

The network interface between the Ygor manager and the CCB server

[Document number: A48001N002, revision 11]

Martin Shepherd
California Institute of Technology

May 25, 2005

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	9
1.1	A guide to reading this manual	9
1.1.1	Points of interest to writers of the CCB Manager program	9
1.1.2	Points of interest to writers of the CCB server program	10
1.1.3	The organization of this manual	10
1.2	The two TCP/IP links used by the library	11
1.3	Connection establishment	11
1.4	Connection authentication	12
1.5	Initial configuration	12
1.6	Single threaded versus multi-threaded	12
1.7	Library usage caveats	13
1.8	Shared libraries and their versioning	14
2	Installation	16
2.1	Getting the source code	16
2.2	The basics of installation	16
2.3	Compiling in a different directory	17
2.4	Specifying where files are installed	17
2.5	Generating this manual	18
2.6	Testing the libraries using the demonstration programs	18
2.6.1	ccb_dummy_client	20
2.7	Run-time configuration files	21
2.8	The ccb_authorized_ips configuration file	21
3	The common parts of the CCB server and client APIs	22
3.1	The configuration of the CCB	22

3.1.1	The configuration of the phase switches	24
3.1.2	The configuration of the calibration diodes	27
3.1.3	The configuration of hardware timing parameters	29
3.1.4	The configuration of sampler control parameters	32
3.2	Integration and scan timing information	35
3.2.1	Interval computations	35
	ccb_scale_interval()	35
	ccb_add_intervals()	36
	ccb_subtract_interval()	36
	ccb_compare_intervals()	36
	ccb_zero_interval()	37
	ccb_interval_is_zero()	37
	ccb_clock_interval()	37
3.2.2	Cal-diode and phase-switch settling times	37
3.2.3	The number of phase-switch states per cycle	38
3.2.4	The physical duration of an integration period	38
3.2.5	The effective integration time	39
3.2.6	The duration of a scan	39
3.2.7	The number of integrations that fit within a time interval	40
3.2.8	The number of integrations in a calibration cycle	40
3.3	Timestamps	41
3.3.1	Getting the current date and time	41
3.3.2	Comparing two timestamps	42
3.3.3	Computing the amount of time remaining until a given time	42
3.3.4	Adding a time-interval to a timestamp	42
3.3.5	Converting a time_t value to a CCBTimeStamp value	42
3.3.6	Getting the clock time from a timestamp	43
4	The CCB client communications API	44
4.1	Include files	46
4.2	The CCB-client communications library	47
4.3	Creating the client resources needed to talk to a CCB server	47
4.4	Connecting to a CCB server	48
4.5	Disconnecting from a CCB server	49

4.6	Deleting a redundant CCBClientLink object	49
4.7	Requesting non-blocking I/O	50
4.8	ccb_client_communicate() – Perform client socket I/O	50
4.9	Client I/O multiplexing	51
4.9.1	Using the select() system call	51
4.9.2	Using the poll() system call	52
4.9.3	Third party event handlers	52
4.9.4	Using threads to multiplex I/O	53
4.10	Registering a command-error callback function	54
4.11	Outgoing CCB commands	55
4.11.1	Outgoing CCB control commands	56
	ccb_queue_start_scan_cmd() – Queuing a start-scan command	56
	ccb_queue_stop_scan_cmd() – Queuing a stop-scan command	58
	ccb_queue_dump_scan_cmd() – Queuing a dump-scan command	59
	ccb_queue_load_driver_cmd() – Queuing a load_driver command	60
	ccb_queue_monitor_cmd() – Queuing a monitor command	60
	ccb_queue_telemetry_cmd() – Queuing a telemetry command	61
	ccb_queue_logger_cmd() – Queuing a logger command	61
	ccb_queue_reset_cmd() – Queuing a reset command	62
	ccb_queue_ping_cmd() – Queuing a ping command	62
	ccb_queue_status_request_cmd() – Queuing a status-request command	63
	ccb_queue_shutdown_cmd() – Queuing a shutdown command	63
	ccb_queue_reboot_cmd() – Queuing a reboot command	64
4.12	Incoming control-link replies	64
	ccb_status_reply_callback() – Routing status-request replies	65
4.13	Incoming telemetry messages	66
	ccb_monitor_msg_callback() – Routing telemetry monitor-data messages	68
	ccb_integ_msg_callback() – Routing telemetry integ-data messages	68
	ccb_log_msg_callback() – Routing telemetry log-message messages	70
4.14	A TCL wrapper around the CCB client API	71
5	The CCB server communications API	80
5.1	Include files	80
5.2	The CCB-server communications library	81

5.3	Creating the resources used to communicate with CCB managers	81
5.3.1	The CCB server's device-driver interface	82
5.4	Shutting down server communications	89
5.5	Server I/O multiplexing	89
5.6	Queuing replies to control commands	90
5.7	Queuing outgoing telemetry messages	90
5.7.1	Queuing outgoing monitor-data messages	90
5.7.2	Queuing outgoing integ-data messages	90
5.7.3	Queuing outgoing log-message messages	91
6	Library internals	92
6.1	The message translation layer	94
6.1.1	Message structure specification	94
6.1.2	Supported data-types within message structures	94
6.1.3	CCBNetMsg - The base-class of all messages	94
6.1.4	Some example message structures	95
6.1.5	CCBNetMsgMember – Message field descriptions	95
6.1.6	CCBNetMsgInfo – Individual message descriptions	96
6.2	The CCB interface layer	97
6.2.1	The message structures of outgoing control messages	98
	CCBPhaseSwitchCmd – The phase-switching configuration command	99
	CCBCalDiodeCmd – The calibration diode configuration command	100
	CCBTimingCmd – The acquisition-timing configuration command .	100
	CCBSamplerCmd – The sampler configuration command	100
	CCBStartScanCmd – The start-scan command	101
	CCBStopScanCmd – The stop-scan command	101
	CCBDumpScanCmd – The dump-scan command	101
	CCBMonitorCmd – The monitor command	102
	CCBTelemetryCmd – The telemetry command	102
	CCBLoggerCmd – The logger command	102
	CCBResetCmd – The reset command	102
	CCBPingCmd – The ping command	103
	CCBStatusRequestCmd – The status-request command	103
	CCBShutdownCmd – The shutdown command	103

CCBRebootCmd – The reboot command	103
CCBLoadDriverCmd – The load_driver command	104
6.2.2 The message structures of incoming control-link replies	104
CCBCntrlPingReply – A reply to a ping command	105
CCBStatusReply – A reply to a status-request command	105
CCBCntrlCmdAck – An acknowledgment to a control command	105
6.2.3 The message structures of incoming telemetry messages	105
CCBIntegMsg – Integration data messages	106
CCBMonitorMsg – Monitor data messages	107
CCBLogMsg – CCB log messages	107
CCBTelemPingReply – A reply to a ping command	108
6.3 Sending network messages	108
6.4 Receiving network messages	109
A Changes to the library in version 2.0	110
A.1 Changes to the shared parts of the client and server APIs	110
A.1.1 Added functions	110
A.1.2 Added datatypes	111
A.1.3 Added constants	112
A.1.4 Functions with modified APIs	112
A.1.5 Modified datatypes	113
A.1.6 Functions with modified behaviors	114
A.1.7 Removed functions	114
A.2 Changes to the client-specific parts of the public API	115
A.2.1 Added functions	115
A.2.2 Functions with modified APIs	115
A.2.3 Modified datatypes	115
A.2.4 Removed functions	116
A.3 Changes to the server-specific parts of the public API	116
A.3.1 Added functions	116
A.3.2 Functions with modified APIs	117
A.3.3 Added datatypes	117
A.3.4 Modified datatypes	118
A.3.5 Removed functions	118

A.3.6	Removed Datatypes	118
A.4	Changes to the demonstration programs	119
A.4.1	The evolution of ccb_demo_server into ccbserver	119
A.4.2	Changes to the features of ccb_demo_client	119
B	Changes to the library in version 3.0	121
B.0.3	Modified datatypes	121
B.0.4	Functions with changed APIs	121
B.0.5	Functions with changed behaviors	121

List of Figures

3.1	Example phase-switching cycles	25
3.2	The anatomy of a data-scan or intra-scan	34
6.1	The CCB communications stack	93

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. Note that any changes to the documented API, that have been made between releases of the library, are summarized in Appendix A.

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 44

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 77

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

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

1.1.2 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 libraries use to exchange messages with each other. This is probably only of interest to the maintainer of these libraries.

After the final chapter, an appendix summarizes changes that have been made to the documented API of the library in new releases.

Following the appendix, 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 a one-way link, entirely under the control of the CCB server. The manager does, however, tell the server what classes of data it expects to receive over this link, and at what frequency.

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 reported to the existing manager as a log message.

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, and disables all telemetry except log messages. The manager then has the option of overriding the server's default configuration parameters with its own, before sending a CCB telemetry command to enable the telemetry that it wants to receive.

1.6 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 is reading from the telemetry socket.
- The library uses no modifiable static data. All modifiable data structures are allocated from the heap.
- The library uses POSIX thread-safe interfaces where available and uses POSIX thread calls to control multi-threaded access to heap-data and other shared resources.

- Within the communications library, before a thread calls any of the manager's callback functions, it first releases any locks that it is holding. It then reacquires those locks when the callback returns. This avoids deadlocks that would otherwise result if a callback were to call another function in the library.

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

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

The server side 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. The library header-files provide macros for declaring and prototyping these functions. For example, a typical callback definition macro might be the following,

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

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

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

and its definition written like,

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

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

1.8 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 L^AT_EX 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`, `xpdf` or `gpdf`.

If when the `configure` was script run, the optional L^AT_EX `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 a simple, GUI-based, demonstration client program, that can be used for testing, and the CCB server program.

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

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

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

Having installed the CCB software, check that the `ccb_authorized_ips` file, which is by-default installed in the `etc/` subdirectory of the chosen top-level installation directory, contains an entry that covers the IP address of the computer on which you will be running the demonstration client (see page 21).

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

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

and in the other terminal window type:

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

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

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

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

By default, the demonstration client addresses a virtual CCB driver within the `ccbserver` program. This driver simulates the CCB hardware, and can thus be used for off-line testing of the manager program. While this driver is being used, the CCB server prints what it is doing, on its parent terminal, indicating any effects that commands sent by the demonstration client (or a real manager), would have on the real hardware. The simulation driver also sends back faked integration and monitoring data, timed and flagged according to the current CCB configuration parameters.

The user of the demonstration client can also ask the CCB server to switch to addressing the real CCB driver, simply by selecting the *Normal* entry of the *Load Driver* option-menu. At the moment, this results in error messages being generated, since the real CCB driver is currently no more than a set of unimplemented stub-functions, but as these functions become implemented, the demonstration client will thus provide a way to test them, and the CCB hardware that they control.

