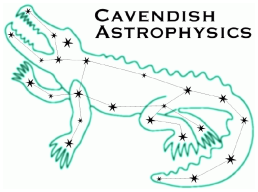
MRO Delay Line

Production Workstation Software Functional Description

INT-406-VEN-1003

John Young <jsy1001@cam.ac.uk>

rev 1.0
22 January 2010



Cavendish Laboratory
JJ Thomson Avenue
Cambridge CB3 0HE
UK

Change Record

Revision	Date	Authors	Changes
0.1	2010-01-18	JSY	Initial version
1.0	2010-01-22	JSY	Rewrote system class section, minor edits to other sections

Objective

To describe the design and implementation of the production workstation software.

Scope of this document

This document forms part of the design documentation for the delay line production software contract. It describes the design and implementation of the workstation software.

The overall architecture of the control software system is described in RD1. The workstation software acts as a supervisor for other delay line subsystems, the software for which is described in separate documents [RD2,RD3,RD4].

Reference Documents

RD1	Production Delay Line Control Software Architecture	INT-406-VEN-1001
RD2	Production Trolley Software Functional Description	INT-406-VEN-1002
RD3	Production Metrology Software Functional Description	INT-406-VEN-1004
RD4	Production Shear Sensor Software Functional Description	INT-406-VEN-1005
RD5	Delay Line Analysis GUI Functional Description	INT-406-VEN-1006
RD6	Requirement Specifications for the MROI "production" delay line software	INT-406-CON-0101
RD7	Design of an MROI System	INT-409-ENG-0020
RD8	Network Message Protocols and Telemetry/Status/Logs File Format	INT-406-VEN-1007
RD9	dlmsg Library Manual	
RD10	An exception-handling system for C software	

Applicable Documents

AD1	(Prototype) Workstation Software Functional Description	INT-406-VEN-0103

Acronyms and Abbreviations

API Application Programming Interface

DL Delay lines

dlmsg Cavendish delay line messaging protocol

ISS Interferometer Supervisory System

MCDB ISS Monitor and Configuration DataBase

MROI Magdalena Ridge Observatory Interferometer

OPD Optical Path Difference

Contents

1 Introduction

The workstation software fulfills several distinct functions:

- Provide supervisory control of the delay lines by generating trajectory demands for the metrology software and managing the state of the delay line subsystems.
- Provide an interface to the MRO Interferometer Supervisory System (ISS) for monitoring and control of the delay lines as integrated systems
- Provide an engineering user interface for monitoring and (when operating without the ISS) control of each delay line, including:
  - System-level controls
  - Selected subsystem-level controls
  - Live display of status information
  - Recording of status and telemetry (monitor data) and logging of commands and log/fault messages

A separate software package is used for off-line processing and analysis of recorded telemetry. This analysis software is described in RD5.

Several new features are required to meet the requirements [RD6], compared with the functionality of the prototype software [AD1]:

System command interface to ISS The workstation system controller software shall expose a commandable system object for each delay line which conforms to the standard characteristics of an MROI monitoring system [RD7].

Monitor data transmission to ISS The system objects implemented by the workstation software shall transmit monitor data (this encompasses several of the classes of data defined for the prototype DL software, namely telemetry and status for both the delay line system and its subsystems) for the delay lines to ISS Data Collector(s). The data available for transmission shall include video from the DL shear cameras.

On-the-fly reconfiguration of monitor data Command(s) shall be provided to deactive individual monitor points. To provide a further capability to reduce the network bandwidth requirements, the workstation shall provide a facility to decimate monitor data by only transmitting every Nth sample to the ISS (with a goal to provide a command to change the decimation factor for each monitor point individually).

Fault notifications The workstation system controller software shall transmit fault notifications to the ISS.

Alert notifications If a serious condition occurs that requires the immediate action of the telescope operator, the software shall transmit an alert notification to the ISS

Separate Engineering Control GUI It shall be possible to run the delay line system controllers without displaying the engineering control GUI. It shall be possible to start and stop the engineering control GUI (typically on a remote display) while the system controllers are running.

