The network interfaces of the CCB server

[Document number: A48001N002, revision 12]

Martin Shepherd
California Institute of Technology

November 2, 2005

This page intentionally left blank.

Abstract

This document describes the network interfaces that the CCB server presents to remote CCB client programs, such as the CCB manager, dump-mode display programs, and standalone diagnostic programs. These interfaces are presented in the form of public APIs, by a suite of shared libraries.

Contents

1 Introduction **10**

1.1 Points of interest to writers of the CCB Manager program 11

1.2 Points of interest to writers of the CCB server program 12

1.3 The organization of this manual 12

1.4 The three TCP/IP links opened by the CCB server 13

1.5 Connection establishment 13

1.6 Connection authentication 14

1.7 Initial configuration 14

1.8 Single threaded versus multi-threaded 14

1.9 Library usage caveats 16

1.10 Shared libraries and their versioning 17

2 Installation **18**

2.1 Getting the source code 18

2.2 The basics of installation 18

2.3 Compiling in a different directory 19

2.4 Specifying where files are installed 20

2.5 Generating this manual and other CCB documentation 20

2.6 Testing the libraries using the demonstration programs 22

2.6.1 ccb_dummy_client 24

2.7 Run-time configuration files 24

2.8 The ccb_authorized_ips configuration file 24

2.9 The assignment of log IDs 25

3 The common parts of the CCB server and client APIs **27**

3.1 The configuration of the CCB 27

3.1.1 The configuration of the phase switches 29
3.1.2 The configuration of the calibration diodes 32
3.1.3 The configuration of hardware timing parameters 34
3.1.4 The configuration of sampler control parameters 38
3.2 Textual configuration 39
3.2.1 ccb_parse_CCBConfig() 41
3.2.2 ccb_read_CCBConfig() 41
3.2.3 ccb_print_CCBConfig() 42
3.2.4 ccb_write_CCBConfig() 42
3.2.5 Parsing and displaying phase-switch specifications 43
ccb_parse_CCBPhaseSwitches() 43
ccb_render_CCBPhaseSwitches() 43
3.2.6 Parsing and displaying cal-diode specifications 44
ccb_parse_CCBCalDiodes() 44
ccb_render_CCBCalDiodes() 44
3.3 Integration and scan timing information 45
3.3.1 Interval computations 46
ccb_scale_interval() 46
ccb_add_intervals() 47
ccb_subtract_interval() 47
ccb_compare_intervals() 47
ccb_zero_interval() 48
ccb_interval_is_zero() 48
ccb_clock_interval() 48
3.3.2 Cal-diode and phase-switch settling times 48
3.3.3 The number of phase-switch states per cycle 49
3.3.4 The physical duration of an integration period 50
3.3.5 The effective integration time 50
3.3.6 The duration of a scan 50
3.3.7 The number of integrations that fit within a time interval 51
3.3.8 The number of integrations in a calibration cycle 51
3.4 Timestamps 52
3.4.1 Zero-initializing a timestamp 53

3.4.2 Getting the current date and time 53
3.4.3 Comparing two timestamps 53
3.4.4 Computing the amount of time remaining until a given time 53
3.4.5 Adding a time-interval to a timestamp 54
3.4.6 Converting a time_t value to a CCBTimeStamp value 54
3.4.7 Converting a CCBTimeStamp value to a time_t value 54
3.4.8 Getting the clock-time from a timestamp 55
3.5 Pseudo-random fake samples 55
3.5.1 The values of integrated fake samples 55
3.5.2 The values of fake samples 56

4 The CCB manager communications API **59**
4.1 Include files 61
4.2 The CCB-client communications library 62
4.3 Creating the client resources needed to talk to a CCB server 62
4.4 Connecting to a CCB server 63
4.5 Disconnecting from a CCB server 64
4.6 Deleting a redundant CCBClientLink object 64
4.7 Requesting non-blocking I/O 65
4.8 ccb_client_communicate() – Perform client socket I/O 65
4.9 Client I/O multiplexing 66
4.9.1 Using the select() system call 66
4.9.2 Using the poll() system call 67
4.9.3 Third party event handlers 68
4.9.4 Using threads to multiplex I/O 69
4.10 Registering a command-error callback function 69
4.11 Outgoing CCB commands 71
4.11.1 Outgoing CCB control commands 71
ccb_queue_start_scan_cmd() – Queuing a start-scan command 72
ccb_queue_stop_scan_cmd() – Queuing a stop-scan command 73
ccb_queue_dump_scan_cmd() – Queuing a dump-scan command . . . 74
ccb_queue_load_driver_cmd() – Queuing a load-driver command . . . 76
ccb_queue_monitor_cmd() – Queuing a monitor command 76
ccb_queue_telemetry_cmd() – Queuing a telemetry command 76

ccb_queue_logger_cmd() – Queuing a logger command 77
ccb_queue_reset_cmd() – Queuing a reset command 77
ccb_queue_ping_cmd() – Queuing a ping command 78
ccb_queue_status_request_cmd() – Queuing a status-request command 79
ccb_queue_shutdown_cmd() – Queuing a shutdown command 79
ccb_queue_reboot_cmd() – Queuing a reboot command 79
ccb_queue_set_dacs_cmd() – Queuing a set-dacs command 80
4.12 Incoming control-link replies 82
ccb_status_reply_callback() – Routing status-request replies 82
4.13 Incoming telemetry messages 83
ccb_monitor_msg_callback() – Routing telemetry monitor-data messages . . 85
ccb_integ_msg_callback() – Routing telemetry integ-data messages 87
ccb_log_msg_callback() – Routing telemetry log-message messages 90
4.14 A TCL wrapper around the CCB client API 91

5 The CCB server communications API **100**
5.1 Include files 100
5.2 The CCB-server communications library 101
5.3 Creating the resources used to communicate with CCB managers 101
5.3.1 The CCB server's device-driver interface 102
5.4 Shutting down server communications 108
5.5 Server I/O multiplexing 109
5.6 Queuing replies to control commands 109
5.7 Queuing outgoing telemetry messages 109
5.7.1 Queuing outgoing monitor-data messages 110
5.7.2 Queuing outgoing integ-data messages 112
ccb_zero_integ_frame() 113
5.7.3 Queuing outgoing dump-frame messages 113
ccb_zero_dump_frame() 114
ccb_phase_switch_sequence() 114
5.7.4 Queuing outgoing log-message messages 115

6 Library internals **116**
6.1 The message translation layer 118

6.1.1 Message structure specification 118
6.1.2 Supported data-types within message structures 118
6.1.3 CCBNetMsg - The base-class of all messages 118
6.1.4 Some example message structures 119
6.1.5 CCBNetMsgMember – Message field descriptions 119
6.1.6 CCBNetMsgInfo – Individual message descriptions 120
6.2 The CCB interface layer 121
6.2.1 The message structures of outgoing control messages 122
CCBPhaseSwitchCmd – The phase-switching configuration command 123
CCBCalDiodeCmd – The calibration diode configuration command 124
CCBTimingCmd – The acquisition-timing configuration command 124
CCBSamplerCmd – The sampler configuration command 124
CCBStartScanCmd – The start-scan command 125
CCBStopScanCmd – The stop-scan command 125
CCBDumpScanCmd – The dump-scan command 125
CCBMonitorCmd – The monitor command 126
CCBTelemetryCmd – The telemetry command 126
CCBLoggerCmd – The logger command 126
CCBResetCmd – The reset command 126
CCBPingCmd – The ping command 127
CCBStatusRequestCmd – The status-request command 127
CCBShutdownCmd – The shutdown command 127
CCBRebootCmd – The reboot command 127
CCBLoadDriverCmd – The load-driver command 128
CCBSetDacsCmd – The set-dacs command 128
6.2.2 The message structures of incoming control-link replies 128
CCBCntrlPingReply – A reply to a ping command 129
CCBStatusReply – A reply to a status-request command 129
CCBCntrlCmdAck – An acknowledgment to a control command . 130
6.2.3 The message structures of incoming telemetry messages 130
CCBIntegMsg – Integration data messages 131
CCBMonitorMsg – Monitor data messages 131
CCBLogMsg – CCB log messages 132

CCBTelemPingReply – A reply to a ping command 133
6.3 Sending network messages 133
6.4 Receiving network messages 134

7 The CCB dump-data communications API 135
7.1 Functions in the libccbdump library 137
7.1.1 new_CCBDumpReader() 137
7.1.2 get_CCBDumpReader_fd() 137
7.1.3 ccb_read_dump_frame() 138
7.1.4 del_CCBDumpReader() 139

8 The CCB diagnostic communications API 140
8.1 Functions in the libccbtestclient library 140
8.1.1 new_CCBTestClient() 140
8.1.2 ccb_tst_connect_client() 141
8.1.3 ccb_tst_disconnect_client() 141
8.1.4 del_CCBTestClient() 142
8.1.5 ccb_tst_update_conf() 142
8.1.6 ccb_tst_start_scan() 143
8.1.7 ccb_tst_read_integ_frame() 143
8.1.8 ccb_tst_start_dump() 144
8.1.9 ccb_tst_read_dump_frame() 145
8.1.10 ccb_tst_mean_of_dump() 145
8.1.11 ccb_tst_get_config() 146
8.1.12 ccb_tst_set_dacs() 146

List of Figures

3.1 Example phase-switching cycles 30
3.2 The anatomy of a data-scan or intra-scan 45

6.1 The CCB communications stack 117

List of Tables

3.1 Textual configuration parameters and their values 40
3.2 The numeric labeling of phase-switch bins 56

4.1 The ID numbers of the CCB cables 87
4.2 The ordering of the integrated data 89

Chapter 1

Introduction

Data acquisition and control of the CCB hardware are performed by a CCB server process, which runs on the CCB embedded computer. This, in turn communicates with programs, such as the CCB manager, running on other computers, via TCP/IP network interfaces. A number of libraries are provided to facilitate this.

- libccbcommon

  This library contains facilities that are called upon by all of the other libraries, as well as being available for the use of CCB programs.

- libccbserverlink

  This library is used by the CCB server process to implement all of its server interfaces. This includes a server-module that receives commands from CCB manager programs, a server-module that sends integrated and monitoring data to CCB manager programs, and a server-module that sends dump samples to dump-mode clients.

- libccbclientlink

  This library is used by CCB manager programs, both to send commands to the CCB server, and to receive integrated and monitoring data from the CCB server.

- libccbdump

  This library is used by passive monitoring programs that wish to receive dump-mode data from the CCB server, while the CCB is being controlled by another program.

- libccbtestclient

This library is used by self-contained diagnostic programs that wish to start scans with specified configurations, and read back and examine the resulting frames of dump-mode or integrated data that these scans return.

This library is actually a simplifying layer that runs on top of the libccbdump, libccb-clientlink and libccbcommon libraries.

This manual is intended not only as a description of the public APIs available to the CCB manager and server programs, but also as documentation of the behavior of some of the library internals.

1.1 Points of interest to writers of the CCB Manager program

Since the developers of the CCB manager program don't need to know about the APIs used by the CCB server, or about the internals of the CCB libraries, they can completely ignore chapters 5 and 6. They can also ignore chapters 7 and 8, which describe the communications APIs used by passive dump-mode monitoring clients, and active diagnostic programs.

The remaining chapters are nonetheless very detailed, and without the benefit of top-down illustrations of how things go together, readers are at risk of not being able to see the wood for the trees. For this reason, the following high-level code examples are provided.

- **Using the C client API in a CCB manager program** . . . . Page 59

  This example illustrates the function calls that are required to implement a manager, and the order in which they are normally called. For the sake of example, it illustrates the use of a select() based event loop in the manager. As documented later, this is only one of many options that the manager has at its disposal. The reader can also refer to a fleshed-out version of this example, by looking at the C code of the included ccb_dummy_client program (see ccb_demo_client.c).

- **Using the Tcl version of the client API** . . . . Page 98

  This example provides a fully working illustration of using the Tcl wrapper library of the client C API. This wrapper was written to facilitate implementation of the GUI client demonstration program, ccb_demo_client, but because of its simplicity, it is potentially also useful for prototyping, experimentation, and testing.

The first-time reader is recommended to glance over these examples before immersing themselves in the documentation of the API.

1.2 Points of interest to writers of the CCB server program

The CCB server is currently the responsibility of the writer of this manual, so although full API documentation is provided for the benefit of a future maintainer, code examples aren't provided. To gain an understanding of the CCB server API, the reader can ignore chapters 4 and 6, which document the client communications API and the library internals, respectively.

1.3 The organization of this manual

The chapter following the one that you are currently reading, provides detailed instructions for downloading, installing, and testing the libraries and demonstration programs.

This is followed by a chapter which documents utility functions that are of use to all CCB programs, such as functions for setting and querying configuration parameters within CCB configuration objects, functions for generating and manipulating timestamps, and functions for computing the timing of integrations and scans. Of particular importance in this chapter are the descriptions of the CCB configuration parameters, and their effects on the behavior of the backend hardware.

The next chapter documents the public API provided by the library that implements the manager side of the communications interface. This is followed by a chapter that documents the public API provided by the library that implements the server side of the communications interface. The latter chapter can be ignored by those writing the manager program.

The next chapter documents the internals of the above two libraries, including the communications protocols that the libraries use to exchange messages with each other. This is probably only of interest to the maintainer of these libraries.

The next chapter describes a library that provides a passive interface for programs that wish to read dump-mode data, and the final chapter describes a library that is used to write remote diagnostic programs for testing the CCB.

After the final chapter, a page-index is provided of all functions, datatypes and macros in the public APIs of the two libraries. This is a very basic index, in that only the page number of the most important reference to each of the specified items is given.

The remaining sections of the introductory chapter that you are now reading provide an overview of various concepts that are needed to understand the remainder of the manual.

1.4 The three TCP/IP links opened by the CCB server

The CCB server process has three TCP/IP ports.

1. **The control port**

   This port is used by the manager end of the communications link, to send commands to the CCB server, and in some cases, receive replies to these commands from the CCB server. The CCB server never sends any unsolicited messages to the manager over this link, so it is effectively completely under the control of the manager.

   The TCP/IP port number of this service is parameterized by the following macro from ccbconstants.h.

   ```
   #define CCB_CONTROL_PORT 5323
   ```

2. **The telemetry port**

   This port is used by the server to send data to the manager. This includes integrated radiometer data, monitoring data and log messages. The CCB manager never sends messages to the server over this link, so this link is essentially a one-way link, entirely under the control of the CCB server. The manager does, however, tell the server what classes of data it expects to receive over this link, and at what frequency.

   The TCP/IP port number of this service is parameterized by the following macro from ccbconstants.h.

   ```
   #define CCB_TELEMETRY_PORT 5324
   ```

3. **The dump-mode port**

   This port is used by the server to send dump-mode data to remote diagnostic programs. The CCB manager does not currently connect to this port. When no client program is connected to this port, dump-mode data are simply discarded.

   The TCP/IP port number of this service is parameterized by the following macro from ccbconstants.h.

   ```
   #define CCB_DUMP_PORT 5322
   ```

1.5 Connection establishment

Connection establishment between the manager and the CCB server, is initiated by the ccb_client_connect() function, described later. This initiates TCP/IP connections to the control and telemetry server ports on a specified computer.

Similarly, passive dump-mode clients connect to the dump port of the server by calling the new_CCBDumpReader() function.

Finally, active diagnostic programs connect to all of the above ports, by calling the ccb_tst_connect_client() function, which in turn calls the ccb_client_connect() and new_CCBDumpReader() functions.

1.6 Connection authentication

For security reasons, the run-time configuration file of the CCB server includes a list of the numeric IP addresses of the computers that are allowed to connect to the CCB server. An asterisk in place of any of the numeric components of these addresses acts as a wild-card, so it is possible to configure access to all computers within a given sub-domain via a single entry.

If the connecting manager isn't connecting from one of these authorized IP addresses, or a new connection is attempted while a manager is already connected to the CCB server, the new connection is rejected. In the case of a manager already being connected, the rejected connection request is reported to the existing manager as a log message.

In practice, although the above scheme is useful for testing the CCB server on a machine outside of the observatory, both the observatory and the CCB embedded computer have their own firewalls, which are configured to reject all connections from machines outside the observatory.

1.7 Initial configuration

Whenever a manager establishes a new connection to a CCB server, the CCB server reinitializes the CCB hardware, sets all CCB configuration parameters to their power-on-defaults, and disables all telemetry, except log messages. The manager then has the option of overriding the server's default configuration parameters with its own, before sending a CCB telemetry command to enable the telemetry that it wants to receive.

1.8 Single threaded versus multi-threaded

Most of the discussions in this document assume that the client and server communications libraries are being used from a single thread in their host programs, and that I/O is multiplexed using select() or poll(), combined with non-blocking I/O. This is how the CCB server

is implemented. However an alternative strategy for the manager program would be to have reads from the control socket, writes to the control socket, and reads from the telemetry socket all be performed by different threads within a multi-threaded program, and for each of these threads to use blocking I/O. This is facilitated as follows.

- All functions in the manager side of the communications library can be called from multiple threads. In particular, if multiple threads call the function that performs I/O on the communications sockets, then one thread can be writing to the control socket, while another is reading from this socket, and a third is reading from the telemetry socket.
- The library uses no modifiable static data. All modifiable data structures are allocated from the heap.
- The library uses POSIX thread-safe interfaces where available and uses POSIX thread calls to control multi-threaded access to heap-data and other shared resources.
- Within the communications library, before a thread calls any of the manager's callback functions, it first releases any locks that it is holding. It then reacquires those locks when the callback returns. This avoids deadlocks that would otherwise result if a callback were to call another function in the library.

To take advantage of this aspect of the library, the manager must observe the following rules.

- The manager must not toggle the non-blocking attribute of the CCB sockets from one thread while other threads are sending or receiving data over those sockets.
- When the manager registers the callback functions that are to be invoked when messages are received from the control and telemetry connections, it is the manager's responsibility to ensure that these callbacks are thread safe. If the callback-data object that the manager registers along with a given callback function, is also used by another callback function that will be called by different thread, it is also the manager's responsibility to ensure that this object is accessed in a thread-safe manner in the two callbacks.

The server-side communications library is designed for a server program that does nothing more than act as a bridge between the server library and the CCB device driver. As such, unlike the manager API of the library, the server end of the library is designed to be driven by a simple select() based I/O event loop in a single-threaded server program.

1.9 Library usage caveats

The following general rules must be observed by the manager when calling functions in the public API of the communications library.

- None of the library functions are async-signal-safe, and should thus not be called from signal handlers.
- Some callback functions are passed pointers to arrays as arguments. These functions must assume that these pointers, and the arrays to which they point, become invalid as soon as the callback function returns. If longer term access to their contents is needed, the callback must make its own copy.

  For example, when log messages are sent to the manager by the log-message callback, the error message string is discarded and its array potentially reused for a different message as soon as the callback function returns.

Incoming messages are delivered to the manager via callback functions that the manager provides. The library header-files provide macros for declaring and prototyping these functions. For example, a typical callback definition macro might be the following,

```
#define CCB_EXAMPLE_CALLBACK_FN(fn) int (fn)(void *data, int x)
```

Now, say that the manager wanted to define a callback function of this type called my_callback(). Its function prototype would be written like:

```
static CCB_EXAMPLE_CALLBACK_FN(my_callback);
```

and its definition written like,

```
static CCB_EXAMPLE_CALLBACK_FN(my_callback)
{
  ...the body of the function...
}
```

It is recommended that these callback function macros be used, because if additions ever need to be made to the argument lists of any of the callback functions, a simple recompile of the manager will then automatically incorporate the new definitions.

1.10 Shared libraries and their versioning

The communications libraries are compiled as shared libraries under both Solaris and Linux. This brings the possibility of strict versioning support from the respective linkers, and the ability to restrict which symbols are exported to application programs, thus preventing the unsupported use of internal library functions. The versioning scheme implemented by the Linux and Solaris run-time linkers is documented at

`http://www.usenix.org/publications/library/proceedings/als2000/browndavid.html`

The basic idea is that libraries have three version numbers, a major number, a minor number and a micro number. These are used as follows:

- When a library update only involves modifications to the internal implementation of the library, without any changes being made to the public interface, the micro version number is incremented by 1. In this case an application can safely run against the new shared library without needing to be recompiled.
- When the existing public interface is augmented with the addition of new functions, without any changes being made to the interfaces of the existing public functions, the minor version number is incremented by one, and the micro version number is reset to zero. In this case a previously compiled application can run against the updated shared library, without needing to be recompiled, but will obviously need to be recompiled if it wishes to make use of any of the added features.
- When any aspect of the existing public interface is changed, the major version number is incremented by one, and the minor and micro version numbers are reset to zero. Since the new library isn't backwardly compatible with the previous one, the application needs to be recompiled before the run-time linker will allow it to use the new library version. This kind of update should be avoided if at all possible.

To enhance the capabilities of the Solaris and Linux run-time linkers, a map file is used when a shared library is created. This lists the symbols that were added in each new minor version of the library. This allows the run-time linker to check that all of the functions that the application actually uses, are provided in the current version of the shared library, even if the current shared library is older than the one that the application was originally linked against.

Configuration of the communication library makefiles to support this scheme are performed by a standard autoconf configure script which, if need be, can later be tailored to future operating systems. Modifications are performed by editing the file configure.in, which is heavily commented, then running the autoconf program to generate a configure script from this.

Chapter 2

Installation

2.1 Getting the source code

The latest version of the library code, plus this documentation can be downloaded from,

```
http://www.astro.caltech.edu/~mcs/GBT/ccb.tar.gz
```

To extract the contents of this tar file, type,

```
gunzip -c ccb.tar.gz | tar xf -
```

This will create a directory called CCB/ in the current directory. Within this directory the source code is collected under sub-directories of CCB/code, documentation is collected under the CCB/doc directory, and the CCB firmware and modelsim simulation scripts are found in the CCB/firmware directory. The remaining files in the CCB/ directory, are associated with the build procedure of the code and documentation.

2.2 The basics of installation

Although the code and documentation can be compiled in-place, within the top-level CCB sub-directory, it is recommended that instead you create a separate build directory, and compile from there. This keeps the source-code hierarchy clean of intermediate files and final binary files, and thus makes it easier to copy or mirror the code between different computers. You can also then simply delete the build directory, to get rid of intermediate

files, without affecting the files in the source-code hierarchy, or delete the CCB/ directory, without also deleting the build directory.

So, for example, a complete installation from scratch would look as follows:

```
gunzip -c ccb.tar.gz | tar xf -
mkdir build
cd build
../CCB/configure
make
make install
```

You could then optionally either delete the build directory, or type:

```
make distclean
```

to remove any files that the makefile and configure script generated. To just remove files that were built by the makefile, without also removing those files that were created by the configure script, you would instead type:

```
make clean
```

After doing this, you could then re-run make from scratch, without having to run the configure script again. This is a good way to ensure that everything that needs to be recompiled, is recompiled, after a change to a source file.

The provided autoconf configuration script, CCB/configure, currently only knows about Solaris and Linux, but its template script, CCB/configure.in is heavily commented to facilitate the addition of configurations for other operating systems. The only parameters of the configuration that need to be changed from one operating-system to the next, are those that refer to shared library creation and versioning.

2.3 Compiling in a different directory

As shown in the above example, compilation can be performed in a different directory from the one that contains the source code, simply by going to the chosen build-directory, and running the CCB/configure script from there, via a relative or absolute directory path to the configure script.

This works, because the makefile that the configure script generates, contains pathnames to each component that it needs.

2.4 Specifying where files are installed

By default, the libraries, demonstration programs, public include files and run-time configuration files are installed, respectively, in the lib/, bin/, include/ and etc/ subdirectories of the /usr/local/ directory. Via the following optional arguments passed to the configure script, these default locations can be overridden.

- **prefix=*pathname***

  This argument changes the choice of /usr/local for the directory in whose sub-directories the files are installed, to the specified directory ***pathname***.

- **libdir=*pathname***

  This argument changes the location where libraries are installed, to the specified directory ***pathname***.

- **bindir=*pathname***

  This argument changes the location where executables, such as the demonstration programs are installed, to the specified directory ***pathname***.

- **includedir=*pathname***

  This argument changes the location where the public include files of the CCB libraries are installed, to the specified directory ***pathname***.

- **sysconfdir=*pathname***

  This argument changes the location where the CCB run-time configuration files, such as the ccb_authorized_ips file, are installed, to the specified directory ***pathname***.

Note that if any of the installation directories don't already exist, then the make install command creates them.

2.5 Generating this manual and other CCB documentation

All of the CCB documentation can be found under the CCB/doc/. Each document, including the one that you are reading, has its own sub-directory under that directory, containing LaTeX source code and associated diagrams.

Having followed the directions in section 2.2, to create a build directory, go into the build directory that you created, and type one of the following:

- `make dvi`
  This builds dvi-format files of the documentation, which can then be read with the xdvi program.

- `make ps`
  This builds postscript versions of the documentation, ready for printing on a postscript printer, or viewing with a program like ghostview. As a side-effect, dvi-format files are also built.

- `make pdf`
  This builds PDF files for each of the documents, ready to be viewed with programs such as Adobe Acrobat. As a side-effect, dvi-format files are also built.

  If the configure script found the optional LaTeX hyperref package, the dvi and pdf versions of the manual include hypertext links. Beware that these links all point to the starts of sections or sub-sections, so a reference in the index to a particular page, actually links to the start of the section that contains that page.

To make just a particular document, simply run make with the name of the document that you want. For example, tpying:

```
make ccb_network_interface.pdf
```

would create a PDF file of the CCB Network Interface document, with the above name. The names of the PDF versions of all of the available documents are as follows:

- ccb_network_interface.pdf
  The document that you are currently reading.

- ccb_diagnostic_programs.pdf
  Documentation of a suite of CCB diagnostic programs.

- ccb_fpga_design.pdf
  Documentation of the CCB firmware.

- ccb_epp_driver.pdf
  Documentation of the EPP driver that controls the CCB firmware.

- ccb_gpio_driver.pdf
  Documentation of the I/O card driver that monitors the general status of the CCB, and drives the front-panel LEDs.

- ccb_external_hardware_interfaces.pdf
  Documentation of the goals and rational behind the eventual design of the external hardware interfaces of the CCB, intermixed with the description of an early potential implementation, of which only a few parts were actually used.

Note that there is no make install target for the documentation, because there isn't an standard for where such documentation should be installed, and autoconf doesn't provide a directory option for this, for use in configure scripts. Thus the PDF, DVI and PS files are simply left in the build directory, and should be copied by hand to their final destinations.

2.6 Testing the libraries using the demonstration programs

In addition to building the CCB client and CCB server libraries, the makefile also compiles, links and installs a simple, GUI-based, demonstration client program, that can be used for testing and controlling the CCB server program, in place of the CCB manager.

Since the installation paths of the shared libraries and the CCB configuration files are embedded within the executables, it is necessary to run the "make install" step before attempting to run the demonstration programs. Thus if you wish to test out the demonstration programs before performing the final installation, first perform the installation in a temporary place, then later recompile and reinstall in the final place. For example, to install under /home/mcs/tmp, one would do the following steps.

```
./configure --prefix=/home/mcs/tmp
make install
```

Later on, to perform the final installation in subdirectories of a different directory, one would repeat this, but replacing the /home/mcs/tmp in the above, with the path of the desired directory.

Having installed the CCB software, check that the ccb_authorized_ips file, which is by-default installed in the etc/ subdirectory of the chosen top-level installation directory, contains an entry that covers the IP address of the computer on which you will be running the demonstration client (see page 24).

Now start two terminal windows, either on the same host, or on two different hosts. For the sake of example, assume that when you ran the configure script, you passed it the arguments prefix=$HOME/tmp, to have the software installed under a temporary directory in your home directory, and that this home directory is visible from both of the hosts that are running the two terminals. Now in one terminal window type:

```
$HOME/tmp/bin/ccbserver
```

and in the other terminal window type:

```
$HOME/tmp/bin/ccb_demo_client
```

Provided that Tcl/Tk is installed on your system, and that the configure script found it, the demonstration client program will now display a graphical user interface, giving you write-access to all CCB configuration parameters and CCB commands. To connect this program to the CCB server, type the name of the computer on which you are running ccbserver, into the entry area to the right of the Connect button, then press this button. The logging area should then display two messages from the CCB server, saying that it is accepting new control and telemetry connections. If this doesn't happen, it probably means that at the time when ccbserver was started, the ccb_authorized_ips file didn't contain an entry authorizing connections from the computer on which you are running the ccb_demo_client program. If so, restart ccbserver after adding an appropriate entry to the latter file.

Assuming that this all worked, the CCB server will initially have all telemetry except log-messages turned off. As a reminder of this, the Off button is initially colored bright red. To enable all telemetry, press the "Ready" button. After doing this, you should see fake integrated data being displayed in the second-to-last beige area from the bottom of the GUI, roughly once per second. After every ten of these updates, the monitoring area below the integrated data area will also display updated monitor data. The initial rates at which the integrated and monitoring data are received and displayed are set by the default configuration parameters shown in the window. These can subsequently be changed. In particular, if you change the number in the entry-area next to the "Configure Monitoring" button, and then press the latter button, the modified number will be used to determine how often fake monitor-data updates are sent.

As described later, changes to the phase-switch, cal-diode, timing, and sampler configuration parameters, in the top three panes of the window, only take effect when a new scan, dump-scan, or intra-scan is started. Thus to change the integration period of the fake integrations, you would change, say the "Integ period" configuration parameter in the "timing configuration" window-pane, then press either the "Start scan", "Stop scan", or "Dump scan" buttons, to start a new scan that operates according to these parameters.

By default, the demonstration client attempts to interact with the real CCB hardware, via the ccbserver program. Alternatively, the CCB server can be instructed to substitute a built-in simulation of the CCB hardware. In the ccb_demo_client program, this is done by selecting the *Virtual* entry of the *Load Driver* option-menu. This virtual-CCB simulation can be used for off-line testing of the manager program. It prints what it is doing, on the parent terminal of the CCB server program, indicating any effects that commands sent by the manager, would have on the real hardware. The simulated CCB also sends back faked integration and monitoring data, timed and flagged according to the current CCB configuration parameters. Diagnostic programs also receive simulated dump-mode data from the dump-mode port of the CCB server, while the virtual CCB simulation is active.

2.6.1 ccb_dummy_client

In addition to the interactive demonstration client that was introduced above, a non-interactive demonstration client called ccb_dummy_client is provided. This initiates a non-blocking connection to the demonstration server, queues a full set of configuration and operating commands to be sent to the server, once the connection has been established, then enters a select() driven event loop. The event loop then informs the communications library whenever it detects an I/O event on any of the sockets that the library tells it to watch. Once the library receives confirmation of the completion of the non-blocking control connection, it sends the previously queued commands using non-blocking I/O, and watches for replies from the server. The library then forwards replies from the server to the demonstration client, by calling the callback functions that the demonstration program provided it. For the sake of demonstration, these callbacks display the contents of the replies on the terminal. This minimal program, which was written before the interactive demo program, is potentially useful for speed tests, since the GUI display of 1ms integration updates would be too fast to follow by eye, even if one were confident that the X server could keep up with this rate. For example, using this program with the demo server, one can verify that the demo server can't generate integrations more frequently than 10ms. This is expected, because that is the granularity of the Linux timers that govern the event loop of the virtual driver in the CCB server program.

