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

Martin Shepherd, California Institute of Technology

February 3, 2003

This page intentionally left blank.

Abstract

This document documents the communications interface between the Ygor manager process and the server of the Caltech Continuum Backend (CCB). It starts with a high-level overview of the 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

List of Figures

Chapter 1

Introduction

Communications between the manager and the CCB server is implemented in a shared communications library. As far as possible, this library hides the specifics of the communications protocols used, the buffering used for non-blocking I/O, and the queuing of output messages. To facilitate the integration of the library with I/O-multiplexing event loops in the manager and server processes, the library allows the calling application to query the socket file-descriptors that are being used, and supports non-blocking socket I/O in both directions. Since the manager and server will probably reside on different computers, and could potentially be linked against different versions of the communications library, the opening dialog between the two systems verifies compatibility between the message definitions seen by the two systems. What follows is a high level description of how the library functions.

1.1 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 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 server.

2. **The telemetry port**

   This port is used by the server to send data to the manager. This includes integrated radiometer data, monitoring data and log messages. The CCB manager never sends messages to the server over this link, so this link is essentially under the control of the CCB server. The classes of the data that are sent by the server over this link, and the

frequency with which periodic data are sent, are configurable via commands sent to the server's control port.

Within this document, discussion of communications on these links are documented from the perspective of the manager. As such, words such as outgoing or incoming respectively refer to messages sent and received by the manager.

1.2 Connection establishment

The manager starts by attempting to connect to the control port of the CCB server, using a TCP/IP port number which is defined in an internal C header file, via the C macro CCB_CONTROL_PORT.

1.2.1 Connection authentication

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

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

1.2.2 Protocol authentication

Although the server and the manager use a common communications library, there is potential for different versions of this library being used by the two systems. If the message definitions have been changed in one copy of the library, but not in the other, it is best that this be detected when the control connection is first established during commisioning, rather than waiting for an unfortunate observer to encounter the problem when the conflicting message type is first exchanged. To do this, the connection establishment function of the manager sends the CCB server a complete description of all of the control and telemetry message types that it knows about. If the server detects any discrepancies between these definitions and its own copy of the message definitions, it closes the control connection to reject the incompatible server.

If the message definitions do match, the CCB server sends a single-byte connection acknowledgment message to the manager, telling it that the connection has now been accepted, and that it can proceed.

1.3 Telemetry connection establishment

Once a control connection to the CCB server has been accepted by the server, the CCB server becomes receptive to connection requests to its telemetry server port, the number of which is provided in a C macro called CCB_TELEMETRY_PORT, in an internal header of the communications library. Only if the originating IP address of this connection-request matches that used to establish the connection to the server's control port, is the connection accepted. Otherwise the server simply closes the suspicious connection.

Note that telemetry connections are closed automatically by the CCB server whenever the manager closes its control connection.

1.3.1 Initial configuration

Once a new manager/server communications connection has been established, the CCB server waits in standby mode. This allows the manager to override the server's default configuration parameters with its own, before sending an "on" command, to awaken the CCB.

Chapter 2

The CCB client communications API

2.1 Connecting to a CCB server

All manager interactions with the communications library are via C function calls. Connection establishment to the CCB server is initiated by the new_CCBClient() constructor function.

```
CCBClient *new_CCBClient(const char *host);
```

In addition to establishing communications with the CCB server on the specified host, this function returns a pointer to the opaque, dynamically allocated CCBClient object which the communications library uses to manage the CCB communications link. Subsequently this object should be passed to all library functions, to communicate with the specified host. Note that since each CCBClient object returned by this function is dynamically allocated and encapsulates all of the configuration and state information of the connection to the specified CCB server, it is possible if desired, for a single manager to simultaneously communicate with both the 3mm and the 1cm CCB servers. The value of the host argument, which should contain the numeric or textual IP address of the target CCB server, determines which CCB server is associated with each new CCBClient object.

If the communications link can't be established, or there are insufficient resources available to allocate the client object, the new_CCBClient() function returns NULL.

Unless an error occurs, forcing new_CCBClient() to cleanup and return NULL, this function doesn't return until it has fully established and authenticated the control and telemetry connections to the specified CCB. Note that this is done using blocking socket I/O. Once new_CCBClient() returns, the caller has the option of switching communications to non-blocking I/O, as will be described later.

2.2 Disconnecting from a CCB server

Closing a CCB connection is performed by deleting its resource object via a call to the del_CCBClient() function.

```
CCBClient *del_CCBClient(CCBClient *ccb);
```

This function both shuts down the specified connection to the CCB server, and returns all resources in the given CCBClient object to the system. The function always returns NULL. This allows the caller to type:

```
CCBClient *ccb;
...
ccb = del_CCBClient(ccb);
```

This sets the invalidated ccb pointer variable to NULL, such that if any statement subsequently tries to access the deleted object through this pointer, it will get a segmentation fault, rather than producing unpredictable behavior.

2.3 Querying the socket file-descriptors of a CCB link

The socket file-descriptors of a given CCB connection, can be obtained via the ccb_client_sockets function.

```
int ccb_client_sockets(CCBClient *client,
                       int *cntrl_socket,
                       int *telem_socket);
```

The file-descriptors of the control and telemetry connections are respectively assigned to the variables pointed to by the cntrl_socket and telem_socket arguments. Either of these arguments can be NULL if there is no interest in the respective socket. Normally this function returns 0, but if client is NULL, a non-zero value is returned to report the error, and errno is set to EINVAL.

The manager should not perform reads from, writes to, or reconfigure the returned file descriptors. They are provided purely for use with the application's choice of the select() or poll() system calls to detect when I/O is possible.

2.4 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.

```
int ccb_client_non_blocking_io(CCBClient *client, 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.

2.5 I/O multiplexing

To perform I/O multiplexing it is necessary to use either select() or poll(), both to watch for the ability to write without blocking when there are any messages to be sent, and to watch for the arrival of data when incoming messages are expected. The library knows when control replies are expected from the CCB server, and when data of outgoing messages remain to be written, whereas the manager is the part responsible for watching for I/O, so the library provides the manager with the ccb_client_io_status() function to query which types of I/O it should be watching for on the control and telemetry sockets.

```
unsigned ccb_client_io_status(CCBClient *client);
```

The return value of this function is a bitwise union of CCBIOStatus enumerators.