1.1 Application Programs

The functionality outlined above is provided by two separate event-driven application programs:

System Controller Performs supervisory control of multiple delay line systems, generates trajectory demands, and provides monitoring and control interfaces to the ISS and engineering control GUI.

Engineering Control GUI User interface, for use in standalone mode (i.e. when operating the delay lines independently of the ISS), or to record data for use with the Analysis GUI [RD5]

The architecture differs from that of the prototype software [AD1] in that the prototype "OpdGui" application has been split into the two applications outlined above. The system controller and engineering control GUI communicate using the same network messaging protocols used for inter-subsystem communication [RD8], and may run on the same computer (using socket connections over the loopback interface) or on separate computers. The system controller acts as both a server (accepting connections from the delay line subsystems as described in RD8, and from the ISS as described in RD7) and as a client (connecting to the engineering GUI in order to receive system commands and transmit status and telemetry for display and recording by the GUI application).

The test controller application described in AD1 will be delivered as part of the production software, but is intended as a development aid and is not needed to fulfill any of the requirements specified in RD1. Hence the test controller application will not be described further in this document.

The system controller is described below in Sec. 3, and the engineering control GUI in Sec. 4.

1.2 Development and Execution Environment

The workstation computer used for delay line tests in Cambridge 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.

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 Sec. 2) and a signalling (callback) system.

The engineering control graphical user interface will be implemented using the GTK+ graphical user interface toolkit (most likely in C++ using the gtkmm classes that wrap GTK+, together with a C++ wrapper for the existing telemetry server described in Sec. 2.1).

The extent to which the system controller application will use C++ has not been decided yet. The system controller must integrate with the C system framework being written by NMT, which can be linked with either C or C++ code. The gtkmm project provides a C++ wrapper library for GLib as well as for GTK+. This wraps most of the GLib functionality with the exception of the GLib C Object System (GObject) and its associated signalling functionality, which is used extensively in the prototype workstation software. A C++ signalling library, libsigc++, that provides the equivalent functionality is included in gtkmm, but these signals cannot straightforwardly be mixed with GObject signals. Hence the system controller can be coded mostly in C or mostly in C++, the middle ground being impractical.

2 General Application Architecture

The system controller and engineering control GUI applications have the same basic event-driven architecture, based around an embedded telemetry server (Sec. 2.1). A C library [RD9] provides network message encoding and decoding for communications between the two applications and with the delay line subsystems.

The event-driven framework is provided by GLib (perhaps via its C++ wrapper): when the application is started a number of modules are initialised, 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.

There are two exceptions to this (where accurate scheduling is needed or lengthy tasks must be performed):

- the system controller's trajectory calculator objects (Sec. 3.2) each run their 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 metrology subsystem;
- cleaning up after the user aborts a logging operation is performed by the telemetry

server in a separate thread, as this would otherwise block the handling of other events for several seconds in the worst case.

Both applications read and parse 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. In the case of the system controller, the configuration file is written by the ISS using data from the Monitor and Configuration Database (MCDB) as part of the system startup process [RD1].

Possible event sources are:

1. Input/output watches: events triggered by activity on an open file, pipe or socket
2. Timeouts: periodic events
3. Idle functions, which run when no higher-priority event is pending

A handler for one of these event types may propagate the event to other components of the software using the signalling system (either GObject or libsigc++). Other components may register any number of callback functions (signal handlers) for a particular signal without the signal emitting code having any knowledge of the possible callbacks. The use of a signalling system thus facilitates a modular event-driven architecture for the software.

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 RD10. 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.

An equivalent system for propagating exceptions is used in the C system framework. The workstation code will convert between the two different C exception representations and/or C++ exceptions as appropriate.

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 signal provided by the telemetry server (Sec. 2.1) which invokes a handler that reports the exception to the user and/or the ISS.

2.1 Telemetry Server

The telemetry server manages TCP/IP socket communications with multiple clients, using the "dlmsg" protocols described in RD8. When embedded in the system controller

application, the clients are the delay line subsystems (trolleys, shear sensors etc.). When embedded in the engineering control GUI, the clients are the delay line system objects belonging to the system controller application. In the latter case, the system objects transmit their own status messages as well as forwarding status and telemetry messages received from the relevant delay line subsystems: data originating from each system/subsystem is treated as a separate client even though a common socket is used.

