The network interface between the Ygor manager and the CCB server.

Martin Shepherd
California Institute of Technology

June 16, 2003

This page intentionally left blank.

Abstract

This document documents the network communications interface between an Ygor manager process and the server of the Caltech Continuum Backend (CCB). It starts with a high-level overview of client and server communications library, proceeds to describe the public communication API that this library provides, and continues with descriptions of successively lower levels within the library.

Contents

1 Introduction **8**

1.1 A guide to reading this manual . . . 8

1.1.1 Points of interest to writers of the CCB Manager program . . . 8

1.1.2 Points of interest to writers of the CCB server program . . . 9

1.1.3 The organization of this manual . . . 9

1.2 The two TCP/IP links used by the library . . . 10

1.3 Connection establishment . . . 10

1.4 Connection authentication . . . 10

1.5 Initial configuration . . . 11

1.6 Single threaded versus multi-threaded . . . 11

1.7 Library usage caveats . . . 12

1.8 Shared libraries and their versioning . . . 13

2 Installation **15**

2.1 Getting the source code . . . 15

2.2 The basics of installation . . . 15

2.3 Compiling in a different directory . . . 16

2.4 Specifying where files are installed . . . 16

2.5 Generating this manual . . . 17

2.6 Testing the libraries using the demonstration programs . . . 17

2.6.1 ccb_dummy_client . . . 19

2.7 Run-time configuration files . . . 20

2.8 The ccb_authorized_ips configuration file . . . 20

3 The common parts of the CCB server and client APIs **21**

3.1 The configuration of the CCB . . . 21

3.1.1 The configuration of the phase switches 23
3.1.2 The configuration of the calibration diodes 26
3.1.3 The configuration of hardware timing parameters 28
3.2 Integration and scan timing information 30
3.2.1 Interval computations 32
ccb_scale_interval() 32
ccb_add_intervals() 32
ccb_subtract_interval() 33
ccb_compare_intervals() 33
ccb_zero_interval() 33
ccb_clock_interval() 34
3.2.2 The calibration diode delay 34
3.2.3 The number of measurements per cycle 36
3.2.4 The physical duration of an integration minus cal-diode delays 36
3.2.5 The physical duration of an integration period 37
3.2.6 The effective integration time 37
3.2.7 The duration of a scan 38
3.3 Timestamps 38
3.3.1 Getting the current date and time 39
3.3.2 Comparing two timestamps 39
3.3.3 Computing the amount of time remaining until a given time 40
3.3.4 Adding a time-interval to a timestamp 40
3.3.5 Converting a time_t value to a CCBTimeStamp value 40
3.3.6 Getting the clock time from a timestamp 41

4 The CCB client communications API **42**
4.1 Include files 44
4.2 The CCB-client communications library 44
4.3 Creating the client resources needed to talk to a CCB server 45
4.4 Connecting to a CCB server 46
4.5 Disconnecting from a CCB server 47
4.6 Deleting a redundant CCBClientLink object 47
4.7 Requesting non-blocking I/O 47
4.8 ccb_client_communicate() – Perform client socket I/O 48

4.9 Client I/O multiplexing 48
4.9.1 Using the select() system call 49
4.9.2 Using the poll() system call 49
4.9.3 Third party event handlers 50
4.9.4 Using threads to multiplex I/O 51
4.10 Registering a command-error callback function 52
4.11 Outgoing CCB commands 53
4.11.1 Outgoing CCB control commands 53
ccb_queue_start_scan_cmd() – Queuing a start-scan command 54
ccb_queue_stop_scan_cmd() – Queuing a stop-scan command 55
ccb_queue_monitor_cmd() – Queuing a monitor command 56
ccb_queue_telemetry_cmd() – Queuing a telemetry command 56
ccb_queue_logger_cmd() – Queuing a logger command 57
ccb_queue_reset_cmd() – Queuing a reset command 57
ccb_queue_standby_cmd() – Queuing a standby command 58
ccb_queue_awaken_cmd() – Queuing an awaken command 58
ccb_queue_ping_cmd() – Queuing a ping command 59
ccb_queue_status_request_cmd() – Queuing a status-request command 59
ccb_queue_shutdown_cmd() – Queuing a shutdown command 60
ccb_queue_reboot_cmd() – Queuing a reboot command 60
4.12 Incoming control-link replies 60
ccb_status_reply_callback() – Routing status-request replies 61
4.13 Incoming telemetry messages 62
ccb_monitor_msg_callback() – Routing telemetry monitor-data messages . . . 64
ccb_integ_msg_callback() – Routing telemetry integ-data messages 65
ccb_log_msg_callback() – Routing telemetry log-message messages 65
4.14 A TCL wrapper around the CCB client API 66

5 The CCB server communications API **74**

5.1 Include files 74
5.2 The CCB-server communications library 75
5.3 Creating the resources used to communicate with CCB managers 75
5.4 Shutting down server communications 81
5.5 Server I/O multiplexing 81

5.6 Queuing replies to control commands 82
5.7 Queuing outgoing telemetry messages 82
5.7.1 Queuing outgoing monitor-data messages 82
5.7.2 Queuing outgoing integ-data messages 83
5.7.3 Queuing outgoing log-message messages 83

6 Library internals 85
6.1 The message translation layer 87
6.1.1 Message structure specification 87
6.1.2 Supported data-types within message structures 87
6.1.3 CCBNetMsg - The base-class of all messages 87
6.1.4 Some example message structures 88
6.1.5 CCBNetMsgMember – Message field descriptions 88
6.1.6 CCBNetMsgInfo – Individual message descriptions 89
6.2 The CCB interface layer 90
6.2.1 The message structures of outgoing control messages 91
CCBPhaseSwitchCmd – The phase-switching configuration command 92
CCBCalDiodeCmd – The calibration diode configuration command . 92
CCBTimingCmd – The acquisition-timing configuration command . . 93
CCBStartScanCmd – The start-scan command 93
CCBStopScanCmd – The stop-scan command 93
CCBMonitorCmd – The monitor command 94
CCBTelemetryCmd – The telemetry command 94
CCBLoggerCmd – The logger command 94
CCBResetCmd – The reset command 95
CCBStandbyCmd – The standby command 95
CCBAwakenCmd – The awaken command 95
CCBPingCmd – The ping command 95
CCBStatusRequestCmd – The status-request command 96
CCBShutdownCmd – The shutdown command 96
CCBRebootCmd – The reboot command 96
6.2.2 The message structures of incoming control-link replies 96
CCBCntrlPingReply – A reply to a ping command 97
CCBStatusReply – A reply to a status-request command 97

CCBCntrlCmdAck – An acknowledgment to a control command . . . 98
6.2.3 The message structures of incoming telemetry messages 98
CCBIntegMsg – Integration data messages 99
CCBMonitorMsg – Monitor data messages 99
CCBLogMsg – CCB log messages 100
CCBTelemPingReply – A reply to a ping command 100
6.3 Sending network messages 100
6.4 Receiving network messages 101

List of Figures

3.1 Example phase-switching cycles . . . 24
3.2 The anatomy of a data-scan or intra-scan . . . 31

6.1 The CCB communications stack . . . 86

Chapter 1

Introduction

Communications between the CCB manager and the CCB server are facilitated by two communications libraries, one for use by the manager and the other for use by the CCB server. These libraries hide the specifics of the communications protocols used, the buffering used for non-blocking I/O, the queuing of outgoing messages, and the receipt of incoming messages. The library used by the manager is designed to operate either under the auspices of an arbitrary I/O event loop of a single-threaded manager, or under the control of I/O-handling threads within a multi-threaded manager.