```
typedef enum {
  CCB_TELEM_READ = 1,  /* Read pending on telemetry socket */
  CCB_TELEM_WRITE = 2, /* Write pending on telemetry socket */
  CCB_CTRL_READ = 4,   /* Read pending on control socket */
  CCB_CTRL_WRITE = 8   /* Write pending on control socket */
} CCBIOStatus;
```

The return value is constructed as follows:

- CCB_TELEM_READ

  This bit is always set, reflecting the fact that asynchronous telemetry data can arrive at any time.

- CCB_TELEM_WRITE

  This bit is always 0, since the manager doesn't send messages to the CCB server over the telemetry link. This enumerator is included because the same enumeration is used in the CCB server, which does write to its end of the telemetry link.

- CCB_CTRL_READ

  Whenever the library finishes writing a message to the control link, it moves the corresponding message structure from the queue of pending outgoing messages to the end of a list of incomplete transactions. The message structure is subsequently removed from this list when all of the replies that it is defined to elicit from the CCB server have been received. Note that all control messages elicit at least an acknowledgment message from the CCB server. Thus, when the list of incomplete transactions contains at least one message, the return value of ccb_client_io_status() includes CCB_CTRL_READ.

- CCB_CTRL_WRITE

  When unsent messages remain in the library's outgoing queue of control-messages, the return value of ccb_client_io_status() includes CCB_CTRL_WRITE.

2.6 Sending queued messages to the CCB server

Later sections describe the functions that the manager uses to queue the various types of outgoing control messages. The subsequent transmission of these messages is the responsibility of the ccb_client_send_control_msgs() function.

```
int ccb_client_send_control_msgs(CCBClient *ccb);
```

This function writes as much as possible before returning. If the ccb_client_non_blocking_io() function has previously been invoked to enable non-blocking I/O, and a write() to the control socket blocks while attempting to send any of the queued messages to the CCB server, this function returns immediately, otherwise it blocks until all unsent messages have been written to the control socket. In both cases, unless an error occurs, ccb_client_send_control_msgs() returns 0. On error non-zero is returned and errno is set to indicate the cause of the error.

Note that the inclusion of CCB_CTRL_WRITE in the return value of ccb_client_io_status() can be used to determine whether any control messages remain to be sent.

2.7 Receiving control replies from the CCB server

As described later, the manager provides the library with a callback function for each of the known types of replies received over the control connection. The ccb_client_recv_control_msgs() function reads replies from the control connection, and delivers them to the manager by calling the corresponding callbacks.

```
int ccb_client_recv_control_msgs(CCBClient *ccb);
```

This function attempts to read all outstanding replies from the control connection. If the ccb_client_non_blocking_io() function has previously been invoked to enable non-blocking I/O, and a read() of the control socket blocks while attempting to read any of the expected replies from the CCB server, this function returns immediately, otherwise it blocks until all anticipated messages have been read from the control socket. In both cases, unless an error occurs, ccb_client_recv_control_msgs() returns 0. On error non-zero is returned and errno is set to indicate the cause of the error.

Note that the inclusion of CCB_CTRL_READ in the return value of ccb_client_io_status() can be used to determine whether any control replies remain to be read.

2.8 Receiving telemetry replies from the CCB server

As described later, the manager provides the library with a callback function for each of the known types of incoming telemetry messages. The ccb_client_recv_telemetry_msgs() function reads these messages from the telemetry connection, and delivers them to the manager by calling the corresponding callbacks.

```
int ccb_client_recv_telemetry_msgs(CCBClient *ccb);
```

If the ccb_client_non_blocking_io() function has previously been invoked, ccb_client_recv_telemetry_msgs() keeps reading messages from the telemetry connection until the read blocks. Otherwise, in blocking-I/O mode, ccb_client_recv_telemetry_msgs() attempts to read a single message from the telemetry link, and blocks the caller until that message has been read. In both cases 0 is returned unless an error is encountered. On error errno is set to indicate the problem, and non-zero is returned.

2.9 Registering a command-error callback function

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

Just in case it is ever necessary to add arguments to this type of callback function, it is recommended that the manager use the CCB_CMD_ERROR_FN macro to both declare function prototypes and define the function itself, and that any pointers that it records to such callback functions be declared using the CCBCmdErrorFn typedef.

```
#define CCB_CMD_ERROR_FN(fn) int (fn)(CCBClient *ccb, void *data, \\
                                      long id, CCBCmdError status)

typedef CCB_CMD_ERROR_FN(CCBCmdErrorFn);

int ccb_cmd_error_callback(CCBClient *ccb, 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_GARBLED,     /* The contents of the command message */
                       /*  were invalid and couldn't be fixed. */
  CCB_CMD_IGNORED,     /* The command didn't make sense at this */
                       /*  time, and was ignored. */
  CCB_CMD_SYSERR       /* An unexpected internal software or hardware */
                       /*  error was encountered while while attempting */
                       /*  to execute the command. */
} CCBCmdError;
```

This enumeration is in the public header file of the communication library, so to prevent ABI problems if new error conditions are added, and somebody forgets to recompile either the library, the CCB server, or the manager, new error enumerators should always be appended to the end of the enumeration, rather than inserted, and old enumerators should not be removed or reordered. With this caveat, if functions that use values from this enumeration

are prepared to handle values that they don't know about, at worst an unknown error condition will elicit a warning, rather than, say accessing a nonexistent element in an array of error conditions, or associating the incorrect error condition with an enumerator.

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

2.10 Outgoing CCB commands

The following two sections describe the configuration and control commands which 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 CCBClient 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.

2.10.1 Outgoing CCB configuration commands

This section describes the public functions that are used to queue CCB configuration commands for subsequent dispatch to the CCB server when ccb_client_send_control_msgs() is called. Note that the changes to the configuration that these commands make, don't take effect until the next start-scan or stop-scan message is received by the CCB server.

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.

ccb_queue_phase_switch_cnf() – Reconfiguring 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 2.1.

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

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

Re-configuration of the phase switches is performed by first calling the ccb_queue_phase-_switch_cnf() function to queue a phase-switch reconfiguration message, followed by one or more calls to ccb_client_send_control_msgs() to send this to the CCB server.