2.6.1 `ccb_dummy_client`

In addition to the interactive demonstration client that was introduced above, a non-interactive demonstration client called `ccb_dummy_client` is provided. This initiates a non-blocking connection to the demonstration server, queues a full set of configuration and operating commands to be sent to the server, once the connection has been established, then enters a `select()` driven event loop. The event loop then informs the communications library whenever it detects an I/O event on any of the sockets that the library tells it to watch. Once the library receives confirmation of the completion of the non-blocking control connection, it sends the previously queued commands using non-blocking I/O, and watches for replies from

the server. The library then forwards replies from the server to the demonstration client, by calling the callback functions that the demonstration program provided it. For the sake of demonstration, these callbacks display the contents of the replies on the terminal. This minimal program, which was written before the interactive demo program, is potentially useful for speed tests, since the GUI display of 1ms integration updates would be too fast to follow by eye, even if one were confident that the X server could keep up with this rate. For example, using this program with the demo server, one can verify that the demo server can't generate integrations more frequently than 10ms. This is expected, because that is the granularity of the Linux timers that govern the event loop of the virtual driver in the CCB server program.

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

2.7 Run-time configuration files

Currently, the communications library 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 17.

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 `CCBConfig` objects. Since these objects are opaque, external functions must be used both to allocate them, and to modify and query their contents. This section describes these functions.

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

3.1.1 The configuration of the phase switches

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

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

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

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

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

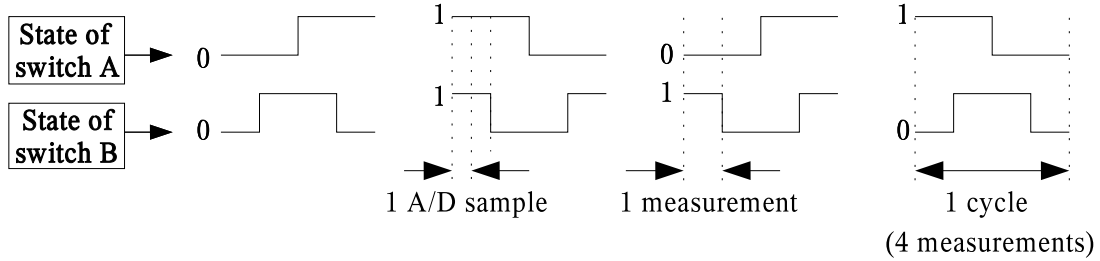
The arguments of this function are interpreted as follows.

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

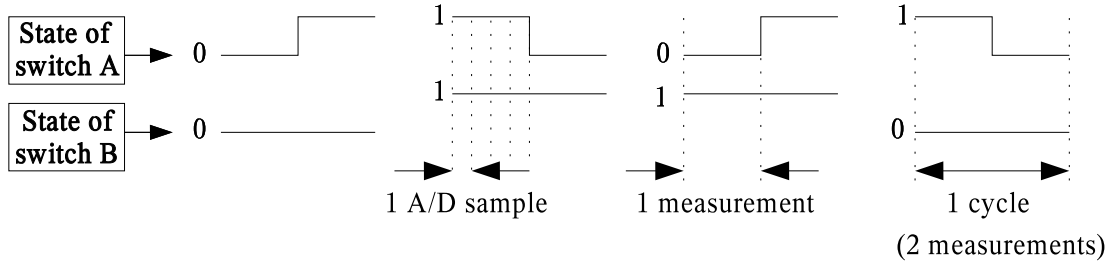
This specifies the set of phase switches that are to be switched during phase-switching cycles, expressed as a bitwise union of `CCBPhaseSwitches` enumerators.

```
typedef enum {
    CCB_NO_PHASE_SWITCHES = 0, /* Neither of the phase switches */
```

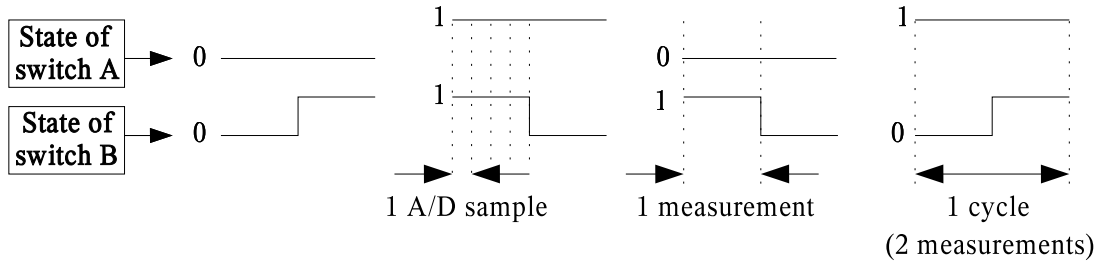
The 4 possible phase-switch cycles with both phase switches switching



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

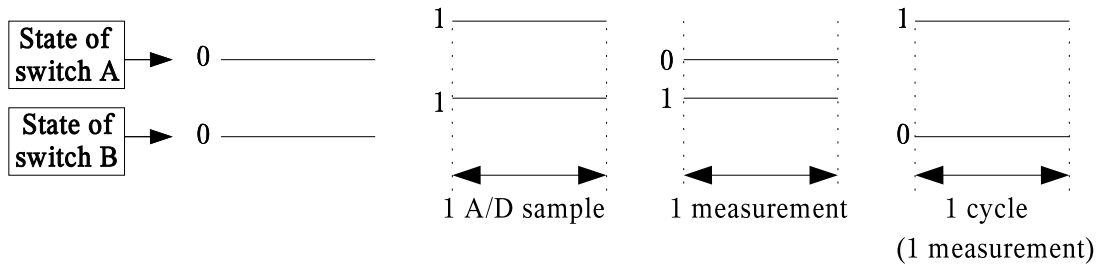


Figure 3.1: Example phase-switching cycles

```

    CCB_PHASE_SWITCH_A = 1,      /* phase switch A */
    CCB_PHASE_SWITCH_B = 2,      /* phase switch B */
    CCB_ALL_PHASE_SWITCHES =     /* Both phase switches */
        CCB_PHASE_SWITCH_A | CCB_PHASE_SWITCH_B
} CCBPhaseSwitches;

```

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

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

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

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

This parameter configures the number of A/D samples that are integrated between changes in the states of either phase-switch. The minimum supported value is given by the `CCB_MIN_SAMP_PER_STATE` macro, which is defined, in `ccbconstants.h`, as follows:

```
#define CCB_MIN_SAMP_PER_STATE 250 /* 25us */
```

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

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

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

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

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

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

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

3.1.2 The configuration of the calibration diodes

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

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

The arguments of this function, are as follows.

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

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

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

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

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

```

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

```

This argument can be NULL if `ncal` is 0.

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

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

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

This argument can be NULL if `ncal` is 0.

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

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

```

int ccb_get_cal_diode_cnf(const CCBCConfig *cnf,
                        CCBCalDiodeCnf *pars);

```

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

```

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

```

```

                                /* 'ncal' steps. */
unsigned long diode_times[CCB_MAX_NCAL]; /* The number of */
                                /* integrations of each */
                                /* of the 'ncal' steps. */
} CCBCalDiodeCnf;

```

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

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

3.1.3 The configuration of hardware timing parameters

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

```

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

```

The arguments of this function are interpreted as follows.

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

This specifies how much of the sample interval is blanked, while the phase switches settle after phase-switch transitions. It is expressed as an integer multiplier of 100ns.

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

This specifies the interval during which the calibration diode signals are unstable after being newly switched on. It is expressed as an integer multiplier of 100ns.

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

This specifies the interval during which any residual calibration diode signals remain present after being newly switched off. It is expressed as an integer multiplier of 100ns.

- **The integration period (integ_period)**

This specifies the number of phase-switch cycles that are to be co-added to form the integrations that are sent to the manager. The physical length of time that this corresponds to depends on the number of samples per phase-switch cycle, and can be calculated by calling the `ccb_integration_duration()` function.

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

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

```
#define CCB_MAX_ROUNDTRIP_DT 255    /* 8 bits */
```

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

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

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

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

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

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

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

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

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

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

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

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

<http://www.linuxdevices.com/articles/AT7751365763.html>

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

The clock signal that is used to clock the ADCs is a phase-shifted copy of the global FPGA clock. The phase shift is implemented by delaying a copy of the global FPGA clock by `adc_delay_dt`×10ns, modulo the 100ns clock period.

