

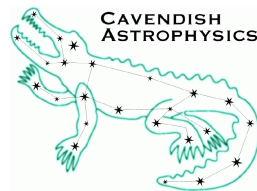
MRO Delay Line

Workstation Software Functional Description

INT-406-VEN-0103

The Cambridge Delay Line Team

rev 1.3
21 January 2010



Cavendish Laboratory
Madingley Road
Cambridge CB3 0HE
UK

Change Record

Revision	Date	Authors	Changes
0.1	2007-07-05	JSY	Initial version
0.2	2007-07-09	JSY	Input from EBS
0.3	2007-07-10	JSY	Added note about system modes
1.1	2008-01-14	JSY	Revised for dual-threaded implementation
1.2	2010-01-12	JSY	Changed to reflect alterations to prototype software since FDR
1.3	2010-01-21	JSY	Modified diagram to show software components for more than one delay line

Objective

To describe the design and implementation of the workstation software.

Scope of this document

This document forms part of the documentation for the delay line Final Design Review. It describes the design and implementation of the workstation software.

The overall architecture of the control software system is described in "Control Software Architecture". The workstation software acts as a supervisor for other delay line sub-systems, the software for which is described in separate documents:

- Trolley Software Functional Description
- VME Software Functional Description
- Shear Camera Software Functional Description

Contents

1	Introduction	3
1.1	Development and Execution Environment	3
1.2	Workstation Application Architecture	4
2	User Interface	5
2.1	DGOpdGui: Workstation Graphical User Interface Class	7
2.2	DGSystemGui: System control graphical user interface class	9
2.3	DGStatusDisplay: Status display widget class	9
2.4	ClientDisplay: single-client status display class	9
2.5	DGTermUi: Terminal User Interface Class	11
3	Telemetry Server	11
3.1	Telemetry/Status/Command Logging	12
4	Connection Manager	13
5	System-level Application Programming Interface	13
5.1	DGSystem: Delay Line System Class	14
5.2	State Machine	15
6	Trajectory Calculator	16

1 Introduction

The workstation software provides several distinct functions:

- Provides a user interface for control of each delay line, including:
 - System-level controls
 - Selected subsystem-level controls
 - Live display of status information
- Transmits a trajectory demand (position and velocity as a function of time) for each delay line
- Receives, buffers, and optionally logs status and telemetry information received from other sub-systems over the Ethernet

A separate software package is used for off-line processing and analysis of recorded telemetry. This analysis software is not described any further in this document.

1.1 Development and Execution Environment

The workstation software makes extensive use of GLib – the low-level core library that forms the basis of GTK+ and GNOME. GLib provides abstract data types such as hash tables and linked lists, as well as an event handling system (the use of which is described in the next section) and an object system.

The GLib Object System (GObject for short) is a framework for object-oriented programming in C. The object system supports inheritance, object “properties” (a generic interface for setting/getting object attributes) and a flexible inter-object messaging system. The messaging system works by means of “signals” (nothing to do with Unix signals). A class may define any number of signals, which are emitted on particular instances of the class. Any number of response functions (signal handlers) may be defined, perhaps within other objects – these are invoked by GLib on signal emission.

The workstation graphical user interface is implemented using the GTK+ graphical user interface toolkit, which makes extensive use of the GLib Object System and its signal mechanism.

The workstation code uses two different styles of object-oriented programming. In the first style, objects are implemented as dynamically-allocated C structs returned by a constructor function. Various method functions take a pointer to the struct as the first argument. The alternative style uses the GLib Object System (we have assigned names with a “DG” prefix to classes implemented in this style). The latter allows GLib signals to be

used, but results in somewhat less readable code. Hence we have restricted use of the second style to high-level objects which must handle a number of events.

The workstation computer used to test the prototype delay line is a standard Dell Optiplex GX620 PC with a 3.4GHz Pentium 4 processor, 1 Gigabyte of RAM, and a 250-Gigabyte capacity hard disk.

The workstation clock is kept synchronised with those of the other computers in the delay line system using the NTP protocol.

1.2 Workstation Application Architecture

The functionality outlined above is provided by an event-driven application program running on the workstation computer.