1.1 A guide to reading this manual

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 the library internals. Most readers can either ignore non-pertinent sections of this manual, or superficially read them to gain a better understanding of how the various parts interoperate. Only the maintainer of the libraries need read all parts.

1.1.1 Points of interest to writers of the CCB Manager program

Since the programmers who write 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.

The remaining chapters that these readers do need to digest 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 42

  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 72

  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 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.

1.1.2 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.

1.1.3 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 both the implementors of the CCB manager and the CCB server, 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 final chapter documents the internals of the two libraries, including the communications protocols that the libaries use to exchange messages with each other. This is probably only of interest to the maintainer of these libraries.

Following 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.

1.2 The two TCP/IP links used by the library

The CCB server process has two 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.

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 under the control of the CCB server. The classes of data that are sent by the server over this link, and the frequency with which periodic data are sent, are configurable via commands sent to the server's control port, as will be described later.

1.3 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. The TCP/IP port numbers of these two servers are defined in ccbconstants.h, as C macros called CCB_CONTROL_PORT and CCB_TELEMETRY_PORT.

1.4 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 logged via the existing manager.

1.5 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, then switches into standby mode. The manager then has the option of overriding the server's default configuration parameters with its own, before sending an "on" command, to awaken the CCB.

1.6 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. An alternative strategy 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. For the manager side of the communications link, 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, one thread can be writing to the control socket while another is reading from this socket, and a third 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 registering callback functions 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, and that if the associated application callback data object is shared between callbacks that are invoked by different threads, that it is accessed in a thread-safe manner.

The server side of the 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.

1.7 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. Since these must be C functions, whereas the manager is apparently written in C++, both the prototypes of these callbacks and their definitions must be given C linkage. To facilitate this, macros are provided to declare and prototype each of the CCB callback functions, and these macros in turn include the macro EXTERNC, which expands to `extern "C"` when the library's public header file is included in a C++ program. For example, a typical callback definition macro might be the following,

```
#define CCB_EXAMPLE_CALLBACK_FN(fn) EXTERNC 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.

1.8 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/libccb.tar.gz
```

To extract the contents of this tar file, type,

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

2.2 The basics of installation

Configuration, compilation and installation are performed in the standard manner used in the free-software community.

```
cd libccb
./configure
make
make install
```

The provided autoconf configuration script currently only knows about Solaris and Linux, but 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

In the above example, compilation is performed in the directory that contains the source code of the CCB libraries. Alternatively, one can perform the compilation in a different directory, simply by going to that directory and running the configure script from there. For example:

```
gunzip -c libccb.tar.gz | tar xf -
mkdir linux
cd linux
../libccb/configure
make install
```

The makefile that the configure script generates contains pathnames to each component that it needs. This allows compilations for multiple operating systems to be performed from a single NFS-mounted copy of the source code.

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, the make install command creates them.

2.5 Generating this manual

A copy of the manual this is included as latex source code in the CCB software distribution. The following make commands can be used to create the manual in any of 3 formats. Note that since there isn't an obvious place to install the manual, the make install command ignores it, and it is left in the current directory.

- **make dvi**

  This command generates a version of the manual which can be viewed with the standard xdvi program.

- **make ps**

  This generates a version of the manual which can either be printed on a postscript printer, or displayed interactively, using a postscript viewer, such as ghostview, kghostview (KDE), pageview (Solaris), or ggv.

- **make pdf**

  This generates a PDF version of the manual, viewable with programs such as acroread or xpdf.

If when the configure was script run, the optional LaTeX hyperref package was found, 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.

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 two simple demonstration programs, these being a demonstration client and a demonstration server.

Since the installation paths of the shared libraries and the CCB configuration files are embedded within the executables, it is necessary to install 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 usually installed in the etc/ subdirectory of the 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 20).

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 argument 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/ccb_demo_server
```

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 ccb_demo_server, 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 ccb_demo_server 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 ccb_demo_server after adding an appropriate entry to the latter file.

Assuming that this all worked, the CCB server will initially be in standby mode, so press the "Awaken" button to wake it up. This button is colored bright green until you do this, to act as a reminder. 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 10 of these have been displayed, and every 10 integrations thereafter, the bottommost beige area will display fake monitoring data updates. The rates at which these data are received and displayed are set by the default configuration parameters shown in the window, and can be changed. In particular, when the "Configure Monitoring" button is pressed, the number in the entry widget to its right specifies how often fake monitor-data updates are sent. As described later, changes to the phase-switch, cal-diode and timing configuration parameters, which are presented for modification in the top three panes of the window, only take effect when a new scan or an 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" or "Stop scan" buttons, to start a fake new scan that operates according to these parameters.

As you use the GUI to send commands to the demonstration CCB server, the latter program prints messages on its parent terminal, indicating any effects that these commands would have on the real CCB device driver.

Neither of these programs knows, or cares whether the program on the other end of the communications link is also a demonstration program. Thus ccb_demo_server can be used to test the real CCB manager, and ccb_demo_client can be used to test the real CCB server.

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 subsequently 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 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. 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 of the CCB-server 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 16.

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 * wildcard. For example, the following two lines authorize all computers within the Green-Bank subnet, plus the author's computer at Caltech.

```
192.33.116.*        # All Green-Bank computers.
131.215.102.18      # The author's computer at Caltech
```

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

Chapter 3

The common parts of the CCB server and client APIs

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 CCB-Config 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,  /* Cal-diode parameters */
  CCB_CNF_TELEMETRY      = 4,  /* Telemetry parameters */
  CCB_CNF_TIMING         = 8,  /* Timing parameters */
/*
 * The union of all of the above.
 */
  CCB_CNF_ALL = CCB_CNF_PHASE_SWITCHES | CCB_CNF_CAL_DIODES |
                CCB_CNF_TELEMETRY | CCB_CNF_TIMING
} CCBConfigType;
```

The ccb_diff_config() function compares two CCB configuration objects and returns a list of the configuration groups whose parameters differ.

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

The return value is a bit-mask union of CCBConfigType enumerators.