```
int ccb_queue_phase_switch_cnf(CCBClient *ccb, long id,
                               unsigned active_switches,
                               unsigned driven_switches,
                               unsigned initial_states,
                               unsigned samp_per_state);
```

The arguments of this function are interpreted as follows.

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

  The two least-significant bits of this integer control which of the two phase switches should be switched, and which should be held in the phase-switch states specified by the initial_states parameter. A bit value of 0 indicates that the corresponding switch should be held in its initial position, whereas a bit value of 1 indicates that it should be switched according to the current switching mode.

- **Which switches are driven?** (driven_switches)

  The 2 least-significant-bits of this parameter control which of the 2 phase-switch control lines are driven. When they aren't being driven, other backends can control the phase switches. A bit value of 1 causes the corresponding line to be driven, whereas a value of zero causes it to be placed in a high impedance state.

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

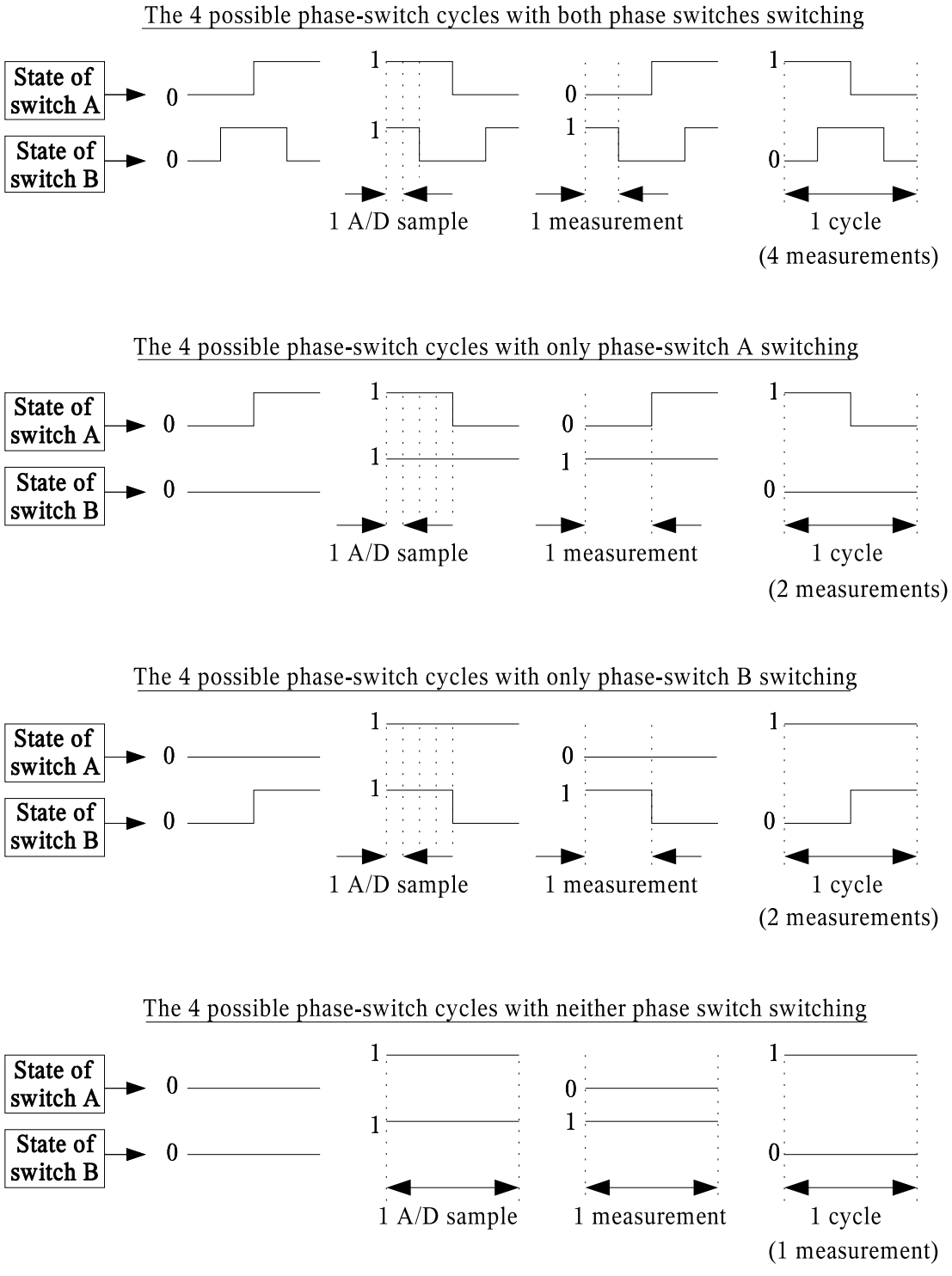

Figure 2.1: Example phase-switching cycles

The 2 least-significant-bits of this parameter specify the states that the 2 phase switches must have at the start of each new phase-switch cycle. Switches that aren't configured to be switching by the active_switches parameter are held in these states throughout each cycle.

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

  This parameter configures the number of A/D samples that are integrated between changes in the states of either phase-switch. Since the underlying state machine for each cycle is clocked by the sample clock, and there is room for 32 states per cycle, the product of the number of samples per measurement and the number of measurements per cycle sets an upper limit on the value of this parameter as follows:

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

ccb_queue_cal_diode_cnf() – Reconfiguring 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 is changed by first calling the ccb_queue_cal_diode_cnf() function to queue a cal-diode reconfiguration message, and then subsequently calling ccb_client_send_control_msgs() to send this to the CCB server.

```
int ccb_queue_cal_diode_cnf(CCBClient *ccb, long id,
                            unsigned ncal,
                            unsigned driven_diodes,
                            unsigned *diode_a,
                            unsigned *diode_b,
                            unsigned long *ninteg);
```

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.

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

  The 2 least significant bits of this parameter specifies which of the two calibration diode control lines are to be driven. A value of 1 means that the corresponding line should be driven, whereas a value of 0 means that it should be placed in a high impedance state.

- **The** ncal **states of diode A** (diode_a)

  The first ncal elements of this array specify whether noise diode A is to be on or off in the corresponding step of the calibration diode state machine. A value of 0 means off, and a value of 1 means on.

- **The** ncal **states of diode B** (diode_b)

  The first ncal elements of this array specify whether noise diode B is to be on or off in the corresponding step of the calibration diode state machine. A value of 0 means off, and a value of 1 means on.

- **The duration of each calibration step** (ninteg)

  Each of the first ncal elements of this parameter specify for how many integrations the state of the corresponding stage of the calibration diode state machine should be maintained. With an integration time of 1ms, the use of a 32-bit value translates to a maximum duration of 48 days. This is clearly overkill, but a 16-bit value would only support up to 65 seconds per state, which might not be enough.

ccb_queue_telemetry_cnf() – Reconfiguring the telemetry connection

The telemetry configuration command specifies what types of data are to be sent to the manager by the CCB server, and how frequently they should be sent. This is sent by first calling the ccb_queue_telemetry_cnf() function to queue a telemetry configuration message, then subsequently calling ccb_client_send_control_msgs() to send this message to the CCB server.

```
int ccb_queue_telemetry_cnf(CCBClient *ccb, long id,
                            unsigned integ_period,
                            unsigned monitor_interval,
                            unsigned stream_selection);
```

The arguments of this function are as follows.

- **The integration period** (integ_period)

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

- **The monitoring update interval** (monitor_interval)

  Instrumental monitoring data are sent to the manager with a period of this many integrations.

- **Data-stream selection** (stream_selection)

  This parameter contains a bit-wise union of CCBTelemetryStream enumerators, used to specify which of the radiometer, monitoring, and log message streams, should be forwarded to the manager, and which should be discarded.

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

  Normally all streams are selected by the manager. However in standby mode, the manager may wish to choose whether to completely ignore the backend, whether to just select instrumental monitoring and log data, or whether to continue to receive all data from the backend.

ccb_queue_timing_cnf() – Acquisition-timing configuration

The acquisition timing configuration command specifies the durations of timers in the CCB hardware. It is sent to the CCB server by first calling ccb_queue_timing_cnf() to queue it for dispatch, then subsequently calling ccb_client_send_control_msgs() to have it sent.