The content of all these messages is defined in RD8.

The telemetry server provides event handlers for clients connecting to and disconnecting from the server and the arrival of status and telemetry messages. The server also buffers and logs the content of status and telemetry messages — this functionality is only used in the engineering GUI.

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. When logging (see Sec. 2.2) 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 RD8.

The telemetry server allows 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 of status and telemetry to the current FITS file
- Close the current FITS file and open another
- Log a command to the current FITS file
- Write a log/fault message 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

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.

2.2 Recording of Telemetry/Status/Logs

The recording functionality provided by the telemetry server is used by the engineering control GUI application (Sec. 4). Two flavours of recording 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.

3 System Controller

The system controller application performs the following functions:

- Provides supervisory control of multiple delay lines by generating trajectory demands for the metrology software and managing the state of the delay line subsystems.
- Provides interfaces to the MRO Interferometer Supervisory System (ISS) or the engineering control GUI for high-level control of the delay lines
- Receives status and telemetry messages from delay line subsystems and forwards their content to the ISS (as monitor data), and/or to the engineering control GUI

The system controller acts as server, with separate TCP/IP server sockets for the dlmsg protocols (accepting connections from the delay line subsystems), and for the ISS protocols (accepting connections from the various components of the ISS).

The system controller also attempts to connect to the engineering control GUI (Sec. 4) at a configurable interval (2 seconds by default), at an address and port specified in the system controller's configuration file. A separate connection is used for each delay line system, initiated by the system controller so that the existing telemetry server can be used unchanged in the engineering GUI. Once a connection is established the resulting socket is used to receive system commands (when operating in standalone mode) and transmit status and telemetry for display and recording, using the dlmsg protocols.

The application architecture is shown diagrammatically in Figure 1. The embedded telemetry server (Sec. 2.1) handles TCP/IP socket communications with the delay line subsystems. The ISS Executive starts the system controller as part of the delay line bootstrap process (see RD1) and uses command line arguments to specify which delay lines the system controller should manage. The corresponding system objects (Sec. 3.1) are then created when the application is started (this is a change from the protoype workstation application, which created system objects in response to trolley subsystems connecting).

3.1 System Class

Instances of the system class perform the following functions:

- Supervisory control of a single delay line, implementing the standard MROI state model and managing the state of the delay line subsystems.
- Receive and execute system commands from the ISS or from the engineering control GUI, issuing subsystem commands as appropriate
- Receive status and telemetry messages from delay line subsystems and forward their content to the ISS (as monitor data) and/or to the engineering control GUI

An associated trajectory calculator object (Sec. 3.2) generates trajectory demands and transmits them to the metrology subsystem.