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 phase_switch_dt; /* The settling time of the phase */
                                   /* switches. */
    unsigned long diode_rise_dt; /* The rise time of a cal diode */
    unsigned long diode_fall_dt; /* The fall time of a cal diode */
    unsigned long integ_period; /* The integration period */
    unsigned short roundtrip_dt; /* The delay between toggling a */
                                   /* receiver control line and */
                                   /* the first effects reaching */
                                   /* the CCB integrators. */
    unsigned short holdoff_dt; /* The minimum time between */
                                   /* generating CPU interrupts */
                                   /* is 25.6us*(holdoff_dt+1) */
    unsigned short adc_delay_dt; /* The offset of the ADC clock */
                                   /* from the main FPGA clock, */
                                   /* expressed as a multiple of */
                                   /* 10ns. */
} CCBTimingCnf;
```

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

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

3.1.4 The configuration of sampler control parameters

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

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

The arguments of this function are interpreted as follows.

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

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

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

The default configuration sets this parameter to `CCB_ADC_SAMPLES`.

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

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

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

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

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

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

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

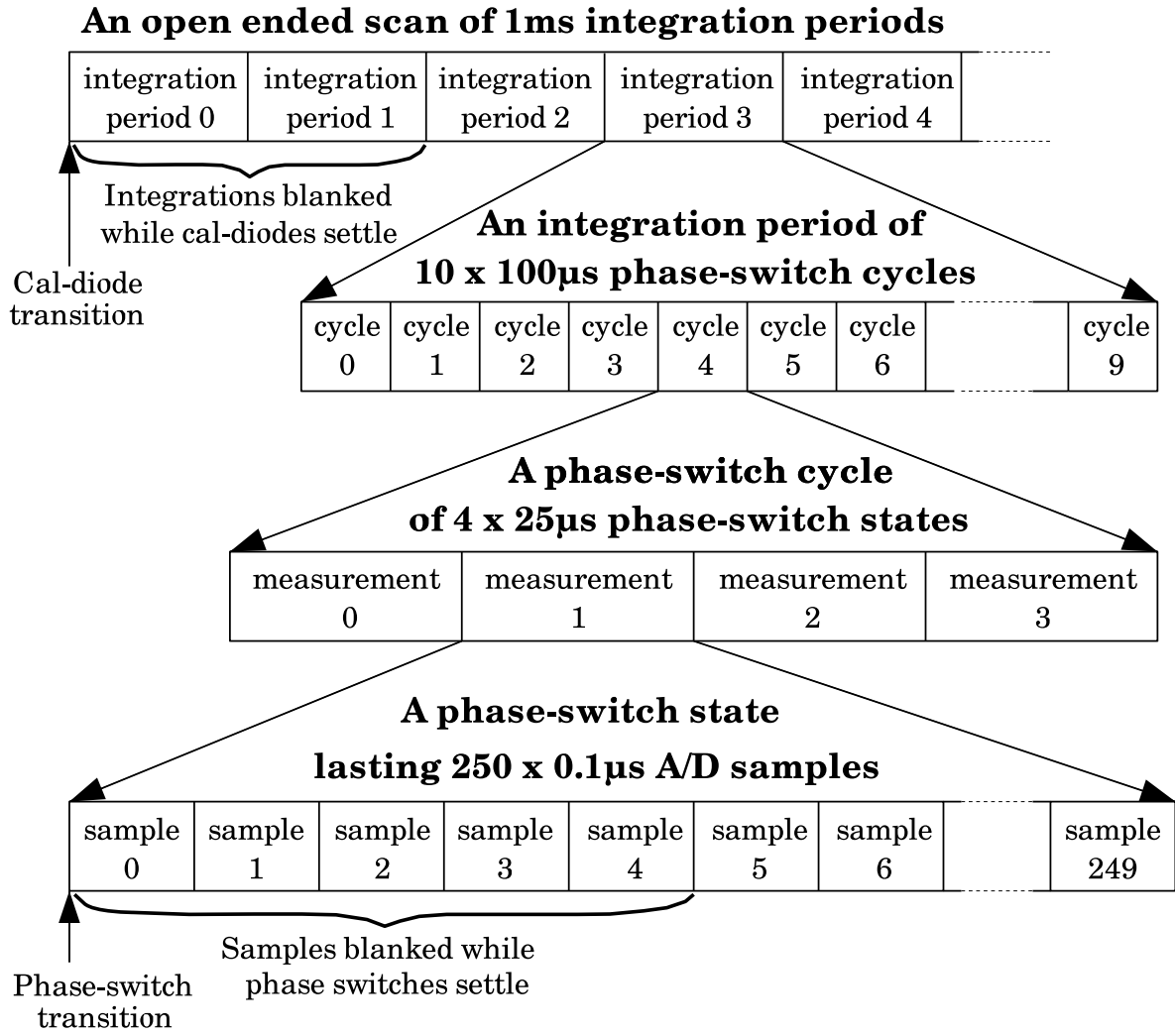


Figure 3.2: The anatomy of a data-scan or intra-scan

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, and because ADC samples are blanked during phase-switch transitions.

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_interval_is_zero()`

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

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

`ccb_clock_interval()`

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

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

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

3.2.2 Cal-diode and phase-switch settling times

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

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

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

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

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

3.2.3 The number of phase-switch states per cycle

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

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

Based on this equation, the `ccb_cycle_length()` function returns the number of phase-switch states per cycle, corresponding to a particular value of the `active_switches` argument of `ccb_set_phase_switch_cnf()` (see page 24).

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

3.2.4 The physical duration of an integration period

The number of ADC samples per integration is the product of three terms; the number of phase-switch cycles per integration (`integ_period`), the number of phase-switch states per

phase-switch cycle, as returned by `ccb_cycle_length()`, and the number of A/D samples per phase-switch state, given by the `samp_per_state` configuration parameter.

The `ccb_integration_duration()` function performs this calculation.

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

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

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

3.2.5 The effective integration time

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

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

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

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

3.2.6 The duration of a scan

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

```
int ccb_scan_duration(const CCBConfig *cnf,
                    unsigned long integrations,
                    CCBInterval *scan_dt);
```

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

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

3.2.7 The number of integrations that fit within a time interval

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

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

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

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

3.2.8 The number of integrations in a calibration cycle

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

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

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

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

3.3 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 of seconds into the day */
    unsigned long ns;     /* The number of nano-seconds into */
                        /* the specified second. */
} CCBTimeStamp;
```

The members of this structure are interpreted as follows.

- **mjd**

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

- **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 a time-interval of zero if the time has already passed.

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

The time of the event of interest is passed in the `ts` argument, and the amount of time remaining before that time is passed is returned in the variable pointed at by the `dt` argument. The returned interval is obviously out of date as soon as it is returned, and is thus of limited use. It was written for the CCB simulator incorporated in the `ccbserver` 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. Load the CCB device driver

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

7. Set up the initial CCB configuration:

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

8. Enable all telemetry streams

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

9. Queue the first start-scan command:

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

10. Now invoke the manager's event loop:

```
while(!shutdown_requested) {
    fd_set rfds; /* The set of file-descriptors to watch */
                /* for readability */
    fd_set wfds; /* The set of file-descriptors to watch */
                /* for writability */
    int nready; /* The number of file descriptors ready */
                /* for I/O. */
    int maxfd; /* The maximum of the file descriptors in */
                /* 'rfds' and 'wfds'. */

    /*
     * Clear the file-descriptor sets.
     */
    FD_ZERO(&rfds);
    FD_ZERO(&wfds);
    maxfd = 0;

    /*
     * Add the file-descriptors that the library wants us to watch,
     * to rfds and wfds.
     */
    if(ccb_client_select_args(cl, &rfds, &wfds, &maxfd))
        return ERROR;

    /*
     * Wait indefinitely for the specified I/O events.
     */
    nready = select(maxfd+1, &rfds, &wfds, NULL, NULL);
```

```

    /*
    * Error?
    */
    if(nready < 1)
        return ERROR;
/*
* Perform the types of I/O indicated by select().
*/
    else if(ccb_client_communicate(cl,
        ccb_client_selected_io(cl, &rfd, &wfd)))
        return 1;
}

```

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

```
ccb_client_disconnect(cl);
```

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

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

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

4.1 Include files

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

- `ccbclientlink.h`

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

- `ccbcommon.h`

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

- `ccbconstants.h`

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

4.2 The CCB-client communications library

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

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

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

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

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

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

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

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

If successful, this function allocates and returns a pointer to an opaque `CCBClientLink` object. This object should thereafter be passed to all other CCB communications-library functions. Note that this function does not itself initiate a connection to a CCB server. That is the role of the `ccb_client_connect()` function, which will be described below.

On error `new_CCBClientLink()` returns `NULL`, and places an error message in the buffer that is pointed to by the `errmsg` argument. The allocated size of this buffer must be specified in the `errdim` argument, such that if the length of an error message exceeds `errdim-1` characters,

it can be truncated to fit. If truncation is necessary, and `errdim` is greater than 0, a `'\0'` terminator is placed in `errmsg[errdim-1]`.

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

4.4 Connecting to a CCB server

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

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

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

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

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

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

Alternatively, if the `nonblocking` argument is non-zero, `ccb_client_connect()` may not get any further than initiating the connections before returning. Watching for the completion of the connections is left to subsequent calls to `ccb_client_communicate()`, in response to I/O activity detected by the manager's event loop.

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

numeric IP address in the `host` argument.

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

4.5 Disconnecting from a CCB server

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

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

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

4.6 Deleting a redundant CCBClientLink object

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

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

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

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

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

4.7 Requesting non-blocking I/O

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

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

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

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

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

4.8 `ccb_client_communicate()` – Perform client socket I/O

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

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

When non-blocking I/O is selected, the `io` argument, which tells `ccb_client_communicate()` which forms of I/O to attempt, must contain a value returned by either `ccb_selected_io()` or `ccb_polled_io()`, which are documented below. Alternatively, when blocking I/O has been selected, and `ccb_client_communicate()` is being called separately by different threads, the value of the `io` argument within a given thread must be chosen according to the type of I/O that that thread has been given responsibility for, as described shortly.

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

4.9 Client I/O multiplexing

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

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

4.9.1 Using the `select()` system call

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

```
int ccb_client_select_args(CCBClientLink *cl, fd_set *rfd,
                          fd_set *wfd, int *maxfd);
```

On input, the `rfd` argument is a pointer to an existing set of the file descriptors that the caller wants `select()` to watch for readability. Similarly the `wfd` 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 `rfd` and `wfd`. On return, `rfd` and `wfd` 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 *rfd,
                               const fd_set *wfd);
```

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

4.9.2 Using the poll() system call

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

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

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

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

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

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

4.9.3 Third party event handlers

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

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

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

```
typedef CCB_CLIENT_SOCKETS_FN(CCBClientSocketsFn);
```

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

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

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

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

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

4.9.4 Using threads to multiplex I/O

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

- `CCB_CNTRL_READ`

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

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

4.10 Registering a command-error callback function

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

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

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

typedef CCB_CMD_ERROR_FN(CCBCmdErrorFn);

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

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

```
typedef enum {
    CCB_CMD_ACCEPTED,    /* This enumerator isn't forwarded to error */
```



```

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

```

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

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

4.11 Outgoing CCB commands

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

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

4.11.1 Outgoing CCB control commands

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

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

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

Name	Description
<code>start-scan</code>	Start a new scan at the end of the current integration
<code>stop-scan</code>	Start a new intra-scan ASAP
<code>dump-scan</code>	Start a new dump-mode scan ASAP
<code>load_driver</code>	Load either the real or the simulation device-driver.
<code>reset</code>	Re-initialize the hardware.
<code>ping</code>	Request a link-verification reply.
<code>status-request</code>	Request a CCB-status report.
<code>shutdown</code>	Shutdown the real-time CPU.
<code>reboot</code>	Reboot the real-time CPU.
<code>monitor</code>	Configure monitoring.
<code>telemetry</code>	Configure the telemetry streams.
<code>logger</code>	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-second boundary that is specified by the `mjd` and `tod` parameters. The accuracy with which the CCB can start the scan at the specified time, depends on how far in advance the `start-scan` command is sent. There are three possibilities:

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

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

2. Less ideally, if the `start-scan` command arrives less than or equal to one second before

the requested start time, then the device-driver in the computer will wait for the next 1-second interrupt, before telling the hardware about the command. It will then tell the hardware to start the new scan as soon as possible, without synchronizing with a 1-second tick.

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

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

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

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

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

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

The arguments of this function are interpreted as follows.

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

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

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

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

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

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

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

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

`ccb_queue_stop_scan_cmd()` – **Queuing a stop-scan command**

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

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

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

```
int ccb_queue_stop_scan_cmd(CCBClientLink *cl, long id,
                           CCBCConfig *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_CCBCConfig()`, with any changes from the default configuration having been established by calling the `ccb_set_*_cnf()` functions described earlier in this document.

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

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

ccb_queue_dump_scan_cmd() – Queuing a dump-scan command

This operates like the `stop-scan` command, except that instead of the integrated data of the resulting intra-scan being sent back to the manager, the raw 100ns samples of a specified ADC are written to a named-pipe on the real-time computer. Since the bandwidth of the USB link between the master FPGA and the real-time computer isn't high enough to sustain continuous readback of ADC samples, only a specified number of samples are collected from the start of each integration.

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

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

The arguments of this function are interpreted as follows.