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);
```

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 measurement in this diagram is just an example of what can be configured, the number of measurements per cycle is fixed by the number of switches that are active, and is thus not otherwise configurable. In addition to the parameters illustrated, it is also be possible to individually switch off each of the TTL outputs, regardless of whether those outputs are configured to be switching or not. This potentially reduces interference while other backends are controlling the phase switches.

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

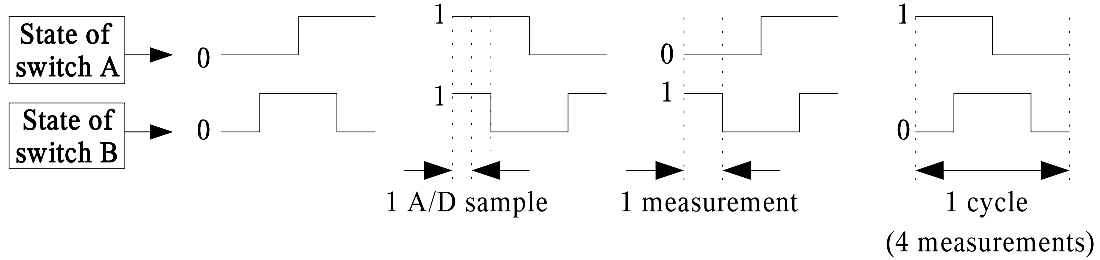
```
int ccb_set_phase_switch_cnf(CCBConfig *cnf,
                             unsigned short active_switches,
                             unsigned short driven_switches,
                             unsigned short closed_switches,
                             unsigned short samp_per_state);
```

The arguments of this function are interpreted as follows.

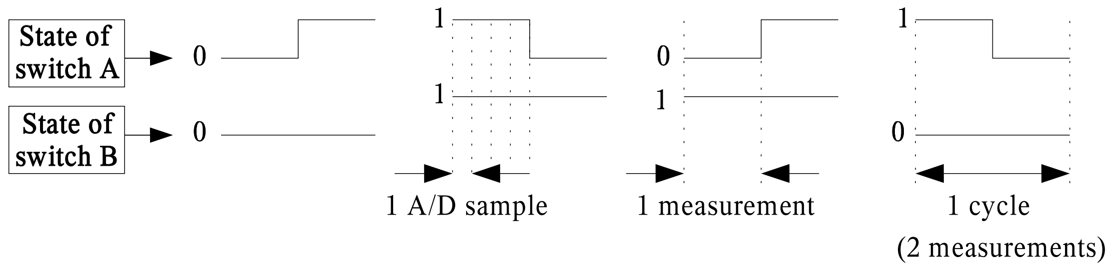
- **Which switches are active?** (active_switches)

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

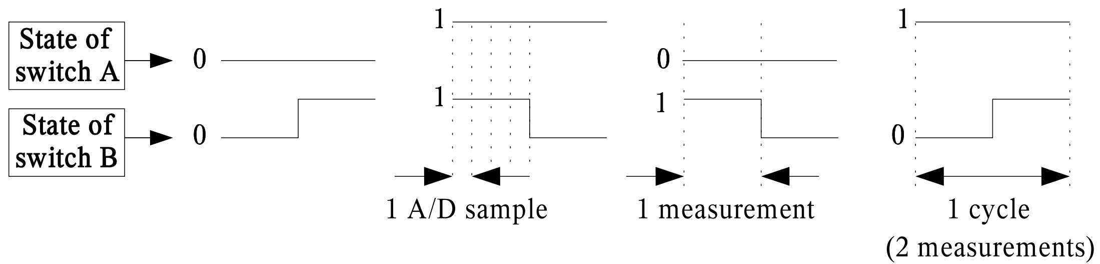
The 4 possible phase-switch cycles with both phase switches switching



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

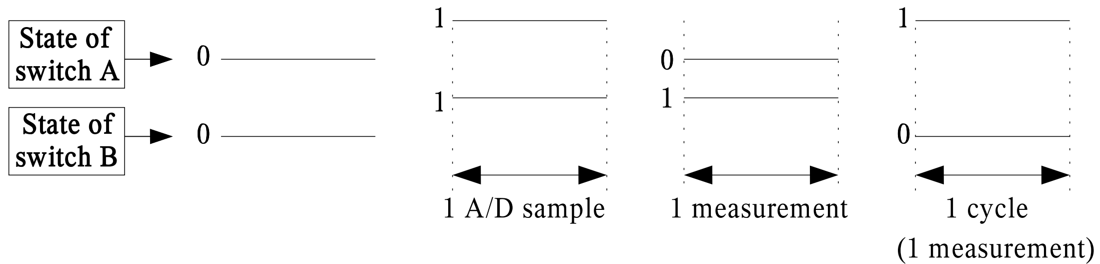


Figure 3.1: Example phase-switching cycles

```
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 are driven?** (driven_switches)

  This specifies the set of phase switches whose control lines are to be driven, expressed as a bitwise union of CCBPhaseSwitches enumerators. The remaining phase-switch control lines are placed in a high impedance state, allowing them to be controlled by a different receiver backend.

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

  This argument specifies the set of phase switches that are to be closed 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. Since the underlying state machine for each cycle is clocked by the sample clock, and there is room for 32 states per cycle, the product of the number of samples per measurement and the number of measurements per cycle sets an upper limit on the value of this parameter as follows:

Number of enabled phase-switches	Maximum number of samples per measurement
0	32
1	16
2	8

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 driven_switches;  /* Which switches are driven? */
  unsigned short closed_switches;  /* Which switches start closed? */
  unsigned short samp_per_state;   /* 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.

```
enum {CCB_MAX_NCAL=32};  /* The maximum number of calibration steps */
```

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 driven_diodes,
                          unsigned short ncal,
                          const short *diode_states,
                          const unsigned long *diode_times);
```

The arguments of this function, are as follows.

- **The set of driven cal-diodes** (driven_diodes)

  The set of calibration diodes whose control lines are to be driven, expressed as 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;
  ```

  The control lines of calibration diodes that aren't specified to be driven, are placed in a high-impedance state, such that a different receiver backend can control them.

- **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.

  This argument can be NULL if ncal is 0.

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

  Each of the first ncal elements of this parameter specifies for how many integrations the state of the corresponding stage of the calibration diode state machine should be maintained. With an 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 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 calibration-diode switches within a CCB configuration object can be queried using 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 driven_diodes;           /* The set of driven */
                                          /*  calibration diodes. */
  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 in 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 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 sample_dt,
                       unsigned short phase_switch_dt,
```