This program also provides a simple working example of how to use the C interface, and can act as a basis for custom test programs, such as one that writes integrated data to disk for later examination. Having said this, the Tcl interface described later is probably more convenient for quick throw-away test programs.

2.7 Run-time configuration files

Currently, the communications library that the CCB-server uses to talk to managers, requires one configuration file, as described below. In future there could conceivably be more. By default, the directory in which these files are installed is /usr/local/etc/, but this can be changed during installation, as described earlier on page 19.

2.8 The ccb_authorized_ips configuration file

This configuration file lists the host computers that are authorized to connect to the CCB server. It consists of one IP address per line. Within these addresses, each numeric field can optionally be replaced with a * wild-card. For example, the following lines authorize all computers within the Green Bank subnets, plus the computer on which the CCB server is running.

```
192.33.116.*        # All Green-Bank computers.
199.88.192.*        # All Green-Bank computers.
127.0.0.1           # The computer that is running the server.
```

As illustrated, comments can be included. These start from a '#' character and extend to the end of the line.

2.9 The assignment of log IDs

Every instrument at the observatory is assigned a unique range of numeric IDs that it can use to uniquely tag its log messages. The CCB is assigned a range that starts at 12445. To ensure that each log message is given a unique number within this range, IDs are assigned automatically to new logging statements by scripts that are run by the CCB makefile. In order that these scripts be able to find not only new places where log IDs need to be assigned, but also find all of the existing log IDs, the log IDs must be specified using the CCB_LOGID(id) macro, which is defined to be:

```
#define CCB_LOGID(x) (x)
```

Although this macro does nothing more than echo its argument, its presence provides a string for the ID assignment scripts to search for. Whenever new code is written that needs a log ID, the above macro should be placed where the ID is needed, but with an argument of a question-mark character, rather than an ID number, so that the assignment scripts know that it needs to have an ID assigned to it. When the makefile is next run, the makefile will notice that the containing file has been modified since the last time that any new log IDs were assigned, and run an assignment script to update it. This proceeds as follows.

1. A perl script called utils/find_max_logids is executed. This searches all source files that potentially contain logging statements, for all instances of the CCB_LOGID(id) macro that have ID numbers as arguments, and figures out the maximum of all of these ID numbers.

2. A second perl script, called utils/fill_logids, is then executed, with the maximum log ID that the previous script found, as an argument. This script also searches all files that potentially contain logging statements, but this time for instances of the string CCB_LOGID(?). Each time that it finds one of these, it replaces the question-mark argument with the next lowest unused log ID, and then increments its internal record of the lowest unused log ID.

3. Finally, after all log IDs have been assigned in the above manner, the find_max_logids script is run a second time, to determine the new maximum log ID. This number

is then passed to a bourne-shell script called utils/write_logid_header, which writes a public header file called ccblogid.h. This header file defines the value of a macro called CCB_MAX_LOGID to have the maximum log ID that is currently used by the CCB libraries.

Given that the CCB installation procedure doesn't know anything about the source files in the CCB manager, if the manager needs to create its own log IDs, without having to worry about clashing with numbers that the CCB libraries have assigned to themselves, then one way to do this would be to include the ccblogid.h script, and specify new log-ids as offsets from the value of the CCB_MAX_LOGID macro. Clearly the makefiles for these programs would need to specify the ccblogid.h header file as a dependency for any source files that assigned log IDs in this way, and the manager would need to be recompiled every time that the CCB libraries were updated. A downside of this approach would be that the IDs of statements in the manager would change whenever this happened. It might thus be better simply to assign the manager a range of IDs significantly above the maximum number that the CCB libraries use, and simply check the ccblogid.h header file by eye, every so often, to make sure that the two ranges aren't getting close to overlapping.

To force the CCB makefile to reassign a contiguous range of log IDs, from scratch, to all of the source files of the CCB libraries, the makefile provides the reset_logids target. This runs a script called utils/clear_logids, which replaces all instances of the CCB_LOGID() macro with the string CCB_LOGID(?), such that the next time that the makefile is executed, log IDs are assigned from scratch.

Chapter 3

The common parts of the CCB server and client APIs

The libccbcommon library includes functions, datatypes and a few macros, that are used by all of the Caltech parts of the CCB software, and the CCB manager. Since each of the other CCB shared libraries are linked with this library, when they are created by the CCB makefile, there is no need to explicitly link with this library, when compiling a CCB program that uses one or more of the other libraries.

3.1 The configuration of the CCB

CCB configuration parameters are exchanged with the client and server libraries using CCB-Config objects. Since these objects are opaque, external functions must be used both to allocate them, and to modify and query their contents. This section describes these functions.

The new_CCBConfig() function allocates CCBConfig objects from the heap, and initializes them with the default power-on configuration of the CCB. On error it returns NULL.

```
CCBConfig *new_CCBConfig(void);
```

To reclaim the resources of a redundant CCBConfig object, the del_CCBConfig function must be called to return the object to the heap.

```
CCBConfig *del_CCBConfig(CCBConfig *cnf);
```

The ccb_default_config() function can be used to replace the current configuration in a CCBConfig object with the power-on-default configuration of the CCB. It returns non-zero and sets errno to EINVAL if its argument is NULL. Otherwise it returns 0.

```
int ccb_default_config(CCBConfig *cnf);
```

The ccb_copy_config() function copies the contents of the configuration object, orig, to the configuration object dest. It returns non-zero and sets errno to EINVAL if either of its arguments is NULL. Otherwise it returns 0.

```
int ccb_copy_config(const CCBConfig *orig, CCBConfig *dest);
```

The ccb_check_config() function checks whether the configuration parameters installed within a given configuration object are valid, and returns 0 if they are. Otherwise, it returns non-zero and places an error message in the buffer that the caller passes via the errmsg argument. The allocated dimension of this buffer must be provided in the errdim argument. Error messages whose length, including the standard '\0' terminator, exceed this size are truncated to fit.

```
int ccb_check_config(CCBConfig *cnf, size_t errdim, char *errmsg);
```

The CCB configuration parameters are partitioned into a number of groups. These groups are enumerated by the CCBConfigType datatype. Note that the values are integer powers of two, such that their values correspond to single bits within an integer.

```
typedef enum {
  CCB_CNF_PHASE_SWITCHES = 1,  /* Phase-switch parameters */
  CCB_CNF_CAL_DIODES     = 2,  /* The calibration-diode parameters */
  CCB_CNF_TIMING         = 4,  /* The hardware timing parameters */
  CCB_CNF_SAMPLER        = 8,  /* The sampler configuration parameters */
/*
 * The union of all of the above.
 */
 CCB_CNF_ALL = CCB_CNF_PHASE_SWITCHES | CCB_CNF_CAL_DIODES |
               CCB_CNF_TIMING | CCB_CNF_SAMPLER
} CCBConfigType;
```

The ccb_diff_config() function compares two CCB configuration objects and returns the bitwise union of the enumerators of the configuration groups whose parameters differ.

```
unsigned ccb_diff_config(CCBConfig *ca, CCBConfig *cb);
```

The following sub-sections describe the functions that are used to set and query the parameters of each of the configuration groups within a CCB configuration object. Each of the querying functions returns the parameters as a group, encapsulated within a structure. The ccb_set_config() function provides a means to set the whole configuration using these encapsulating arguments.

```
int ccb_set_config(CCBConfig *cnf,
                   const CCBPhaseSwitchCnf *phase,
                   const CCBCalDiodeCnf *cal,
                   const CCBTimingCnf *timing,
                   const CCBSamplerCnf *sampler);
```

Since any of the configuration arguments can be NULL, one can use this function to update either the whole configuration or just a subset of the configuration groups. The datatypes of the configuration arguments are described in detail in the following sections. On error, this function returns non-zero and sets errno accordingly. Otherwise it returns 0.

3.1.1 The configuration of the phase switches

The digital backend generates two phase-switch TTL control signals, both of which are used by the 1cm receiver, and only one of which is used by the 3mm receiver. The CCB server supports the 16 phase-switching modes illustrated in figure 3.1.

Each row of this diagram displays the 4 possible cycles of a particular combination of active switches, with each of these cycles corresponding to a different pair of initial phase-switch states.

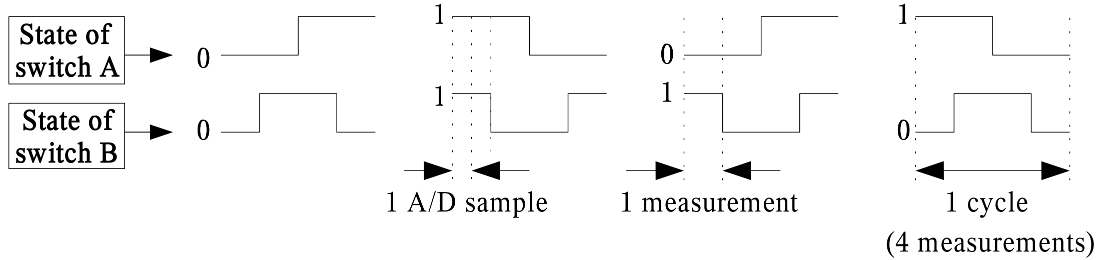
Note that whereas the number of A/D samples per phase-switch state in this diagram is just an example of what can be configured, the number of phase-switch states per cycle is fixed by the number of switches that are active, and is thus not otherwise configurable.

The configuration of the phase switches within a CCB configuration object can be changed by calling ccb_set_phase_switch_cnf().

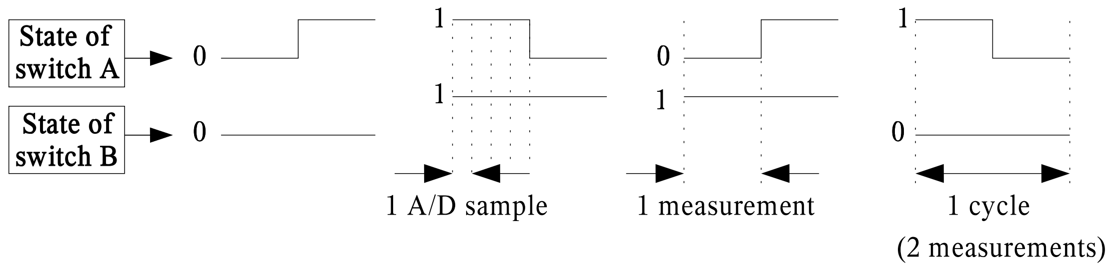
```
int ccb_set_phase_switch_cnf(CCBConfig *cnf,
                             unsigned short active_switches,
                             unsigned short closed_switches,
                             unsigned short samp_per_state);
```

The arguments of this function are interpreted as follows.

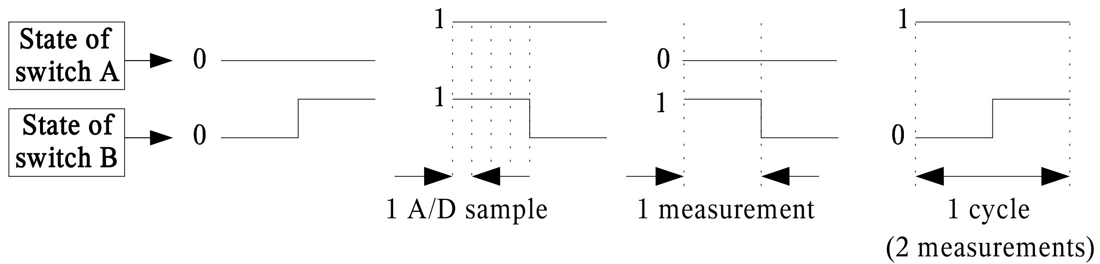
The 4 possible phase-switch cycles with both phase switches switching



The 4 possible phase-switch cycles with only phase-switch A switching



The 4 possible phase-switch cycles with only phase-switch B switching



The 4 possible phase-switch cycles with neither phase switch switching

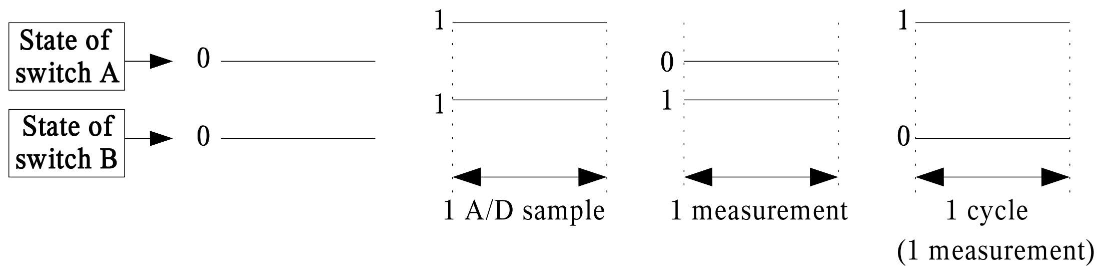


Figure 3.1: Example phase-switching cycles

- **Which switches are active?** (active_switches)

  This specifies the set of phase switches that are to be switched during phase-switching cycles, expressed as a bitwise union of CCBPhaseSwitches enumerators.

  ```
  typedef enum {
    CCB_NO_PHASE_SWITCHES = 0,  /* Neither of the phase switches */
    CCB_PHASE_SWITCH_A = 1,     /* phase switch A */
    CCB_PHASE_SWITCH_B = 2,     /* phase switch B */
    CCB_ALL_PHASE_SWITCHES =    /* Both phase switches */
      CCB_PHASE_SWITCH_A | CCB_PHASE_SWITCH_B
  } CCBPhaseSwitches;
  ```

  Any phase switches that aren't specified to be switched, are held in the positions indicated by the closed_switches argument.

- **Which switches start closed?** (closed_switches)

  This argument specifies the set of phase switches that are to be on at the start of each new phase-switch cycle, expressed as a bitwise union of CCBPhaseSwitches enumerators.

- **Samples per phase-switch state** (samp_per_state)

  This parameter configures the number of A/D samples that are integrated between changes in the states of either phase-switch. The range of supported values are given by the following macros from ccbconstants.h.

  ```
  #define CCB_MIN_SAMP_PER_STATE 250U    /* 25us */
  #define CCB_MAX_SAMP_PER_STATE 65535U  /* 6.5535ms */
  ```

The ccb_set_phase_switch_cnf() function normally returns zero, but returns non-zero and sets errno appropriately on error. Beware that a successful return doesn't necessarily mean that the configuration is valid when combined with other configuration parameters. To verify this, the ccb_check_config() function should be called once all of the configuration parameters have been set to their desired values.

The configuration parameters of the phase switches within a CCB configuration object can be queried using the ccb_get_phase_switch_cnf() function.

```
int ccb_get_phase_switch_cnf(const CCBConfig *cnf,
                             CCBPhaseSwitchCnf *pars);
```

This returns the phase-switch configuration parameters in the variable pointed at by the pars argument. This is a variable of type CCBPhaseSwitchCnf.

```
typedef struct {
  unsigned short active_switches;  /* Which switches are active? */
  unsigned short closed_switches;  /* Which switches start closed? */
  unsigned short samp_per_state;   /* ADC samples per phase-switch */
                                   /*  state. */
} CCBPhaseSwitchCnf;
```

The members of this datatype have the same meanings as the synonymous arguments of the ccb_set_phase_switch_cnf() function.

The ccb_get_phase_switch_cnf() function normally returns zero, but if either cnf or pars are NULL, it returns non-zero and sets errno to EINVAL.

3.1.2 The configuration of the calibration diodes

The digital backend generates two noise-diode TTL control signals, both of which are used by the 1cm receiver, and only one of which is used by the 3mm receiver. Since the device driver sets the on/off state of these diodes at the boundaries between integrations, each cal-diode state lasts an integral number of integrations. For each scan it is thus necessary to specify the sequence of states that the noise-diodes should go through, and how many integrations each state should last. This sequence starts with the first integration of the scan, and thereafter is repeated indefinitely until the next scan is started. Since it isn't clear how many calibration steps might be needed for future observations, the maximum number of steps is parameterized as CCB_MAX_NCAL, which is defined in the public include file of the communications library.

```
#define CCB_MAX_NCAL 32
```

The configuration of the calibration diodes within a CCB configuration object is changed by calling the ccb_set_cal_diode_cnf() function.

```
int ccb_set_cal_diode_cnf(CCBConfig *cnf,
                          unsigned short ncal,
                          const short *diode_states,
                          const unsigned long *diode_times);
```

The arguments of this function, are as follows.

- **The number of calibration steps** (ncal)

The number of steps in the calibration diode state machine. This must be less than or equal to CCB_MAX_NCAL.

Note that a value of zero can be used if the calibration diodes are to be left turned off throughout the parent scan.

- **The ncal calibration diode states** (diode_states)

  The first ncal elements of this array specify the set of calibration diodes that are to be turned on for the duration of the corresponding step of the calibration diode state machine. Each element is a bitwise union of CCBCalDiodes enumerators.

  ```
  typedef enum {
    CCB_NO_CAL_DIODES = 0,  /* Neither calibration diode */
    CCB_CAL_DIODE_A = 1,    /* Calibration diode A */
    CCB_CAL_DIODE_B = 2,    /* Calibration diode B */
    CCB_ALL_CAL_DIODES =    /* Both calibration diodes */
      CCB_CAL_DIODE_A | CCB_CAL_DIODE_B
  } CCBCalDiodes;
  ```

  This argument can be NULL if ncal is 0.

- **The durations of the ncal cal-diode states** (diode_times)

  Each element of the first ncal elements of this parameter, specifies the duration of the state in the corresponding element of the diode_states parameter. The duration is interpreted as an integer number of integrations.

  For the minimum integration time of 1ms, the use of a 32-bit value translates to a maximum duration of 48 days. This is clearly overkill, but a 16-bit value would only support up to 65 seconds per state, which might not be enough.

  This argument can be NULL if ncal is 0.

The ccb_set_cal_diode_cnf() function normally returns zero, but returns non-zero and sets errno appropriately on error. Beware that a successful return value doesn't necessarily mean that the configuration is valid when combined with other configuration parameters. To verify this, the ccb_check_config() function should be called once all of the configuration parameters have been set to their desired values.

The calibration-diode configuration parameters, within a given CCB configuration object, can be queried by calling the ccb_get_cal_diode_cnf() function.

```
int ccb_get_cal_diode_cnf(const CCBConfig *cnf,
                          CCBCalDiodeCnf *pars);
```

This returns the cal-diode configuration parameters in the variable pointed at by the pars argument. This is a variable of type CCBCalDiodeCnf.

```
typedef struct {
  unsigned short ncal;                    /* The number of steps per */
                                          /*  calibration cycle. */
  unsigned short diode_states[CCB_MAX_NCAL]; /* The set of calibration */
                                          /*  diodes that are ON */
                                          /*  during each of the */
                                          /*  'ncal' steps. */
  unsigned long diode_times[CCB_MAX_NCAL]; /* The number of */
                                          /*  integrations of each */
                                          /*  of the 'ncal' steps. */
} CCBCalDiodeCnf;
```

The members of this datatype have the same meanings as the synonymous arguments of the ccb_set_cal_diode_cnf() function.

The ccb_get_cal_diode_cnf() function normally returns zero, but if either cnf or pars are NULL, it returns non-zero and sets errno to EINVAL.

3.1.3 The configuration of hardware timing parameters

The hardware timing configuration parameters determine the durations of configurable timers in the CCB hardware. Within a CCB configuration object, these parameters are changed by calling ccb_set_timing_cnf().

```
int ccb_set_timing_cnf(CCBConfig *cnf,
                       unsigned short phase_switch_dt,
                       unsigned long diode_rise_dt,
                       unsigned long diode_fall_dt,
                       unsigned long integ_period,
                       unsigned short roundtrip_dt,
                       unsigned short holdoff_dt,
                       unsighed short adc_delay_dt);
```

The arguments of this function are interpreted as follows.

- **Phase-switch blanking interval** (phase_switch_dt)

  This specifies how much of the sample interval is blanked, while the phase switches settle after phase-switch transitions. It is expressed as an integer multiplier of 100ns. It has a minimum of 0, and a maximum that is set by the following macro from ccbconstants.h.

```
#define CCB_MAX_PHASE_SWITCH_DT 255U /* 2^8 - 1 */
```

- **Calibration diode rise time** (diode_rise_dt)

  This specifies the interval during which the calibration diode signals are unstable after being newly switched on. It is expressed as an integer multiplier of 100ns, and has a maximum value that is parameterized the the following macro definition in ccbconstants.h.

  ```
  #define CCB_MAX_DIODE_RISE_DT 4294967295U /* 2^32 - 1 */
  ```

- **Calibration diode fall time** (diode_fall_dt)

  This specifies the interval during which any residual calibration diode signals remain present after being newly switched off. It is expressed as an integer multiplier of 100ns, and has a maximum value that is parameterized the the following macro definition in ccbconstants.h.

  ```
  #define CCB_MAX_DIODE_FALL_DT 65535U /* 2^16 - 1 */
  ```

- **The integration period** (integ_period)

  This specifies the number of phase-switch cycles that are to be co-added to form the integrations that are sent to the manager. The physical length of time that this corresponds to depends on the number of samples per phase-switch cycle, and can be calculated by calling the ccb_integration_duration() function. It has a maximum value that is parameterized the the following macro definition in ccbconstants.h.

  ```
  #define CCB_MAX_INTEG_PERIOD 65535U /* 2^16 - 1 */
  ```

- **The round-trip propagation delay** (roundtrip_dt)

  This specifies the expected delay between the CCB hardware toggling any of the switch control-lines, and the first effects of this reaching the CCB digital integrators. The roundtrip_dt parameter specifies this in units of 100ns, and has a maximum value of CCB_MAX_ROUNDTRIP_DT, which is a macro which is defined in ccbconstants.h, as follows.

  ```
  #define CCB_MAX_ROUNDTRIP_DT 255U /* 2^8 - 1 */
  ```

  The specified roundtrip delay should be a lower-limit to the expected delay, such that the CCB hardware can safely assume that all data that arrive at the integrators for this long after a switch-change is commanded, can be assumed to be associated with the previous states of the phase and cal-diode switches. The underestimate of this parameter should be compensated by overestimating the cal-diode and phase-switch settling times.

Note that the predictable contributions to this delay include the propagation delay of the opto-isolators at the ends of the receiver control-cables, the group-delay of the 2MHz low-pass Bessel filters, and the pipeline delays of the ADCs and the input latches of the CCB hardware. These add up to about 700ns. When the group-delays of the RFI filters, control-signal pulse shapers, and the rest of the electronics are taken into account, the final round-trip delay will thus probably be over 1μs. The maximum supported delay of 255×100ns, is thus 25 times greater than the expected value.

The default value of roundtrip_dt is 5, which corresponds to a physical delay of 500ns, which is probably overly-cautious.

- **The interrupt-generation holdoff delay** (holdoff_dt)

  This sets the minimum interval between the interrupts that the CCB hardware can generate, and has three goals.

  1. To prevent the CPU from locking up if an interrupt source in the CCB hardware, for some unforeseen reason, attempts to generate interrupts at an extreme rate.
  2. To reduce the number of interrupts that need to be sent, by allowing interrupts of multiple interrupt sources to be signaled by one hardware interrupt.
  3. To set the repeat interval at which unacknowledged interrupts are to be signaled again.

  The actual holdoff interval that corresponds to a given value of the holdoff_dt parameter, is given by.

$$\mathrm{dt} = 25.6\mu\mathrm{s} \times (\mathrm{holdoff_dt} + 1) \tag{3.1}$$

  Thus, since the holdoff_dt parameter is allowed to take any value in the range 0-CCB_MAX_HOLDOFF_DT, where CCB_MAX_HOLDOFF_DT is a macro which is defined in ccbconstants.h, as follows,

```
#define CCB_MAX_HOLDOFF_DT 31U    /* 5 bits */
```

  the supported range of holdoff intervals is 25.6μs – 0.8192ms. The lower-limit was chosen to be just over the claimed average interrupt latency of the Linux 2.6 kernel, in an attempt to ensure that regardless of the configuration parameters, the Linux kernel will not be overwhelmed by CCB interrupts. The upper-limit is set to be less than the minimum, 1ms, integration time. This is necessary, since integration-configuration interrupts must be generated and responded to, on average, within less than the integration time. The default value of the holdoff_dt parameter is 7, which translates to a physical holdoff interval of 204.8μs. Note that this is larger than the reported 181μs maximum interrupt latency of the Linux 2.6 kernel, and should thus reduce the probability of interrupts having to be redundantly signaled when not acknowledged quickly. At the same time, it is small enough to allow several integration-configuration interrupts to be generated and responded to per integration period.

Note that the above-quoted characteristics of the Linux 2.6 kernel were obtained from an article at the following URL.

```
http://www.linuxdevices.com/articles/AT7751365763.html
```

- **The ADC clock delay** (adc_delay_dt)

  The clock signal that is used to clock the ADCs is a phase-shifted copy of the global FPGA clock. The phase shift is implemented by delaying a copy of the global FPGA clock by adc_delay_dt×10ns. The adc_delay_dt parameter has a maximum value that is parameterized by the following macro in ccbconstants.h.

  ```
  #define CCB_MAX_ADC_DELAY 9U
  ```

The ccb_set_timing_cnf() function normally returns zero, but returns non-zero and sets errno appropriately on error. Beware that a successful return doesn't necessarily mean that the configuration is valid when combined with other configuration parameters. To verify this, the ccb_check_config() function should be called once all of the configuration parameters have been set to their desired values.

Note that this function will complain, not only if the above parameters are outside the ranges mentioned above, but also if the resulting physical duration of each integration is less than the value of the following macro, which is defined in ccbconstants.h.

```
#define CCB_MIN_INTEG_DT 1000000  /* 1000000 x 1ns = 1ms */
```

The configuration parameters of the hardware timing within a CCB configuration object can be queried using the ccb_get_timing_cnf() function.

```
int ccb_get_timing_cnf(const CCBConfig *cnf, CCBTimingCnf *pars);
```

This returns the timing configuration parameters in the variable pointed at by the pars argument. This is a variable of type CCBTimingCnf.

```
typedef struct {
  unsigned short phase_switch_dt;  /* The settling time of the phase */
                                   /*  switches. */
  unsigned long diode_rise_dt;     /* The rise time of a cal diode */
  unsigned long diode_fall_dt;     /* The fall time of a cal diode */
  unsigned long integ_period;      /* The integration period */
  unsigned short roundtrip_dt;     /* The delay between toggling a */
```

```
                                   /*  receiver control line and */
                                   /*  the first effects reaching */
                                   /*  the CCB integrators. */
    unsigned short holdoff_dt;     /* The minimum time between */
                                   /*  generating CPU interrupts */
                                   /*  is 25.6us*(holdoff_dt+1) */
    unsigned short adc_delay_dt;   /* The offset of the ADC clock */
                                   /*  from the main FPGA clock, */
                                   /*  expressed as a multiple of */
                                   /*  10ns. */
  } CCBTimingCnf;
```

The members of this datatype have the same meanings as the synonymous arguments of the ccb_set_timing_cnf() function.

The ccb_get_timing_cnf() function normally returns zero, but if either cnf or pars are NULL, it returns non-zero and sets errno to EINVAL.

3.1.4 The configuration of sampler control parameters

The sampler configuration parameters control the digitized samples that are integrated in normal integration mode, and collected verbatim, in dump mode. Within a CCB configuration object, these parameters are changed by calling ccb_set_sampler_cnf().

```
int ccb_set_sampler_cnf(CCBConfig *cnf, CCBSampleType sample_type);
```

The arguments of this function are interpreted as follows.

- **The digitized samples to integrate or dump** (sample_type)

  This argument specifies what type of digitized samples are to be used by the hardware. These can be either the real ADC samples, or fake pseudo-random samples. The latter are used for testing the digital parts of the CCB hardware. The value of this argument must be one of the following enumerated values.

  ```
  typedef enum {
    CCB_ADC_SAMPLES,   /* Use the real ADC samples */
    CCB_FAKE_SAMPLES   /* Use fake pseudo-random samples */
  } CCBSampleType;
  ```

  The default configuration sets this parameter to CCB_ADC_SAMPLES.

The ccb_set_sampler_cnf() function normally returns zero, but returns non-zero and sets errno appropriately on error. Beware that a successful return doesn't necessarily mean that the configuration is valid when combined with other configuration parameters. To verify this, the ccb_check_config() function should be called once all of the configuration parameters have been set to their desired values.

The configuration parameters of the hardware sampler within a CCB configuration object can be queried using the ccb_get_sampler_cnf() function.

```
int ccb_get_sampler_cnf(const CCBConfig *cnf, CCBSamplerCnf *pars);
```

This returns the sampler configuration parameters in the variable pointed at by the pars argument. This is a variable of type CCBSamplerCnf.

```
typedef struct {
  unsigned short sample_type;  /* The type of digitizer samples */
} CCBSamplerCnf;
```

The members of this datatype have the same meanings as the synonymous arguments of the ccb_set_sampler_cnf() function.

The ccb_get_sampler_cnf() function normally returns zero, but if either cnf or pars are NULL, it returns non-zero and sets errno to EINVAL.

3.2 Textual configuration

The previously described functions manipulate a CCB configuration object using binary representations of parts of the configuration. This is efficient, but tedious for people wanting to write quick diagnostic programs. Thus, primarily for the use of diagnostic programs that use the libccbtestclient library, to be described later, a simpler method has been provided, that uses human-readable parameter assignments in text-strings.

The following is an example of a configuration string.

```
"integ_period=100 active_switches=AB"
```

This would change the integ_period parameter, which sets the number of phase-switch cycles per integration-period, to have the value 100, and would configure both phase-switches A and B to be active. It would leave all other configuration parameters within a given CCBConfig object, unchanged.

All of the recognized parameter names, along with examples of corresponding legal assignment values are shown in table 3.1.

