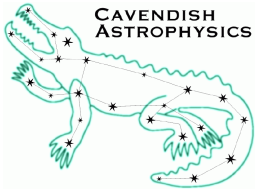
MRO Delay Line

Workstation Software Functional Description

The Cambridge Delay Line Team

rev 0.1
5 July 2007



Cavendish Laboratory
Madingley Road
Cambridge CB3 0HE
UK

Change Record

Revision	Date	Authors	Changes
0.1	2007-07-05	JSY	Initial version

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 subsystems, 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

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.

1.2 Workstation Application Architecture

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

Two alternative application programs 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.

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.

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").

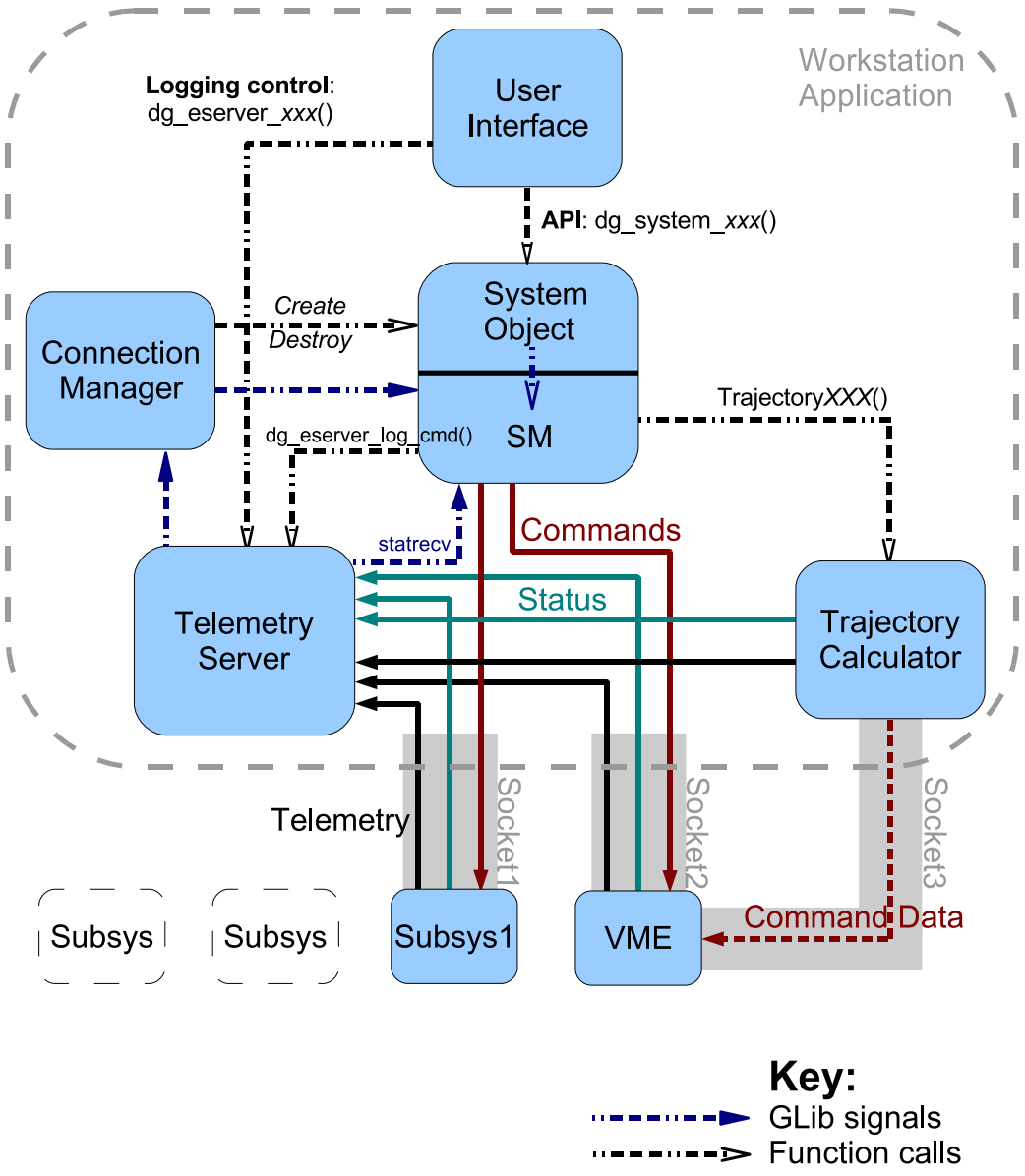


Figure 1: Diagram of workstation application architecture, showing the component objects and the communication links between them.

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

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 DGUi: Parent User Interface Class

DGUi provides virtual signal handler methods (so child classes may reimplement signal handlers as appropriate without having to worry about signal connection) and detection of exceptions (user or programmer errors – see "An Exception Handling System for C Software"). DGOpdGui inherits from this class (we plan to modify DGTermUi so that it also inherits from this base class).