Two alternative versions of this application have been written:

OpdGui System-level Graphical User Interface (GUI); intended for high-level control

Test Controller Text-based interface; intended for subsystem-level control as a development aid

Both applications have the same basic architecture, which we now describe.

The event-driven framework is provided by GLib: when the application is started a number of objects are instantiated (coded in either style), each of which registers one or more event sources (such as input/output watches) within a single instance of the GLib Main Event Loop. The program then enters the main event loop, which runs until the user quits the program.

The OpdGui application reads and parses an ASCII configuration file prior to instantiating the objects and starting the main loop. This configuration file contains a small number of configuration parameters formatted as “[key]=[value]”, organised into several named sections.

There are two exceptions to this:

- the trajectory calculator object (Sec. 6), runs its own Main Event Loop instance in a secondary thread. The only event source associated with this main loop is the periodic task used to transmit trajectory demands to the VME system;
- Cleaning up after the user aborts a logging operation is performed in a separate thread, as this would otherwise block the handling of other events for several seconds in the worst case.

Possible event sources are:

1. Input/output watches: events triggered by activity on an open file, pipe or socket
2. Timeouts: periodic events
3. GLib signals, emitted on the application's component GObject-based objects
4. Idle functions, which run when no higher-priority event is pending

Of these event sources, GLib signals have the unique feature that any number of response functions (signal handlers) may be registered for a particular event (signal). Thus an event triggered by a component object may result in (different) responses from several other objects.

As C has no built-in exception-handling system, programmer and user errors are handled using a locally-written exception library, which is described in the document "An exception-handling system for C software". As in most exception-handling systems, the library allows the detection of errors anywhere in the code and for this error to propagate back through the function-call hierarchy to an appropriate level for handling the error. All the information about a thrown exception is stored in a variable, conventionally called status, which is passed as the last parameter of any function call (in most cases). This variable is a pointer to a structure which contains multiple items of information about the exception, including an error message.

In the case of exceptions thrown during the processing of an event, these are propagated back up to the top-level event handler function that was invoked by the main event loop. If the event handler is part of the user interface code it directly calls a method to report the exception to the user. Other handlers emit a GLib signal provided by the telemetry server (Sec. 3) which invokes a handler within the user interface code. This handler reports the exception to the user. The main event loop is then resumed.

The application component objects, together with their event sources and event handlers are described in the following sections. The application architecture is shown diagrammatically in Figure 1.

The initial objects created are the User Interface (Sec. 2), Telemetry Server (Sec. 3), and Connection Manager (Sec. 4, OpdGui only). Further objects are created and destroyed in response to sub-systems connecting and disconnecting (according to the connection/disconnection protocols described in "Control Software Architecture").

2 User Interface

Two different user interfaces have been implemented. The System-level GUI has a graphical user interface implemented using GTK+, whereas the Test Controller has a text-based interface implemented using the Ncurses library. Both interfaces provide user logging controls (which activate the logging functions provided by the Telemetry Server – see