```
int ccb_queue_timing_cnf(CCBClient *ccb,
```

```
unsigned sample_dt,
unsigned phase_switch_dt,
unsigned analog_reset_dt,
unsigned long diode_rise_dt,
unsigned long diode_fall_dt);
```

The arguments of this function are interpreted as follows.

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

  This refers to the amount of time taken per A/D sample, including the time used to blank phase-switch transitions, but not including the time spent resetting the analog integrators. It is expressed as an integer multiplier of 100ns.

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

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

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

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

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

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

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

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

2.10.2 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_send_control_msgs() is called. Unlike the configuration commands of the previous section, the CCB server responds to control commands as soon as it receives them.

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

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

Name	Description
start-scan	Start a new scan at the end of the current integration
stop-scan	Start a new scan at the end of the current sample
reset	Re-initialize the hardware.
standby	Relinquish control of the receiver.
awaken	Take control of the receiver.
test-link	Request a link-verification reply.
check-status	Request a status reply.
shutdown	Shutdown the real-time CPU.
reboot	Reboot the real-time CPU.

ccb_queue_start_scan_cmd() – Queuing a start-scan command

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

The new scan adopts any changes to the configuration parameters that have been made since the last start-scan or stop-scan command was received.

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

```
int ccb_queue_start_scan_cmd(CCBClient *ccb, long id,
                             unsigned long date,
                             unsigned long tod);
```

The arguments of this function are interpreted as follows.

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

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

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

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

ccb_queue_stop_scan_cmd() – Queuing a stop-scan command

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

Note that although this command starts a new scan, it is called stop-scan because it stops an observing scan. The scan that it then starts, which can usefully be refered 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.

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

```
int ccb_queue_stop_scan_cmd(CCBClient *ccb, long id);
```

ccb_queue_reset_cmd() – Queuing a reset command

This command resets both the hardware and the device driver.

On receiving this command, the server tells the device driver to order the FPGA to reload its firmware from EPROM. The device driver then disconnects from the CCB server process. The CCB server process reacts to this by unloading and reloading the device driver. The device driver then proceeds to detect and re-initialize the hardware with its built-in default configuration parameters. The server then re-opens its connection to the device driver, then sends the reloaded driver the current configuration parameters, then behaves as though a standby command had been received.

reset commands are sent by first calling ccb_queue_reset_cmd() to queue the command for dispatch, then subsequently calling ccb_client_send_control_msgs() to send it to the CCB server.

```
int ccb_queue_reset_cmd(CCBClient *ccb, long id);
```

ccb_queue_standby_cmd() – Queuing a standby command

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

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

```
int ccb_queue_standby_cmd(CCBClient *ccb, long id,
                          unsigned stream_mask);
```

The arguments of this function are interpreted as follows.

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

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

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

ccb_queue_awaken_cmd() – Queuing an awaken command

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

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

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

```
int ccb_queue_awaken_cmd(CCBClient *ccb, long id);
```

ccb_queue_test_link_cmd() – Queuing a test-link command

On receiving this command the CCB server replies to the manager with a ctrl-link-reply message over the control connection, and a telem-link-reply message over the telemetry connection. This provides a means by which the manager can check if both the CCB server, and its TCP/IP links are alive. The intention is that the following sequence of operations be performed by the manager every few minutes.

1. The manager clears the two internal reply-received flags which it asserts whenever telem-link-reply and ctrl-link-reply messages are received.
2. The manager then sends a test-link command to the CCB server.
3. Rather than waiting for a reply, the manager simply arranges to be notified when data arrives over the command and telemetry links, then goes back to doing whatever it was doing.
4. Subsequently, when a ctrl-link-reply message is received by ccb_client_recv_control_msgs() or a telem-link-reply is received by ccb_client_recv_telemetry_msgs(), the manager's corresponding callback function is invoked, as documented later, and this function asserts the corresponding internal reply-received flag, then returns, to allow the manager to continue doing whatever it was doing.
5. At the time at which the manager is due to send the next test-link command, it checks that since the previous one was sent, both a ctrl-link-reply and a telem-link-reply have been received, by checking the corresponding internal reply-received flags. If not, this denotes a problem either with one of the links, or with the CCB server, and the manager should alert the operator.
6. The manager goes back to step 1.

test-link commands are sent by first calling ccb_queue_test_link_cmd() to queue the command for dispatch, then subsequently calling ccb_client_send_control_msgs() to send it to the CCB server.

```
int ccb_queue_test_link_cmd(CCBClient *ccb, long id);
```

ccb_queue_check_status_cmd() – Queuing a check-status command

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

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

```
int ccb_queue_check_status_cmd(CCBClient *ccb, long id);
```

ccb_queue_shutdown_cmd() – **Queuing a** shutdown **command**

On receiving this command the CCB server places the hardware in standby mode, disables CCB interrupts, unloads the CCB device driver, then uses the linux `reboot()` system call to shutdown linux, and if possible turn off the real-time computer.

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

```
int ccb_queue_shutdown_cmd(CCBClient *ccb, long id);
```

ccb_queue_reboot_cmd() – **Queuing a** reboot **command**

On receiving this command the CCB server places the hardware in standby mode, disables CCB interrupts, unloads the CCB device driver, then uses the linux `reboot()` system call to reboot linux.

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

```
int ccb_queue_reboot_cmd(CCBClient *ccb, long id);
```

2.11 Incoming control-link replies

This section describes the library functions that are used by the manager to register callback functions for ccb_client_recv_control_msgs() to subsequently use to deliver control-link 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_recv_control_msgs() 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
ctrl-link-reply	A reply to a test-link command.
ctrl-status-reply	A reply to a check-status command.

Beware that since all of the callback functions described below are C functions, whereas the manager is a C++ program, within the manager, both the prototypes of the callbacks and their definitions must be enclosed in `extern "C" {}` blocks. Along with each callback function, the manager can specify an arbitrary `void *` pointer to be passed to the callback whenever it is called. This should be used by the manager to pass the callback function any resources that it needs to handle the corresponding reply message. In addition to this pointer, each callback function is passed a pointer to the CCBClient object that received the message, plus any arguments corresponding to the contents of the message.

ccb_ctrl_link_reply_callback() – Routing control-socket test-link replies

The public ccb_ctrl_link_reply_callback() function is used to register the callback function that will subsequently be called by ccb_client_recv_control_msgs() whenever it receives a ctrl-link-reply message. Just in case it is ever necessary to add arguments to this type of callback function, it is recommended that the manager use the CCB_CTRL_LINK_REPLY_FN macro to both declare function prototypes and define the function itself, and that any pointers that it records to such functions be declared using the CCBCtrlLinkReplyFn typedef.