2.2 DGOpdGui: 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 logging of telemetry, status and commands.

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 as a notebook page (and will be destroyed when the Connection Manager signals that the delay line is down). This system control

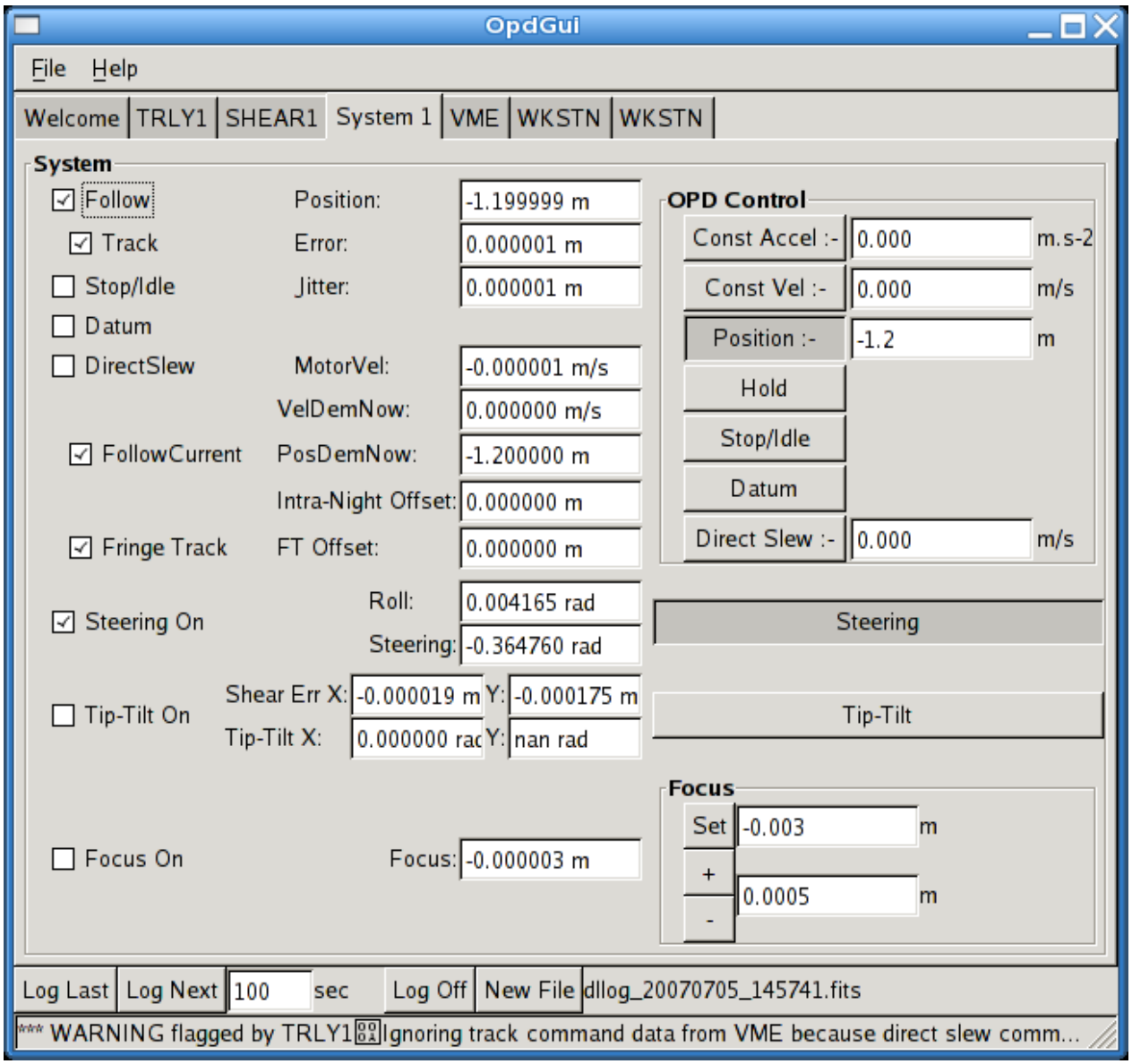


Figure 2: Screenshot of the system-level GUI application OpdGui, showing the system control interface.

interface incorporates logical groupings of action buttons and widgets for displaying system status items (see Sec 5.2).

DGOpdGui 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 (see Sec. 5.2), rather than by the GUI.

The GTK+ widget layouts use by DGOpdGui are stored in XML files generated by the Glade interface design software, and realised using libglade. DGOpdGui inherits from the DGUi parent user interface class.

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

2.4 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 idle function which polls the Ncurses keyboard handler.

