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

Martin Shepherd, California Institute of Technology

January 9, 2003

This page intentionally left blank.

Abstract

This document documents the communications interface between the Ygor manager process and the Linux server of the Caltech Continuum Backend (CCB).

Contents

List of Figures

1 Connection establishment

The CCB server process has two TCP/IP ports, one for synchronous control transactions, and one for asynchronous data delivery.

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 a shared C header file, via the C macro CCB_CONTROL_PORT.

1.1 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, thus simplifying the process of authorizing access to all computers within a given domain or sub-domain.

If the connecting manager isn't connecting from one of the 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 Protocol authentication

Although the server and the manager use a common communications library and common message-definition files, there is potential for these files to get out of sync between the two systems. In fact there are potentially 4 separate components that could become out of sync, each one having been compiled against a different version of the public header file which defines the message structures that are exchanged with the communications library. These are the manager process, the server process and, if the communications library is made into a shared library, the two copies of this shared library on the two systems. It is vitally important that failure to recompile any of these components to accomodate a revised version of the messaging structures, be detected by the combined system, the moment that anybody attempts to use it.

Rather than have a manually maintained version number, the way that this is done in the CCB communications library is to have the library makefile compute a CRC-32 checksum of the its public C header file, and record this number as a macro assignment at the top of the exported copy of this header file. This is done as follows.

```
$(INCLUDE)/ccbnetcoms.h: ccbnetcoms.h
```

```
echo "CCB_HEADER_CKSUM `cksum ccbnetcoms.h | awk '{print $1}'`" > $@
cat ccbnetcoms.h >> $@
```

This checksum can then be interchanged between the 4 components to check that they were all compiled against the same version of the public header file.

1.2.1 Program vs library consistency check

The server checks that it is compiled against a compatible version of the library by comparing its copy of the checksum with that returned by the library function, ccb_library_cksum(). If the comparison indicates a problem, the CCB server aborts. The manager follows the same procedure to verify compatibility with its copy of the library.

1.2.2 Server vs client consistency check

As soon as a successful connection has been established by the manager to the CCB server, the CCB manager is required to pass its copy of the header-file checksum to the CCB server in a 32-bit network-byte order unsigned integer. If this number doesn't match the CCB server's copy of the checksum, the CCB server closes the TCP/IP connection, to reject it. Otherwise the CCB server responds by sending a CCBConnectionAck message to the manager. This tells the manager that the control connection has been accepted.

1.3 Telemetry connection establishment

Once a control connection to the CCB server has been accepted by the server, the CCB server expects the manager to also connect to its telemetry server port, whose number is provided in a C macro called CCB_TELEMETRY_PORT. This is a separate TCP/IP port, which is used by the CCB server to send asynchronous messages, such as the integration data, monitoring data and log messages, to the manager. Only telemetry connection whose originating IP addresses match that of the manager's control connection, are accepted. Telemetry connections are closed automatically by the CCB server when the manager closes its control connection.

1.4 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.

2 CCB Control messages

This section describes the messages that are sent to the control port of the CCB server.

Note that the structures described in this section are only used to pass these messages to and from the communications library, and don't denote the form in which data are sent via TCP/IP. That will be described later.

All control messages, start with an initial CCBControlHeader member. This is defined as follows.

```
typedef struct {
  CCBControlType type;    /* The type of control message */
} CCBControlHeader;
```

The members of this structure are interpreted as follows.

- **Message type** (CCBControlHeader::type)

  This member identifies the type of control message encoded in the parent structure. This is expressed using an enumeration of the following type.

```
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 */
} CCBControlType;
```