```
#define CCB_CTRL_LINK_REPLY_FN(fn) int (fn)(CCBClient *ccb, void *data)

typedef CCB_CTRL_LINK_REPLY_FN(CCBCtrlLinkReplyFn);

int ccb_ctrl_link_reply_callback(CCBClient *ccb, CCBCtrlLinkReplyFn *fn,
                                 void *data);
```

The callback function is registered via the fn argument of ccb_ctrl_link_reply_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. ctrl-link-reply messages contain no information, since the fact that they are received at all tells the manager that the control link is OK, so the callback function doesn't take any other arguments.

ccb_ctrl_status_reply_callback() – **Routing** check-status **replies**

The public ccb_ctrl_status_reply_callback() function is used to register the callback function that will subsequently be called by ccb_client_recv_control_msgs() whenever it receives a ctrl-status-reply message. Just in case it is ever necessary to add arguments to this type of callback function, it is recommended that the manager use the CCB_CTRL_STATUS_REPLY-_FN macro to both declare function prototypes, and to define the function itself, and that any pointers to these callbacks be declared using the CCBCtrlStatusReplyFn typedef.

```
#define CCB_CTRL_STATUS_REPLY_FN(fn) int (fn)(CCBClient *ccb, void *data,
                                              unsigned long status)

typedef CCB_CTRL_STATUS_REPLY_FN(CCBCtrlStatusReplyFn);

int ccb_ctrl_status_reply_callback(CCBClient *ccb,
                                   CCBCtrlStatusReplyFn *fn, void *data);
```

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

```
typedef enum {
  CCB_LINK_DOWN   = 1,  /* The telemetry link is down */
  CCB_BUFFER_FULL = 2,  /* The telemetry output buffer filled */
                        /*  up and hasn't drained yet, so data */
                        /*  are being discarded. */
  CCB_HARD_FAULT  = 4,  /* A hardware fault has been detected */
  CCB_SOFT_FAULT  = 8,  /* A software fault has been detected */
  CCB_STANDING_BY = 16  /* The CCB is in standby mode */
} CCBGeneralStatus;
```

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

2.12 Incoming telemetry messages

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

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_recv_control_msgs() responds to this by returning non-zero, the manager can inspect errno to see what happened.

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

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

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

Message type	Priority	Buffer size	Buffer overflow disposition
monitor-data	0	1 message	Overwrite the previous unsent message
integ-data	1	2MB (>= 30s)	Discard new data while the buffer drains
log-message	2	Dynamic	N/A
telem-link-reply	3	1 message	Overwrite the previous unsent message

As can be seen, replies to test-link 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-link-reply reply is requested.

log-message messages have the second highest priority, to prevent important messages from being held up indefinitely. They are stored in a dynamically sized queue, since it is important not to loose any of these messages. To guard against a rapidly repeated sequence of error messages consuming all of the available memory, log message strings are recorded in a hash-table, and each entry in the queue of log messages points at the corresponding hash-table entry. Since their aren't too many potential error messages, messages in the hash-table are

never removed once entered. Along with each message string, each hash-table entry also records the time at which the string was last reported to the manager. So, whenever a log message is generated, the manager looks it up in the hash table, adds it if it hasn't been added before, then looks to see if the same message has been reported within a certain period (eg. 1 minute). If it has, then the server doesn't redundantly report the repeated message to the manager. This prevents a rapidly repeating error condition from hogging both memory in the server, and the telemetry link.

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 is 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 monitoring data isn'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_recv_telemetry_msgs() to subsequently use to deliver telemetry messages.

All telemetry message callback functions have 2 arguments in common, these being the CCBClient object that received the message, and a pointer to a CCBTelemInfo structure, which reports the date, time, and the value of the scan counter when the message was originally generated.

```
typedef struct {
  unsigned long date;   /* The Modified Julian Day number */
  unsigned long tod;    /* The time of day in milliseconds */
  unsigned long scan;   /* The value of the scan counter */
} CCBTelemInfo;
```

The members of this structure are interpreted as follows.

- **date**

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

- **tod**

This is the time of day at which the telemetry message was assembled. The time of day is specified as the number of milli-seconds that have passed since 0H UTC on the day indicated by the date argument.

- **scan**

  This identifies the scan during which the message was generated, using the value of a sequential scan counter.

  start-scan and stop-scan commands are tagged on receipt with the value of this counter, which is initialized to zero whenever the device driver is loaded, and thereafter incremented by one whenever a new start-scan or stop-scan command is received. Whenever the scan associated with one of these commands actually starts, the value of the scan counter associated with that command is thereafter used as the value of the scan counter to be included in subsequent telemetry messages.

  Placing this value in each data packet allows the manager to associate each integration with the corresponding scan configuration parameters that were instantiated when the new scan was started.

ccb_monitor_data_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_data_msg_callback() function is used to register the callback function that will subsequently be called by ccb_client_recv_telemetry_msgs() whenever it receives a monitor-data message. Just in case it is ever necessary to add arguments to this type of callback function, it is recommended that the manager use the CCB_MONITOR_DATA-_MSG_FN macro to both declare function prototypes and define the function itself, and that any pointers that it records to such functions be declared using the CCBMonitorDataMsgFn typedef.

```
#define CCB_MONITOR_DATA_MSG_FN(fn) int (fn)(CCBClient *ccb, void *data,\\
                                CCBTelemInfo *info, unsigned long number,
                                unsigned long *values, unsigned nvalues)

typedef CCB_MONITOR_DATA_MSG_FN(CCBMonitorDataMsgFn);

int ccb_monitor_data_msg_callback(CCBClient *ccb, CCBMonitorDataMsgFn *fn,
                                  void *data);
```

The callback function is registered via the fn argument of ccb_monitor_data_msg_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. The number argument is the value of an integer counter in the device

driver, used to give monotonically increasing numbers to each new monitor message within a scan. This counter is reset to zero at the start of each new scan, and thereafter incremented by 1 whenever a new monitor message is prepared for transmission. 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_data_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_data_msg_callback() function is used to register the callback function that will subsequently be called by ccb_client_recv_telemetry_msgs() whenever it receives an integ-data message. Just in case it is ever necessary to add arguments to this type of callback function, it is recommended that the manager use the CCB_INTEG_DATA_MSG_FN macro to both declare function prototypes and define the function itself, and that any pointers that it records to such functions be declared using the CCBIntegDataMsgFn typedef.