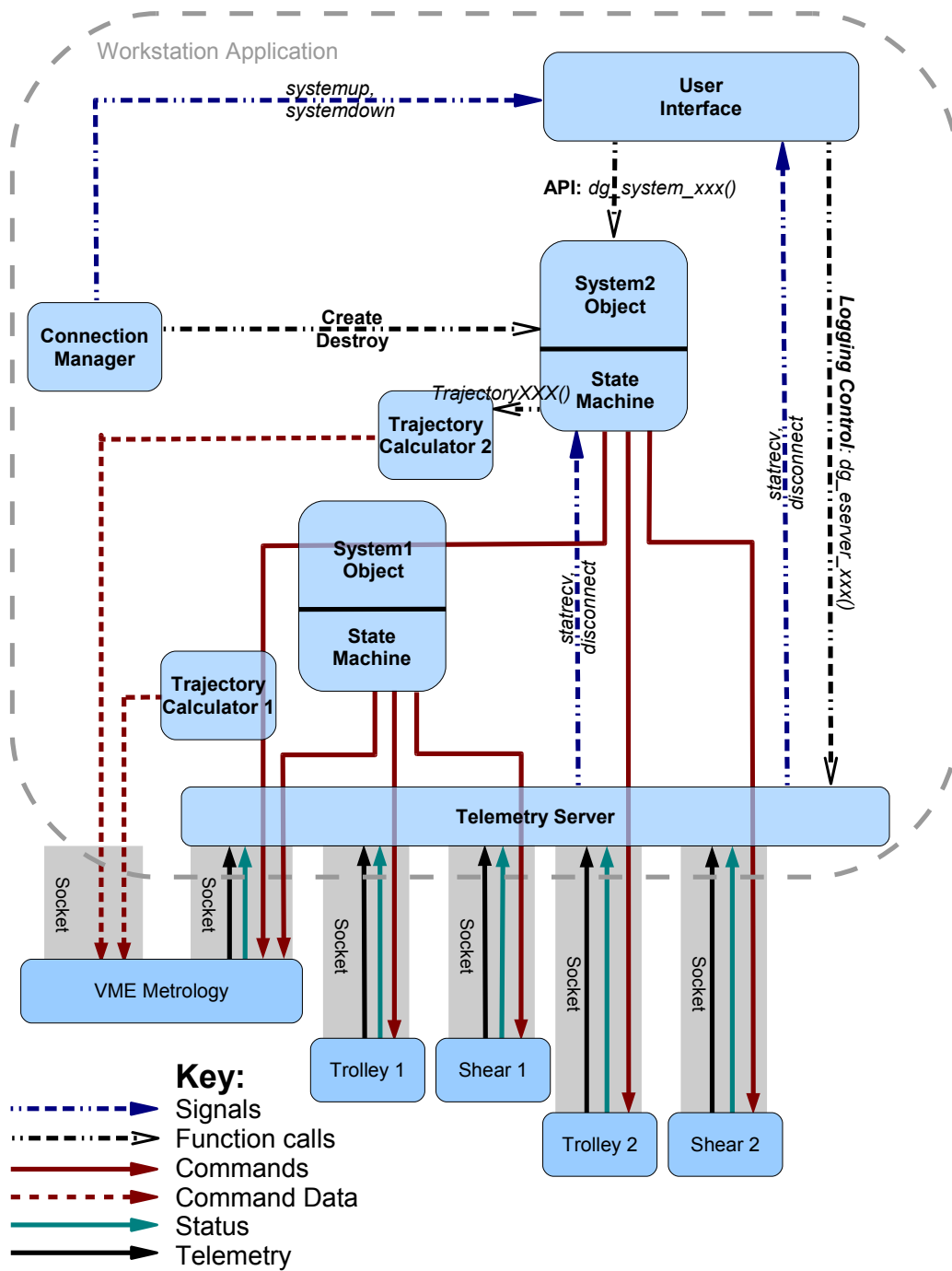


Figure 1: Diagram of workstation application architecture, showing the component objects and the communication links between them, when operating two delay lines. The solid arrows denote TCP/IP network messages, and the dashed arrows intra-application information flows. For clarity, the dashed arrows associated with one of the delay lines have been omitted.

Sec. 3.1) and a real-time display of the status items received from each connected sub-system.

The two interfaces provide different functionality for controlling the delay line. The Test Controller allows the user to type in commands defined by the messaging protocol, which are transmitted to a user-selected sub-system. OpdGui provides graphical controls (buttons and entry fields) for system-level control of delay lines. Each set of these controls (responsible for a single delay line) is created and destroyed in response to GLib signals from the Connection Manager (Sec. 4).

The implementation of OpdGui defines handlers for button click signals which call functions from the system-level Application Programming Interface (API) described in Sec. 5.

The user interfaces are implemented using the GObject-derived classes described in the following sections.

2.1 DGOpdGui: Workstation Graphical User Interface Class

This class implements the graphical user interface for system-level control of multiple delay lines, as well as providing controls for recording of telemetry, status, and shear camera video. The class also displays log/fault messages (above a user-specified verbosity level) that are generated within the workstation software (using the GLib message logging API) or received from delay line sub-systems. The displayed log/fault messages are logged to the same FITS file used to record status and telemetry.

The class provides the main application window (see Figure 2), which has logging controls plus a message display at the bottom and a GtkNotebook widget at the top. The notebook acts as a container for multiple pages of widgets, any one of which may be selected for display by the user. The notebook is exported as an object property so that other objects may add pages – DGStatusDisplay displays its widgets by this mechanism.