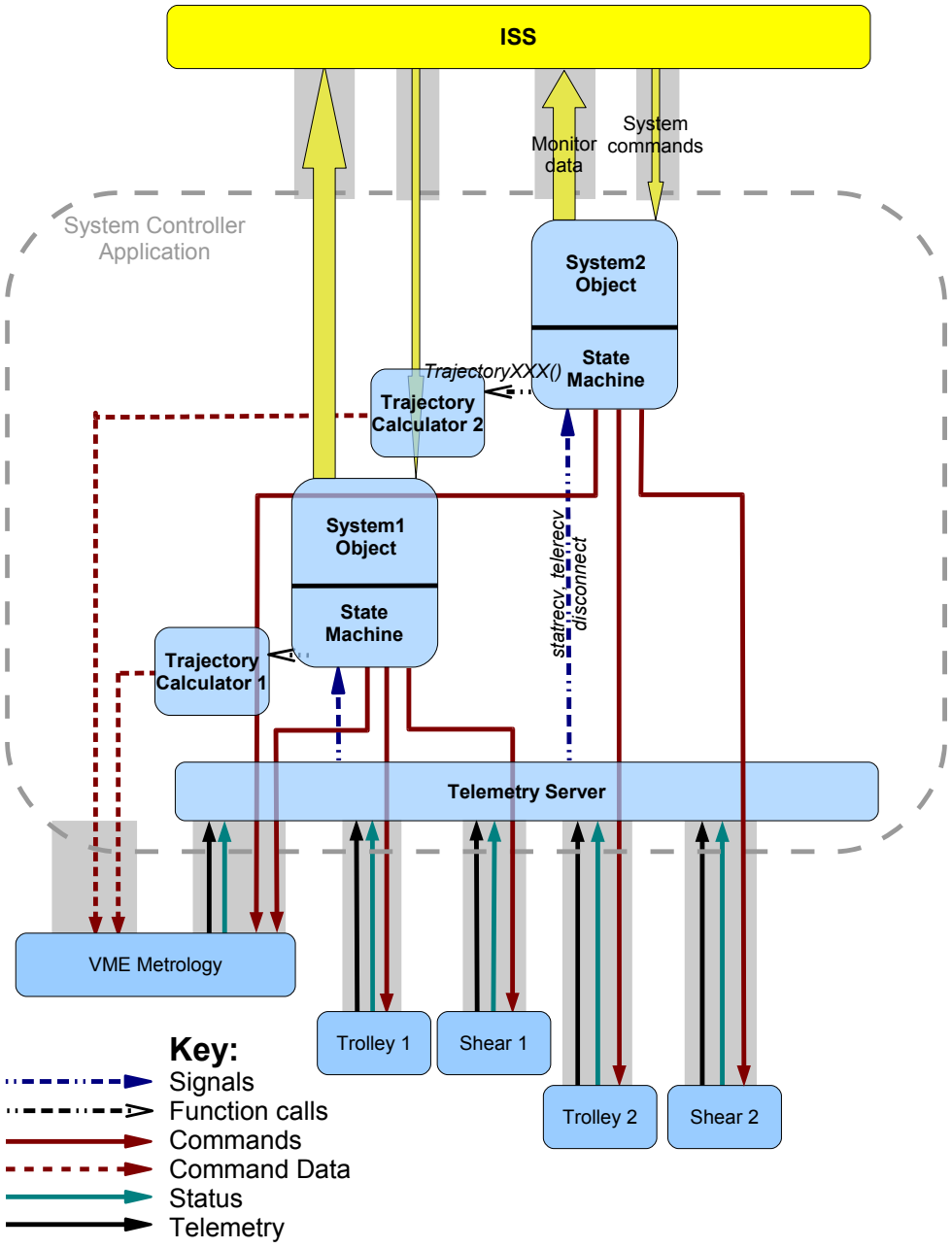


Figure 1: Diagram of workstation system controller architecture, showing the application's component objects and the communication links between them, as well as TCP/IP socket connections (shown in grey) to other computers. The solid arrows denote network messages, and the dashed arrows intra-application information flows. For clarity, the interfaces to the engineering control GUI are not shown.

3.1.1 Use of MROI C system framework

From the perspective of the ISS, the delay lines will appear as standard MROI systems. This will be accomplished using a C code-generation framework being written by NMT. The framework implements the custom TCP/IP messaging protocols for monitoring systems described in RD7. The facilities provided by the C framework will be integrated into the (mostly single-threaded) event-handling system described in Sec. 2 above.

The NMT-designed framework is based on a set of system methods which are common to all C systems and are fully implemented in a generic manner. These methods perform message decoding and encoding for all the commands and responses defined by the ISS TCP/IP-based communications protocol [RD7]. The framework supplements these generic methods with system-specific methods generated using Xpand from XML worksheets that define the commands and monitor points for the delay line systems.

The framework can also create threads to poll the sockets and call the appropriate system methods, but we have chosen not to use this functionality. Instead we have opted to retain the event-driven architecture based on the GLib Main Event Loop that was developed for the prototype workstation software, defining additional events to manage socket communications with the ISS. The handlers for these new events will call framework methods for message decoding and encoding, and hence do not themselves need to know the details of the ISS messaging protocol.