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

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

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

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

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

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

```
#define CCB_NUM_ADC 16
```

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

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

```
#define CCB_DUMP_ALL_SAMPLES 0
```

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

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

At the start of each new integration of a dump-scan, data are collected and sent to the CCB computer. This argument specifies how many of these per-integration data-frames are to be written to the named pipe on the CCB computer, for processing or archiving by an external program. A value of zero, as parameterized by the macro `CCB_DUMP_ALL_FRAMES`, specifies that data-frames should be delivered indefinitely.

```
#define CCB_DUMP_ALL_FRAMES 0
```

`ccb_queue_load_driver_cmd()` – **Queuing a load_driver command**

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

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

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

`ccb_queue_monitor_cmd()` – **Queuing a monitor command**

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

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

The arguments of this function are interpreted as follows.

- **The monitoring period (period)**

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

`ccb_queue_telemetry_cmd()` – **Queuing a telemetry command**

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

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

The arguments of this function are interpreted as follows.

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

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

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

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

`ccb_queue_logger_cmd()` – **Queuing a logger command**

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

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

The arguments of this function are interpreted as follows.

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

As discussed later (see page 66), a record of historically sent log messages is used to prevent repeated messages from being sent to the manager. The `period` argument specifies how often this historical record should be purged, expressed as an integer number of seconds, and thus the minimum time between repeated messages being queued to be sent to the manager.

`ccb_queue_reset_cmd()` – **Queuing a reset command**

When a manager first connects to the CCB, the server resets both itself, the CCB device-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 turns off all telemetry except log messages, and starts a dummy initial intra-scan with a scan ID of 0.

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

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

`ccb_queue_ping_cmd()` – **Queuing a ping command**

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

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

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

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

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


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

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

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

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

`ccb_queue_status_request_cmd()` – Queuing a status-request command

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

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

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

`ccb_queue_shutdown_cmd()` – Queuing a shutdown command

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

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

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

`ccb_queue_reboot_cmd()` – Queuing a reboot command

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

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

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

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
<code>cntrl-ping-reply</code>	A reply to a <code>ping</code> command.
<code>status-reply</code>	A reply to a <code>status-request</code> command.

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

CCBClientLink object that received the message, plus any arguments corresponding to the contents of the message.

`ccb_status_reply_callback()` – Routing status-request replies

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

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

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

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

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

Beware that unless care is taken to subsequently recompile every component of the system (and update this documentation), none of the existing values in this enumeration should either be removed or have their values changed. If necessary, new enumerators can be appended with the next highest unused power-of-2 value, and to support this possibility all

software that uses these values should not assume anything about the values of currently undefined bits.

4.13 Incoming telemetry messages

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

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

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

Name	Description
<code>monitor-data</code>	Instrumental monitoring data
<code>integ-data</code>	Integrated radiometer data
<code>log-message</code>	CCB log messages
<code>telem-ping-reply</code>	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
<code>monitor-data</code>	0	1 message	Overwrite the previous unsent message
<code>integ-data</code>	1	3MB (≥ 10 s)	Allow the queue to drain
<code>log-message</code>	2	100 messages	Overwrite the oldest unsent message
<code>telem-ping-reply</code>	3	1 message	Overwrite the previous unsent message

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

`log-message` messages have the second highest priority, to prevent important messages from being held up indefinitely. The queuing strategy for log messages is complicated by the

need to prevent rapidly repeating messages from consuming too much memory and bandwidth. Detecting repeating messages is further complicated by the fact that a given message-reporting statement can include changeable content in its messages, such as IP addresses, `errno` information and problematic values. The logging strategy adopted by the CCB server is thus as follows. In addition to a fixed size queue of outgoing log messages, the server maintains a periodically purged table containing the checksums of recently generated log messages. The table of checksums records the checksums of up to `CCB_MAX_LOG_VARIANTS` different messages for each logging statement. Before appending a message to the queue of outgoing log messages, the CCB server first checks to see if, since the last time that this checksum-table was cleared, the originating statement has either already reported the same message, or has generated an excessive number of varying messages. The message is not queued if either of these conditions are true.

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

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

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

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

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

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

ccb_monitor_msg_callback() – Routing telemetry monitor-data messages

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

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

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

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

typedef CCB_INTEG_MSG_FN(CCBIntegMsgFn);

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

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

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

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

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

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

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

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

```
typedef enum {
    CCB_CAL_A_ON = 1,    /* Included if cal-diode A was on */
    CCB_CAL_B_ON = 2,    /* Included if cal-diode B was on */
    CCB_INTEG_OK = 4,    /* Included if the integration is usable */
    CCB_SLAVE0_OK = 8,   /* Included if slave-FPGA 0 was present */
    CCB_SLAVE1_OK = 16,  /* Included if slave-FPGA 1 was present */
```

```

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

```

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

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

The first `nvalues` elements of the array pointed to by the `values[]` argument, contain the radiometer integrations.

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

This argument specifies the number of significant elements at the start of the `values` array.

`ccb_log_msg_callback()` – Routing telemetry log-message messages

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

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

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

```

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

```



```
typedef CCB_LOG_MSG_FN(CCBLogMsgFn);

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

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

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

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

4.14 A TCL wrapper around the CCB client API

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

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

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

- `ccb connect host`

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

- `ccb disconnect`

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

- `ccb send ...`

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

- `ccb send start_scan mjd seconds`

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

- `ccb send stop_scan`

This starts an intra-scan ASAP.

- `ccb send dump_scan adc samples frames`

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

- `ccb send load_driver type`

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

`normal virtual`

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

- `ccb send reset`

This resets the CCB.

- `ccb send ping`

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

- `ccb send status_request`

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

- `ccb send shutdown`

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

- `ccb send reboot`

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

- `ccb send monitor period`

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

- `ccb send telemetry streams`

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

```
integ_stream  monitor_stream  log_stream
no_streams   all_streams
```

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

- `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* *closed_switches*
samp_per_state

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

`switch_a` `switch_b` `no_switches` `all_switches`

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

The final, `samp_per_state` argument is expressed as an integer.

- `ccb configure cal_diode` *ncal* *diode_states* *diode_times*

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

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

`diode_a` `diode_b` `no_diodes` `all_diodes`

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

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

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

- `ccb configure timing` *phase_switch_dt* *diode_rise_dt* *diode_fall_dt*
integ_period *roundtrip_dt* *holdoff_dt* *adc_delay_dt*

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

- `ccb configure sampler` *sample_type*

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

`adc_samples` `fake_samples`

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

- `ccb attach ...`

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

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

- `ccb attach status command`

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

- `ccb attach monitor command`

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

- `ccb attach integ command`

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

- `ccb attach log command`

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

- `ccb attach error command`

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

- `ccb get ...`

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

The single argument of the `ccb get` command specifies what type of information is to be retrieved. The retrieved data is passed back as the result string of the command. For those not familiar with Tcl, the result string is written to `stdout` if the command

is typed in at the command-line of `wish` or `tclsh`, or it can be interpolated into an argument of another Tcl command by invoking it between square brackets.

The available information requests are as follows.

– `ccb get status`

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

```
link_down  buffer_full  hard_fault  soft_fault
```

These correspond to the similarly named `CCBGeneralStatus` enumerators (see page 65).

– `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. `flags` - This is a Tcl list of zero or more of the following integration status-flags:
`cal_a_on, cal_b_on, integ_ok, slave0_ok,`
`slave1_ok, slave2_ok, slave3_ok`

These flag-names have the same meanings as their similarly named `CCBIntegFlags` enumerator counterparts.

7. `nvalue` - The number of integrated values.
8. `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 71).

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 54). Note that since the CCB library only tells us the completion statuses of commands that fail, when this command returns the word `accepted`, this means that no command has failed yet.

– `ccb get time`

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

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

```

tclsh tcl_demo

# Load the CCB Tcl interface.

load libccbtclclient.so

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

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

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

ccb configure phase_switches all_switches all_switches 250

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

ccb configure timing 10 1000 1000 40000 10 7 0

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

ccb send stop_scan

# Connect to the CCB server on the local machine.

ccb connect localhost

# Load the hardware-simulating device driver.

ccb send load_driver virtual

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

```



```
ccb send telemetry all_streams
```

```
# Start the Tcl event loop.
```

```
set ::guard 0
```

```
vwait ::guard
```

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

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

The arguments of this function are interpreted as follows.

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

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

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

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

5.3.1 The CCB server's device-driver interface

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

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

The arguments of `new_CCBServerDriver()` are interpreted as follows.

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

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

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

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

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

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

```
void *data)
```

```
typedef CCB_DRIVER_LOAD_FN(CCBDriverLoadFn);
```

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

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

This argument specifies the function that the `CCBServerLink` object should call when it needs to unload the CCB device driver. Beware that this function may be called when no driver is currently loaded, and that this shouldn't be interpreted as an error. Suitable functions to pass in the `unload_driver` argument should be declared and prototyped using the `CCB_DRIVER_UNLOAD_FN()` macro. Pointers to them can be recorded in variables of type `CCBDriverUnloadFn`.

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

```
typedef CCB_DRIVER_UNLOAD_FN(CCBDriverUnloadFn);
```

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

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

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

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

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

```
typedef CCB_DRIVER_TELL_FN(CCBDriverTellFn);
```

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

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

```
typedef enum {
    CCB_DRV_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 */
    CCB_DRV_DUMP_SCAN /* Start a dump-mode intra-scan ASAP */
} CCBDriverCmdID;
```

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

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

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

– **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. */
    CCBSamplerCnf sampler;  /* The sampler */
                               /* configuration */
} CCBDrvConfScan;

```

The members of this structure are encapsulated configuration groupings of the types that are returned by the `ccb_get_*_cnf()` configuration lookup functions.

– Stage a new observing scan

When the server library receives a `start-scan` command, it uses the `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.

– Start a dump-mode intra-scan

When the server library receives a `dump-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 `dump-scan` driver-command to immediately initiate a dump-mode intra-scan. The parameters of this command are encapsulated within a structure of the following type.

```
typedef struct {
    unsigned long scan;      /* The numeric scan identifier */
    unsigned short adc;     /* The ADC to dump */
    unsigned long samples;  /* The max number of samples to */
                           /* collect per integration. */
    unsigned long frames;   /* The number of per-integration */
                           /* data-frames to deliver. */
} CCBDrvDumpScan;
```

The `adc` member should be an integer between 0 and `CCB_NUM_ADC-1`, specifying the ADC whose samples are to be collected. The `samples` member, should either be `CCB_DUMP_MAX_SAMPLES`, to specify that the maximum number of samples/integration be collected, or a positive integer specifying the desired number. The `frames` member should either be `CCB_DUMP_ALL_FRAMES`, to specify that per-integration data-frames be delivered continuously, or a positive integer specifying how many frames to deliver per dump-mode scan.