```
                        unsigned short analog_reset_dt,
                        unsigned long diode_rise_dt,
                        unsigned long diode_fall_dt,
                        unsigned long integ_period);
```

The arguments of this function are interpreted as follows.

- **A/D sample interval** (sample_dt)

  This refers to the amount of time taken per A/D sample, including the time used to blank phase-switch transitions, but not including the time spent resetting the analog integrators. It is expressed as an integer multiplier of 100ns, and has a nominal value of 250 (25μs).

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

  This specifies how much of the sample interval is lost to waiting for the phase switches to settle after phase-switch transitions. It is expressed as an integer multiplier of 100ns.

- **Integrator-reset blanking interval** (analog_reset_dt)

  This specifies how long to wait for the analog integrator to reset before starting to integrate a new sample. It is expressed as an integer multiplier of 100ns, so for the predicted reset-time of around 1μs, analog_reset_dt would be 10.

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

  This specifies the minimum time to wait before starting any integration that starts with either of the calibration diodes being newly switched on. It is expressed as an integer multiplier of 100ns.

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

  This specifies the minimum time to wait before starting any integration that starts with either of the calibration diodes being newly switched off. It is expressed as an integer multiplier of 100ns.

- **The integration period** (integ_period)

  This specifies the number of phase-switch cycles that are co-added to form the integrations that are sent to the manager. The physical length of time that this corresponds to depends on the length of an A/D sample and the number of samples per phase-switch cycle.

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.

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 sample_dt;        /* The duration of an A/D sample */
  unsigned short phase_switch_dt;  /* The settling time of the phase */
                                   /*  switches. */
  unsigned short analog_reset_dt;  /* The amount of time needed to */
                                   /*  reset the analog integrators. */
  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 */
} 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.

3.2 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.

1. The integration-time, which is the total amount of time during which data are being accumulated in the integration 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, but also because of the delays that have to be inserted between ADC samples to account both for the time spent resetting the analog integrator, and for the settling times of the phase-switches and calibration diodes.

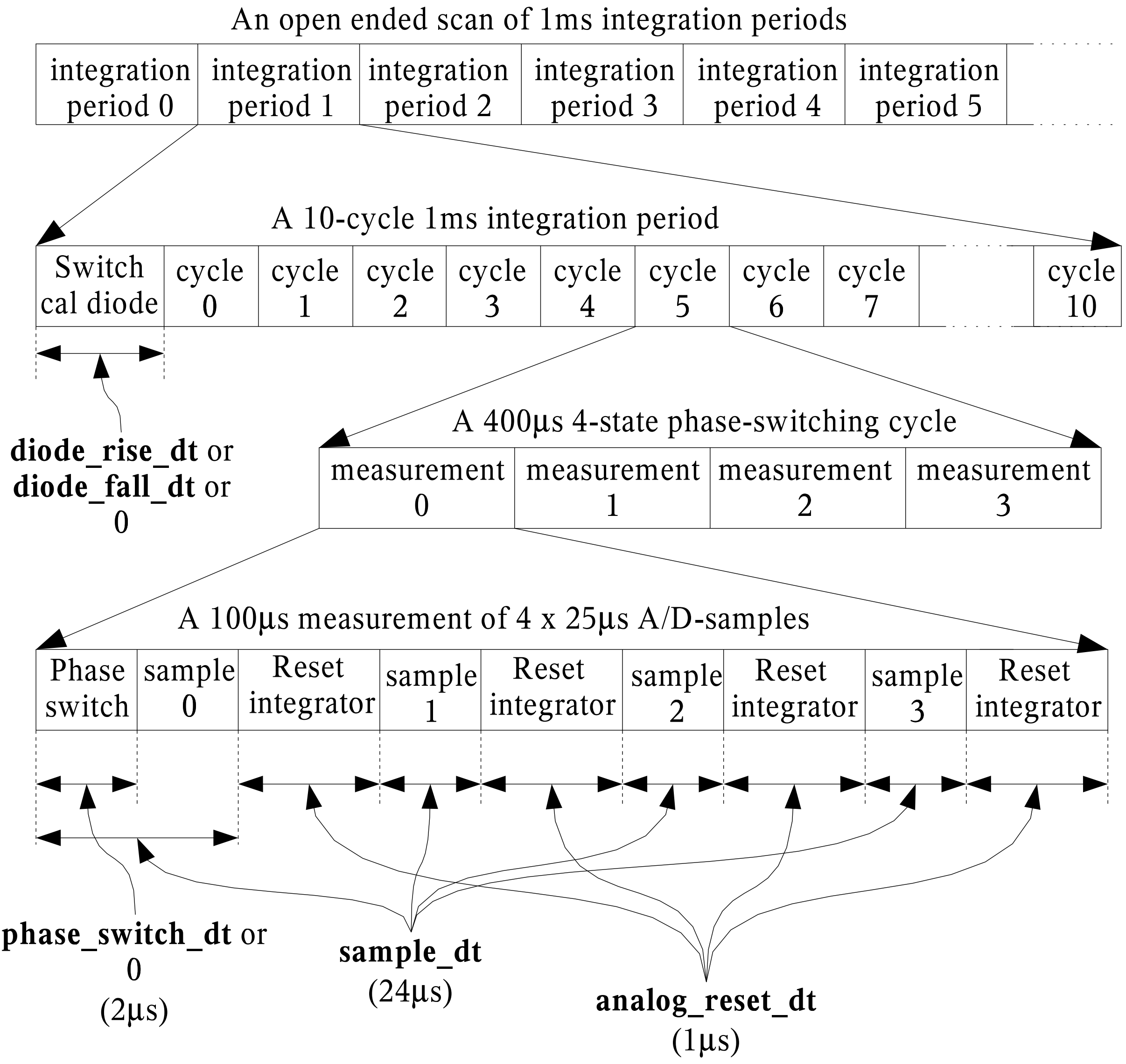


Figure 3.2: The anatomy of a data-scan or intra-scan

3.2.1 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_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.

3.2.2 The calibration diode delay

The settling time of the calibration diodes is not constant from one integration to the next. This is because:

1. At the start of the first integration of a new scan, the CCB doesn't know what states the calibration diodes and phase switches are currently in, so the initial calibration diode delay has to be set to the worst case settling times to reach the desired initial states of the calibration diodes and phase switches. The separate delays, whose maximum is adopted, are as follows.

   The delay needed by calibration diode A:
   If calibration diode A is not configured to be driven:

   $$\text{diode_a_delay} = 0 \tag{3.1}$$

   Otherwise, if calibration diode A should initially be on,

   $$\text{diode_a_delay} = \text{diode_rise_dt} \tag{3.2}$$

   Otherwise,

   $$\text{diode_a_delay} = \text{diode_fall_dt} \tag{3.3}$$

   The delay needed by calibration diode B:
   diode_b_delay is computed similarly to diode_a_delay.

   The delay needed by the phase switches:
   If neither of the phase switches is configured to be driven,

   $$\text{phase_switch_delay} = 0 \tag{3.4}$$

   Otherwise, if at least one of the phase switches is being driven,

   $$\text{phase_switch_delay} = \text{phase_switch_dt} \tag{3.5}$$

The combined delay:

The delay that is needed at the start of the first integration is thus the value of:

$$\mathsf{delay} = \max\left[\mathsf{diode_a_delay}, \mathsf{diode_b_delay}, \mathsf{phase_switch_delay}\right] \tag{3.6}$$

2. At the start of the remaining integrations of the scan, the settling time of the calibration diodes depends on whether diodes are being switched on or off. Again, the appropriate delay is determined by computing the individual delays needed by the two diodes, and taking the maximum.

The delay needed by calibration diode A:

If calibration diode A is not configured to be driven:

$$\mathsf{diode_a_delay} = 0 \tag{3.7}$$

Otherwise, if calibration diode A is being switched from off to on,

$$\mathsf{diode_a_delay} = \mathsf{diode_rise_dt} \tag{3.8}$$

Otherwise, if diode A is being switched from on to off,

$$\mathsf{diode_a_delay} = \mathsf{diode_fall_dt} \tag{3.9}$$

Otherwise, diode A must be remaining in the same state, so

$$\mathsf{diode_a_delay} = 0 \tag{3.10}$$

The delay needed by calibration diode B:

diode_b_delay is computed similarly to diode_a_delay.

The combined delay:

The delay that is needed at the start of the first integration is thus the value of:

$$\mathsf{delay} = \max\left[\mathsf{diode_a_delay}, \mathsf{diode_b_delay}\right] \tag{3.11}$$

So the durations of neighboring integration periods differ, and to determine the duration of a given integration period one needs to know its sequential number within the parent scan, and the set of states through which the diodes cycle from one integration to the next. To enable the manager to predict the durations of integration periods, the ccb_cal_diode_delay() function is provided for computing the diode-delay of a given integration within a scan.