```
#define CCB_INTEG_DATA_MSG_FN(fn) int (fn)(CCBClient *ccb, void *data,\\
                                CCBTelemInfo *info, unsigned long number,
                                unsigned long *values, unsigned nvalues)

typedef CCB_INTEG_DATA_MSG_FN(CCBIntegDataMsgFn);

int ccb_integ_data_msg_callback(CCBClient *ccb, CCBIntegDataMsgFn *fn,
                                  void *data);
```

The callback function is registered via the fn argument of ccb_integ_data_msg_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. The number argument is the value of the scan counter, as previously documented. The manager can use this to check for missing integ-data messages. The first nvalues elements of the array pointed to by the values[] argument, contain the radiometer integrations.

ccb_log_msg_callback() – Routing telemetry log-message messages

The server sends both 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_recv_telemetry_msgs() whenever it receives a log-message message. Just in case it is ever necessary to add arguments to this type of callback function,

it is recommended that the manager use the CCB_LOG_MSG_FN macro to both declare function prototypes and define the function itself; and that any pointers that it records to such functions be declared using the CCBLogMsgFn typedef.

```
#define CCB_LOG_MSG_FN(fn) int (fn)(CCBClient *ccb, void *data,\\
                                    CCBTelemInfo *info, const char *msg)

typedef CCB_LOG_MSG_FN(CCBLogMsgFn);

int ccb_log_msg_callback(CCBClient *ccb, 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.

ccb_telem_link_reply_callback() – Routing telem-link-reply messages

When a test-link command is received by the CCB server, the server sends messages to the manager over both the control and telemetry links. The telemetry side of this reply is passed as a telem-link-reply message.

The public ccb_telem_link_reply_callback() function is used to register the callback function that will subsequently be called by ccb_client_recv_telemetry_msgs() whenever it receives a telem-link-reply message. Just in case it is ever necessary to add arguments to this type of callback function, it is recommended that the manager use the CCB_TELEM_LINK_REPLY-_FN macro to both declare function prototypes and define the function itself, and that any pointers that it records to such functions be declared using the CCBTelemLinkReplyFn typedef.

```
#define CCB_TELEM_LINK_REPLY_FN(fn) int (fn)(CCBClient *ccb, void *data,\\
                                             CCBTelemInfo *info)

typedef CCB_TELEM_LINK_REPLY_FN(CCBTelemLinkReplyFn);

int ccb_telem_link_reply_callback(CCBClient *ccb, CCBTelemLinkReplyFn *fn,
                                  void *data);
```

The callback function is registered via the fn argument of ccb_telem_link_reply_callback(), and any application-specific resources that should be passed to the callback are specified via the data argument. telem-link-reply messages contain no information, so the callback function has no further arguments.

2.13 Shared libraries and their versioning

The communications library is compiled as a shared library 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 must be used when the 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.

Chapter 3

Library internals

The communications library is comprised of three distinct 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 is passed a completely read message within the internal packet buffer of the input stream, and decomposes and decodes the contents of the message into local 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 3.1.

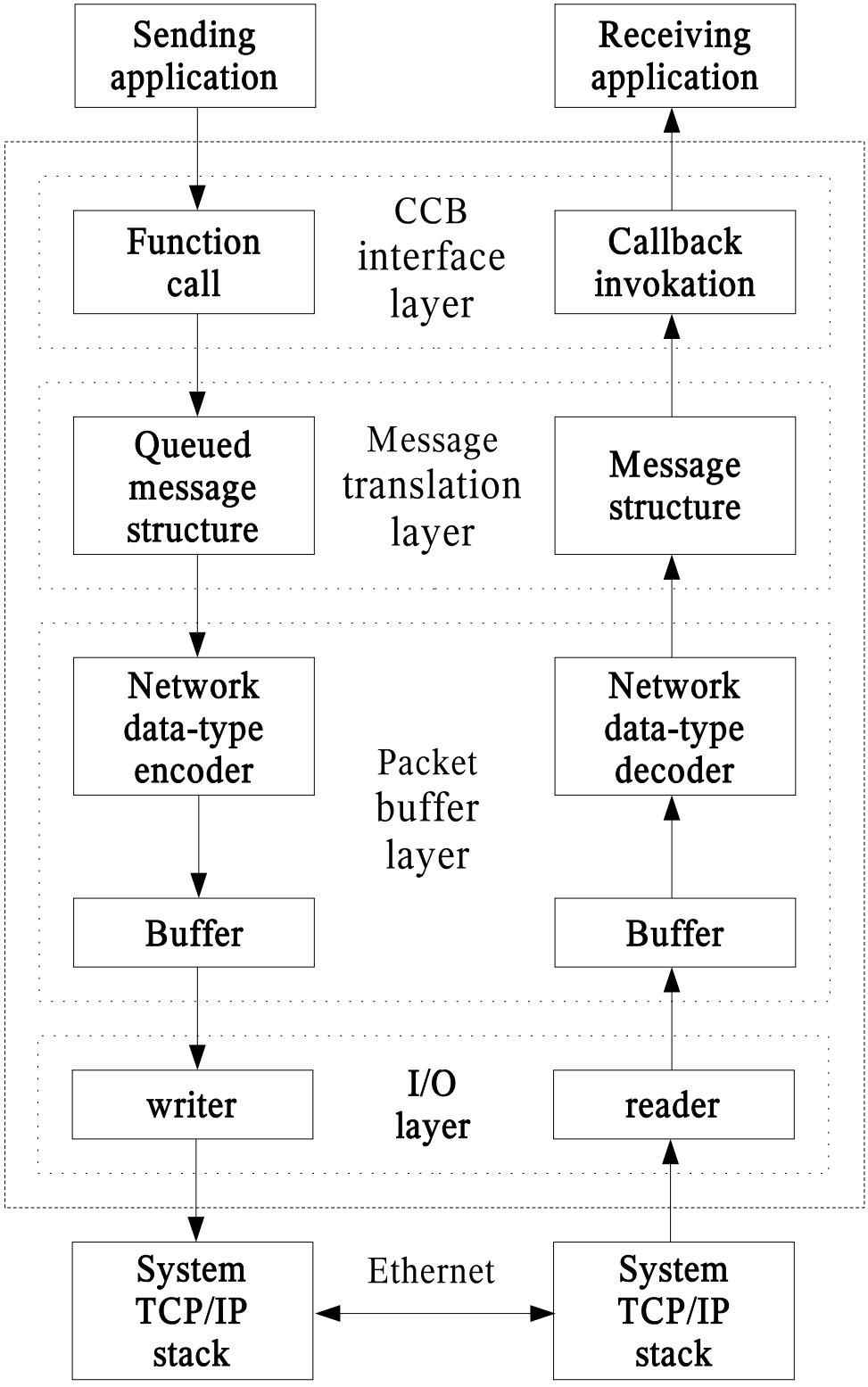

Figure 3.1: The CCB communications stack

3.1 The message translation layer

3.1.1 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.