- **The callback that indicates events to watch (`select_events`)**

This argument specifies a function which is called each time just before the CCB server's event loop invokes `select()` to wait for I/O. The driver is expected to add any file-descriptors that it wants to have watched for activity, and/or register an inactivity-timeout.

Suitable functions to pass in the `select_events` argument should be declared and prototyped using the `CCB_DRIVER_SELECT_EVENTS_FN()` macro. Pointers to them can be recorded in variables of type `CCBDriverSelectEventsFn`.


```

#define CCB_DRIVER_SELECT_EVENTS_FN(fn) int (fn)( \
    CCBServerLink *sl, void *data, \
    fd_set *rfds, fd_set *wfds, int *maxfd, \
    struct timeval *timeout)

typedef CCB_DRIVER_SELECT_EVENTS_FN(CCBDriverSelectEventsFn);

```

When this function is called, it is passed the value of the `data` argument of `new_CCBServerLink()`. The callback should use the standard `FD_SET()` macro to install file-descriptors of interest in `*rfds` and `*wfds`, where the former is for descriptors to be watched for readability, and the latter is for those to be watched for writability. If the value of `*maxfd` is less than the maximum value of any descriptor that the callback has added to `rfds` or `wfds`, then it should be set to the latter maximum. If the driver wants to be called on inactivity-timeouts, then it should register the timeout that it wants in the `timeout` argument.

Normally the callback should return 0 to indicate success. If an unrecoverable error occurs, it should return 1, and set `errno` accordingly.

- **The callback that checks for driver events (`check_events`)**

This argument specifies a function which is called each time when the CCB server's call to `select()` returns. The driver is expected to handle any indicated file-descriptor activity, or inactivity-timeout.

Suitable functions to pass in the `check_events` argument should be declared and prototyped using the `CCB_DRIVER_CHECK_EVENTS_FN()` macro. Pointers to them can be recorded in variables of type `CCBDriverCheckEventsFn`.

```

#define CCB_DRIVER_CHECK_EVENTS_FN(fn) int (fn)( \
    CCBServerLink *sl, void *data, \
    fd_set *rfds, fd_set *wfds)
typedef CCB_DRIVER_CHECK_EVENTS_FN(CCBDriverCheckEventsFn);

```

When this function is called, it is passed the value of the `data` argument of `new_CCBServerLink()`. If the `select()` inactivity timeout was triggered, the `rfds` and `wfds` arguments are both `NULL`. Otherwise they contain the sets of file-descriptors that are now ready for reading and writing, respectively.

Normally the callback should return 0 to indicate success. If an unrecoverable error occurs, it should return 1, and set `errno` accordingly.

- **The callback that reboots the computer (`reboot_rtc`)**

This argument specifies the function that the `CCBServerLink` object should call to reboot the CCB computer.

Suitable functions to pass in the `reboot_rtc` argument should be declared and prototyped using the `CCB_REBOOT_RTC_FN()` macro. Pointers to them can be recorded in variables of type `CCBRebootRTCFn`.

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

typedef CCB_REBOOT_RTC_FN(CCBRebootRTCFn);
```

When this function is called, it is passed the value of the `data` argument of `new_CCBServerLink()`. When a reboot is successfully initiated, `reboot_rtc` callbacks should return 0. Otherwise they should return 1, and set `errno` accordingly.

- **The callback that shuts-down the computer (`shutdown_rtc`)**

This argument specifies the function that the `CCBServerLink` object should call to shutdown both the CCB computer and the CCB hardware.

Suitable functions to pass in the `shutdown_rtc` argument should be declared and prototyped using the `CCB_SHUTDOWN_RTC_FN()` macro. Pointers to them can be recorded in variables of type `CCBShutdownRTCFn`.

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

typedef CCB_SHUTDOWN_RTC_FN(CCBShutdownRTCFn);
```

When this function is called, it is passed the value of the `data` argument of `new_CCBServerLink()`. When a shutdown is successfully initiated, `shutdown_rtc` callbacks should return 0. Otherwise they should return 1, and set `errno` accordingly.

The return value of `new_CCBServerDriver()` is a pointer to an opaque `CCBServerDriver` object that can be passed to `new_CCBServerLink()`, or `NULL` if an error occurred.

Once a `CCBServerDriver` object has been passed to `new_CCBServerLink()`, it shouldn't be deleted until after `del_CCBServerLink()` is called to delete the corresponding server object. Objects returned by `new_CCBServerDriver()` are deleted by calling `del_CCBServerDriver()`.

```
CCBServerDriver *del_CCBServerDriver(CCBServerDriver *drv);
```

The argument of this function is the object to be deleted (which can be `NULL`). The return value of the function is always `NULL`, so that one can type:

```
drv = del_CCBServerDriver(drv);
```

This both deletes the object pointed to by `drv`, and then sets this pointer to `NULL`, to ensure that subsequent illegal attempts to access the object through this pointer produce a segmentation-fault, rather than appearing to work, while actually doing something bad.

5.4 Shutting down server communications

When the CCB server shuts down, it releases the resources that were allocated by `new_CCBServerLink()` and closes all of its sockets, by calling `del_CCBServerLink()`.

```
CCBServerLink *del_CCBServerLink(CCBServerLink *sl);
```

This function always returns `NULL` to allow the caller to type:

```
CCBServerLink *sl;  
...  
sl = del_CCBServerLink(sl);
```

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

5.5 Server I/O multiplexing

To enable the CCB server to handle the telemetry and control links at the same time as interacting with the CCB device driver, the server library uses non-blocking socket I/O when reading and writing messages. The driver is expected to do the same. The `select()`-based event-loop is invoked by the server by calling the `ccb_server_event_loop()` function.

```
int ccb_server_event_loop(CCBServerLink *ccb);
```

The `rfd`s and `wfd`s arguments are the file descriptor sets that `select()` returned. Normally `ccb_server_select_args()` returns 0, but if an error occurs, it returns 1 and sets `errno` accordingly.

5.6 Queuing replies to control commands

All replies to control commands are queued internally by the library, so there are no public API functions related to this.

5.7 Queuing outgoing telemetry messages

Messages to be sent to the manager over the telemetry link are placed in message-specific queues within the corresponding `CCBServerLink` object, as described in section 4.13. The `ccb_server_event_loop()` dispatches such messages to the manager. Whenever it finishes sending a message, it chooses a new message from the highest priority queue that contains at least one message, encodes this and starts to send the result to the manager.

The functions that the CCB server uses to queue messages in the appropriate queues, are documented in the following subsections.

5.7.1 Queuing outgoing monitor-data messages

The CCB server uses the `ccb_queue_monitor_msg()` function to queue monitor-data messages for later transmission.

```
int ccb_queue_monitor_msg(CCBServerLink *sl,
                        const CCBTimeStamp *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,
                      unsigned short flags,
```

```
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.

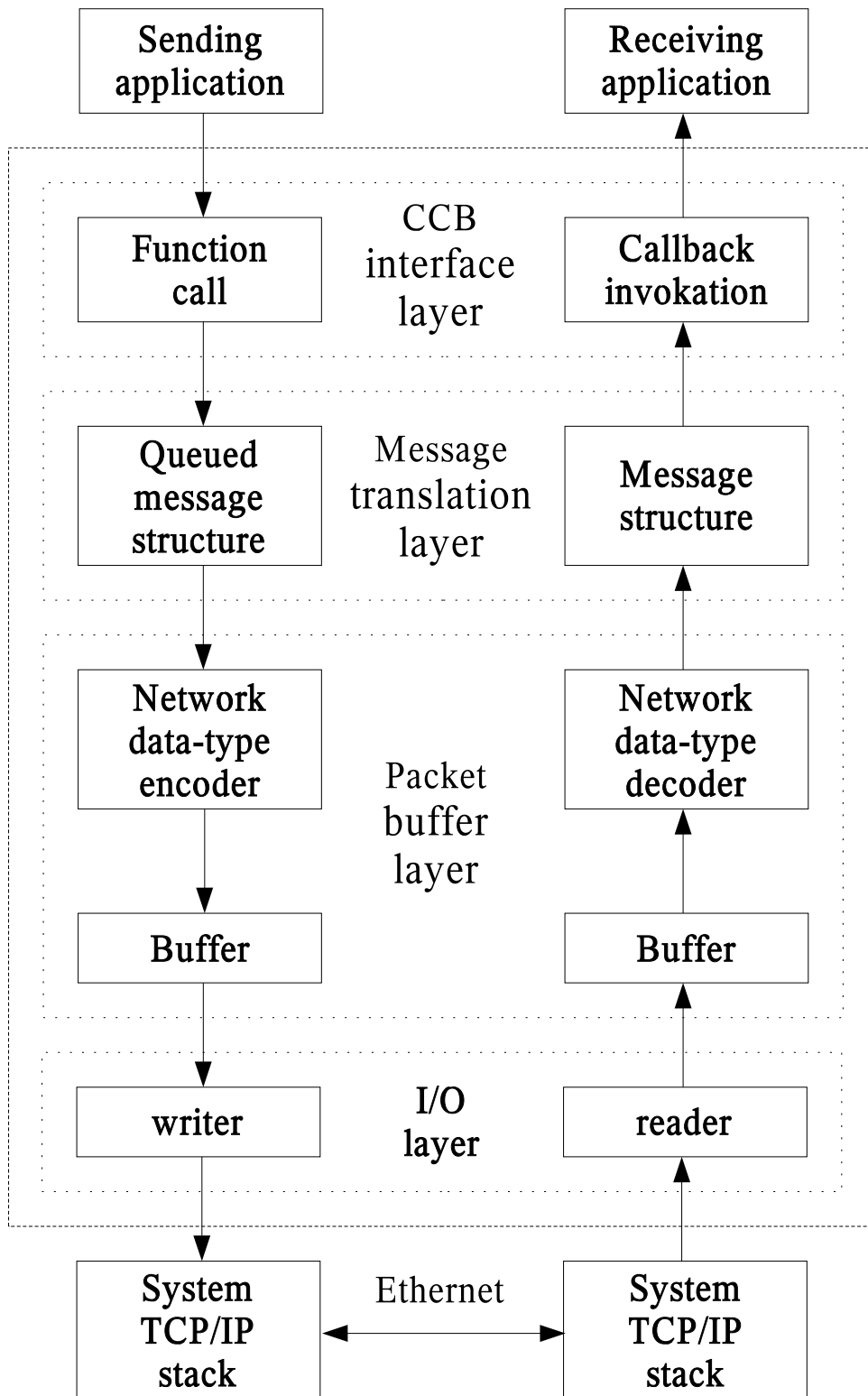


Figure 6.1: The CCB communications stack

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 transferred over the network in big-endian, 2's-complement format, and that the two floating point data-types are transferred 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 transferred 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 */
    CCBNetMsg 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 */
    CCBNetMsg base;    /* The message identification header */
    unsigned long foo; /* A <= 32-bit unsigned number */
} CCBExampleMsg2;