Parameter name	Example value	Legal values	Description
active_switches	AB	AB or A or B or NONE	Which phase switches should be active?
closed_switches	NONE	AB or A or B or NONE	Which phase switches should start closed?
samp_per_state	250	Integers 250 to 65535	The number of samples per phase-switch state
cal_steps	B*10,AB*5	(see below)	The sequence of states of the calibration diodes
phase_switch_dt	1	Integers 0 to 255	The number of samples to blank per phase-switch transition
diode_rise_dt	10	Integers 0 to 4294967295	The cal-diode turn-on settling time ($\times$100ns)
diode_fall_dt	5	Integers 0 to 65535	The cal-diode turn-off settling time ($\times$100ns)
integ_period	10	Integers 0 to 65535	The number of phase-switch cycles per integration period
roundtrip_dt	7	Integers 0 to 255	The roundtrip control delay ($\times$100ns)
holdoff_dt	0	Integers 0 to 31	The interrupt holdoff delay is 25.6μs $\times$ (holdoff_dt + 1)
adc_delay_dt	5	Integers 0 to 9	The ADC clock-delay ($\times$10ns)
sample_type	ADC	ADC or FAKE	The source of ADC samples

Table 3.1: Textual configuration parameters and their values

The cal_steps parameter needs further explanation. This sets the sequence of states that the calibration-diodes cycle through, and the number of integration periods that each state should last. For example, the assignment:

```
cal_steps=B*10,AB*5,NONE*100
```

Tells the CCB to turn on just cal-diode B, for 10 integration periods, then turn on both of cal-diodes A and B for 5 integration periods, and then turn both diodes off for 100 integration periods. This cycle repeats indefinitely.

3.2.1 ccb_parse_CCBConfig()

The configuration parameters within a given CCBConfig object can be selectively modified by a configuration string, by calling the ccb_parse_CCBConfig() function. This has the following prototype:

```
int ccb_parse_CCBConfig(const char *text, CCBConfig *cnf,
                        char *errmsg, size_t errdim);
```

The first argument is the configuration string, which should contain one or more assignments to configuration parameters, as described in section 3.2. The second argument is the configuration object whose contents are to be modified. The values of any parameters that aren't specified in the configuration string, are left unchanged.

If the configuration string is valid, then the function returns 0. Otherwise it returns 1, and places an error message of up to errdim characters (including the `'\0'` terminator), in the character array that is pointed to by the errmsg argument.

3.2.2 ccb_read_CCBConfig()

The configuration parameters within a given CCBConfig object can also be selectively modified by the contents of a text file, by calling the ccb_read_CCBConfig() function. This has the following prototype:

```
int ccb_read_CCBConfig(const char *file, CCBConfig *cnf,
                       char *errmsg, size_t errdim);
```

The first argument is the name of the text file that contains the textual configuration assignments. Each assignment, which should follow the format described in section 3.2, can optionally be placed on a separate line, or simply be separated from its neighbors by spaces. Comments can be placed anywhere in the file. They start with the # character and extend to the end of the line.

The second argument of ccb_read_CCBConfig() is the configuration object whose contents are to be modified. The values of any parameters that aren't specified in the configuration file, are left unchanged.

If the contents of the file are valid, then the function returns 0. Otherwise it returns 1, and places an error message of up to errdim characters (including the `'\0'` terminator), in the character array that is pointed to by the errmsg argument.

3.2.3 ccb_print_CCBConfig()

The contents of a configuration object can be printed to a text stream, by calling the ccb_print_CCBConfig() function. This has the following prototype:

```
int ccb_print_CCBConfig(FILE *fp, CCBConfig *cnf,
                        char *errmsg, size_t errdim);
```

The first argument is the stdio text stream that the configuration will be printed to. Each parameter assignment is printed on a separate line, using the same format as was described in section 3.2.

The second argument of ccb_print_CCBConfig(), is the configuration object whose contents are to be printed. Assignments are printed for all of the parameters within this object.

If the arguments are valid, and the configuration is printed without any I/O errors, then ccb_print_CCBConfig() returns 0. Otherwise it returns 1, and places an error message of up to errdim characters (including the '\0' terminator), in the character array that is pointed to by the errmsg argument.

3.2.4 ccb_write_CCBConfig()

The contents of a configuration object can be written to a text file, by calling the ccb_write_CCBConfig() function. This has the following prototype:

```
int ccb_write_CCBConfig(const char *file, CCBConfig *cnf, char *errmsg,
                        size_t errdim);
```

The first argument is the name to give the new text file. A file of this name will be created, and the configuration will then be written to it, with one parameter assignment per line, in the form that was described in section 3.2.

The second argument of ccb_write_CCBConfig(), is the configuration object whose contents are to be written to the file. Assignments are written for all of the parameters within this object.

If the arguments are valid, and the configuration is written without any I/O errors, then ccb_write_CCBConfig() returns 0. Otherwise it returns 1, and places an error message of up to errdim characters (including the '\0' terminator), in the character array that is pointed to by the errmsg argument.

3.2.5 Parsing and displaying phase-switch specifications

In order to encourage a common textual format for entering and displaying sets of phase-switches, a couple of functions are provided for decoding and encoding phase-switch specification strings. These are used by the above configuration parsing and displaying functions, but can also be used by other programs.

The recognized specification strings, and the corresponding bit-wise ORs of CCBPhaseSwitches enumerators, are as follows:

Specification	CCBPhaseSwitches equivalent	Interpretation
"All"	`CCB_ALL_PHASE_SWITCHES`	All phase-switches.
"AB"	`CCB_PHASE_SWITCH_A | CCB_PHASE_SWITCH_B`	Phase-switches A and B.
"BA"	`CCB_PHASE_SWITCH_A | CCB_PHASE_SWITCH_B`	Phase-switches A and B.
"A"	`CCB_PHASE_SWITCH_A`	Just phase-switch A.
"B"	`CCB_PHASE_SWITCH_B`	Just phase-switch B.
"None"	`CCB_NO_PHASE_SWITCHES`	Neither phase-switch.

Comparisons to these strings are case-insensitive.

ccb_parse_CCBPhaseSwitches()

The ccb_parse_CCBPhaseSwitches() function can be called to decode specifications of the above type.

```
int ccb_parse_CCBPhaseSwitches(const char *s, unsigned short *mask);
```

The first argument is the string that contains the phase-switch specification that is to be decoded. On return, if the string contained a valid specification, then the function returns zero, and assigns the corresponding bit-wise OR of CCBPhaseSwitches enumerators to the variable that is pointed to by the mask argument.

ccb_render_CCBPhaseSwitches()

The ccb_render_CCBPhaseSwitches() function can be called to return a constant string in the above format, that describes a given set of phase-switches.

```
const char *ccb_render_CCBPhaseSwitches(unsigned short mask);
```

The mask argument should hold a bit-wise OR of one or more CCBPhaseSwitches enumerators, to specify which phase-switches have been selected. The return value is a pointer to an immutable string that contains the corresponding textual specification.

3.2.6 Parsing and displaying cal-diode specifications

Similarly, in order to encourage a common textual format for entering and displaying sets of cal-diodes, a couple of functions are provided for decoding and encoding cal-diode specification strings.

The recognized specification strings, and the corresponding bit-wise ORs of CCBCalDiodes enumerators, are as follows:

Specification	CCBCalDiodes equivalent	Interpretation
"All"	`CCB_ALL_CAL_DIODES`	All cal-diodes.
"AB"	`CCB_CAL_DIODE_A | CCB_CAL_DIODE_B`	Cal-diodes A and B.
"BA"	`CCB_CAL_DIODE_A | CCB_CAL_DIODE_B`	Cal-diodes A and B.
"A"	`CCB_CAL_DIODE_A`	Just cal-diode A.
"B"	`CCB_CAL_DIODE_B`	Just cal-diode B.
"None"	`CCB_NO_CAL_DIODES`	Neither cal-diode.

Comparisons to these strings are case-insensitive.

ccb_parse_CCBCalDiodes()

The ccb_parse_CCBCalDiodes() function can be called to decode specifications of the above type.

```
int ccb_parse_CCBCalDiodes(const char *s, unsigned short *mask);
```

The first argument is the string that contains the cal-diode specification that is to be decoded. On return, if the string contained a valid specification, then the function returns zero, and assigns the corresponding bit-wise OR of CCBCalDiodes enumerators to the variable that is pointed to by the mask argument.

ccb_render_CCBCalDiodes()

The ccb_render_CCBCalDiodes() function can be called to return a constant string in the above format, that describes a given set of cal-diodes.

```
const char *ccb_render_CCBCalDiodes(unsigned short mask);
```

The mask argument should hold a bit-wise OR of one or more CCBCalDiodes enumerators, to specify which cal-diodes have been selected. The return value is a pointer to an immutable string that contains the corresponding textual specification.

3.3 Integration and scan timing information

The details of the timing of an integration are illustrated in figure 3.2. There are two measures of integration time that are of interest to users and the manager.

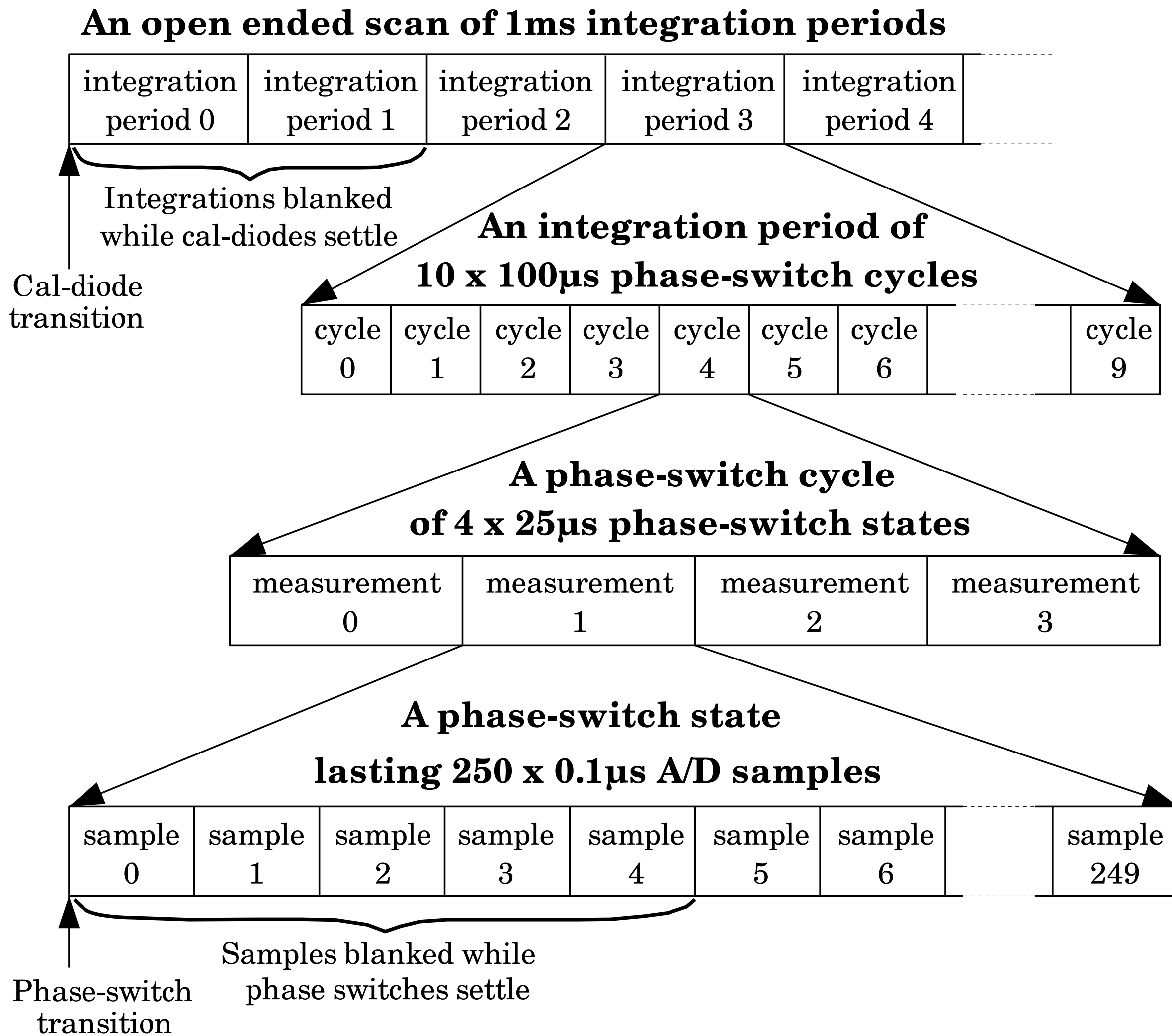


Figure 3.2: The anatomy of a data-scan or intra-scan

1. The integration-time, which is the total amount of time during which data are being accumulated in the integration-bin of a given phase switch state.
2. The integration-duration, which is the total amount of clock time that passes between the start of one integration period and the start of the next.

The integration-time is shorter than the integration-duration, both because the latter has to be split evenly between the separate integrations of different phase-switch bins, and because ADC samples are blanked during phase-switch transitions.

3.3.1 Interval computations

Since the duration of a scan may exceed the number of 100ns clock ticks that will fit within C's unsigned long integer datatype, time intervals returned by functions in this section are expressed using a pair of integers, one recording the number of full seconds in the interval, and the other containing the remaining nanoseconds. These integers are encapsulated within the CCBInterval datatype.

```
typedef struct {       /* A time interval (t=sec+1.0e-9*ns) */
  unsigned long sec;   /* The number of complete seconds */
  unsigned long ns;    /* The remaining number of nano-seconds */
} CCBInterval;
```

On a computer that has 32-bit long integers, CCBInterval datatypes can hold time intervals between 0ns and 136 years, with nano-second precision. Note however that the hardware clock has a period of 100ns, so this sets the actual precision achievable.

The following functions are provided for manipulating intervals that are stored this way.

ccb_scale_interval()

The ccb_scale_interval() function returns the product of a time interval and an unsigned long integer.

```
int ccb_scale_interval(CCBInterval *dt, unsigned long factor,
                       CCBInterval *ans);
```

A pointer to the time interval to be scaled is presented via the dt argument, the scale factor is specified via the factor argument, and the answer is recorded within the variable pointed

to by the ans argument. The ans argument and the dt argument can be pointers to the same variable, in which case the answer will replace the original time interval within dt. Normally the return value of the function is 0. If dt or ans are NULL, or the result overflows the huge bounds of the CCBInterval datatype, then errno is set accordingly, and the function returns 1.

ccb_add_intervals()

The ccb_add_intervals() function computes the sum of two time intervals.

```
int ccb_add_intervals(CCBInterval *dt1, const CCBInterval *dt2,
                      CCBInterval *sum);
```

Pointers to the variables that contain the two intervals to be added, are passed via the dt1 and dt2 arguments. The sum is assigned to the variable pointed to by the sum argument, which is allowed to be the same variable as that pointed to by dt1. Normally the return value of the function is 0, but if any of the arguments are NULL, or the sum overflows the bounds of the CCBInterval datatype, then errno is set accordingly, and the function returns 1.

ccb_subtract_interval()

The ccb_subtract_interval() function subtracts a small time interval from a larger time interval.

```
int ccb_subtract_interval(CCBInterval *dt1, CCBInterval *dt2,
                          CCBInterval *dif);
```

Normally ccb_subtract_interval() places the difference dt1 - dt2 in the dif argument and returns zero, but if the time interval being subtracted is greater than the interval that it is being subtracted from, or any of the arguments is NULL, dif is left unchanged, ccb_subtract_interval() returns 1 to indicate that an error occurred, and errno is set accordingly.

Note that dif and dt1 are allowed to point at the same variable, thus implementing the equivalent of dt1 -= dt2.

ccb_compare_intervals()

The ccb_compare_intervals() function compares two time intervals and returns an indication of their ordering.

```
int ccb_compare_intervals(const CCBInterval *dt1,
                          const CCBInterval *dt2);
```

The return value is -1 if dt1 $<$ dt2, 0 if dt1 $==$ dt2, or 1 if dt1 $>$ dt2.

ccb_zero_interval()

The ccb_zero_interval() function initializes a time interval to zero.

```
void ccb_zero_interval(CCBInterval *dt);
```

ccb_interval_is_zero()

The ccb_interval_is_zero() function returns non-zero if its argument denotes an interval of zero seconds and zero nanoseconds.

```
int ccb_interval_is_zero(CCBInterval *dt);
```

ccb_clock_interval()

The ccb_clock_interval() function converts from a time expressed as a number of 100ns hardware clock ticks, to a time interval recorded in a CCBInterval datatype.

```
void ccb_clock_interval(unsigned long ticks, CCBInterval *dt);
```

The time interval that corresponds to the number of clock ticks in the ticks argument, is returned in the variable pointed to by the dt argument.

3.3.2 Cal-diode and phase-switch settling times

At the start of some integrations, calibration-diodes and/or phase-switches change state, and the effects on the detected signals take some time to settle. During this settling time, the hardware flags integrations that should be blanked by the off-line software. Similarly, during each phase-switch cycle, each change in the states of the phase switches, is followed by ADC samples being discarded while the signals settle to their new values. The settling times of

the calibration-diodes potentially depends on whether they are being turned on or off, so in order to compute the settling time, both *before* and *after* states of the diodes are needed.

The overall settling time of a given set of simultaneously commanded switch transitions, is the maximum of the settling times of the individual contributing transitions. This is calculated by the ccb_settling_time() function, which takes the pre-transition and post-transition states of the calibration diodes and phase-switches and returns the longest settling time of these transitions.

```
int ccb_settling_time(const CCBConfig *cnf,
                      unsigned prev_cal, unsigned next_cal,
                      unsigned prev_phs, unsigned next_phs,
                      CCBInterval *dt)
```

The cnf argument specifies the configuration of the CCB in the parent scan. The prev_cal and next_cal arguments specify the pre-transition and post-transition states of the calibration diodes, expressed as bitwise unions of CCBCalDiodes enumerators, and the prev_phs and next_phs arguments specify the pre-transition and post-transition states of the phase switches, expressed as bitwise unions of CCBPhaseSwitches enumerators. The dt argument should be a pointer to the CCBInterval variable in which to return the settling time.

The return value of ccb_settling_time() is normally 0, but if an error prevents the settling time from being computed, errno is set accordingly, and 1 is returned.

3.3.3 The number of phase-switch states per cycle

The total amount of time within an integration that is lost to phase-switch blanking, depends on the number of phase-switch states within a phase-switching cycle. This, as previously illustrated in figure 3.1, depends on how many phase-switches are configured to be switching, and can be calculated as:

$$\text{measurements_per_cycle} = 2^{\text{nswitching}} \tag{3.2}$$

Based on this equation, the ccb_cycle_length() function returns the number of phase-switch states per cycle, corresponding to a particular value of the active_switches argument of ccb_set_phase_switch_cnf() (see page 29).

```
unsigned ccb_cycle_length(unsigned active_switches);
```

3.3.4 The physical duration of an integration period

The number of ADC samples per integration is the product of three terms; the number of phase-switch cycles per integration (integ_period), the number of phase-switch states per phase-switch cycle, as returned by ccb_cycle_length(), and the number of A/D samples per phase-switch state, given by the samp_per_state configuration parameter.

The ccb_integration_duration() function performs this calculation.

```
int ccb_integration_duration(const CCBConfig *cnf, CCBInterval *dt);
```

The cnf argument specifies the configuration of the CCB during the target scan, and the integration period is returned in the variable pointed to by the dt argument.

The return value of ccb_integration_duration() is normally 0, but if an error prevents the integration duration from being computed, errno is set accordingly, and 1 is returned.

3.3.5 The effective integration time

The actual amount of time per integration period that is spent integrating data, is less than the period between the start of one integration and the start of the next, due to the blanking of ADC samples during phase-switch transitions. The actual integration time per active phase-switch bin is calculated by the ccb_integration_time() function.

```
int ccb_integration_time(const CCBConfig *cnf, CCBInterval *dt);
```

The cnf argument specifies the configuration of the target scan, and the corresponding integration time is returned in the variable pointed to by the dt argument.

The return value of ccb_integration_time() is normally 0, but if an error prevents the integration time from being computed, errno is set accordingly, and 1 is returned.

3.3.6 The duration of a scan

The physical duration of a scan of a known number of integration periods is calculated by the ccb_scan_duration() function. This function simply uses ccb_scale_interval() to multiply the return value of ccb_integration_duration() by the specified number of integration periods.

```
int ccb_scan_duration(const CCBConfig *cnf,
```

```
                        unsigned long integrations,
                        CCBInterval *scan_dt);
```

The cnf argument specifies the configuration of the target scan. The integrations argument specifies the number of integrations within the scan, and the answer is returned in the variable pointed to by the scan_dt argument.

The return value of ccb_scan_duration() is normally 0, but if an error prevents the scan duration from being computed, 1 is returned, and the contents of scan_dt are undefined.

3.3.7 The number of integrations that fit within a time interval

The number of complete integrations that will fit within a given time interval, along with the remaining time of any final fractional integration, can be calculated with the ccb_integ_per_interval() function.

```
int ccb_integ_per_interval(const CCBConfig *cnf, CCBInterval *duration,
                           unsigned long *n, CCBInterval *rdt);
```

The cnf argument specifies the configuration of the target scan. The duration argument specifies the desired physical time-duration. The number of complete integrations that would fit into the specified time duration, is assigned to the variable pointed to by the n argument. The remaining time interval of any fractional final integration needed to reach the specified time interval, is assigned to the rdt argument. Note that if the value returned in either of the n and rdt arguments is unneeded, the corresponding argument pointer can be passed as NULL.

The return value of ccb_integ_per_interval() is normally 0, but if an error prevents the function from completing successfully, 1 is returned, and the values of *n and *rdt become undefined.

3.3.8 The number of integrations in a calibration cycle

A single calibration cycle consists of up to CCB_MAX_NCAL calibration-diode states, with each state being sustained for an integer number of integrations, as dictated by ccb_set_cal_diode_cnf(). The ccb_cal_cycle_length() function totals up the number of integrations spent in each of these states, and thus returns the number of integrations taken to perform a single calibration cycle.

```
int ccb_cal_cycle_length(const CCBConfig *cnf, unsigned long *ninteg);
```

The cnf argument specifies the configuration of the target scan, and the total number of integrations in a calibration-cycle is returned in the variable pointed to by the ninteg argument.

The return value of ccb_cal_cycle_length() is normally 0, but if an error prevents the scan duration from being computed, 1 is returned, and the value of *ninteg is undefined.

3.4 Timestamps

There are various places where the date and time of an event need to be recorded and communicated with high accuracy. In particular every telemetry message includes a timestamp which tells the manager when the corresponding event occurred. In both the server and client libraries, these timestamps are exchanged in CCBTimeStamp structures.

```
typedef struct {
  unsigned long mjd;    /* The Modified Julian Day number */
  unsigned long sec;    /* The number of seconds into the day */
  unsigned long ns;     /* The number of nano-seconds into */
                        /*  the specified second. */
} CCBTimeStamp;
```

The members of this structure are interpreted as follows.

- **mjd**

  This is the date at which the telemetry message was assembled. The date is expressed in UTC, as a Modified Julian Day number. Specifically, this is the integer part of (Julian_Date − 2400000.5).

- **sec**

  This is the time of day at which the telemetry message was assembled, expressed as the number of seconds that have passed since 0H UTC on the day indicated by the mjd argument.

- **ns**

  This is the number of nano-seconds that have elapsed since the start of the second that is indicated by the sec parameter.

The following utility functions manipulate timestamps.

3.4.1 Zero-initializing a timestamp

The ccb_zero_timestamp() function sets all of the fields of a specified CCBTimeStamp variable to zero.

```
void ccb_zero_timestamp(CCBTimeStamp *ts);
```

3.4.2 Getting the current date and time

The ccb_get_timestamp() function returns the current date and time in a specified CCBTimeStamp structure.

```
int ccb_get_timestamp(CCBTimeStamp *t);
```

The current date and time are returned in the variable pointed to by the t argument. The function normally returns 0, but on error returns 1 and sets errno accordingly.

3.4.3 Comparing two timestamps

The ccb_compare_timestamps() function compares the dates and times in two timestamps and returns an indication of their ordering.

```
int ccb_compare_timestamps(CCBTimeStamp ta, CCBTimeStamp tb);
```

The return value of this function is -1, 0 or 1, depending on whether ta < tb, ta==tb, or ta > tb respectively.

3.4.4 Computing the amount of time remaining until a given time

The ccb_time_until() function returns the amount of time remaining until a specified time, returning a time-interval of zero if the time has already passed.

```
int ccb_time_until(CCBTimeStamp ts, CCBInterval *dt);
```

The time of the event of interest is passed in the ts argument, and the amount of time remaining before that time is passed is returned in the variable pointed at by the dt argument. The returned interval is obviously out of date as soon as it is returned, and is thus of limited use. However it was written for the CCB simulator, in the ccbserver program, where the actual timing achieved isn't critical.

3.4.5 Adding a time-interval to a timestamp

The ccb_add_to_timestamp() function computes the timestamp of an event a given amount of time in the future of an existing timestamp.

```
int ccb_add_to_timestamp(const CCBTimeStamp *ta,
                         const CCBInterval *dt,
                         CCBTimeStamp *tb);
```

The existing timestamp should be in the value pointed at by the ta argument, and the time-interval to be added to this in the value pointed at by the dt argument. The addition of these two values is returned in the variable pointed at by the tb argument. Note that tb and ta are allowed to point at the same variable.

3.4.6 Converting a time_t value to a CCBTimeStamp value

The ccb_time_to_timestamp() function takes a time_t value returned by any of the functions in the standard C library, plus a fraction of a second expressed in nanoseconds, and returns the CCBTimeStamp equivalent.

```
int ccb_time_to_timestamp(time_t t, unsigned long ns,
                          CCBTimeStamp *ts);
```

The result is returned in the variable pointed to by the ts argument. The return value of ccb_time_to_timestamp() is normally 0, but if an error occurs, 1 is returned, and errno is set accordingly.

3.4.7 Converting a CCBTimeStamp value to a time_t value

The ccb_timestamp_to_time() function takes a CCBTimeStamp value and converts it to a standard time_t value, for use with functions in the standard C library, plus a remainder in nanoseconds.

```
int ccb_timestamp_to_time(CCBTimeStamp ts, time_t *secs, unsigned long *ns);
```

The time_t seconds part of the value is returned in the variable pointed to by the secs argument, and any remaining fractional part of the second, is assigned to the variable that is pointed to by the ns argument, expressed as an integral number of nanoseconds.

The return value of ccb_timestamp_to_time() is normally 0, but if an error occurs, 1 is returned, and errno is set accordingly.

3.4.8 Getting the clock-time from a timestamp

The ccb_hms_of_timestamp() function, returns the clock-time of a timestamp in hours, minutes and seconds.

```
void ccb_hms_of_timestamp(CCBTimeStamp ts, unsigned *hours,
                          unsigned *mins, unsigned *secs);
```

The clock-time is returned in the variables pointed to by the hours, minutes and seconds arguments.

3.5 Pseudo-random fake samples

In section 3.1.4 it was shown how the CCB can be configured to substitute fake, pseudo-random samples for real ADC samples. The pseudo-random sequence of fake samples is restarted at the start of each integration period. This means that one can precisely predict what each sample of the integration period should be, and what the corresponding integration sums should be. This section documents the functions and macros that are provided for computing what these values should be.

3.5.1 The values of integrated fake samples

In normal integration-mode, samples are directed alternately into one phase-switch bin at a time, according to the phase-switch configuration. Thus the sequence of fake samples is sampled at different times by the different phase-switch bins, and the different bins thus end up with different values in them. The integrated values that should be found in each of the phase-switch bins, at the end of every integration period, can be calculated by the ccb_fake_integrations() function.

```
int ccb_fake_integrations(const CCBConfig *cnf,
                          unsigned long bins[CCB_NUM_PHASE_BINS]);
```

Since each CCB input port receives the same sequence of fake samples, this function reports the numbers that should be found in the integration bins of all of the input-ports.

The first argument must hold the configuration of the parent scan of the integrations. If successful, the function returns 0, and places the predicted integrated values within the bins[] argument. This must be an array of one element per phase-switch bin, where the number of phase-switch bins is parameterized by the constant, CCB_NUM_PHASE_BINS, which is defined as follows.

```
#define CCB_NUM_PHASE_BINS 4
```

The individual phase-switch bins are labeled by a 2-bit integer whose least-significant bit is 1 if phase-switch A is closed, or 0 if phase-switch A is open, and whose most-significant bit is 1 if phase-switch B is closed, or 0 if phase-switch B is open. This integer is also used to index the bins array. This is summarized in table 3.2.

Index	Phase-switches	
of bin	B	A
0	open	open
1	open	closed
2	closed	open
3	closed	closed

Table 3.2: The numeric labeling of phase-switch bins

3.5.2 The values of fake samples

The generation of fake samples relies on a 14-bit linear-feedback shift-register, which, at the start of each clock-cycle, uses the current value of the register to compute its next value. The resulting sequence of numbers repeats every CCB_FAKE_CYCLE_LENGTH samples, where CCB_FAKE_CYCLE_LENGTH is defined as:

```
#define CCB_FAKE_CYCLE_LENGTH ((1<<CCB_ADC_SAMPLE_SIZE)-1)
```

Given that CCB_ADC_SAMPLE_SIZE is defined to be 14 (see section 5.7.3), CCB_FAKE_CYCLE_LENGTH has the value 16383. Each CCB_FAKE_CYCLE_LENGTH samples are unique, and include every possible 14-bit number, except zero.

Since the sequence automatically repeats itself, the starting point of the sequence is arbitrary. The CCB chooses to start the sequence with all but the top-most of the 14 bits set to 1. The starting value of the sequence is thus parameterized as follows:

```
#define CCB_FIRST_FAKE_SAMPLE ((1<<(CCB_ADC_SAMPLE_SIZE-1))-1)
```

An implementation of the recurrence relation that computes the next sample in the sequence from its previous value, is provided in the form of a macro, in the CCB libraries, and is defined as follows:

```
#define CCB_NEXT_FAKE_SAMPLE(s) (CCB_ADC_SAMPLE_MASK & \
    (((s)<<1) | (((((s)>>0)^((s)>>2)) ^ (((s)>>4)^((s)>>13))) & 1)))
```

The `s` argument of this macro must evaluate to an unsigned integral type of at least 2 bytes, and its evaluation must not have side-effects, since it is evaluated by the macro several times. Note that a macro was used, rather than a function, because the only uses of this feature to-date have used the macro in inner loops, where a function call would have added too much overhead. An example of using this is the following simple program that displays the sequence of fake values to the terminal.

```
#include <stdio.h>
#include <ccbcommon.h>

int main(int argc, char *argv[]) {
  unsigned short s; /* The latest sample in the sequence */
  int i;
/*
 * Get the first sample of the sequence.
 */
  s = CCB_FIRST_FAKE_SAMPLE;
/*
 * Loop over and print the remaining samples of the sequence.
 */
  for(i=0; i<CCB_FAKE_CYCLE_LENGTH; i++) {
/*
 * Print the value of the current sample in the sequence.
 */
    printf("Sample %5d: %hu\n", i, s);
/*
 * Get the next sample in the sequence.
 */
    s = CCB_NEXT_FAKE_SAMPLE(s);
```

```
  }
  return 0;
}
```

Chapter 4

The CCB manager communications API

The following sections describe the functions that the CCB communications library provides for use by the manager. The following illustrates a typical time sequence of function calls for a single-threaded manager.