3.1.2 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	typedef name	Network data-type
CCB_NET_BYTE	signed char	CCBNetByte	8-bit signed integer
CCB_NET_UBYTE	unsigned char	CCBNetUByte	8-bit unsigned integer
CCB_NET_SHORT	short	CCBNetShort	16-bit signed integer
CCB_NET_USHORT	unsigned short	CCBNetUShort	16-bit unsigned integer
CCB_NET_LONG	long	CCBNetLong	32-bit signed integer
CCB_NET_ULONG	unsigned long	CCBNetULong	32-bit unsigned integer
CCB_NET_FLOAT	float	CCBNetFloat	32-bit floating point
CCB_NET_DOUBLE	double	CCBNetDouble	64-bit floating point

Note that all integer types are transfered over the network in big-endian, 2's-complement format, and that the two floating point data-types are transfered in big-endian IEEE-754 format.

3.1.3 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 {
  CCBNetLong 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.

3.1.4 Some example message structures

To see how the contents of message structures are described to the translation-layer of the communications library, consider the following two example message structures, called CCBExampleMsg1, and CCBExampleMsg2:

```
#define SDIM 20;      /* The size of the example string member */
                      /*  in the following message structure. */

typedef struct {              /* Example message structure 1 */
  CCNNetMsg base;             /* The message identification header */
  CCBNetUByte string[SDIM];   /* A string to be transmitted */
  CCBNetUShort slen;          /* strlen(string) */
} CCBExampleMsg1;

typedef struct {              /* Example message structure 2 */
  CCNNetMsg base;             /* The message identification header */
  CCBNetULong foo;            /* A <= 32-bit unsigned number */
} CCBExampleMsg2;
```

3.1.5 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 */
  NetDataType 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 "ccbnetcoms.h"

/* The description of the members of CCBExampleMsg1 */

static const CCBNetMsgMember ccb_example_msg1_members[] = {
  {"string", offsetof(CCBExampleMsg1, string),      CCB_NET_UBYTE, 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},
};
```

3.1.6 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 CCBNetMsg::type member of the 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 CCBNetMsg::type member of the 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.

3.2 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.

3.2.1 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 CCBControl-CmdType enumeration serves this role for outgoing messages on the control link.

```
typedef enum {
  CCB_PHASE_SWITCH_CNF, /* A phase-switch config command */
  CCB_CAL_DIODE_CNF,    /* A cal-diode config command */
  CCB_TELEMETRY_CNF,    /* A telemetry config command */
  CCB_TIMING_CNF,       /* An timing config command */
  CCB_START_SCAN_CMD,   /* A start-scan command */
  CCB_STOP_SCAN_CMD,    /* A stop-scan command */
  CCB_RESET_CMD,        /* A reset command */
  CCB_AWAKEN_CMD,       /* An awaken command */
  CCB_STANDBY_CMD,      /* A standby command */
  CCB_TEST_LINK_CMD,    /* a test-link command */
  CCB_CHECK_STATUS_CMD, /* a check-status command */
  CCB_SHUTDOWN_CMD,     /* A computer shutdown command */
  CCB_REBOOT_CMD,       /* A computer reboot command */
} CCBControlCmdType;
```

As documented in the description of the ccb_cmd_error_callback() function, and evidenced by the fact that all of the public functions for queuing control messages, have an id argument, all outgoing control 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 IDs used by the library. All outgoing control messages thus have two members in common, the mandatory CCBNetMsg initial member of all CCB network messages, which contains the internal message-type identifier of the library, and a manager-provided message identifier. To allow generic access to these two common members by the CCB interface layer, regardless of control-message type, they are aggregated into a CCBCtrlMsgHeader structure, which is the first member of all outgoing control-message structures.

```
typedef struct {
  CCBNetMsg base;   /* The initial member of all messages */
  CCBNetLong id;    /* The manager's identifier of the */
                    /*  parent message. */
} CCBCtrlMsgHeader;
```

Since all message structures start with a CCBCtrlMsgHeader member, whose first member is a CCBNetMsg object, a pointer to the head.base member of the following union of all outgoing control-message structures can portably be used to exchange any of these messages between the CCB-interface layer and message-translation layer of the communications library. The actual type of message passed in this way can be determined from the type member of the CCBNetMsg object.

```
typedef union {
  CCBCtrlMsgHeader head;       /* The common control message header */
  CCBPhaseSwitchCnf phase_cnf; /* head.base.type=CCB_PHASE_SWITCH_CNF */
  CCBCalDiodeCnf diode_cnf;    /* head.base.type=CCB_CAL_DIODE_CNF */
  CCBTelemetryCnf telem_cnf;   /* head.base.type=CCB_TELEMETRY_CNF */
  CCBTimingCnf timing_cnf;     /* head.base.type=CCB_TIMING_CNF */
  CCBStartScanCmd start_scan;  /* head.base.type=CCB_START_SCAN_CMD */
  CCBStopScanCmd stop_scan;    /* head.base.type=CCB_STOP_SCAN_CMD */
  CCBResetCmd reset;           /* head.base.type=CCB_RESET_CMD */
  CCBAwakenCmd awaken;         /* head.base.type=CCB_AWAKEN_CMD */
  CCBStandbyCmd standby;       /* head.base.type=CCB_STANDBY_CMD */
  CCBTestLinkCmd test_link;    /* head.base.type=CCB_TEST_LINK_CMD */
  CCBCheckStatusCmd chk_status;/* head.base.type=CCB_CHECK_STATUS_CMD */
  CCBShutdownCmd shutdown;     /* head.base.type=CCB_SHUTDOWN_CMD */
  CCBRebootCmd reboot;         /* head.base.type=CCB_REBOOT_CMD */
} CCBControlMessage;
```

CCBPhaseSwitchCnf – The phase-switching configuration command

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

```
typedef struct {
  CCBCtrlMsgHeader head;         /* head.base.type=CCB_PHASE_SWITCH_CNF */
  CCBNetUShort active_switches;  /* Which switches are active? */
  CCBNetUShort driven_switches;  /* Which switches are driven? */
  CCBNetUShort initial_states;   /* Which switches start closed? */
  CCBNetUShort samp_per_state;   /* Samples per phase-switch state */
} CCBPhaseSwitchCnf;
```

CCBCalDiodeCnf – The calibration diode configuration command

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

```
typedef struct {
  CCBCtrlMsgHeader head;              /* head.base.type=CCB_CAL_DIODE_CNF*/
  CCBNetUShort ncal;                  /* The number of cal steps */
  CCBNetUShort driven_diodes;         /* The set of driven cal diodes */
  CCBNetUShort diode_a[CCB_MAX_NCAL];/* The ncal states of diode A */
  CCBNetUShort diode_b[CCB_MAX_NCAL];/* The ncal states of diode A */
  CCBNetULong ninteg[CCB_MAX_NCAL];  /* The duration of each step, */
                                      /*  (number of integrations) */
} CCBCalDiodeCnf;
```

CCBTelemetryCnf – The telemetry configuration command

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

