests of the CCB server and the CCB communications libraries.

- /usr/local/bin/ccb_sequence_leds

  This program was intended to be run at boot time, but in the end wasn't used for this, since it would have slowed down the boot process too much. It lights one LED at a time of the CCB front-panel LEDs that can be controlled by the GPIO board, in a sequence that moves from top to bottom, and left to right. Although it isn't used by the boot procedure, it can still be run by hand, at any time, to check that all of the LEDs are functioning. Beware, however, that if the ccb_monitor_status program is

also running, it will also set the LEDs each second, during this sequence, leading to hard-to-interpret results. Thus, if the ccb_monitor_status program is running, it may be a good idea to suspend it temporarily, while this program runs, by typing:

```
pkill -f -TSTP ccb_monitor_status
ccb_sequence_leds
pkill -f -CONT ccb_monitor_status
```

Note that the above -f argument to the pkill command is necessary in this case, because otherwise the process name that is matched is limited to 15 characters, which is shorter than the name of the ccb_monitor_status program.

- /usr/local/bin/ccb_fake_samples

  This informational program can be invoked to display the expected values of dump-mode samples when the CCB is configured to substitute pseudo-random samples for the real ADC samples. It isn't used for anything by the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_fake_integ

  This informational program can be invoked, to display the expected values of integrations returned by the CCB, when the CCB is configured to substitute pseudo-random samples for the real ADC samples, according to the specified scan-configuration. It isn't used for anything by the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_test_dc_response

  This diagnostic program can be called upon to control the CCB and read back data, to check the DC response of the ADCs as a function of input voltage. It isn't used for anything by the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_test_fake_integ

  This diagnostic program can be used to control the CCB, and read back data to check that the CCB integrated data is correct when fake samples are being substituted for real ADC samples. It isn't used by anything else in the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_test_fake_samples

  This diagnostic program can be used to control the CCB, and read back dump-mode data to check that dump-mode samples are correct when fake samples are being substituted for real ADC samples. It isn't used by anything else in the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_test_phase_timing

  This diagnostic program can be used to control the CCB, and read back dump-mode data to investigate the time responses of the phase-switches. It isn't used by anything else in the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_test_adc_delay

  This diagnostic program can be used to control the CCB, and read back dump-mode data, to determine the value of the ADC clock-delay, that yields the best signal-to-noise ratio in all channels. It isn't used by anything else in the CCB.

  See the ccb_diagnostic_programs document for more details.

- /usr/local/bin/ccb_test_sample_stats

  This diagnostic program can be used to control the CCB, and read back dump-mode data, to determine the mean value and the noise around the mean, of a dump-mode frame of samples. It isn't used by anything else in the CCB.

  See the ccb_diagnostic_programs document for more details.

### 3.3.5 The CCB scripts that are installed

- /usr/local/bin/ccb_assign_ip

  This script is invoked by the init_ccbipaddr boot-time initialization script below. It temporarily loads the GPIO-board device-driver, uses this to query the cable ID of the CCB, and then consults the previously mentioned /usr/local/etc/ccb_ip_addresses file to deduce the corresponding network configuration parameters. The parameters that it looks up are the IP address for the computer, the netmask to apply to this, and the IP address of the gateway router of the local network. These values are printed to the standard output of the script, in the form of assignments to the following three environment variables:

| | |
|---|---|
| CCB_IPADDR | The numeric IP address to assign to the CCB computer. |
| CCB_NETMASK | The TCP/IP netmask associated with the above address. |
| CCB_GATEWAY | The address of the gateway of the parent network of the above address. |

From a Bourne-shell (or bash) shell, the output of the command is thus designed to be evaluated, as follows.

```
CCB_IP_INFO=`/usr/local/bin/ccb_assign_ip`
RETVAL=$?
if [ $RETVAL -eq 0 ]; then
  eval $CCB_IP_INFO
fi
```

Note that if either no cable is plugged in, or the GPIO card isn't installed, then the IP address associated with a cable ID of 0 is used. This is the ID of the lab cable. Thus the CCB will attempt to assign itself the ccblab.gb.nrao.edu IP address.

- /usr/local/bin/init_ccbipaddr

  This script is a system-V boot-time initialization script, derived from the template initialization-script that Fedora provides. Although the file resides in the /usr/local/bin/ directory, a symbolic link to this file, called ccbipaddr, is included in the normal /etc/init.d/ directory, as will be described later. During the boot process this script is called upon to configure the IP address, before networking is started. It is then subsequently invoked when the computer is shutdown, after networking has been shutdown.

  The init_ccbipaddr script uses the previously described ccb_assign_ip script to determine the network configuration parameters that correspond to the cable that is plugged into the front panel of the CCB, and writes these to the configuration file for the eth0 ethernet interface. This file is named:

  ```
  /etc/sysconfig/network-scripts/ifcfg-eth0
  ```

  Thus, when the normal networking initialization-script subsequently runs, the modified contents of this file are used to configure the eth0 interface. The idea behind this is that the computer should take on an identity that is associated with the cable of a particular receiver or lab hook-up, such that regardless of which CCB is connected to a given receiver, the CCB Manager will know which IP address to use to control that receiver.