```

6.1.5 CCBNetMsgMember – Message field descriptions

With the exception of the obligatory initial CCBNetMsg member, each member of each message structure is described to the library using a CCBNetMsgMember structure.

```

typedef struct {
    const char *name; /* The textual name of the member */
    size_t offset; /* The byte-offset of the member in the */
                  /* local message structure */
    CCBNetDataType type; /* The enumerated data-type of the member */
    int ntype; /* The number of elements in the member */
} CCBNetMsgMember;

```

The following example code shows how arrays of these `CCBNetMsgMember` structures are used to describe the elements of the two example message structures.

```
#include <stddef.h>
#include "ccbnetobj.h"

/* The description of the members of CCBExampleMsg1 */

static const CCBNetMsgMember ccb_example_msg1_members[] = {
    {"string", offsetof(CCBExampleMsg1, string),      CCB_NET_ASCII, SDIM},
    {"slen",   offsetof(CCBExampleMsg1, slen),        CCB_NET_USHORT, 1},
};

/* The description of the members of CCBExampleMsg2 */

static const CCBNetMsgMember ccb_example_msg2_members[] = {
    {"foo",   offsetof(CCBExampleMsg2, foo),          CCB_NET_ULONG, 1},
};
```

6.1.6 CCBNetMsgInfo – Individual message descriptions

In addition to descriptions of the contents of each message type, the communications library needs to know both the host-dependent size of the message data-structures, and a convenient way for the various parts of the library to refer each other to a given type of message. Each message is thus further described using a `CCBNetMsgInfo` structures.

```
typedef struct {
    int type;                /* The message-type enumerator */
    const char *name;        /* The name of this message-type */
    const CCBNetMsgMember *member; /* Descriptions of each member */
    int nmember;            /* The number of elements in member[] */
    size_t native_size;     /* The host-dependent size of the */
                           /* the corresponding message structure */
} CCBNetMsgInfo;
```

The `name` field, which isn't currently used by the library, may in future be used when printing out the contents of messages for debugging purposes.

For a given network connection, the communications library needs separate descriptions of the messages that it is expected to transmit, and those that it is expected to receive. To do this the CCB interface layer registers two arrays of `CCBNetMsgInfo` structures per connection, one describing outgoing messages, while the other describes incoming messages.

The indexes of elements in these arrays are the means by which the various parts of the library, at both ends of the communications link, refer each other to a given message type. Since the index associated with a given message type will change if somebody inserts a new message type in the middle of a message-description array, the CCB interface layer assigns a copy of the enumerator that it uses to refer to each message type, to the `type` field of the corresponding `CCBNetMsgInfo` message-definition element. This allows the message-translation layer to verify that these enumerators match the array indexes of the messages to which they refer. Thereafter, whenever the CCB interface layer passes a message structure to the message-translation layer for transmission over the network, it sets the `type` member of the `CCBNetMsg` structure accordingly, to tell the message-translation layer what type of message structure it is being passed. Similarly, when the message-translation layer receives a message from the network, it records the type of message that it received, in the `type` member of the `CCBNetMsg` structure that it returns.

Returning to the example, the types of the example messages would be enumerated, and described in a message-definition array, as follows:

```
typedef enum {
    CCB_EXAMPLE_MSG1, /* The index of the first example message */
    CCB_EXAMPLE_MSG2 /* The index of the second example message */
} CCBExampleMsgTypes;

static const CCBNetMsgInfo ccb_example_messages[] = {
    {CCB_EXAMPLE_MSG1, "example1", ccb_example_msg1_members,
     NET_ARRAY_DIM(ccb_example_msg1_members), sizeof(CCBExampleMsg1)},
    {CCB_EXAMPLE_MSG2, "example2", ccb_example_msg2_members,
     NET_ARRAY_DIM(ccb_example_msg2_members), sizeof(CCBExampleMsg2)},
};
```

In this example `CCBExampleMsgTypes` associates symbolic names with the indexes of the correspondingly messages in the `ccb_example_messages[]` array, while the latter array provides the description of all messages for one direction of a communications link.

6.2 The CCB interface layer

For each of the message queuing and received-message callback functions in the public API, the CCB interface layer defines a message structure for passing the corresponding message to and from the message-translation layer of the library. The following sub-sections briefly describe these structures. Note that since these structures are hidden within the communications library, provided that the library is compiled as a shared library, the contents of the message structures can be rearranged without requiring a recompilation of the manager or

the CCB server.

6.2.1 The message structures of outgoing control messages

As previously mentioned, the message-translation layer requires that all messages being transmitted over a particular network connection be internally enumerated by the CCB-interface layer. This enumeration is used to communicate message types both between the CCB-interface and message-translation layers of the library, and between the separate message-translation layers at the two ends of the communications link. The `CCBControlCommandType` enumeration serves this role for outgoing messages on the control link.

```
typedef enum {
    CCB_PHASE_SWITCH_CMD,    /* A phase-switch config command */
    CCB_CAL_DIODE_CMD,      /* A cal-diode config command */
    CCB_TIMING_CMD,         /* An timing config command */
    CCB_SAMPLER_CMD,        /* An sampler config command */
    CCB_START_SCAN_CMD,     /* A start-scan command */
    CCB_STOP_SCAN_CMD,      /* A stop-scan command */
    CCB_DUMP_SCAN_CMD,      /* A dump-scan command */
    CCB_MONITOR_CMD,        /* A monitoring control command */
    CCB_TELEMETRY_CMD,      /* A telemetry stream control command */
    CCB_LOGGER_CMD          /* A log control command */
    CCB_RESET_CMD,          /* A reset command */
    CCB_PING_CMD,           /* a ping command */
    CCB_STATUS_REQUEST_CMD, /* a status-request command */
    CCB_SHUTDOWN_CMD,       /* A computer shutdown command */
    CCB_REBOOT_CMD,         /* A computer reboot command */
    CCB_LOAD_DRIVER_CMD     /* A device-driver loading command */
} CCBControlCommandType;
```

As documented on page 54, 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 `CCBNetMessage` 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 {
    CCBNetMessage 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 `CCBNetMessage` 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 `CCBNetMessage` object.

```

typedef union {
    CCBControlCommandHeader head; /* The common control message header */
    CCBPhaseSwitchCmd phase_cmd; /* head.base.type=CCB_PHASE_SWITCH_CMD */
    CCBCalDiodeCmd diode_cmd; /* head.base.type=CCB_CAL_DIODE_CMD */
    CCBTimingCmd timing_cmd; /* head.base.type=CCB_TIMING_CMD */
    CCBSamplerCmd sampler_cmd; /* head.base.type=CCB_SAMPLER_CMD */
    CCBStartScanCmd start_scan; /* head.base.type=CCB_START_SCAN_CMD */
    CCBStopScanCmd stop_scan; /* head.base.type=CCB_STOP_SCAN_CMD */
    CCBDumpScanCmd dump_scan; /* head.base.type=CCB_DUMP_SCAN_CMD */
    CCBMonitorCmd monitor; /* head.base.type=CCB_MONITOR_CMD */
    CCBTelemetryCmd telemetry; /* head.base.type=CCB_TELEMETRY_CMD */
    CCBLoggerCmd logger; /* head.base.type=CCB_LOGGER_CMD */
    CCBResetCmd reset; /* head.base.type=CCB_RESET_CMD */
    CCBPingCmd ping; /* head.base.type=CCB_PING_CMD */
    CCBStatusRequestCmd status; /* head.base.type=CCB_STATUS_REQUEST_CMD */
    CCBShutdownCmd shutdown; /* head.base.type=CCB_SHUTDOWN_CMD */
    CCBRebootCmd reboot; /* head.base.type=CCB_REBOOT_CMD */
    CCBLoadDriverCmd driver; /* head.base.type=CCB_LOAD_DRIVER_CMD */
} CCBControlCommand;

```

CCBPhaseSwitchCmd – The phase-switching configuration command

The `ccb_queue_start_start_cmd()` and `ccb_queue_stop_scan_cmd()` functions both queue message structures of the following type for transmission when the phase-switch parameters of the commanded scan differ from those of the previous scan.

```

typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_PHASE_SWITCH_CMD */
    CCBPhaseSwitchCnf cnf; /* The phase-switch configuration */
    /* parameters. */
} CCBPhaseSwitchCmd;

```

CCBCalDiodeCmd – The calibration diode configuration command

The `ccb_queue_start_scan_cmd()` and `ccb_queue_stop_scan_cmd()` functions both queue message structures of the following type for transmission when the cal-diode parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_CAL_DIODE_CMD*/
    CCBCalDiodeCnf cnf;          /* The cal-diode configuration */
                                /* parameters. */
} CCBCalDiodeCmd;
```

CCBTimingCmd – The acquisition-timing configuration command

The `ccb_queue_start_start_cmd()` and `ccb_queue_stop_scan_cmd()` functions both queue message structures of the following type for transmission when the hardware-timing parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_TIMING_CMD */
    CCBTimingCnf cnf;            /* The timing configuration */
                                /* parameters. */
} CCBTimingCmd;
```

CCBSamplerCmd – The sampler configuration command

The `ccb_queue_start_start_cmd()` and `ccb_queue_stop_scan_cmd()` functions both queue message structures of the following type for transmission when the hardware-sampler parameters of the commanded scan differ from those of the previous scan.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_SAMPLER_CMD */
    CCBSamplerCnf cnf;           /* The sampler configuration */
                                /* parameters. */
} CCBSamplerCmd;
```

CCBStartScanCmd – The start-scan command

The `ccb_queue_start_scan_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_START_SCAN_CMD */
    unsigned long scan;           /* The numeric ID to give the new */
                                /* scan. */
    unsigned long mjd;           /* The MJD UTC day number */
    unsigned long tod;           /* The time of day (seconds since */
                                /* 0H UTC). */
} CCBStartScanCmd;
```

CCBStopScanCmd – The stop-scan command

The `ccb_queue_stop_scan_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_STOP_SCAN_CMD */
    unsigned long scan;           /* The numeric ID to give the new */
                                /* intra-scan. */
} CCBStopScanCmd;
```

CCBDumpScanCmd – The dump-scan command

The `ccb_queue_dump_scan_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_DUMP_SCAN_CMD */
    unsigned long scan;           /* The numeric ID to give the new */
                                /* intra-scan. */
    unsigned short adc;          /* The ADC to dump */
    unsigned long samples;       /* The max number of samples to */
                                /* collect per integration. */
    unsigned long frames;        /* The number of per-integration */
                                /* data-frames to deliver. */
} CCBDumpScanCmd;
```

CCBMonitorCmd – The monitor command

The `ccb_queue_monitor_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_MONITOR_CMD */
    unsigned short period;        /* The interval between monitoring */
                                /* updates. */
} CCBMonitorCmd;
```

CCBTelemetryCmd – The telemetry command

The `ccb_queue_telemetry_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_TELEMETRY_CMD */
    unsigned short streams;       /* The telemetry-streams to report */
} CCBTelemetryCmd;
```