When the application starts it opens server (listening) sockets on the main and data TCP/IP ports, sets up watch events for the server sockets, then runs the main event loop. The server is now able to accept connections from remote ISS clients.

The main event loop regularly polls any sockets for which watch events have been set up. Hence when it detects a connection attempt on a server socket the appropriate event handler is called. These event handlers call the generic `acceptMROIServerSocket()` function to open a socket connection to the client. A watch event is then set up in the main loop for the new socket.

Incoming data on a socket connected to a client thus triggers a custom handler which calls the generated `executeDelayLineSystemRequest()` method. In turn that method decodes the request and calls the appropriate generic or hand-coded function. This function may be one that implements a synchronous or asynchronous system-specific command or one that performs the system-specific actions for a transition between standard system states.

3.1.2 Standard state transitions

There are three state transitions which may involve system-specific actions:

- Start → Initialize
- Operational → Shutdown

- Any state → AboutToAbort

Because initialising the system involves opening socket connections between the delay line subsystems and the system controller, the initialize server command must be asynchronous. The current ISS protocols allow for this, but a small extension of the current C framework is also required. When the INITIALIZE_SYSTEM_ASYNC command (see RD7) is received `executeDelayLineSystemRequest()` calls a hand-coded function which:

1. Starts the dlmsg telemetry server (Sec. 2.1), opening a server socket to listen for subsystem connections and registering an event handler for connection events on that socket
2. Registers a callback for the signal emitted by the telemetry server when a client (DL subsystem) connects

The function then returns, thus the event loop resumes. When a subsystem connects, the connection callback is invoked. If the metrology system has connected, the handler opens a command data connection (shared amongst all the delay line systems, see Figure 1) to the metrology system for transmission of trajectory data. The handler checks whether all the subsystems needed to operate the delay line have connected. Once this has happened, the monitor points defined for the system are validated (see Sec. 3.1.4), and finally the callback function for the asynchronous command is called (which sends a message to the ISS to indicate that the command has completed).

The `shutdownAction()` method performs the shutdown actions specific to a delay line system:

- Terminate any asynchronous commands in progress and put DL subsystems into idle states
- Break connections to ISS clients (except the one issuing the shutdown command)
- Break connections to DL subsystems that are no longer required, deactivate monitoring
- Break connection to engineering GUI (if connected)

The `aboutToAbortAction()` method performs the pre-abort actions specific to a delay line system:

- Put DL subsystems into idle states
- Flush logfiles

3.1.3 System commands

The system class also has methods to implement commands that are unique to the delay lines. Most of the commands involve communicating with delay line subsystems in order to execute the command and determine when it has completed — these will be implemented as asynchronous commands. A preliminary list of commands is given in RD1. The most important of these are:

- Specify OPD mode (control of metrology loop) [all asynchronous]:
  - 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 [synchronous]
- Activate/deactivate steering loop [asynchronous]
- Activate/deactivate shear loop [asynchronous]
- Adjust focus [asychronous]

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 metrology subsystem, which is not required to do anything in this mode.

Datum Mode In this mode a `Datum` command is sent to the metrology subsystem, 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 metrology subsystem, 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.

The system class relies on a set of state machines to translate asynchronous system commands into commands to delay line subsystems. Each control axis of each trolley has its own independent state machine, each with several possible states (note that these are all sub-states of the OPERATIONAL standard system state):

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

State information from these state machines is made available through methods of the system object, and is also transmitted to the embedded telemetry server over the loopback interface using the dlmsg status message protocol. The system status also includes log and fault messages generated by the system object.

3.1.4 Monitoring

The delay line system class schedules monitor data transmission using the GLib event loop, rather than using threads created by the framework. This allows the transmission events to be synchronized with the availability of new data from the delay line subsystems. The custom event handlers call functions provided by the framework to determine what data has been requested and to transmit the data.

The workstation code treats all data arriving from relevant subsystems or generated internally by the system object as potential monitor data. Any of these potential monitor points may be designated as external DL monitor points in the appropriate worksheets. The worksheet definitions must have matching data types, intervals and units. Any mismatch will be detected at run-time by validating the worksheet definitions against the content of incoming telemetry and status messages when the system is initialised.