```
int ccb_cal_diode_delay(const CCBConfig *cnf,
                        unsigned long integration,
                        CCBInterval *dt);
```

The cnf argument specifies the configuration of the CCB in the target scan. The integration argument is the sequential number of the integration period within that scan, starting at zero for the first integration period. The answer is returned in the variable pointed to by the dt argument.

The return value of ccb_cal_diode_delay() is normally 0, but if an error prevents the scan duration from being computed, then errno is set accordingly, and the function returns 1.

3.2.3 The number of measurements per cycle

The total amount of time within an integration that is lost to phase-switch transitions and resetting the analog integrator, depends on the number of measurements 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:

$$\text{measurements_per_cycle} = 2^{\text{nswitching}} \tag{3.12}$$

Based on this equation, the ccb_cycle_length() function returns the number of measurements per cycle, corresponding to a particular value of the active_switches argument of ccb_set_phase_switch_cnf() (see page 23).

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

3.2.4 The physical duration of an integration minus cal-diode delays

When we ignore the calibration-diode settling-time delays that precede the commencement of integration, then the physical duration of an integration is given by the product of the number of A/D samples per integration and the period between the start of one A/D sample and start of the the next.

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

The period between the start of one A/D sample and the start of the next, is the sum of the settling time of the analog integrator (analog_reset_dt) and the A/D sample interval (sample_dt).

This duration can be computed by calling the ccb_nocal_duration() function.

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

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

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

3.2.5 The physical duration of an integration period

The period between the start of one integration and the start of the next is the sum of the times returned by ccb_nocal_duration() and ccb_cal_diode_delay(). This computation is performed by the ccb_integration_duration() function.

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

The cnf argument specifies the configuration of the CCB during the target scan, the integration argument specifies which integration's period is required, and the integration period is returned in the period variable.

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.

3.2.6 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. The difference is due to the delays that are added to accommodate calibration-diode switching at the start of some integrations, the phase-switch transition delays that are subtracted from the first A/D sample of each new phase-switch state (only when phase switching), and the time needed to reset the analog integrators at the start of each new A/D sample. This calculation is performed 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.

3.2.7 The duration of a scan

Since in general the calibration diode settling time differs from one integration to the next, the ccb_scan_duration() function is provided to compute the physical amount of time taken by a scan of a given number of integrations.

```
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.

This function tallies up the number of times each of the stages of the calibration-diode state-machine is repeated during the scan, and scales these numbers by the corresponding settling times. Account is taken for the first integration of the initial calibration cycle of a scan potentially requiring a different calibration-diode delay than those of subsequent calibration cycles.

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 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 seconds into the day */
  unsigned long ns;     /* The number of nano-seconds within */
                        /*  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.

3.3.1 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.

3.3.2 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.

3.3.3 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 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. It was written for the CCB simulator incorporated in the ccb_demo_server program, where the actual timing achieved isn't critical.

3.3.4 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.

3.3.5 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.

3.3.6 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.

Chapter 4

The CCB client 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. **Set up the initial CCB configuration:**
```
if(ccb_set_phase_switch_cnf(cnf, ...) ||
   ccb_set_cal_diode_cnf(cnf, ...) ||
   ccb_set_timing_cnf(cnf, ...))
  return ERROR;
```
7. **Queue the first start-scan command:**
```
if(ccb_queue_start_scan_cmd(cl, id, cnf, ...))
  return ERROR;
```
8. **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;
    }
```

9. **Disconnect from the current CCB server:**
   ```
   ccb_client_disconnect(cl);
   ```

10. **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 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 -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 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 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 has to provide 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 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 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) EXTERNC 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 can 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) EXTERNC 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 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 reordered. 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 nonexistent 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 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 scan at the end of the current sample
reset	Re-initialize the hardware.
standby	Relinquish control of the receiver.
awaken	Take control of the receiver.
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.

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-PPS boundary specified by the mjd and tod parameters. If the command is received less than 1 integration time before the requested time, the new scan is not started on the desired 1-PPS, but instead starts as soon as possible, just like a stop-scan command. A log message is dispatched to alert the operator when this happens.

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 least 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-PPS. 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_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. Note that during standby mode, this selection is bitwise ANDed with the temporary stream mask specified by the standby command.

  ```
  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 */
    CCB_ALL_STREAMS         /* All of the above 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 63), 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 itself, the CCB device-driver 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 driver. This not only resets the device driver, but also resets the CCB hardware. The CCB server then places the CCB in standby mode, with only log messages selected to be sent to the manager. Finally it 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 subsequently calling ccb_client_communicate() to send it to the CCB server.

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

ccb_queue_standby_cmd() – Queuing a standby command

On receiving this command, the CCB server tells the hardware to stop driving any of the receiver control lines. Its argument also allows the manager to specify what forms of telemetry data should continue to be sent to it.

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

```
int ccb_queue_standby_cmd(CCBClientLink *cl, long id,
                          unsigned short stream_mask);
```

The arguments of this function are interpreted as follows.

- **The telemetry data-stream selection mask** (stream_selection)

  This parameter contains a bit-wise union of CCBTelemetryStream enumerators, specifying which of the currently enabled telemetry streams should continue to be sent to the manager. To determine this, the CCB server takes the union of the set of streams specified in this parameter with the set previously specified by the last CCBTelemetryCmd command.

  For example, if all currently enabled streams should continue to be sent, this parameter should be specified as CCB_ALL_STREAMS, whereas if no streams should continue be sent to the manager, it should be set to CCB_NO_STREAMS, and if all but the integration data should continue to be sent, it should be set to (CCB_All_STREAMS & C̃CB_INTEG_STREAMS).

ccb_queue_awaken_cmd() – Queuing an awaken command

On receiving this command, the server tells the hardware to start driving any receiver control lines that are currently configured to be driven, and to start sending subsequent integrations to the manager. It also undoes the effect of the telemetry stream mask that was specified by the preceding standby command.

Note that a new scan isn't automatically started by this command.