1. **Create the object needed to talk to a CCB server:**

   ```
   char errmsg[CCB_MAX_LOG];
   CCBClientLink *cl = new_CCBClientLink(sizeof(errmsg), errmsg);
   if(!cl) {
     fprintf(stderr, "%s\n", errmsg);
     exit(1);
   }
   ```

2. **Create a CCB configuration object:**

   ```
   CCBConfig *cnf = new_CCBConfig();
   if(!cnf)
     return ERROR;
   ```

3. **Tell the CCBClientLink object how to deliver log messages:**

   ```
   if(ccb_log_msg_callback(cl, ...))
     return ERROR;
   ```

4. **Install message-received callback functions:**

   ```
   if(ccb_cmd_error_callback(cl, ...) ||
      ccb_status_reply_callback(cl, ...) ||
      ccb_integ_msg_callback(cl, ...) ||
      ccb_monitor_msg_callback(cl, ...))
     return ERROR;
   ```

5. **Initiate a non-blocking connection to a CCB server:**

```
if(ccb_client_connect(cl, host, 1))
  return ERROR;
```

6. **Load the CCB device driver**

```
if(ccb_queue_load_driver_cmd(cl, id, CCB_NORMAL_DRIVER))
  return ERROR;
```

7. **Set up the initial CCB configuration:**

```
if(ccb_set_phase_switch_cnf(cnf, ...) ||
   ccb_set_cal_diode_cnf(cnf, ...) ||
   ccb_set_timing_cnf(cnf, ...) ||
   ccb_set_sampler_cnf(cnf, CCB_ADC_SAMPLES))
  return ERROR;
```

8. **Enable all telemetry streams**

```
if(ccb_queue_telemetry(cl, id, CCB_ALL_STREAMS))
  return ERROR;
```

9. **Queue the first start-scan command:**

```
if(ccb_queue_start_scan_cmd(cl, id, cnf, ...))
  return ERROR;
```

10. **Now invoke the manager's event loop:**

```
while(!shutdown_requested) {
  fd_set rfds;  /* The set of file-descriptors to watch */
                /* for readability */
  fd_set wfds;  /* The set of file-descriptors to watch */
                /* for writability */
  int nready;   /* The number of file descriptors ready */
                /* for I/O. */
  int maxfd;    /* The maximum of the file descriptors in */
                /*  'rfds' and 'wfds'. */
/*
 * Clear the file-descriptor sets.
 */
  FD_ZERO(&rfds);
  FD_ZERO(&wfds);
  maxfd = 0;
/*
 * Add the file-descriptors that the library wants us to watch,
 * to rfds and wfds.
 */
  if(ccb_client_select_args(cl, &rfds, &wfds, &maxfd))
    return ERROR;
/*
```

```
 * Wait indefinitely for the specified I/O events.
 */
  nready = select(maxfd+1, &rfds, &wfds, NULL, NULL);
/*
 * Error?
 */
  if(nready < 1)
    return ERROR;
/*
 * Perform the types of I/O indicated by select().
 */
  else if(ccb_client_communicate(cl,
           ccb_client_selected_io(cl, &rfds, &wfds)))
    return 1;
}
```

11. **Disconnect from the current CCB server:**

```
ccb_client_disconnect(cl);
```

12. **Reclaim the CCB communication resources:**

```
cl = del_CCBClientLink(cl);
```

As documented later, note that when ccb_client_communicate() receives messages, it calls the appropriate callback function from the callbacks that were registered in step 4, to deliver these messages to the manager.

4.1 Include files

The datatype-declarations, function-prototypes and constants of the public API of the CCB-client communications-library are contained in the following include files.

- ccbclientlink.h

  This header-file contains all of the public function-prototypes and datatype declarations that are specific to the client side of the communications link.

- ccbcommon.h

  This header-file contains the public function-prototypes and datatype declarations that are shared between the client and server communications libraries. It need not be included explicitly by the client code, since it is already included by ccbclientlink.h.

- ccbconstants.h

This header-file contains all of the constants that affect the operation of the communications link. It need not be included explicitly by client code, since it is already included by ccbcommon.h.

4.2 The CCB-client communications library

The library that implements the CCB client-communications API, is a shared library called libccbclientlink.so. Under Solaris and Linux, this filename is actually a symbolic link to the most recent version of the library.

Among other advantages, the use of a shared library rather than a static library has the benefit, at least under Solaris and Linux, of allowing one to restrict which symbols are exported into the name-space of the application. This not only prevents programs from using unstable private interfaces, but also greatly reduces name-space pollution and the possibility of symbol-name clashes.

Linking a C program with this library under either Linux or Solaris can be done as follows.

```
gcc -o foo *.o -lccbclientlink
```

Note that linkage instructions built into the shared library cause other required libraries, such as -lsocket under Solaris, to be linked automatically.

4.3 Creating the client resources needed to talk to a CCB server

All of the resources that are needed to communicate with a remote CCB server are encapsulated within an opaque CCBClientLink object. Each instance of a CCBClientLink object is capable of communicating with a single CCB server at a time, so to simultaneously talk to multiple CCB servers, multiple CCBClientLink objects must be created, with one object assigned to each server. Alternatively, if only one backend is to be controlled/monitored at a time, a single CCBClientLink object can initially be connected to one CCB server, and then later be connected to a different CCB server. To create a CCBClientLink object, it is necessary to call new_CCBClientLink().

```
CCBClientLink *new_CCBClientLink(size_t errdim, char *errmsg);
```

If successful, this function allocates and returns a pointer to an opaque CCBClientLink object. This object should thereafter be passed to all other CCB communications-library functions.

Note that this function does not itself initiate a connection to a CCB server. That is the role of the ccb_client_connect() function, which will be described below.

On error new_CCBClientLink() returns NULL, and places an error message in the buffer that is pointed to by the errmsg argument. The allocated size of this buffer must be specified in the errdim argument, such that if the length of an error message exceeds errdim-1 characters, it can be truncated to fit. If truncation is necessary, and errdim is greater than 0, a '\0' terminator is placed in errmsg[errdim-1].

If new_CCBClientLink() returns successfully, subsequent errors detected by library functions are reported as log messages. If this happens before the manager has got around to calling ccb_log_msg_callback() to tell the library how to deliver log messages, the error message is displayed to the local stderr. Thus, to ensure that all log messages get recorded for posterity, it is important that the manager call ccb_log_msg_callback() as soon as possible after new_CCBClientLink() returns.

4.4 Connecting to a CCB server

The ccb_client_connect() function initiates a pair of connections to the control and telemetry ports of the CCB server.

```
int ccb_client_connect(CCBClientLink *cl, const char *host,
                       int nonblocking);
```

The cl argument must be an object previously returned by new_CCBClientLink(). If this object is already connected to a CCB server, the existing connection is terminated before the new one is initiated.

The value of the host argument should contain the numeric or textual IP address of the target CCB server.

The nonblocking argument specifies whether ccb_client_connect() should initially place the control and telemetry sockets in non-blocking I/O mode. Note that after ccb_client_connect() returns, the manager can toggle the non-blocking behavior of the sockets by calling ccb_client_non_blocking_io().

If the nonblocking argument is zero, blocking socket I/O is used, and ccb_client_connect() doesn't return until it has either established both the control and telemetry connections, or an error occurs.

Alternatively, if the nonblocking argument is non-zero, ccb_client_connect() may not get any further than initiating the connections before returning. Watching for the completion of

the connections is left to subsequent calls to ccb_client_communicate(), in response to I/O activity detected by the manager's event loop.

Note that regardless of the value of the nonblocking argument, if the host argument contains a textual address, ccb_client_connect() blocks the caller while it queries a name server for the corresponding IP address. This is due to the non-existence of a non-blocking interface to query name-servers. As such, if non-blocking behavior is required, it is best to supply a numeric IP address in the host argument.

If ccb_client_connect() detects an error, it returns non-zero. Otherwise it returns 0 to indicate successful connection initiation. When using blocking I/O, a successful return-value indicates that both the telemetry and control connections have been completed successfully.

4.5 Disconnecting from a CCB server

The ccb_client_disconnect() function terminates the telemetry and control connections of a CCBClientLink object.

```
void ccb_client_disconnect(CCBClientLink *cl);
```

If no connection is currently established, ccb_client_disconnect() does nothing. This function is called internally by ccb_client_connect() before initiating a new connection, and also by del_CCBClientLink() before deleting a CCBClientLink object, so the manager probably won't need to call it explicitly.

4.6 Deleting a redundant CCBClientLink object

Once a CCBClientLink object is no longer needed, its resources can be returned to the system by calling the del_CCBClientLink() function.

```
CCBClientLink *del_CCBClientLink(CCBClientLink *cl);
```

This function both shuts down any CCB server connection associated with the specified CCBClientLink object, and returns all dynamically allocated resources, associated with the object, to the system. The function always returns NULL. This allows the caller to type:

```
CCBClientLink *cl;
...
cl = del_CCBClientLink(cl);
```

This sets the invalidated cl pointer variable to NULL, such that if any statement subsequently tries to access the deleted object through this pointer, it is rewarded with a tell-tale segmentation fault, rather than producing unpredictable behavior.

4.7 Requesting non-blocking I/O

To prevent network congestion from blocking the manager process, when it could be doing other things, the ccb_client_non_blocking_io() function allows the manager to place the client sockets into non-blocking I/O mode. Note that this is redundant if the nonblocking argument of new_CCBClientLink() was non-zero when the CCBClientLink object was created.

```
int ccb_client_non_blocking_io(CCBClientLink *cl, int on);
```

This turns on non-blocking I/O when the on argument is non-zero, or turns it off when the on argument is zero. On error, this function sets errno appropriately and returns non-zero. Otherwise it returns zero to indicate success.

Used in conjunction with select() or poll() to perform I/O multiplexing, non-blocking I/O allows the manager to do other things during network congestion.

In a multi-threaded manager this function should not be called when any other threads might be reading from or writing messages to the CCB control and/or telemetry sockets.

4.8 ccb_client_communicate() – Perform client socket I/O

The ccb_client_communicate() function is responsible for all socket-level I/O over the control and telemetry links. According to the contents of its io argument, it incrementally sends previously queued outgoing control messages, receives incoming control-link replies, and/or receives incoming telemetry messages.

```
int ccb_client_communicate(CCBClientLink *cl, unsigned io);
```

When non-blocking I/O is selected, the io argument, which tells ccb_client_communicate() which forms of I/O to attempt, must contain a value returned by either ccb_selected_io() or ccb_polled_io(), which are documented below. Alternatively, when blocking I/O has been selected, and ccb_client_communicate() is being called separately by different threads, the

value of the io argument within a given thread must be chosen according to the type of I/O that that thread has been given responsibility for, as described shortly.

The return value of ccb_client_communicate() is normally 0, but if an error occurs, errno is set accordingly, and a non-zero value is returned.

4.9 Client I/O multiplexing

The client communications library expects to be told by the manager whenever I/O that it wants to perform can be done. It assumes that the manager is either using an event loop based on functions like select() or poll() to watch for the readability or writability of the library's sockets, or that it is devoting multiple threads to perform blocking I/O on these sockets. Since only the library knows which types of I/O it wants the manager to watch for, it provides facilities for keeping the manager informed of this.

The following subsections describe the facilities provided for different I/O multiplexing options.

4.9.1 Using the select() system call

When using select() to watch for I/O, the manager should use the ccb_client_select_args() function to augment the contents of the arguments that are to be passed to select(), according to the needs of the manager's CCBClientLink object.

```
int ccb_client_select_args(CCBClientLink *cl, fd_set *rfds,
                           fd_set *wfds, int *maxfd);
```

On input, the rfds argument is a pointer to an existing set of the file descriptors that the caller wants select() to watch for readability. Similarly the wfds argument is a pointer to an existing set of the file descriptors that the caller wants to watch for writability. Finally the maxfd argument is a pointer to a variable that contains the maximum of all file descriptors that the caller has placed in rfds and wfds. On return, rfds and wfds are augmented with the file descriptors that *cl wants to have watched, and if any of these descriptors exceeded the input value of *maxfd, it is updated accordingly. Normally this function returns 0, but on error it returns non-zero.

Note that when subsequently passing these arguments to select() the first argument of select() should be *maxfd + 1.

After select() returns, the sets of file descriptors that it found to be readable and/or writable can be converted to the form required by the io argument of ccb_client_communicate() by

calling ccb_client_selected_io().

```
unsigned ccb_client_selected_io(CCBClientLink *cl,
                                const fd_set *rfds,
                                const fd_set *wfds);
```

The rfds and wfds arguments are pointers to the sets of file descriptors that select() indicated were readable and writable, respectively. The return value is the value to pass to ccb_client_communicate() to tell it what types of I/O to attempt.

4.9.2 Using the poll() system call

When using poll() to watch for I/O, the manager should use the ccb_client_poll_args() function to augment the contents of the arguments that poll() requires, according to the forms of I/O that the manager's CCBClientLink object is awaiting.

```
int ccb_client_poll_args(CCBClientLink *cl, int size,
                         struct pollfd *fds, int *nfds);
```

The fds argument is an array of dimension size, in which *nfds elements at the start of the array are occupied. ccb_client_poll_args() adds up-to two socket file descriptors to this array, and increments *nfds accordingly. Note that size - *nfds must be at least 2. Normally this function returns 0, but on error it returns non-zero.

After poll() subsequently returns, the sets of file descriptors that it found to be readable and/or writable can be converted to the form required by the io argument of ccb_client_communicate() by calling ccb_client_polled_io().

```
unsigned ccb_client_polled_io(CCBClientLink *cl,
                              const struct pollfd *fds,
                              int nfds);
```

The fds argument is the array in which ccb_client_poll_args() placed its file descriptors, and poll() subsequently flagged I/O events, and the nfds argument is the number of occupied elements within this array. The return value is the value to pass to ccb_client_communicate() to tell it what types of I/O to attempt.

4.9.3 Third party event handlers

When using an event handler which hides the call to select() or poll() behind a custom API, the above functions clearly aren't sufficient. To cope with this more general case, the CCB communications library allows the application to register an optional callback function, which the library calls whenever the library's socket file descriptors change, and whenever the I/O events that the event handler should be watching these sockets for, change.

The CCB_CLIENT_SOCKETS_FN macro should be used for the declarations and prototypes of suitable callback functions, and the CCBClientSocketsFn typedef can be used for recording pointers to them.

```
#define CCB_CLIENT_SOCKETS_FN(fn) int (fn)(CCBClientLink *cl, \
               void *data, int cntrl_sock, int telem_sock, unsigned io)

typedef CCB_CLIENT_SOCKETS_FN(CCBClientSocketsFn);

int ccb_client_sockets_callback(CCBClientLink *cl,
                                CCBClientSocketsFn *fn,
                                void *data);
```

The callback function is registered via the fn argument of ccb_client_sockets_callback(), and any application-specific resources to be passed to the callback are specified via the data argument.

The cntrl_sock and telem_sock arguments of the callback function report the socket file descriptors associated with the control and telemetry links, respectively. When one or both of these links is not connected, the corresponding file descriptor is reported as -1. The set of I/O events that the event handler should watch these sockets for, is specified in the io argument of the callback function, expressed as a bitwise union of CCBClientIOStatus enumerators.

```
  typedef enum {
    CCB_CNTRL_READ = 1,   /* Readability of the control socket */
    CCB_CNTRL_WRITE = 2,  /* Writability of the control socket */
    CCB_TELEM_READ = 4,   /* Readability of the telemetry socket */
    CCB_TELEM_WRITE = 8   /* Writability of the telemetry socket */
  } CCBClientIOStatus;
```

Subsequently, when the application's event handler reports any of the specified events, the manager should call the ccb_client_communicate() function with an io argument that is the bitwise union of the CCBClientIOStatus enumerators that denote the events that were detected.

4.9.4 Using threads to multiplex I/O

In a threaded program, provided that blocking I/O is selected, one thread should be devoted to reading from the control socket, another to writing to the control socket, and a third to reading from the telemetry socket. First a connection must be established by calling ccb_client_connect() with its nonblocking argument specified as 0. Then each of the three threads must call ccb_client_communicate() with a different one of the following 3 CCBClientIOStatus enumerators as the value of its io argument.

- CCB_CNTRL_READ

  Read messages from the control link until either an error occurs, of the control link is terminated.

- CCB_CNTRL_WRITE

  Repeatedly wait for and write queued messages to the control link until either an error occurs or the control link is terminated.

- CCB_TELEM_READ

  Read messages from the control link until either an error occurs, of the telemetry link is terminated.

Note that in each case, ccb_client_communicate() doesn't return until either an error occurs, or the control link is terminated.

4.10 Registering a command-error callback function

Whenever the CCB server receives a command message from the manager, over the control link, it examines the contents of the message, then sends back an acknowledgment reply to report whether any problems were encountered. When the ccb_client_communicate() function receives one of these acknowledgments, if the message says that the command had a problem, ccb_client_communicate() calls an error-reporting callback function provided by the manager. To register this callback function, the manager calls ccb_cmd_error_callback().

The CCB_CMD_ERROR_FN macro should be used for the declarations and prototypes of suitable callback functions, and the CCBCmdErrorFn typedef can be used for recording pointers to them.

```
#define CCB_CMD_ERROR_FN(fn) int (fn)(CCBClientLink *cl, \
                                      void *data, long id, \
                                      CCBCmdStatus status)
```

```
typedef CCB_CMD_ERROR_FN(CCBCmdErrorFn);

int ccb_cmd_error_callback(CCBClientLink *cl, CCBCmdErrorFn *fn,
                           void *data);
```

The callback function is registered via the fn argument of ccb_cmd_error_callback(), and any application-specific resources to be passed to the callback are specified via the data argument. The id argument of the callback function is passed the value of the message identifier that was specified when the problematic command was queued by the manager. The status argument of the callback is used to report coarse information about the error.

```
typedef enum {
  CCB_CMD_ACCEPTED,     /* This enumerator isn't forwarded to error */
                        /*  callbacks */
  CCB_CMD_GARBLED,      /* The contents of the command message */
                        /*  were invalid and couldn't be fixed. */
  CCB_CMD_IGNORED,      /* The command didn't make sense at this */
                        /*  time, and was ignored. */
  CCB_CMD_SYSERR        /* An unexpected internal software or */
                        /*  hardware error was encountered while */
                        /*  attempting to execute the command. */
} CCBCmdStatus;
```

This enumeration is in the public header-file of the communication library, so to prevent ABI problems if new error conditions are added, and somebody forgets to recompile either the library, the CCB server, or the manager, new error enumerators should always be appended to the end of the enumeration, rather than inserted, and old enumerators should not be removed or re-ordered. With this caveat, if functions that use values from this enumeration are prepared to handle values that they don't know about, at worst an unknown error condition will elicit a warning, rather than, say accessing a non-existent element in an array of error conditions, or associating the incorrect error condition with an enumerator.

Note that the reported error conditions aren't meant to be very precise. For more detailed information, the maintainer should look at the corresponding log messages that the CCB server sends to the manager via the telemetry connection. As such, it is hoped that few, if any, new enumerators will need to be added. In practice, after debugging the manager and the server, the only error that should be expected during normal operations should be CCB_CMD_SYSERR, which will be sent if a hardware failure is detected. As such, adding more finely targeted error conditions seems pointless, especially given that there isn't much that the manager can do in response, other than report which command evoked the error messages that appear in the log, and perhaps attempt a CCB reset.

4.11 Outgoing CCB commands

The following two sections describe the control commands that are sent to the CCB server over its control link. The functions that queue each of these commands, all take the same two initial arguments, which are interpreted as follows.

1. A pointer to the CCBClientLink object that identifies the remote backend that the command is to be sent to.
2. An arbitrary manager-chosen integer message-identifier to be passed to the application's error callback in the event that the CCB server encounters problems with this command. This could, for example, be a manager-defined message-type enumerator, or the value of a command sequence-counter.

4.11.1 Outgoing CCB control commands

This section describes the public functions that are used to queue CCB control commands, for subsequent dispatch to the CCB server when ccb_client_communicate() is called.

Each of these functions returns an integer, which is 0 on success and non-zero otherwise. On failure, which could, for example, be due to running out of memory to queue the new message, or due to the manager passing invalid arguments, errno is set accordingly.

A summary of the available commands is given in the following table, along with the names by which they are referred to elsewhere in the text.

Name	Description
start-scan	Start a new scan at the end of the current integration
stop-scan	Start a new intra-scan ASAP
dump-scan	Start a new dump-mode scan ASAP
load-driver	Load either the real or the simulation device-driver.
reset	Re-initialize the hardware.
ping	Request a link-verification reply.
status-request	Request a CCB-status report.
shutdown	Shutdown the real-time CPU.
reboot	Reboot the real-time CPU.
monitor	Configure monitoring.
telemetry	Configure the telemetry streams.
logger	Configure the message logger.
set-dacs	Set the misc analog outputs of the GPIO card.

ccb_queue_start_scan_cmd() – Queuing a start-scan command

The start-scan command causes a new scan, with a specified configuration, to be started on the 1-second boundary that is specified by the mjd and tod parameters. The accuracy with which the CCB can start the scan at the specified time, depends on how far in advance the start-scan command is sent. There are three possibilities:

1. Ideally, the start-scan command should be timed to arrive more than one second before the requested time. If it does, then the device driver will tell the CCB hardware about the command, from its 1PPS interrupt handler, precisely one second in advance of the requested start time, telling it to start the scan at the next 1 second tick. This will then give the hardware one second to halt the previous scan, finish sending its data to the CPU, reconfigure itself for the new scan, and finally be ready and waiting to start the next scan, before the specified 1-second tick.

   Note that in this case, the scan should start at the specified time, regardless of the status of any previously running scan.

2. Less ideally, if the start-scan command arrives less than or equal to one second before the requested start time, then the device-driver in the computer will wait for the next 1-second interrupt, before telling the hardware about the command. It will then tell the hardware to start the new scan as soon as possible, without synchronizing with a 1-second tick.

   In this case, the actual start time will be delayed slightly, with respect to specified 1-second boundary, both by software latencies, and by the time needed by the hardware to halt the previous scan, finish sending any half sent data-frame to the computer, and reconfigure the hardware for the new scan.

   If the previous scan was a normal observing scan, then at worst, the time needed by the hardware to shutdown the previous scan and start the next will be much less than 1μs, and the startup delay will thus be dominated by software latencies within the computer.

   On the other hand, if the previous scan was a dump-mode scan, then the time needed by the hardware to shutdown the previous scan and start a new one could be as much as 32ms, and this should dominate the startup delay.

3. Finally, if the start-scan command is received after the requested time, then the new scan will be started as soon as possible, just like a stop-scan command, without any attempt to synchronize it to a 1 second boundary.

start-scan commands are sent by first calling ccb_queue_start_scan_cmd() to queue the command for dispatch, then subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_start_scan_cmd(CCBClientLink *cl, long id,
                             const CCBConfig *cnf,
                             unsigned long scan,
                             unsigned long mjd,
                             unsigned long tod);
```

The arguments of this function are interpreted as follows.

- **The configuration of the CCB during the new scan**

  This argument specifies the behavior of the CCB during the requested scan. It must have been previously allocated by calling new_CCBConfig(), with any changes from the default configuration having been established by calling the ccb_set_*_cnf() functions described earlier in this document.

- **A numeric ID to give the scan**

  This is a manager-chosen numeric identifier, which is thereafter transmitted along with the data of each integration of the new scan.

- **The date at which to start the scan** (mjd)

  This is the date at which the scan should be started, expressed in UTC, as a Modified Julian Day number. To be precise, this is the integer part of (Julian_Date $-$ 2400000.5).

- **The time-of-day at which to start the scan** (tod)

  This is the time of day at which the scan should be started, specified as the integer number of seconds after 0H UTC on the day indicated by mjd. The scan starts at the start of the specified second, provided that the command is received at more than one second in advance of this time.

ccb_queue_stop_scan_cmd() – Queuing a stop-scan command

This operates like the start-scan command, except that on receipt by the server, a new scan is started as quickly as possible, rather than waiting for a specified 1-second tick. The resulting truncated integration from the previous scan is discarded.

Note that although this command starts a new scan, it is called stop-scan because it stops an observing scan. The scan that it then starts, which can usefully be referred to as an *intra-scan*, is basically a scan that is used only for monitoring purposes, and is not recorded in the observer's FITS file.

A stop-scan command is sent by first calling ccb_queue_stop_scan_cmd() to queue the command for dispatch, then subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_stop_scan_cmd(CCBClientLink *cl, long id,
                            CCBConfig *cnf,
                            unsigned long scan);
```

The arguments of this function are interpreted as follows.

- **The configuration of the CCB during the new intra-scan**

  This argument specifies the behavior of the CCB during the requested intra-scan that follows the stop command. It must have been previously allocated by calling new_CCBConfig(), with any changes from the default configuration having been established by calling the ccb_set_*_cnf() functions described earlier in this document.

- **A numeric ID to give the scan**

  This is a manager-chosen numeric identifier, which is thereafter transmitted along with the data of each integration of the new intra-scan.

ccb_queue_dump_scan_cmd() – Queuing a dump-scan command

This operates like the stop-scan command, except that instead of the integrated data of the resulting intra-scan being sent back to the manager, the raw 100ns samples of a specified ADC are sent to any client that is connected to the dump-mode port of the CCB server.

Since the bandwidth of the USB link between the master FPGA and the real-time computer isn't high enough to sustain continuous readback of ADC samples, only a specified number of samples are collected from the start of each integration.

A dump-scan command is sent by first calling ccb_queue_dump_scan_cmd() to queue the command for dispatch, then subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_dump_scan_cmd(CCBClientLink *cl, long id,
                            CCBConfig *cnf,
                            unsigned long scan,
                            unsigned short adc,
                            unsigned long samples,
                            unsigned long frames);
```

The arguments of this function are interpreted as follows.

- **The configuration of the CCB during the dump-scan** (cnf)

This argument specifies the behavior of the CCB during the requested intra-scan that the dump command starts. It must have been previously allocated by calling new_CCBConfig(), with any changes from the default configuration having been established by calling the ccb_set_*_cnf() functions described earlier in this document.

- **A numeric ID to give the scan**

  This is a manager-chosen numeric identifier, which is thereafter transmitted along with the dumped data and the monitor data of each integration of the intra-scan.

- **The digitizer whose samples are to be collected** (adc)

  This specifies which ADC channel is to have its raw samples siphoned off to be sent to the CCB computer. This must be a number between 0 and CCB_NUM_ADC−1, where CCB_NUM_ADC, which is defined as follows, in ccbconstants.h, parameterizes the number of digitizers in the CCB.

```
#define CCB_NUM_ADC 16
```

- **The maximum number of samples to collect per integration** (samples)

  The number of 100ns ADC samples that can be collected and dispatched to the CCB computer at the start of each integration, is limited by the size of the frame-buffer in the CCB's master FPGA. If fewer samples than the hard-limit are desired per integration, then this argument can be used to specify a smaller number. Otherwise the argument can be specified as CCB_DUMP_MAX_SAMPLES, which is a macro that parameterizes the maximum number of samples that can be acquired per integration period. This is defined as follows.

```
#define CCB_FPGA_FIFO_WORD_SIZE 16384      /* 16-bit words */
...
#define CCB_DUMP_MAX_SAMPLES (CCB_FPGA_FIFO_WORD_SIZE-1)
```

  Where CCB_FPGA_FIFO_WORD_SIZE denotes the actual size of the FIFO that implements the firmware frame-buffer.

  Note that if the samples argument specifies more samples than can actually be accumulated, the result will be as though CCB_DUMP_MAX_SAMPLES had been specified.

- **The maximum number of integrations to dump** (frames)

  At the start of each new integration of a dump-scan, data are collected and sent to the CCB computer. This argument specifies how many of these per-integration data-frames are to be sent, per scan, to remote dump-mode clients. A value of zero, as parameterized by the macro CCB_DUMP_ALL_FRAMES, specifies that data-frames should be delivered indefinitely.

```
#define CCB_DUMP_ALL_FRAMES 0
```

ccb_queue_load_driver_cmd() – Queuing a load-driver command

Load either the real CCB device driver, or the CCB simulation device driver.

```
typedef enum {
  CCB_NORMAL_DRIVER,  /* The real CCB device-driver */
  CCB_VIRTUAL_DRIVER  /* The CCB simulation device-driver */
} CCBDriverType;

int ccb_queue_load_driver_cmd(CCBClientLink *cl, long id,
                              CCBDriverType type);
```

ccb_queue_monitor_cmd() – Queuing a monitor command

monitor commands specify how frequently messages containing monitoring should be sent to the manager over the telemetry stream.

```
int ccb_queue_monitor_cmd(CCBClientLink *cl, long id,
                          unsigned short period);
```

The arguments of this function are interpreted as follows.

- **The monitoring period** (period)

  This argument specifies the interval between monitoring updates, expressed as an integer multiple of the integration period.

ccb_queue_telemetry_cmd() – Queuing a telemetry command

telemetry commands specify which telemetry streams are to be sent to the manager. A telemetry command is sent by first calling ccb_queue_telemetry_cmd() to queue the command for dispatch, then subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_telemetry_cmd(CCBClientLink *cl, long id,
                            unsigned short streams);
```

The arguments of this function are interpreted as follows.

- **The telemetry data-stream selection** (streams)

  This parameter contains a bit-wise union of CCBTelemetryStream enumerators, specifying which streams should be sent to the manager.

  ```
  typedef enum {
    CCB_INTEG_STREAM = 1,   /* The stream of integrated data */
    CCB_MONITOR_STREAM = 2, /* The stream of monitoring data */
    CCB_LOG_STREAM = 4,     /* The stream of log messages */
    CCB_NO_STREAMS = 0,     /* None of the above streams */
  /* All of the above streams */
    CCB_ALL_STREAMS = CCB_INTEG_STREAM | CCB_MONITOR_STREAM |
                      CCB_LOG_STREAM
  } CCBTelemetryStream;
  ```

  For example, if the manager is interested in receiving all types of telemetry, the argument of this command should be CCB_ALL_STREAMS.

ccb_queue_logger_cmd() – Queuing a logger command

logger commands configure the log-message dispatcher in the CCB telemetry server.

```
int ccb_queue_logger_cmd(CCBClientLink *cl, long id,
                         unsigned long period);
```

The arguments of this function are interpreted as follows.

- **The log-history purging interval** (period)

  As discussed later (see page 84), a record of historically sent log messages is used to prevent repeated messages from being sent to the manager. The period argument specifies how often this historical record should be purged, expressed as an integer number of seconds, and thus the minimum time between repeated messages being queued to be sent to the manager.

ccb_queue_reset_cmd() – Queuing a reset command

When a manager first connects to the CCB, the server resets both itself, the CCB device-drivers and the CCB hardware to a default state; such that the manager always sees this same state when it first connects. Thereafter the CCB can be returned to this state either by disconnecting and reconnecting to the CCB server, or by sending a reset command.

On receiving a reset command, the CCB server first unloads, then reloads the CCB device-drivers. This not only resets the device drivers, but also resets the CCB hardware. The CCB server then turns off all telemetry except log messages, and starts a dummy initial intra-scan with a scan ID of 0.