```
typedef struct {
  CCBCtrlMsgHeader head;            /* head.base.type=CCB_TELEMETRY_CNF */
  CCBNetUShort integ_period;        /* The integration period */
  CCBNetUShort monitor_interval;    /* The monitoring update interval */
  CCBNetUShort stream_selection;    /* The set of desired data streams */
} CCBTelemetryCnf;
```

CCBTimingCnf – The acquisition-timing configuration command

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

```
typedef struct {
  CCBCtrlMsgHeader head;          /* head.base.type=CCB_TIMING_CNF */
  CCBNetUShort sample_dt;         /* The duration of an A/D sample */
  CCBNetUShort phase_switch_dt;   /* The settling time of the phase */
                                  /*  switches. */
  CCBNetUShort analog_reset_dt;   /* The amount of time needed to */
                                  /*  reset the analog integrators. */
  CCBNetULong diode_rise_dt;      /* The rise time of a cal diode */
  CCBNetULong diode_fall_dt;      /* The fall time of a cal diode */
} CCBTimingCnf;
```

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 {
  CCBCtrlMsgHeader head;     /* head.base.type=CCB_START_SCAN_CMD */
  CCBNetULong date;          /* The MJD UTC day number */
  CCBNetULong tod;           /* The time of day (ms 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 {
  CCBCtrlMsgHeader head;     /* head.base.type=CCB_STOP_SCAN_CMD */
} CCBStopScanCmd;
```

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 {
  CCBCtrlMsgHeader head;     /* head.base.type=CCB_RESET_CMD */
} CCBResetCmd;
```

CCBStandbyCmd – The standby command

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

```
typedef struct {
  CCBCtrlMsgHeader head;         /* head.base.type=CCB_STANDBY_CMD */
  CCBNetUShort stream_mask;      /* The telemetry selection mask */
} CCBStandbyCmd;
```

CCBAwakenCmd – The awaken command

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

```
typedef struct {
  CCBCtrlMsgHeader head;          /* head.base.type=CCB_AWAKEN_CMD */
} CCBAwakenCmd;
```

CCBTestLinkCmd – The test-link command

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

```
typedef struct {
  CCBCtrlMsgHeader head;          /* head.base.type=CCB_TEST_LINK_CMD */
} CCBTestLinkCmd;
```

CCBCheckStatusCmd – The check-status command

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

```
typedef struct {
  CCBCtrlMsgHeader head;          /* head.base.type=CCB_CHECK_STATUS_CMD */
} CCBCheckStatusCmd;
```

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 {
  CCBCtrlMsgHeader 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 {
  CCBCtrlMsgHeader head;          /* head.base.type=CCB_REBOOT_CMD */
} CCBRebootCmd;
```

3.2.2 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_CTRL_LINK_REPLY, /* A test-link reply */
  CCB_STATUS_REPLY,   /* A check-status reply */
} CCBControlReplyType;
```

Since all message structures start with a CCBNetMsg member, a pointer to the base member of the following union of all incoming control-link message structures can portably be used to exchange any control-link reply message between the internal layers of the library, with the base.type member of the union being used to determine what type of message is actually being passed.

```
typedef union {
  CCBNetMsg base;                /* The base-class of all messages */
  CCBCtrlLinkReply ctrl_link;    /* base.type = CCB_CTRL_LINK_REPLY */
  CCBStatusReply status,         /* base.type = CCB_STATUS_REPLY */
} CCBControlReplyMessage;
```

CCBCtrlLinkReply – **A reply to a** test-link **command**

Callback functions registered with ccb_ctrl_link_reply_callback() are invoked whenever a message structure of the following type is received by the communications library.

```
typedef struct {
  CCBNetMsg base;                 /* base.type=CCB_CTRL_LINK_REPLY */
} CCBCtrlLinkReply;
```

CCBCtrlStatusReply – **A reply to a** check-status **command**

Callback functions registered with ccb_ctrl_status_reply_callback() are invoked whenever a message structure of the following type is received by the communications library.

```
typedef struct {
  CCBNetMsg base;                /* base.type=CCB_STATUS_REPLY */
  CCBNetUlong status;            /* The status of the CCB */
} CCBCtrlStatusReply;
```

3.2.3 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_TEL_LINK_REPLY /* A reply to a test-link 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 */
  CCBNetULong date;       /* The MJD UTC day number */
  CCBNetULong tod;        /* The time of day (millisec since 0H UTC) */
  CCBNetULong scan;       /* The sequential ID of the parent scan */
} CCBTelemetryHeader;
```

Note that the obligatory CCBNetMsg member of all network messages is the first member of this structure. The remaining members are interpreted as documented earlier for the initial members of all telemetry callback functions.

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 */
  CCBIntegData integ;          /* An integration data message */
  CCBMonitorData monitor;      /* A monitor data message */
  CCBLogData log;              /* A log message */
  CCBTelemLinkReply link_ok;   /* A reply to a test-link command */
} CCBTelemetryMessage;
```

The data-structures within this union, are declared as follows.

CCBIntegData – Integration data messages

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

```
enum {CCB_MAX_INTEG=64};  /* The maximum number of total-power */
                          /* measurements from any instrument */
typedef struct {
  CCBTelemetryHeader head;         /* head.base.type=CCB_INTEG_MSG */
  CCBNetULong id;                  /* The integration ID */
  CCBNetULong data[CCB_MAX_INTEG]; /* The integrated data */
} CCBIntegData;
```

CCBMonitorData – Monitor data messages

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

```
enum {CCB_MAX_MONITOR=?}; /* The maximum number of monitoring */
                          /* measurements from any instrument */

typedef struct {
  CCBTelemetryHeader head;           /* head.base.type=CCB_MONITOR_MSG */
  CCBNetULong id;                    /* The monitor ID */
  CCBNetULong data[CCB_MAX_MONITOR]; /* The monitor data */
} CCBMonitorData;
```

CCBLogData – **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.

```
enum {CCB_MAX_LOG=?};     /* The maximum length of a log message */

typedef struct {
  CCBTelemetryHeader head;          /* head.base.type=CCB_LOG_MSG */
  unsigned char msg[CCB_MAX_LOG];  /* The message to be logged */
} CCBLogData;
```

CCBTelemLinkReply – **A reply to a** test-link **command**

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

```
typedef struct {
  CCBTelemetryHeader head;    /* head.base.type=CCB_TEL_LINK_REPLY */
} CCBTelemLinkReply;
```

3.3 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_send_control_msgs(). While ccb_client_send_control_msgs() 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_send_control_msgs() again when output again becomes possible, so that it can resume sending the current message.

3.4 Receiving network messages

As already documented, messages are read from the telemetry port of the server by calling ccb_client_recv_telemetry_msgs(). 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-_recv_telemetry_msgs() 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_recv_telemetry_msgs() 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_recv_telemetry_msgs(), 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 ccb_client_recv_control_msgs() is called in place of ccb_client_recv_telemetry_msgs() and different callback functions are called to deliver messages to the manager.