- /usr/local/bin/init_ccbserver

  This is a another system-V boot-time initialization script. It is normally accessed through a symbolic-link, called ccbserver, in the /etc/init.d/ directory. This script, which runs at boot-time after networking has been started, loads the CCB device-drivers and starts the CCB daemons. Subsequently, when the computer is being shut down, it terminates the CCB daemons and unloads the CCB device-drivers.

- /usr/local/bin/load_driver

  This script is used to load one or more device drivers at a time.

  When invoked with a non-CCB device-driver as an argument, then it simply delegates the task of loading that driver to the standard /sbin/modprobe command. Otherwise, it locates the CCB device-driver file, loads this, using the /sbin/insmod command, and then creates the associated device file in /dev/, giving this the same name as the device-driver.

  If a CCB device-driver is specified with an explicit pathname prefix, then the specified file is loaded. Otherwise the script looks for the file in /usr/local/lib/modules/, where all of the CCB device drivers are installed.

  As previously described, sudo is configured to allow the ccb account to run this script, as though it were root. In order that the user not have to remember to run it through sudo, to take advantage of this, the script re-runs itself via sudo, if it finds that the user ID of its process is not that of the root user.

- /usr/local/bin/unload_driver

  This script is used to unload device drivers. The specified driver, or drivers, are unloaded by calling the standard Linux /sbin/rmmod command. Additionally, when a driver is seen to be a CCB device driver, then the device file of that driver in the /dev/ directory is also removed.

  As previously described, sudo is configured to allow the ccb account to run this script, as though it were root. In order that the user not have to remember to run it through sudo, to take advantage of this, the script re-runs itself via sudo, if it finds that the user ID of its process is not that of the root user.

- /usr/local/bin/mkramboot

  This script creates the /boot/custom_initrd script. This script contains the root file-system of the RAM-based version of Linux, that is used for hard-disk maintenance tasks. It also modifies the boot-loader configuration file, /etc/grub.conf to include the option of booting into RAM. This script is described in a bit more detail in section 3.5.

- /usr/local/bin/ccb_backup_computer

  This script is used to perform a network backup of the /boot and / filesystems, while booted into the RAM-based maintenance OS. It is described in section 5.1.

- /usr/local/bin/ccb_restore_backup

  This script is used to update the /boot and / filesystems, by performing network restorations of previously made backups, while booted into the RAM-based maintenance OS. It is described in section 5.2.

### 3.3.6 The CCB device-drivers that are installed

The following device-drivers are installed by the CCB installation script.

- /usr/local/lib/modules/ccbepp.ko

  This device-driver is used by the CCB server to control the CCB firmware via the computer's EPP-enabled parallel-port.

- /usr/local/lib/modules/ccbgpio.ko

  This device-driver uses the GPIO card to read back monitoring information from the CCB, reload the CCB firmware, and control some of the front-panel status and configuration LEDs of the CCB.

- /usr/local/lib/modules/ftdi_pio.ko

  This device-driver receives data from the CCB firmware, via an FT245BM USB module. Although not currently used by the CCB, it can also be used to send data to the USB module.

## 3.4 Configuring the automatic startup and shutdown of the CCB software

As already documented above, two system-V initialization scripts are installed in the /usr/local/bin/ directory. Symbolic links were thus created to these files from the system-V initialization directory. This was done as follows.

```
cd /etc/init.d
ln -s /usr/local/bin/init_ccbserver ccbserver
ln -s /usr/local/bin/init_ccbipaddr ccbipaddr
```

Each of these files starts with a comment section which includes some specially formed comments that are read by the /sbin/chkconfig command. One of these comment-lines tells chkconfig in which levels to start and stop the corresponding service, and at what time, in relation to other services within that level. Thus when chkconfig is called upon to add these services, it creates start and stop links to these files in the initialization directories that correspond to these levels (see "man chkconfig" for details). This was done as follows:

```
cd /etc/init.d
/sbin/chkconfig --add ccbserver
/sbin/chkconfig ccbserver on
/sbin/chkconfig --add ccbipaddr
/sbin/chkconfig ccbipaddr on
```

## 3.5 Creating a RAM-bootable maintenance OS

To allow maintenance to be performed on the root filesystem of the hard-disk, without having to disassemble the CCB computer to boot it from an external CDROM drive, a way was needed to boot the computer entirely into RAM, from an image in the /boot partition. To understand how this works, consider how Linux normally boots. In general there are two images that are loaded into RAM by the boot-loader. One is an image of the kernel, which runs entirely in RAM, once loaded. The other, usually called initrd, is an image of a minimal root-filesystem, which is loaded into a RAM-disk by the kernel, when the kernel starts. This root-filesystem generally just contains the device-drivers that are needed to mount the hard-disks, a minimal shell, and a script called /linuxrc, which is run by the kernel. This script normally loads the device-drivers that are needed to mount the hard disks, mounts the root file-system on the disk, tells the kernel to switch to using the disk-based root-filesystem, and finally runs the /sbin/init program from the hard-disk. The /sbin/init process then runs the system-V initialization procedures that initiate Linux.