To permit the network bandwidth needed for transmission to the ISS to be reduced, the NMT-written framework allows for decimation of individual monitor points, i.e. transmitting every Nth sample only. Decimation is performed by the workstation, except for shear camera video which is decimated by the shear sensor software prior to transmission. It is also possible to disable the transmission of individual monitor points on-the-fly, which is accomplished by setting their decimation factors to zero.

We also allow for transmission of regularly-sampled high-rate data in chunks, where each monitor data message contains an array of samples plus a timestamp for the first sample and a sampling interval. Note that chunking is already implemented in the dlmsg telemetry protocol used to send data from delay line subsystems to the workstation. The

server chooses the outgoing chunk size to be the size of an incoming chunk after decimation. Transmission of chunked data will require an extension of the current monitor data message format. This extension only needs to handle scalar datatypes, as for larger datatypes such as images the size of a timestamp is negligible compared with the size of a data sample.

3.2 Trajectory Calculator

Delay line system objects incorporate a trajectory calculator object, to transmit a trajectory demand to the metrology subsystem.

Each object generates the demand for a specific trolley (specified at construction). A soft real-time periodic task 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 transmission times for different delay lines are staggered to avoid scheduling conflicts.

The workstation software uses a dedicated second thread for the trajectory calculator. This thread runs its own 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 multi-threaded implementation provides more precise event scheduling, which may be needed when handling multiple delay lines.

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
- 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 (i.e. the one being applied by the metrology subsystem). A boolean status item, `FollowCurrent`N, indicates whether the latest trajectory has come into force.

4 Engineering Control GUI

The engineering control GUI application program provides a user interface that is almost identical to that of the prototype software "OpdGui" application [AD1], but which can be started and stopped independently of the delay line system controller.

The GUI has a graphical user interface implemented using GTK+. The interface provides user logging controls (which activate the logging functions provided by the telemetry server — see Sec. 2.2) and a real-time display of the status items received from each connected subsystem.

The engineering GUI 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 when a system object belonging to the system controller connects to the engineering GUI's embedded telemetry server.

When the delay lines are operating under the control of the ISS (this could be indicated by a new boolean status item), the engineering GUI controls that affect the state of the delay lines will be disabled, hence the engineering GUI will provide display and logging functions only.

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

4.1 EngGui: Main engineering control GUI class

This class manages 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.

When a delay line system has initialised and connected to the engineering GUI, a user interface for controlling that delay line is created by instantiating a SystemGui object, together with a corresponding RemoteSystem object to act as a proxy for the delay line being commanded. The SystemGui's widgets are displayed on a new notebook page.

The EngGui class also implements a graphical display of all status items, grouped by subsystem. The status items from each subsystem are displayed on a separate page of the notebook (see Figure 3).