CCBLoggerCmd – The logger command

The `ccb_queue_logger_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_LOGGER_CMD */
    unsigned long period;         /* The interval at which */
                                /* historical log messages are */
                                /* forgotten (seconds). */
} CCBLoggerCmd;
```

CCBResetCmd – The reset command

The `ccb_queue_reset_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_RESET_CMD */
} CCBResetCmd;
```


CCBPingCmd – The ping command

The `ccb_queue_ping_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_PING_CMD */
} CCBPingCmd;
```

CCBStatusRequestCmd – The status-request command

The `ccb_queue_status_request_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type = */
                                   /* CCB_STATUS_REQUEST_CMD */
} CCBStatusRequestCmd;
```

CCBShutdownCmd – The shutdown command

The `ccb_queue_shutdown_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_SHUTDOWN_CMD */
} CCBShutdownCmd;
```

CCBRebootCmd – The reboot command

The `ccb_queue_reboot_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_REBOOT_CMD */
} CCBRebootCmd;
```

CCBLoadDriverCmd – The load_driver command

The `ccb_queue_load_driver_cmd()` function queues message structures of the following type, ready for later exchange with the message translation layer.

```
typedef struct {
    CCBControlCommandHeader head; /* head.base.type=CCB_LOAD_DRIVER_CMD */
    unsigned short type;          /* A CCBDriverType enumerator */
} CCBLoadDriverCmd;
```

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 {
    CCBCControlReplyHeader 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 {
    CCBCControlReplyHeader 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 {
    CCBCControlReplyHeader 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 short flags; /* Single-bit status flags */
    unsigned long data[CCB_MAX_INTEG]; /* The integrated data */
} CCBIntegMsg;
```

CCBMonitorMsg – Monitor data messages

Callback functions registered with `ccb_monitor_msg_callback()` are invoked whenever a message structure of the following type is received by the communications library over the telemetry link.

```
#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.

Appendix A

Changes to the library in version 2.0

Version 1 of the CCB communications libraries was released before much work had been done on either the CCB firmware or the CCB manager. Subsequent work on those components revealed the need for a few changes to the libraries. The following sections describe changes to the public library APIs and to the behavior of the demonstration/test programs. Bug-fixes are not discussed.

A.1 Changes to the shared parts of the client and server APIs

A.1.1 Added functions

- `ccb_settling_time()`

This function returns the settling time corresponding to any given combination of changes in cal-diode and phase-switch states, according to the specified CCB configuration. In particular, this is now used by the software simulator to determine how many integrations to flag when it toggles the states of one or both of the cal-diode switches.

- `ccb_integ_per_interval()`

This function computes and returns a count of the number of integration periods that will completely fit within a given time-interval, and the remaining time-duration of any final partial integration. In particular, this can be used to compute the number of integrations needed to generate a scan of a given time-duration.

- `ccb_cal_cycle_length()`

This returns a count of the number of integrations needed to complete a single calibration cycle.

- `ccb_set_sampler_cnf()`

This installs the configuration of the sample inputs of the slave FPGAs. This is used to select whether the integrators and dump-mode frames get their input samples from the real ADCs, or from a generator of fake, pseudo-random samples.

- `ccb_get_sampler_cnf()`

This returns the current sampler configuration from a given configuration object.

- `ccb_interval_is_zero()`

This is a simple utility function which returns a truth value if its `CCBInterval` argument denotes a time interval of zero.

A.1.2 Added datatypes

- `CCBSampleType`

This is an enumeration which lists the types of sample inputs that can be configured when calling the `ccb_set_sampler_cnf()` function.

- `CCBSamplerCnf`

This is a structure which encapsulates the configuration information that is returned by calling the `ccb_get_sampler_cnf()` function, or installed when calling the `ccb_set_config()` function.

- `CCBDriverType`

This is an enumeration which lists the types of target drivers that can be loaded when calling the `ccb_queue_load_driver_cmd()` function.

- `CCBIntegFlags`

This is an enumeration of the set of integration-status flags that may be encountered, bitwise OR'd together, within the newly added `flags` header-parameter of each integration.

A.1.3 Added constants

- CCB_NUM_ADC

This constant parameterizes the number of slots for ADCs in the CCB (ie. 16).

- CCB_DUMP_MAX_SAMPLES

When commanding a dump-mode scan, this value can be used to request that as many samples as possible, be collected from each integration of the scan.

- CCB_DUMP_ALL_FRAMES

When commanding a dump-mode scan, this value tells the driver to indefinitely dump one dump-mode frame per integration, until a new scan or intra-scan is commanded.

- CCB_MAX_ROUNDTRIP_DT

This value of this constant is the maximum permitted value of the CCB↔receiver round-trip propagation-delay configuration parameter.

- CCB_MAX_HOLDOFF_DT

This value of this constant is the maximum permitted value of the interrupt holdoff-delay configuration parameter.

A.1.4 Functions with modified APIs

- `ccb_set_config()`

This function now has an additional `sampler` argument, used for installing the newly added configuration of the hardware input sampling.

- `ccb_integration_duration()`

Now that the hardware specification no longer calls for settling-time delays to be inserted at the start of some integrations, all integrations now have the same physical duration. Thus the `integration` argument, which specified which integration's duration was to be characterized, has been removed.

- `ccb_set_phase_switch_cnf()`

Now that the receivers have taken on the responsibility for determining which backend drives their cal-diode and phase switches, there is no longer any point in tri-stating the CCB's switch-control outputs. Thus the `driven_switches` argument has been removed.

- `ccb_set_cal_diode_cnf()`

As described above for the phase-switch drivers, the `driven_diodes` argument is now superfluous, and has been removed.

- `ccb_set_timing_cnf()`

The `sample_dt` and `analog_reset_dt` timing-configuration arguments were specific to a now-abandoned scheme, where analog integrators were used to drive low-speed ADCs. Now that this scheme has been replaced by one that uses digital integrators and fast ADCs, these arguments have no meaning, and have thus been removed.

Also, while working through the designs of the analog and digital electronics, it became apparent that there would be roughly a $1\mu\text{s}$ delay between the CCB backend commanding a change in state of any of the cal-diode and phase switches, and the first effects of this reaching the inputs of the digital integrators. Although this delay could have been accommodated by extending the cal-diode and phase-switch settling times, such that samples during this round-trip delay were blanked, this would have unnecessarily thrown away some good data each time that the phase-switches were toggled. Thus a new `roundtrip_dt` configuration argument has been added to parameterize this propagation delay, and thus allow the CCB hardware to continue to collect data during this interval.

Finally, the implementation of the interrupt-generation logic indicated the need for a configuration parameter to limit the maximum rate at which the CCB hardware should generate interrupts. Thus a new `holdoff_dt` configuration argument has been added. This specifies how long the hardware should wait between sending one interrupt and sending the next, including how often it should re-signal any unacknowledged interrupts.

A.1.5 Modified datatypes

- `CCBGeneralStatus`

Now that the CCB no longer has a hardwired standby state, the CCB library no longer knows when the manager has placed the CCB in what it deems to be a standby state. Thus the CCB status-reports can no longer include an indication of when the CCB is in standby mode. The `CCB_STANDING_BY` enumerator has thus been removed from the status enumeration.

- **CCBConfigType**

The group of configuration parameters represented by the new sampler configuration, required the addition of a corresponding enumerator (`CCB_CNF_SAMPLER`) to the `CCBConfigType` enumeration.

- **CCBPhaseSwitchCnf**

The superfluous `driven_switches` parameter has been removed, as already explained for the `ccb_set_phase_switch_cnf()` function.

- **CCBCalDiodeCnf**

The superfluous `driven_diodes` parameter has been removed, as already explained for the `ccb_set_cal_diode_cnf()` function.

- **CCBTimingCnf**

As explained already for the `ccb_set_timing_cnf()` function, the `sample_dt` and `analog_reset_dt` members of the timing configuration have been removed, and new `roundtrip_dt` and `holdoff_dt` members have been added.

A.1.6 Functions with modified behaviors

- `ccb_integration_duration()`

As already mentioned, this function no longer inserts settling-time delays at the start of integrations.

A.1.7 Removed functions

- `ccb_cal_diode_delay()`

This function determined the calibration-diode settling-time delay to insert at the start of a given integration. Now that such delays are no longer inserted at the start of integrations, this function has been removed. For simulation purposes, the new `ccb_settling_time()` function provides the equivalent functionality for determining how much data will be flagged after arbitrary cal-diode and/or phase-switch changes.

- `ccb_nocal_duration()`

This function computed the constant part of the physical duration of an integration that followed any calibration-diode settling-time delays that were inserted between integrations. Now that no such delays are inserted, the more appropriately named `ccb_integration_duration()` function supersedes this function.

A.2 Changes to the client-specific parts of the public API

A.2.1 Added functions

- `ccb_queue_load_driver_cmd()`

This should be called before any commands are sent to the CCB, and determines whether subsequent commands will be sent to the real hardware or to a software simulator of the hardware. It can thereafter be sent at any time to switch between drivers.

- `ccb_queue_dump_scan_cmd()`

This function is used to initiate a dump-mode scan. This type of scan writes data-frames of unintegrated samples to a named pipe on the CCB computer, instead of sending frames of integrated data to the manager.

A.2.2 Functions with modified APIs

- `ccb_integ_msg_callback()`

The arguments of the `CCBIntegMsgFn` callback function argument have changed, as described below.

A.2.3 Modified datatypes

- `CCBIntegMsgFn`

Functions of this type are called to deliver integrated data to the manager. Such callback functions now include an extra `flags` argument, which contains a set of integration-status flags, describing such things as the states of the cal-diodes, which slave FPGA boards are present, and whether the parent integration should be flagged.

A.2.4 Removed functions

- `ccb_queue_standby_cmd()`

Previously this function did two things. It temporarily turned off specified telemetry streams to the manager, and placed its receiver control outputs in a high impedance state. Now that the receiver-control lines are no longer tri-stated, this function was left solely with the duty of controlling which telemetry streams were active during, what it considered to be its standby state. Since this could already be done, in a more general way, by the `ccb_queue_telemetry_cmd()` function, and because the library's idea of what constituted standby mode, differed from Green Bank conventions, the redundant `ccb_queue_standby_cmd()` function has been removed.

- `ccb_queue_awaken_cmd()`

This function's sole purpose was to undo what the `ccb_queue_standby()` function did, so now that the latter function has been removed, the `ccb_queue_awaken_cmd()` function has also been removed.

A.3 Changes to the server-specific parts of the public API

A.3.1 Added functions

- `new_CCBServerDriver()`

This function allocates an object which encapsulates the method functions and data that implement the server's interface to either the real or the virtual CCB drivers.

- `del_CCBServerDriver()`

This function reclaims the resources of an object that was previously returned by `new_CCBServerDriver()`.