Since all control message structures share the same initial header structure, a pointer to the following union of all message types can portably be used to pass messages of any type to and from functions of the communications library, with the header member of the union being used to determine what type of message is actually being passed.

```
typedef union {
  CCBControlHeader header;   /* The shared control header */
```

```
  CCBPhaseSwitchCnf,          /* header.type == CCB_PHASE_SWITCH_CNF */
  CCBCalDiodeCnf,             /* header.type == CCB_CAL_DIODE_CNF */
  CCBTelemetryCnf,            /* header.type == CCB_TELEMETRY_CNF */
  CCBTimingCnf,               /* header.type == CCB_TIMING_CNF */
  CCBStartScanCmd,            /* header.type == CCB_START_SCAN_CMD */
  CCBStopScanCmd,             /* header.type == CCB_STOP_SCAN_CMD */
  CCBResetCmd,                /* header.type == CCB_RESET_CMD */
  CCBAwakenCmd,               /* header.type == CCB_AWAKEN_CMD */
  CCBStandbyCmd               /* header.type == CCB_STANDBY_CMD */
} CCBControlMessage;
```

2.1 CCB configuration messages

This sub-section describes the subset of control messages that configure the parameters of the next scan. The modified configuration doesn't take affect until the next start-scan or stop-scan message is sent.

2.1.1 The phase-switching configuration command

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 1.

In this diagram the supported cycles are arranged into columns of cycles of the same initial switch states, and rows of cycles with a particular combination of active switches. Note that whereas the number of A/D samples per measurement 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.

Phase-switching configuration parameters are passed to and from the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;         /* The control-message header */
  unsigned short active_switches;  /* Which switches are active? */
  unsigned short driven_switches;  /* Which switches are driven? */
  unsigned short initial_states;   /* Which switches start closed? */
  unsigned short samp_per_state;   /* Samples per phase-switch state */
} CCBPhaseSwitchCnf;
```

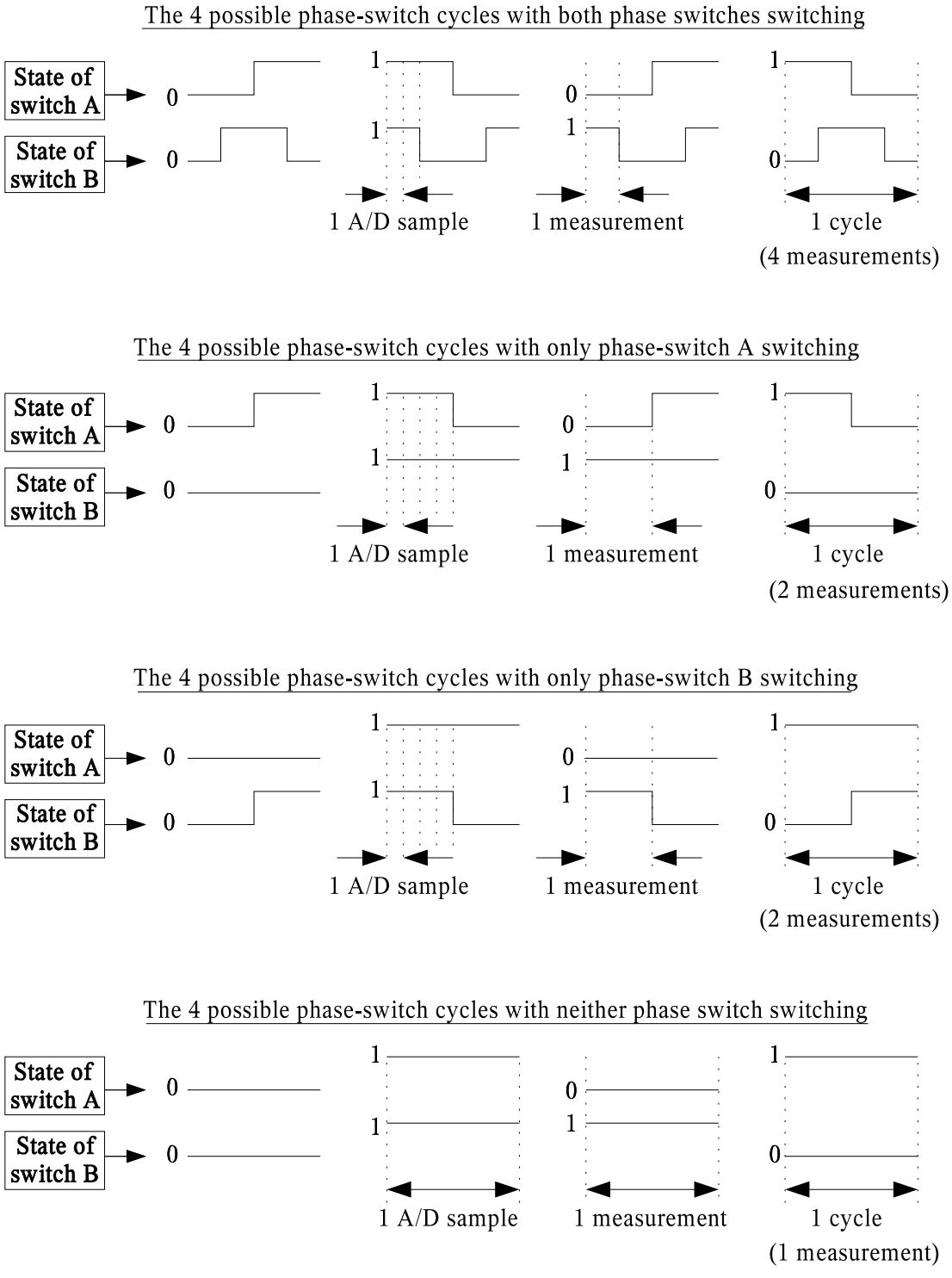

Figure 1: Example phase-switching cycles

The members of this structure are interpreted as follows.

- **Which switches are active?** (CCBPhaseSwitchCnf::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 CCBPhaseSwitchCnf::initial_states parameter. A value of 0 indicates that the corresponding switch should be held in its initial position, whereas a value of 1 indicates that it should be switched according to the current switching mode.

- **Which switches are driven?** (CCBPhaseSwitchCnf::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 value of 1 causes the corresponding line to be driven. A value of zero causes it to be placed in a high impedance state.

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

  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 CCBPhaseSwitchCnf::active_switches parameter are held in these states throughout each cycle.

- **Samples per phase-switch state** (CCBPhaseSwitchCnf::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

2.1.2 Calibration diode configuration

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.

Calibration diode configuration parameters are passed to and from the communications library using structures of the following type.