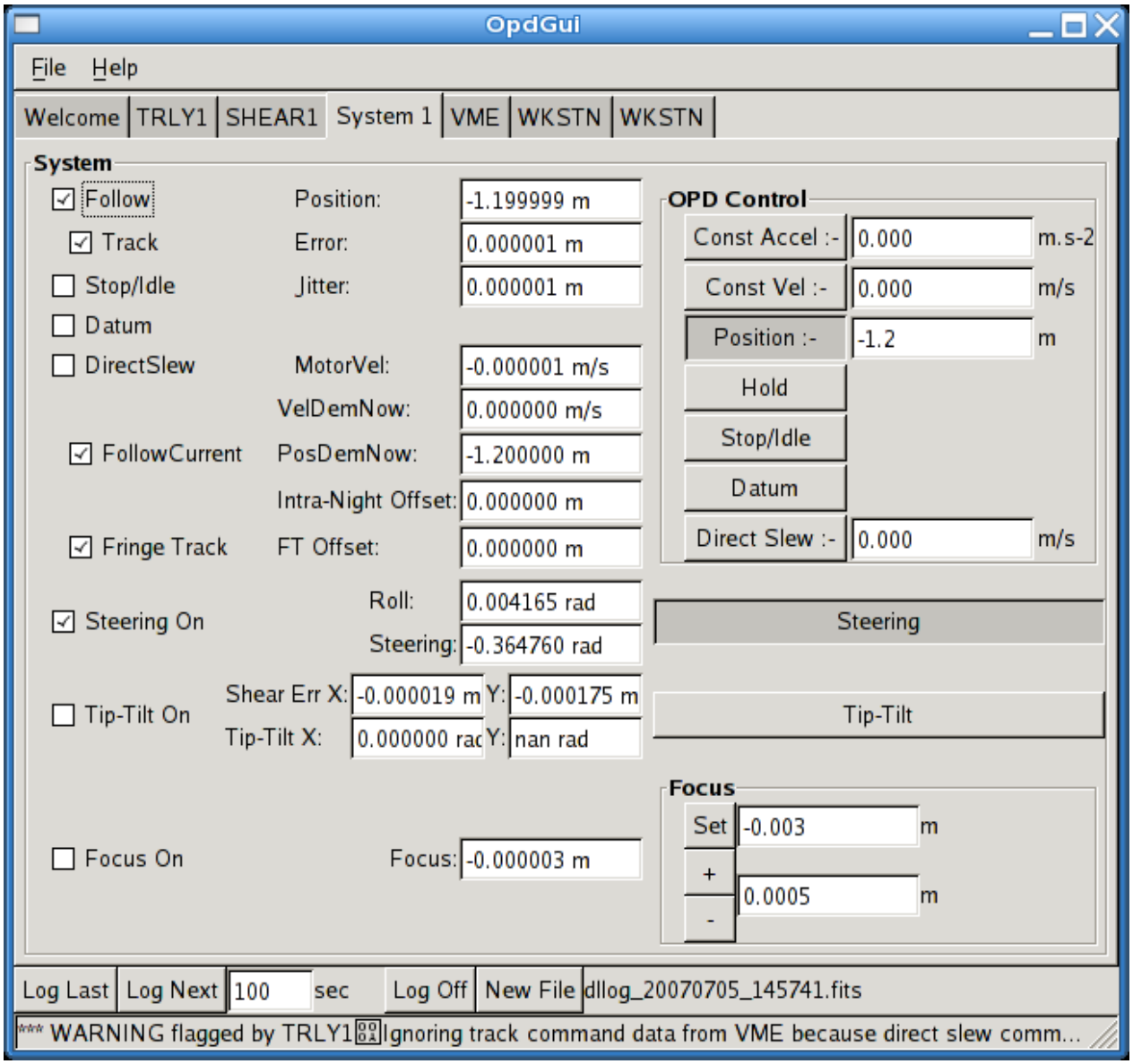


Figure 2: Screenshot of the prototype GUI application OpdGui, showing the system control interface. The production engineering control GUI will provide an identical user interface for controlling the delay lines in standalone mode.

4.2 LogGui: Logging user interface class

This class provides GUI controls to start and stop logging to the current FITS file, and to open a new FITS file. The logging control widgets are packed into a container widget supplied to the constructor.

The class defines signals that are emitted when a recording is started or stopped. These are used by SystemGui in order to synchronise recording of shear camera video with status and telemetry logging.

4.3 SystemGui: 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.

The class defines handlers for button click signals which call RemoteSystem methods.

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

SystemGui 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, SystemGui responds to a signal from LogGui that the user has initiated logging by commanding the relevant shear sensor to start recording.

4.4 RemoteSystem: System proxy class

This class provides methods for sending commands to the remote system controller over a TCP/IP socket connection, hiding calls to the dlmsg library.

4.5 ClientDisplay: single-client status display class

This module is used by EngGui and SystemGui to optionally create and to update the widgets used to display status from a single client of the embeddded telemetry server (i.e. a system object or the status originating from a particular 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.