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

```
int ccb_queue_awaken_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 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 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 driver, which has the side effect of turning off all receiver control lines and stopping all CCB interrupts, then initiates a 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 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 driver, which has the side effect of turning off all receiver control lines and 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);
```

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 whenever it is called. This should be used by the manager to pass the callback function any resources that it needs to handle 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) EXTERNC 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 */
  CCB_STANDING_BY = 16  /* The CCB is in standby mode */
} 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	Allow the queue to drain
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 log 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) EXTERNC int (fn)(CCBClientLink *cl, \
                                                void *data, \
                                                const CCBTimeStamp *ts,   \
                                                unsigned long scan, \
                                                unsigned long number,       \
                                                const unsigned long *values,\
                                                unsigned nvalues)

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 first nvalues elements of the array pointed to by the values[] argument, contain the monitoring data points.

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) EXTERNC int (fn)(CCBClientLink *cl, \
                                              void *data, \
                                              const CCBTimeStamp *ts,   \
                                              unsigned long scan, \
                                              unsigned long number,       \
                                              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 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 integration within that scan, starting from 0. The manager can use the integration number to check for missing integ-data messages. The first nvalues elements of the array pointed to by the values[] argument, contain the radiometer integrations.

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) EXTERNC 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 reset
    This resets the CCB.

  – ccb send standby *streams*
    This places the CCB in standby mode. The streams argument specifies the set of telemetry streams that the CCB server should continue to send us, 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 56).

- ccb send awaken

  This cancels any previously sent standby command.

- 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, the streams argument specifies the list of streams to send, using the same format as already documented for the ccb send standby command.

- 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 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 driven_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 3 arguments, which each 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 *driven_diodes 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 driven_diodes argument, which specifies a set of calibration-diodes, 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_states argument is represented as a Tcl list of calibration-diode sets, each member of the list having the same format as the driven_diodes.

  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 *sample_dt phase_switch_dt analog_reset_dt diode_rise_dt diode_fall_dt integ_period*

  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 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  standing_by
```

These correspond to the similarly named CCBGeneralStatus enumerators (see page 61).

– 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 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. nvalue - The number of integrated values.
7. values... - The nvalue integrated values.

– 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. nvalue - The number of integrated values.
7. values... - The nvalue 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 66).
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 52). 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 show_integ {puts "Integration: [ccb get integ]"}
```

```
ccb attach monitor show_monitor {puts "Monitor: [ccb get monitor]"}
ccb attach log show_log {puts "Log: [ccb get log]"}
ccb attach error show_error {puts "Error: [ccb get error]"}
ccb attach status show_status {puts "Status: [ccb get status]"}

# Change the default timing, to slow down integration periods from
# the default of 1ms to 1s.

ccb configure timing 250 10 10 1000 1000 40000

# 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

# The CCB starts out in standby-mode, so send the command to awaken it.

ccb send awaken

# 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 ccb_demo_server 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, and that the CCB_CONF_DIR environment variable specifies the directory where said file resides.

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(void *data,
                                 CCBDriverLoadFn *load_driver,
                                 CCBDriverUnloadFn *unload_driver,
                                 CCBDriverTellFn *tell_driver,
                                 CCBRebootRTCFn *reboot_rtc,
                                 CCBShutdownRTCFn *shutdown_rtc);
```

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.

- **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)(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)(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)(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_INTR_MASK,   /* Enable/disable interrupts */
    CCB_DRV_LINE_MASK,   /* Specify a control-line drive */
                         /*  mask to be bitwise ANDed with */
                         /*  the configured set of driven */
                         /*  lines. */
    CCB_DRV_ABORT_SCAN,  /* Abort the current scan, and */
                         /*  don't start a new one. */
    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 */
  } 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 {
  ```

```
    CCBDrvIntrMask intr;   /* type==CCB_DRV_INTR_MASK */
    CCBDrvLineMask line;   /* type==CCB_DRV_LINE_MASK */
    CCBDrvConfScan conf;   /* type==CCB_DRV_CONF_SCAN */
    CCBDrvStageScan stage; /* type==CCB_DRV_STAGE_SCAN */
    CCBDrvIntraScan intra; /* type==CCB_DRV_INTRA_SCAN */
  } pars;
};
```

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

– **Set the CCB interrupt mask**
  This function is used to enable and/or disable generation of interrupts by the CCB hardware. There are two interrupt sources in the CCB hardware, and these are enumerated by the CCBDrvInterrupt type.

```
typedef enum {
  CCB_1PPS_INTR=1,  /* The 1-PPS interrupt */
  CCB_INTEG_INTR=2  /* The integration-done interrupt */
} CCBDrvInterrupt;
```

  When the type member of a CCBDriverCmd argument is CCB_DRV_INTR_MASK, the intr member of the pars union, contains the parameters of the command, encapsulated within a CCBDrvIntrMask structure.

```
typedef struct {
  unsigned enable;  /* A bitmask union of */
                    /*  CCBDrvInterrupt enumerators */
                    /*  specifying which interrupts */
                    /*  to enable. */
} CCBDrvIntrMask;
```

– **Set the control-line driver mask**
  This command specifies which receiver control lines can be selected to be driven by the scan configuration, and which lines are to be placed in a high impedance state, regardless of the scan configuration. These control lines are individually enumerated by the CCBControlLine datatype.

```
typedef enum {
  CCB_DIODE_A_LINE = 1, /* Calibration-diode A's */
                        /*   control lines. */
  CCB_DIODE_B_LINE = 2, /* Calibration-diode B's */
                        /*   control lines. */
  CCB_PHASE_A_LINE = 4, /* Phase-switch A's */
```

```
                          /*  control line. */
  CCB_PHASE_B_LINE = 8, /* Phase-switch B's */
                          /*  control line. */
  CCB_ALL_LINES =         /* All of the above */
    CCB_DIODE_A_LINE | CCB_DIODE_B_LINE |
    CCB_PHASE_A_LINE | CCB_PHASE_B_LINE,
  CCB_NO_LINES = 0        /* None of the above */
} CCBControlLine;
```

When the type member of a CCBDriverCmd argument is CCB_DRV_LINE_MASK, the line member of the pars union, contains the parameters of the command, encapsulated within a CCBDrvLineMask structure.

```
typedef struct {
  unsigned lines;  /* A bitmask union of */
                   /*  CCBControlLine enumerators */
                   /*  specifying which receiver */
                   /*  control lines can be driven, */
                   /*  if so configured. */
} CCBDrvLineMask;
```

The specified bit-mask of control lines is ANDed with the set of control lines that are selected to be driven by the scan configuration.

– **Abort the current scan**

  This command immediately aborts any ongoing scan or intra-scan, and makes the hardware receptive to scan configuration changes. No new scan is started. This command takes no arguments, so there is no corresponding member within the pars union.

– **Configure the next scan**

  Between sending an abort-scan command and either a stage-scan or an intra-scan command to the driver, CCBServerLink objects invoke this driver command to configure the hardware for the next scan or intra-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. */
} 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 abort-scan driver-command to terminate any existing scan or intra-scan, the configure-scan driver-command to install the configuration parameters of the requested scan, then finally 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 */
  unsigned long mjd;  /* The date on which to start, */
                      /*  expressed as a Modified */
                      /*  Julian day number. */
  unsigned long secs; /* The time of day at which to */
                      /*  start the scan, expressed */
                      /*  in seconds since 0H UTC on */
                      /*  the day specified in 'mjd'. */
} 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 abort-scan driver-command to terminate any existing scan or intra-scan, the configure-scan driver-command to install the configuration parameters of the requested scan, then finally 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.

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. For its part, the server is expected to use select() to watch for both device-driver I/O, and any socket I/O that the library is currently interested in. Since only the library knows what I/O it is waiting for, the library provides the ccb_server_select_args() function, which augments the existing contents of specified select() file-descriptor sets with the descriptors that it wants the server to watch.