```
enum {CCB_MAX_NCAL=?};                  /* The max number of cal steps */

typedef struct {
  CCBControlHeader header;               /* The control-message header */
  unsigned short ncal;                   /* The number of cal steps */
  unsigned short driven_diodes;          /* The set of driven cal diodes */
  unsigned short diode_a[CCB_MAX_NCAL]; /* The ncal states of diode A */
  unsigned short diode_b[CCB_MAX_NCAL]; /* The ncal states of diode A */
  unsigned long ninteg[CCB_MAX_NCAL];   /* The duration of each step, */
                                         /*  (number of integrations) */
} CCBCalDiodeCnf;
```

The members of this structure are interpreted as follows.

- **The number of calibration steps** (CCBCalDiodeCnf::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** (CCBCalDiodeCnf::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** CCBCalDiodeCnf::ncal **states of diode A** (CCBCalDiodeCnf::diode_a)

  The first CCBCalDiodeCnf::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** CCBCalDiodeCnf::ncal **states of diode B** (CCBCalDiodeCnf::diode_b)

  The first CCBCalDiodeCnf::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** (CCBCalDiodeCnf::ninteg)

Each of the first CCBCalDiodeCnf::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.

2.1.3 Telemetry configuration

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. Commands of this type are passed to and from the communications library, using structures of the following type.

```
typedef struct {
  CCBControlHeader header;           /* The control-message header */
  unsigned short integ_period;       /* The integration period */
  unsigned short monitor_interval;   /* The monitoring update interval */
  unsigned short stream_selection;   /* The set of desired data streams */
} CCBTelemetryCnf;
```

The members of this structure are interpreted as follows.

- **The integration period** (CCBTelemetryCnf::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** (CCBTelemetryCnf::monitor_interval)

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

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

  This parameter contains a bit-wise union of CCBTelemetryStream enumerators, used to specify which, of the integration data stream and the monitor data stream, 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.

2.1.4 Acquisition timing configuration

The acquisition timing configuration specifies the durations of timers in the CCB hardware. Timing configuration parameters are passed to and from the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;          /* The control-message header */
  unsigned short sample_dt;         /* The duration of an A/D sample */
  unsigned short phase_switch_dt;   /* The settling time of the phase */
                                    /*  switches. */
  unsigned short analog_reset_dt;   /* The amount of time needed to */
                                    /*  reset the analog integrators. */
  unsigned long diode_rise_dt;      /* The rise time of a cal diode */
  unsigned long diode_fall_dt;      /* The fall time of a cal diode */
} CCBTimingCnf;
```

The members of this structure are interpreted as follows.