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,

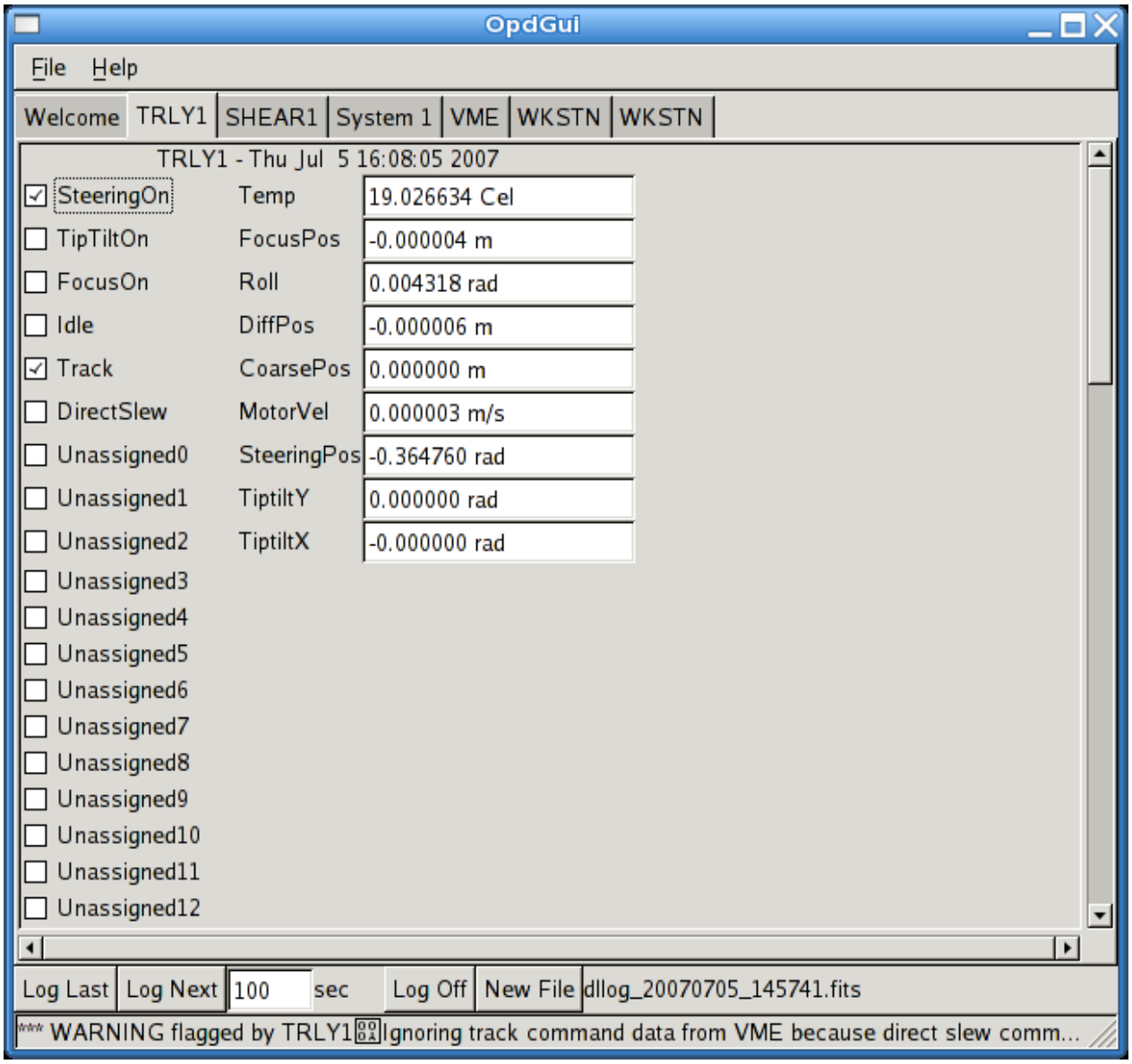


Figure 3: Screenshot of the system-level GUI application OpdGui, showing a sub-system status display.

and will throw an exception if this is violated. 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:

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

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

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
- 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)
- Destroy the server, which breaks the connections to any connected clients

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.

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
- VME system connected

The latter condition could be removed provided more intelligence is added to the system state machines (so that they handle user requests differently depending upon whether the VME is connected).

The connection manager also stores data that should be shared between multiple delay lines, such as the common offset.

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:
  - Idle (Stop)
  - Datum seek
  - Follow specified trajectory:
    - Sidereal trajectory
    - Fixed position trajectory
    - Constant velocity trajectory
    - Constant acceleration trajectory
    - Slew trajectory
  - Direct Slew (slew command direct to trolley micro)
- Return current position/velocity
- Activate/deactivate steering loop

- Activate/deactivate shear loop
- Adjust focus

Note that although most of these functions 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 preceeded 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. These system status items are retrieved by DGOpdGui using the API and displayed in the application system control page (Figure 2).

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 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 (`PosDemNow`N) and velocity (`VelDemNow`N) demand. If the trajectory type or parameters

have changed recently this will correspond to the previous trajectory. A boolean status item, `FollowCurrent`N,indicates whether the latest trajectory has come into force.