When the Connection Manager signals that a delay line system has come up, a user interface for controlling that delay line is created by instantiating a DGSystemGui object. The DGSystemGui's widgets are displayed on a new notebook page (the instance and its widgets will be destroyed when the Connection Manager signals that the delay line is down).

DGOpdGui provides controls to start and stop logging to FITS files. DGSystemGui provides a checkbutton for selecting whether shear camera video should be recorded contemporaneously with status and telemetry logging for that delay line. If the button is checked, DGOpdGui commands the relevant shear sensor to start recording when logging is initiated by the user.

The GTK+ widget layouts used by DGOpdGui and DGSystemGui are stored in XML files generated by the Glade interface design software, and realised using libglade.

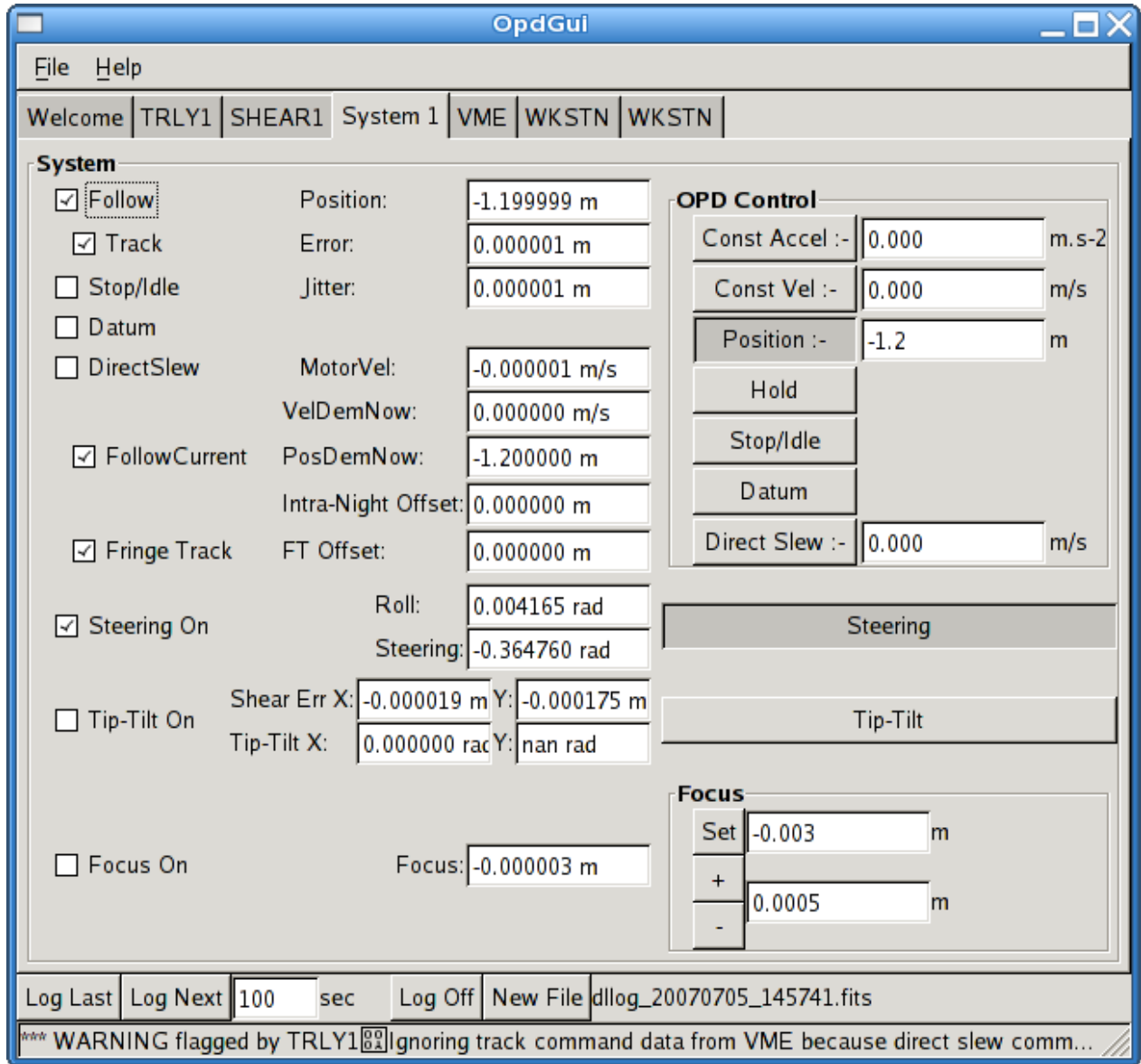


Figure 2: Screenshot of the system-level GUI application OpdGui, showing the system control interface.

2.2 DGSystemGui: System control graphical user interface class

This class provides a graphical user interface for control of a single delay line. The interface incorporates logical groupings of action buttons as well as widgets for displaying system status items. These widgets are packed into a frame widget, which is exposed as an object property so that it can be displayed by the code that instantiates DGSystemGui.

DGSystemGui defines handlers for GTK+ button click signals which call functions from the system-level API (Sec. 5). User requests which are not permitted in the current delay line state are filtered out by the state machines embedded within the API, rather than by the GUI.

The system status display is implemented using ClientDisplay (see below) and uses a pre-defined set of display widgets that are loaded using libglade.

2.3 DGStatusDisplay: Status display widget class

This class implements a graphical display of all status items, grouped by sub-system.

The status items from each connection to the telemetry server are displayed on a separate page of the notebook provided by DGOpdGui (see Figure 3). The status display object responds to signals from the Connection Manager in order to automatically add and remove pages as clients connect and disconnect to/from the server.

The widget layouts are generated automatically using ClientDisplay (see next section) when clients connect, so that the displayed widgets are always consistent with the status items contained in the network messages. Thus clients can be modified to add new status items, and these status items will be displayed without any changes to the workstation software.

2.4 ClientDisplay: single-client status display class

This class is used by DGStatusDisplay and DGSystemGui to optionally create and to update the widgets used to display status from a single delay line subsystem.

A ClientDisplay instance can either create its own widgets or use existing widgets from a libglade widget tree (depending on which constructor is called). In the latter case the widgets must follow a particular naming convention so that the constructor can find the appropriate widget to associate with each status item.

The class provides a method to update the widgets to display a new set of status values.