- **A/D sample interval** (CCBTimingCnf::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.

- **Phase-switch blanking interval** (CCBTimingCnf::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.

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

  This specifies how long to wait for the analog integrator to reset before starting to integrate a new sample. This is expected to be around 1μs.

- **Calibration diode rise time** (CCBTimingCnf::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.

- **Calibration diode fall time** (CCBTimingCnf::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.

2.2 CCB control commands

The CCB control commands described in this section are acted on as soon as they are received by the CCB server. 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.
awaken	Take control of the receiver.
standby	Relinquish control of the receiver.

The following subsections elaborate on these commands.

2.2.1 start-scan

The start-scan command causes a new scan to be started on the 1-PPS boundary specified by the CCBStartScanCmd::date and CCBStartScanCmd::tod parameters. If the command is received less than 1 integration time before the specified time, the new scan is not be started on the desired 1-PPS, but instead starts on the next 1-PPS that follows the current integration.

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 exchanged with the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;   /* The control-message header */
  unsigned long date;        /* The MJD UTC day number */
  unsigned long tod;         /* The time of day (ms since 0H UTC) */
} CCBStartScanCmd;
```

The members of this structure are interpreted as follows.

- **The date at which to start the scan** (CCBStartScanCmd::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** (CCBStartScanCmd::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 CCBStartScanCmd::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.

2.2.2 stop-scan

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.

stop-scan commands are exchanged with the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;   /* The control-message header */
} CCBStopScanCmd;
```

2.2.3 reset

This command resets both the hardware and the device driver.

On receiving this command, the device driver tells 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 and tells it to start a new scan.

reset commands are exchanged with the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;   /* The control-message header */
} CCBResetCmd;
```

2.2.4 standby

On receiving this command, the CCB server tells the hardware to stop driving any of the receiver control lines.

standby commands are exchanged with the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;          /* The control-message header */
  unsigned short stream_mask;       /* The telemetry selection mask */
} CCBStandbyCmd;
```

The members of this structure are interpreted as follows.

- **The telemetry data-stream selection mask** (CCBStandbyCmd::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).

2.2.5 awaken

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 be started by this command.

awaken commands are exchanged with the communications library using structures of the following type.

```
typedef struct {
  CCBControlHeader header;   /* The control-message header */
} CCBAwakenCmd;
```

3 Asynchronous telemetry

Integrated data, monitoring data and log messages are sent to the manager via the server's telemetry link. In this section, the structures that are used to pass such messages to and from the communications library are described. Note that these structures are not the form in which data are sent via TCP/IP. That will be described later.

The first member of all telemetry message structures is a CCBTelemetryHeader structure, which is defined as follows.

```
typedef struct {
  CCBTelemetryType type;  /* The type of telemetry message */
  unsigned long date;     /* The MJD UTC day number */
  unsigned long tod;      /* The time of day (ms since 0H UTC) */
  unsigned long scan;     /* The sequential ID of the parent scan */
} CCBTelemetryHeader;
```

The members of this structure are interpreted as follows.

- **Message type** (CCBTelemetryHeader::type)

  This member identifies the type of telemetry message encoded in the parent structure. This is expressed using an enumeration of the following type.

  ```
  typedef enum {
    CCB_INTEG_MSG,     /* An integration data message */
    CCB_MONITOR_MSG,   /* A monitoring data message */
    CCB_LOG_MSG        /* A log message */
  } CCBTelemetryType;
  ```

- **The date at which the message was created** (CCBTelemetryHeader::date)

  This is the date at which the telemetry message was assembled. The date is 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 the message was created** (CCBTelemetryHeader::tod)

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

- **The sequential number of the parent scan** (CCBTelemetryHeader::scan)

  This identifies the scan during which the message was generated. Scans are identified by the value of an integer counter in the device driver, which is initialized to zero whenever the device driver is loaded, and thereafter incremented by one whenever a new scan is started. In other words, whenever a new scan is started by either a start-scan or stop-scan command, the device driver increments this counter by one. Placing this in each data packet allows the manager to associate each integration with the corresponding scan configuration parameters that were established before the new scan was started.