Thus to instead create a version of Linux that boots into RAM, and has a RAM-based root filesystem, instead of the one on the disk, one simply needs to make a custom initrd script, whose minimal root file-system has been enlarged to contain all of the utilities that one needs to run Linux, and a /linuxrc script that starts /sbin/init from the RAM-based root file-system, rather than mounting and using the root file-system on the hard-disk. Thus, once the kernel image, and the custom initrd image have been loaded from the /boot partition of the disk, the disk is unused by Linux. This allows programs like restore, fsck etc, to be used to update or fix the file-systems on the disk, while they aren't being used.

In order to be able to perform network backups and restores of the hard-disk, using this system, the RAM-based root file-system needs to contain everything that is needed to support networking and ssh. This requires that a lot of utilities, scripts and libraries be present in the root file-system. To make this as robust as possible, it was decided to clone parts of the normal hard-disk based root-filesystem, rather than write custom networking scripts, that could need future maintenance. Much of the system-V initialization process needed to be cloned, in order to start up networking. Although determining which files were needed, required a lot of hands-on reading of scripts, and some trial and error, many parts, like figuring out which shared libraries were needed by selected programs, regenerating chains of symbolic links etc, could be automated.

Thus a script was written that would create the custom initrd image, and create a menu

entry for it in the boot loader's configuration file.

This was run as follows:

```
mkramboot -exes '/usr/local/bin/ccb_backup_computer /usr/local/bin/ccb_restore_backup'
```

This created a file called /boot/custom_initrd, containing the custom root file-system, and modified the /etc/grub.conf file, to list the ram-booting option. Since the mkramboot script was written to be generally usable on any system, not just the CCB, the CCB backup and restoration scripts weren't included, by default in this script. Hence the above -exes argument, which tells the script about any extra executables that it should install.

## 3.6 The final microdrive usage statistics

After everything had been installed, and had been running for a few weeks, the disk usage was checked as follows, and listed here for future reference:

```
$ df -k
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hdc3              1703964   1148272    467740  72% /
/dev/hdc1               101086     12862     83005  14% /boot
/dev/shm                241640         0    241640   0% /dev/shm
$
```

## 3.7 Taking an image of the microdrive

Once the computer had been configured and all of the CCB software had been installed, a backup-image was taken of the microdrive. This was done by removing the microdrive from the flash-card carrier of the computer, and placing it in a flash-card reader on a desktop computer. After identifying the corresponding device, and assigning its device-file to the MICRODRIVE environment variable (see section 2.4), the following command was used to make an image of the microdrive.

```
dd if=$MICRODRIVE of=microdrive_image
```

This wrote the image to a file, in the current directory, called microdrive_image. This ended up being a 2GB file. Since this included both used and unused blocks of the microdrive, the file could then be usefully compressed.

```
gzip microdrive_image
```

This resulted in a file, called microdrive_image.gz, which was just 600MB in size.

Note that in retrospect the compression could have been done on-the-fly. This would have avoided the need to have enough space on the local disk for both the uncompressed and compressed images. This would have been done with the following statement.

```
dd if=$MICRODRIVE | gzip > microdrive_image.gz
```

## 3.8 Cloning the microdrive

To avoid having to repeat the installation procedure for the second CCB computer, the image of the microdrive that was taken in the previous section, was copied to the second CCB microdrive. This was possible because the two microdrives are of the same model, and thus have identical disk-geometries. The second microdrive was placed in the flash-card reader on a desktop PC, and after identifying the corresponding device, and assigning its device-file to the MICRODRIVE environment variable (see section 2.4), the following command was used to copy the image that was made in the previous section, to the microdrive.

```
gunzip -c microdrive_image.gz | dd of=$MICRODRIVE
```

# Chapter 4

# Performing maintenance on the microdrive

Traditionally, to perform maintenance on the hard-disk that contains the root file-system of a computer, one plugs in a CDROM drive, and boots from a rescue CD. However, since it is necessary to take a CCB into the lab and take it apart in order to plug in a CDROM drive, an alternative way has been provided for performing maintenance. Since this involves booting from an image in the /boot partition of the microdrive, this won't work if the /boot partition is seriously damaged. However it can be used for routine tasks, such as backing up and restoring the root file-system, and for fixing the root file-system with fsck.

The maintenance OS boots itself entirely into RAM, in order that the disk not be used at all, once the OS has booted. This is implemented by embedding a minimal root file-system in the initial RAM-disk image (initrd) that the boot-loader loads, and arranging that Linux continue to run from this, rather than switching to the disk-based root file-system. Although this file-system is only 16MB in size, it includes most simple Linux command-line programs, plus maintenance programs, such as dump, restore, fsck, vi, more etc..., and supports networking. It doesn't include any graphical interfaces, however, since that would make the image too big to fit in the /boot partition, and in RAM.