```
int ccb_server_select_args(CCBServerLink *sl, fd_set *rfds,
                           fd_set *wfds, int *maxfd);
```

The rfds argument is the set of file descriptors that select() is to watch for the arrival of new data, and the wfds argument is the set of file descriptors that it should watch for the ability to write at least one byte without blocking. On input *maxfd should contain the maximum of any file descriptors that the server has currently placed in rfds and wfds. On output, if any of the socket file descriptors which the library adds to rfds and wfds exceed the input value of *maxfd, then *maxfd is given the value of the maximum socket file descriptor that it adds. Note that the first argument to select() should be the value of *maxfd + 1, not *maxfd. Normally ccb_server_select_args() returns 0, but if an error occurs, it returns 1 and sets errno accordingly.

The server calls ccb_server_select_args() before each call to select(), and then when select() returns, it calls ccb_server_communicate() to perform any socket I/O that select() indicated.

```
int ccb_server_communicate(CCBServerLink *ccb,
                           const fd_set *rfds,
                           const fd_set *wfds);
```

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. When ccb_server_communicate() is called to send pending telemetry 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 *ts,
                          unsigned long scan,
                          unsigned long number,
                          const unsigned long *values,
                          unsigned nvalues);
```

Apart from the initial sl argument, the arguments of this function are as described in section 4.

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 CCBTimeStamp *ts,
                        unsigned long scan,
                        unsigned long number,
                        const unsigned long *values,
                        unsigned nvalues);
```

Apart from the initial sl argument, the arguments of this function are as described in section 4.

5.7.3 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.

The id argument, which must be specified using the CCB_LOGID() documented below, macro is a numeric identifier of the log message. The first CCB message is assigned the value of 0, and subsequent messages are assigned successively higher numbers. Within ccb_log_server_msg(), this identification number is added to the integer offset specified in the macro CCB_BASE_LOGID, before being sent to the manager. The value of this macro is the start of the range of message-id numbers that are uniquely assigned by Green Bank to the CCB. When writing a new ccb_log_server_msg() statement, the id argument must be set to CCB_LOGID(). This allows the scripts that are used by the CCB makefile to search for, and subsequently fill in, the IDs of new log statements. These scripts simply place the newly assigned ID as the argument of the CCB_LOGID() macro, which although it does nothing but echo its argument, is retained, so that other scripts can search for it when figuring out which IDs have been used so far. This is performed both to figure out what numbers to give new ccb_log_server_msg() calls, and to set the value of the CCB_MAX_LOGID parameter in the ccblogid.h header file.

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.

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.

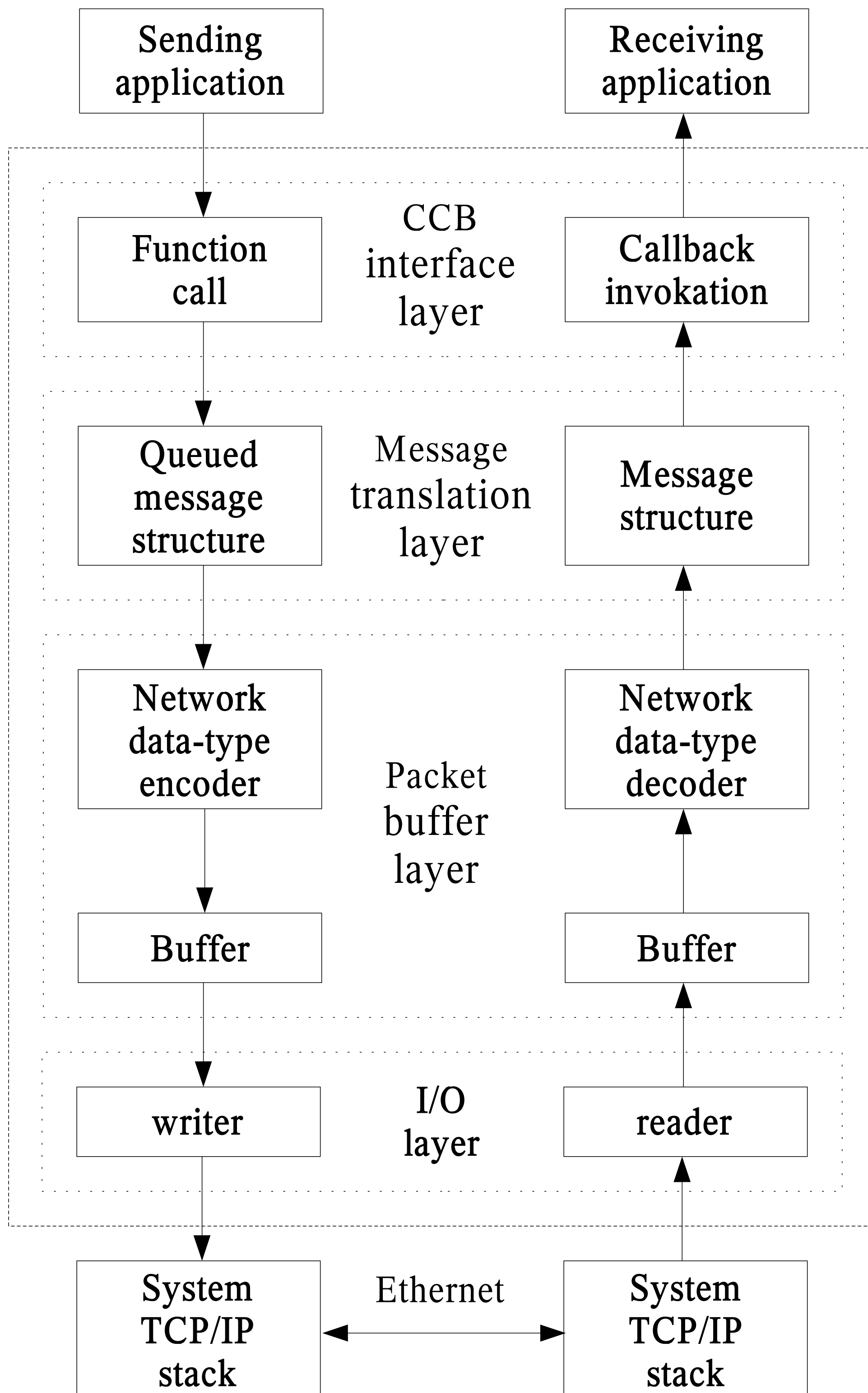


Figure 6.1: The CCB communications stack

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_MSG,   /* A phase-switch config command */
  CCB_CAL_DIODE_MSG,      /* A cal-diode config command */
  CCB_TIMING_MSG,         /* An timing config command */
  CCB_START_SCAN_CMD,     /* A start-scan command */
  CCB_STOP_SCAN_CMD,      /* A stop-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_AWAKEN_CMD,         /* An awaken command */
  CCB_STANDBY_CMD,        /* A standby 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 */
} CCBControlCommandType;
```