A reset command is sent by first calling ccb_queue_reset_cmd() to queue the command for dispatch, then by subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_reset_cmd(CCBClientLink *cl, long id);
```

ccb_queue_ping_cmd() – Queuing a ping command

On receiving this command the CCB server replies to the manager with a cntrl-ping-reply message over the control connection, and a telem-ping-reply message over the telemetry connection.

ping commands are sent by first calling ccb_queue_ping_cmd() to queue the command for dispatch, then by subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_ping_cmd(CCBClientLink *cl, long id);
```

The manager can subsequently check whether replies to this ping were received by calling the ccb_ping_echos() command.

```
unsigned ccb_ping_echos(CCBClientLink *cl);
```

This function returns a bitwise union of CCBLinkType enumerators, denoting the set of links over which replies to the most recent ping command, have been received.

```
typedef enum {
  CCB_CNTRL_LINK = 1,  /* The link to the CCB telemetry server */
  CCB_TELEM_LINK = 2,  /* The link to the CCB control server */
  CCB_ALL_LINKS = CCB_CNTRL_LINK | CCB_TELEM_LINK;
} CCBLinkType;
```

Provided that the manager waits for a reasonable amount of time between sending a ping command and checking for its echos, then the ccb_ping_echos() function should return CCB_ALL_LINKS. If not, then one or both of the server connections are down for some reason.

Ping commands are designed to be used as follows. Every few minutes the manager should first call ccb_ping_echos() to see if replies were received from the last ping command, and then call ccb_queue_ping_cmd() to send a new ping command. If ccb_ping_echos() doesn't return CCB_ALL_LINKS, then the manager should advise the operator that something has gone wrong. To facilitate this usage, if ccb_ping_echos() is called before the first ping command has been sent over a newly established connection, CCB_ALL_LINKS is returned. Thus ccb_ping_echos() can always be called just before ccb_queue_ping_cmd(), without reporting a bogus link problem at startup.

ccb_queue_status_request_cmd() – Queuing a status-request command

On receiving this command, the CCB server queues a status-reply message to be sent back to the manager over the control connection. This reply, which is documented later, reports on the health of the CCB.

status-request commands are sent by first calling ccb_queue_status_request_cmd() to queue the command for dispatch, then subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_status_request_cmd(CCBClientLink *cl, long id);
```

ccb_queue_shutdown_cmd() – Queuing a shutdown command

On receiving this command, the CCB server attempts to unload the CCB device drivers, which has the side effect of stopping all CCB interrupts, then initiates a computer-shutdown process, with the intention of both shutting down the operating system and switching off the real-time computer.

shutdown commands are sent by first calling ccb_queue_shutdown_cmd() to queue the command for dispatch, then by subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_shutdown_cmd(CCBClientLink *cl, long id);
```

ccb_queue_reboot_cmd() – Queuing a reboot command

On receiving this command the CCB server attempts to unload the CCB device drivers, which has the side effect of stopping all CCB interrupts, then initiates a reboot of the real-time computer.

reboot commands are sent by first calling ccb_queue_reboot_cmd() to queue the command for dispatch, then subsequently calling ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_reboot_cmd(CCBClientLink *cl, long id);
```

ccb_queue_set_dacs_cmd() – Queuing a set-dacs command

This command changes one or more of the miscellaneous analog outputs of the GPIO board. These outputs aren't used for anything by the CCB, but can be used to generate test voltages that can then be fed back into the CCB using custom cables.

To send a set_dacs command, first call ccb_queue_set_dacs_cmd() to queue the command for dispatch, then subsequently call ccb_client_communicate() to send it to the CCB server.

```
int ccb_queue_set_dacs_cmd(CCBClientLink *cl, long id,
                           unsigned short counts[CCB_GPIO_NDAC]);
```

The counts[] argument is an array of the count inputs to the four digital-to-analog converters on the GPIO card. Since the GPIO card latches changes to all of the outputs simultaneously, it makes sense to allow all of the outputs to be changed in one function call. To leave a given output voltage unchanged, its element in the counts[] argument should be set to the special value, CCB_GPIO_LAST_DAC_COUNT, which is defined to hold a value that is outside the input-count range of the digital-to-analog converter.

Note that since the current GPIO card has 4 analog outputs, the CCB_GPIO_NDAC macro is defined as follows:

```
#define CCB_GPIO_NDAC 4
```

The minimum and maximum count values that the digital-to-analog converters accept are parameterized as:

```
#define CCB_GPIO_MIN_DAC_COUNT 0
#define CCB_GPIO_MAX_DAC_COUNT 4095
```

The corresponding minimum and maximum output voltages are parameterized as:

```
#define CCB_GPIO_MIN_DAC_VOLTS 0.0
#define CCB_GPIO_MAX_DAC_VOLTS 5.0
```

Note that the validity of these numbers requires that the jumpers on the GPIO-board remain in their factory-default positions.

The following examples show how to use existing macros to convert between DAC input-counts and output voltages.

```
unsigned short count = 1200;  /* A DAC input count */
float volts = 3.3;            /* A DAC output voltage */
...
count = CCB_GPIO_VOLTS_TO_COUNTS(volts);
...
volts = CCB_GPIO_COUNTS_TO_VOLTS(count);
```

To feed the output voltage of the first DAC output channel into a CCB ADC-input, it is necessary to swap the normal ribbon cable that connects the GPIO-card to the CPU board, with a test ribbon cable that breaks out the DAC's output voltage to a coaxial cable, terminated with a BNC connector. This BNC connector is then plugged into the unterminated input of the test detector-module, and the differential output of the test detector module plugged into a CCB input port. The differential input voltage that arrives at the CCB input port, for a given DAC output-voltage, can be calculated with the following macro:

```
differential_voltage = CCB_TEST_DETECTOR_S2D(single_ended_voltage);
```

where the S2D part of the macro-name stands for *single-ended to differential*.

Similarly the opposite conversion is performed as follows:

```
single_ended_voltage = CCB_TEST_DETECTOR_D2S(differential_voltage);
```

Beware that the test-detector input can't tolerate DAC output-voltages outside of the range 0-1V. The corresponding range of differential outputs matches the range that is supported by the CCB inputs.

Thus to generate a differential input voltage of ccb_volts at the chosen CCB input port, via the test detector-module, one would need to command a DAC input count that was calculated as follows.

```
dac_count = CCB_GPIO_VOLTS_TO_COUNTS(CCB_TEST_DETECTOR_D2S(ccb_volts));
```

Note that the CCB inputs can only tolerate differential input voltages within the range specified by the following two macro definitions in ccbconstants.h.

```
#define CCB_MIN_DIFF_INPUT (-2.5)  /* volts */
#define CCB_MAX_DIFF_INPUT  (2.5)  /* volts */
```

Thus the ccb_volts argument in the preceding example should be kept between these two limits.

4.12 Incoming control-link replies

This section describes the library functions that are used by the manager to register callback functions for ccb_client_communicate() to subsequently use to deliver control-link ping and status-reply messages.

Each of the callback-registration functions returns an integer, which is 0 on success and non-zero otherwise. On failure, errno is set according to the error. The manager's callback functions are also required to return an integer, which should be 0 on success and 1 on failure. When a callback reports an error in this way, it should also set errno appropriately, so that when ccb_client_communicate() responds to this by returning non-zero, the manager can inspect errno to see what happened.

A summary of the possible control-link replies is given in the following table, along with the names by which they are referred to elsewhere in the text.

Name	Description
cntrl-ping-reply	A reply to a ping command.
status-reply	A reply to a status-request command.

Along with each callback function, the manager can specify an arbitrary `void *` pointer to be passed to the callback function whenever it is called. This should be used by the manager to pass the callback function any resources that it needs when handling the corresponding reply message. In addition to this pointer, each callback function is passed a pointer to the CCBClientLink object that received the message, plus any arguments corresponding to the contents of the message.

ccb_status_reply_callback() – Routing status-request replies

The public ccb_status_reply_callback() function is used to register the callback function that will subsequently be called by ccb_client_communicate() whenever it receives a status-reply message.

The CCB_STATUS_REPLY_FN macro should be used for the declarations and prototypes of suitable callback functions, and the CCBStatusReplyFn typedef can be used for recording pointers to them.

```
  #define CCB_STATUS_REPLY_FN(fn) int (fn)(CCBClientLink *cl, \
                                           void *data, \
                                           unsigned long status)

  typedef CCB_STATUS_REPLY_FN(CCBStatusReplyFn);

  int ccb_status_reply_callback(CCBClientLink *cl,
                                CCBStatusReplyFn *fn, void *data);
```

The callback function is registered via the fn argument of ccb_status_reply_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. The contents of the message are passed to the callback via the status argument, which reports the overall health of the CCB software and hardware. This is represented by a bit-wise union of CCBGeneralStatus enumerators, each of which represents the value of a single bit within the status argument.

```
typedef enum {
  CCB_LINK_DOWN   = 1,  /* The telemetry link is down */
  CCB_BUFFER_FULL = 2,  /* The telemetry output buffer filled */
                        /*  up and hasn't drained yet, so data */
                        /*  are being discarded. */
  CCB_HARD_FAULT  = 4,  /* A hardware fault has been detected */
  CCB_SOFT_FAULT  = 8   /* A software fault has been detected */
} CCBGeneralStatus;
```

Beware that unless care is taken to subsequently recompile every component of the system (and update this documentation), none of the existing values in this enumeration should either be removed or have their values changed. If necessary, new enumerators can be appended with the next highest unused power-of-2 value, and to support this possibility all software that uses these values should not assume anything about the values of currently undefined bits.

4.13 Incoming telemetry messages

As described above for incoming control-link ping and status reply messages, incoming telemetry messages from the CCB server are delivered to the manager via callback functions. These are invoked by the ccb_client_communicate().

Each of the callback-registration functions returns an integer, which is 0 on success and non-zero otherwise. On failure, errno is set according to the error. The manager's callback functions are also required to return an integer, which should be 0 on success and 1 on failure. When a callback reports an error in this way, it should also set errno appropriately, so that when ccb_client_communicate() responds to this by returning non-zero, the manager can inspect errno to see what happened.

A summary of the possible telemetry messages is given in the following table, along with the names by which they are referred to elsewhere in the text.

Name	Description
monitor-data	Instrumental monitoring data
integ-data	Integrated radiometer data
log-message	CCB log messages
telem-ping-reply	Telemetry-link replies to ping commands

The following table indicates the buffering and prioritization of these messages. Messages with higher priority values are sent before lower priority messages.

Message type	Priority	Queue length	Queue overflow disposition
monitor-data	0	1 message	Overwrite the previous unsent message
integ-data	1	3MB (>= 10s)	Allow the queue to drain
log-message	2	100 messages	Overwrite the oldest unsent message
telem-ping-reply	3	1 message	Overwrite the previous unsent message

As can be seen, replies to ping commands are given the highest priority, since they are time sensitive. There is no need to queue these messages, since they contain no information, so the output queue only has a single entry, which is overwritten every time that a new telem-ping-reply reply is requested.

log-message messages have the second highest priority, to prevent important messages from being held up indefinitely. The queuing strategy for log messages is complicated by the need to prevent rapidly repeating messages from consuming too much memory and bandwidth. Detecting repeating messages is further complicated by the fact that a given message-reporting statement can include changeable content in its messages, such as IP addresses, errno information and problematic values. The logging strategy adopted by the CCB server is thus as follows. In addition to a fixed size queue of outgoing log messages, the server maintains a periodically purged table containing the checksums of recently generated log messages. The table of checksums records the checksums of up to CCB_MAX_LOG_VARIANTS different messages for each logging statement. Before appending a message to the queue of outgoing log messages, the CCB server first checks to see if, since the last time that this checksum-table was cleared, the originating statement has either already reported the same message,

or has generated an excessive number of varying messages. The message is not queued if either of these conditions are true.

By default, the table of historical checksums is cleared by the library every CCB_LOG_PURGE_DT seconds, such that a repeated message sent after this time interval again be reported. Thus within each period of CCB_LOG_PURGE_DT seconds, up to CCB_MAX_LOG_VARIANTS unique messages per logging statement are reported to the manager.

The interval at which the table of checksums is cleared can be changed from its default by sending a logger control command.

integ-data messages have the next highest priority. They are stored in a large, fixed sized ring buffer, with sufficient room to bridge reasonable periods of network congestion. If the observer selects such a short integration period that the buffer becomes full; rather than new messages overwriting old messages in the ring buffer, new messages are thrown away until the buffer has completely drained. This potentially supports short periods of contiguous data-taking at high data rates, interleaved with gaps when no data are recorded.

Finally, monitor-data messages have the lowest priority, since they are only intended as a visual indication of the instantaneous health of the system. Old monitor values aren't very useful, so the output buffer of unsent monitoring messages is only one message long, and if a new monitor message is generated before the old one has been queued for transmission, the old one is simply discarded and replaced with the new one.

The following sections describe the library functions used by the manager to register callback functions for ccb_client_communicate() to subsequently use to deliver telemetry messages.

All telemetry message callback functions have 3 arguments in common, these being the CCBClientLink object that received the message, arbitrary application-supplied callback data, and a pointer to a CCBTimeStamp structure, which reports the date and time at which the message was originally generated.

ccb_monitor_msg_callback() – Routing telemetry monitor-data messages

Instrumental monitoring data are sent to the manager over the telemetry link, at the end of every monitor_interval'th integration, in a monitor-data message.

The public ccb_monitor_msg_callback() function is used to register the callback function that will subsequently be called by ccb_client_communicate() whenever it receives a monitor-data message.

The CCB_MONITOR_MSG_FN macro should be used for the declarations and prototypes of suitable callback functions, and the CCBMonitorMsgFn typedef can be used for recording pointers to them.

```
#define CCB_MONITOR_MSG_FN(fn) int (fn)(CCBClientLink *cl, void *data, \
                                        const CCBTimeStamp *ts, \
                                        unsigned long scan, \
                                        unsigned long number, \
                                        CCBMonitorData *md)

typedef CCB_MONITOR_MSG_FN(CCBMonitorMsgFn);

int ccb_monitor_msg_callback(CCBClientLink *cl, CCBMonitorMsgFn *fn,
                             void *data);
```

The callback function is registered via the fn argument of ccb_monitor_msg_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. The scan argument identifies the parent scan, and has the value that the manager specified in the stop-scan or start-scan command that initiated the originating scan or intra-scan. The number argument is the sequential number of the monitoring message within the current scan, starting from 0. The manager can use this to check for discarded monitor messages.

The md argument is a pointer to a CCBMonitorData object, which contains the latest monitoring information. This is interpreted as follows.

```
typedef struct {
  float fan12v;           /* The 12V fan PSU voltage */
  float a8v;              /* The analog 8V PSU voltage */
  float d5v;              /* The digital 5V PSU voltage */
  int cnf_done;           /* True if all of the FPGAs have */
                          /* been programmed. */
  int high_temp;          /* True if the CCB enclosure is too hot */
  int ccb_id;             /* The 2-bit ID of the CCB */
  CCBFpgaMonitorData fpga[CCB_NUM_FPGA]; /* Monitoring data for each */
                          /*  FPGA, where index 0 refers to the */
                          /*  master FPGA, and indexes 1-4 refer to */
                          /*  the four slave FPGAs */
} CCBMonitorData;
```

Where CCB_NUM_FPGA is a macro that specifies the number of FPGAs in the CCB, including both the master and slave FPGAs. It is defined as:

```
#define CCB_NUM_SLAVE 4                /* The number of slave FPGAs */
#define CCB_NUM_FPGA (CCB_NUM_SLAVE+1) /* The total number of FPGAs */
```

The ccb_id field reports the numeric ID that is associated with the cable-harness that is plugged into the front-panel of the CCB. Different cables are associated with the following receivers and locations, as shown in table 4.1. These associations are stored in the ccb_ip_addresses file, which is installed in the CCB configuration directory.

Cable ID	IP address	Association
0	ccblab.gb.nrao.edu	The lab (or no cable)
1	ccb1cm.gbt.nrao.edu	The 1cm receiver
2	ccb3mm.gbt.nrao.edu	The 3mm receiver
3	ccbspare.gb.nrao.edu	The spare lab cable

Table 4.1: The ID numbers of the CCB cables

The fpga array holds information that is specific to each FPGA in the, CCB, starting with that of the master FPGA, followed by that of the CCB_NUM_SLAVE slave FPGAs. Each element provides monitoring information for one FPGA, and is defined as follows.

```
typedef struct {
  float d1_2v;   /* The digital 1.2V PSU voltage */
  float d2_5v;   /* The digital 2.5V PSU voltage */
  float d3_3v;   /* The digital 3.3V PSU voltage */
  float a5v;     /* The analog 5V PSU voltage */
  float hb;      /* The mean heartbeat voltage */
  int cnf_error; /* True if the FPGA the firmware failed */
                 /*  the checksum test. */
  int cnf_done;  /* True if the FPGA is programmed */
} CCBFpgaMonitorData;
```

Note that the contents of CCBMonitorData objects can be displayed on stdout by calling the ccb_display_monitor_data() function. This has the following prototype:

```
int ccb_display_monitor_data(CCBMonitorData *cal);
```

If successful, this function returns 0. Otherwise it returns 1.

ccb_integ_msg_callback() – Routing telemetry integ-data messages

Integrated data are sent to the manager in integ-data messages, at the end of each integration.

The public ccb_integ_msg_callback() function is used to register the callback function that will subsequently be called by ccb_client_communicate() whenever it receives an integ-data message. The

The CCB_INTEG_MSG_FN macro should be used for the declarations and prototypes of suitable callback functions, and the CCBIntegMsgFn typedef can be used for recording pointers to them.

```
#define CCB_INTEG_MSG_FN(fn) int (fn)(CCBClientLink *cl, \
                                      void *data, \
                                      const CCBTimeStamp *ts, \
                                      unsigned long scan, \
                                      unsigned long number, \
                                      unsigned short flags, \
                                      const unsigned long *values,\
                                      unsigned nvalues)

typedef CCB_INTEG_MSG_FN(CCBIntegMsgFn);

int ccb_integ_msg_callback(CCBClientLink *cl, CCBIntegMsgFn *fn,
                           void *data);
```

The callback function is registered via the fn argument of ccb_integ_msg_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. The arguments of this function contain the following information.

- **The scan identification number** (scan)

  The scan argument identifies the parent scan, and has the value that the manager specified in the stop-scan or start-scan command that initiated the originating scan or intra-scan.

- **The integration identification number** (number)

  The number argument is the sequential number of the integration within that scan, starting from 0. The manager can use the integration number to check for missing integ-data messages.

- **Single-bit status flags** (flags)

  Individual bits within the flags argument describe pertinent information about the hardware-status during the integration. This is a bit-wise union of power-of-2 values from the CCBIntegFlags enumeration, which is defined (in ccbcommon.h) as follows.

```
typedef enum {
  CCB_CAL_A_ON = 1,   /* Included if cal-diode A was on */
  CCB_CAL_B_ON = 2,   /* Included if cal-diode B was on */
  CCB_INTEG_OK = 4,   /* Included if the integration is usable */
  CCB_SLAVE0_OK = 8,  /* Included if slave-FPGA 0 was present */
```

```
    CCB_SLAVE1_OK = 16, /* Included if slave-FPGA 1 was present */
    CCB_SLAVE2_OK = 32, /* Included if slave-FPGA 2 was present */
    CCB_SLAVE3_OK = 64  /* Included if slave-FPGA 3 was present */
  } CCBIntegFlags;
```

Note that during integrations where the calibration diodes are in the process of switching to a new state, the cal-diode status bits denote the target states of the calibration diodes, but the CCB_INTEG_OK bit is omitted, to indicate that they aren't stable enough for the integration to be used. The slave-present status bits reflect corresponding single-bit signals that the master FPGA receives from the individual slave-FPGA boards. If a slave board is unplugged, fails to load its firmware, or has a fault that prevents it from driving this signal, then the corresponding slave-present status-bit is not included in the flags argument.

- **The integrated data-values** (values)

  The first nvalues elements of the array pointed to by the values[] argument, contain the radiometer integrations. For the 1cm and 3mm CCB's, the values in this array are ordered as indicated in table 4.2.

Array Index	Port Name	Phase Switch	
		B	A
0	J1	open	open
1	J1	open	closed
2	J1	closed	open
3	J1	close	closed
4	J2	open	open
5	J2	open	closed
6	J2	closed	open
7	J2	close	closed
⋮	⋮	⋮	⋮
60	J16	open	open
61	J16	open	closed
62	J16	closed	open
63	J16	closed	closed

Table 4.2: The ordering of the integrated data

In other words the 4 phase-switch bins of the port that is labeled J1 on the front-panel of the CCB, are followed by those of port J2, and so-on, up to the final port, J16.

Each element within this array is a 32-bit accumulation of samples within a single phase-switch bin. If this accumulation overflowed, or one of the ADC samples that contributed to it, overflowed, then the accumulation is flagged by replacing it with the special value CCB_SATURATED_INTEGRATION. This is the value with all 32-bits set to 1, and is thus defined as follows.

```
#define CCB_SATURATED_INTEGRATION 0xFFFFFFFFUL
```

- **The number of integrated data-values** (nvalues)

  This argument specifies the number of significant elements at the start of the values array. Currently this is always equal to CCB_MAX_INTEG, which is defined as follows.

  ```
  #define CCB_MAX_INTEG 64  /* The maximum number of total-power */
                            /* measurements from any instrument */
  ```

  Beware that although the nvalues argument currently has a fixed value, it was included just in case another instrument was subsequently built that had a different number of continuum channels.

ccb_log_msg_callback() – Routing telemetry log-message messages

When the server sends error and informational messages to the manager, to be logged, they are sent as log-message messages over the telemetry link.

The public ccb_log_msg_callback() function is used to register the callback function that will subsequently be called by ccb_client_communicate() whenever it receives a log-message message. Since the same callback function is invoked whenever the client end of the communications library needs to report an internal error, it is recommended that ccb_log_msg_callback() be the first CCB library function called after new_CCBClientLink() returns. Otherwise some error messages may end up being reported to the manager program's stderr, which may not be visible to the observer.

The CCB_LOG_MSG_FN macro should be used for the declarations and prototypes of suitable callback functions, and the CCBLogMsgFn typedef can be used for recording pointers to them.

```
#define CCB_LOG_MSG_FN(fn) int (fn)(CCBClientLink *cl, \
                                    void *data, \
                                    const CCBTimeStamp *ts, \
                                    const char *msg, \
                                    unsigned long id, \
                                    CCBLogLevel level)

typedef CCB_LOG_MSG_FN(CCBLogMsgFn);

int ccb_log_msg_callback(CCBClientLink *cl, CCBLogMsgFn *fn, void *data);
```

The callback function is registered via the fn argument of ccb_log_msg_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. The log message itself is passed as a normal '\0' terminated C string, via the msg argument, and the corresponding unique numeric identifier of the message is passed in the id argument. The level argument reports the significance of the message, as enumerated by the CCBLogLevel type.

```
typedef enum {
  CCB_INFO,      /* A purely informational message */
  CCB_NOTICE,    /* A note about a probably inconsequential event */
  CCB_WARNING,   /* A warning about a potentially problematic event */
  CCB_ERROR,     /* A report of an event requiring operator attention */
  CCB_FAULT,     /* A report of a condition that is corrupting data */
  CCB_FATAL      /* A report of a system-wide failure */
} CCBLogLevel;
```

Note that these enumerators simply provide symbolic names for the level values defined by YGOR.

4.14 A TCL wrapper around the CCB client API

Ostensibly for the purpose of facilitating a GUI demonstration CCB client using Tcl/Tk, but also useful for quick test programs, a dynamically loadable Tcl wrapper interface is provided for the CCB client communications library. This can either be linked with directly by any program that embeds Tcl, or can be loaded into a running copy of the standard wish or tclsh shell programs that come with the Tcl/Tk distribution. To load the library into an already executing copy of wish, one types:

```
load ./libccbtclclient.so
```

Note that if the above library isn't in the current directory, the ./ component in the above should be replaced by the path of the directory where it is located. Alternatively, if the library is installed in one of the directories that are searched automatically by the run-time linker, then there is no need to specify a directory at all. The Tcl wrapper defines a single Tcl command called ccb. The first argument of this command is a sub-command, and must be one of the following.

- ccb connect *host*

This command initiates a non-blocking connection to the specified host. The host argument can either be a numeric IP address or a textual IP address. Before returning, this command registers the sockets that it opens with the Tcl event loop.

- ccb disconnect

  This terminates any existing connection to a CCB server, and withdraws the defunct sockets from the Tcl event loop.

- ccb send ...

  This command is the command responsible for queuing all commands destined for the remote CCB server. Its first argument identifies the type of control-command to be sent, and is followed by any arguments that the command requires. The following commands are defined.

  - ccb send start_scan *mjd seconds*

    This starts a new scan on the day specified by the Modified Julian Day number mjd, at the time of day specified by the seconds argument.

  - ccb send stop_scan

    This starts an intra-scan ASAP.

  - ccb send dump_scan *adc samples frames*

    Start a dump-mode intra-scan ASAP. The *adc* argument should be a number between 0 and 15, specifying the ADC whose samples are to be collected. The *samples* argument should be a positive integer specifying the maximum number of samples that should be collected per integration, or the word max to specify that the maximum number possible be collected. The *frames* argument should be a positive integer specifying how many per-integration data-frames should be delivered to the remote dump-mode clients, or the word all to specify that frames should be delivered indefinitely.

  - ccb send load_driver *type*

    Load either the real CCB device driver, or a driver that emulates the real device device driver and its hardware. The type argument can take any of the following values.

    ```
    normal virtual
    ```

    These names have the same meanings as the similarly named members of the CCBDriverType datatype.

  - ccb send reset

    This resets the CCB.

– ccb send ping

If any previously sent ping has not been responded to, this function throws an error (use the Tcl catch command to see this). Otherwise it sends a ping command to the CCB server.

– ccb send status_request

This asks the CCB server to send us a message reporting the status of the CCB backend. How the subsequent response is caught and responded to is documented below.

– ccb send shutdown

This tells the remote CCB server to place the CCB electronics in a safe state, then shutdown the backend computer.

– ccb send reboot

This tells the remote CCB server to place the CCB electronics in a safe state, then reboot the backend computer.

– ccb send monitor *period*

This command configures the frequency of monitoring updates. The period argument must be an integer specifying the monitoring period as a number of integration periods.

– ccb send telemetry *streams*

This command tells the CCB server which telemetry streams it should send to us. Specifically, the streams argument specifies the set of telemetry streams that the CCB server should continue to send, expressed as a space separated list of zero or more of the following names.

```
integ_stream  monitor_stream  log_stream
no_streams  all_streams
```

These correspond to the similarly named CCBTelemetryStream enumerators (see page 77).

– ccb send logger *period*

This command tells the CCB server how often to discard the history of sent log messages, thus specifying the maximum rate at which repeated log messages will be sent to us.

– ccb send set_dacs *counts0 counts1 counts2 counts3*

This command tells the CCB server to set the miscellaneous analog outputs of the GPIO card to specified values, where the values are interpreted as the integer count inputs of the corresponding digital-to-analog converters. There are 4 analog outputs, so up to 4 arguments are required. Each argument should either be an integer DAC count between 0 and 4095, or the word last, to request that the specified DAC input not be changed. More details about the analog outputs can be found in section 4.11.1.

- ccb configure ...

  This command configures specified parameters of the next scan or intra-scan. Its first argument identifies the group of configuration parameters to be modified, and this is followed by the corresponding configuration values. The following configuration commands are defined.

  – ccb configure phase_switches *active_switches closed_switches samp_per_state*

    This configures the phase-switches in the receiver front-end. The arguments have the same meanings as the synonymous arguments of the ccb_set_phase_switch_cnf() function. The first 2 arguments, which both refer to sets of phase-switches, are expressed as space-separated lists of zero or more of the following names.

    ```
    switch_a  switch_b  no_switches  all_switches
    ```

    These have the same meanings as the similarly named CCBPhaseSwitches enumerators.

    The final, samp_per_state argument is expressed as an integer.

  – ccb configure cal_diode *ncal diode_states diode_times*

    This configures the calibration-diodes in the receiver front-end. The arguments have the same meanings as the synonymous arguments of the ccb_set_cal_diode_cnf() function.

    The diode_states argument is represented as a Tcl list of calibration-diode sets, each of which is expressed as a Tcl list of zero or more of the following names.

    ```
    diode_a  diode_b  no_diodes  all_diodes
    ```

    These have the same meanings as the similarly named CCBCalDiodes enumerators.

    The diode_times argument is represented as a Tcl list of integers.

    Note that the number of elements in the diode_states and diode_times arguments must both be at least equal to the value of the integer ncal argument.

  – ccb configure timing *phase_switch_dt diode_rise_dt diode_fall_dt integ_period roundtrip_dt holdoff_dt adc_delay_dt*

    This configures the parameters which affect the timing of integrations. The arguments are all integers and have the same interpretations as the synonymous arguments of the ccb_set_timing_cnf() function.

  – ccb configure sampler *sample_type*

    This configures the parameters which affect the digitized samples that are input to the integrators and collected verbatim during dump-mode. The *sample_type* argument determines where the hardware gets the samples that it collects and integrates. It should be one of the following names.

    ```
    adc_samples fake_samples
    ```

which have the same meanings as their similarly named CCBSampleType enumerator counterparts.

- ccb attach ...

This command specifies a Tcl command that should be invoked when a given event occurs, such as the reception of a particular type of message from the CCB server. When the event next occurs, the specified Tcl command is invoked verbatim, without any arguments being appended. Where information is associated with the event, the specified Tcl command can use the ccb get ... commands, documented shortly, to get that information.

The type of event to attach the Tcl command to is specified via the fist argument of the ccb attach command, and the Tcl command that is to subsequently be invoked by the event, is specified as the second argument. The possible events are as follows.

  - ccb attach status *command*

    Whenever a reply to a ccb send status_request command is received the specified Tcl command is executed. This Tcl command can use the ccb get status command to retrieve the corresponding status information.

  - ccb attach monitor *command*

    Whenever a new packet of monitoring data is received, the specified Tcl command is invoked. This Tcl command can use the ccb get monitor command to retrieve the received monitoring data.

  - ccb attach integ *command*

    Whenever data are received from a newly completed integration, the specified Tcl command is invoked. This Tcl command can use the ccb get integ command to retrieve the integrated data.

  - ccb attach log *command*

    Whenever a log message is received from the CCB server or the CCB client library, the specified Tcl command is invoked. This Tcl command can use the ccb get log command to retrieve the log message.

  - ccb attach error *command*

    If a command that was sent to the CCB fails for any reason, the the specified Tcl command is invoked. This Tcl command can use the ccb get error command to retrieve the problematic completion status of the original command.

- ccb get ...

This command provides a means of querying information from the Tcl wrapper. Notably it allows one to get the contents of the last of each of a number of types of message received from the CCB server.

The single argument of the ccb get command specifies what type of information is to be retrieved. The retrieved data is passed back as the result string of the command. For those not familiar with Tcl, the result string is written to stdout if the command is typed in at the command-line of wish or tclsh, or it can be interpolated into an argument of another Tcl command by invoking it between square brackets.

The available information requests are as follows.

- ccb get status

  This retrieves the most recent CCB status that has been received in response to a preceding ccb send status_request command. The result string is a space separated list of zero or more of the following status indicators.