To boot into this OS, it is necessary to first log into the CCB as root from the serial console of the CCB, and type:

```
/sbin/reboot
```

Now wait for Linux to shut itself down, for the computer to perform its power-on self-test, and for the boot-loader to present a menu of available operating-systems. When the boot-loader menu is presented, press the down-arrow key before its 3 second countdown expires. This will cancel the countdown and thus allow you to carefully use the down-arrow key to move the cursor down to the menu entry that says:

```
Boot Into RAM
```

Once the cursor is over this entry, press return to boot the computer into RAM. This will eventually present a login prompt, at which you should log in as root, using the usual CCB password.

By default, the hard disks aren't mounted. However they do have entries in /etc/fstab, such that they can be mounted as follows:

```
mount /disk
mount /disk/boot
```

This mounts the disk-based root file-system at /disk/ in the RAM-based root file-system, then mounts the disk-based /boot file-system at /disk/boot/. Thus, for example, after doing this, the CCB home directory on the disk-based root file-system, could be listed by typing:

```
ls /disk/home/ccb
```

Once you have finished whatever maintenance was to be performed, either power-cycle the computer, or reboot it by typing:

```
reboot
```

Then simply allow the computer to boot itself, without interacting with the boot-loader's menu. It will thus boot itself into the default, disk-based OS.

# Chapter 5

# Backup, recovery and mirroring of the CCB microdrives

Whereas a raw image of the microdrive was used to mirror the initial CCB computer installation between CCB microdrives, doing this would be a liability for long term backups, because raw disk images can only be restored onto disks with identical geometries. In the future it may be necessary to replace one or more of the CCB microdrives, after head crashes, and it is unlikely that a microdrive with the same geometry will be available for long.

A better method is to use the Linux dump and restore programs. Ideally one would make a full backup of the CCB, which would be kept indefinitely, and then make periodic incremental backups relative to this backup. This would provide a stable initial backup to revert to if needed, plus an archive of small incremental backups that could be used to update a CCB computer to a given date. Unfortunately, after much time wasted elaborating this scheme, the following clause was found at the end of the manual page for the restore program.

> "A level 0 dump must be done after a full restore. Because restore runs in user code, it has no control over inode allocation; thus a full dump must be done to get a new set of directories reflecting the new inode numbering, even though the content of the files is unchanged."

This is a show-stopper, for the following reason. Imagine two CCB computers, A and B. Further imagine that we were to make a full backup, using dump, on computer A, and then restore this onto computer B. Subsequently, if some changes were made to the files on computer B, then according to the above statement, we wouldn't be able to make an incremental backup of these changes.

Thus the only obvious option that remains, is to always perform full backups, using dump and restore, whenever a CCB computer is changed. Unfortunately, in order to allow reversion to an earlier state, this will require us to keep an archive of full-backups of the CCB.

## 5.1 The ccb_backup_computer command

The ccb_backup_computer script creates full backups of the two file-systems of a given CCB computer, in a given directory of a specified remote computer. It does this by piping the output of the Linux dump command through ssh.

The script is designed to be invoked after booting the CCB into its optional RAM-based OS (see section 4), and it will complain if this isn't the case. It is invoked as follows:

ccb_backup_computer *username@hostname directory*

Where, the *username@hostname* argument specifies the account name and host-name of the computer on which the backup files should be stored, and the *directory* argument specifies the directory that the files should be placed in, on that computer.

The steps that the ccb_backup_computer script performs, are as follows:

1. Complain and abort if the operating-system isn't running from the RAM-based OS.

2. If mounted, unmount the CCB's normal disk-based file-systems.

3. Run the dump command on the /disk/boot file-system, and pipe the compressed output of the dump command, via ssh to the specified location of the backup file, on the specified remote computer. Note that this prompts for the password of the account on the remote computer. The resulting file is called:

   ccb_boot_backup_*yyyymmdd*.gz

   Where *yyyymmdd* is the date of the backup. Note that sorting such filenames into ascending alpha-numeric order, results in a date-ordered list. The ccb_restore_backup command relies on this.

4. Run the dump command on the root file-system, and pipe the output of the dump command, via ssh to the specified location of the backup file, on the remote computer. This again prompts for the password of the account on the remote computer. The resulting file is called:

   ccb_root_backup_*yyyymmdd*.gz

5. Report that the backup operation completed successfully.

Note that this script does not update the /etc/dumpdates file. This file is irrelevant, because we aren't planning to perform any incremental backups.

## 5.2 The ccb_restore_backup command

The ccb_restore_backup program replaces the contents of a microdrive with those of the most recent backup files on a remote computer. To do this, it searches the backup directory of the specified remote computer, for the boot and root backup files that have the most recent dates in their filenames.

This script is designed to be invoked after booting the CCB into its optional RAM-based OS (see section 4), and it will complain if this isn't the case. It is invoked as follows:

ccb_restore_backup *username@hostname directory*

Where, the *username@hostname* argument specifies the account name and host-name of the computer on which the backup files should be stored, and the *directory* argument specifies the directory that the files should be placed in, on that computer.

The steps that the ccb_restore_backup script performs, are as follows:

1. Complain and abort if the operating-system isn't running from the RAM-based OS.
2. Use ssh to get the names of the most recent backups of the root and boot file-systems from the specified directory on the remote host computer.