As documented on page 52, 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 */
  CCBStartScanCmd start_scan;  /* head.base.type=CCB_START_SCAN_CMD */
  CCBStopScanCmd stop_scan;    /* head.base.type=CCB_STOP_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 */
  CCBAwakenCmd awaken;         /* head.base.type=CCB_AWAKEN_CMD */
  CCBStandbyCmd standby;       /* head.base.type=CCB_STANDBY_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 */
} 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_start_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;
```

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 */
                                /*  OH 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;
```

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;
```

CCBStandbyCmd – **The** standby **command**

The ccb_queue_standby_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_STANDBY_CMD */
  unsigned short stream_mask;    /* The telemetry selection mask */
} CCBStandbyCmd;
```

CCBAwakenCmd – **The** awaken **command**

The ccb_queue_awaken_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_AWAKEN_CMD */
} CCBAwakenCmd;
```

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;
```

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 0H 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.

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

```
#define CCB_MAX_MONITOR=16 /* The maximum number of monitoring */
                           /* measurements from any instrument */

typedef struct {
  CCBTelemetryHeader head;         /* head.base.type=CCB_MONITOR_MSG */
```

```
  unsigned long scan;              /* The number of the parent scan */
  unsigned long id;                /* The monitor ID */
  unsigned long data[CCB_MAX_MONITOR];  /* The monitor data */
} 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.

Index

Function index

ccb_add_intervals() 32
ccb_add_to_timestamp() 40
ccb_cal_diode_delay() 35
ccb_check_config 22
ccb_client_communicate() 48
ccb_client_connect() 46
ccb_client_disconnect() 47
ccb_client_poll_args() 49
ccb_client_select_args() 49
ccb_client_non_blocking_io() 47
ccb_client_selected_io() 49
ccb_client_sockets_callback() 50
ccb_clock_interval() 34
ccb_cmd_error_callback() 52
ccb_compare_intervals() 33
ccb_compare_timestamps() 39
ccb_copy_config() 22
ccb_status_reply_callback() 61
ccb_cycle_length() 36
ccb_default_config() 21
ccb_diff_config() 22
ccb_get_cal_diode_cnf() 28
ccb_get_phase_switch_cnf() 25
ccb_get_timestamp() 39
ccb_get_timing_cnf() 30
ccb_hms_of_timestamp() 41
ccb_integ_msg_callback() 65
ccb_integration_duration() 37
ccb_integration_time() 37
ccb_log_msg_callback() 65
ccb_log_server_msg() 83
ccb_monitor_msg_callback() 64
ccb_nocal_duration() 37
ccb_ping_echos() 59
ccb_queue_awaken_cmd() 58
ccb_queue_integ_msg() 83
ccb_queue_logger_cmd() 57
ccb_queue_monitor_msg() 82
ccb_queue_monitor_cmd() 56
ccb_queue_reboot_cmd() 60
ccb_queue_reset_cmd() 57
ccb_queue_shutdown_cmd() 60
ccb_queue_standby_cmd() 58
ccb_queue_start_scan_cmd() 54
ccb_queue_status_request_cmd() . . . 59
ccb_queue_stop_scan_cmd() 55
ccb_queue_telemetry_cmd() 56
ccb_queue_ping_cmd() 59
ccb_scale_interval() 32
ccb_scan_duration() 38
ccb_server_communicate() 82
ccb_server_select_args() 81
ccb_set_cal_diode_cnf() 26
ccb_set_config() 22
ccb_set_phase_switch_cnf() 23
ccb_set_timing_cnf() 28
ccb_subtract_intervals() 33
ccb_time_to_timestamp() 40
ccb_time_until() 40
ccb_zero_interval() 33
del_CCBClientLink() 47
del_CCBConfig() 21
del_CCBServerLink() 81
new_CCBClientLink() 45
new_CCBConfig() 21
new_CCBServerLink() 75

Datatype index

CCBAwakenCmd 95
CCBCalDiodeCnf 28
CCBCalDiodeCmd 93
CCBCalDiodes 27
CCBClientIOStatus 51

CCBClientLink 45
CCBClientSocketsFn 50
CCBCmdErrorFn 52
CCBCmdStatus 52
CCBCntrlCmdAck 98
CCBConfig 21
CCBConfigType 22
CCBControlReply 97
CCBControlReplyType 96
CCBControlCommand 92
CCBControlCommandHeader 91
CCBControlCommandType 91
CCBControlLine 78
CCBCntrlPingReply 97
CCBStatusReply 97
CCBStatusReplyFn 61
CCBDriverCmd 77
CCBDriverCmdID 77
CCBDriverLoadFn 76
CCBDriverTellFn 77
CCBDriverUnloadFn 76
CCBDrvConfScan 79
CCBDrvInterrupt 78
CCBDrvIntraScan 80
CCBDrvIntrMask 78
CCBDrvLineMask 79
CCBDrvStageScan 80
CCBGeneralStatus 61
CCBIntegMsg 99
CCBIntegMsgFn 65
CCBInterval 32
CCBLinkType 59
CCBLoggerCmd 94
CCBLogMsg 100
CCBLogLevel 66
CCBLogMsgFn 66
CCBMonitorCmd 94
CCBMonitorMsg 99
CCBMonitorMsgFn 64
CCBNetMsg 87
CCBNetMsgInfo 89
CCBNetMsgMember 88
CCBPhaseSwitchCnf 25
CCBPhaseSwitchCmd 92
CCBRebootCmd 96
CCBResetCmd 95
CCBServerLink 75
CCBShutdownCmd 96
CCBStandbyCmd 95
CCBStartScanCmd 93
CCBStatusRequestCmd 96
CCBStopScanCmd 93
CCBTelemetryCmd 94
CCBTelemetryHeader 98
CCBTelemetryStream 56
CCBTelemetryType 98
CCBTelemPingReply 100
CCBPingCmd 95
CCBTimeStamp 38
CCBTimingCnf 30
CCBTimingCmd 93

Macro index

CCB_BASE_LOGID 83
CCB_CLIENT_SOCKETS_FN 50
CCB_CMD_ERROR_FN 52
CCB_CONTROL_PORT 10
CCB_STATUS_REPLY_FN 61
CCB_DRIVER_LOAD_FN 76
CCB_DRIVER_TELL_FN 77
CCB_DRIVER_UNLOAD_FN 76
CCB_INTEG_MSG_FN 65
CCB_LOGID 83
CCB_LOG_MSG_FN 66
CCB_LOG_PURGE_DT 63
CCB_MAX_INTEG 99
CCB_MAX_LOG 100
CCB_MAX_MONITOR 99
CCB_MAX_NCAL 26
CCB_MAX_LOG_VARIANTS 63
CCB_MONITOR_MSG_FN 64
CCB_TELEMETRY_PORT 10