  ```
  link_down  buffer_full  hard_fault  soft_fault
  ```

  These correspond to the similarly named CCBGeneralStatus enumerators (see page 83).

- ccb get monitor

  This returns the most recently received batch of periodically sampled monitoring data. The result string contains a space-separated list of the following items:

  1. mjd - The date at which the message was generated, expressed as a Modified Julian day number
  2. sec - The time at which the message was generated, expressed as the number of complete seconds that had elapsed since the start of the above day.
  3. ns - The fractional-seconds part of the time, expressed as an integer number of nanoseconds.
  4. scan - The scan-identification number that was sent with the start-scan or stop-scan command that initiated the originating scan.
  5. number - The sequential number of the integration within the originating scan.
  6. fan12v - The actual voltage of the 12V fan PSU.
  7. a8v - The actual voltage of the analog 8V PSU.
  8. d5v - The actual voltage of the digital 5V PSU.
  9. cnf_done - A boolean value specifying whether all of the FPGAs have had firmware downloaded into them. It has one of the values true or false.
  10. high_temp - A boolean value specifying whether the temperature inside the CCB is too high. It has one of the values true or false.
  11. ccb_id - The ID of the cable-harness that is plugged into the front-panel of the CCB. See table 4.1, for details.
  12. master_info - A brace-enclosed array of monitoring information that pertains to the master FPGA board (see below).
  13. slave1_info - A brace-enclosed array of monitoring information that pertains to the first slave FPGA board (see below).

14. slave2_info - A brace-enclosed array of monitoring information that pertains to the second slave FPGA board (see below).
15. slave3_info - A brace-enclosed array of monitoring information that pertains to the third slave FPGA board (see below).
16. slave4_info - A brace-enclosed array of monitoring information that pertains to the fourth slave FPGA board (see below).

The master_info, slave1_info, slave2_info, slave3_info and slave4_info fields, are each brace-enclosed lists of the following items:

1. d1_2v - The actual voltage of the digital 1.2V PSU.
2. d2_5v - The actual voltage of the digital 2.5V PSU.
3. d3_3v - The actual voltage of the digital 3.3V PSU.
4. d3_3v - The actual voltage of the analog 5V PSU.
5. hb - The mean voltage of the heartbeat signal.
6. cnf_done - A boolean value specifying whether firmware has been loaded into the FPGA. It has one of the values true or false.
7. cnf_error - A boolean value specifying whether the firmware that was downloaded into the FPGA is corrupt. It has one of the values, true or false.

– ccb get integ

  This returns the integrated data from the most recently completed integration period. The result string contains a space-separated list of the following integers:

  1. mjd - The date at which the message was generated, expressed as a Modified Julian day number
  2. sec - The time at which the message was generated, expressed as the number of complete seconds that had elapsed since the start of the above day.
  3. ns - The fractional-seconds part of the time, expressed as an integer number of nanoseconds.
  4. scan - The scan-identification number that was sent with the start-scan or stop-scan command that initiated the originating scan.
  5. number - The sequential number of the integration within the originating scan.
  6. flags - This is a Tcl list of zero or more of the following integration status-flags:

     ```
     cal_a_on, cal_b_on, integ_ok, slave0_ok,
     slave1_ok, slave2_ok, slave3_ok
     ```

     These flag-names have the same meanings as their similarly named CCBIntegFlags enumerator counterparts.
  7. nvalues - The number of integrated values.
  8. values... - The nvalues integrated values.

– ccb get log

  This returns the most recently received log message. The result string is a space separated list of the following items.

1. mjd - The date at which the message was generated, expressed as a Modified Julian day number
2. sec - The time at which the message was generated, expressed as the number of complete seconds that had elapsed since the start of the above day.
3. ns - The fractional-seconds part of the time, expressed as an integer number of nanoseconds.
4. id - The identifier of the error-reporting statement that generated the message.
5. level - One of the following words, indicating the significance of the message.

   `info  notice  warning  error  fault  fatal`

   These correspond to the similarly named CCBLogLevel enumerators (see page 91).
6. text - The log-message itself, rendered as a properly formed Tcl list element.

– ccb get error

  This retrieves the error completion status of the last control command that suffered an error. The result string contains one of the following values.

  `accepted  garbled  ignored  syserr`

  These correspond to the similarly named CCBCmdStatus enumerators (see page 70). Note that since the CCB library only tells us the completion statuses of commands that fail, when this command returns the word accepted, this means that no command has failed yet.

– ccb get time

  Return the current date and time as two integers, the first being the date as a Modified Julian Day number, and the second being the time of day, expressed as the number of seconds elapsed since the start of the day.

The following is a short example Tcl script, giving an overview of how to use the interface. To try this script, cut and paste it into a file called tcl_demo, then type:

```
tclsh tcl_demo
```

```
# Load the CCB Tcl interface.

load libccbtclclient.so

# Arrange for messages that are received from the CCB server to to be
# displayed to stdout. Note that the Tcl puts command is like C's
# puts() function, and that in Tcl, sub-strings consisting of square
# brackets surrounding Tcl commands are replaced by the output that
# is generated by executing those commands.
```

```
ccb attach integ {puts "Integration: [ccb get integ]"}
ccb attach monitor {puts "Monitor: [ccb get monitor]"}
ccb attach log {puts "Log: [ccb get log]"}
ccb attach error {puts "Error: [ccb get error]"}
ccb attach status {puts "Status: [ccb get status]"}

# Turn on both phase switches, and configure 250 samples per
# phase-switch state (ie. 25us).

ccb configure phase_switches all_switches all_switches 250

# Change the default timing, to slow down integration periods from
# the default of 1ms to 1s (ie. 40000 * 25us = 1s).

ccb configure timing 10 1000 1000 40000 10 7 0

# Queue a stop-scan command to be sent, along with the above changed
# configuration, once a connection is established to the CCB server.

ccb send stop_scan

# Connect to the CCB server on the local machine.

ccb connect localhost

# Load the hardware-simulating device driver.

ccb send load_driver virtual

# The CCB starts out with all telemetry disabled, so send the
# to enable all telemetry.

ccb send telemetry all_streams

# Start the Tcl event loop.

set ::guard 0
vwait ::guard
```

Before running this, make sure that libccbtclclient.so is in the normal run-time shared-library path, or add that directory to your LD_LIBRARY_PATH variable. Also, of course, first run the ccbserver program, so that the script has something to talk to, after making sure that the IP address of the host that you run the script on is in the ccb_authorized_ips file.

Chapter 5

The CCB server communications API

The CCB server-communications library performs most of the work needed to implement the CCB server. It basically acts as a gateway between the manager and both the CCB device driver and the operating system. Writing a complete server involves providing callback functions that load and unload the device driver, send commands to the device driver, and reboot and shutdown the real-time CPU, along with a select() based event loop, controlled by the library.

5.1 Include files

The datatype-declarations, function-prototypes and constants of the public API of the CCB-server communications-library are contained in the following include files.

- ccbserverlink.h

  This header-file contains all of the public function-prototypes and datatype declarations that are specific to to the server side of the communications link.

- ccbcommon.h

  This header-file contains the public function-prototypes and datatype declarations that are shared between both the client and the server communications libraries. Since this function is included by ccbserverlink.h, it isn't usually necessary for the application to explicitly include it.

- ccbconstants.h

  This header-file contains all of the constants that affect the operation of the communications link. Since this function is included by ccbcommon.h, it isn't usually necessary for the application to explicitly include it.

5.2 The CCB-server communications library

The library that implements the CCB-server communications API, is a shared library called libccbserverlink.so. Under Solaris and Linux, this filename is actually a symbolic link to the most recent version of the library.

Among other advantages, the use of a shared library rather than a static library has the benefit, at least under Solaris and Linux, of allowing one to restrict which symbols are exported into the namespace of the application. This not only prevents programs from using unstable private interfaces, but also greatly reduces namespace pollution and the possibility of symbol-name clashes.

Linking a C program with this library under either Linux or Solaris can be done as follows.

```
gcc -o foo *.o -lccbserverlink
```

Note that linkage instructions built into the shared library cause other unspecified libraries, such as -lsocket under Solaris, to be linked automatically.

5.3 Creating the resources used to communicate with CCB managers

The CCB server creates the resources that are needed for communications with the manager by calling new_CCBServerLink().

```
CCBServerLink *new_CCBServerLink(CCBServerDriver *normal,
                                 CCBServerDriver *virtual);
```

In addition to allocating resources, this binds the server to the CCB control and telemetry TCP/IP ports, whose numbers are parameterized, as mentioned earlier, by the CCB_CONTROL_PORT and CCB_TELEMETRY_PORT macros in ccbconstants.h. It doesn't wait for a manager to connect, but it does make the control and telemetry ports receptive to incoming connections, specifying a queue length of 1. Both ports are configured to be non-blocking, such that if a connection request is dropped between select() reporting activity, and accept() being called, the process doesn't block forever in accept(). The returned CCBServerLink object pointer is opaque, meaning that the definition of the structure that it points to is not exported to applications in the public header-file.

The arguments of this function are interpreted as follows.

- **The normal CCB device driver** (normal)

  The interface object of the real CCB device driver is passed in this argument. It must have been allocated by calling new_CCBServerDriver(). To facilitate development, this argument can be NULL, provided that the virtual argument isn't also NULL.

- **The simulated CCB device driver** (virtual)

  The interface object of a CCB device-driver emulator is passed in this argument. It must have been allocated by calling new_CCBServerDriver(). The emulator should simulate both the CCB device driver, and the CCB hardware. It is used for off-line testing of both the server and client software.

5.3.1 The CCB server's device-driver interface

The CCB server talks to a given CCB device driver via a set of method functions encapsulated in a corresponding CCBServerDriver object. Objects of this type are allocated by calling new_CCBServerDriver().

```
CCBServerDriver *new_CCBServerDriver(void *data,
                                     CCBDriverLoadFn *load_driver,
                                     CCBDriverUnloadFn *unload_driver,
                                     CCBDriverTellFn *tell_driver,
                                     CCBDriverSelectEventsFn *select_events,
                                     CCBDriverCheckEventsFn *check_events,
                                     CCBRebootRTCFn *reboot_rtc,
                                     CCBShutdownRTCFn *shutdown_rtc,
                                     CCBSetDacsFn *set_dacs);
```

The arguments of new_CCBServerDriver() are interpreted as follows.

- **Application specific callback data** (data)

  This argument is a pointer to any resources that the calling application needs to have passed to its callback functions.

- **The callback which loads the device driver** (load_driver)

  This argument specifies the function that the CCBServerLink object should call when it needs to load the CCB device driver. It is guaranteed that before the first call to this function, and thereafter between calls to this function, the driver will have been unloaded by calling the unload_driver callback function.

  Suitable functions to pass in the load_driver argument should be declared and prototyped using the CCB_DRIVER_LOAD_FN() macro. Pointers to them can be recorded in variables of type CCBDriverLoadFn.

```
#define CCB_DRIVER_LOAD_FN(fn) int (fn)(CCBServerLink *sl, \
                                        void *data)

typedef CCB_DRIVER_LOAD_FN(CCBDriverLoadFn);
```

When this function is called, it is passed the value of the data argument of new_CCBServerLink(). If successful, load_driver callbacks should return 0. Otherwise they should return 1, and set errno accordingly.

- **The callback which unloads the device driver** (unload_driver)

  This argument specifies the function that the CCBServerLink object should call when it needs to unload the CCB device driver. Beware that this function may be called when no driver is currently loaded, and that this shouldn't be interpreted as an error.

  Suitable functions to pass in the unload_driver argument should be declared and prototyped using the CCB_DRIVER_UNLOAD_FN() macro. Pointers to them can be recorded in variables of type CCBDriverUnloadFn.

```
#define CCB_DRIVER_UNLOAD_FN(fn) int (fn)(CCBServerLink *sl, \
                                          void *data)

typedef CCB_DRIVER_UNLOAD_FN(CCBDriverUnloadFn);
```

  When this function is called, it is passed the value of the data argument of new_CCBServerLink(). If successful, unload_driver callbacks should return 0. Otherwise they should return 1, and set errno accordingly.

- **The callback that controls the device driver** (tell_driver)

  This argument specifies the function that the CCBServerLink object should call when it needs to send a command to the CCB device driver. This function isn't called when the device driver isn't loaded.

  Suitable functions to pass in the tell_driver argument should be declared and prototyped using the CCB_DRIVER_TELL_FN() macro. Pointers to them can be recorded in variables of type CCBDriverTellFn.

```
#define CCB_DRIVER_TELL_FN(fn) int (fn)(CCBServerLink *sl, \
                            void *data, const CCBDriverCmd *cmd)

typedef CCB_DRIVER_TELL_FN(CCBDriverTellFn);
```

  When this function is called, it is passed the value of the data argument of new_CCBServerLink(), plus a command argument, as described below. If successful, tell_driver callbacks should return 0. Otherwise they should return 1, and set errno accordingly.

  The CCBDriverCmdID enumeration lists the types of commands that can be sent to the CCB device driver.

```
typedef enum {
  CCB_DRV_CONF_SCAN,   /* Install a new scan */
                       /*  configuration. */
  CCB_DRV_STAGE_SCAN,  /* Start a new scan at a */
                       /*  specified time */
  CCB_DRV_INTRA_SCAN,  /* Start an intra-scan ASAP */
  CCB_DRV_DUMP_SCAN    /* Start a dump-mode intra-scan ASAP */
} CCBDriverCmdID;
```

The CCBDriverCmd datatype contains a union of all driver commands, prefixed with a CCBDriverCmdID member identifying which member of the union is to be used.

```
struct CCBDriverCmd {
  CCBDriverCmdID type;     /* The type of command */
  union {
    CCBDrvConfScan conf;       /* type==CCB_DRV_CONF_SCAN */
    CCBDrvStageScan stage;     /* type==CCB_DRV_STAGE_SCAN */
    CCBDrvIntraScan intra;     /* type==CCB_DRV_INTRA_SCAN */
    CCBDrvDumpScan dump;       /* type==CCB_DRV_DUMP_SCAN */
  } pars;
};
```

The individual commands communicated by this structure are defined as follows.

– **Configure the next scan**

  Before starting a new scan with the stage-scan, intra-scan or dump-scan commands, CCBServerLink objects invoke this driver command to configure the hardware for the new scan. The parameters of the scan are encapsulated in the conf member of the pars union in a structure of the following type.

```
typedef struct {
  CCBPhaseSwitchCnf phase;  /* The phase-switch */
                            /*  configuration. */
  CCBCalDiodeCnf cal;       /* The calibration diode */
                            /*  configuration. */
  CCBTimingCnf timing;      /* The hardware timing */
                            /*  configuration. */
  CCBSamplerCnf sampler;    /* The sampler */
                            /*  configuration */
} CCBDrvConfScan;
```

  The members of this structure are encapsulated configuration groupings of the types that are returned by the ccb_get_*_cnf() configuration lookup functions.

– **Stage a new observing scan**

When the server library receives a start-scan command, it uses the configure-scan driver-command to pass the configuration parameters of the requested scan to the firmware, then invokes the stage-scan driver-command to initiate the scan at a specified time. The parameters of this command are encapsulated within a structure of the following type.

```
typedef struct {
  unsigned long scan; /* The numeric scan identifier */
  time_t start_time;  /* The date and time of the second */
                      /* at which to start the scan, */
                      /* expressed as a UNIX time. */
} CCBDrvStageScan;
```

The scan member forwards the numeric scan identifier that the manager sent in the start-scan command, so that the driver can use it to tag integration and monitoring data from the new scan. After receiving this command, the device driver waits until the rising edge of the 1-PPS signal that matches the specified start time and date, before starting the new scan.

– **Start an intra-scan**

When the server library receives a stop-scan command from the manager, it uses the configure-scan driver-command to pass the configuration parameters of the requested scan to the firmware, then invokes the intra-scan driver-command to immediately initiate an intra-scan. The parameters of this command are encapsulated within a structure of the following type.

```
typedef struct {
  unsigned long scan; /* The numeric scan identifier */
} CCBDrvIntraScan;
```

The scan member forwards the numeric scan identifier that the manager sent in the stop-scan command, so that the driver can use it to tag integration and monitoring data from the new intra-scan.

– **Start a dump-mode intra-scan**

When the server library receives a dump-scan command from the manager, it uses the configure-scan driver-command to pass the configuration parameters of the requested scan to the firmware, then finally invokes the dump-scan driver-command to immediately initiate a dump-mode intra-scan. The parameters of this command are encapsulated within a structure of the following type.

```
typedef struct {
  unsigned long scan;     /* The numeric scan identifier */
  unsigned short adc;     /* The ADC to dump */
  unsigned long samples;  /* The max number of samples to */
                          /*  collect per integration. */
} CCBDrvDumpScan;
```

The adc member should be an integer between 0 and CCB_NUM_ADC−1, specifying the ADC whose samples are to be collected. The samples member, should either be CCB_DUMP_MAX_SAMPLES, to specify that the maximum number of samples/integration be collected, or a positive integer specifying the desired number.

- **The callback that indicates events to watch** (select_events)

  This argument specifies a function which is called each time just before the CCB server's event loop invokes select() to wait for I/O. The driver is expected to add any file-descriptors that it wants to have watched for activity, and/or register an inactivity-timeout.

  Suitable functions to pass in the select_events argument should be declared and prototyped using the CCB_DRIVER_SELECT_EVENTS_FN() macro. Pointers to them can be recorded in variables of type CCBDriverSelectEventsFn.

```
#define CCB_DRIVER_SELECT_EVENTS_FN(fn) int (fn)( \
              CCBServerLink *sl, void *data, \
              fd_set *rfds, fd_set *wfds, int *maxfd, \
              struct timeval *timeout)

typedef CCB_DRIVER_SELECT_EVENTS_FN(CCBDriverSelectEventsFn);
```

  When this function is called, it is passed the value of the data argument of new_CCBServerLink(). The callback should use the standard FD_SET() macro to install file-descriptors of interest in *rfds and *wfds, where the former is for descriptors to be watched for readability, and the latter is for those to be watched for writability. If the value of *maxfd is less than the maximum value of any descriptor that the callback has added to rfds or wfds, then it should be set to the latter maximum. If the driver wants to be called on inactivity-timeouts, then it should register the timeout that it wants in the timeout argument.

  Normally the callback should return 0 to indicate success. If an unrecoverable error occurs, it should return 1, and set errno accordingly.

- **The callback that checks for driver events** (check_events)

  This argument specifies a function which is called each time when the CCB server's call to select() returns. The driver is expected to handle any indicated file-descriptor activity, or inactivity-timeout.

  Suitable functions to pass in the check_events argument should be declared and prototyped using the CCB_DRIVER_CHECK_EVENTS_FN() macro. Pointers to them can be recorded in variables of type CCBDriverCheckEventsFn.

```
#define CCB_DRIVER_CHECK_EVENTS_FN(fn) int (fn)( \
              CCBServerLink *sl, void *data, \
              fd_set *rfds, fd_set *wfds)
typedef CCB_DRIVER_CHECK_EVENTS_FN(CCBDriverCheckEventsFn);
```

When this function is called, it is passed the value of the data argument of new_CCBServerLink(). If the select() inactivity timeout was triggered, the rfds and wfds arguments are both NULL. Otherwise they contain the sets of file-descriptors that are now ready for reading and writing, respectively.

Normally the callback should return 0 to indicate success. If an unrecoverable error occurs, it should return 1, and set errno accordingly.

- **The callback that reboots the computer** (reboot_rtc)

  This argument specifies the function that the CCBServerLink object should call to reboot the CCB computer.

  Suitable functions to pass in the reboot_rtc argument should be declared and prototyped using the CCB_REBOOT_RTC_FN() macro. Pointers to them can be recorded in variables of type CCBRebootRTCFn.

```
#define CCB_REBOOT_RTC_FN(fn) int (fn)(CCBServerLink *sl, \
                                       void *data)

typedef CCB_REBOOT_RTC_FN(CCBRebootRTCFn);
```

  When this function is called, it is passed the value of the data argument of new_CCBServerLink(). When a reboot is successfully initiated, reboot_rtc callbacks should return 0. Otherwise they should return 1, and set errno accordingly.

- **The callback that shuts-down the computer** (shutdown_rtc)

  This argument specifies the function that the CCBServerLink object should call to shutdown both the CCB computer and the CCB hardware.

  Suitable functions to pass in the shutdown_rtc argument should be declared and prototyped using the CCB_SHUTDOWN_RTC_FN() macro. Pointers to them can be recorded in variables of type CCBShutdownRTCFn.

```
#define CCB_SHUTDOWN_RTC_FN(fn) int (fn)(CCBServerLink *sl, \
                                         void *data)

typedef CCB_SHUTDOWN_RTC_FN(CCBShutdownRTCFn);
```

  When this function is called, it is passed the value of the data argument of new_CCBServerLink(). When a shutdown is successfully initiated, shutdown_rtc callbacks should return 0. Otherwise they should return 1, and set errno accordingly.

- **The callback that sets the misc analog outputs** (set_dacs)

  This argument specifies the function that the CCBServerLink object should call to send new values to the digital-to-analog converters of the GPIO-card's miscellaneous analog outputs.

Suitable functions to pass in the set_dacs argument should be declared and prototyped using the CCB_SET_DACS_FN() macro. Pointers to them can be recorded in variables of type CCBSetDacsFn.

```
#define CCB_SET_DACS_FN(fn) int (fn)(CCBServerLink *sl, void *data, \
                         unsigned short counts[CCB_GPIO_NDAC])

typedef CCB_SET_DACS_FN(CCBSetDacsFn);
```

When this function is called, it is passed the value of the data argument of new_CCBServerLink(). Its counts[] argument is interpreted as documented in section 4.11.1, for the ccb_queue_set_dacs_cmd() function.

Callbacks of this type should return 0 if successful. Otherwise they should return 1, and set errno accordingly.

The return value of new_CCBServerDriver() is a pointer to an opaque CCBServerDriver object that can be passed to new_CCBServerLink(), or NULL if an error occurred.

Once a CCBServerDriver object has been passed to new_CCBServerLink(), it shouldn't be deleted until after del_CCBServerLink() is called to delete the corresponding server object. Objects returned by new_CCBServerDriver() are deleted by calling del_CCBServerDriver().

```
CCBServerDriver *del_CCBServerDriver(CCBServerDriver *drv);
```

The argument of this function is the object to be deleted (which can be NULL. The return value of the function is always NULL, so that one can type:

```
drv = del_CCBServerDriver(drv);
```

This both deletes the object pointed to by drv, and then sets this pointer to NULL, to ensure that subsequent illegal attempts to access the object through this pointer produce a segmentation-fault, rather than appearing to work, while actually doing something bad.

5.4 Shutting down server communications

When the CCB server shuts down, it releases the resources that were allocated by new_CCBServerLink() and closes all of its sockets, by calling del_CCBServerLink().

```
CCBServerLink *del_CCBServerLink(CCBServerLink *sl);
```

This function always returns NULL to allow the caller to type:

```
CCBServerLink *sl;
...
sl = del_CCBServerLink(sl);
```

This sets the invalidated sl pointer variable to NULL, such that if any statement subsequently tries to access the deleted object through this pointer, it is rewarded with a segmentation fault, rather than producing unpredictable behavior.

5.5 Server I/O multiplexing

To enable the CCB server to handle the telemetry and control links at the same time as interacting with the CCB device driver, the server library uses non-blocking socket I/O when reading and writing messages. The driver is expected to do the same. The select()-based event-loop is invoked by the server by calling the ccb_server_event_loop() function.

```
int ccb_server_event_loop(CCBServerLink *ccb);
```

The rfds and wfds arguments are the file descriptor sets that select() returned. Normally ccb_server_select_args() returns 0, but if an error occurs, it returns 1 and sets errno accordingly.

5.6 Queuing replies to control commands

All replies to control commands are queued internally by the library, so there are no public API functions related to this.

5.7 Queuing outgoing telemetry messages

Messages to be sent to the manager over the telemetry link are placed in message-specific queues within the corresponding CCBServerLink object, as described in section 4.13. The ccb_server_event_loop() dispatches such messages to the manager. Whenever it finishes sending a message, it chooses a new message from the highest priority queue that contains at least one message, encodes this and starts to send the result to the manager.

The functions that the CCB server uses to queue messages in the appropriate queues, are documented in the following subsections.

5.7.1 Queuing outgoing monitor-data messages

The CCB server uses the ccb_queue_monitor_msg() function to queue monitor-data messages for later transmission.

```
int ccb_queue_monitor_msg(CCBServerLink *sl,
                          const CCBTimeStamp *t,
                          unsigned long scan,
                          unsigned long number,
                          const CCBRawMonitorData *rmd);
```

Apart from the initial sl argument, the arguments of this function are as described in section 4.13, except for the rmd argument, which is a pointer to a structure of the following form.

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

In this structure, monitored analog values are represented by the integer counts that were read from the corresponding analogue-to-digital converters of the GPIO card. These can be converted to voltages using the CCB_GPIO_COUNTS_TO_VOLTS() macro (see section 4.11.1).

FPGA-specific values are recorded in the fpga[] member, wherein each CCB FPGA is represented by an element of the following type:

```
typedef struct {
  unsigned short d1_2v;     /* The voltage of the digital 1.2V PSU */
  unsigned short d2_5v;     /* The voltage of the digital 2.5V PSU */
```

```
  unsigned short d3_3v;     /* The voltage of the digital 3.3V PSU */
  unsigned short a5v;       /* The voltage of the analog 5V PSU */
  unsigned short hb;        /* The mean voltage of the heartbeat */
                            /*  signal. */
  unsigned short cnf_error; /* True if an error occurred while */
                            /* loading the FPGA firmware. */
  unsigned short cnf_done;  /* True if the FPGA firmware loading */
                            /*  process has completed. */
} CCBRawFpgaMonitorData;
```

Note that the ccb_calibrate_monitor_data() function can be used to convert all of the values in a CCBRawMonitorData structure from ADC counts, to the corresponding floating-point monitor voltages. This function has the following prototype:

```
int ccb_calibrate_monitor_data(CCBRawMonitorData *raw,
                               CCBMonitorData *cal);
```

The first argument of this function is the object containing raw ADC counts. The second argument is a pointer to where to place the corresponding values, after they have been converted to voltages. This has the CCBMonitorData data-type, which has already been described in section 4.13.

If successful, this function returns 0. Otherwise it returns 1.

To facilitate programs that use the monitoring values received in CCBRawMonitorData objects, to set the states of the LEDs on the front-panel of the CCB, the ccb_compute_led_states() function can be called to determine which LEDs should be turned on, and which should be turned off.

```
int ccb_compute_led_states(CCBRawMonitorData *raw,
                           int *light, int *snuff);
```

If this function is successful, it returns 0, and assigns bit-mask of the LEDs that are to be turned on, and turned off, respectively to the *light and *snuff arguments. Each of the LEDs is represented by a single bit, which has the value assigned to it by the CCBPanelLed enumeration.

```
typedef enum {
  CCB_SAMPLE_MODE_LED = 1,    /* CCB is in sample mode */
  CCB_SLAVE4_READY_LED = 2,   /* Slave-FPGA 4 firmware loaded */
  CCB_SLAVE3_READY_LED = 4,   /* Slave-FPGA 3 firmware loaded */
```

```
  CCB_SLAVE2_READY_LED = 8,   /* Slave-FPGA 2 firmware loaded */
  CCB_SLAVE1_READY_LED = 16,  /* Slave-FPGA 1 firmware loaded */
  CCB_MASTER_READY_LED = 32,  /* Master-FPGA firmware loaded */
  CCB_HEARTBEATS_OK_LED = 64, /* All FPGAs have healthy heartbeats. */
  CCB_POWER_OK_LED = 128,     /* All power-supllies ok */
  CCB_SLAVE4_ERROR_LED = 256, /* Slave-FPGA 4 firmware load-error */
  CCB_SLAVE3_ERROR_LED = 512, /* Slave-FPGA 3 firmware load-error */
  CCB_SLAVE2_ERROR_LED = 1024,/* Slave-FPGA 2 firmware load-error */
  CCB_SLAVE1_ERROR_LED = 2048,/* Slave-FPGA 1 firmware load-error */
  CCB_MASTER_ERROR_LED = 4096,/* Master-FPGA firmware load-error */
  CCB_CAL_MODE_LED = 8192,    /* Cal test-mode */
  CCB_DUMP_MODE_LED = 16384,  /* Dump mode */
  CCB_TEST_MODE_LED = 32768   /* Test mode */
} CCBPanelLed;
```

5.7.2 Queuing outgoing integ-data messages

The CCB server uses the ccb_queue_integ_msg() function to queue integ-data messages for later transmission.

```
int ccb_queue_integ_msg(CCBServerLink *sl,
                        const CCBIntegFrame *integ);
```

The integ argument must be a pointer to the variable that contains the header information and integrated data of the originating integration period. This variable has the following data-type.

```
typedef struct {
  CCBTimeStamp ts;             /* The start-time of the */
                               /*  integration-period. */
  unsigned long scan;          /* The unique ID of the parent scan */
  unsigned long integ;         /* The sequential ID of the */
                               /*  integration period within the */
                               /*  parent scan. */
  unsigned short flags;        /* A bitwise union of CCBIntegFlags */
                               /*  enumerators. */
  unsigned long values[CCB_MAX_INTEG]; /* The integrated data. */
} CCBIntegFrame;
```

The ordering and properties of integrated values within the values member of the CCBIntegFrame structure are discussed in section 4.13.

ccb_zero_integ_frame()

CCBIntegFrame objects can be zero-initialized by calling the function, ccb_zero_integ_frame(), which has the prototype:

```
void ccb_zero_integ_frame(CCBIntegFrame *f);
```

5.7.3 Queuing outgoing dump-frame messages

The CCB server uses the ccb_queue_dump_msg() function to queue dump-frame messages for later transmission to remote dump-mode client programs.

```
int ccb_queue_dump_frame_msg(CCBServerLink *sl,
                             const CCBDumpFrame *df);
```

The df argument must be a pointer to the variable that contains the header information and dump-mode samples of the originating integration period. This variable has the following data-type.