- `ccb_server_event_loop()`

The server event-loop has now been moved inside the library, since it needs to operate above the drivers. This function invokes the CCB server's event-loop.

A.3.2 Functions with modified APIs

- `new_CCBServerLink()`

Previously this function's arguments were the method functions and callback data that implemented the CCB driver interface. Now that the server needs to support switching between two such interfaces, to allow targeting either the CCB hardware, or a software simulator, the interface functions and data of a given driver have been moved into `CCBServerDriver` objects, which are now allocated by the new `new_CCBServerDriver()` function. The original arguments of the `new_CCBServerLink()` function have thus been replaced by pointers to two `CCBServerLink` objects, representing the real and virtual drivers.

- `ccb_queue_integ_msg()`

Integrations now include a new integration-status parameter, so this function, which queues an integration to be sent to the manager, now includes a corresponding extra `flags` argument.

A.3.3 Added datatypes

- `CCBServerDriver`

As already mentioned, this new type of object encapsulates the method functions and callback data which implement the interface of a given CCB driver.

- `CCBDrvDumpScan`

This structure encapsulates the parameters that determine the dump-mode-specific behavior of a dump-mode scan, and is used to pass this data to a CCB driver.

- `CCBDriverSelectEventsFn`

This is a new driver function which tells the server which `select()` file-descriptor and timeout events it wishes to have it watch for.

- `CCBDriverCheckEventsFn`

This is a new driver function which responds to events reported by `select()`.

A.3.4 Modified datatypes

- CCBDriverCmd

In the original version of the library, the server could control which CCB interrupts were enabled, and when. In retrospect there doesn't seem to be any good reason for it to do this, since the driver can turn on interrupts at load time, and turn them off at unload time. Thus the `intr` (interrupt mask) member of the `pars` union has been removed.

A new `dump` member has been added to the `pars` union, for passing the driver the dump-specific configuration parameters of a dump-mode scan.

- CCBDriverCmdID

The removal of the driver interrupt-mask command, and the addition of the dump-scan command, dictated the removal of the `CCB_DRV_INTR_MASK` enumerator, and the addition of the `CCB_DRV_DUMP_SCAN` enumerator.

- CCBDrvConfScan

The addition of the sampler configuration group, required the addition of a sampler member in the driver interface's configuration-parameter object.

- CCBDriverLoadFn, CCBDriverUnloadFn, CCBDriverTellFn, CCBRebootRTCFn, CCBShutdownRTCFn

All of these functions now take an extra initial argument, which provides them with the `CCBServerLink` object of the server.

A.3.5 Removed functions

- `ccb_server_communicate` and `ccb_server_select_args`

Now that the CCB server's event-loop has been moved inside the new `ccb_server_event_loop()` function, the above two functions have been made private to the library.

A.3.6 Removed Datatypes

- CCBDrvInterrupt

This enumeration listed the types of interrupt that could be enabled and disabled by the server. Now that the server doesn't control interrupts, this enumeration is superfluous.

- CCBDrvIntrMask

This structure, which previously was used to communicate the server’s choice of interrupt mask to the driver, was superfluous.

A.4 Changes to the demonstration programs

A.4.1 The evolution of `ccb_demo_server` into `ccbserver`

The `ccb_demo_server` program has been replaced by the new `ccbserver` program. The new program, which will become the production CCB server, is able to switch between a virtual (demonstration) driver and the real CCB driver. Thus, once the manager has connected to the CCB server, it will be able to switch, at any time, between off-line testing using the virtual driver, and on-line interaction with the real CCB hardware. This is achieved by the manager, by calling the new `ccb_queue_load_driver_cmd()` function to load either the real or virtual CCB drivers. When the virtual driver is loaded, `ccbserver` acts like `ccb_demo_server` used to act, except that the simulation has been updated to accommodate the new timing model, and other new features in the library. In particular, note that the simulation goes as far as flagging fake integrations that are generated during the settling-time delays following cal-diode switches, just as would happen in the CCB hardware.

At the moment, if the manager uses the `ccb_queue_load_driver_cmd()` function to request that the real CCB driver be loaded, the server does load the real driver. However, since the real driver is currently just a set of stubbed functions which report a “not implemented yet” error to the server and to the manager, the server ends up unloading the problematic driver, and then goes back to waiting for commands from the manager, without any driver loaded. Thus, until the manager next tells the server to load the virtual driver, subsequent commands sent to the driver result in “no driver loaded” error messages being sent back to the manager.

A.4.2 Changes to the features of `ccb_demo_client`

In the `ccb_demo_client` demonstration program, there is a new pane for initiating and controlling dump-mode scans, and a new pane for specifying whether ADC samples or pseudo-random, fake, samples should be collected. There is also a new parameter for switching between the simulation driver and the real CCB driver, although for the moment, selecting the real driver elicits an error message from the demo-server, saying that the driver’s functions aren’t implemented. The timing-control panel no longer has entry areas for the removed `analog_reset_dt` and `sample_dt` parameters, but now has additional fields for the new `roundtrip_dt` and `holdoff_dt` parameters. The *standby* and *awaken* buttons have been removed, and the *stream-selection* radio-button-menu has been replaced by three buttons, called *off*, *standby* and *ready*. The latter three buttons invoke the `ccb_queue_telemetry_cmd()` function,

to select appropriate sets of telemetry streams, as follows:

- *stop* – Log messages.
- *standby* – Log messages, and monitoring data.
- *ready* – Log messages, monitoring data, and integrated data.

These are intended to reflect the modes of the same name that are used by the manager. After making a new connection to a server, the demo automatically loads whichever driver is specified by the *Load driver* option menu, and configures the telemetry stream to the *off* state, as indicated by the *off* button being red. It is necessary to hit the *ready* button before any integrated data will be displayed.

Appendix B

Changes to the library in version 3.0

B.0.3 Modified datatypes

- CCBTimingCnf

The `adc_delay_dt` configuration parameter was added. This configures the offset between the ADC clock and the main FPGA clock. Previously, the job of offsetting the ADC clock was to be done by the slave FPGAs, where it would have been difficult to support dynamic configuration, and it was to be implemented using compile-time delay-locked-loop options. Now, however, the task of delaying the ADC clock signal is performed in the master FPGA, using a revised technique that is much easier to dynamically control.

B.0.4 Functions with changed APIs

- `ccb_set_timing_cnf()`

The addition of the above-mentioned new `adc_delay_dt` parameter to the `CCBTimingCnf` data-type, required the addition of an identically named parameter to the `ccb_set_timing_cnf()` command, to allow it to be configured.

B.0.5 Functions with changed behaviors

- `ccb_integration_time()`

Originally this function computed the total time spent integrating ADC samples within a single integration period. However, it was requested that this be changed to return

the amount of time spent integrating samples within an individual phase-switch bin.
The new version does this.

Index

Function index

<code>ccb_add_intervals()</code>	36
<code>ccb_add_to_timestamp()</code>	42
<code>ccb_cal_cycle_length()</code>	40
<code>ccb_check_config</code>	23
<code>ccb_client_communicate()</code>	50
<code>ccb_client_connect()</code>	48
<code>ccb_client_disconnect()</code>	49
<code>ccb_client_poll_args()</code>	52
<code>ccb_client_select_args()</code>	51
<code>ccb_client_non_blocking_io()</code>	50
<code>ccb_client_selected_io()</code>	51
<code>ccb_client_sockets_callback()</code>	52
<code>ccb_clock_interval()</code>	37
<code>ccb_cmd_error_callback()</code>	54
<code>ccb_compare_intervals()</code>	37
<code>ccb_compare_timestamps()</code>	42
<code>ccb_copy_config()</code>	23
<code>ccb_cycle_length()</code>	38
<code>ccb_default_config()</code>	22
<code>ccb_diff_config()</code>	23
<code>ccb_get_cal_diode_cnf()</code>	28
<code>ccb_get_phase_switch_cnf()</code>	26
<code>ccb_get_sampler_cnf()</code>	33
<code>ccb_get_timestamp()</code>	41
<code>ccb_get_timing_cnf()</code>	31
<code>ccb_hms_of_timestamp()</code>	43
<code>ccb_integ_msg_callback()</code>	68
<code>ccb_integ_per_interval()</code>	40
<code>ccb_integration_duration()</code>	39
<code>ccb_integration_time()</code>	39
<code>ccb_interval_is_zero()</code>	37
<code>ccb_log_msg_callback()</code>	70
<code>ccb_log_server_msg()</code>	91
<code>ccb_monitor_msg_callback()</code>	68
<code>ccb_ping_echos()</code>	62
<code>ccb_queue_dump_scan_cmd()</code>	59

<code>ccb_queue_integ_msg()</code>	90
<code>ccb_queue_load_driver_cmd()</code>	60
<code>ccb_queue_logger_cmd()</code>	61
<code>ccb_queue_monitor_cmd()</code>	60
<code>ccb_queue_monitor_msg()</code>	90
<code>ccb_queue_ping_cmd()</code>	62
<code>ccb_queue_reboot_cmd()</code>	64
<code>ccb_queue_reset_cmd()</code>	62
<code>ccb_queue_shutdown_cmd()</code>	63
<code>ccb_queue_start_scan_cmd()</code>	56
<code>ccb_queue_status_request_cmd()</code>	63
<code>ccb_queue_stop_scan_cmd()</code>	58
<code>ccb_queue_telemetry_cmd()</code>	61
<code>ccb_scale_interval()</code>	35
<code>ccb_scan_duration()</code>	39
<code>ccb_server_event_loop()</code>	89
<code>ccb_set_cal_diode_cnf()</code>	27
<code>ccb_set_config()</code>	23
<code>ccb_set_phase_switch_cnf()</code>	24
<code>ccb_set_sampler_cnf()</code>	32
<code>ccb_set_timing_cnf()</code>	29
<code>ccb_settling_time()</code>	38
<code>ccb_status_reply_callback()</code>	65
<code>ccb_subtract_intervals()</code>	36
<code>ccb_time_to_timestamp()</code>	43
<code>ccb_time_until()</code>	42
<code>ccb_zero_interval()</code>	37
<code>del_CCBClientLink()</code>	49
<code>del_CCBConfig()</code>	22
<code>del_CCBServerDriver()</code>	88
<code>del_CCBServerLink()</code>	89
<code>new_CCBClientLink()</code>	47
<code>new_CCBConfig()</code>	22
<code>new_CCBServerDriver()</code>	82
<code>new_CCBServerLink()</code>	81

Datatype index

CCB_MAX_LOG	107
CCB_MAX_LOG_VARIANTS	67
CCB_MAX_MONITOR	107
CCB_MAX_NCAL	27
CCB_MIN_SAMP_PER_STATE	26
CCB_MONITOR_MSG_FN	68
CCB_NUM_ADC	59
CCB_REBOOT_RTC_FN	88
CCB_SHUTDOWN_RTC_FN	88
CCB_STATUS_REPLY_FN	65
CCB_TELEMETRY_PORT	12