   Note that this relies on the fact that the ccb_backup_computer script composes filenames for the dumps which, when sorted alpha-numerically, are listed in increasing order of date.

   Also note that ssh prompts for the password of the remote user-account at this point.
3. If no backup files are found for either of the file-systems, complain, then abort the script.
4. Mount the disk-based root file-system at /disk, read/write.
5. Change directory to /disk
6. Use ssh to pipe the newest backup file of the / file-system into the restore command. Note that this prompts for the password of the user-account on the remote computer.
7. Mount the disk-based /boot file-system at /disk/boot, read/write.
8. Change directory to /disk/boot
9. Use ssh to pipe the newest backup file of the /boot file-system into the restore command. Note that this again prompts for the password of the user-account on the remote computer.
10. Unmount the /disk and /disk/boot file-systems.
11. Report that the restore operation completed successfully.

## 5.3 Preparing a replacement microdrive

This section describes the steps needed to prepare a newly acquired microdrive for use in the CCB computers.

### 5.3.1 Partitioning a replacement microdrive

Before the previously described backups of the CCB file-systems can be restored onto a new microdrive, the microdrive needs to be partitioned to have the following primary file-system partitions.

| Partition | Mount point | Size | Type | Bootable? |
|---|---|---|---|---|
| 1 | /boot | 100MB | Linux Native | Yes |
| 2 | swap | 128MB | Linux Swap | No |
| 3 | / | All remaining space | Linux Native | No |
| 4 | *(unused)* | 0 | – | No |

On a 2GB microdrive, the root partition (*ie.*/) ends up having about 1.7GB.

To partition a new microdrive, first place it in a flash-card reader on a desktop PC or in the flash-card carrier of a CCB computer, booted from a rescue CD or live CD, and assign the corresponding device file-name of the microdrive, to the MICRODRIVE environment variable (see section 2.4). Then either use the /sbin/fdisk program to enter the above partitioning information interactively, or, use the non-interactive /sbin/sfdisk command, and carefully type the following:

```
/sbin/sfdisk $MICRODRIVE -O /tmp/saved_partitions
/sbin/sfdisk -uM $MICRODRIVE << EOF
,100,83,*
,128,82,-
,,83,-
;
EOF
```

Be very careful to type this in exactly as specified. To check that the result makes sense, type:

```
/sbin/sfdisk $MICRODRIVE -uM -l
```

This will list the partitions on the microdrive. Compare this list against the table given above. On the original microdrive, partitioned by the Fedora installation program, the output of this command was:

```
# /sbin/sfdisk /dev/hdc -l -uM

Disk /dev/hdc: 3968 cylinders, 16 heads, 63 sectors/track
Warning: The partition table looks like it was made
  for C/H/S=*/255/63 (instead of 3968/16/63).
For this listing I'll assume that geometry.
Units = mebibytes of 1048576 bytes, blocks of 1024 bytes, counting from 0

   Device Boot Start   End    MiB    #blocks   Id  System
/dev/hdc1   *     0+   101-   102-    104391   83  Linux
/dev/hdc2       101+   227-   126-    128520   82  Linux swap / Solaris
/dev/hdc3       227+  1945-  1718-   1759117+  83  Linux
/dev/hdc4         0      -      0          0    0  Empty
```

Note that although the number of cylinders, heads and sectors/track, listed for a different model of microdrive, may be significantly different from this, the only significant difference in the partition table itself should be the size of the third partition, which will differ if the replacement microdrive isn't a 2GB drive.

Also run the following command to perform consistency checks on the partition table.

```
/sbin/sfdisk $MICRODRIVE -V
```

On the original microdrive, partitioned by the Fedora installation program, this command yielded the following output:

```
[root@ccb1cm ccb]# /sbin/sfdisk $MICRODRIVE -V
Warning: partition 1 does not end at a cylinder boundary
[root@ccb1cm ccb]#
```

Since this refered to the partition table that was generated by Fedora's installation procedure, and there were no complaints at that time, this particular warning appears to be irrelevant. If a more serious error is revealed with respect to a newly created partition table, then the partition table that it displaced, can be restored from the precautionary file that was written above, by typing:

```
/sbin/sfdisk $MICRODRIVE -I /tmp/saved_partitions
```

A new attempt at partitioning the drive can then be made.

### 5.3.2 Creating file-systems on a replacement microdrive

Once the microdrive has been partitioned successfully, ext3 file-systems should be created in the /boot and / partitions, and the swap partition should be initialized, as follows:

```
/sbin/mke2fs -j ${MICRODRIVE}1
/sbin/mkswap ${MICRODRIVE}2
/sbin/mke2fs -j ${MICRODRIVE}3
```

## 5.4 Restoring the backed-up CCB file-systems onto a new microdrive

Having partitioned and created file-systems on a new hard-disk, as described above, and with the microdrive still in either a flash-card reader on a PC, or in the CCB, booted from a CD, restore the / directory as follows:

```
mkdir /mnt/root
mount -t ext3 ${MICRODRIVE}1 /mnt/root
cd /mnt/root
ssh user@host dd /wherever/ccb_root_backup_yyyymmdd.gz | restore -rf -
```

Where *user* is the user-name under which the backup files are installed on the computer named by *host*, and */wherever* denotes the directory in which the backup files are stored on that computer. *yyyymmdd* should be replaced with the date of the most recently made backup. You can determine this by simply listing the files in the backup directory, and noting the date embedded in the name of the final file that is listed.