In the former case the widget layout is generated automatically when clients connect, so that the displayed widgets are always consistent with the status items contained in the

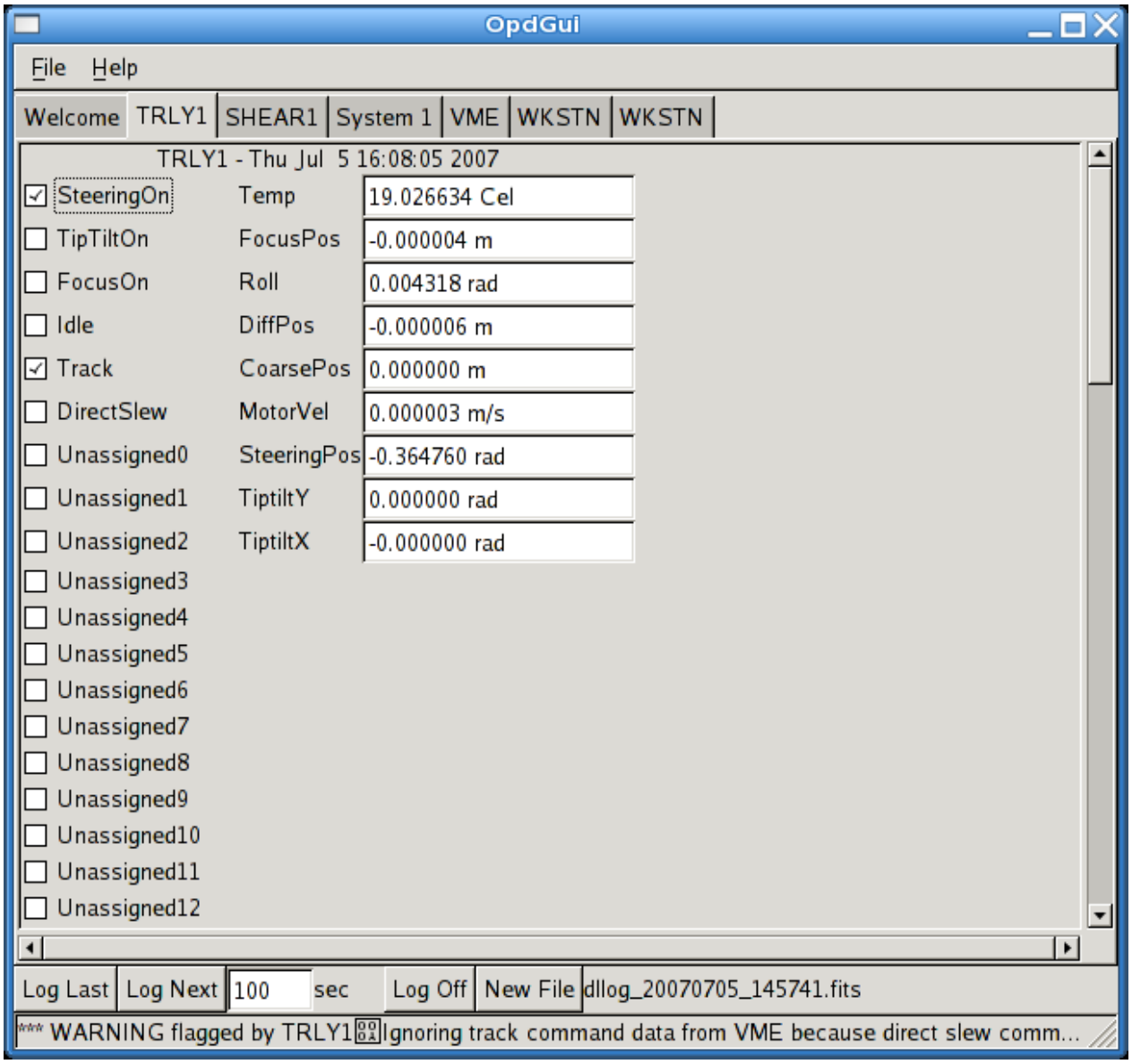


Figure 3: Screenshot of the prototype GUI application OpdGui, showing a subsystem status display generated by ClientDisplay. Identical functionality will be provided by the production engineering control GUI.

network messages. Thus clients can be modified to add new status items, and these status items will be displayed without any changes to the GUI software.

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