Since all telemetry message structures share the same initial header structure, a pointer to the following union of all message types can portably be used to pass messages of any type to and from the communications library, with the header member of the union being used to determine what type of message is actually stored there.

```
typedef union {
  CCBTelemetryHeader header;   /* The common telemetry header */
  CCBIntegData integ;          /* An integration data message */
  CCBMonitorData monitor;      /* A monitor data message */
  CCBLogData log;              /* A log message */
} CCBTelemetryMessage;
```

3.1 Integration data messages

Integrated data are sent to the manager, at the end of each integration. The data are passed to and from the communications library in a structure of the following form.

```
enum {CCB_MAX_INTEG=64};  /* The maximum number of total-power */
                          /* measurements from any instrument */
typedef struct {
  CCBTelemetryHeader header;         /* The telemetry message header */
```

```
  unsigned long id;                   /* The integration ID */
  unsigned long data[CCB_MAX_INTEG]; /* The integrated data */
} CCBIntegData;
```

The members of this structure are interpreted as follows.

- **The common message header** (CCBIntegData::header)

  This member, which is common to all messages sent from the server to the manager, identifies the type of message being sent.

- **Integration ID** (CCBIntegData::integ)

  This identifies the originating integration, by recording the value of the integration counter in the device driver. This counter is reset to zero at the start of each new scan, and thereafter incremented by 1 whenever a new integration interrupt is received from the hardware. The manager can use this to check for missing integrations.

- **Integrated data** (CCBIntegData::data)

  This is a sequence of 64 32-bit unsigned integer integration values.

3.2 Monitor data messages

Instrumental monitoring data are sent to the manager over the telemetry link, at the end of every CCBTelemetryCnf::monitor_interval'th integration. The data are passed to and from the communications library in a structure of the following form.

```
enum {CCB_MAX_MONITOR=?}; /* The maximum number of monitoring */
                          /* measurements from any instrument */
typedef struct {
  CCBTelemetryHeader header;           /* The telemetry message header */
  unsigned long id;                    /* The monitor ID */
  unsigned long data[CCB_MAX_MONITOR]; /* The monitor data */
} CCBMonitorData;
```

The members of this structure are interpreted as follows.

- **The common message header** (CCBMonitorData::header)

  This member, which is common to all messages sent from the server to the manager, identifies the type of message being sent.

- **Monitor update ID** (CCBMonitorData::integ)

  This is the value of another integer counter in the device driver. This counter is reset to zero at the start of each new scan, and thereafter incremented by 1 whenever a new integration interrupt is received from the hardware. The manager can use this to check for missing integrations.

- **Monitor data** (CCBMonitorData::data)

  This is a sequence of a currently unknown number of 32-bit unsigned integer monitoring measurements.

3.3 CCB log messages

The server sends both error and informational messages to the manager, to be logged. They are passed to and from the communications library in a structure of the following form.

```
enum {CCB_MAX_LOG=?};     /* The maximum number of total-power */
                          /* measurements from any instrument */
typedef struct {
  CCBTelemetryHeader header;       /* The telemetry message header */
  unsigned char msg[CCB_MAX_LOG];  /* The message to be logged */
} CCBLogData;
```

The members of this structure are interpreted as follows.

- **The telemetry message header** (CCBLogData::header)

  This member, which is common to all telemetry messages, identifies the type of message being sent.

- **The log message** (CCBLogData::msg)

  Log messages are transfered as normal C-style strings, terminated by a `'\0'`. Messages that won't fit within the CCB_MAX_LOG elements of the msg[] array, are truncated to fit within the msg[] array, and terminated with a `'\0'` character in the last element of this array.