Similarly, once this has completed, do the same for the root directory, as follows:

```
mount -t ext3 ${MICRODRIVE}3 /mnt/root/boot
cd /mnt/root/boot
ssh user@host dd /wherever/ccb_boot_backup_yyyymmdd.gz | restore -rf -
```

### 5.4.1 Installing the GRUB boot-loader on a new microdrive

Once Linux and the CCB software have been installed on the new microdrive, the remaining task is to install the GRUB boot-loader in master-boot-record (MBR) of the microdrive. The very basic boot-loader that comes with the microdrive needs to be replaced, because it lacks CCB-specific configuration requirements, such as console redirection of the boot-loader shell.

Unfortunately, since the GRUB program refuses to install itself on a disk that the BIOS doesn't know about, this task can not be performed while the microdrive is in a flash-card reader on a workstation PC. Instead the microdrive must be installed in a CCB computer, where the BIOS will see it as a fixed hard-disk. Having done this, plug a USB CDROM drive into the CCB computer, and boot the CCB computer using a Linux live-CD or a Fedora rescue-CD. Then assign the device-file of the microdrive to the MICRODRIVE environment variable (see section 2.4), and type:

```
mkdir /mnt/root
mount -t ext3 ${MICRODRIVE}3 /mnt/root
mount -t ext3 ${MICRODRIVE}1 /mnt/root/boot
/usr/sbin/chroot /mnt/root /sbin/grub-install --root-directory=/boot $MICRODRIVE
umount /mnt/root/boot
umount /mnt/root
rmdir /mnt/root
```

# Chapter 6

# Updating the CCB progams and drivers

If the source code of the CCB programs and drivers, is modified, the updated utilities can be installed as follows. Start by logging in to the computer on which the updated source-code hierarchy is found, and cd to the directory in which the top-level CCB directory of this hierarchy resides. Then, to update the source-code on a CCB computer in the lab, type:

```
rsync -rpltcz --delete -e ssh --exclude /firmware/ --exclude /doc/ --exclude '*~' CCB/. ccb@ccblab:CCB/.
```

Having done the above, log in to the CCB computer, using the ccb user-account, and type:

```
cd ~/build
make distclean
./configure
make
```

This will compile the updated utilities. To install the updated utilities, type the following, and enter the root password, when prompted.

```
/bin/su -c 'make install'
```

To run the updated programs, either reboot the CCB computer, or restart the CCB server, by typing:

```
/bin/su -c '/sbin/service ccbserver restart'
```

# Chapter 7

# Controlling the CCB startup procedures

## 7.1 Starting and stopping the CCB server manually

When a CCB computer boots, it automatically loads the CCB device drivers, and starts the CCB server and the CCB system-monitoring daemon. This is done by a system-V initialization script, whose service-name is ccbserver. When the CCB computer is subsequently shutdown, the same script shuts down these CCB programs, and unloads the CCB device-drivers.

The programs and drivers that the ccbserver service starts, can also temporarily be shut-down by typing:

```
/sbin/service ccbserver stop
```

and then subsequently restarted by typing:

```
/sbin/service ccbserver start
```

Alternatively, the two operations above can also be performed by typing:

```
/sbin/service ccbserver restart
```

Stopping the ccbserver service, using the first of the above commands, doesn't stop this service from being automatically restarted when the computer is next rebooted. To stop it from being started at boot time, the following command can be used.

```
/sbin/chkconfig ccbserver off
```

Subsequently, the following command can be used to re-enable boot-time startup.

```
/sbin/chkconfig ccbserver on
```

## 7.2 Starting and stopping the CCB IP-address allocation scheme

Whenever a CCB computer is rebooted, a custom system-V initialization service, called ccbipaddr, figures out which IP-address to assign to that computer. If the general-purpose I/O board of that CCB is present and functioning, and a standard CCB cable is plugged into its front-panel, then the CCB will select the IP-address that corresponds to the ID built into that cable. In this way, each CCB computer takes on the identity of the receiver that it is connected to. If no cable is connected, or the I/O board is missing or not working, then the IP-address assigned to the CCB in the lab, is adopted, as a fall-back.

Since the ccbipaddr service only runs for a short interval at boot-time, there is no point in using the /sbin/service command to temporarily turn it on or off, once the system has already booted. Doing so would have no effect.

However it may occasionally be useful to stop this service from running automatically at boot time, to prevent the automatic assignment of a CCB computer's IP address. In particular, if multiple CCBs were ever in the lab, and connected to the network, then there would be a conflict if all of them tried to use the lab, ccblab.gb.nrao.edu, address.