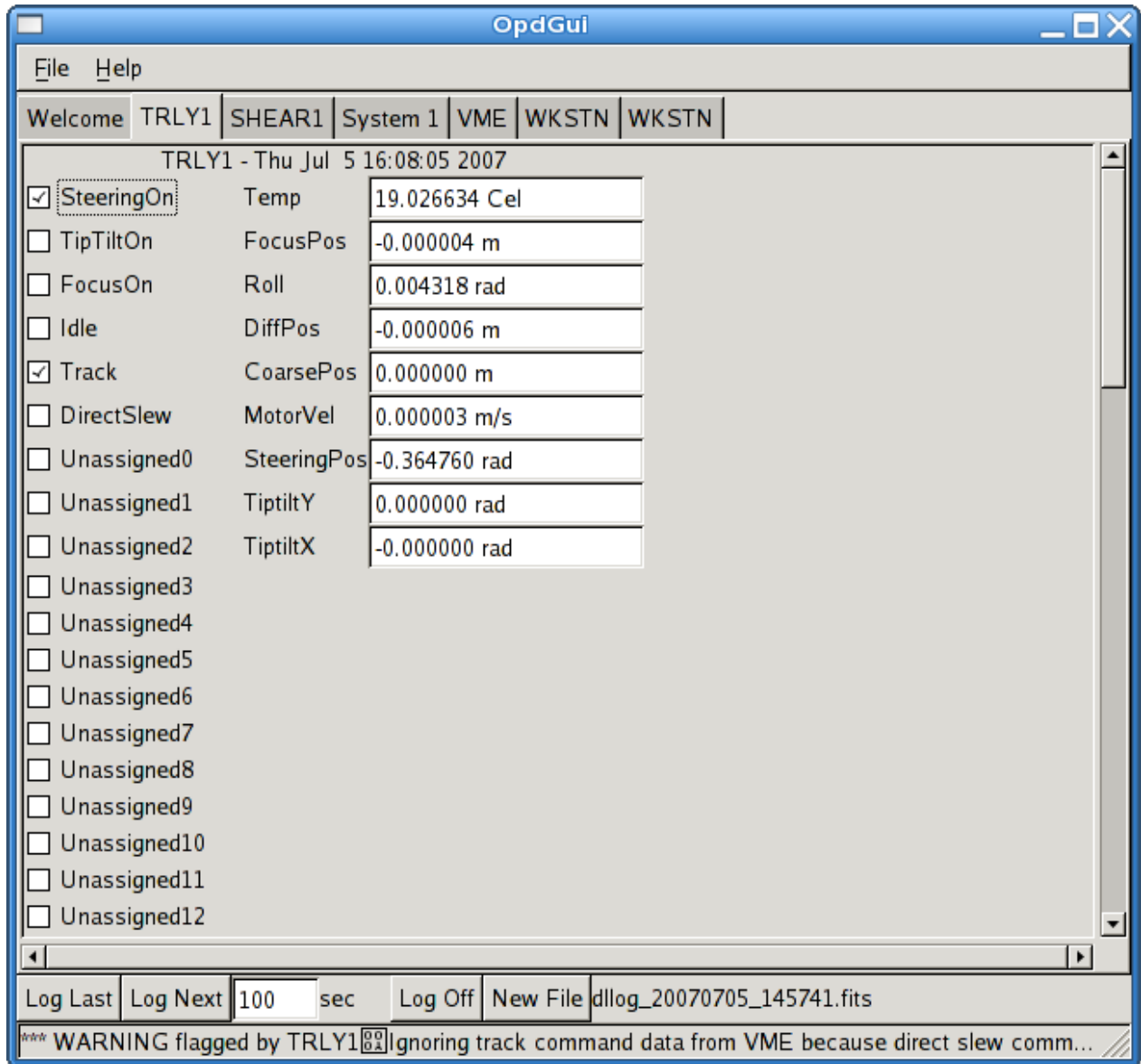


Figure 3: Screenshot of the system-level GUI application OpdGui, showing a sub-system status display.

2.5 DGTermUi: Terminal User Interface Class

This class implements the text-based user interface for subsystem-level control as well as providing a user interface for the logging functionality provided by the Telemetry Server. Keypress events are detected by a GLib timeout function which polls the Ncurses keyboard handler at 50 ms intervals.

3 Telemetry Server

Buffering and logging of status and telemetry messages (see “Control Software Architecture”) received from other delay line sub-systems is handled by a telemetry server object embedded within the application.

Once initialized, the server listens at a pre-arranged TCP/IP port for connection attempts from remote clients. Any number of clients may connect, disconnect, and reconnect as necessary. Once a client has connected, status and telemetry data received from the client are stored in a client-specific set of circular buffers (normally sized to store 100 seconds of data). The server expects a single sequence of status messages (i.e. messages containing the same set of data items) and a single sequence of telemetry messages from each client, and will throw an exception if this is violated. However, messages originating from several clients may arrive on the same socket — this is to allow receipt of messages forwarded from the workstation to another application which also embeds an instance of the telemetry server (we anticipate that this functionality will be used in the production software). When logging (see Sec. 3.1) is activated, data is retrieved from the buffers and written to a disk file.

The connection protocol and message formats supported by the server are described in detail in the document “Control Software Architecture”. Note that TCP/IP socket connections can be made over the “loopback” interface from a computer to itself. This mechanism is used to send system status from each system object (Sec. 5.1) and status and telemetry from the Trajectory Calculator (Sec. 6) to the server.

Two alternative interfaces between the embedding application and the server have been implemented:

E`Server` Embeddable Telemetry Server "class" - uses hook (callback) functions for notification of server events

DG`E`Server GObject-based embeddable Telemetry Server class - uses GLib signals for notification of server events (implemented as a thin wrapper around E`Server`)

The latter interface is used in the workstation application programs. Both interfaces allow the embedding application to:

- Be notified of client connection and disconnection events
- Be notified of message arrival events
- Be notified of exceptions thrown in telemetry server event handlers
- Query which clients are connected
- Activate and deactivate logging to the current FITS file
- Close the current FITS file and open another
- Log a command to the current FITS file
- Retrieve the latest status information from a client
- Access the socket for a client (e.g. to send commands)
- Close the connection to a specified client
- Destroy the server, which breaks the connections to any connected clients

The exception signal can also be used by the code that embeds the telemetry server as a general-purpose means of propagating exceptions that occur in event handlers.