```
typedef struct {
  CCBTimeStamp ts;             /* The start-time of the integration period */
  unsigned long scan;          /* The unique ID of the parent scan */
  unsigned long integ;         /* The sequential ID of the integration */
                               /*  period within the parent scan. */
  unsigned short flags;        /* A bitwise union of CCBIntegFlags */
                               /*  enumerators. */
  unsigned short pswlen;       /* The number of samples per phase-switch */
                               /*  state. */
  unsigned char phase_a;       /* One bit denoting each of the 4 repeated */
                               /*  states of the switching pattern of */
                               /*  phase-switch A, where the least */
                               /*  significant bit denotes the state */
                               /*  of the switch at the start of each */
                               /*  phase-switch cycle. */
  unsigned char phase_b;       /* One bit denoting each of the 4 repeated */
                               /*  states of the switching pattern of */
                               /*  phase-switch B, where the least */
                               /*  significant bit denotes the state */
                               /*  of the switch at the start of each */
                               /*  phase-switch cycle. */
```

```
  unsigned short nsample;      /* The number of samples in samples[] */
  unsigned short samples[CCB_DUMP_MAX_SAMPLES]; /* The ADC samples */
} CCBDumpFrame;
```

The samples member should be an array of nsample samples, where nsample should be the smaller of, the number of samples that the manager originally requested, or CCB_DUMP_MAX_SAMPLES (see section 4.11.1).

The i'th element of the samples array should contain the ADC sample that started to be acquired i $\times$ 100ns from the start of the integration period. This must be a 14-bit ADC sample, plus a 15th bit that should be 1 if the ADC reported an overflow, or zero otherwise. The remaining bits should be zero. The number of bits per ADC sample, is parameterized by the CCB_ADC_SAMPLE_SIZE macro, as follows:

```
#define CCB_ADC_SAMPLE_SIZE 14
```

A bit-mask that can be used to select these bits, is parameterized by the CCB_ADC_SAMPLE_MASK macro, as follows.

```
#define CCB_ADC_SAMPLE_MASK ((1<<CCB_ADC_SAMPLE_SIZE)-1)
```

Similarly, a bit-mask that can be used to select the overflow bit, is parameterized by the CCB_ADC_OVERFLOW_MASK macro, as follows.

```
#define CCB_ADC_OVERFLOW_MASK (1<<14)
```

ccb_zero_dump_frame()

CCBDumpFrame objects can be zero-initialized by calling the function, ccb_zero_dump_frame(), which has the prototype:

```
void ccb_zero_dump_frame(CCBDumpFrame *f);
```

ccb_phase_switch_sequence()

The appropriate values to assign to the phase_a and phase_b fields of a CCBDumpFrame variable, can be computed by calling the ccb_phase_switch_sequence() function. This has the following prototype:

```
int ccb_phase_switch_sequence(const CCBConfig *cnf,
                              unsigned char *phase_a,
                              unsigned char *phase_b);
```

The first argument must be the configuration of the originating scan of the dump-mode data, and pointers to the phase_a and phase_b fields of a CCBDumpFrame object, should be passed directly to this function, via the correspondingly named function-arguments.

5.7.4 Queuing outgoing log-message messages

The CCB server uses the ccb_log_server_msg() function to queue formatted log messages for later transmission.

```
int ccb_log_server_msg(CCBServerLink *sl, CCBLogLevel level,
                       unsigned long id, const char *fmt, ...);
```

The level argument indicates the significance of the message, as described in section 4.13.

The id argument should be specified as CCB_LOGID(?). As documented in section 2.9, this tells scripts that the makefile runs before compilation, to replace the ? character with a unique log ID.

The fmt argument is a standard printf-style format string, and is followed by the arguments that its format-specifiers refer to. Note that if gcc's -Wformat warning option is used when compiling code that calls this function, both the contents of the format string and the types of the corresponding arguments are checked by the compiler.

Where necessary, the formatted log message is silently truncated to fit within the CCB_MAX_LOG byte maximum that is imposed by the CCBLogMsg message structure described in section 6.2.3.

Normally ccb_log_server_msg() returns 0, but if a serious error occurs, non-zero is returned, and errno is set accordingly. Truncation is not considered to constitute a serious error.

Chapter 6

Library internals

The client and server communications libraries are comprised of three logical layers. Going from the highest level to the lowest level layer, the layers are as follows.

- The CCB interface layer. This is the only part of the library that is specific to the CCB. In addition to providing the public-interface functions described above, it defines all of the CCB message types and aggregates the resources of the control and telemetry connections.

- The message translation layer. This layer interprets the message definitions specified by the CCB interface layer.

- The packet buffer layer. For output messages, this layer converts host-specific datatypes to corresponding portable byte streams, and aggregates the results within the internal packet buffer of the output stream, starting with a byte count, ready for transmission. For input messages this layer, which is passed a completely read message within the internal packet buffer of the input stream, decodes the contents of the message, and passes the result to the message translation layer using native datatypes.

- The I/O layer. This layer handles non-blocking reading and writing of the raw byte streams, of which each message is composed, using the initial 4-byte integer of each message to determine how much to read and write.

This is illustrated in the communication stack shown in figure 6.1.

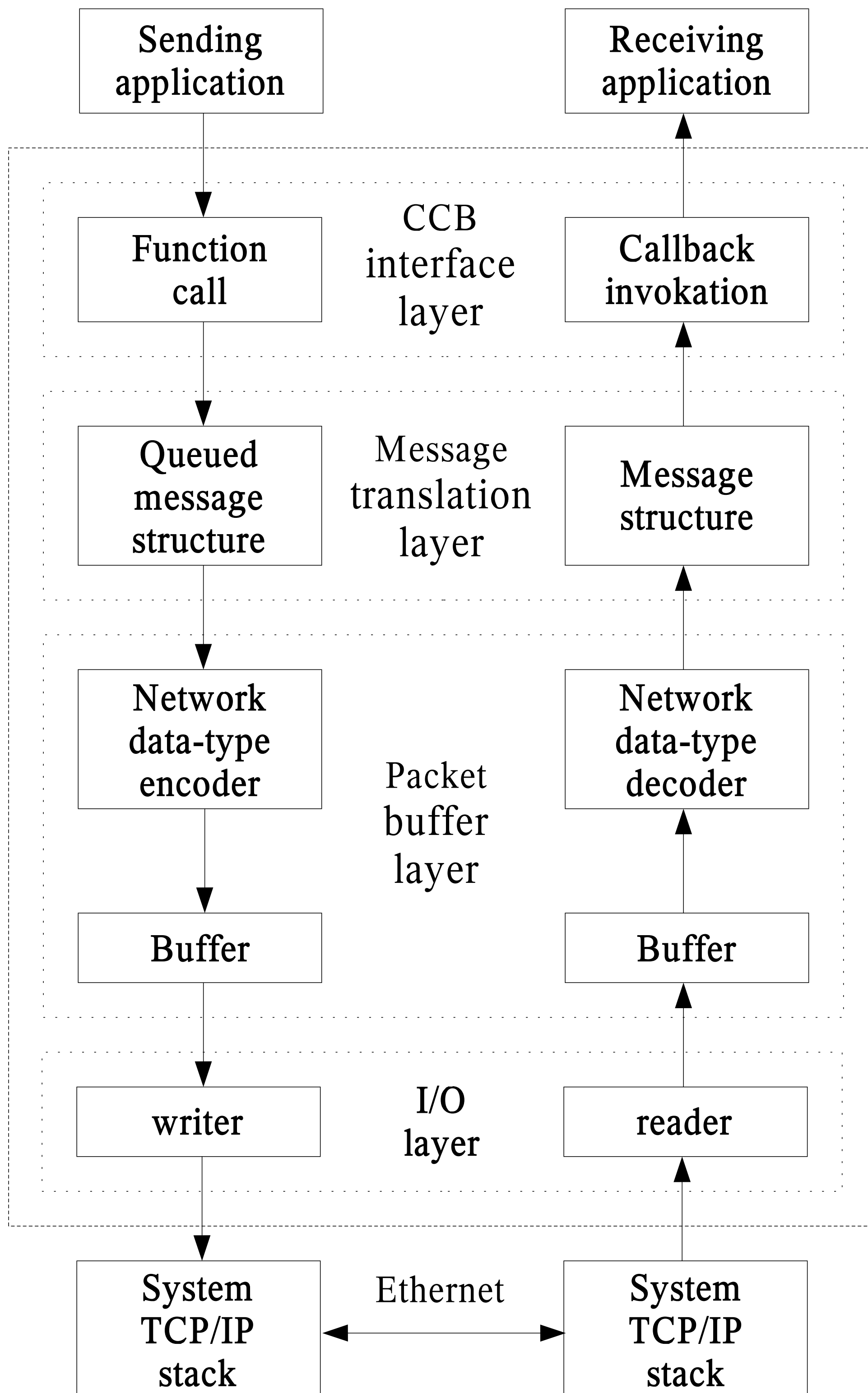


Figure 6.1: The CCB communications stack

6.1 The message translation layer

6.1.1 Message structure specification

In order to convert the contents of the previously described message structures to and from portable network byte streams, the message-translation layer of the library needs to know exactly what these structures contain, and how to access each of their members. This section explains how this information is provided.

6.1.2 Supported data-types within message structures

Since the communications library can only encode and decode data-types that it knows about, all message structures are required to have members that are declared using the types described in the following table.

Enumerator	Host data-type	Network data-type
CCB_NET_ASCII	char	8-bit unsigned char
CCB_NET_BYTE	signed char	8-bit signed integer
CCB_NET_UBYTE	unsigned char	8-bit unsigned integer
CCB_NET_SHORT	short	16-bit signed integer
CCB_NET_USHORT	unsigned short	16-bit unsigned integer
CCB_NET_LONG	long	32-bit signed integer
CCB_NET_ULONG	unsigned long	32-bit unsigned integer
CCB_NET_FLOAT	float	32-bit floating point
CCB_NET_DOUBLE	double	64-bit floating point

Note that all integer types are transfered over the network in big-endian, 2's-complement format, and that the two floating point data-types are transfered in big-endian IEEE-754 format.

Also note that the CCB_NET_ASCII enumerators tells the message translation layer that the associated arrays of characters should be interpreted as `'\0'` terminated C strings. These are actually transfered over the network as variable length arrays of bytes, preceded by length counts.

6.1.3 CCBNetMsg - The base-class of all messages

The communications library requires that the first member of all message structures be a CCBNetMsg member.

```
typedef struct {
  long type;  /* The type of the parent message-structure */
} CCBNetMsg;
```

This allows message structures to be passed to the message translation layer of the library using pointers to their initial CCBNetMsg structure. As will be described shortly, the value of the type member of this structure refers the library to a description of the actual message structure that has been passed.

6.1.4 Some example message structures

To see how the contents of message structures are described to the translation-layer of the communications library, consider the following two example message structures, called CCBExampleMsg1, and CCBExampleMsg2:

```
#define SDIM 20;        /* The size of the example string member */
                        /*  in the following message structure. */

typedef struct {        /* Example message structure 1 */
  CCNNetMsg base;       /* The message identification header */
  char string[SDIM];    /* A string to be transmitted */
  unsigned short slen; /* strlen(string) */
} CCBExampleMsg1;

typedef struct {        /* Example message structure 2 */
  CCNNetMsg base;       /* The message identification header */
  unsigned long foo;    /* A <= 32-bit unsigned number */
} CCBExampleMsg2;
```

6.1.5 CCBNetMsgMember – Message field descriptions

With the exception of the obligatory initial CCBNetMsg member, each member of each message structure is described to the library using a CCBNetMsgMember structure.

```
typedef struct {
  const char *name;     /* The textual name of the member */
  size_t offset;        /* The byte-offset of the member in the */
                        /*  local message structure */
  CCBNetDataType type; /* The enumerated data-type of the member */
  int ntype;            /* The number of elements in the member */
} CCBNetMsgMember;
```

The following example code shows how arrays of these CCBNetMsgMember structures are used to describe the elements of the two example message structures.

```
#include <stddef.h>
#include "ccbnetobj.h"

/* The description of the members of CCBExampleMsg1 */

static const CCBNetMsgMember ccb_example_msg1_members[] = {
  {"string", offsetof(CCBExampleMsg1, string),      CCB_NET_ASCII, SDIM},
  {"slen",   offsetof(CCBExampleMsg1, slen),        CCB_NET_USHORT,  1},
};

/* The description of the members of CCBExampleMsg2 */

static const CCBNetMsgMember ccb_example_msg2_members[] = {
  {"foo",    offsetof(CCBExampleMsg2, foo),         CCB_NET_ULONG,  1},
};
```

6.1.6 CCBNetMsgInfo – Individual message descriptions

In addition to descriptions of the contents of each message type, the communications library needs to know both the host-dependent size of the message data-structures, and a convenient way for the various parts of the library to refer each other to a given type of message. Each message is thus further described using a CCBNetMsgInfo structures.

```
typedef struct {
  int type;                       /* The message-type enumerator */
  const char *name;               /* The name of this message-type */
  const CCBNetMsgMember *member; /* Descriptions of each member */
  int nmember;                    /* The number of elements in member[] */
  size_t native_size;             /* The host-dependent size of the */
                                  /* the corresponding message structure */
} CCBNetMsgInfo;
```

The name field, which isn't currently used by the library, may in future be used when printing out the contents of messages for debugging purposes.

For a given network connection, the communications library needs separate descriptions of the messages that it is expected to transmit, and those that it is expected to receive. To do this the CCB interface layer registers two arrays of CCBNetMsgInfo structures per

connection, one describing outgoing messages, while the other describes incoming messages. The indexes of elements in these arrays are the means by which the various parts of the library, at both ends of the communications link, refer each other to a given message type. Since the index associated with a given message type will change if somebody inserts a new message type in the middle of a message-description array, the CCB interface layer assigns a copy of the enumerator that it uses to refer to each message type, to the type field of the corresponding CCBNetMsgInfo message-definition element. This allows the message-translation layer to verify that these enumerators match the array indexes of the messages to which they refer. Thereafter, whenever the CCB interface layer passes a message structure to the message-translation layer for transmission over the network, it sets the type member of the CCBNetMsg structure accordingly, to tell the message-translation layer what type of message structure it is being passed. Similarly, when the message-translation layer receives a message from the network, it records the type of message that it received, in the type member of the CCBNetMsg structure that it returns.

Returning to the example, the types of the example messages would be enumerated, and described in a message-definition array, as follows:

```
typedef enum {
  CCB_EXAMPLE_MSG1,  /* The index of the first example message */
  CCB_EXAMPLE_MSG2   /* The index of the second example message */
} CCBExampleMsgTypes;

static const CCBNetMsgInfo ccb_example_messages[] = {
  {CCB_EXAMPLE_MSG1, "example1", ccb_example_msg1_members,
    NET_ARRAY_DIM(ccb_example_msg1_members), sizeof(CCBExampleMsg1)},
  {CCB_EXAMPLE_MSG2, "example2", ccb_example_msg2_members,
    NET_ARRAY_DIM(ccb_example_msg2_members), sizeof(CCBExampleMsg2)},
};
```

In this example CCBExampleMsgTypes associates symbolic names with the indexes of the correspondingly messages in the ccb_example_messages[] array, while the latter array provides the description of all messages for one direction of a communications link.

6.2 The CCB interface layer

For each of the message queuing and received-message callback functions in the public API, the CCB interface layer defines a message structure for passing the corresponding message to and from the message-translation layer of the library. The following sub-sections briefly describe these structures. Note that since these structures are hidden within the communications library, provided that the library is compiled as a shared library, the contents of the

message structures can be rearranged without requiring a recompilation of the manager or the CCB server.

6.2.1 The message structures of outgoing control messages

As previously mentioned, the message-translation layer requires that all messages being transmitted over a particular network connection be internally enumerated by the CCB-interface layer. This enumeration is used to communicate message types both between the CCB-interface and message-translation layers of the library, and between the separate message-translation layers at the two ends of the communications link. The CCBControlCommandType enumeration serves this role for outgoing messages on the control link.

```
typedef enum {
  CCB_PHASE_SWITCH_CMD,   /* A phase-switch config command */
  CCB_CAL_DIODE_CMD,      /* A cal-diode config command */
  CCB_TIMING_CMD,         /* An timing config command */
  CCB_SAMPLER_CMD,        /* An sampler config command */
  CCB_START_SCAN_CMD,     /* A start-scan command */
  CCB_STOP_SCAN_CMD,      /* A stop-scan command */
  CCB_DUMP_SCAN_CMD,      /* A dump-scan command */
  CCB_MONITOR_CMD,        /* A monitoring control command */
  CCB_TELEMETRY_CMD,      /* A telemetry stream control command */
  CCB_LOGGER_CMD          /* A log control command */
  CCB_RESET_CMD,          /* A reset command */
  CCB_PING_CMD,           /* a ping command */
  CCB_STATUS_REQUEST_CMD, /* a status-request command */
  CCB_SHUTDOWN_CMD,       /* A computer shutdown command */
  CCB_REBOOT_CMD,         /* A computer reboot command */
  CCB_LOAD_DRIVER_CMD,    /* A device-driver loading command */
  CCB_SET_DACS_CMD        /* Set the count inputs of external */
                          /*  digital-to-analog converters, to generate */
                          /*  diagnostic calibration voltages. */
} CCBControlCommandType;
```

As documented on page 69, all control command messages include a manager-provided integer identifier, which is used by the CCB server to associate acknowledgment replies with the messages that they refer to. Beware that this is unrelated to the internal enumerated command-type IDs used by the library. All outgoing control messages thus have two members in common, the mandatory CCBNetMsg initial member of all CCB network messages, which contains the internal message-type identifier of the library, and a manager-provided message identifier. To allow generic access to these two common members by the CCB interface layer, regardless of control-message type, they are aggregated into a CCBControlCommandHeader structure, which is the first member of all outgoing control-message structures.

```
typedef struct {
  CCBNetMsg base;    /* The initial member of all messages */
  long id;           /* The manager's identifier of the */
                     /*  parent message. */
} CCBControlCommandHeader;
```

Since all message structures start with a CCBControlCommandHeader member, whose first member is a CCBNetMsg object, a pointer to the head.base member of the following union of all outgoing control-message structures can portably be used to exchange any of these messages between the CCB-interface layer and message-translation layer of the communications library. The actual type of message passed in this way can be determined from the type member of the CCBNetMsg object.

```
typedef union {
  CCBControlCommandHeader head;/* The common control message header */
  CCBPhaseSwitchCmd phase_cmd; /* head.base.type=CCB_PHASE_SWITCH_CMD */
  CCBCalDiodeCmd diode_cmd;    /* head.base.type=CCB_CAL_DIODE_CMD */
  CCBTimingCmd timing_cmd;     /* head.base.type=CCB_TIMING_CMD */
  CCBSamplerCmd sampler_cmd;   /* head.base.type=CCB_SAMPLER_CMD */
  CCBStartScanCmd start_scan;  /* head.base.type=CCB_START_SCAN_CMD */
  CCBStopScanCmd stop_scan;    /* head.base.type=CCB_STOP_SCAN_CMD */
  CCBDumpScanCmd dump_scan;    /* head.base.type=CCB_DUMP_SCAN_CMD */
  CCBMonitorCmd monitor;       /* head.base.type=CCB_MONITOR_CMD */
  CCBTelemetryCmd telemetry;   /* head.base.type=CCB_TELEMETRY_CMD */
  CCBLoggerCmd logger;         /* head.base.type=CCB_LOGGER_CMD */
  CCBResetCmd reset;           /* head.base.type=CCB_RESET_CMD */
  CCBPingCmd ping;             /* head.base.type=CCB_PING_CMD */
  CCBStatusRequestCmd status;  /* head.base.type=CCB_STATUS_REQUEST_CMD */
  CCBShutdownCmd shutdown;     /* head.base.type=CCB_SHUTDOWN_CMD */
  CCBRebootCmd reboot;         /* head.base.type=CCB_REBOOT_CMD */
  CCBLoadDriverCmd driver;     /* head.base.type=CCB_LOAD_DRIVER_CMD */
  CCBSetDacsCmd set_dacs;      /* head.base.type=CCB_SET_DACS_CMD */
} CCBControlCommand;
```

CCBPhaseSwitchCmd – The phase-switching configuration command

The ccb_queue_start_start_cmd() and ccb_queue_stop_scan_cmd() functions both queue message structures of the following type for transmission when the phase-switch parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_PHASE_SWITCH_CMD */
```

```
  CCBPhaseSwitchCnf cnf;        /* The phase-switch configuration */
                                /*  parameters. */
} CCBPhaseSwitchCmd;
```

CCBCalDiodeCmd – The calibration diode configuration command

The ccb_queue_start_scan_cmd() and ccb_queue_stop_scan_cmd() functions both queue message structures of the following type for transmission when the cal-diode parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_CAL_DIODE_CMD*/
  CCBCalDiodeCnf cnf;           /* The cal-diode configuration */
                                /*  parameters. */
} CCBCalDiodeCmd;
```

CCBTimingCmd – The acquisition-timing configuration command

The ccb_queue_start_start_cmd() and ccb_queue_stop_scan_cmd() functions both queue message structures of the following type for transmission when the hardware-timing parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_TIMING_CMD */
  CCBTimingCnf cnf;             /* The timing configuration */
                                /*  parameters. */
} CCBTimingCmd;
```

CCBSamplerCmd – The sampler configuration command

The ccb_queue_start_start_cmd() and ccb_queue_stop_scan_cmd() functions both queue message structures of the following type for transmission when the hardware-sampler parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_SAMPLER_CMD */
  CCBSamplerCnf cnf;            /* The sampler configuration */
                                /*  parameters. */
} CCBSamplerCmd;
```

CCBStartScanCmd – The start-scan command

The ccb_queue_start_scan_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_START_SCAN_CMD */
  unsigned long scan;           /* The numeric ID to give the new */
                                /*  scan. */
  unsigned long mjd;            /* The MJD UTC day number */
  unsigned long tod;            /* The time of day (seconds since */
                                /*  0H UTC). */
} CCBStartScanCmd;
```

CCBStopScanCmd – The stop-scan command

The ccb_queue_stop_scan_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_STOP_SCAN_CMD */
  unsigned long scan;           /* The numeric ID to give the new */
                                /*   intra-scan. */
} CCBStopScanCmd;
```

CCBDumpScanCmd – The dump-scan command

The ccb_queue_dump_scan_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_DUMP_SCAN_CMD */
  unsigned long scan;           /* The numeric ID to give the new */
                                /*   intra-scan. */
  unsigned short adc;           /* The ADC to dump */
  unsigned long samples;        /* The max number of samples to */
                                /*  collect per integration. */
  unsigned long frames;         /* The number of per-integration */
                                /*  data-frames to deliver. */
} CCBDumpScanCmd;
```

CCBMonitorCmd – The monitor command

The ccb_queue_monitor_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_MONITOR_CMD */
  unsigned short period;        /* The interval between monitoring */
                                /*  updates. */
} CCBMonitorCmd;
```

CCBTelemetryCmd – The telemetry command

The ccb_queue_telemetry_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_TELEMETRY_CMD */
  unsigned short streams;       /* The telemetry-streams to report */
} CCBTelemetryCmd;
```

CCBLoggerCmd – The logger command

The ccb_queue_logger_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_LOGGER_CMD */
  unsigned long period;         /* The interval at which */
                                /*  historical log messages are */
                                /*  forgotten (seconds). */
} CCBLoggerCmd;
```

CCBResetCmd – The reset command

The ccb_queue_reset_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_RESET_CMD */
} CCBResetCmd;
```

CCBPingCmd – The ping command

The ccb_queue_ping_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_PING_CMD */
} CCBPingCmd;
```

CCBStatusRequestCmd – The status-request command

The ccb_queue_status_request_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type = */
                                /*  CCB_STATUS_REQUEST_CMD */
} CCBStatusRequestCmd;
```

CCBShutdownCmd – The shutdown command

The ccb_queue_shutdown_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_SHUTDOWN_CMD */
} CCBShutdownCmd;
```

CCBRebootCmd – The reboot command

The ccb_queue_reboot_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_REBOOT_CMD */
} CCBRebootCmd;
```

CCBLoadDriverCmd – The load-driver command

The ccb_queue_load_driver_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_LOAD_DRIVER_CMD */
  unsigned short type;          /* A CCBDriverType enumerator */
} CCBLoadDriverCmd;
```

CCBSetDacsCmd – The set-dacs command

The ccb_queue_set_dacs_cmd() function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
  CCBControlCommandHeader head; /* head.base.type=CCB_SET_DACS_CMD */
  unsigned short counts[CCB_GPIO_NDAC]; /* The count inputs of the */
                                        /*  digital-to-analog */
                                        /*  converters. */
} CCBSetDacsCmd;
```

6.2.2 The message structures of incoming control-link replies

For incoming messages sent by the CCB server to the manager over the control-link, the CCB-interface layer enumerates the known types of reply message as follows.

```
typedef enum {
  CCB_CNTRL_PING_REPLY,     /* A ping reply */
  CCB_STATUS_REPLY,         /* A status-request reply */
  CCB_CNTRL_CMD_ACK         /* A control command-acknowledgement reply */
} CCBControlReplyType;
```

The first member of all control-reply message structures is a CCBControlReplyHeader structure.

```
typedef struct {
  CCBNetMsg base;         /* The base-class of all messages */
} CCBControlReplyHeader;
```

Since all message structures start with a CCBControlReplyHeader member, a pointer to the head member of the following union of all incoming control-link message structures can portably be used to exchange any control-link ping reply message between the internal layers of the library, with the head.base.type member of the union being used to determine what type of message is actually being passed.

```
typedef union {
  CCBControlReplyHeader head;   /* The common header of all control */
                                /*  link replies. */
  CCBCntrlPingReply ping;       /* base.type = CCB_CNTRL_PING_REPLY */
  CCBStatusReply status,        /* base.type = CCB_STATUS_REPLY */
  CCBCntrlCmdAck cmd_ack;       /* base.type = CCB_CNTRL_CMD_ACK */
} CCBControlReply;
```

CCBCntrlPingReply – A reply to a ping command

Replies to ping commands over the control link are exchanged with the message translation layer in structures of the following type.

```
typedef struct {
  CCBControlReplyHeader head;   /* head.base.type=CCB_CNTRL_PING_REPLY */
} CCBCntrlPingReply;
```

CCBStatusReply – A reply to a status-request command

Callback functions registered with ccb_status_reply_callback() are invoked whenever a message structure of the following type is received by the communications library.

```
typedef struct {
  CCBControlReplyHeader head;   /* head.base.type = */
                                /*  CCB_STATUS_REPLY */
  unsigned long status;         /* The status of the CCB */
} CCBStatusReply;
```

CCBCntrlCmdAck – An acknowledgment to a control command

Whenever a message structure of the following type is received by the communications library, if the status member is anything other than CCB_CMD_ACCEPTED, then the library invokes the callback that the manager previously provided when it called ccb_cmd_error_callback().

```
typedef struct {
  CCBControlReplyHeader head;   /* head.base.type=CCB_CNTRL_CMD_ACK */
  unsigned long id;             /* The manager-specified ID of */
                                /*  the command that is being */
                                /*  acknowledged. */
  unsigned long status;         /* A CCBCmdStatus enumerator */
} CCBCntrlCmdAck;
```

6.2.3 The message structures of incoming telemetry messages

This section documents the data structures that are exchanged between the CCB interface layer and the message translation layer at both ends of the telemetry link. The CCB interface layer defines the following enumeration to distinguish between the various message types encoded in these message structures.

```
typedef enum {
  CCB_INTEG_MSG,       /* An integration data message */
  CCB_MONITOR_MSG,     /* A monitoring data message */
  CCB_LOG_MSG,         /* A log message */
  CCB_TELEM_PING_REPLY /* A reply to a ping command */
} CCBTelemetryType;
```

The first member of all telemetry message structures is a CCBTelemetryHeader structure, which is defined as follows.

```
typedef struct {
  CCBNetMsg base;         /* The base-class of all messages */
  unsigned long mjd;      /* The MJD UTC day number */
  unsigned long sec;      /* The time of day (seconds since OH UTC) */
  unsigned long ns;       /* The number of nanoseconds from */
                          /* the start of the specified second. */
} CCBTelemetryHeader;
```

Note that the obligatory CCBNetMsg member of all network messages is the first member of this structure. The remaining members report the date and time at which the message was generated.

Since all telemetry message structures start with a CCBTelemetryHeader member, a pointer to the head member of the following union can be used as a pointer to any type of telemetry message. The base.type member of this header can then be used to determine which type of telemetry message the pointer actually refers to.

```
typedef union {
  CCBTelemetryHeader head;     /* The common telemetry header */
  CCBIntegMsg integ;           /* An integration data message */
  CCBMonitorMsg monitor;       /* A monitor data message */
  CCBLogMsg log;               /* A log message */
  CCBTelemPingReply ping;      /* A reply to a ping command */
} CCBTelemetryMessage;
```

The data-structures within this union, are declared as follows.

CCBIntegMsg – Integration data messages

Callback functions registered with ccb_integ_msg_callback() are invoked whenever a message structure of the following type is received by the communications library over the telemetry link.

```
typedef struct {
  CCBTelemetryHeader head;           /* head.base.type=CCB_INTEG_MSG */
  unsigned long scan;                /* The number of the parent scan */
  unsigned long id;                  /* The integration ID */
  unsigned short flags;              /* Single-bit status flags */
  unsigned long data[CCB_MAX_INTEG]; /* The integrated data */
} CCBIntegMsg;
```

CCBMonitorMsg – Monitor data messages

Callback functions registered with ccb_monitor_msg_callback() are invoked whenever a message structure of the following type is received by the communications library over the telemetry link.

```
typedef struct {
```

```
  CCBTelemetryHeader head;   /* head.base.type=CCB_MONITOR_MSG */
  unsigned long scan;        /* The scan number */
  unsigned long id;          /* The monitor ID */
  unsigned short fan12v;     /* The voltage of the 12V fan PSU */
  unsigned short a8v;        /* The voltage of the analog 8V PSU */
  unsigned short d5v;        /* The voltage of the digital 5V PSU */
  unsigned short cnf_done;   /* True if the FPGAs have been programmed */
  unsigned short high_temp;  /* True if the CCB enclosure is too hot */
  unsigned short ccb_id;     /* The 2-bit ID of the CCB hardware */
/*
 * The following members are specific to each FPGA, and are thus arrays
 * of one value per FPGA.
 */
  unsigned short fpga_d1_2v[CCB_NUM_FPGA];    /* The digital 1.2V PSU */
                                              /*  voltage. */
  unsigned short fpga_d2_5v[CCB_NUM_FPGA];    /* The digital 2.5V PSU */
                                              /*  voltage. */
  unsigned short fpga_d3_3v[CCB_NUM_FPGA];    /* The digital 3.3V PSU */
                                              /*  voltage. */
  unsigned short fpga_a5v[CCB_NUM_FPGA];      /* The analog 5V PSU */
                                              /* voltage. */
  unsigned short fpga_hb[CCB_NUM_FPGA];       /* The mean heartbeat */
                                              /* voltage. */
  unsigned short fpga_cnf_error[CCB_NUM_FPGA];/* Firmware checksum */
                                              /*  error. */
  unsigned short fpga_cnf_done[CCB_NUM_FPGA]; /* FPGA programmed */
} CCBMonitorMsg;
```

CCBLogMsg – CCB log messages

Callback functions registered with ccb_log_msg_callback() are invoked whenever a message structure of the following type is received by the communications library over the telemetry link.

```
#define CCB_MAX_LOG 128      /* The maximum length of a log */
                             /* message. */
typedef struct {
  CCBTelemetryHeader head;   /* head.base.type=CCB_LOG_MSG */
  char msg[CCB_MAX_LOG];     /* The message to be logged */
  unsigned long id;          /* A unique message identifier */
  unsigned short level;      /* The severity level of the message */
} CCBLogMsg;
```

CCBTelemPingReply – A reply to a ping command

Replies to ping commands over the telemetry link are exchanged with the message translation layer in structures of the following type.

```
typedef struct {
  CCBTelemetryHeader head;  /* head.base.type=CCB_TELEM_PING_REPLY */
} CCBTelemPingReply;
```

6.3 Sending network messages

As mentioned earlier, output control messages are queued for transmission in a queue of message structures, then dispatched to the server by one or more calls to ccb_client_communicate(). While ccb_client_communicate() is running, if the I/O layer finishes transmitting a message, the message-translation layer does the following.