If such a case ever arises, do the following:

- Temporarily disconnect the ethernet cable.
- Boot the CCB computer. Note that when the computer tries to start up networking, it will try for a while, before giving up, and complaining that there is a network cable problem. Ignore this.
- Once the computer has booted, log in as root, and type:

  ```
  /sbin/chkconfig ccbipaddr off
  cd /etc/sysconfig/network-scripts
  cp -p original_ifcfg-eth0 ifcfg-eth0
  ```

  Note that the final line restores the DHCP network-startup script that was saved as part of the installation procedure.

- Plug the network cable into the computer.
- Reboot the computer. It should then use DHCP to get a unique address, and will continue to do so, every time that the CCB is rebooted.

To subsequently re-enable the normal boot-time CCB IP-address assignment scheme, simply type:

```
/sbin/chkconfig ccbipaddr on
```

Note that this service automatically rewrites the ifcfg-eth0 script. So there is no need to explicitly undo the restoration of the DHCP script that was done above.

On rebooting, the computer should thereafter revert to dynamically assigning its IP address, according to the instrument-cable that is plugged into it.

# Chapter 8

# Diagnosing ccbserver problems

## 8.1 Messages sent to the Linux logging facility

At boot-time the ccbserver program is started as a background daemon. In this mode, since the program has no terminal to write diagnostic messages to, it writes messages to the Linux logging facility. Thus, when the manager is unable to connect to the CCB server program, the way to figure out what is going on, is to look at the local log-file on the CCB computer. This log file, which is also used by many other Linux daemons, is called:

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

In this file, all messages from the CCB server are prefixed by the date, the host-name of the CCB computer, and the word ccbserver:. Thus to see just the messages from the CCB server program, type the following:

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

This will show historically generated log messages from the CCB server. Alternatively, to see newly arriving messages, as they arrive, type:

```
grep ccbserver: /var/log/messages | tail -f
```

## 8.2 Checking that all necessary processes are running

There are two programs and three device drivers that should always be running when the CCB computer has been booted. Their existence can be checked by typing:

```
pgrep ccbserver
pgrep -f ccb_monitor_status
pgrep ccbgpio
pgrep ccbepp
pgrep ftdi_pio
```

Note that the -f flag is used to search for ccb_monitor_status, because its name is longer than the maximum that pgrep otherwise supports.

Each of the above pgrep commands should report a process ID number. If any of them don't, then this means that the corresponding process is not running.

The above programs and drivers are as follows:

| | |
|---|---|
| ccbserver | The CCB server program. |
| ccb_monitor_status | The CCB system-monitoring and LED-control program. |
| ccbgpio | The device driver that interacts with the general purpose I/O board. |
| ccbepp | The device driver that controls the master FPGA, via the EPP-enabled parallel port. |
| ftdi_pio | The device driver that receives radiometer data from the FTDI USB parallel-I/O chip on the master-FPGA board. |

If any of these processes or device drivers are not running, try restarting everything, by typing:

```
/sbin/service ccbserver restart
```

## 8.3 Looking at the network connections of the CCB

The CCB server watches for incoming connections on three TCP/IP ports. These are:

5324 The telemetry port.
5323 The control port.
5322 The dump-mode port.

To see whether the CCB server is listening for connections and whether any manager, or other client is connected to those connections, type the following:

```
ps -p -a | grep ccbserver
```

A typical result is the following:

```
tcp 0 0 *:5322                  *:*                     LISTEN      1562/ccbserver
tcp 0 0 *:5323                  *:*                     LISTEN      1562/ccbserver
tcp 0 0 *:5324                  *:*                     LISTEN      1562/ccbserver
tcp 0 0 ccb1cm.gbt.nrao.edu:5323 fire.gbt.nrao.edu:40269 ESTABLISHED 1562/ccbserver
tcp 0 0 ccb1cm.gbt.nrao.edu:5324 fire.gbt.nrao.edu:40270 ESTABLISHED 1562/ccbserver
```

In the above example, the first three lines show that the CCB server program is listening to its three server ports, as expected, and the last two lines show that a program running on fire.gbt.nrao.edu, is connected to both the telemetry and control ports of the CCB server. The fact that no dump-mode client is shown to be connected, is typical, since the manager doesn't attempt to connect to that port.

### 8.3.1 Packet analysis

If things really get desperate, the tcpdump program can be used to look at the raw data that are being sent and received. For example, if the ccb_demo_client program was connected, and one typed:

```
/usr/sbin/tcpdump -x -X -c 3 'tcp and port 5323 and tcp[tcpflags] & tcp-push != 0'
```