Note that the server does not provide a mechanism for remote clients to retrieve status or telemetry data – this feature is not required for the chosen control software architecture.

3.1 Telemetry/Status/Command Logging

Two flavours of logging have been implemented:

A Posteriori Logging Log previous N seconds (N specified by user), while continuing to buffer incoming telemetry and status.

A Priori Logging Buffer and log next N seconds (N specified by user).

When logging is inactive, incoming telemetry and status is still buffered.

For both flavours, telemetry FITS tables are created on disk when logging commences, sized for N seconds of data. Initially-empty status tables are created in a memory buffer, using a facility provided by the cfitsio library (these will be copied to disk when logging ends). A pair of event sources are set up for each client, to trigger (a) writing of chunks of telemetry data to disk and (b) status to memory. These work slightly differently for the two logging flavours:

A Posteriori Logging An idle function logs one second of data each time it is invoked.

A Priori Logging A 1 Hz timeout function logs all available data (to catch up in case of delays) each time it is invoked.

The server keeps track of how much data has been logged for each client, and when the requested N seconds have been logged the event sources for that client are cancelled. When all logging event sources have been destroyed the status tables are moved from memory to the disk file. This also occurs when a logging operation is cancelled by the user, in which case the server also removes unused reserved space from the telemetry tables. Removal of reserved space and moving of status tables are carried out in a separate thread; while this is running new logging operations are prevented from starting.

Subsequent logging operations record new sets of tables in the same FITS file on disk, until the server receives a request to open a new file.

4 Connection Manager

The Connection Manager facilitates the implementation of a control user interface where the available operations depend on which delay line sub-systems are available.

The manager is implemented as a GLib object which keeps track of client connections to the Telemetry Server (by connecting to its signals). When the conditions for a particular delay line being available have been fulfilled, it instantiates a delay line system object (Sec. 5.1), and announces its existence by emitting a signal. The manager continues to monitor connections to the server, and destroys the system object (emitting another signal) when the conditions cease to be fulfilled.

The conditions for a delay line being “available” are currently:

- Relevant trolley micro connected

If other sub-systems (such as the VME system) are disconnected, the system state machines only provide access to an appropriate subset of the delay line system functionality. Hence the delay line system can be operated in a test rig with no metrology system, using direct slew mode and with the shear loop deactivated.

The connection manager also manages resources and data that should be shared between multiple delay lines, such as the common offset and the socket connection to the VME system.

5 System-level Application Programming Interface

In the OpdGui application, delay line system objects are created by the Connection Manager once the relevant sub-systems have connected to the Telemetry Server. Each such

object (an instance of the `DGSystem` class) maintains high-level state information for a single delay line.

Control of delay line functions (such as following a particular trajectory or activating the shear loop) is accomplished by calling methods of the system object.

5.1 `DGSystem`: Delay Line System Class

The `DGSystem` class implements an application programming interface for control of a single delay line. The methods provided include (the following is not an exhaustive list):

- Specify OPD mode (control of metrology loop):
 - Stop/Idle
 - Datum seek
 - Follow specified trajectory:
 - * Sidereal trajectory
 - * Fixed position trajectory
 - * Constant velocity trajectory
 - * Constant acceleration trajectory
 - Direct Slew (slew command direct to trolley micro)
- Return current position/velocity
- Activate/deactivate steering loop
- Activate/deactivate shear loop
- Adjust focus

The above list includes methods to select one of three mutually-exclusive “system OPD modes”:

Stop/Idle Mode In this mode the trolley is stopped by the workstation sending a `DirectSlew` command with a demand velocity of zero to the on-board trolley micro. An `Idle` command is sent to the VME CPU, which is not required to do anything in this mode.

Datum Mode In this mode a `Datum` command is sent to the VME sub-system, instructing it to transmit a sequence of slew velocities to the trolley micro in order to seek the datum switch, and then reset the metrology count.