1. It removes the message structure of the next oldest message from the queue.
2. It then calls a function in the packet-buffer layer which:
   (a) Clears the output buffer and resets its read and write pointers to point to the start of the buffer.
   (b) Writes a zero-valued big-endian byte-count in the first 4 bytes of the buffer.
   (c) Writes the enumerated type of the message, as passed to it by the message-translation layer, expressed as an unsigned 2-byte big-endian integer.
   (d) Increments the buffer write-pointer to point to the byte following the above two items.
3. For each member within the message structure, the message-translation layer then calls a function in the API of the packet-buffer layer, chosen according to the type of the structure member, to have the value of the member appended to the current message within the buffer. These functions all increment the buffer write-pointer to point to the byte in the buffer which follows the data that they appended.
4. Once all structure members have been packed into the buffer, the message-translation layer then calls a function of the packet-buffer API to terminate the message in the buffer. This function replaces the zero-valued byte-count at the start of the buffer with the count of the actual number of bytes used by the message in the buffer.
5. Finally, the message-translation layer calls a function in the I/O layer to start writing the contents of the buffer to the control socket. As the I/O layer does this, it increments

the read-pointer of the packet-buffer, so that it knows from where to resume if the socket blocks when non-blocking I/O is in use. If it completes writing the latest message, it goes back to step one, to get the next unsent message. Otherwise, it returns control to the manager, and tells the manager to call ccb_client_communicate() again when output again becomes possible, so that it can resume sending the current message.

6.4 Receiving network messages

As already documented, messages are read from the telemetry port of the server by calling ccb_client_communicate(). At the start of reading each new message, this function does the following:

1. It tells the packet-buffer layer of the telemetry connection to clear its input buffer. This also resets the read and write pointers of the buffer to point to its first byte.
2. It instructs the I/O layer to attempt to read the initial 4 byte, byte count into the message buffer.
3. In practice, if non-blocking I/O is in effect, a few calls may be needed to ccb_client_communicate() before the byte count is completely read.
4. Once the I/O layer has the byte count, it knows how many more bytes it will need to read to acquire the new message.
5. The I/O layer then attempts to read the rest of the message. Again, this may require multiple calls to ccb_client_communicate() when non-blocking I/O is being used.
6. Once the message has been completely read into the input packet-buffer, the message translation layer then decodes the message-type enumeration that follows the byte count, and uses this to identify the type of the message within its table of message definitions.
7. According to the member descriptions in the definition of the message, the message-translation layer now calls the appropriate datatype-specific functions in the packet-buffer layer to decode the values of each member of the message, and records the results in an internal message structure.
8. The completed message structure is then passed up to ccb_client_communicate(), which invokes the corresponding callback function to deliver the contents of the message to the manager.

The equivalent procedure is of course performed for the replies received over the control link, and this uses all of the same functions, except that different callback functions are called to deliver messages to the manager.

Chapter 7

The CCB dump-data communications API

Programs which read dump-mode data from the CCB server while the CCB server is under the control of the CCB manager or the ccb_demo_client program, can be written by using functions that are provided by the libccbdump library.

This provides a very simple interface to the dump-mode TCP/IP port of the CCB server. The following is a simple example of a program that uses this interface.

```
#include <stdlib.h>
#include <stdio.h>

#include "ccbdump.h"

int main(int argc, char *argv[])
{
  CCBDumpReader *dr;   /* The object which manages the network connection */
  CCBDumpFrame f;      /* A dump-mode frame of data */
  const char *host;    /* The CCB host-computer to contact */
  int i;
/*
 * Get the name of the computer on which the CCB server is running,
 * from the command line.
 */
  if(argc == 2) {
    host = argv[1];
  } else {
    fprintf(stderr, "Usage: %s <hostname>\n", argv[0], host);
    return 1;
```

```
  };
/*
 * Connect to the CCB server that is running on the specified
 * computer. This function call returns an opaque object, which
 * manages the open connection to the CCB server's dump-mode port.
 */
  dr = new_CCBDumpReader(host);
  if(!dr)
    return 1;
/*
 * Read one dump-mode frame of data at a time, and display its
 * contents to standard output.
 */
  while(ccb_read_dump_frame(dr, &f)==0) {
/*
 * Display items from the header.
 */
    printf("Scan=%lu Integ=%lu flags=%hu pswlen=%hu nsample=%hu\n",
           f.scan, f.integ, f.flags, f.pswlen, f.nsample);
/*
 * Display the individual dump-mode samples of the frame.
 */
    for(i=0; i<f.nsample; i++) {
      unsigned long sample = f.samples[i];
/*
 * Print the sample number, within the frame.
 */
      printf("%2d: ", i);
/*
 * Is the overflow bit set?
 */
      if((sample & CCB_ADC_OVERFLOW_MASK) != 0)
        printf("(overflowed)\n");
      else
        printf("%05ld\n", sample);
    };
  };
/*
 * Disconnect from the dump-mode server.
 */
  dr = del_CCBDumpReader(dr);
  return 0;
}
```

Assuming that the CCB include files are installed in /usr/local/include/ and the CCB libraries are installed in /usr/local/lib, and that the above code is placed in a file called test_demo.c this program could be compiled and linked by typing:

```
gcc -o test_demo test_demo.c -I/usr/local/include -L/usr/local/lib \
    -Xlinker -R/usr/local/lib -lccbdump
```

The program would then be run like:

```
./test_demo ccblab.gb.nrao.edu
```

7.1 Functions in the libccbdump library

The functions in the library are defined as follows.

7.1.1 new_CCBDumpReader()

This function opens a connection to the dump-mode port of the CCB server process, on a specified computer, and returns an opaque object that should be used for subsequent communication with that server.

It's function-prototype is as follows:

```
CCBDumpReader *new_CCBDumpReader(const char *host);
```

It's sole argument is a string containing the name or IP-address of the computer where the CCB server is running. If it succeeds in opening a connection to the server, then it returns a non-NULL pointer, which should subsequently be passed to other functions in the library, to communicate with the server.

On error, this function returns NULL, after writing an error message to the standard-error output-stream.

7.1.2 get_CCBDumpReader_fd()

Once a connection to a CCB server has been opened by a call to new_CCBDumpReader() the get_CCBDumpReader_fd() function can optionally be used to find out which file descriptor

dump-mode data will be arriving on. This is only intended to be used by programs that use the select() system call to watch for the arrival of data. It should not be used by the program to read or write data, other than through the ccb_read_dump_frame() function.

It's function-prototype is as follows:

```
int ccb_get_CCBDumpReader_fd(CCBDumpReader *dr);
```

The sole argument of this function is the pointer that was returned by the preceding call to new_CCBDumpReader(). If the argument is NULL, then the returned value will be -1. Otherwise it will be the file descriptor of the TCP/IP connection to the server.

7.1.3 ccb_read_dump_frame()

Once a connection to a CCB server has been opened by a call to new_CCBDumpReader() the ccb_read_dump_frame() function can be called to read data from the server. Each call to this function returns one frame of data from a single integration period.

It's function-prototype is as follows:

```
int ccb_read_dump_frame(CCBDumpReader *dr, CCBDumpFrame *f);
```

Its first argument is the pointer that was returned by the preceding call to new_CCBDumpReader(). Its second argument is a pointer to the variable in which to store the received frame of data. Its return value is 0 if new data were read, or 1 if an unrecoverable error occurred, such as the loss of the connection to the server.

The contents of CCBDumpFrame variables have already been described in section 5.7.3.

Macros are provided for deducing the states of the phase-switches in each sample. For example, if the CCBDumpFrame variable, whose pointer was passed to ccb_read_dump_frame(), were called df, and one wanted to know what the states of phase switches A and B were, when element i of df.samples[] was acquired, then one would type:

```
int state_of_switch_a, state_of_switch_b;
...
state_of_switch_a = CCB_DUMP_PHASE_A(df, i);
state_of_switch_b = CCB_DUMP_PHASE_B(df, i);
```

When these macros return 0, this means that the phase-switch was open. When they return 1, this means that the phase-switch was closed.

7.1.4 del_CCBDumpReader()

This function closes a previously opened connection to the dump-mode port of the CCB server process, relinquishes the resources that were used to communicate with it, and returns NULL.

It's function-prototype is as follows:

```
CCBDumpReader *del_CCBDumpReader(CCBDumpReader *dr);
```

It's sole argument is the pointer that was returned by a previous call to new_CCBDumpReader(). Note that it is legal for this to be NULL.

This function always returns NULL and is designed to be used in the following manner.

```
dr = del_CCBDumpReader(dr);
```

where dr is a CCBDumpReader pointer that was previously returned by new_CCBDumpReader(). By assigning the NULL return value of the function, to the pointer whose object has just been deleted, one will see a segmentation fault if the program subsequently incorrectly tries to continue to use the deleted object.

Chapter 8

The CCB diagnostic communications API

In order to facilitate the writing of new diagnostic programs, a library, libccbtestclient, has been provided that can be used to configure and start scans, and read back either integrated or dump-mode data. It constitutes a simplifying layer on top of the libccbcommon, libccbclient and libccbdump libraries.

Unlike the lower level libraries, which report the arrival of data from the CCB server, by calling asynchronous callback functions, the libccbtestclient library hides the asynchronous, non-blocking aspects of the lower-level libraries behind a simpler, synchronous, blocking API. One can thus send a command to the CCB server, to start a scan, by calling one function, and then call another function to await and acquire a frame of data from the new scan.

8.1 Functions in the libccbtestclient library

The functions in the library are defined as follows.

8.1.1 new_CCBTestClient()

This function creates the resources that it needs to configure and subsequently get data from a CCB server, and returns a pointer to the opaque, dynamically allocated object that contains these resources. It's function-prototype is:

```
CCBTestClient *new_CCBTestClient(const char *config);
```

It's sole argument is a string whose contents can be used to override the values of specified configuration parameters. This can thus be used to set the initial configuration, before connecting to the CCB server. See section 3.2 for a description of the format of this string.

If successful, the function returns a pointer to an opaque CCBTestClient, which contains the resources that are needed to talk to the CCB server. This pointer should be passed to subsequent calls to functions in the library. Note that this function doesn't actually open a connection to a CCB server. That must be done by subsequently calling ccb_tst_connect_client(). On error, the function reports an error to stderr and returns NULL.

8.1.2 ccb_tst_connect_client()

Once a CCBTestClient object has been created by a call to new_CCBTestClient(), a connection to a CCB server should be opened by calling the ccb_tst_connect_client() function. This has the following function prototype:

```
int ccb_tst_connect_client(CCBTestClient *ctc, const char *host);
```

The first argument is the pointer that was returned by the preceding call to new_CCBTestClient(), and the second argument is the name or IP address of the computer on which the CCB server is running. If successful, this function returns 0. Otherwise it reports an error to stderr and returns 1.

If this function is called when the specified CCBTestClient object is already connected to a server, then the former connection is quietly terminated via an internal call to ccb_tst_disconnect_client().

8.1.3 ccb_tst_disconnect_client()

The connection of a CCBTestClient object to a CCB server can be terminated either by deleting the CCBTestClient object, or by calling ccb_tst_disconnect_client(). The latter function has the following prototype:

```
void ccb_tst_disconnect_client(CCBTestClient *ctc);
```

Its sole argument is the pointer that was returned by the preceding call to new_CCBTestClient().

Note that it is also legal to call this function when there is no connection to be terminated.

8.1.4 del_CCBTestClient()

This function closes a previously opened connection to the CCB server, relinquishes the resources that were used to communicate with it, and returns NULL.

It's function-prototype is as follows:

```
CCBTestClient *del_CCBTestClient(CCBTestClient *ctc);
```

It's sole argument is the pointer that was returned by a previous call to new_CCBTestClient(). Note that it is legal for this to be NULL.

This function always returns NULL and is designed to be used in the following manner.

```
ctc = del_CCBTestClient(ctc);
```

where ctc is a CCBTestClient pointer that was previously returned by new_CCBTestClient(). By assigning the NULL return value of the function, to the pointer whose object has just been deleted, one can expect to see a segmentation fault if the program subsequently incorrectly tries to continue to use the deleted object.

8.1.5 ccb_tst_update_conf()

The configuration of subsequent scans can either be modified via an argument to the function that starts the scan, or can be modified by the ccb_tst_update_conf() function. The two methods can be used in conjunction, with ccb_tst_update_conf() being used to set up the parameters that remain constant during subsequent scans, and the scan-specific parameters being set up when each individual scan is started. The prototype of the ccb_tst_update_conf() function is as follows:

```
int ccb_tst_update_conf(CCBTestClient *ctc, const char *conf);
```

The first argument is the pointer that was returned by a preceding call to new_CCBTestClient(). The second argument is a string containing assignment specifications to any subset of the scan configuration parameters. The form of the contents of this string is described in section 3.2.

Take the following example.

```
ccb_tst_update_conf("integ_period=100 cal_steps=AB*5,NONE*10", cnf);
```

This would change the integ_period parameter, which sets the number of phase-switch cycles per integration-period, to have the value 100, and would configure the cal-diodes to repeatedly cycle through a sequence of being switched on for 5 integration periods then switched off for 10 integration periods.

8.1.6 ccb_tst_start_scan()

The ccb_tst_start_scan() function tells the CCB to start an integration-mode scan. Its prototype is as follows:

```
int ccb_tst_start_scan(CCBTestClient *ctc, const char *conf);
```

The first argument is the pointer that was returned by a preceding call to new_CCBTestClient(). The second argument is an optional string, containing assignment specifications to any subset of the scan configuration parameters. These configuration changes will take affect when the new scan starts. Unspecified parameters retain the values that they had in the previous scan, except where modified by an intervening call to ccb_tst_update_conf().

If the arguments are valid, and a connection exists to the CCB server, then the function will queue the request to start the scan, to be sent to the server, and return 0. Otherwise it will report an error to stderr and return 1.

Once this function has been called, integrated data from the scan can be received by calling the ccb_tst_read_integ_frame() function. Beware that the queued request to start the scan doesn't actually get sent until ccb_tst_read_integ_frame() is called, since the underlying event-loop only runs when waiting for data to arrive.

8.1.7 ccb_tst_read_integ_frame()

Once ccb_tst_start_integ() has been called, to tell the CCB to start a integration-mode scan, the ccb_tst_read_integ_frame() function can be called to read back one frame of integrated data at a time. Its prototype is as follows:

```
int ccb_tst_read_integ_frame(CCBTestClient *ctc, CCBIntegFrame *integ);
```

The first argument is the pointer that was returned by a preceding call to new_CCBTestClient(). The second argument must be a pointer to the variable into which the received data should be copied. The data-type of this variable is described in section 5.7.2.

If the connection to the CCB server is lost, or one of the arguments is NULL, or no integration-mode scan has been started, then ccb_tst_read_integ_frame() reports an error to stderr, and returns 1. Otherwise, once an integration-mode frame of data from the most recently requested scan, arrives from the CCB server, the function returns the received data in the variable pointed to by the integ argument, and returns 0, to indicate success.

Beware that the CCB server discards any integration-mode frames that arrive while a previous frame is still being dispatched to the CCB manager. Thus when the integration-period is short, then the manager may not receive every integration-mode frame that is acquired by the CCB.

8.1.8 ccb_tst_start_dump()

The ccb_tst_start_dump() function tells the CCB to start a dump-mode scan. Its prototype is as follows:

```
int ccb_tst_start_dump(CCBTestClient *ctc, const char *conf,
                       unsigned short adc, unsigned long samples,
                       unsigned long frames);
```

The first argument is the pointer that was returned by a preceding call to new_CCBTestClient(). The second argument is an optional string, containing assignment specifications to any subset of the scan configuration parameters. These configuration changes will take affect when the dump-scan starts. Unspecified parameters retain the values that they had in the previous scan, except where modified by an intervening call to ccb_tst_update_conf().

The third argument specifies from which of the CCB ADC-inputs, the samples should be acquired, and must be a number between 0 and CCB_NUM_ADC-1. An ADC designation of 0 corresponds to the input whose label on the front panel is J1.

The fourth argument specifies how many samples to acquire per integration period. If the requested number exceeds the maximum of CCB_DUMP_MAX_SAMPLES (currently 16383), then the number returned will be equal to this maximum.

Finally, the fifth argument specifies for how many integration periods of dump-mode frames should be sent to the program. Beware that the CCB server discards dump-mode frames that are generated while a previous dump-mode frame is still being dispatched to a client program. Thus if you ask for a particular number of frames, although you will receive the specified number of frames, they may not be from contiguous integration periods. To tell the CCB server to keep sending dump-mode frames indefinitely, specify the value CCB_DUMP_ALL_FRAMES.

If the arguments are valid, and a connection exists to the CCB server, then the function will queue the request to start the scan, to be sent to the server, and return 0. Otherwise it will report an error to stderr and return 1.

Once this function has been called, dump-mode data from the scan can be received by calling the ccb_tst_read_dump_frame() function. Beware that the queued request to start the scan doesn't actually get sent until ccb_tst_read_dump_frame() is called, since the underlying event-loop only runs when waiting for data to arrive.

8.1.9 ccb_tst_read_dump_frame()

Once ccb_tst_start_dump() has been called, to tell the CCB to start a dump-mode scan, the ccb_tst_read_dump_frame() function can be called to read back one dump-mode frame of data at a time. Its prototype is as follows:

```
int ccb_tst_read_dump_frame(CCBTestClient *ctc, CCBDumpFrame *dump);
```

The first argument is the pointer that was returned by a preceding call to new_CCBTestClient(). The second argument must be a pointer to the variable into which the received data should be copied. Note that the CCBDumpFrame datatype is described in section 7.1.3.

If the connection to the CCB server is lost, or one of the arguments is NULL, or no dump-mode scan has been started, then ccb_tst_read_dump_frame() reports an error to stderr, and returns 1. Otherwise, once a dump-mode frame of data, from the most recently requested scan, arrives from the CCB server, the function returns the received data in the variable pointed to by the dump argument, and returns 0, to indicate success.

Beware that the CCB server discards any dump-mode frames that arrive while a previous frame is still be dispatched to a dump-mode client. Thus when the integration-period is short, a dump-mode client program may not receive every dump-mode frame that is acquired by the CCB.

8.1.10 ccb_tst_mean_of_dump()

This function can be used to compute the mean and RMS of the samples within a dump frame that has previously been read from the CCB by the ccb_tst_read_dump_frame() function. Its prototype is as follows:

```
int ccb_tst_mean_of_dump(CCBDumpFrame *dump, double *mean,
                         double *rms, unsigned *n);
```

The first argument is a pointer to a dump-frame object that was filled in by a previous call to ccb_tst_read_dump_frame(). The remaining arguments should be pointers to the variables where the results should be returned. The function assigns the arithmetic mean of the samples in the dump-mode frame, to *mean, the root-mean-square deviation of the samples from this mean level, to *rms, and the number of unsaturated samples that contributed to the mean and RMS, to *n.

Note that only samples that don't have their overflow bits set, are included in the mean. Thus if all of the samples are saturated, then ccb_tst_mean_of_dump() returns non-zero, to indicate that it wasn't able to compute the mean. Otherwise it returns 0, to indicate success.

8.1.11 ccb_tst_get_config()

For cases where the use of a string to specify changes to the configuration, is inefficient or awkward, the ccb_tst_get_config() function can be used to get a pointer to the object that contains the binary configuration. This pointer can then be used with functions from libccbcommon, such as ccb_get_timing_cnf() and ccb_set_config() to modify the configuration that will be used by subsequently started scans. The prototype of ccb_tst_get_config() is as follows:

```
CCBConfig *ccb_tst_get_config(CCBTestClient *ctc);
```

The only argument is the pointer that was returned by a preceding call to new_CCBTestClient().

8.1.12 ccb_tst_set_dacs()

This function can be used to set the voltages of the analog outputs of the GPIO card. Although these outputs aren't currently used for anything by the CCB, they can be used to generate known voltages for DC input tests, as in the ccb_test_dc_response program. The function prototype of ccb_tst_set_dacs() is as follows:

```
int ccb_tst_set_dacs(CCBTestClient *ctc,
                     unsigned short counts[CCB_GPIO_NDAC]);
```

The first argument is the pointer that was returned by a preceding call to new_CCBTestClient(). For an explanation of the second argument please see the documentation of the ccb_queue_set_dacs_cmd() function, in section 4.11.1.

Index

Function index

ccb_add_intervals() 47
ccb_add_to_timestamp() 54
ccb_cal_cycle_length() 51
ccb_calibrate_monitor_data() 111
ccb_check_config 28
ccb_client_communicate() 65
ccb_client_connect() 63
ccb_client_disconnect() 64
ccb_client_poll_args() 67
ccb_client_select_args() 66
ccb_client_non_blocking_io() 65
ccb_client_selected_io() 67
ccb_client_sockets_callback() 68
ccb_clock_interval() 48
ccb_cmd_error_callback() 69
ccb_compare_intervals() 47
ccb_compare_timestamps() 53
ccb_compute_led_states() 111
ccb_copy_config() 28
ccb_cycle_length() 49
ccb_default_config() 28
ccb_diff_config() 28
ccb_fake_integrations() 55
ccb_get_cal_diode_cnf() 33
ccb_get_CCBDumpReader_fd 138
ccb_get_phase_switch_cnf() 31
ccb_get_sampler_cnf() 39
ccb_get_timestamp() 53
ccb_get_timing_cnf() 37
ccb_hms_of_timestamp() 55
ccb_integ_msg_callback() 87
ccb_integ_per_interval() 51
ccb_integration_duration() 50
ccb_integration_time() 50
ccb_interval_is_zero() 48
ccb_log_msg_callback() 90
ccb_log_server_msg() 115
ccb_monitor_msg_callback() 85
ccb_parse_CCBCalDiodes 44
ccb_parse_CCBConfig() 41
ccb_parse_CCBPhaseSwitches 43
ccb_phase_switch_sequence 114
ccb_ping_echos() 78
ccb_print_CCBConfig() 42
ccb_queue_dump_frame_msg() 113
ccb_queue_dump_scan_cmd() 74
ccb_queue_integ_msg() 112
ccb_queue_load_driver_cmd() 76
ccb_queue_logger_cmd() 77
ccb_queue_monitor_cmd() 76
ccb_queue_monitor_msg() 110
ccb_queue_ping_cmd() 78
ccb_queue_reboot_cmd() 79
ccb_queue_reset_cmd() 77
ccb_queue_set_dacs_cmd() 80
ccb_queue_shutdown_cmd() 79
ccb_queue_start_scan_cmd() 72
ccb_queue_status_request_cmd() . . . 79
ccb_queue_stop_scan_cmd() 73
ccb_queue_telemetry_cmd() 76
ccb_read_CCBConfig() 41
ccb_read_dump_frame() 138
ccb_render_CCBCalDiodes 44
ccb_render_CCBPhaseSwitches 43
ccb_scale_interval() 46
ccb_scan_duration() 50
ccb_server_event_loop() 109
ccb_set_cal_diode_cnf() 32
ccb_set_config() 29
ccb_set_phase_switch_cnf() 29
ccb_set_sampler_cnf() 38
ccb_set_timing_cnf() 34
ccb_settling_time() 49

ccb_status_reply_callback() 82
ccb_subtract_intervals() 47
ccb_time_to_timestamp() 54
ccb_time_until() 53
ccb_tst_connect_client() 141
ccb_tst_disconnect_client() 141
ccb_tst_get_config() 146
ccb_tst_mean_of_dump() 145
ccb_tst_read_dump_frame() 145
ccb_tst_read_integ_frame() 143
ccb_tst_set_dacs() 146
ccb_tst_start_dump() 144
ccb_tst_start_scan() 143
ccb_tst_update_conf() 142
ccb_write_CCBConfig() 42
ccb_zero_dump_frame() 114
ccb_zero_integ_frame() 113
ccb_zero_interval() 48
ccb_zero_timestamp() 53
del_CCBClientLink() 64
del_CCBConfig() 27
del_CCBDumpReader() 139
del_CCBServerDriver() 108
del_CCBServerLink() 108
del_CCBTestClient() 142
new_CCBClientLink() 62
new_CCBConfig() 27
new_CCBDumpReader() 137
new_CCBServerDriver() 102
new_CCBServerLink() 101
new_CCBTestClient() 140

Datatype index

CCBCalDiodeCmd 124
CCBCalDiodeCnf 33
CCBCalDiodes 33
CCBClientIOStatus 68
CCBClientLink 62
CCBClientSocketsFn 68
CCBCmdErrorFn 69
CCBCmdStatus 70
CCBCntrlCmdAck 130
CCBCntrlPingReply 129
CCBConfig 27
CCBConfigType 28
CCBControlCommand 123
CCBControlCommandHeader 122
CCBControlCommandType 122
CCBControlReply 129
CCBControlReplyType 128
CCBDriverCheckEventsFn 106
CCBDriverCmd 104
CCBDriverCmdID 103
CCBDriverLoadFn 102
CCBDriverSelectEventsFn 106
CCBDriverTellFn 103
CCBDriverType 76
CCBDriverUnloadFn 103
CCBDrvConfScan 104
CCBDrvIntraScan 105
CCBDrvStageScan 105
CCBDumpFrame 113
CCBDumpReader 137
CCBDumpScanCmd 125
CCBFpgaMonitorData 87
CCBGeneralStatus 83
CCBIntegFlags 88
CCBIntegFrame 112
CCBIntegMsg 131
CCBIntegMsgFn 88
CCBInterval 46
CCBLinkType 78
CCBLoadDriverCmd 128
CCBLoggerCmd 126
CCBLogLevel 91
CCBLogMsg 132
CCBLogMsgFn 90
CCBMonitorCmd 126
CCBMonitorData 86
CCBMonitorMsg 131
CCBMonitorMsgFn 85
CCBNetMsg 118
CCBNetMsgInfo 120
CCBNetMsgMember 119
CCBPanelLed 111
CCBPhaseSwitchCmd 123
CCBPhaseSwitchCnf 31
CCBPingCmd 127
CCBRawFpgaMonitorData 110

CCBRawMonitorData 110
CCBRebootCmd 127
CCBRebootRTCFn 107
CCBResetCmd 126
CCBSampleType 38
CCBSamplerCmd 124
CCBServerDriver 102
CCBServerLink 101
CCBSetDacsCmd 128
CCBSetDacsFn 108
CCBShutdownCmd 127
CCBShutdownRTCFn 107
CCBStartScanCmd 125
CCBStatusReply 129
CCBStatusReplyFn 83
CCBStatusRequestCmd 127
CCBStopScanCmd 125
CCBTelemetryCmd 126
CCBTelemetryHeader 130
CCBTelemetryStream 77
CCBTelemetryType 130
CCBTelemPingReply 133
CCBTestClient 140
CCBTimeStamp 52
CCBTimingCmd 124
CCBTimingCnf 37

Macro index

CCB_ADC_OVERFLOW_MASK 114
CCB_ADC_SAMPLE_SIZE 114
CCB_CLIENT_SOCKETS_FN 68
CCB_CMD_ERROR_FN 69
CCB_CONTROL_PORT 13
CCB_DRIVER_CHECK_EVENTS_FN . 106
CCB_DRIVER_LOAD_FN 102
CCB_DRIVER_SELECT_EVENTS_FN . 106
CCB_DRIVER_TELL_FN 103
CCB_DRIVER_UNLOAD_FN 103
CCB_DUMP_ALL_FRAMES 75
CCB_DUMP_MAX_SAMPLES 75
CCB_DUMP_PHASE_A 138
CCB_DUMP_PHASE_B 138
CCB_DUMP_PORT 13
CCB_FAKE_CYCLE_LENGTH 56
CCB_FIRST_FAKE_SAMPLE 57
CCB_NUM_PHASE_BINS 56
CCB_GPIO_COUNTS_TO_VOLTS . . 81
CCB_GPIO_LAST_DAC_COUNT . . . 80
CCB_GPIO_MAX_DAC_COUNT . . . 80
CCB_GPIO_MAX_DAC_VOLTS 80
CCB_GPIO_MIN_DAC_COUNT 80
CCB_GPIO_MIN_DAC_VOLTS 80
CCB_GPIO_NDAC 80
CCB_GPIO_VOLTS_TO_COUNTS . . 81
CCB_INTEG_MSG_FN 88
CCB_LOG_MSG_FN 90
CCB_LOG_PURGE_DT 85
CCB_LOGID 25
CCB_MAX_ADC_DELAY 37
CCB_MAX_DIODE_FALL_DT 35
CCB_MAX_DIODE_RISE_DT 35
CCB_MAX_HOLDOFF_DT 36
CCB_MAX_INTEG 90
CCB_MAX_INTEG_PERIOD 35
CCB_MAX_LOG 132
CCB_MAX_LOGID 25
CCB_MAX_LOG_VARIANTS 84
CCB_MAX_NCAL 32
CCB_MAX_PHASE_SWITCH_DT . . . 34
CCB_MAX_ROUNDTRIP_DT 35
CCB_MAX_SAMP_PER_STATE 31
CCB_MIN_INTEG_DT 37
CCB_MIN_SAMP_PER_STATE 31
CCB_MONITOR_MSG_FN 85
CCB_NEXT_FAKE_SAMPLE 57
CCB_NUM_ADC 75
CCB_NUM_FPGA 86
CCB_NUM_SLAVE 86
CCB_REBOOT_RTC_FN 107
CCB_ADC_SAMPLE_MASK 114
CCB_SATURATED_INTEGRATION . . 89
CCB_SET_DACS_FN 108
CCB_SHUTDOWN_RTC_FN 107
CCB_STATUS_REPLY_FN 83
CCB_TELEMETRY_PORT 13
CCB_TEST_DETECTOR_D2S 81
CCB_TEST_DETECTOR_S2D 81