and then hit the "Request Status" button of the ccb_demo_client program, then one should see something like the following:

```
19:01:02.059794 fire.gbt.nrao.edu.1374 > ccb1cm.gbt.nrao.edu.5323: ...etc..
0x0000   4500 0040 f8ce 4000 4006 43e7 7f00 0001        E..@..@.@.C.....
0x0010   7f00 0001 055e 14cb cfe7 8280 cfd0 8e01        .....^..........
0x0020   8018 7fff dd72 0000 0101 080a 002a dc49        .....r.......*.I
0x0030   002a 73fc 0000 000c 0000 000c 0000 001f        .*s.............
19:01:02.061268 ccb1cm.gbt.nrao.edu.5323 > fire.gbt.nrao.edu.1374: ...etc..
0x0000   4500 0040 0d34 4000 4006 2f82 7f00 0001        E..@.4@.@./.....
0x0010   7f00 0001 14cb 055e cfd0 8e01 cfe7 828c        .......^........
0x0020   8018 7fff 7543 0000 0101 080a 002a dc49        ....uC.......*.I
0x0030   002a dc49 0000 000c 0000 0001 0000 0000        .*.I............
19:01:02.061357 ccb1cm.gbt.nrao.edu.5323 > fire.gbt.nrao.edu.1374: ...etc..
0x0000   4500 0044 0d35 4000 4006 2f7d 7f00 0001        E..D.5@.@./}....
0x0010   7f00 0001 14cb 055e cfd0 8e0d cfe7 828c        .......^........
0x0020   8018 7fff 750e 0000 0101 080a 002a dc4a        ....u........*.J
0x0030   002a dc49 0000 0010 0000 0002 0000 001f        .*.I............
0x0040   0000 0000                                      ....
```

This shows three packets, the first going from the manager, running on fire.gbt.nrao.edu, to the control-port of the CCB server, which is running on ccb1cm.gbt.nrao.edu. The second

and third packets are replies to this, both going from the CCB server to the manager. The starts of these packets are TCP-specific information that we don't care about. The actual CCB-specific parts of these packets, which start at byte 0x0034, of each packet, are the following.

| Direction | Length | Type | Message-specific data | |
|---|---|---|---|---|
| Manager → Server | 0000 000C | Request status<br>0000 000C | Command number<br>0000 001f | |
| Server → Manager | 0000 000C | CCB status reply<br>0000 0001 | CCB status<br>0000 0000 | |
| Server → Manager | 0000 0010 | Command status<br>0000 0002 | Command number<br>0000 001f | Completion status<br>0000 |

As shown, the first 4 bytes report the total number of bytes in the CCB part of the packet, expressed as a big-endian integer, and the, the second 4 bytes specify the type of the CCB message, again as a big-endian integer. The remaining bytes are the data carried by the messages. The message types are listed in the next section. By looking at these, we can see that the first of the above 3 messages is the "Request-status" command that was invoked by pressing the "Request Status" button of the ccb_demo_client. The second message, is a reply to this, containing the CCB status, and the third message is a command-status reply, which tells the manager whether or not the command completed successfully.

### Message types

The contents of the messages sent over the three communication links that the CCB server supports, are described in the ccb_network_interfaces document. Below, for convenience, just the types of the various messages are shown, in the same format as shown by the tcpdump command.

For commands being sent from the manager to the CCB server, over the control-link, the message types are encoded as follows.

Manager → Server control-link commands

| Message Type | Description |
|---|---|
| 0000 0000 | A phase-switch configuration command. |
| 0000 0001 | A cal-diode configuration command. |
| 0000 0002 | A timing configuration command. |
| 0000 0003 | A sampler configuration command. |
| 0000 0004 | A start-scan command. |
| 0000 0005 | A stop-scan command. |
| 0000 0006 | A dump-scan command. |
| 0000 0007 | A monitoring control command. |
| 0000 0008 | A telemetry-stream control command. |
| 0000 0009 | A log control command. |
| 0000 000A | A reset command. |
| 0000 000B | A ping command. |
| 0000 000C | Request the status of the CCB. |
| 0000 000D | Shutdown the computer. |
| 0000 000E | Reboot the computer. |
| 0000 000F | Load a particular device driver. |
| 0000 0010 | Set the DAC counts on the GPIO card. |

The contents of all of the above messages start with a 4-byte sequence number. This number, which is supplied by the manager, is unique from one command to the next, so that it can be matched against the command sequence-number that is contained in the command completion-status message that is subsequently sent back by the CCB server. The rest of the contents of these messages are described in the ccb_network_interfaces document.

For replies to control-commands sent by the Server to the manager, over the control-link, the message types are as follows:

Server → Manager control-link replies

| Message Type | Description |
|---|---|
| 0000 0000 | A reply to a ping command. |
| 0000 0001 | A reply to a request-status command. |
| 0000 0002 | A control-command acknowledgment reply. |

The contents of the above control-link reply messages are described in the ccb_network-_interfaces document.

For messages sent from the CCB server to the manager, over the telemetry link, the message types are as follows.

Server → Manager telemetry messages

| Message Type | Description |
|---|---|
| 0000 0001 | An integration data message |
| 0000 0002 | A monitoring data message |
| 0000 0003 | A log message |
| 0000 0004 | A reply to a ping command |

Finally, for the dump-mode link, packets are only sent from the CCB server to dump-mode clients, and there is only one type of message, whose contents are those of the CCBDumpFrame structure, which is described in the ccb_network_interfaces document.

Server → Dump-client messages

| Message Type | Description |
|---|---|
| 0000 00FF | A frame of dump-mode data |