Follow Mode In follow mode, the workstation transmits a trajectory demand to the VME sub-system, which controls the cat's-eye and carriage in order to follow the demand, ideally in track mode but automatically switching into slew mode to reposition the trolley as necessary.

Note that although most of the functions provided by the system API require coordination of several delay line sub-systems, several only involve commanding the trolley micro (direct slew, steering on/off and focus control).

5.2 State Machine

The system class relies on a set of state machines to translate API function calls into commands to delay line sub-systems, and to provide feedback on the success or failure of requests made via the API.

Each control axis of each trolley has its own independent state machine, each with several possible states:

OPD state machine Controls axial motion of the carriage and cat's-eye

Steering state machine Controls the state of the steering servo

Tip-tilt state machine Controls the state of the tip-tilt servo

Focus state machine Controls the state of the cat's-eye focus servo

Each state of the above state machines is implemented as a GObject class. Communication between the system API and the state machines is accomplished by means of GLib signals, emitted by DGSystem objects. Each state machine class only provides handlers for signals that it makes sense to respond to, for example the "Steering On" state only has a handler for the "steeringoff" signal. Hence responses to user requests delivered via the system API will depend on the current system state.

State changes occur in response to changes in sub-system status (obtained from the Telemetry Server), which in many cases must be preceded by receipt of an appropriate command acknowledgement. State changes are also triggered by sub-systems disconnecting from the workstation.

A parent state class provides a framework for state transitions, including:

- Virtual signal handlers, connected at construction.
- A mechanism for conditional state transitions, triggered by sub-system status changes.
- Convenience methods for:

- Sending commands
- Querying sub-system status
- Unconditional state transitions

Information on the current state of the delay line is made available through the DGSystem API, and is also transmitted to the Telemetry Server over the loopback interface using the dlmsg status message protocol. These system status items are retrieved by DGSystemGui using the API and displayed in the relevant application system control page (Figure 2).

The system status also includes log and fault messages generated by the system object. These are displayed and logged by DGOpdGui in the same way as log/fault messages sent by delay line sub-systems.

6 Trajectory Calculator

Delay line system objects (Sec. 5.1) incorporate a trajectory calculator object, to transmit a trajectory demand to the VME system.

Each instance of the Trajectory “class” generates the demand for a specific trolley (specified at construction). A soft real-time periodic task (implemented using the GLib event loop) is used to send command data messages (and a copy of the data as telemetry to the Telemetry Server) containing the demanded position and velocity as a function of time, for times slightly later than the transmission time. The periodic task is rescheduled as necessary so that the lead time stays between configurable lower and upper soft limits (the current values are 990 and 1010 ms). If the lead time drops below a configurable hard limit (currently 100 ms), the trajectory sequence is restarted (giving rise to a discontinuity in the trajectory).

The workstation software now uses a dedicated second thread for the trajectory calculator, implemented using the GLib interface to the POSIX thread libraries. This thread runs a second instance of the GLib main event loop. The only event source associated with this main loop is the periodic task described in the previous paragraph, all other events being handled by the primary thread’s main loop. The dual-threaded implementation was adopted to prevent occasional delays to the periodic task caused by display updates, such as redrawing the entire application window.

The following types of trajectory are supported; switching between different trajectories is accomplished with a method call.

- Sidereal trajectory
- Constant velocity trajectory (fixed position is a special case of this)
- Constant acceleration trajectory

- Slew trajectory
- Undefined trajectory

If an undefined trajectory is specified, the periodic task continues to run and to transmit telemetry (containing dummy values), but command data messages are not transmitted.

Constant velocity and constant acceleration trajectories may either be specified to pass through a specific point (position and time), or to match up with the last demand transmitted for the previous trajectory. The latter behaviour avoids discontinuities when changing trajectory, so that the trolley responds smoothly.

The trajectory object also transmits status messages at 10Hz containing the *current* position (`PosDemNowN`) and velocity (`VelDemNowN`) demand. If the trajectory type or parameters have changed recently this will correspond to the previous trajectory. A boolean status item, `FollowCurrentN`, indicates whether the latest trajectory has come into force.